Responsive Multithreaded Processorの命令実行機構

伊藤 務[†] 山崎信行[†]

本論文は，分散リアルタイム処理用プロセッサである *Responsive MultiThreaded*（*RMT*）*Processor* の命令実行機構について述べる．*RMT Processor* で採用している *RMT* アーキテクチャは，細粒度マルチスレッディングに優先度を取り入れ，ハードウェアレベルで優先度を扱うことにより，より細かい時間粒度でリアルタイム処理の制御を可能にしている．一方，ソフトリアルタイム処理では動画等のマルチメディア処理に高い演算性能が要求される．このような要求を満たすために *RMT Processor* の命令実行機構にベクトル演算機構を設計する．*RMT* アーキテクチャでは，複数スレッドが並列に実行されるため，これらのスレッドが同時にベクトル演算を行う場合がある．そこでベクトル演算に必要な大きさだけベクトルレジスタを確保することによりベクトルレジスタを効率良く共有する．*RMT* アーキテクチャでは，優先度の低いスレッドは命令の発行率が低下するため，結果としてベクトル演算器の使用率が低下する．そこで，レイテンシの長い複合演算命令を実行することにより，ベクトル演算器の使用効率を改善する．ベクトル演算を用いた場合，ベクトル演算を用いない場合に比べて大幅に演算性能を向上した．また，複数スレッドが並列に実行されていて命令発行率が低下した場合でも，複合演算機構を用いることにより 1 スレッド単体で実行したときと同等の実行時間を実現した．

The Instruction Execution Mechanism for Responsive Multithreaded Processor

Tsutomu Itou[†] and Nobuyuki Yamasaki[†]

This paper describes the instruction execution mechanism for *Responsive MultiThreaded*（*RMT*）*Processor* for distributed real-time processing. The *RMT* architecture, adopted by *RMT Processor*, is able to control real-time processing with the finer grain by using fine-grained multithreading and dealing with priority by hardware. Here, current real-time applications require the high computing performance for soft real-time processing including multimedia processing. In order to achieve this computing performance at the instruction execution mechanism of *RMT Processor*, we design a flexible vector operation mechanism. Since multiple threads are performed in parallel in the *RMT* architecture, these threads perform vector operation in parallel. By reserving a size required for the executing vector operation, vector registers are shared by multiple threads, efficiently. Moreover, the issue rate of instructions of the low priority threads falls. As a result, the usage rate of the vector operation units falls in the *RMT* architecture. Then, we improve the usage rate of the vector operation units by executing the compound operations. When the vector operation units are used, compared with the case where vector operation units are not used, the computing performance is improved greatly. Moreover, even when multiple threads are performed in parallel and the instruction issue rate of the low priority threads falls, by using the compound operation mechanism, the performance equivalent of performing with single threads is achieved.

1. はじめに

リアルタイムシステムでは様々な時間制約を持った処理が同時に行われる．このような場合，リアルタイムオペレーティングシステム（RT-OS）の導入が不可欠である．RT-OS では時間制約とデットラインによりそれぞれの処理に優先度を付け，優先度に従って処理の順番を制御する．RT-OS では一定時間ごとに実行するタスクを切り替えながら処理を進めていくが，その切替え間隔の時間粒度には限界がある．より細かい時間粒度でリアルタイム処理の制御を行いたい場合，ハードウェアレベルでのサポートが必要になってくる．*Responsive MultiThreaded*（*RMT*）*Processor* はこのようなリアルタイム処理の要求を実現するためのプロセッサである．プロセッサ内の処理に優先度を

† 慶應義塾大学
Keio University

用いることにより，細かい時間粒度で処理の制御を行うことが可能である．一方，ストリーミング処理等のように大量の演算を必要とするソフトリアルタイム処理のために，高い演算性能を実現している．本論文では *RMT Processor* において，演算性能を向上させるための命令実行機構について述べる．

2. 背　　景

2.1 リアルタイム処理

ある時間内に処理を完了することを保証する処理をリアルタイム処理という．リアルタイム処理はその時間制約により大きく 2 つに分かれる．

- ハードリアルタイム処理
  必ず時間内に処理が完了する必要があり，時間内に処理が完了しない場合，価値がただちに 0 になる処理
- ソフトリアルタイム処理
  時間内に処理が完了しなくても価値がただちに 0 にはならないが，その結果の品質が時間経過とともに低下する処理

制御等のハードリアルタイム処理では，時間制約が厳しいため，実行時間の予測性が重要となる．一方，ストリーミング処理等のソフトリアルタイム処理では次々と送られてくるデータを演算しなければならないため，時間制約とともに高い演算性能が必要となる．リアルタイム処理では時間制約を守るために，優先度を用いて処理の順番を制御する．

リアルタイム処理を行うタスクの多くは周期的で，ハードリアルタイム性を持つタスクは周期の時間粒度が 100 us から 10 ms 程度と短かく，CPU の処理時間も短かい．ソフトリアルタイム性を持つタスクの周期は時間粒度が 10 ms から 1 s 程度と比較的大きい．リアルタイムタスクは周期やデッドラインの条件によりスケジューリングされ，優先度が付与される．スケジューラはある一定時間間隔ごとに優先度の高い順にプロセッサ資源を与えてタスクを実行する．

2.2 Responsive Multithreaded Processor

Responsive MultiThreaded（*RMT*）*Processor* は分散リアルタイム処理をハードウェアレベルで支援するためのプロセッサであり，各種 I/O を 1 チップに集積した system on a chip である[1]．*RMT Processor* のプロセッシングユニットである *RMT Processing Unit*（*RMT PU*）で採用されている *RMT* アーキテクチャは，8 way の細粒度マルチスレッディングにリアルタイム処理で用いられる優先度を取り入れたアーキテクチャになっている．

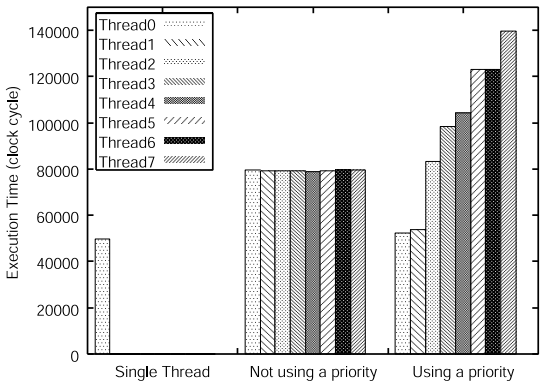

図 1 優先度を用いない場合と用いた場合の比較
Fig. 1 Comparing the case where a priority is used with the case where it is not used.

2.2.1 優先度を用いた資源の競合の調停

RT-OS ではリアルタイム処理の時間制約を守るために優先度を用いて処理の順番をスケジューリングする．*RMT* アーキテクチャではこの優先度をプロセッサ内の資源の競合の調停に用いている[2),3)]．

RMT アーキテクチャでは複数のスレッドが並列に実行されるため，スレッド間で演算器やキャッシュシステム等の計算資源の競合が起きる．競合が起こった場合，*RMT* アーキテクチャではスレッドごとに設定された優先度を基に，優先度のより高い命令に対して先に計算資源を割り当てる．これにより並列に実行しているスレッドの中で，優先度の高いスレッドから順に実行が行われる．

図 1 はソートプログラムを 8 スレッド同時実行した場合において，マルチスレッディングに優先度を用いて資源の競合を調停した場合と，優先度を用いない場合（通常のマルチスレッディング）の実行時間を示している．優先度を用いた場合 Thread0 が最も優先度が高く，段階的に優先度が低くなり，Thread7 の優先度が最も低くなっている．優先度を用いることにより，Thread0 に優先的に計算資源が割り当てられるため，Thread0 から順に処理が完了している．

このように，*RMT PU* では優先度の高いスレッドから優先的に実行され，優先度の低いスレッドの実行は待たされる．RT-OS が適切にタスクスケジューリングを行って優先度を各スレッドに付与し，ハードウェア内に保持しているスレッドをコンテキストスイッチを行うことなく優先度を用いた RMT 方式で実行した場合，ソフトウェアでコンテキストスイッチを行いながら実行する通常のリアルタイム実行と同様の効果を得る．よって *RMT PU* では，Rate Monotonic 等の静的スケジューリングによってスケジューリング（優先

度が付与）されたスレッドが 8 スレッド以内であれば，スケジューラなしでリアルタイム実行を行うことが可能である．この場合，コンテキストスイッチのオーバヘッドはなくなる．スケジューリングされたスレッドが 8 スレッドより多い場合や EDF 等の動的スケジューリングを行う場合は，ソフトウェアによるスケジューリング（スケジューラ）が必要となる．しかし，*RMT PU* のリアルタイム実行の効果と次に述べるコンテキストキャッシュを利用することにより，RT-OS はより短かい周期でスケジューリングを行うことが可能となる．さらに，細粒度マルチスレッディングによって優先度の高いスレッドが長いレイテンシの命令を実行している場合は，空いている計算資源を優先度の低いスレッドが使用できるため，優先度の高いスレッドの実行を妨げることなく全体のスループットを向上することが可能である．

2.2.2 コンテキストキャッシュ

RMT アーキテクチャではプロセッサ内に保持することのできるスレッドの数は限られているため，それより多くのスレッドを実行する場合，コンテキストスイッチが発生する．コンテキストスイッチでは，今まで実行されていたスレッドのコンテキスト（汎用レジスタやプログラムカウンタ，ステータスレジスタ等）の退避を行い，新しく実行されるスレッドのコンテキストの復帰を行わなければならないため，大きなオーバヘッドとなる．特にリアルタイムシステムでは頻繁にコンテキストスイッチが発生するため，このオーバヘッドが大きな問題となる．

RMT PU ではコンテキストを格納するための専用キャッシュをオンチップに用意し，レジスタファイルとの間をバンド幅の広い専用バス（GPR: 256 bit，FPR: 128 bit）で接続している．コンテキストの退避と復帰を高速なオンチップメモリとバンド幅の広い専用バスを用いてハードウェアで行うことにより，大幅にオーバヘッドを削減している．

2.2.3 RMT アーキテクチャの課題

RMT アーキテクチャは複数のスレッドから依存性のない命令を並列に実行してスループットを向上しているため，演算器や命令発行スロットの競合が起こり，結果として個々のスレッドのスループット自体は 1 スレッド単体で実行した場合に比べて低くなる可能性がある．よってストリーミング処理等に必要な演算性能を得られないといった問題が発生する．そこで本研究では *RMT* アーキテクチャにおいて，ソフトリアルタイム処理に要求される高い演算性能を達成するための演算機構を設計する．

3. 関連研究

細粒度マルチスレッディングは CPU 内に複数のコンテキストを保持し，これらを 1 クロックサイクルで切り替え，レイテンシの長い命令を別のスレッドを実行することにより隠蔽し，全体としてのスループットを向上している．Simultaneous MultiThreading（SMT）[4),5)] は細粒度マルチスレッディングとスーパースカラを合わせた特徴を持ち，1 クロックサイクルに複数のスレッドから複数の命令を発行する．同時に複数のスレッドから命令を発行することにより依存性の少ない命令を発行することができるため，Instruction Level Parallelism（ILP）が向上し，プロセッサ全体のスループットをさらに向上させている．リアルタイムシステムでは複数のスレッドを切り替えながら実行するため，SMT のように複数のスレッドが並列に実行できることは有用である．

文献 6) ではベクトル演算におけるメモリアクセスの性能を改善するために，マルチスレッディングを用いている．マルチスレッディングによって複数のスレッドを実行することにより，メモリポートの使用率（メモリアクセス率）を向上し，ベクトル演算の性能を向上している．

4. 設　　計

4.1 設計方針

ストリーミング等のソフトリアルタイム処理で扱われるデータは並列性が高く，大量のデータに対して繰り返し同じ演算を行うといった特徴がある．データ並列性を利用して演算性能を向上させる方法として以下のものがある．

- SIMD 演算
- ベクトル演算

SIMD（Single Instruction Stream Multiple Data Stream）演算は，1 つの命令で複数のデータを演算する．汎用プロセッサでは 1 つのレジスタを複数の領域に分割して，それぞれの領域に対して同じ演算を並列に行う．既存のレジスタファイルを利用することによりハードウェア量の増加を防ぐことができ，演算にかかるレイテンシも小さいが，演算の並列度は小さい．

ベクトル演算はベクトルレジスタを用いてベクトル要素をパイプライン的に演算する．ベクトルレジスタに対してデータの Load/Store を一度に行うため，メモリアクセスによるオーバヘッドが小さい．ベクトルレジスタのベクトル長を長くすることにより並列度を上げることができる一方，演算にかかるレイテンシが

増加し，ハードウェア量も増加する．

先に述べたように，*RMT* アーキテクチャでは優先度を用いて計算資源の調停を行っている．この時，命令キャッシュへのアクセスが優先度による調停の影響を大きく受ける．通常は優先度の高いスレッドが命令キャッシュをアクセスし，長いレイテンシの命令等で優先度の高いスレッドが命令キャッシュをアクセスしないときに，より優先度の低いスレッドが命令キャッシュをアクセスして命令を実行する．

ハードリアルタイム性を持つスレッドの時間制約を保証するために，時間制約の厳しくないソフトリアルタイム性を持つスレッドはハードリアルタイム性を持つスレッドよりも低い優先度で実行される．そのため，ハードリアルタイムのスレッドが実行されている間は，ハードリアルタイムのスレッドが命令キャッシュをアクセスしないときにのみ，ソフトリアルタイムのスレッドの命令がフェッチされる．この命令フェッチを有効に使用することにより，ソフトリアルタイム処理の性能を向上することができると考えられる．

そこで本研究では，少ない命令数で高い並列度を実現することができるベクトル演算を用いる．これにより少ない命令フェッチで高い演算性能を実現する．また，優先度の高いスレッドの実行が完了し，命令フェッチが多く行われるようになった場合でも，細粒度マルチスレッディングにより，ベクトル演算にかかるレイテンシは，別のスレッドを実行することで隠蔽される．

4.2 ベクトルレジスタの確保と解放

RMT アーキテクチャでは細粒度マルチスレッディングにより，複数のスレッドが並列に実行される．先に述べたとおり，ベクトル演算でマルチスレッディングを用いることによりメモリアクセスの効率を向上し，ベクトル演算の性能が向上する．しかしベクトル演算の並列度を大きくするとベクトルレジスタに必要なハードウェア量も増加するため，スレッドごとに大量のベクトルレジスタを持たせるとことはできない．一方，リアルタイムシステムでは複数のプログラムが並列に実行されるために，ベクトル演算を行うスレッドを 1 スレッドとし，他のプログラムを並列に実行する場合も考えられる．このようにシステムやプログラムにより，ベクトル演算に必要となるスレッド数やベクトルレジスタの構成は異なってくる．このような場合，ベクトルレジスタの構成を固定してしまうと，ベクトルレジスタを効率良く利用できない．そこで本研究ではベクトルレジスタを共有して使用し，動的にベクトルレジスタの構成を変化させることにより複数のスレッドで柔軟なベクトル演算を可能とする．

各スレッドはベクトル演算を行う場合，最初にベクトルレジスタを確保する．ベクトル演算を終了しベクトルレジスタが必要なくなった場合に確保していたベクトルレジスタを解放する．これにより次に別のスレッドがベクトルレジスタを確保しベクトル演算を行うことを可能にする．

ベクトル演算に必要なベクトルレジスタの個数，ベクトル長はアプリケーションによって異なるため，ベクトルレジスタを効率良く共有するには，適切な大きさのベクトルレジスタを割り当てる必要がある．そこで，各スレッドはベクトルレジスタを確保するときに，必要な大きさとベクトル長を一緒に指定する．指定された大きさのベクトルレジスタを確保することができればそのスレッドにベクトルレジスタの一部を割り当て，確保できなければその要求を拒否する．スレッドに割り当てた領域とベクトル長等の構成はテーブルとして保存しておく．

実際にベクトル演算を行う場合，デコードされたレジスタ ID とテーブル内に保持されている割り当て情報とベクトル長からデータが格納されているベクトルレジスタの実アドレスを計算し，ベクトルレジスタにアクセスする．

4.3 複合演算命令

ソフトリアルタイム処理の多くは積和演算等の繰り返し演算である．そこで一連の演算を複合演算としてプログラマが定義し，それを 1 命令で実行することにより，より少ない命令数でより長いレイテンシの演算が可能になる．これにより命令フェッチ率が低い場合でも，ベクトル演算器の使用率を向上させる．

5. 実　　装

5.1 RMT Processing Unit

RMT PU のブロック図を**図 2** に示す．また，*RMT PU* の概要を**表 1** に示す．

RMT PU は 8 つのスレッドを各々独立したレジスタファイル内に保持し，これらを優先度付き SMT で実行する．Thread Control Unit はスレッドの制御を行い，各スレッドの実行，停止，コンテキストキャッシュへの退避，復帰等を行う．コンテキストキャッシュは先に述べたコンテキストを格納するためのオンチップメモリであり，32 スレッド分のコンテキストを保持し，4 クロックサイクルでプロセッサ内のレジスタファイルと入替えを行う．Instruction Unit は優先度に基づいて 1 クロックに 1 スレッドから命令をフェッチする．フェッチされた命令はデコードされ，各スレッドごとの命令バッファに格納される．命令バッファから

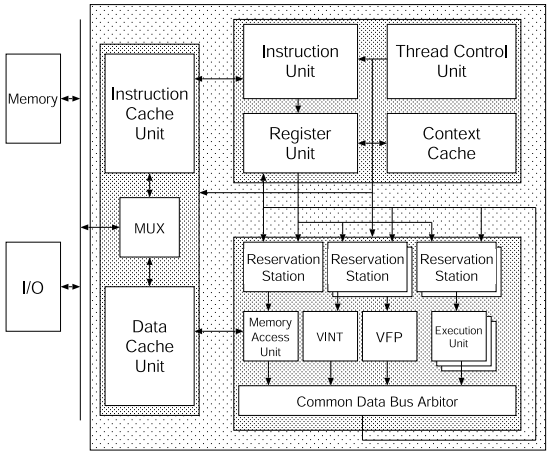

図 2 *RMT Processing Unit* のブロック図
Fig. 2 The block diagram of *RMT Processing Unit*.

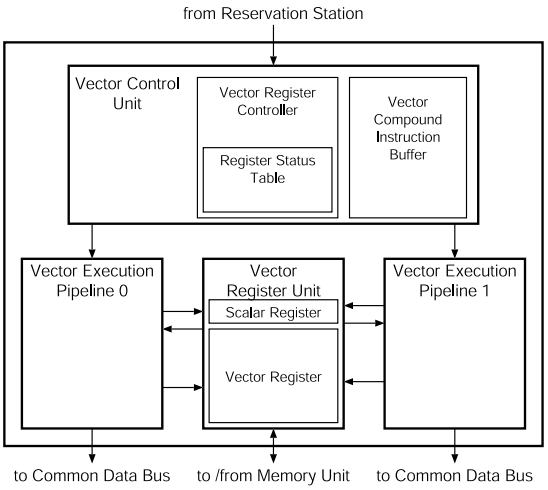

図 3 ベクトル演算器のブロック図
Fig. 3 The block diagram of a vector operation unit.

表 1 *RMT PU* の概要
Table 1 The outline of *RMT PU*.

アクティブスレッド数	8
キャッシュスレッド数	32
同時命令発行数	4
同時命令完了数	4
整数レジスタ数	32 bit × 32 entry × 8 set
整数リネームレジスタ数	32 bit × 64 entry
浮動小数点レジスタ数	64 bit × 8 entry × 8 set
浮動小数点リネームレジスタ数	64 bit × 64 entry
優先度の指定	256 level
整数演算器	4 + 1 (Divider)
浮動小数点演算器	2 + 1 (Divider)
64 bit 整数演算器	1
分岐ユニット	2
メモリアクセスユニット	1
整数ベクトル演算器	1 (8 IU × 2 line)
浮動小数点ベクトル演算器	1 (4 FPU × 2 line)

は，どのスレッドの命令かに関係なく優先度に従って発行する命令を選択し，Register Unit にてレジスタファイルをアクセスし，リザベーションステーションに命令を送る．命令の実行は Out of Order に行われる．キャッシュシステム，リザベーションステーション，リオーダバッファからの命令の完了においてもスレッドの優先度が用いられる．本研究で設計したベクトル演算器は VINT（Vector Integer）と VFP（Vector Floating Point）にあたる．

5.2 ベクトル演算器

ベクトル演算器のブロック図を**図 3** に示す．ベクトル制御ユニット（Vector Control Unit）は先に述べたベクトルレジスタ（Vector Register）の確保，解放，ベクトルレジスタへアクセスするためのアドレス計算，複合演算命令の処理を行う．またプログラマにより定義された複合演算命令を格納するためのバッファを持つ．同時に複数のスレッドのベクトル演算要求を処理するために，整数，浮動小数点ともに演算パイプライン（Vector Execution Pipeline）を 2 本設ける．ベクトル制御ユニットはリザベーションステーションから命令が発行されると，どのスレッドからのベクトル演算かに関係なく，空いている演算パイプラインに命令を送る．演算パイプラインは演算すべきベクトル要素の数だけベクトルレジスタからデータを読み出し演算を行う．

整数ベクトル演算器の演算パイプラインでは算術，論理，シフト，比較および積和演算を行う．パイプラインはベクトルレジスタからの read，演算に 2 段，ベクトルレジスタへの write の計 4 段で構成され，完全にパイプライン化されている．また 1 つの演算パイプラインで 8 つの整数演算器を持つことにより，1 クロックで 8 つのベクトル要素を並列に演算する．

浮動小数点ベクトル演算器の演算パイプラインは算術，比較，積和演算および整数フォーマットとの変換を行う．パイプラインはベクトルレジスタからの read，演算に 5 段，ベクトルレジスタへの write の計 7 段で構成され，完全にパイプライン化されている．また 1 つの演算パイプラインで 4 つの浮動小数点演算器を持つことにより，1 クロックで 4 つのベクトル要素を並列に演算する．

5.3 ベクトルレジスタ

ベクトル演算ではベクトル長を長くすることにより演算の並列度が上がる．しかし，その分ベクトルレジスタが大きくなり結果としてハードウェア量が増加する．*RMT PU* ではハードウェア量とのトレードオフ

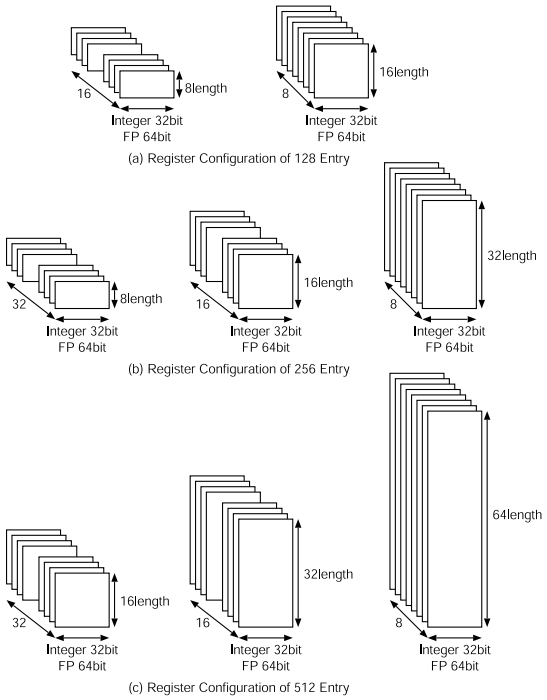

図 4 ベクトルレジスタの構成
Fig. 4 Configuration of vector registers.

から整数，浮動小数点ともに 512 セットのレジスタをベクトルレジスタとして用いる．よって各スレッドはこの 512 セットのレジスタを共有してベクトル演算を行う．

ベクトルレジスタを確保する場合，ベクトルレジスタの個数とベクトル長を指定するが，任意の値を指定するとベクトルレジスタを割り当てるためのロジックが複雑化し，ハードウェア量が増加する．また確保，解放を繰り返すうちに割り当てる領域の断片化が起こる．そこで，マルチメディア処理で用いられる逆離散コサイン変換が 8 × 8 の行列演算であることから，最小のベクトル長を 8 とし，8length，16length，32length，64length の中から選択することにする．**図 4** に *RMTPU* で選択できるベクトルレジスタの構成を示す．

(a) は 128 個のベクトルレジスタを確保した場合の構成を示している．この場合，選択可能な構成は，ベクトル長 8 のレジスタを 16 個，もしくはベクトル長 16 のレジスタを 8 個のどちらかとなる．(b) は 256 個のベクトルレジスタを確保した場合の構成で，ベクトル長 8 のレジスタを 32 個，ベクトル長 16 のレジスタを 16 個，もしくはベクトル長 32 のレジスタを 8 個持つといった構成が選択可能である．(c) は 512 個のベクトルレジスタを確保した場合で，ベクトル長 16 のレジスタを 32 個，ベクトル長 32 のレジスタを 16 個，

6	5:4	3:0
Busy	Address	Mode

図 5 レジスタステータステーブルのフォーマット
Fig. 5 The format of a register status table.

ベクトル長 64 のレジスタを 8 個といった構成が選択可能である．

本機構ではベクトルレジスタの確保，解放を行うために新たに 2 つの命令を追加する．Vector Reserve（VRES）命令はベクトルレジスタを確保する命令で，オペランドとして先に述べたベクトルレジスタの構成を指定する．各スレッドは各々自分の使用するレジスタ数に合わせて構成を選択することにより，図 4 の (a)，(b)，(c) の構成を混在して使用することが可能である．VRES 命令が実行されるとベクトル制御ユニットはベクトルレジスタの空き領域を調べ，割当て可能な場合は割り当てた領域とベクトルレジスタの構成をレジスタステータステーブルに書き込み，デスティネーションレジスタに成功したことを示す 1 を書き戻す．割当て可能な領域がない場合はデスティネーションレジスタに失敗したことを示す 0 を書き戻す．プログラマはデスティネーションレジスタの値を調べることにより，ベクトルレジスタが割り当てられたかどうかを確認することが可能である．Vector Release（VREL）命令はベクトルレジスタを解放する命令である．VREL 命令が実行されるとベクトル制御ユニットは，そのスレッドに割り当てていたベクトルレジスタを解放し，レジスタステータステーブルをクリアする．開放されたベクトルレジスタは，他のスレッドの VRES 命令により再び割り当てられる．

レジスタステータステーブル（Register Status Table）はベクトル制御ユニットのレジスタコントローラ（Vector Register Controller）内に位置し，各スレッドのベクトルレジスタの割当て情報を格納する．**図 5** にレジスタステータステーブルのフォーマットを示す．Busy bit はそのスレッドがベクトルレジスタを確保しているかどうかを示す．Address は確保したレジスタの先頭アドレスを示す．先に述べたとおり，512 個のレジスタを最小 128 個単位で割り当てるため，2 bit で先頭アドレスを表す．Mode は図 4 に示したベクトル長とレジスタ個数といったベクトルレジスタの構成を示す．

ベクトル制御ユニットは，ベクトル演算命令を受け取るとそのスレッドのレジスタステータステーブルを調べる．ベクトルレジスタを確保せずにベクトル演算を行おうとした場合は例外を発生させる．ベクトルレ

29	28:23	22:17	16:11	10:0
Next	Rd	Rt	Rs	Operation

図 6 複合命令バッファのフォーマット
Fig. 6 The format of a compound instruction buffer.

ジスタが確保されていた場合，ベクトル制御ユニットはデコードされたレジスタ ID をベクトルレジスタにアクセスするための実アドレスに変換する．レジスタステータステーブルの Mode からベクトル長を調べ，ベクトル長によりレジスタ ID を左シフトする．さらに Address から割り当てられた領域の先頭アドレスを取得し，それを足すことにより実際にアクセスするアドレスを得る．演算パイプラインはベクトルレジスタのこのアドレスからベクトル長だけ連続してデータを読み込み，演算を行う．

5.4 複合演算機構

複合演算命令は，プログラマがベクトル制御ユニット内の複合命令バッファ（Vector Compound Instruction Buffer）に一連の命令を定義する．複合命令バッファのフォーマットを**図 6** に示す．

組み合わせる命令 1 つにつき 1 つのエントリを使用し，複数のエントリを用いることにより複合命令を定義する．複合演算バッファは 32 エントリあり，ベクトル演算を行うスレッドで共有して用いる．使用できるバッファ数は確保したベクトルレジスタの個数に比例し，128 個を確保した場合は 8 エントリ，256 個を確保した場合は 16 エントリ，512 個を確保した場合は 32 エントリを使用する．

Operation には演算パイプラインで演算するための制御情報を格納する．Rs，Rt，Rd にはそれぞれ演算で使用するソースおよびデスティネーションのベクトルレジスタを示すためのオフセット値を指定する．実際に演算で用いるベクトルレジスタは，次に述べる複合演算実行命令で指定されたオペランドに Rs，Rt，Rd で指定された値を足した ID のベクトルレジスタを用いる．エントリで指定する Rs，Rt，Rd を変えることにより異なるベクトルレジスタを用いることができる．Next Bit は複合演算が続くかどうかを指定する．複合命令の定義は複数のエントリを用いて行うため，Next Bit により複合命令の終わりを指定する．Next Bit が 1 の場合は続く命令があることを示し，Next Bit が 0 の場合は複合命令が終了したことを示す．Next Bit を用いて複合命令の定義の終わりを示すことにより，複合命令バッファが空いていれば複数の複合命令を定義することが可能である．

複合命令バッファへの書込みは，Define Compound Instruction（DCI）命令で行う．この命令は，複合命令バッファの指定したアドレスにプログラマが図 6 のように定義した命令を書き込む．

プログラマが定義した命令は新たに追加した Execute Compound Instruction（ECI）命令を用いて実行する．繰り返し複合演算を行う場合は ECI 命令を複数回実行する必要がある．ベクトル制御ユニットは ECI 命令が発行されると複合命令バッファの最初のエントリの命令を取り出し，演算に使用するベクトルレジスタのアドレスを計算し，Operation とともに空いている演算パイプラインへ送る．Next bit がセットされている場合は次の命令を複合命令バッファから取り出し，次々と空いている演算パイプラインに送る．ベクトル演算器はパイプラインチェイニングを行うことにより，最初のベクトル要素の演算が完了すると，その演算結果を用いてすぐに次のベクトル演算を開始する．一方 Next bit がセットされていない場合は，リザベーションステーションから送られてくる次のベクトル命令を処理する．

6. 評　　価

RMT Processor は現在，実機による評価環境が整っていないため，評価は実チップの設計・実装に用いた RTL によるシミュレーションによって行った．

6.1 ベクトルレジスタの動的構成変更の評価

本研究で設計したベクトル演算機構を評価するために，ベクトル演算を用いて 8×8 の逆離散コサイン変換を行うプログラムを作成した．実行スレッド数を 1，2，4 スレッドとしたときに，ベクトルレジスタの構成を固定にした場合とスレッド数に合わせて動的に変更した場合について評価を行った．構成を固定にした場合，ベクトルレジスタは各スレッドで 128 セットを使用し，動的に変更した場合，ベクトルレジスタの構成は 1 スレッドのときは 512 セット，2 スレッドのときは 256 セット，4 スレッドのときは 128 セット使用した．プログラムでは 45%がベクトル演算命令になっている．このプログラムを用いて，それぞれ逆離散コサイン変換にかかる実行時間を計測した．

図 7 に実行結果を示す．並列に実行するスレッド数を増やすことにより，ベクトル演算の性能が 120%（4 スレッドの場合）向上した．また，スレッド数に合わせてベクトルレジスタの構成を変更することにより，ハードウェア資源を効率的に使用することができ，ベクトルレジスタの構成を固定にした場合よりも 19%（1 スレッドの場合）性能が向上した．

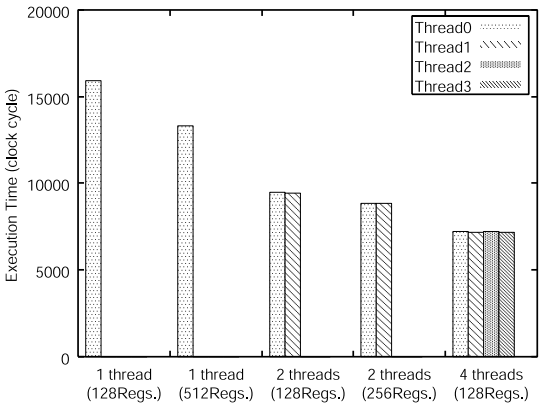

図 7 逆離散コサイン変換の実行時間
Fig. 7 The IDCT execution time.

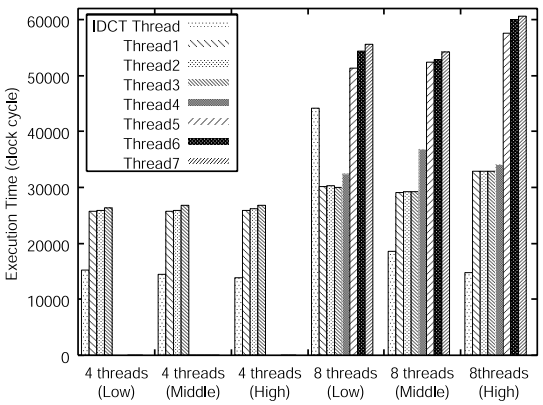

図 8 優先度を用いた場合の逆離散コサイン変換
Fig. 8 The IDCT execution time with a priority.

6.2 優先度制御時の評価

RMT PU は優先度を用いて各スレッドの命令実行を制御する．そこで複数のスレッドが優先度付きで並列に実行されている場合に，ベクトル演算を用いて逆離散コサイン変換の実行にかかる時間を測定した．並列に実行されるスレッドとして整数演算を行うプログラムを用いた．このプログラムは逆離散コサイン変換をスカラ演算で行うもので，これにより命令フェッチ（命令キャッシュアクセス），リザベーションステーションへの命令発行スロット，メモリアクセス（データキャッシュアクセス），リオーダバッファからのコミット処理においてベクトル演算を行うスレッドとの資源競合が起きる．並列に整数演算を行うスレッドの数を 3 と 7 にした場合について，ベクトル演算を行うスレッドの優先度を全スレッド中最低にした場合と，ちょうど中間になるようにした場合，最高になるようにした場合について評価を行った．

図 8 に実行結果を示す．IDCT Thread は逆離散コサイン変換を行うスレッドで Thread1 から 7 は整数演算を行うスレッドを示している．また整数演算を行うスレッドは Thread1 が最も優先度が高く，Thread7 が最も優先度が低くなるように設定した．Low は IDCT Thread の優先度を全スレッド中最低にした場合を示し，Middle は優先度を全スレッドの中間にした場合，High は最も優先度を高くした場合を示している．

実行しているスレッドが 4 スレッドの場合，ベクトル演算を行うと優先度が低い場合でも 1 スレッド単体で実行したときとほぼ同じ実行時間を実現している．これはベクトル演算により，少ない命令フェッチを有効に利用することができたためだと考えられる．

一方，8 スレッドが並列に実行されている場合，逆離散コサイン変換を行うスレッドの優先度が低い場合は，優先度による計算資源の調停のために，逆離散コサイン変換を行うスレッドの実行があとになるため，単独で実行した場合に比べて実行時間が増加している．また，優先度を中間にした場合においても，実行するスレッド数が多いため，優先度の高いスレッドが実行されている間は，ベクトル演算の命令フェッチが 4 スレッドのときほど頻繁に行われず，単独で実行した場合に比べて実行時間が増加している．

優先度が高い場合は，優先的にベクトル演算の命令フェッチが行われるため，スレッド数が増加した場合でも，実行時間は増加しなかった．

6.3 複合演算機構の評価

複合演算機構の評価を行うために前節のベクトル演算を用いた逆離散コサイン変換のプログラムを，複合演算命令を使用したプログラムに変更して実行時間を計測した．プログラムでは 8 つの積和演算を行う複合命令を定義した．**図 9** に実行結果を示す．

複合演算機構を用いることにより，複合演算機構を用いない場合に比べて 39%（4 スレッドの場合）性能が向上した．これは先に述べたとおり，優先度の低いスレッドは優先度の高いスレッドが実行されている間は，フェッチされる命令数が少ないため，1 つの命令でより多くの演算を行うことによりベクトル演算器の使用率が上がり，性能が改善したと考えられる．特に 8 スレッドを並列に実行し，優先度を中間にした場合は，複合演算機構を用いなかった場合，命令フェッチが少ないため演算時間が増加したのに対し，複合演算機構を用いることにより，より長いレイテンシの命令を実行するため，少ない命令数でも演算性能が向上した．これにより 1 スレッド単体で実行した場合と同等の性能を達成した．優先度が高い場合でも性能が改善したのは，Instruction Issue Unit において，1 つ

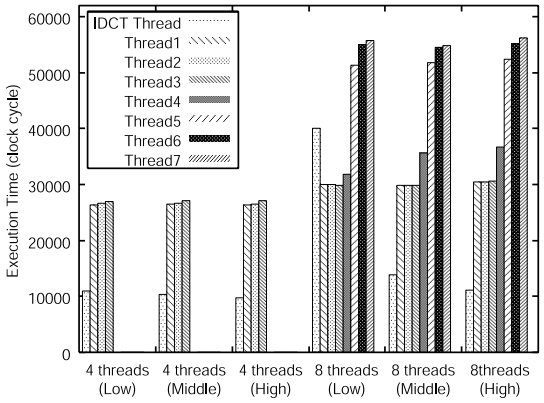

図 9 複合演算を用いた逆離散コサイン変換
Fig. 9 The IDCT execution time with compound operations.

のスレッドは最大で 2 サイクルに一度命令フェッチを行うようになっており，また，複合命令により 8 つの命令が 1 つの複合命令に置き換わったことにより，リオーダバッファや命令バッファが有効に使われたためと考えられる．

一方 8 スレッド並列実行で優先度を低くした場合は，複合演算機構を用いない場合と同様に，優先度の高いスレッドの実行が完了するまで命令キャッシュにアクセスできないため，複合演算命令がフェッチされず，ベクトル演算器が使用されなかった．そのため単体で実行したときに比べて実行時間が増加している．しかし，優先度の高いスレッドが終了し，命令キャッシュへのアクセスが少しずつ割り当てられるようになると，複合演算命令によりベクトル演算器の使用率が増加するため，複合演算機構を用いなかった場合に比べて 10%性能が向上した．

複合演算を用いた場合，複合演算を用いない場合に比べて優先度の高いスレッドの実行時間が少し増加している．先に述べたとおり，ベクトル演算は命令フェッチ，リザベーションステーションへの命令発行スロット，データアクセス，リオーダバッファからのコミット処理において，優先度の高いスレッドのスカラ演算と資源の競合が起こる．命令フェッチ，リザベーションステーションへの命令発行およびリオーダバッファからのコミット処理では，*RMT PU* の優先度を用いた制御により，クロックごとにより優先度の高いスレッドの命令から処理が行われる．データキャッシュアクセスにおいても優先度による制御が行われるため，優先度による制御が行われるが，ベクトルデータの Load/Store はデータ数が多いため 1 クロックサイクルでは終了しない場合がある．そのため，ベクトルデータの Load/Store が行われているときに，より優先度の高いスレッドのデータキャッシュアクセスがあると，その命令はベクトルデータの Load/Store が完了するまで待たされることになり，優先度逆転の問題が発生する．またデータキャッシュミスが起こり，メモリアクセスが発生した場合においても，ベクトルデータのメモリアクセスが開始されると，後から優先度の高い命令が来た場合，その命令はベクトルデータの転送が終わるまで待たされる．複合演算を行うことにより，ベクトル演算のスループットが増加するため，データキャッシュへのアクセスが増加する．このため優先度逆転の問題が増加し，優先度の高いスレッドの実行時間が増加したと考えられる．しかし，データキャッシュとベクトルレジスタの間のバスを 256 bit とし，メモリアクセスにおいてバースト転送を行うことにより，ベクトルデータの Load/Store にかかるクロック数を小さくすることにより，この影響を小さくすることができると考えられる．また，キャッシュアクセスにおいては，ベクトルデータのアクセスを中断する機構を追加することにより，優先度逆転の問題を回避できると考えられる．

7. 結　　論

本研究では，リアルタイム処理用プロセッサである *Responsive MultiThreaded*（*RMT*）*Processor* の命令実行機構を設計・実装した．命令実行機構では，特にソフトリアルタイム処理で要求される演算性能を達成させるためにベクトル演算機構を設計した．

RMT PU では細粒度マルチスレッディングに優先度を取り入れて，リアルタイム処理を支援している．通常，マルチスレッディングでは複数のスレッドから依存関係の少ない命令を実行することにより，全体のスループットを改善している．しかし *RMT PU* では，時間制約を満たすために，優先度の高いスレッドを優先して実行するため，分岐予測ミス等により全体のスループットは低下する．また優先度の高いスレッドがストールしているときにのみ，細粒度マルチスレッディングにより，優先度の低いスレッドが即座に実行される．このため，優先度の低いスレッドの命令発行率は低くなる．

そこで，ベクトル演算機構を用いて少ない命令数で多くのデータを演算することにより，優先度の低いスレッドのスループットを向上した．また，動的にベクトルレジスタの構成を変更することにより，柔軟に複数のスレッドで並列にベクトル演算を行うことを可能とした．複合演算機構を用いることにより，より少な

い命令数で演算を行うことができ，ベクトル演算器の使用率を向上させ，ベクトル演算の性能が向上した．

本研究のベクトル演算機構を実装した *RMT Processor* は現在実チップを用いた評価環境を構築中であり，今後は実機を用いて，より現実的な環境で評価を行っていく予定である．

謝辞 本論文の研究は，文部科学省の科学技術振興調整費の支援による．また本研究の一部は，科学技術振興事業団 SORST の支援による．

参 考 文 献

1) 山﨑信行，堀 俊夫：分散リアルタイムネットワーク用プロセッサとその応用，情報処理，Vol.44, No.1, pp.6–13 (2003).
2) Utiyama, M., Itou, T., Sato, J., Yamasaki, N. and Anzai, Y.: A new processor architecture for real-time systems, *IFAC Conference on New Technologies for Computer Control 2001* (2001).
3) 佐藤純一，内山真郷，伊藤 務，山﨑信行，安西祐一郎：リアルタイム処理用マルチスレッディングプロセッサの優先度に基づくキャッシュサブシステム，情報処理学会研究報告 2001-ARC-143, Vol.2001, pp.37–42 (2001).
4) Eggers, S.J., Emer, J.S., Henry M, L., Lo, J.K., Stamm, R.L. and Dean, M. T.: Simultaneous multithreading: A platform for next-generation processors, *IEEE Micro*, Vol.17, No.5, pp.12–19 (1997).
5) Tullsen, D.M., Eggers, S.J., Emer, J.S., Henry, M.L., Lo, J.K. and Stamm, R.L.: Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor, *Proc. 23rd Annual International Symposium on Computer Architecture* (1996).
6) Espasa, R. and Valero, M.: Multithreaded Vector Architectures, *Third International Symposium on High-Performance Computer Architecture*, pp.237–249 (1997).

(平成 15 年 2 月 3 日受付)
(平成 15 年 5 月 8 日採録)

伊藤 務（学生会員）

1977 年生．2000 年慶應義塾大学理工学部情報工学科卒業，2002 年同大学大学院理工学研究科開放環境科学専攻修士課程修了．現在，同博士課程に在籍．リアルタイムシステム，システム LSI 等の研究に従事．

山﨑 信行（正会員）

1966 年生．1991 年慶應義塾大学理工学部物理学科卒業，1996 年同大学大学院理工学研究科計算機科学専攻博士課程修了．博士（工学）．同年電子総合研究所入所，1998 年 10 月慶應義塾大学理工学部情報工学科助手．2000 年 4 月同専任講師．現在，産業技術研究所客員研究員を兼任．並列分散処理，リアルタイムシステム，システム LSI 等の研究に従事．