リアルタイム処理用
Responsive Multithreaded Processorの
スレッド制御機構及び演算機構に関する研究

平成17年度

伊藤 務

論文要旨

現在，ロボット，ホームオートメーション，ファクトリオートメーションなど多くのシステムでリアルタイム処理が行われている．リアルタイム処理にはハードリアルタイム処理とソフトリアルタイム処理がある．ハードリアルタイム処理は時間制約が厳しく，演算量が少ない．ソフトリアルタイム処理は時間制約は厳しくないが，演算量が多い．このような性質の異なる処理を同時に実行する場合，タスクの時間制約を基に優先度を決定し，優先度に従ってタスクを切り替えながら実行する．タスクを切り替える場合，コンテキストスイッチが発生するが，ソフトウェアによるコンテキストスイッチは多くのメモリアクセスを行うため，オーバーヘッドが大きいという問題がある．また，ソフトリアルタイム処理では大量のデータを演算するため，演算資源を占有するという問題がある．よって，ハードウェアでリアルタイム処理を支援する場合，小さいオーバーヘッドでコンテキストスイッチを行う機能と，ソフトリアルタイム処理の大量のデータを演算する機能が必要であると考えられる．

リアルタイム処理用プロセッサとして設計が行なわれている Responsive Multithreaded（RMT）Processor のプロセッシングユニットである RMT PU は，スループットを向上しながら，8 個までのスレッド（タスク）を，コンテキストスイッチを行なわずに優先度に従って順番に実行する機構を持つ．本研究は 9 個以上のスレッドを実行した場合でも，小さいオーバーヘッドでコンテキストスイッチを行うために，スレッド制御機構を設計，実装する．また，ソフトリアルタイム処理の大量のデータを演算するために，ベクトル演算機構を設計，実装する．

スレッド制御機構では，コンテキストキャッシュと呼ぶオンチップメモリに各スレッドのコンテキストを保持し，ハードウェアで RMT PU のハードウェアコンテキストと入れ替える．割り込みに対する応答性を向上するために，割り込み要因によりスレッドを自動的に起動する．起動されたスレッドは優先度に従ってハードウェアが自動的にコンテキストスイッチを行い，RMT PU のハードウェアコンテキストと入れ替える．スレッド制御機構により，ソフトウェアによるコンテキストスイッチに比べて，0.4%の時間でコンテキストスイッチが完了した．スレッド制御機構により，処理時間が短いスレッドを実行した場合でも，RMT PU の優先度制御機構によってコンテキストスイッチのオーバーヘッドが隠蔽されることを示した．

ベクトル演算機構では，複数のスレッドでベクトルレジスタを効率よく共有するために，柔軟にベクトルレジスタの構成を変更する機構を持つ．複合演算機構により，より多くの演算を一命令で実行する．RMT PU の優先度制御機構により，スレッドの優先度が低く，命令スロットが割り当てられない場合でも，複合演算機構を用いることにより，ベクトル演算器はベクトル演算を続けることができるため，演算性能が維持されることを示した．また，ソフトリアルタイム処理の演算をベクトル演算器で行うため，RMT PU の優先度制御機構により，空いている演算資源を用いてより優先度の低いスレッドを実行することができ，システム全体のスループットが向上することを示した．

本研究で提案した機構により，ハードリアルタイム処理とソフトリアルタイム処理をハードウェアレベルから支援でき，より高度なリアルタイムシステムを構築することが可能となる．

Abstract

Real-time processing is required for realizing a lot of systems including the robot, the home automation, the factory automation, and so on now. Priority is added to each task based on the timing constraint in real-time processing. Each task is switched and executed in priority order to guarantee the timing constraint. Context switching occurs when tasks are switched. An overhead of context switching is large if context switching is performed by software. It becomes a problem when the frequency of task switching increases.

Responsive Multithreaded (RMT) Processor is designed for distributed real-time processing. RMT PU, a processing unit of RMT Processor, executes eight threads (tasks) in priority order without context switching. But Context switching arises as well as traditional processors if nine threads or more are executed.

Large amount of calculation is required for soft real-time processing. As a lot of computing resources are occupied by soft real-time processing, other operations are not likely to be executed. In this paper, we propose the thread control mechanism and the vector operation mechanism to resolve these problems and support hard real-time processing and soft real-time processing by hardware.

Each thread is held in on-chip memory called context cache in the thread control mechanism. Context switching between the hardware context of RMT PU and context cache is performed by hardware. Threads are automatically started by hardware based on the interrupt factor to improve the interrupt response. The started thread is automatically switched and executed by hardware based on priority. The time of context switching has reduced by 0.4% by the thread control mechanism compared with software. It showed that an overhead of context switching was hidden by the priority control mechanism of RMT PU even when the thread with short processing time was executed in the thread control mechanism.

Multiple threads share vector registers in the vector operation mechanism. A configuration of vector registers is changed flexibly to share vector registers efficiently. Multiple vector operations are performed with one instruction to continue vector calculations even if instruction slots are not allocated by the priority control mechanism of RMT PU. Lower priority threads are executed by using the remaining resources because the calculations for soft real-time processing are performed by the vector unit. So the throughput of entire system improved.

It is considered that more advanced real-time systems can be built by using the proposed mechanism because hard real-time processing and soft real-time processing are supported by hardware.

目 次

図 目 次

表 目 次

第1章

序論

リアルタイム性とは，処理の真偽が結果の真偽だけでなく，時間にも依存する性質である [A. 88]．狭義には与えられた時間制約（デッドライン）を守ることを意味する．ロボット，ホームオートメーション，ファクトリオートメーションなど多くのシステムでリアルタイム性を持った処理を実行することが多くなってきている．

リアルタイム処理はその性質により大きくハードリアルタイム処理とソフトリアルタイム処理に分かれる．ハードリアルタイム処理は時間制約が厳しく，時間制約が守られなかった場合，システムに与える影響が大きい．ソフトリアルタイム処理はハードリアルタイム処理ほど時間制約が厳しくなく，時間制約が守られない場合でも，システムに与える影響は少ない．しかし，ソフトリアルタイム処理では大量のデータを演算する必要がある．

例えば動画を表示する場合，一定時間毎に次々と画像を切り替えて描画することにより画像が動いているように見える．描画処理に時間がかかり，時間内に次の画像の表示が行われない場合，動画はなめらかに表示されず，見ているユーザは違和感を感じてしまう．つまり，一定時間毎に描画が完了しなければならないという時間制約を持つ．しかし，時間制約が守られなかった場合でもシステムに与える影響は少ないため，このような処理はソフトリアルタイム処理となる．

ロボットのように，より複雑なシステムでは，時間制約を持った処理が複数存在する．例えばロボットが人間と対話をしながら前進するような場合，アクチュエータを動作させる処理，人間とコミュニケーションを行うための処理などが実行される．アクチュエータを動作させる処理では，一定時間毎にセンサなどからデータを取得し，アクチュエータの操作量を演算し，アクチュエータを制御する．センサが障害物を検知した場合，アクチュエータの操作が遅れるとロボットは障害物に激突してしまう．そのため，時間制約が守られなかった場合，システムに与える影響は大きい．よって制御のような処理はハードリアルタイム処理である．一方，コミュニケーションを行うための処理では，人間と会話を行うために，マイクからの入力を音声認識し，人間からの問いかけに対する返事を生成し，音声合成を行ってスピーカから返事を出力する．会話の応答が遅れると，人間とコミュニケー

ションをとるのは難しくなる．しかし，時間制約が守られない場合でも，システムに与える影響は少ない．よってこのような処理はソフトリアルタイム処理である．これらの処理はシステム内で同時に実行され，どちらの処理も時間内に完了する必要がある．

現在，プロセッサの設計は主にスループットを向上することが主眼になっている．製造プロセスの微細化により，1チップに集積できるゲート数が増加し，ゲート遅延が小さくなってきている．プロセッサのスループットを向上するためには，動作周波数を上げれば良い．そのため現在では動作周波数がGHz台のプロセッサが登場している．一方では動作周波数の限界からプロセッサ内で複数の処理を並列実行することにより，スループットを向上するプロセッサ（オンチップマルチプロセッサ，マルチスレッドプロセッサ）が登場しはじめている．このように，現在のプロセッサ設計はスループットを向上することを目的としているため，リアルタイムシステムにおいて，各処理の時間制約を守るための機構は主にソフトウェアで実装されている．

従来のリアルタイムシステムでは，まずスケジューラが，各処理の時間制約（実行周期やデッドライン）を基に，優先度（実行順序）を決定する．スケジューラは優先度に従って処理を切り替えながら実行することにより，各処理の時間制約を守る．実行する処理を切り替える場合，コンテキストスイッチが発生する．ソフトウェアによるコンテキストスイッチは大量のメモリアクセスが発生するため，オーバーヘッドが大きい．

今後，より高度なリアルタイムシステムを構築する場合，システム内で実行する処理の数が増加することが考えられる．また，より細かい制御を行うために，より短かい実行周期で処理を行うことが考えられる．このような場合，処理の切り替え回数が増加するため，コンテキストスイッチよるオーバーヘッドの割り合いが大きくなる．よって従来どおりソフトウェアのみで時間制約を守ることは困難になる．今後はソフトウェアだけでなく，ハードウェアレベルからリアルタイム処理を行う機構が必要であると考えられる．

ロボットのようなシステムではハードリアルタイム処理とソフトリアルタイム処理といった，性質の異なる処理を同時に実行する．このような場合，各処理の時間制約を守りながら，ソフトリアルタイム処理の大量のデータを演算しなければならない．しかし，大量のデータを演算すると，ソフトリアルタイム処理が演算資源を占有してしまい，他の処理を行うことができなくなる．よってハードウェアで大量の演算を行う機構が必要となる．

以上のことから，ハードリアルタイム処理とソフトリアルタイム処理の両方を同時にハードウェアレベルから支援するために，小さいオーバーヘッドでコンテキストスイッチを行う機能と，ソフトリアルタイム処理の大量のデータを演算する機能が必要であると考えられる．小さいオーバーヘッドでコンテキストスイッチを行うことにより，時間制約を守るために処理を頻繁に切り替えられた場合でも，システム全体でコンテキストスイッチに費やす時間の割り合いを削減する．また，ソフトリアルタイム処理の大量のデータを演算す

る機能により，ソフトリアルタイム処理でデータを演算するために，演算資源を占有してしまい，他の処理が実行できなくなることを防ぐ．

コンテキストスイッチのオーバーヘッドを削減する方法として，マルチスレッドアーキテクチャを用いる手法がある．しかし，ハードウェアコンテキストよりも多くの処理を実行しようとした場合，ソフトウェアによるコンテキストスイッチを行わなければならないため，オーバーヘッドが生じる．ソフトリアルタイム処理のデータを演算するための方法として，データ並列性を用いた手法がある．しかし，この手法はハードリアルタイム処理とソフトリアルタイム処理の両方を同時に実行することを考えていない．ハードリアルタイム処理を同時に実行した場合，優先度によって処理が切り替えられて実行されるため，ソフトリアルタイム処理の優先度が低い場合に十分な演算性能が得られない．

Responsive Multithreaded（RMT）Processorはリアルタイム処理を行うためのプロセッサとして設計，実装が行われている．RMT Processorのプロセッシングユニットである RMT PUはSimultaneous Multithreadingアーキテクチャに優先度制御を取り入れることにより，コンテキストスイッチを行わずに複数のスレッドを優先度に従って実行する機能を持つ．しかし，コンテキストスイッチを発生させずに実行できるのは，ハードウェアコンテキスト数までであり，それ以上のスレッドを実行する場合は従来通りソフトウェアによるコンテキストスイッチが発生するといった問題がある．また，ソフトリアルタイムスレッドを実行する場合，大量のデータを演算するが，スレッドの優先度が低い場合，RMT PUの優先度制御により十分に演算資源が割り当てられないといった問題がある．

本研究ではハードリアルタイム処理とソフトリアルタイム処理の両方を同時に実行した場合に生じる，以上の問題を解決するために，スレッド制御機構とベクトル演算機構を設計，実装する．スレッド制御機構は，ソフトウェアで行っていたスレッドの切り替えを，ハードウェアで行うことにより，オーバーヘッドの小さいコンテキストスイッチを実現する．ベクトル演算機構は，RMT PUの優先度制御を考慮してベクトル演算を行うことにより，優先度の高いスレッドが実行中であっても，ソフトリアルタイム処理の大量のデータを効率良く演算する．

小さいオーバーヘッドでコンテキストスイッチを行うことにより，より短かい時間粒度でのスレッドの切り替えが可能となる．例えば，ロボットが動作を行う場合，センサからデータを取得し，得られたデータと目標値からPID制御のようなフィードバック制御を行って次の操作量を演算する．演算を行う周期を短かくすることにより，より細かく操作量を制御することが可能となる．また，コンテキストスイッチのオーバーヘッドが削減された分，より多くのスレッドを実行することが可能となる．

リアルタイムシステムでは割り込みにより，スレッドが処理を開始する場合が多い．例えばロボットの場合，センサが障害物を感知すると，割り込みを発生させ，回避行動を行

うスレッドを起動する．この場合，障害物の感知から回避行動まで時間がかかると，ロボットは障害物に激突する．そのため割り込みに対するリアクティブ性 [Benveniste *et al.* 92] が重要になる．割り込み処理では，割り込みハンドラにより割り込み要因が調べられ，コンテキストスイッチを行って割り込みに応答するスレッドが起動される．コンテキストスイッチのオーバーヘッドを削減することにより，割り込み応答性を向上することが可能となる．

ソフトリアルタイム処理を行う場合，大量のデータを演算するためには，多くの演算資源が必要である．RMT PUでは優先度によってスレッドに演算資源を割り当てているため，優先度の高いスレッドを実行している場合，優先度の低いスレッドは十分に演算資源を割り当てられない．ベクトル演算機構により，優先度が低い場合でも十分に演算資源を使用することが可能となる．また，ソフトリアルタイム処理の演算をベクトル演算器で行うことにより，通常の演算器（スカラ演算器）が空くため，それを利用して別の処理を実行することにより，システム全体としてのスループットを向上することが可能となる．

以上，本研究により，ハードリアルタイム処理とソフトリアルタイム処理の両方を同時に実行する場合，性質の異なるこれらの処理をハードウェアレベルから効率良く実行することが可能となる．本論文の構成は以下の通りである．2章ではリアルタイム処理について述べ，リアルタイム処理を行うための従来手法について述べる．そして，従来研究の問題点からハードウェアレベルでリアルタイム処理を実現するために必要な機能を検討する．3章では関連研究について述べ，従来研究の問題点について述べる．4章ではリアルタイム処理用プロセッサとして設計，実装が行われている Responsive Multithreaded（RMT）Processor のプロセッシングユニットである RMT PU について説明し，本研究の研究範囲を明確にする．5章ではハードウェアによるスレッド制御機構を提案する．6章では RMT PU の優先度制御を考慮したベクトル演算機構を提案する．7章でスレッド制御機構とベクトル演算機構の有効性を示す．そして8章で本研究をまとめる．

第2章

リアルタイム処理

本章では，リアルタイム処理について述べる．まず，リアルタイム性の特徴について述べ，リアルタイム処理の基本的な実現方法について述べる．そして従来研究の問題点から，ハードウェアレベルでリアルタイム処理を支援するために必要な機能を検討する．

2.1 リアルタイム性

リアルタイム性とは，処理の真偽が結果の真偽だけでなく，時間にも依存する性質のことを言う．狭義には与えられた時間制約（デッドライン）を守ることを意味する．

リアルタイム性は，時間制約の種類により大きく2つに分けられる．

- ハードリアルタイム性
- ソフトリアルタイム性

ハードリアルタイム性とは，必ず時間制約を守る必要がある性質である，時間制約が守られなかった場合，結果の価値が直ちに0になる．ハードリアルタイム性を持つ処理は時間制約が厳しいため，実行時間の見積もりの正確さ（実行時間の予測性）が重要になる．ハードリアルタイム性を持つ処理は実行周期が100μsから10msと短い，演算量が少ないといった特徴がある．ロボットのアクチュエータをPID制御する処理などがハードリアルタイム性を持つ．

例えばロボットが障害物を回避する場合，時間内にアクチュエータを操作し，進行方向を変更する必要がある．時間内にアクチュエータを制御して方向を変えなければ，ロボットは障害物に衝突してしまい，場合によってはロボットが破損してしまうため，システムに重大な影響をおよぼす．

ソフトリアルタイム性とは，時間制約を破ることを許容する性質である．時間制約が守られなかった場合でも結果の価値が直ちに0にはならず，時間経過とともに結果の価値が減少していく．ソフトリアルタイム性を持つ処理は時間周期が10msから1sと比較的大き

表 2.1: リアルタイム性

	ハード リアルタイム性	ソフト リアルタイム性
デッド ラインミス時の影響	厳しい	比較的厳しくない
実行周期	短かい	長い
演算量	少ない	多い

い，演算量が多いといった特徴がある．音声合成や画像処理のように，マルチメディア処理などがソフト リアルタイム性を持つ．ソフト リアルタイム性を持つ処理はデッド ラインミスに関してはハード リアルタイム性を持つ処理よりも寛容だが，大量のデータを演算する．例えば動画を表示する場合，時間内に一枚の画像を完成しなければならない．一枚の画像を完成させるためには 1 ピクセル毎に画素値を計算するため演算量が多くなる．時間内に画像が完成すれば，動画はスムーズに動いて見えるが，時間内に画像が完成しないと，動画はスムーズに動いて見えない．しかし，多少デッド ラインをミスしたとしても，動画の表示に与える影響は少ない．

以上，ハード リアルタイム性とソフト リアルタイム性について特徴をまとめると表 2.1 のようになる．リアルタイム処理では，このように性質の異なる処理が実行される．

2.2 スケジューリング

リアルタイムシステムではシステムで実行される処理の時間制約を守る必要がある．システムで実行する処理の数が少なく，閉じた環境の場合は，全ての処理の振る舞いを解析し，プログラマがシステムで実行する全ての処理の時間制約を守れるようにプログラミングを行うことにより，時間制約を守ることができる．しかし実行する処理の数が多くなり，外界の要因により処理が変わるようなシステムでは解析が複雑になる．このような場合，従来のリアルタイムシステムではリアルタイムオペレーティングシステム（RT-OS）を導入することにより，各処理の時間制約を守る．

RT-OS では各処理の時間制約を守るために，リアルタイムスケジューラが以下のことを行う．

- 処理の時間制約（実行周期，デッド ラインなど）を基に，各処理の優先度（実行順序）を決定
- 優先度に従って処理を切り替えて実行

スケジューラはシステムで実行する全ての処理に対して，優先度を決定する．優先度の決定方法は様々な手法が提案されており，システムの実行前に静的に決定する方法や，システムの実行中に動的に優先度が変化する方法がある．優先度によって処理の実行順序が決定すると，スケジューラは実行可能な処理の中から優先度の高い順に，処理を切り替えながら実行する．

2.2.1 Rate Monotonic スケジューリング

代表的なスケジューリングアルゴリズムに Rate Monotonic（RM）スケジューリング [Lehiczky *et al.* 89] がある．RM スケジューリングは周期スレッドに対して，実行周期の短かいスレッドに高い優先度を与え，優先的に処理を行うスケジューリング方法である．通常，周期は一定であるため，優先度はシステムの実行前から固定で与えることができる．

RM スケジューリングは固定優先度スケジューリングで最適である [Liu *et al.* 73]．RM スケジューリングにおいて，スケジューリングが可能であるための CPU 利用率の最小の上限値 U_{lub} は，スレッド数 n に対して $U_{lub} = n(2^{\frac{1}{n}} - 1)$ である．n が増加すると最終的に $U_{lub} = \ln 2 \cong 0.69$ に収束する．ただし，これは十分条件であるため，実際にスケジューリングが可能であるかどうか判定するためには，Response Time Analysis（RTA）を用いる．RTA では，まずスレッド τ_i の最悪反応時間 R_i を以下の式により，$R_i^{(k)} = R_i^{(k-1)}$ となるまで反復して計算する．

$$
\begin{aligned}
R_i^{(0)} &= C_i \\
R_i^{(k)} &= C_i + \sum_{j:D_j<D_i} \left\lceil \frac{R_i^{(k-1)}}{T_i} \right\rceil C_j
\end{aligned}
$$

求められた全てのスレッドの最悪反応時間が，デッドラインよりも短かい場合，つまり，

$$\forall \tau_i, R_i < D_i$$

を満たす場合にスケジューリング可能であると判断できる．RM スケジューリングでは，スケジューリング可能な CPU 利用率の上限は，次に述べる Earliest Deadline First よりも下回る．

図 2.1 に RM でスケジューリングされたスレッドの実行例を示す．図の矢印はスレッドの起動時刻をあらわしている．それぞれのスレッドは周期的に起動する．例では Thread 0

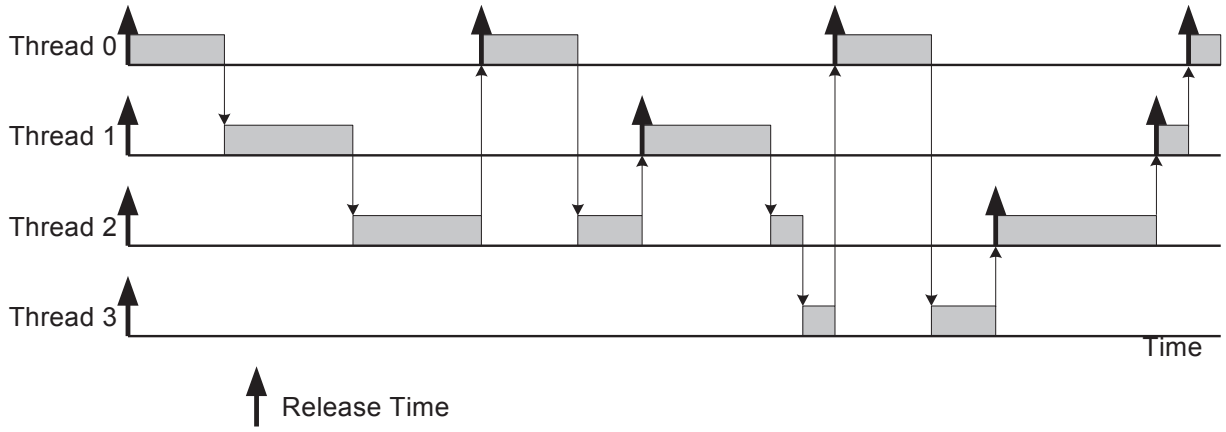

図 2.1: RM スケジューリングの例

の実行周期が最も短かく，Thread 3 の実行周期が最も長い．RM スケジューリングでは周期の短かいスレッドに高い優先度を与えるため，Thread 0 に最も高い優先度が与えられ，Thread 3 に最も低い優先度が与えられる．

最初に全てのスレッドが起動し実行可能になるが，優先度に従ってスレッドを実行するため，最も高い優先度が与えられている Thread 0 が最初に実行される．Thread 0 の実行が完了すると，実行可能なスレッドの中で次に優先度の高い Thread 1，Thread 2 と順番に実行していく．

例では Thread 2 の実行中に Thread 0 が起動し実行可能になる．RM スケジューリングはスレッドのプリエンプション（横取り）を前提としている．プリエンプションでは，あるスレッドの実行中により優先度の高いスレッドが実行可能になると，優先度の高いスレッドの実行を優先するために実行するスレッドを切り替える．そのため，Thread 2 の実行を中断し，Thread 0 に切り替える．そして Thread 0 の実行を開始する．Thread 0 の実行が完了すると，実行可能なスレッドの中で最も優先度の高い Thread 2 の実行を再開する．

Thread 2 の実行中に Thread 1 が起動し実行可能になるため，Thread 2 の実行を再び中断し，実行するスレッドを Tread 1 に切り替えて，Thread 1 を先に実行する．Thread 1 の実行が完了すると Thread 2 の実行を再開する．Thread 2 の実行が完了すると，Thread 3 を実行する．

2.2.2 Earliest Deadline First スケジューリング

RM スケジューリングの他に，代表的なスケジューリングアルゴリズムとして Earliest Deadline First（EDF）スケジューリング [Liu *et al.* 73] がある．EDF スケジューリングでは，デッドラインまでの時間が短かいスレッドに高い優先度を与える．RM スケジューリ

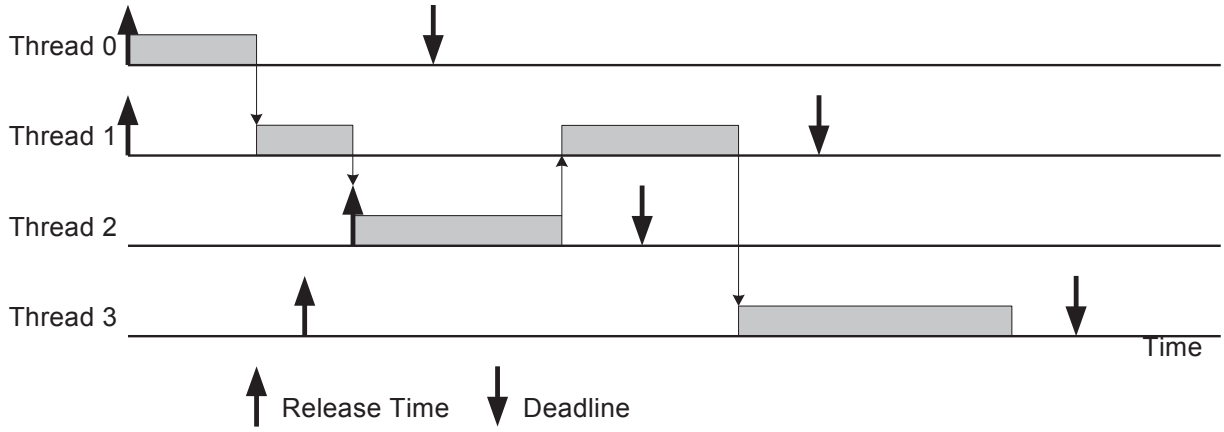

図 2.2: EDF スケジューリングの例

ングと異なり，EDF スケジューリングでは時間経過によってデッドラインまで時間が変化するため，優先度は動的に変化していく．また，EDF スケジューリングは RM スケジューリングと同様にプリエンプションを前提としているため，現在実行中のスレッドよりも優先度の高いスレッドが実行可能になると，そのスレッドに実行が切り替わる．

EDF スケジューリングでは，スケジューリング可能であるための必要十分条件は各スレッドの CPU 利用率の合計 U に対して $U < 1$ である [Liu *et al.* 73]．そのため各スレッドの CPU 利用率の合計からスケジューリング可能であるかを判定することができる．ユニプロセッサでは CPU 利用率が 100%を越えてスケジューリングできないため [Buttazo 97]，EDF スケジューリングは動的優先度スケジューリングにおいて最適なものの 1 つである．

図 2.2 に EDF でスケジューリングされたスレッドの実行例を示す．Release Time（上向き矢印）は各スレッドの起動時刻，Deadline（下向き矢印）は各スレッドのデッドラインをあらわしている．EDF スケジューリングではデッドラインまでの時間が短かいスレッドから高い優先度が与えられるため，例では，Thread 0，Thread 2，Thread 1，Thread 3 の順番に高い優先度が与えられる．

最初に Thread 0 と Thread 1 が起動し，実行可能になる．EDF スケジューリングにより Thread 1 よりも Thread 0 の方に高い優先度が与えられているため，最初に Thread 0 を実行する．Thread 0 の実行が完了すると，実行可能なスレッドの中で最も優先度の高い Thread 1 を実行する．

Thread 1 の実行中に，Thread 2 が起動し実行可能になる．Thread 2 は Thread 1 よりも優先度が高く，EDF スケジューリングはプリエンプションを前提としているため，スケジューラは Thread 2 の実行を優先するために，Thread 1 の実行を中断し，Thread 2 に実行を切り替える．Thread 2 の実行が完了すると，実行可能なスレッドの中で最も優先度の

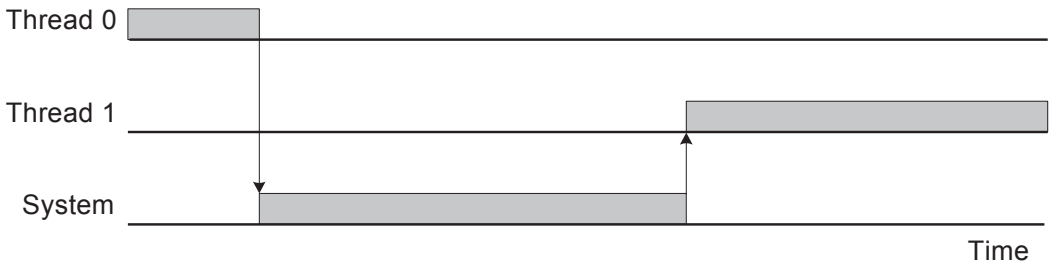

図 2.3: コンテキストスイッチ

高い Thread 1 の実行を再開する．Thread 1 の実行が完了すると，実行可能なスレッドの中で最も優先度の高い Thread 3 を実行する．

2.3 コンテキストスイッチ

スケジューラは決められた優先度を基に，優先度の高いスレッドから順番に切り替えて実行する．現在実行しているスレッドが終了した場合，または現在実行しているスレッドよりもより優先度の高いスレッドが実行可能になった場合，スケジューラは実行するスレッドを切り替える．実行するスレッドを切り替える場合，直ちに次のスレッドが実行できるのではなく，実際にはコンテキストスイッチが発生する．

図 2.3 はコンテキストスイッチにより，Thread 0 から Thread 1 へ実行が切り替わった場合の例を示している．Thread 0 の実行が終了すると，スケジューラは Thread 0 のコンテキスト（汎用レジスタ，浮動小数点レジスタ，プログラムカウンタやステータスレジスタなどの制御レジスタ）をメモリへ退避する．実行可能なスレッドの優先度を調べ，次に実行すべきスレッド（図 2.3 の場合は Thread 1）を決定し，そのスレッドのコンテキストをメモリから復帰し，スレッドの実行を開始する．

ソフトウェアでコンテキストスイッチを行う場合，まず Store 命令を用いてコンテキストを 1 つ 1 つメモリへと退避する．次に Load 命令を用いてコンテキストを 1 つ 1 つメモリから復帰する．よってソフトウェアによるコンテキストスイッチでは，大量のメモリアクセスが発生する．メモリアクセスはレイテンシが大きいため，ソフトウェアによるコンテキストスイッチではオーバーヘッドが大きい．

より高度なシステムを構築する場合，システムで実行するスレッドの数が増大すると考えられる．制御スレッドのように，より細かく制御を行おうとした場合，制御スレッドの実行周期を短くすることが考えられる．このような場合，単位時間当たりのスレッドの切り替え回数が増大する．スレッドの切り替え回数が増加すると，コンテキストスイッチに費やす時間の割り合いが増加する．

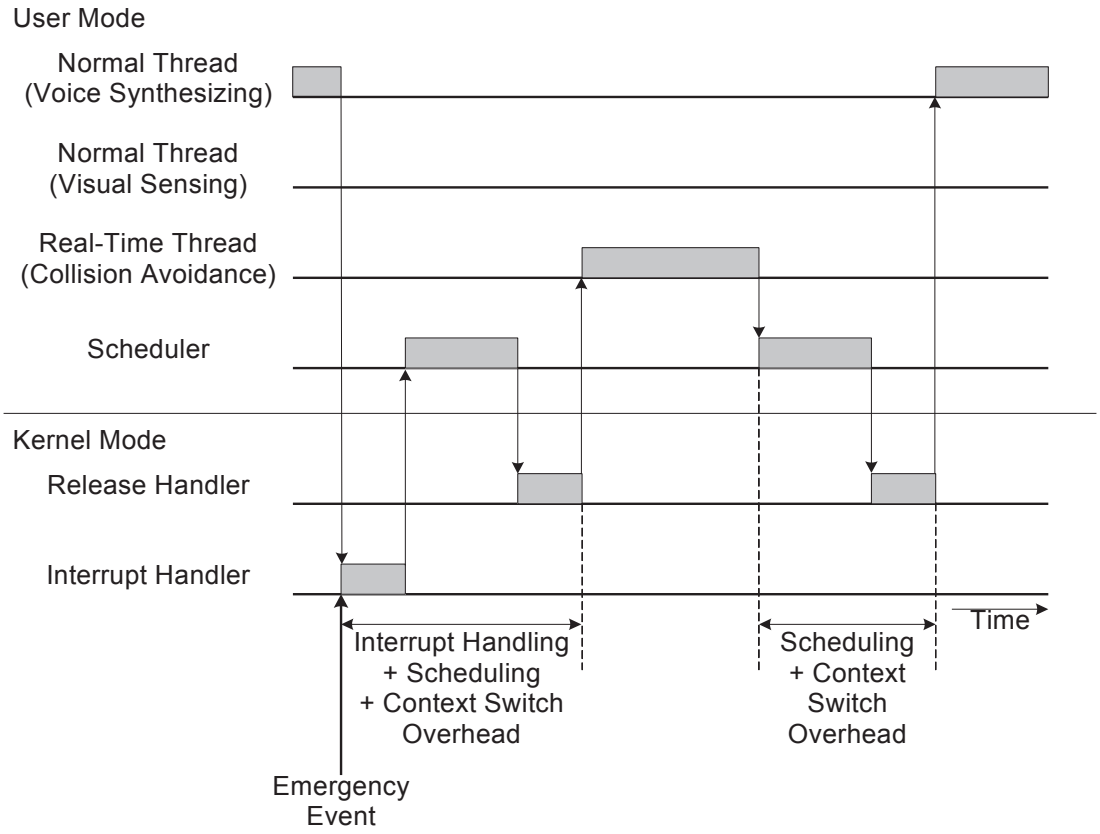

図 2.4: μ-PULSER による割り込み処理

2.4 割り込み

リアルタイムシステムでは，スレッドの起動は主に割り込みによって実現される．例えば周期スレッドの場合，一定時間毎に発生するタイマ割り込みによりスレッドが起動する．ロボットの障害物回避スレッドの場合，センサからの割り込みにより起動し，障害物を回避する．割り込みがかかってから，時間内に回避行動を行わなければロボットは障害物に激突する．このように，リアルタイムシステムでは割り込みに対して素早く応答する必要がある．

図 2.4 に μ-PULSER[矢向 他 94] による割り込み処理の例を示す．μ-PULSER はシステム自体のスレッド，リアルタイムスレッド，割り込みスレッド，継続スレッドの 4 種類のスレッドからなり，これらを必要に応じてスケジューリングすることでロボットの制御を行う．スレッドの切り替えはセンサ入力などの割り込みにより発生する．スレッドの切り替えにはスケジューリング，コンテキストスイッチのオーバーヘッドを伴なう．

例では，通常のスレッド（Voice Synthesizing）を実行中にセンサが障害物を検知した場合を示している．センサは割り込み（Emergency Event）を発生し，障害物を避けるため

に障害物回避スレッド（Collision Avoidance）に実行を切り替えている．障害物回避スレッドが終了すると，再び元のスレッドを実行している．

障害物回避スレッドはハードリアルタイム性が要求され，通常のスレッドから障害物回避スレッドに切り替わるまでの時間は可能な限り短かい方が良い．しかし，切り替えを行うためにはコンテキストスイッチを行わなければならない．よって割り込みに対する応答性を向上するためには，コンテキストスイッチにかかる時間を短縮する必要がある．

2.5　リアルタイム処理用プロセッサに必要な機能

従来の研究では，スケジューラが時間制約を基に各スレッドに優先度を決定し，優先度に従って順番にスレッドを切り替えて実行することにより，各スレッドの時間制約を守る．スレッドを切り替える場合，コンテキストスイッチが発生する．ソフトウェアによるコンテキストスイッチでは大量のメモリアクセスを行うため，オーバーヘッドが大きい．

より高度なリアルタイムシステムを構築する場合，実行するスレッド数の増加や，より短かい時間周期でスレッドを実行するといった要求が考えられる．このような場合，スレッド切り替えの回数が増加するため，コンテキストスイッチによるオーバーヘッドの割り合いが増加し，問題となる．この問題を解決するためには，より小さいオーバーヘッドでコンテキストスイッチを行う必要があると考えられる．

ソフトリアルタイム処理では，大量のデータを演算する．そのためソフトリアルタイムスレッドを実行する場合，ソフトリアルタイムスレッドが演算資源を多く占有することになる．ハードリアルタイムスレッドとソフトリアルタイムスレッドを同時に実行しようとした場合，ハードリアルタイムスレッドの優先度が高ければ，ソフトリアルタイムスレッドに十分な演算資源が割り当てられず，大量のデータを演算することができない．ソフトリアルタイムスレッドの優先度が高ければ，ソフトリアルタイムスレッドに演算資源を占有され，ハードリアルタイムスレッドを実行できず，ハードリアルタイムスレッドがデッドラインミスを起こしてしまう．このようにハードリアルタイム処理とソフトリアルタイム処理といった性質の異なる処理を同時に実行する場合，問題となる．

以上の問題を解決するために，リアルタイム処理用プロセッサでは，小さいオーバーヘッドでコンテキストスイッチを行う機能と，ソフトリアルタイム処理の大量のデータを演算する機能が必要であると考えられる．

Responsive Multithreaded（RMT）Processor[Yamasaki 05] はリアルタイム処理をハードウェアレベルから支援するためのプロセッサとして設計されている．RMT Processor のプロセッシングユニットである，RMT PU は，Simultaneous Multithreading（SMT）に優先度制御機構を持つ．優先度制御機構により，コンテキストスイッチを行わずにハード

ウェアコンテキストを優先度の高い順に切り替えて実行しつつ，SMT により，プロセッサ全体のスループットを向上している．RMT PU については 4 章で詳しく述べる．

2.6 本章のまとめ

本章では，リアルタイム処理について述べた．そしてリアルタイム処理を実現するための従来研究について述べ，従来研究の問題点からリアルタイム処理用プロセッサに必要な機能について述べた．

リアルタイム処理では，様々な時間制約を持った処理が実行される．これらの処理の時間制約を守るために，スケジューラが処理の優先度を決定し，優先度に従って処理を切り替えながら実行する．実行する処理を切り替える場合，コンテキストスイッチが発生する．ソフトウェアによるコンテキストスイッチは大量のメモリアクセスが発生するため，オーバーヘッドが大きい．システムの高機能化により，処理の切り替え回数が増加するとソフトウェアによるコンテキストスイッチでは，オーバーヘッドの割り合いが大きくなるため問題となる．

ソフトリアルタイム処理は，画像処理や音声合成などのマルチメディアアプリケーションのように大量のデータを演算する．このような処理をハードリアルタイム処理と同時に実行する場合，ソフトリアルタイム処理に演算資源が占有され，ハードリアルタイム処理の実行に影響を与える．また，ソフトリアルタイム処理よりも優先度の高い処理が実行されている場合は，十分に演算資源が割り当てられず，演算を行うことができない．

以上より，リアルタイム処理用プロセッサは，小さいオーバーヘッドでコンテキストスイッチを行うための機能と，ソフトリアルタイム処理の大量のデータを演算する機能が必要であると考えられる．

次章では，従来の研究において，コンテキストスイッチのオーバーヘッドを削減するための手法と，ソフトリアルタイム処理の大量のデータを演算するための手法について述べる．

第3章

関連研究

本章ではハードウェアによりコンテキストスイッチを削減するための手法と，ソフトリアルタイム処理の大量のデータを演算するための手法について述べる．

3.1 マルチスレッドアーキテクチャ

ソフトウェアによるコンテキストスイッチでは，コンテキストを1つ1つLoad / Store命令を用いてメモリに退避，復帰を行うため，オーバーヘッドが大きいといった問題がある．このオーバーヘッドを削減するために，マルチスレッドアーキテクチャを使用する手法がある．

本来マルチスレッドアーキテクチャは，プロセッサ内に複数のスレッドを保持し，これらのスレッドから依存関係のない命令を実行することによる，スレッドレベルの並列性を利用して，プロセッサ全体のスループットを向上する手法である．マルチスレッドアーキテクチャはスレッドの切り替え粒度によって以下の3つがある．

- 粗粒度マルチスレッディング
- 細粒度マルチスレッヂング
- Simultaneous Multithreading

図3.1に粗粒度マルチスレッディングの概念を示す．粗粒度マルチスレッディングは数クロック毎に実行するスレッドを切り替える方式である．キャッシュミスなどのレイテンシの長い命令が実行された場合に，他のスレッドに切り替える．切り替えられたスレッドの命令はレイテンシの長い命令とは依存関係がないため，直ちに演算を行うことができる．その結果，プロセッサ全体としてのスループットが向上する．粗粒度マルチスレッディングは既存のプロセッサを大きく変更せずに採用することができるが，数クロック毎にスレッドを切り替えるため，データ依存性のように短かいレイテンシの命令を隠蔽することができない．

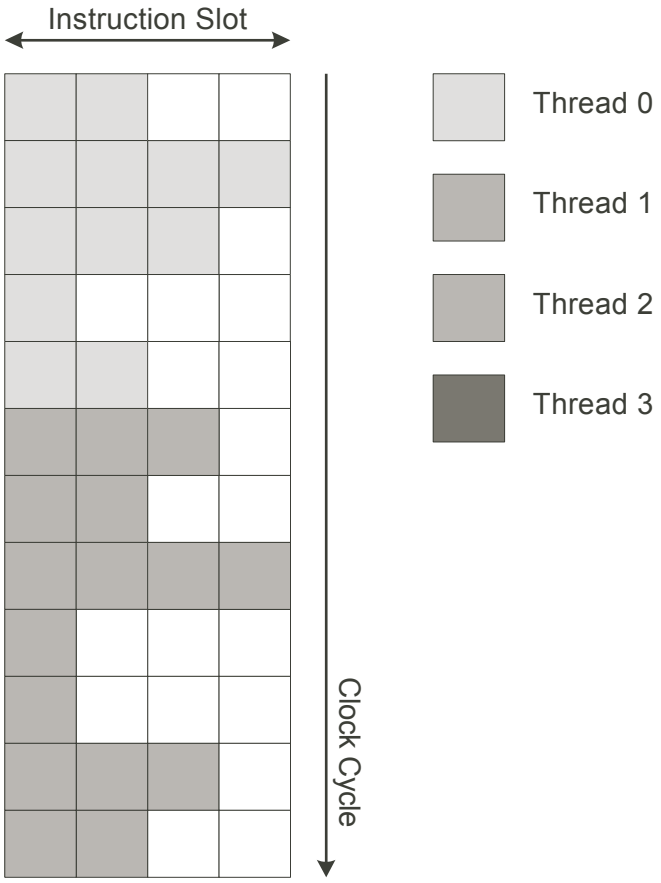

図 3.1: 粗粒度マルチスレッディングの概念

図 3.2 に細粒度マルチスレッディングの概念を示す．細粒度マルチスレッディングは 1 クロック毎にスレッドを切り替える方式である．毎クロック異なるスレッドから命令を実行することができるので，粗粒度マルチスレッディングと異なり，データ依存性のような短かいレイテンシを隠蔽することができる．また，各スレッドは数クロック毎に実行されるため，分岐命令による投機実行を減らすことができ，分岐予測ミスによるペナルティを低減することができる．しかし，同一クロックで実行できる命令は一つのスレッドからのみであるため，依存関係により同一クロックで一緒に実行できない命令がある場合，命令スロットに空きが生じる．

図 3.3 に Simultaneous Multithreading（SMT）[Tullsen *et al.* 95][Eggers *et al.* 97] の概念を示す．細粒度マルチスレッディングと異なり，SMT では 1 クロック毎に複数のスレッドから命令を実行することができる．細粒度マルチスレッディングの場合には空いてしまった命令スロットを，SMT では別のスレッドの命令で埋めることができるため，スループットをさらに向上することができる．しかし，命令スロットに対する命令の割り当て方法が複雑になるため，ハードウェアが複雑化する．

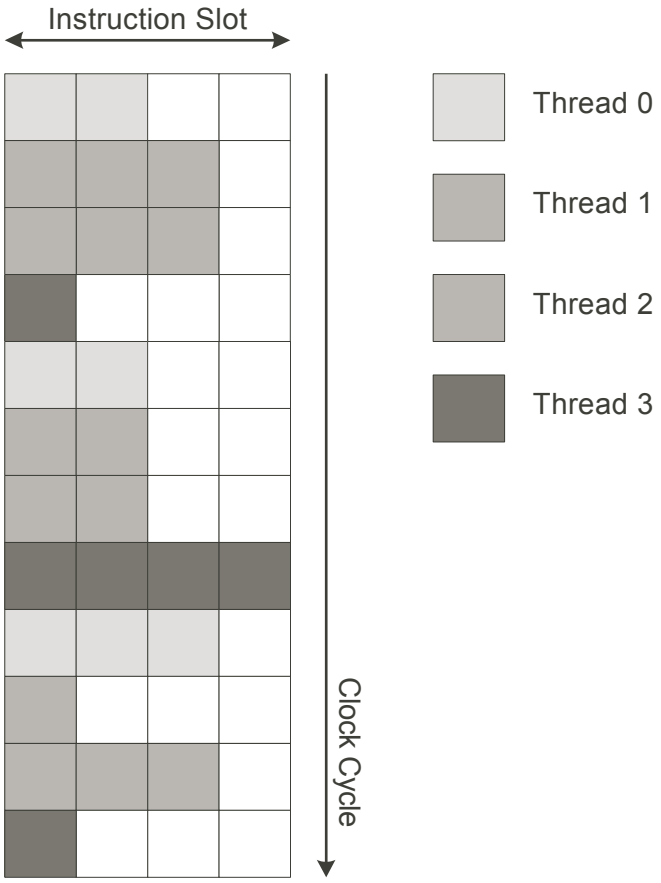

図 3.2: 細粒度マルチスレッディングの概念

リアルタイム処理においてマルチスレッドアーキテクチャを用いる場合，マルチスレッドアーキテクチャがスレッドレベルの並列性を利用するために，複数のコンテキストをプロセッサ内に保持することに注目している．プロセッサ内で保持しているコンテキスト（ハードウェアコンテキスト）は，マルチスレッドアーキテクチャにより適宜切り替えられて実行される．プロセッサ内に保持されているスレッド同士を切り替える場合は，コンテキストスイッチが発生しないため，マルチスレッドアーキテクチャを用いないプロセッサに比べてコンテキストスイッチのオーバーヘッドを削減することができる．

Komodo Microcontroller[Brinkschulte *et al.* 99]はリアルタイム処理用に設計されたMultithreaded Java Processor Coreである．マルチスレッドアーキテクチャを利用することにより，割り込み発生時のコンテキストスイッチのオーバーヘッドを0クロックとすることで，外部イベントに対する割り込み応答性を向上している．

図3.4にKomodo Microcontrollerのアーキテクチャを示す．プロセッサ内に4つのコンテキストを保持する．各スレッドのプログラムカウンタに従って命令をフェッチし，スレッド毎に割り当てられたInstruction Window（IW）にフェッチし命令を格納する．

図 3.3: SMT の概念

各 IW はカウンタや優先度，スレッドの実行状態といった情報を保持しており，Priority Manager はそれらの情報に従って，どの IW から命令を発行するか決定する．Priority Manager にはリアルタイムシステムにおけるスケジューリングポリシがハードウェアで複数実装されている．実装されているポリシは Fixed Priority Preemptive（FPP），EDF，Least Laxity First（LLF），Guaranteed Percentage（GP）[Kreuzinger *et al.* 00]がある．これらポリシの中から 1 つを選択して使用する．

Komodo Microcontroller では Signal Unit が，外部割り込みに対して各 IW を制御する．例えば，PC1 を使用して，あるスレッドが実行しているときに割り込みが発生した場合，マルチスレッディングにより PC2 に切り替えて割り込み処理を行うことにより，コンテキストスイッチを行わずに 0 クロックで割り込み処理を行うことができる．

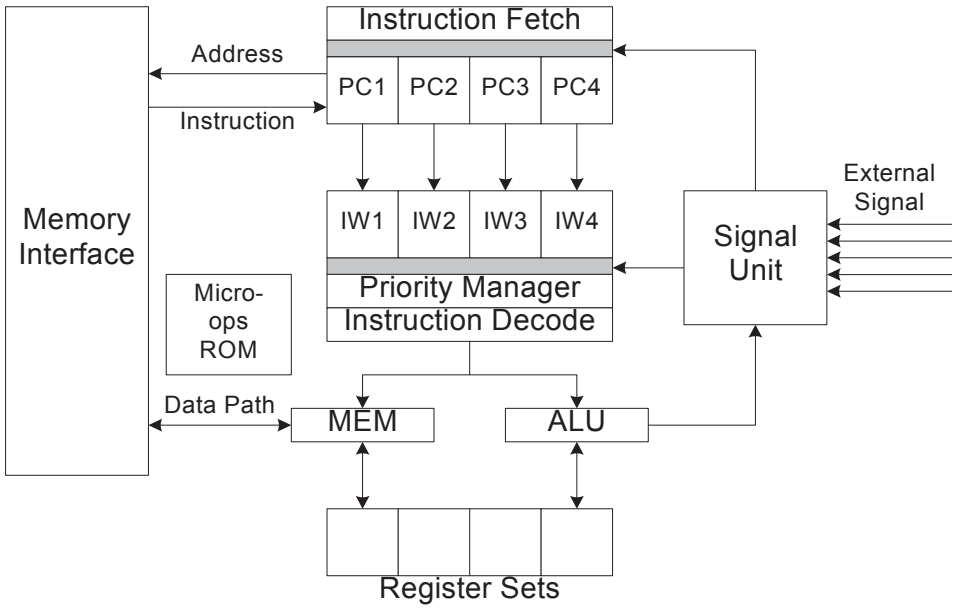

図 3.4: Komodo Microcontroller

3.2 マルチメディア処理用アーキテクチャ

ソフトリアルタイム処理では大量のデータを演算する．ソフトリアルタイム処理で扱うデータはデータ並列性が高いという特徴がある．従来研究ではこの特徴を利用して演算性能を向上している．データ並列性を利用した演算には，SIMD 演算とベクトル演算がある．

3.2.1 SIMD 演算

Single Instruction Multiple Data（SIMD）演算は 1 つの命令で複数のデータに対して演算を行うことにより，スループットを向上する演算方法である．Ultra SPARC，Pentium など多くの汎用プロセッサが SIMD 演算ユニットを実装している．

MMX

マルチメディアアプリケーションはデータ並列性が高いという特徴がある．また，マルチメディアアプリケーションで扱うデータの多くは，32bit よりも小さいビット幅であらわされる．例えば画像の色は 8bit であらわされ，オーディオデータは 16bit であらわされる．このようなデータを 32bit の汎用レジスタで演算すると，上位ビットが使用されずに無駄になる．

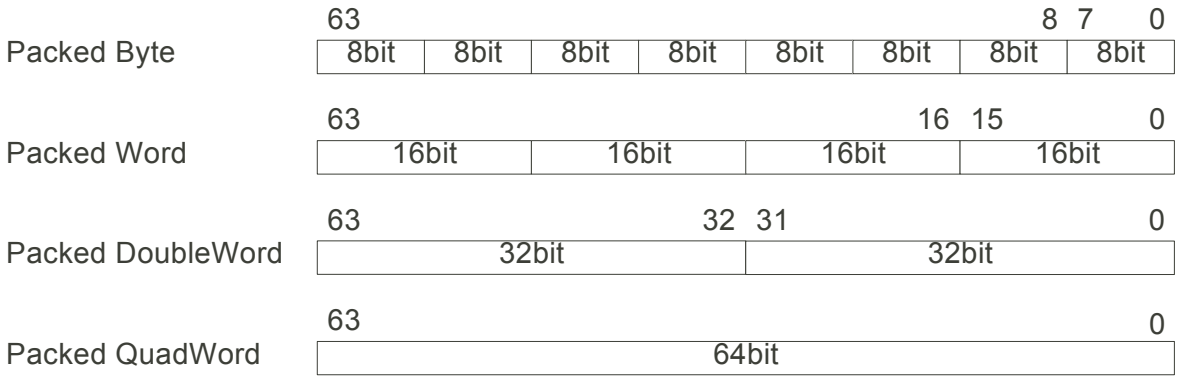

図 3.5: MMX のデータタイプ

図 3.6: MMX 命令の例

MMX[Peleg *et al.* 97] では独立した複数の 8bit や，16bit のデータを複数集め，それを 64bit の浮動小数点レジスタに詰め込む．図 3.5 に MMX で定義された 4 つのデータタイプを示す.

- Packed Byte

  64bit のレジスタに 8 つの 8bit データを詰める

- Packed Word

  64bit のレジスタに 4 つの 16bit データを詰める

- Packed Doubleword

  64bit のレジスタに 2 つの 32bit データを詰める

- Packed Quadword

64bit のレジスタに 1 つの 64bit データを詰める

これらのデータを図 3.6 に示すように専用の演算器を用いて並列演算を行なう．図 3.6 では，4 つの 16bit データを並列に加算している．SIMD 演算を行わない場合，4 つのデータを加算するためには 4 クロックかかる．しかし SIMD 演算を行うと 1 クロックで 4 つのデータを加算するため，スループットを向上することができる．MMX では，既存のレジスタを用いて演算を行うため，回路規模の増加を抑えながら，高い演算性能を達成している．

このように 1 つのレジスタに複数のデータを格納して演算する機構は少ないコストで高い演算性能が得られる．このような機構を備えたプロセッサは MMX 以外に SUN の Visual Instruction Set（VIS）[Tremblay *et al.* 96]，MIPS の Mips Digital Media Extension（MDMX）[MIPS Technologies, Inc.97]，Motorola の Altivec[Diefendorff *et al.* 00] などがある．

SH-5

SH-5[Biswas *et al.* 00] は 64bit アーキテクチャの組み込み用プロセッサである．整数SIMD 演算命令とマルチメディア処理のアルゴリズムに適した命令を追加することにより，マルチメディアアプリケーションの処理性能を向上している．

SH-5 の SIMD 演算は 64bit の汎用整数レジスタを MMX と同様に 32bit × 2，16bit × 4，8bit × 8 の領域に分割し，各領域ごとに算術演算，論理演算，シフトを行う．

また，マルチメディアアプリケーション用に新たに積和演算命令と Sum of Absolute Difference 命令が追加されている．図 3.7 に積和演算命令の動作を示す．積和演算命令は 4 つの 16bit 乗算，3 つの加算，1 つのアキュムレート演算を 1 クロックで行う．この命令を用いることにより，長さ 4 のベクトルの内積演算を 1 クロックで行うことができる．

Sum of Absolute Difference 命令は図 3.8 に示すように，各領域の差分を合計する命令である．この命令は例えば MPEG エンコーディングアルゴリズムで活用することができる．

そのため 3DNow!では逆数や平方根の逆数を求める命令を持つ．

Streaming SIMD Extensions

Streaming SIMD Extensions（SSE）[Raman *et al.* 00] は MMX を拡張して，グラフィックス処理能力を向上させている．

グラフィックス処理アルゴリズムは単精度浮動小数点演算が多く用いられる．よって SSE では単精度浮動小数点 SIMD 演算の機能を追加している．さらにプリフェッチやキャッシュ階層の制御など 70 命令が追加されている．

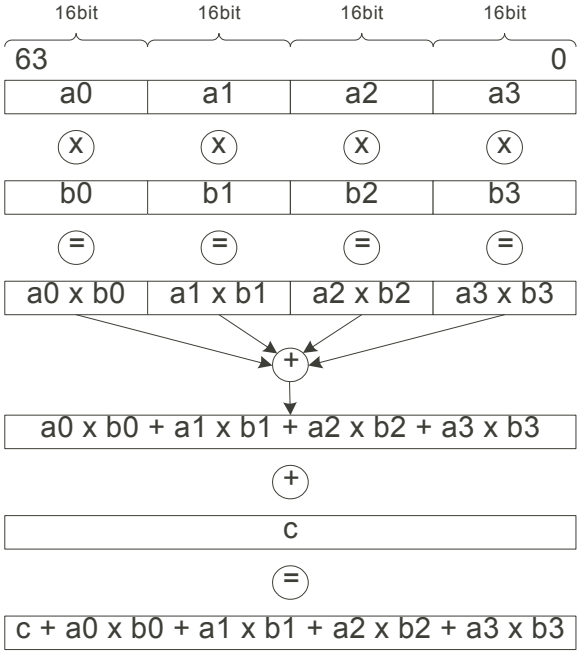

図 3.7: SH-5 の積和演算命令

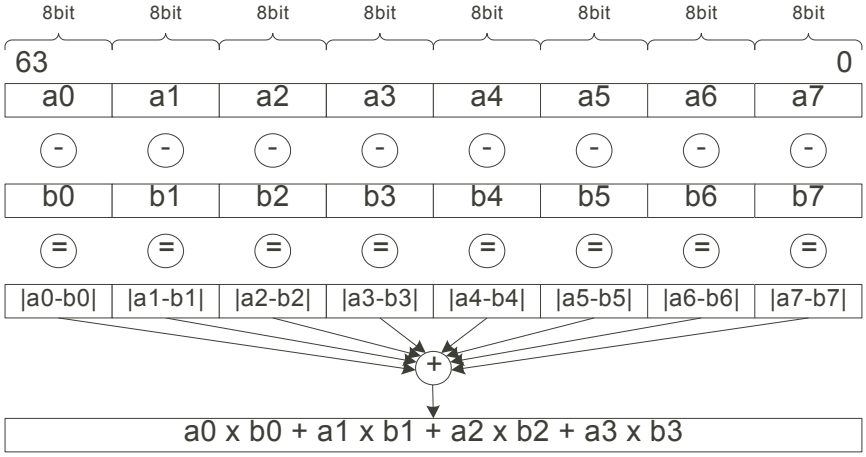

図 3.8: SH-5 の Sum of Absolute Difference 命令

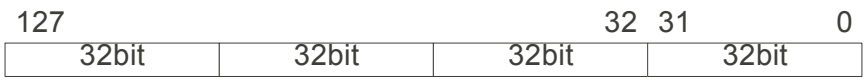

図 3.9: SSE のデータタイプ

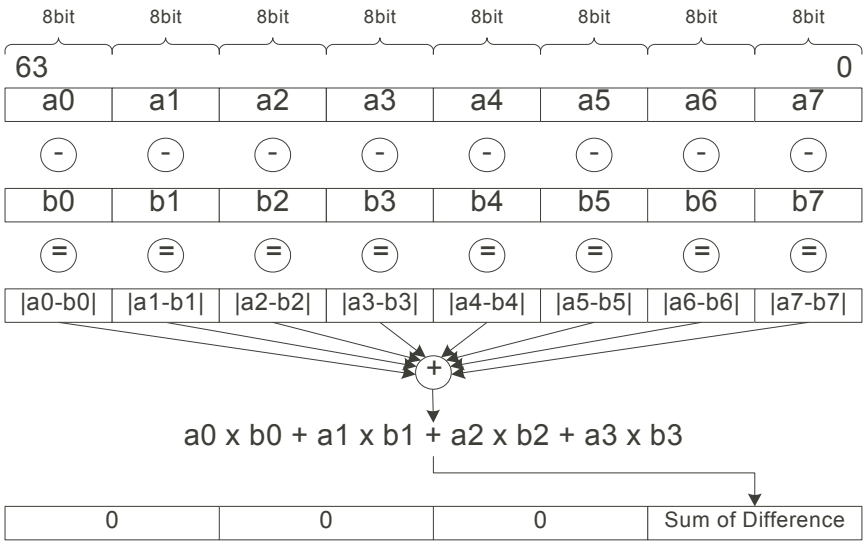

図 3.10: PSADBW 命令

図 3.9 に SSE で新たに追加されたデータタイプを示す．図 3.9 に示すように 128bit を 4 つのフィールドに分け，32bit の単精度浮動小数点 SIMD 演算を行う．128bit のデータを保持するために，SSE では MMX レジスタとは別に 8 つの 128bit XMM レジスタを用意している．

SSE ではマルチメディア処理のアルゴリズムを効率良く実行するために，いくつかの新しい命令が追加されている．例えば PAVG 命令は各フィールドの平均を計算する命令である．これは画像の画素値の平均を求めるために用いられる．PSADBW 命令は 8bit ごとの差分の合計を計算する．図 3.10 に PSADBW 命令の動作を示す．この演算を MMX 命令で行う場合，7 個分の命令に相当する．

通常 load 命令がキャッシュミスを起こした場合，メモリアクセスに時間がかかり，命令の終了が遅れる．このとき Reorder Buffer などの制限によりパイプラインがストールする．SSE で追加されたプリフェッチ命令はキャッシュミスを起こしても即座に終了する．そのためパイプラインをストールさせずにデータを取ってくることが可能となる．

ストリーミングアプリケーションではデータは頻繁に使われるもの（Temporal Data）と一度しか使われないもの（Nontemporal Data）に分けることができる．通常，データは必要な時にメモリから L2 キャッシュに取ってこられる．しかし SSE ではプログラマがキャッシュを制御することにより，Temporal Data を L2 キャッシュに，Nontemporal Data をメモリに配置することができる．頻繁に使うデータを常にキャッシュの中に置くことができ，キャッシュミス率を改善している．

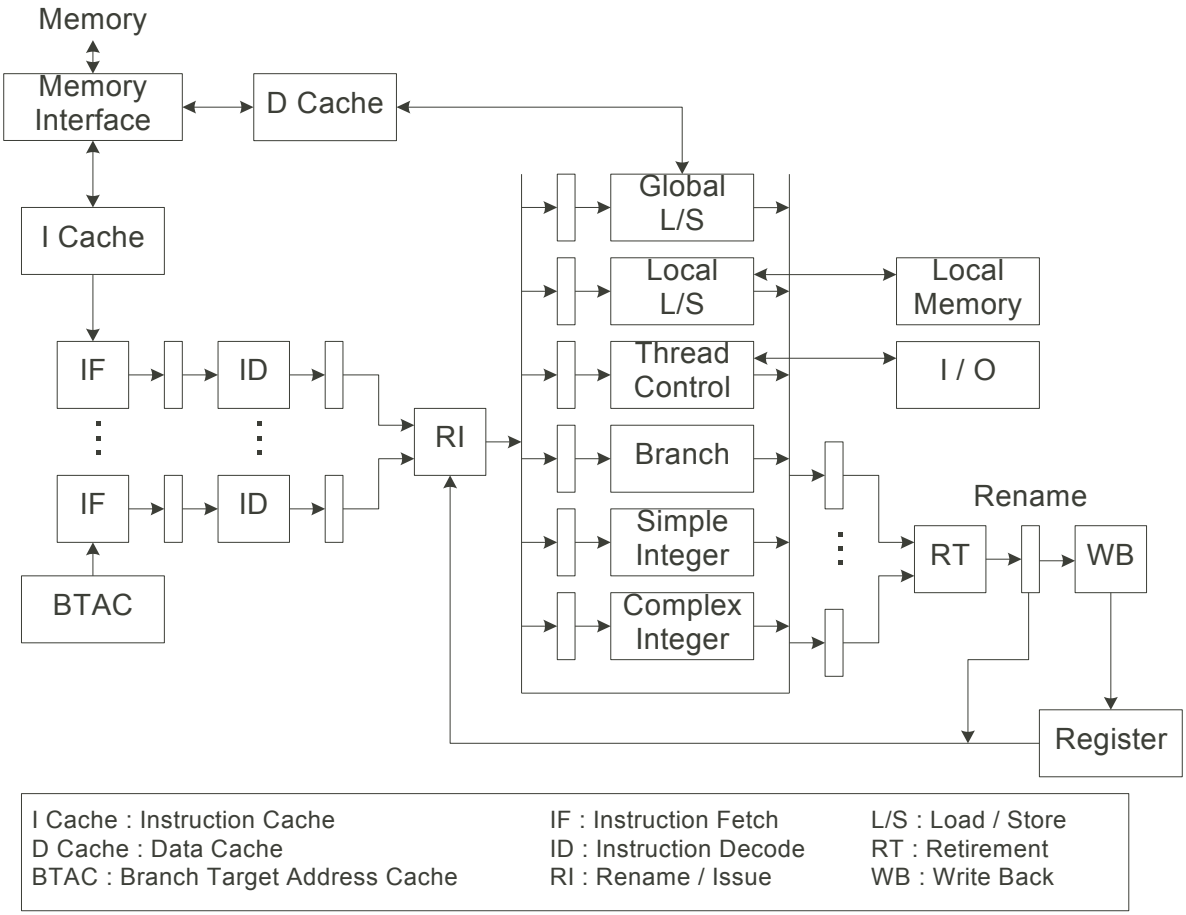

図 3.11: マルチメディア処理用 SMT Processor のブロック図

マルチメディア処理用 Simultaneous Multithreaded Processor

MPEG-2 復号アルゴリズムでは，逆量子化，逆離散コサイン変換，動き補償による動きの補完を各 8×8 ブロックで並列に処理することができる．

マルチメディア処理用 Simultaneous Multithreaded（SMT）Processor[Oehring *et al.* 99] では，この並列性を利用して，複数のスレッドを実行することにより，MPEG の復号を高速に処理する．図 3.11 にマルチメディア処理用 SMT Processor のブロック図を示す．

このプロセッサでは Integer Unit が SIMD 命令，飽和演算といった処理を実行できるように拡張を行っている．スレッドの開始，停止，同期，I/O 操作は Thread Control Unit が行う．メモリアクセスを高速化するために Local Load / Store Unit とオンチップの Local RAM を持つ．

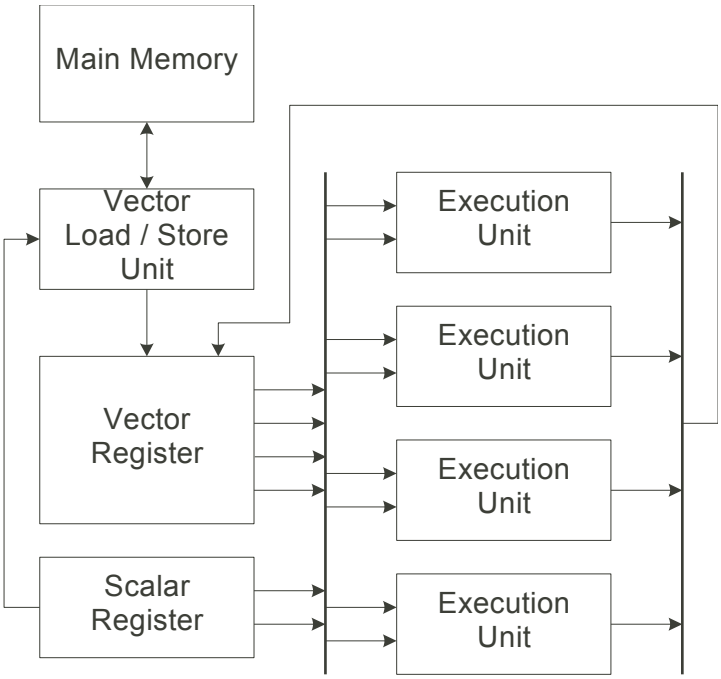

図 3.12: ベクトル演算

3.2.2 ベクトル演算

データ並列性を利用した演算方法として，SIMD 演算の他にベクトル演算がある．ベクトル演算はもともとスーパーコンピュータによって一般化されたアーキテクチャである [Hennessy *et al.* 95]．1 つの命令で複数のデータを処理する点は SIMD 演算と同じである．SIMD 演算では 1 クロックで全てのデータを並列演算する．一方，ベクトル演算ではパイプライン化された演算ユニットを用いて，1 クロックに数個のベクトル要素を演算し，複数クロックをかけて全てのデータを演算する点が異なる．

図 3.12 にベクトル演算の例を示す．ベクトル演算ではベクトル要素をベクトルレジスタと呼ばれる巨大なレジスタに格納し，それをパイプライン的に演算する．

ベクトル演算ではメモリとベクトルレジスタの間でデータ転送を行う場合，Load / Store 命令によりベクトル要素を一度に転送する．メモリ間の転送を一斉に行うことにより，データ転送のスループットを向上している．

SH-4

SH-4[Arakawa *et al.* 98] はホームビデオゲーム，Handheld PC を主な用途にした組み込み用プロセッサである．ホームビデオゲームでは強力な 3D グラフィックス演算性能が要求される．SH-4 は SIMD 演算と異なり，ベクトル演算命令を持つ．ベクトル演算命令はベク

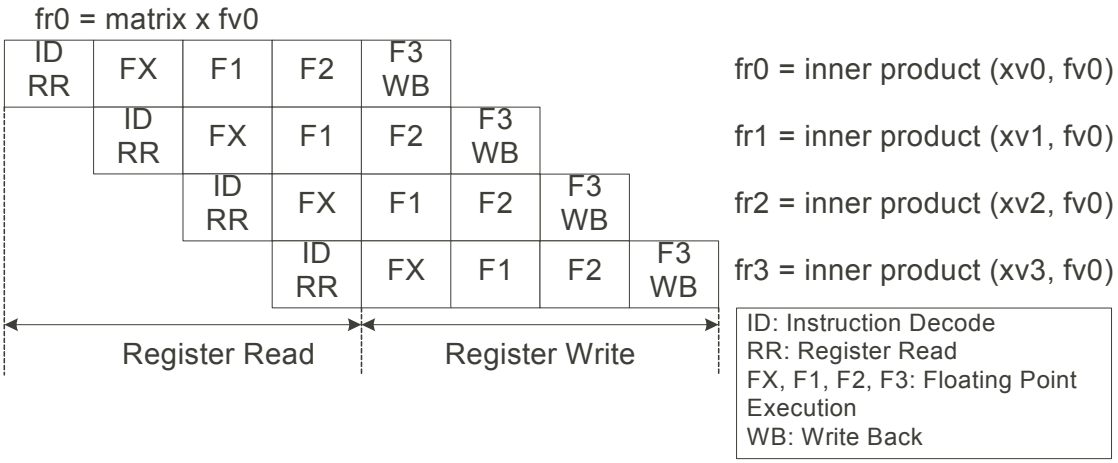

図 3.13: SH-4 における行列変換の演算

トル長 4 の浮動小数点内積演算命令と 4 × 4 行列の浮動小数点行列変換命令を持つ．内積演算命令は 3D グラフィックスにおける，ベクトルの正規化，強度計算，3 次元処理での表面判定などに使われる．4 × 4 行列変換命令は 3 次元画像処理において，座標変換，変換行列の計算に使用することができる．

内積演算は 16 個の浮動小数点レジスタを 4 要素，ベクトル長 4 の行列として扱う．浮動小数点ベクトル変換命令と 4 回の内積演算で 4 × 4 の配列とベクトル長 4 のベクトルの積（行列変換）を求めることができる．図 3.13 に行列変換のパイプラインを示す．結果の書き戻しが行われる前にレジスタの読み込みが完了するため，ワークレジスタを必要とせず，効率良く演算を行なうことができる．

Emotion Engine

Emotion Engine[Oka *et al.* 99][Suzuoki *et al.* 99] はホームエンターテインメント用に設計されたプロセッサで，MIPS ベースのコアと 2 つの Vector Processing Unit（VPU）により構成される [Kunimatsu *et al.* 00]．VPU のベクトル演算機構により，コンシューマビデオゲームなどに必要な三次元幾何学演算を高速に処理する．

図 3.14 に Emotion Engine のブロック図を示す．Emotion Engine は 2 つの VPU（VPU0 と VPU1）を持つ．2 つの VPU はキャッシュサイズと VPU1 に余分に FMAC，FDIV がある点以外は同じアーキテクチャである．しかし 2 つの VPU の役割は明確に決められている．

VPU0 は MIPS コアと直結するバスを持ち，MIPS コアのコプロセッサとして働く．そのため VPU0 では人の動きや物理シミュレーションなどの複雑な演算を行う．一方，VPU1 はデータを描画エンジン（Graphics Synthesizer）へ直接送ることができるパスを持つ．よっ

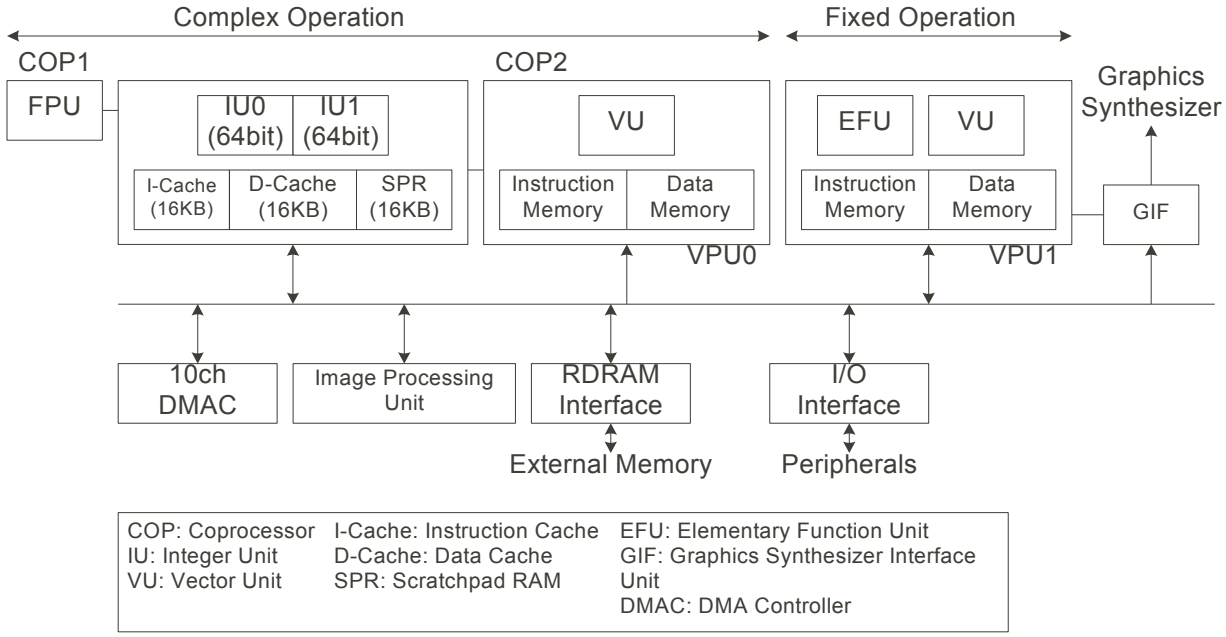

図 3.14: Emotion Engine のブロック図

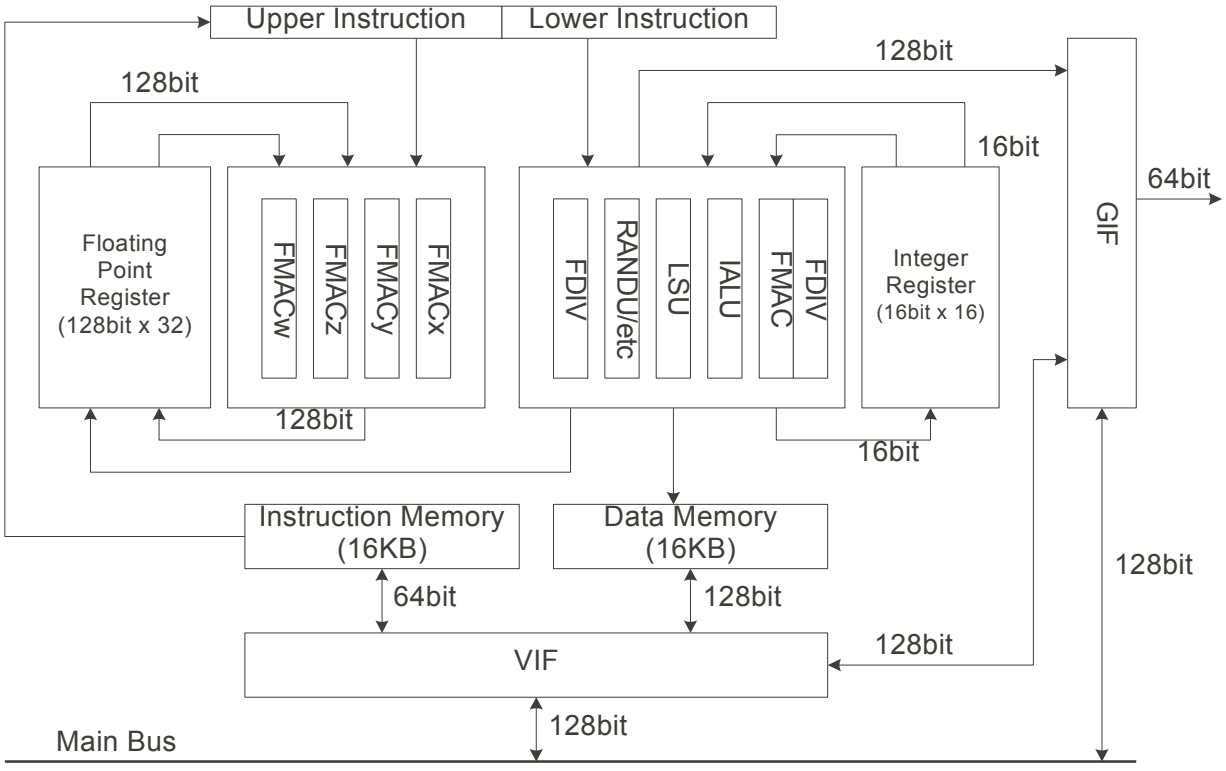

図 3.15: VPU1 のブロック図

て VPU1 は背景描画などの単純でデータが大量に発生する演算を処理する.

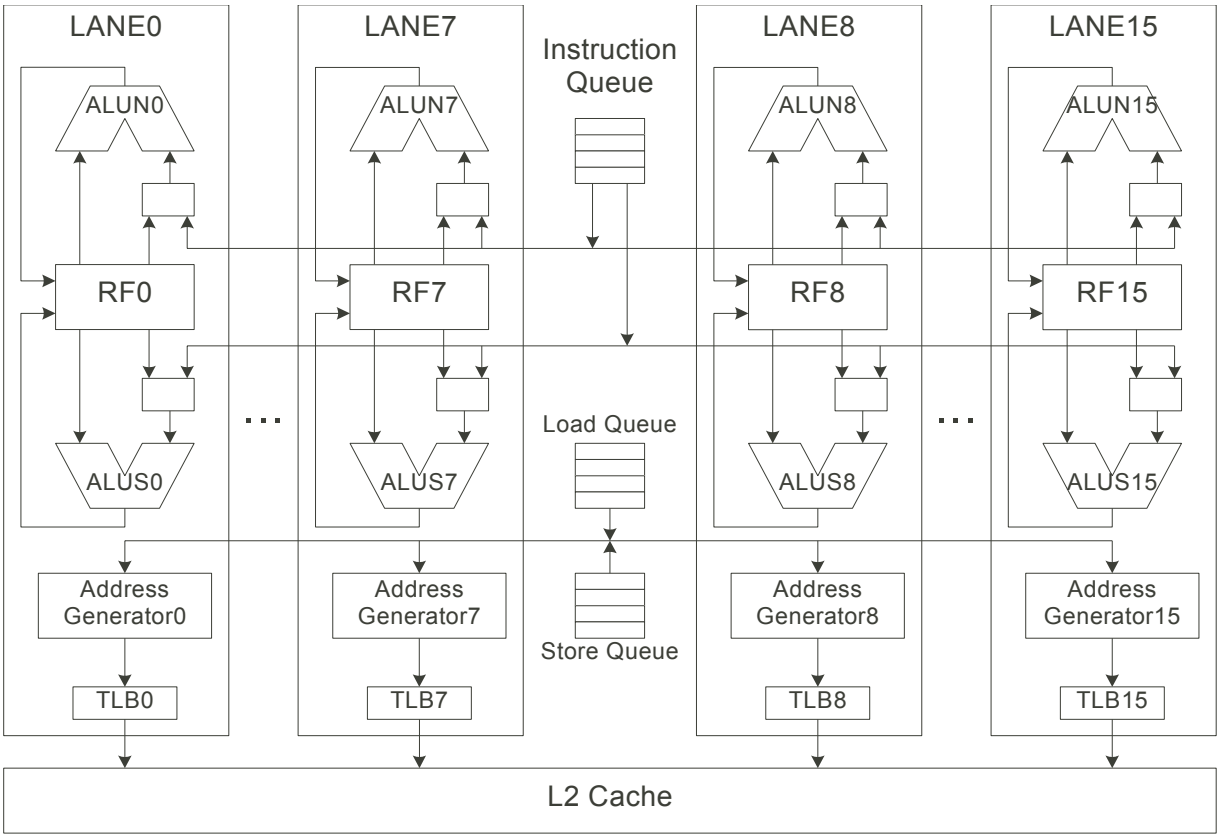

図 3.16: Tarantula のベクトルユニット

図 3.15 に VPU1 のブロック図を示す．VPU の命令は 64bit の VLIW で 32bit づつ，Upper Instruction と Lower Instruction に分けられる．Upper Instruction は 4 つの 32bit 単精度浮動小数点ベクトル演算を行い，Lower Instruction は浮動小数点除算，Load / Store，整数演算を行う．ベクトル演算にはベクトル同士を演算する Vector Type と，ベクトルとスカラの演算を行う Broadcast Type がある．

このアーキテクチャにより，VPU は 7 サイクルで 1 つの三次元座標変換を処理する．

Tarantula

Trantula[Espasa *et al.* 02] は EV8（Alpha 21464）に浮動小数点ベクトルユニットを追加したプロセッサである．ベクトルユニットは図 3.16 に示すように 16 レーンで構成される．

ベクトルユニットは 32 個の演算器を持ち，64bit でベクトル長 128 のベクトルレジスタが 32 個ある．ALUN0 から ALUN15，ALUS0 から ALUS15 は同じベクトルレジスタの異なる要素に対して並列演算を行う．ベクトルユニットの命令ポートは 2 つあり，簡単なスケジューラで 32 個の演算器を制御する．

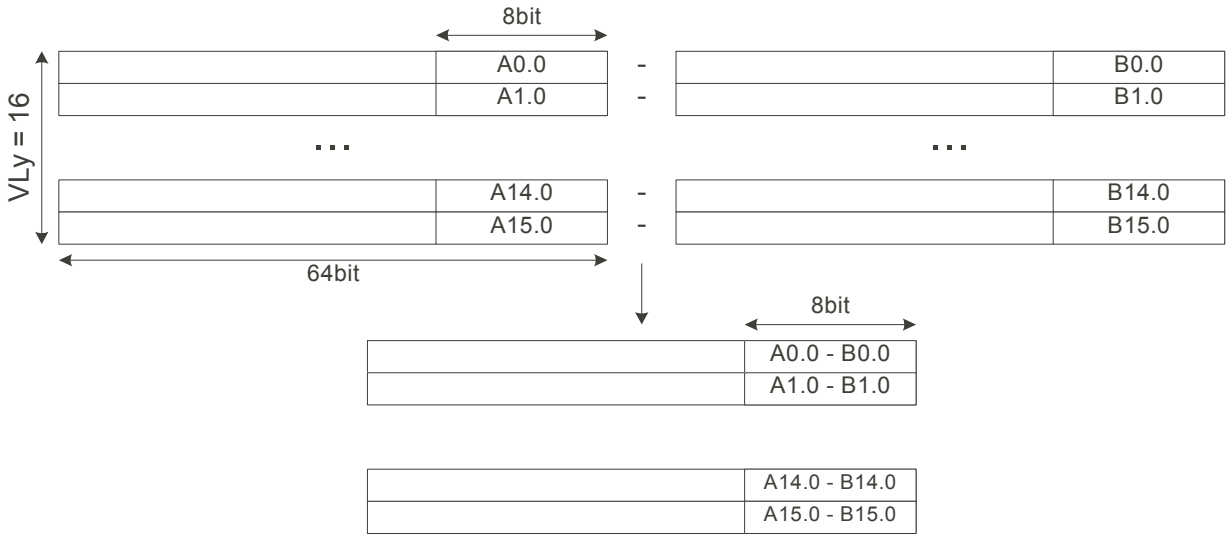

図 3.17: 従来のベクトル演算

マルチメディアアプリケーションでは大量のデータを扱う．L1データキャッシュではデータを格納するには小さすぎる．また，EV8 の L2 キャッシュは物理的に 16 個のバンクが並んだ構成になっているため，16 ポートのキャッシュとみなすことができる．そのためベクトルユニットは L2 キャッシュに直接接続している．L2 キャッシュの 16 ポートとレーン毎の Address Generator と TLB によって，同時に 16 個のメモリアクセスを L2 キャッシュに対して行うことが可能となっている．

Matrix Oriented Multimedia Extention

Matrix Oriented Multimedia（MOM）Extension[Corbal *et al.* 99b][Corbal *et al.* 99a] は SIMD に似た Instruction Set Architecture（ISA）と Vector ISA の両方の特徴を持つ，Matrix Oriented の ISA である．MOM はマルチメディアアプリケーションで用いられる小さな行列構造に適しており，マルチメディアアプリケーションの性能を向上させることができる．

図 3.17 は従来のベクトル演算を示している．従来のベクトル演算方式では，1 つのレジスタに 1 つのデータを格納する．図 3.17 のように 8bit のデータを格納する場合，1 つのレジスタの下位部分に 8bit のデータを格納する．そのため行列を演算する場合，ベクトルレジスタには 1 列分のデータしか保持することができない．図 3.18 は従来の SIMD 演算を示している．SIMD 演算の場合は 1 つのレジスタに複数のデータを格納する．図 3.18 のように 8bit のデータを格納する場合，64bit レジスタに 8 つのデータを格納する．そのため行列を演算する場合，1 列分のデータを保持することができる．しかし SIMD 演算では 1 つ分の 64bit レジスタしか演算できないため，行数分だけ演算を繰り返すことになる．このよ

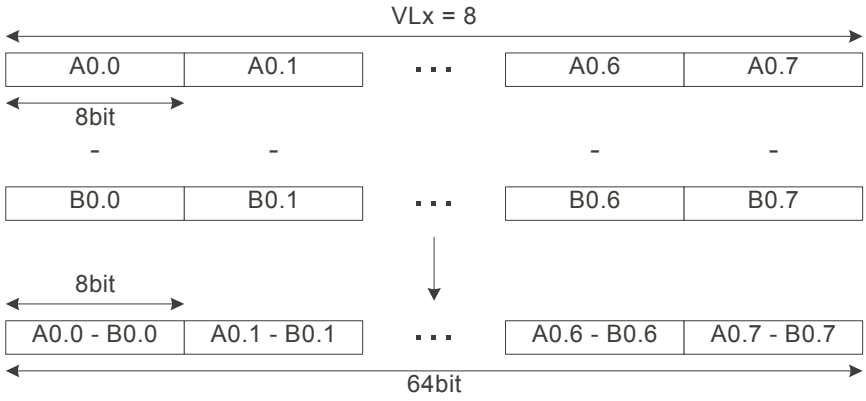

図 3.18: 従来の SIMD 演算

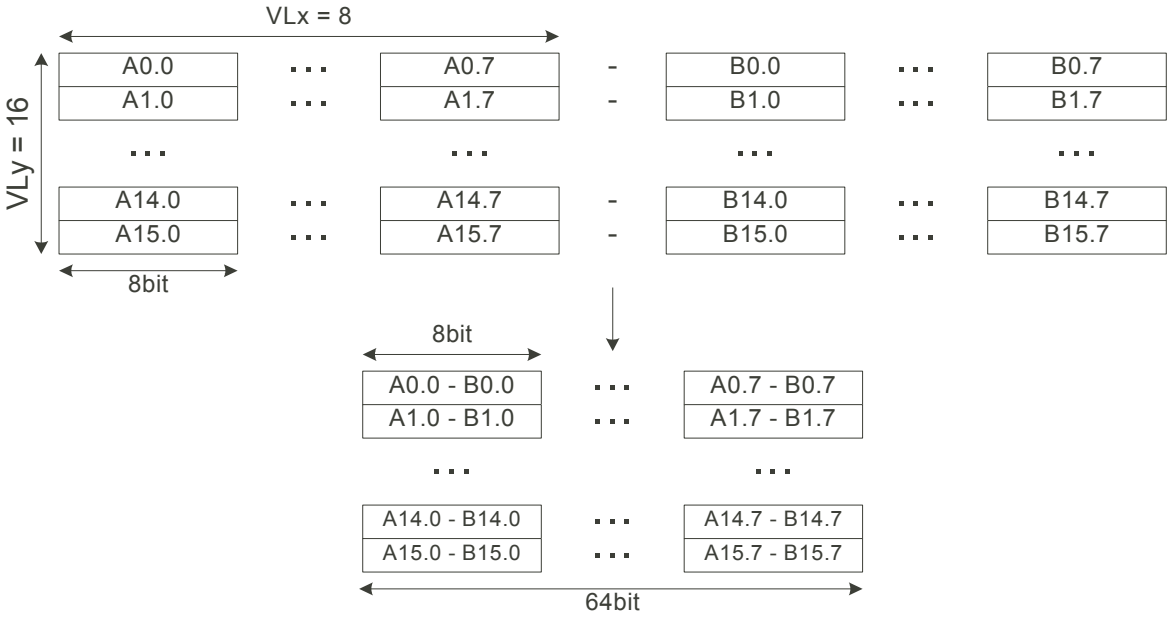

図 3.19: Matorix Oriented ISA

うに従来のベクトル演算や SIMD 演算では行列演算を行う場合，行と列の両方の情報を保持することができない．

図 3.19 に MOM の ISA を示す．MOM では各ベクトルレジスタを複数の領域に分割し，1 つの 64bit レジスタに複数のデータを保持する．これにより MOM では行列構造をそのままベクトルレジスタに保持することが可能となる．このデータ構造を用いて，そのまま行列演算を行うことができる．

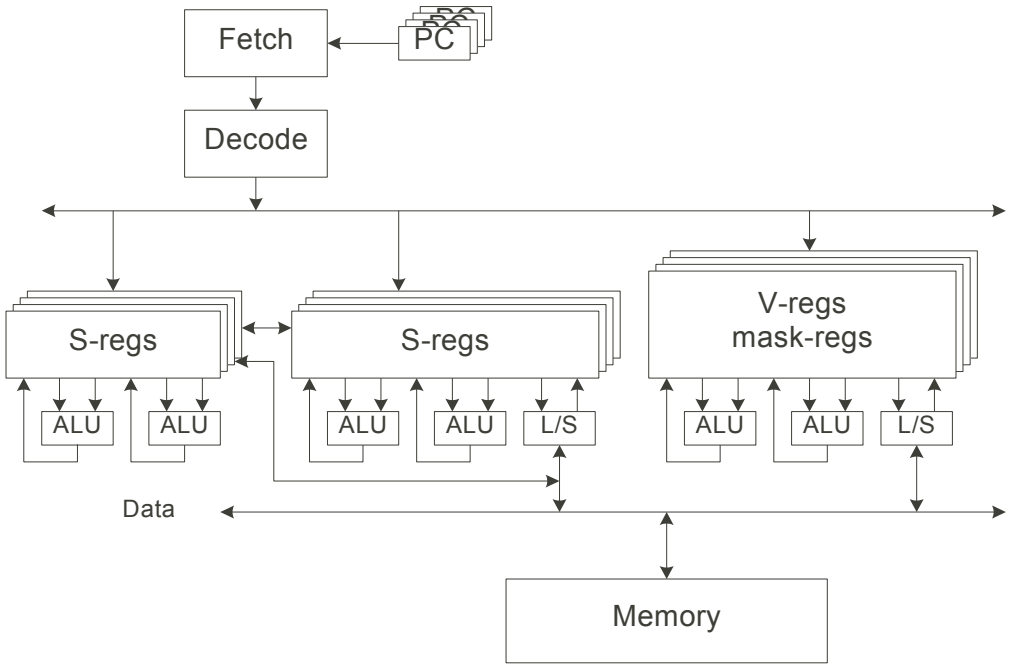

図 3.20: Multithreaded Vector Architecture のブロック図

Multithreaded Vector Architecture

Multithreaded Vector Architecture 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では複数のスレッドを同時に実行することができるため，メモリポートの使用率（メモリアクセス率）を向上することができ，ベクトル演算の性能を向上している．

図 3.20 に Multithreaded Vector Architecture のブロック図を示す．ベクトルユニットは Convex C3400 をベースとしている．マルチスレッド版ではハードウェアコンテキストを複数持つために，それぞれのレジスタ（A-register，S-register，V-register）を複数（4 個まで）持つ．各サイクルで Decode unit は 1 つのスレッドからの命令をデコードする．命令がディスパッチ可能なら，その命令を対応する演算ユニットに送る．ディスパッチができなければ，スイッチロジックが別のスレッドを選択し，次のサイクルでそのスレッドからの命令がデコードされる．このベクトル演算器は Out-of-Order 実行や Simultaneous といった命令発行機構を持たない．1 サイクルで最大 1 命令がフェッチされ，最大 1 命令が演算ユニットに送られる．

スカラレジスタ（A-register，S-register）は 1 コンテキストに 8 つあればよいため，32 個のレジスタですむ．しかしベクトルレジスタ（V-register）は $8 \times 128 = 1024$ の 64bit レジスタが必要になり，回路規模だけでなく，レジスタと演算ユニット間の read / write crossbar

のコストが問題となる．そこで read / write crossbar を二重化することによりこの問題を解決している．

3.3 本章のまとめ

本章では，従来研究において，コンテキストスイッチのオーバーヘッドを削減するための手法と，ソフトリアルタイム処理の大量のデータを演算するための手法について述べた．

コンテキストスイッチのオーバーヘッドを削減するための手法として，マルチスレッドアーキテクチャを利用する方法がある．Komodo Microcontroller ではマルチスレッドアーキテクチャを利用し，プロセッサ内に複数のスレッドを保持する．これらのスレッドはマルチスレッディングによりハードウェアで切り替えられる．ソフトウェアによるコンテキストスイッチを行わずにスレッドを切り替えることができるため，割り込みに対する応答性を向上している．

しかし，マルチスレッドアーキテクチャではコンテキストスイッチを行わずに切り替えることができるスレッドの数はハードウェアコンテキストの数だけである．Komodo Microcontroller の場合，4 つのスレッドをプロセッサ内に保持することができるが，それ以上のスレッドを実行する場合，コンテキストスイッチによってスレッドを入れ替える必要がある．この場合，従来通りソフトウェアによってコンテキストスイッチを行うことになるため，オーバーヘッドが生じる．また，Komodo Microcontroller は単純なシングルスカラのパイプラインであるため，ソフトリアルタイム処理のような大量のデータを演算する場合，演算性能に問題がある．

ソフトリアルタイム処理の大量のデータを演算するための手法として，データ並列性を利用した演算手法が提案されている．また，多くのプロセッサではソフトリアルタイム処理で用いられるアルゴリズムの中で頻繁に使用される演算（積和演算や Sum of Absolute 演算など）を高速に計算するための命令を新たに追加している．専用命令を用いることにより，演算性能を向上している．

しかしデータ並列性を利用した手法では，演算性能を向上させるだけであるため，ハードリアルタイム処理とソフトリアルタイム処理の両方を実行する場合は，ソフトウェアによりコンテキストスイッチを行わなければならないためオーバーヘッドが生じる．

以上ように，従来の手法ではハードリアルタイム処理とソフトリアルタイム処理の両方を同時に実行する場合に問題となる．本研究ではハードリアルタイム処理とソフトリアルタイム処理の両方を扱うためのプロセッサを設計する．次章ではリアルタイム処理用プロセッサである Responsive Multithreaded Processor について述べ，本研究で解決する課題について述べる．

第4章

Responsive Multithreaded Processor

本章では，まず本研究で想定しているリアルタイムシステムについて述べる．次にリアルタイム処理用プロセッサとして設計，実装している Responsive Multithreaded（RMT）Processor の概要を述べ．RMT Processor のプロセッシングユニットである，RMT PU の優先度制御機構によるスレッドの実行について述べる．そして RMT PU における本研究の研究範囲について述べる．

4.1 想定するシステム

本研究では様々な時間制約を持った複数のスレッドを実行するシステムを想定している．つまり，ハードリアルタイム処理とソフトリアルタイム処理の両方が複数実行されるシステムを想定している．

例えばロボットの場合，ハードリアルタイム性を持つ処理として制御を行うスレッドがある．制御スレッドでは各センサの値を取得したり，アクチュエータを PID 制御してロボットを動作させる．一方ソフトリアルタイム性を持つ処理として画像処理を行うスレッドや，音声認識を行うスレッドが考えられる．

リアルタイムシステムでは，各スレッドの時間制約を守るために，各スレッドの時間制約に基づいて優先度を与え，優先度の高いスレッドから順番に実行する．ハードリアルタイム性を持つ処理は実行周期が短かく，時間制約が厳しいといった特徴がある．一方，ソフトリアルタイム性を持つ処理は実行周期が長く，時間制約が比較的厳しくないといった特徴がある．RM スケジューリングでハードリアルタイムスレッドとソフトリアルタイムスレッドをスケジューリングした場合，実行周期の短かいハードリアルタイムスレッドに高い優先度が与えられ，実行周期の長いソフトリアルタイムスレッドに低い優先度が与えられる．EDF スケジューリングでスケジューリングした場合はデッドラインまでの時間が短いスレッドに高い優先度が与えられる．ハードリアルタイムスレッドは実行周期が短かいため，ソフトリアルタイムスレッドに比べてデッドラインが近い場合が多い．そのため

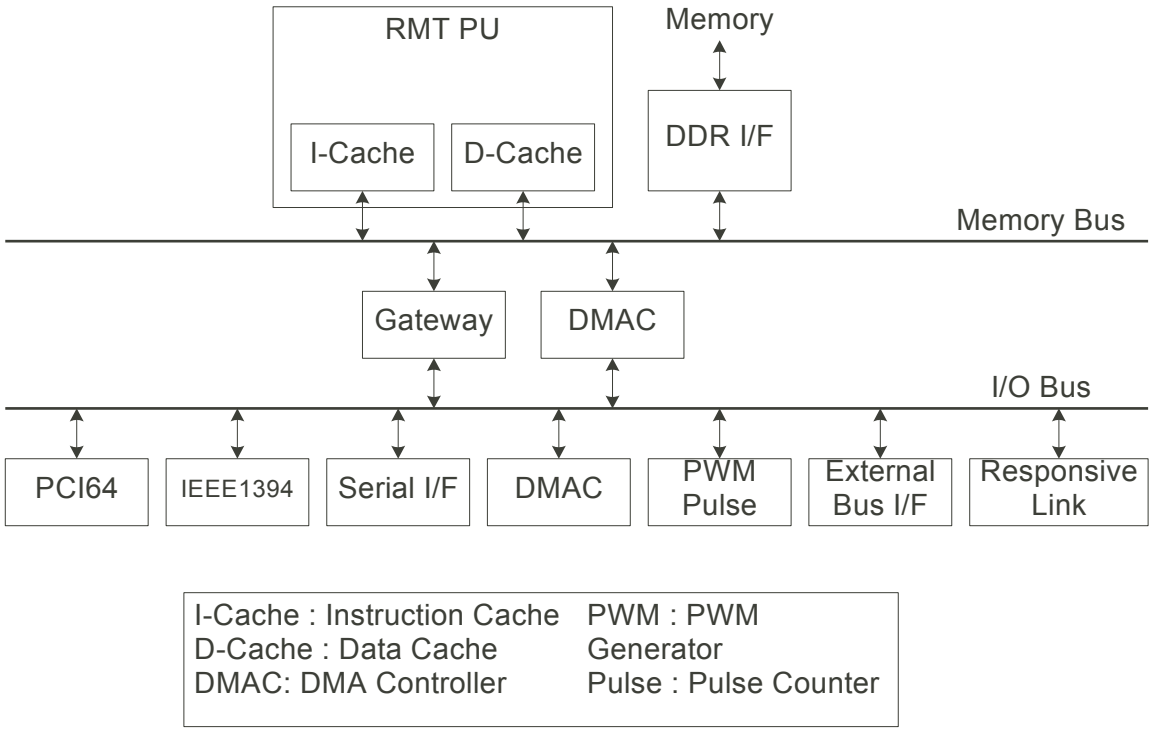

図 4.1: RMT Processor のブロック図

ソフトリアルタイムスレッドに比べてハードリアルタイムスレッドの優先度が高い時間が長いと考えられる．このように，リアルタイムシステムではハードリアルタイムスレッドとソフトリアルタイムスレッドが同時に実行されている場合，ハードリアルタイムスレッドの方が優先して実行される．

4.2 Responsive Multithreaded Processorの概要

Responsive Multithreaded（RMT）Processor[Yamasaki 05] はリアルタイム処理を行うためのプロセッサとして設計，実装が行われている．RMT Processor はプロセッシングユニットである RMT PU，リアルタイム通信用規格である Responsive Link[Yamasaki 01]，DDR SDRAM I/F，DMA コントローラ，PCI64 I/F，IEEE 1394 I/F，シリアル I/F，外部 I/F といったコンピュータ用 I/O，PWM ジェネレータ，パルスカウンタといった制御用 I/O を 1 チップに集積した System on a Chip（SoC）である．1 チップに制御用の I/O とコンピュータ用の I/O を集積しているため，メモリをつなぐだけでシステムを構築することができる．また，Responsive Link により，分散処理を可能としている．

図 4.1 に RMT Processor のブロック図を示す．RMT PU はメモリアクセスのスループットを向上するめに，Memory Bus（256bit 幅）を介してメモリインターフェースと接続し

ている．各I/Oは回路規模を抑えるためにI/O Bus（32bit幅）に接続している．Memory BusとI/O Busはゲートウェイ（GW）を介して接続されている．それぞれのバスを流れるデータはゲートウェイにおいてバスサイジングが行われ，もう片方のバスに送られる．また，Memory BusとI/O Busの両方に接続されているDMAコントローラがあり，Memory BusとI/O Busの間でバスサイジングを行いながらDMA転送を行うことができる．

Responsive Linkのイベントリンクは RMT PUのメモリアクセスユニットに直接接続されており，RMT PUからはバスを介さず，制御レジスタの一部としてアクセスすることできる．これにより，高速にイベントリンクにアクセスすることができる．

4.3 RMT PUの優先度制御機構

リアルタイム処理では，スレッドを切り替える場合に発生するコンテキストスイッチのオーバーヘッドを削減することが重要である．従来研究では，マルチスレッドアーキテクチャを用いることにより，コンテキストスイッチを発生させずにスレッドの切り替えを行っている．RMT PUはSMTアーキテクチャに優先度を導入したアーキテクチャになっている．

SMTアーキテクチャでは，命令スロットに対してどのようにスレッドを割り当てていくかが重要になる．スループットを向上させる割り当て方法として，パイプライン内の命令数をもとに，最も命令数の少ないスレッドから割り当てる方式（ICOUNT）[Tullsen *et al.* 96]がある．しかし，リアルタイム処理ではスケジューラが決定した優先度に従って，スレッドを順番に実行することが重要である．そのため，RMT PUでは優先度を用いて命令スロットの割り当てを行う．

優先度に従った命令スロットの割り当て方法を図4.2に示す．図ではThread 0の優先度が最も高く，Thread 7の優先度が最も低い．RMT PUでは最も優先度の高いスレッドから命令スロットを割り当てる．よってThread 0の命令を最初に命令スロットに割り当てる．最も優先度の高いスレッドの命令が依存関係などにより，命令スロットに割り当てられない場合，次に優先度の高いスレッドの命令を割り当てる．よってThread 0の命令が割り当てられない場合はThread 1の命令が空いている命令スロットに割り当てられる．このようにして優先度の高いスレッドから命令スロットを割り当て，優先度の高いスレッドの命令が割り当てられない場合は次に優先度の高いスレッドから命令を割り当てることにより，優先度の高いスレッドから順番に実行しつつ，命令スロットの空きを無くし，スループットを向上している．

図4.3にシングルプロセッサによる，従来のリアルタイム処理のシーケンスを示す．従来のリアルタイム処理では，各スレッドは優先度に従って，優先度の高いスレッドから順番に実行される．実行するスレッドが切り替わる場合，コンテキストスイッチを行ってか

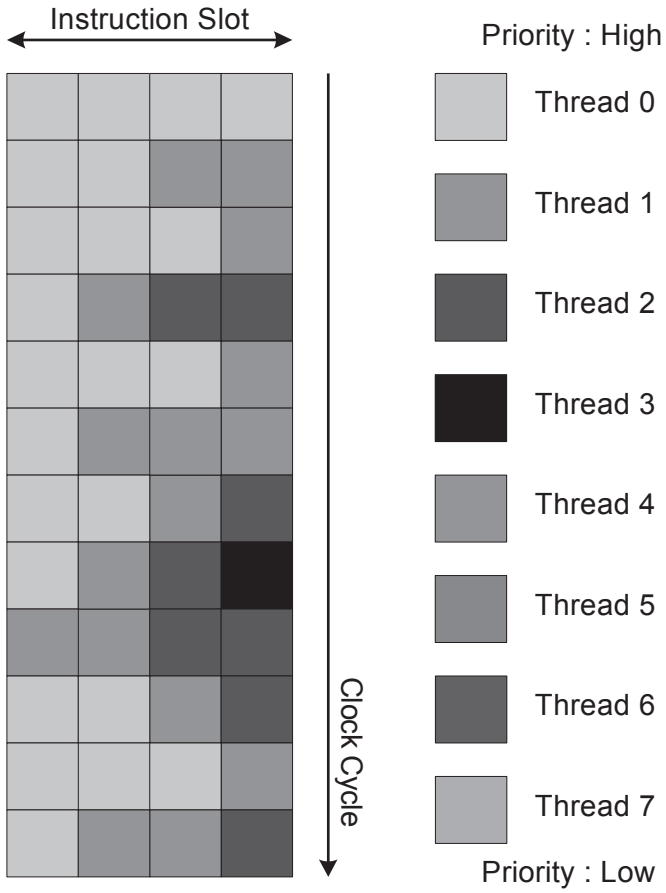

図 4.2: RMT PUにおける優先度による命令スロットの割り当て

ら，次のスレッドの実行が開始される．シングルプロセッサではコンテキストスイッチをソフトウェアで行う．そのためコンテキストスイッチのオーバーヘッドが問題となる．

図4.4にマルチスレッドプロセッサに優先度を用いた場合のリアルタイム処理のシーケンスを示す．マルチスレッドプロセッサに優先度を用いることにより，ハードウェアコンテキストに保持されているスレッドを優先度に従って切り替えて実行する．スレッドはハードウェアコンテキストに保持されているため，スレッドを切り替える場合にコンテキストスイッチは発生しない．従来のソフトウェアによるコンテキストスイッチを優先度付きマルチスレッディングに置き換えることにより，コンテキストスイッチを行わずに優先度の高いスレッドから順番に実行している．

図4.5にRMT PUによるリアルタイム処理のシーケンスを示す．RMT PUにおいても，ハードウェアコンテキストの保持されているスレッドを優先度に従って切り替えて実行する．従来のソフトウェアによるコンテキストスイッチを優先度付きSMTに置き換えることにより，コンテキストスイッチを行わずに優先度の高いスレッドから順番に実行することができる．また，優先度付きマルチスレッディングと異なり，SMTアーキテクチャによ

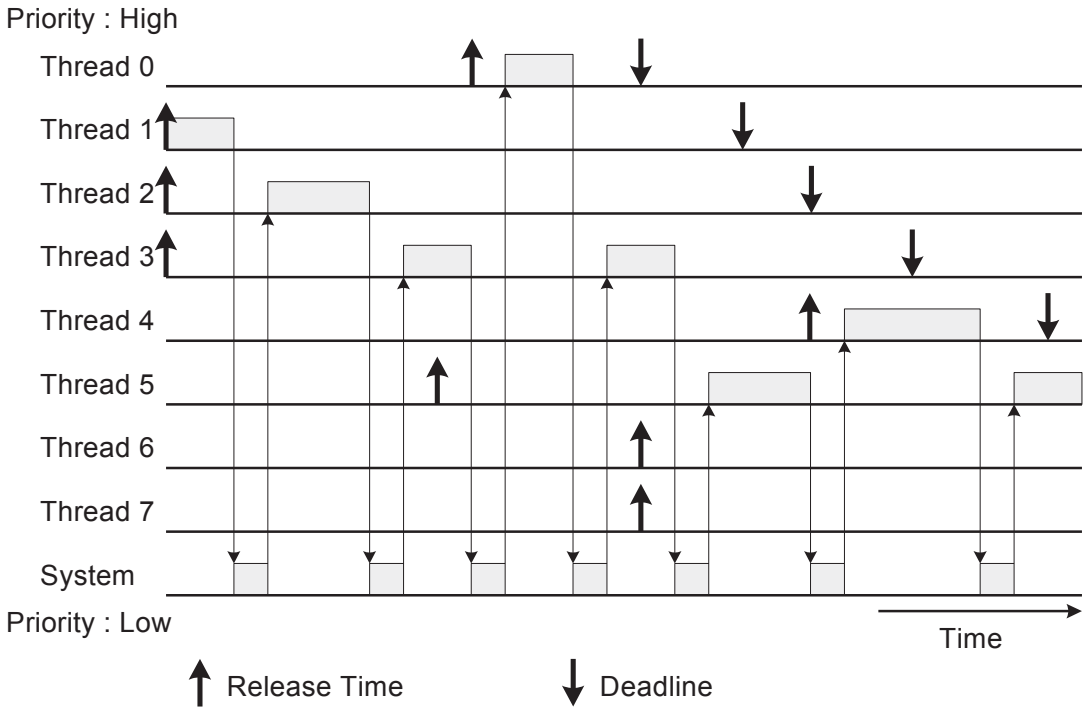

図 4.3: シングルプロセッサによる従来の実行

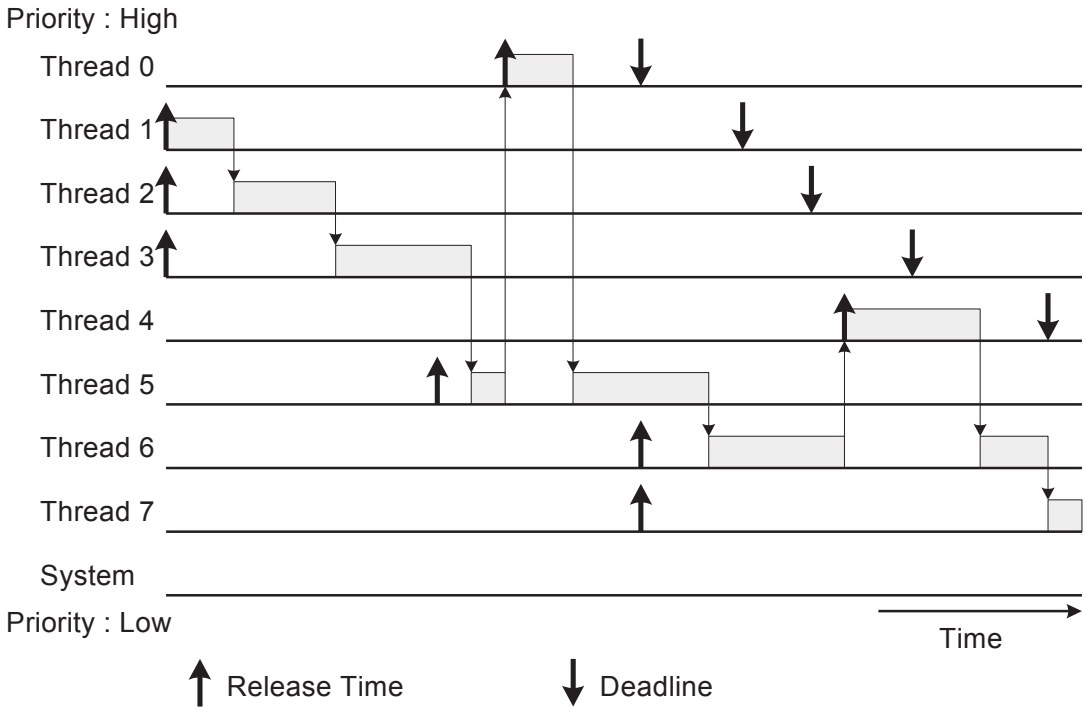

図 4.4: シングルパイプラインのマルチスレッドプロセッサに優先度を用いた実行

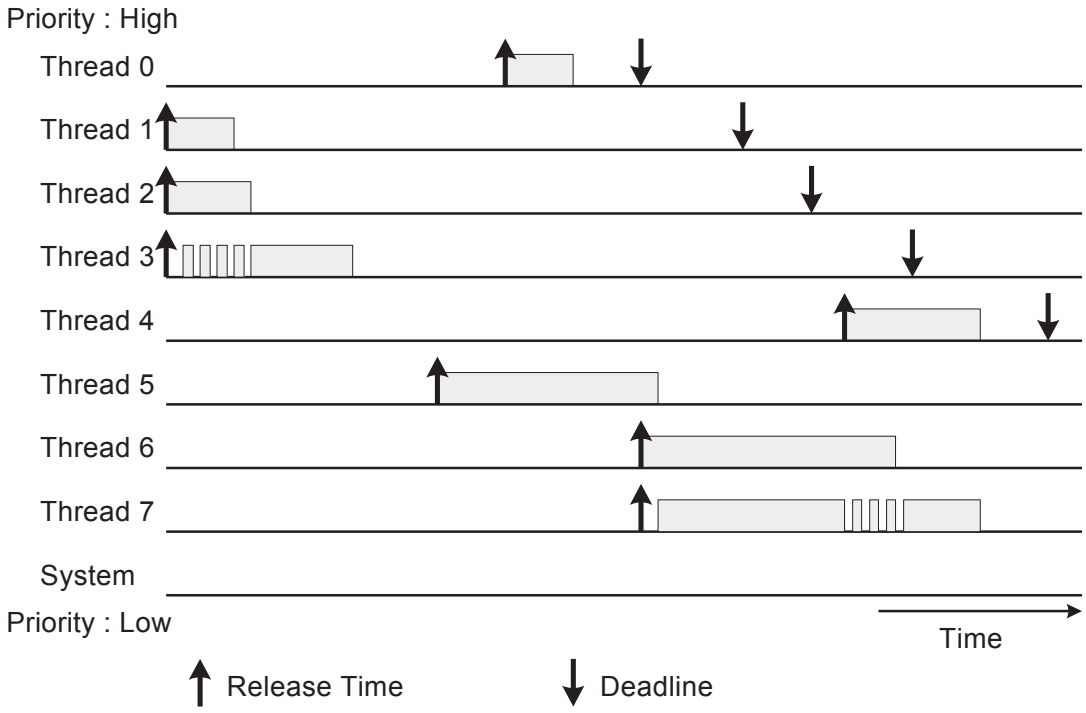

図 4.5: RMT PU による実行

り，空いている命令スロットを優先度に従って別のスレッドに割り当てることができるため，複数のスレッドを並列に実行することができ，全体のスループットが向上する．

図 4.6 に優先度を付与した 8 つのスレッドを RMT PU で実行した場合の各スレッドの実行の様子を示す．図では Thread 0 に最も高い優先度を与え，Thread 7 に最も低い優先度を与えている．横軸は時間経過，縦軸は単位時間当たりの命令コミット数をあらわしている．図では最初に Thread 0，Thread 1，Thread 2 が実行されている．これは，RMT PU の優先度制御により最も優先度の高い Thread 0 に命令スロットが割り当てられるためである．Thread 0 に命令スロットを割り当てられない場合，Thread 1 に命令スロットが割り当てられるため，Thread 1 が並列に実行されている．Thread 0，Thread 1 に命令スロットが割り当てられない場合は Thread 2 に命令スロットが割り当てられるため，Thread 2 も並列に実行されている．ただし，Thread 0 や Thread 1 に比べて命令スロットを割り当てられる率が少ないため，単位時間当たりの命令コミット数は Thread 0 や Thread 1 と比較して少ない．Thread 0，Thread 1 の実行が終了すると，Thread 2 の優先度が最も高くなるため，Thread 2 に対して優先的に命令スロットが割り当てられる．そのため，Thread 0，Thread 1 の実行が完了すると，Thread 2 の命令コミット数が増加する．また，Thread 2 に対して命令スロットが割り当てられない場合は Thread 3，Thread 4 に対して命令スロットが割り当てられるため，Thread 3，Thread 4 が実行されている．このように，RMT PU

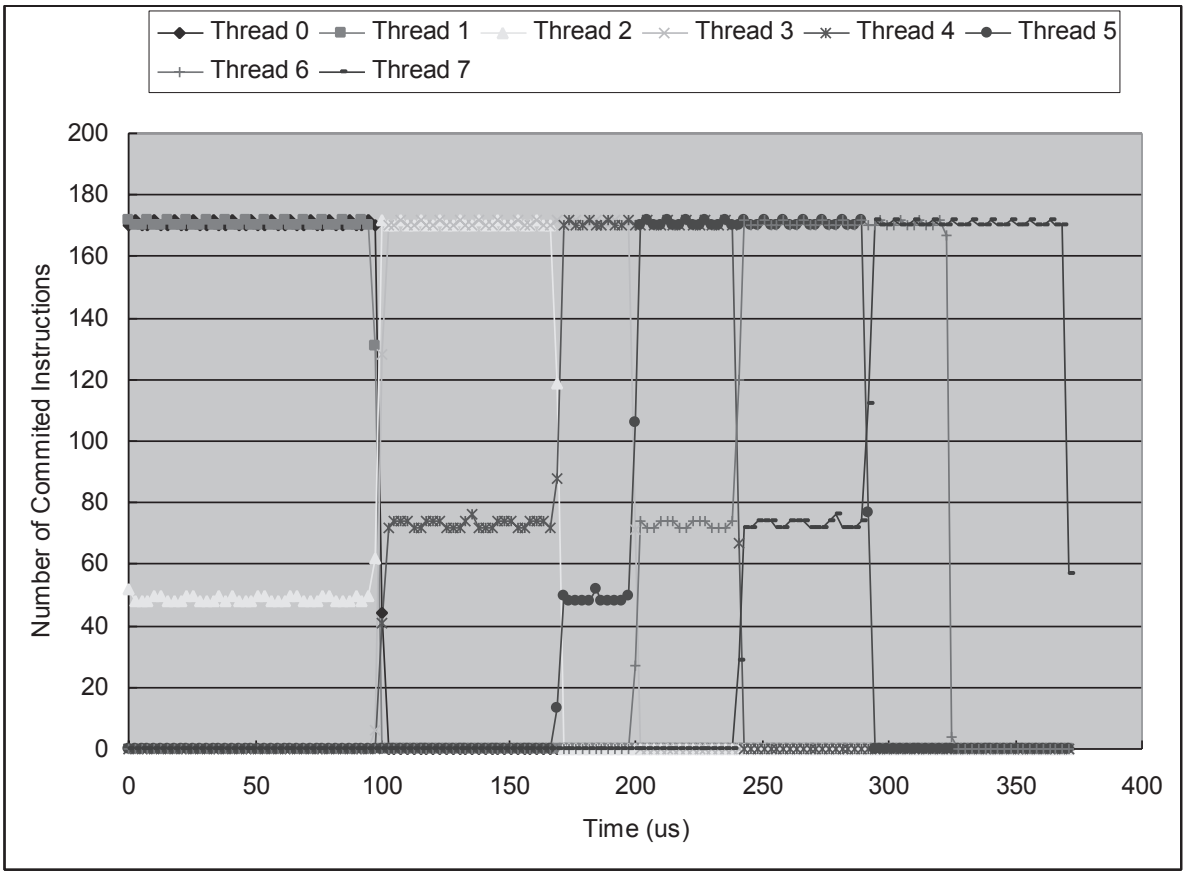

図 4.6: RMT PU による実際の実行

では優先度付き SMT により，コンテキストスイッチを行わずに，優先度に従ってスレッドを実行しつつ，全体のスループットを向上している．

4.4 本研究の研究範囲

2.5 節において，ハードリアルタイム処理とソフトリアルタイム処理を同時に行うために，リアルタイム処理用プロセッサの機能として，小さいオーバーヘッドでコンテキストスイッチを行う機能と，ソフトリアルタイム処理の大量のデータを演算する機能が必要であることを述べた．

RMT PU では，優先度付き SMT アーキテクチャにより，ハードウェアコンテキストに保持され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，優先度に従って実行する．これによりコンテキストスイッチのオーバーヘッドを削減している．しかし，ハードウェアコンテキスト数（RMT PU では 8 個）よりも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を発生させないためにはプロセッサで保持することのできるハードウェアコンテキスト数を増やす方法が考えられる．例えばハードウェアコンテキスト数を 16 個にすれば，16 個までのスレッドをコンテキストスイッチを発生させずに RMT PU で優先度に従って実行することができる．しかし 16 個分のコンテキストを保持するためのレジスタを用意する必要があり，回路規模が増加する．また，RMT PU では優先度に従って命令スロットを割り当てていくため，ハードウェアコンテキストの数が増加すると，命令を選択するためのハードウェアが複雑化し，回路規模の増加，動作周波数の低下につながる．よってハードウェアコンテキスト数を増加させる方法は，数に限界があり，スレッドの数がさらに増加した場合に，柔軟に対応することができない．よって，ハードウェアコンテキスト数よりも多くのスレッドを実行する場合に，小さいオーバーヘッドでコンテキストスイッチを行うための機構が必要となる．本研究では，この要求を実現するために，ハードウェアでコンテキストスイッチを行う，スレッド制御機構を設計，実装する．

ソフトリアルタイム処理では大量のデータを演算する．ソフトリアルタイム処理では 4.1 節で述べたように，ハードリアルタイム処理に比べて優先度が低い．RMT PU の優先度制御機構は，優先度の高いスレッドに対して命令スロットを多く割り当てるため，ハードリアルタイム処理とソフトリアルタイム処理の両方を同時に実行する場合，ハードリアルタイム処理を行うスレッドに命令スロットが多く割り当てられ，ソフトリアルタイム処理を行うスレッドに十分に命令スロットが割り当てられず，大量のデータを演算することができない．よって，ソフトリアルタイム処理の大量のデータを演算するために，RMT PU の優先度制御を考慮したベクトル演算機構を設計，実装する．スレッド制御機構とベクトル演算機構により，RMT PU がハードリアルタイム処理とソフトリアルタイム処理の両方を同時に実行することを可能とする．

4.5 本章のまとめ

本章では，リアルタイム処理用プロセッサとして設計が行われている RMT Processor の概要について述べ，RMT PU による優先度制御と本研究が扱う研究範囲について述べた．

RMT Processor のプロセッシングユニットである RMT PU は優先度付き SMT アーキテクチャにより，ハードウェアコンテキストに保持されているスレッドを優先度に従って順番に実行する機能を持つ．ハードウェアコンテキストに保持されているスレッドはコンテキストスイッチを行わずに切り替えることができるため，8 個までのスレッドを実行する場合はコンテキストスイッチは発生しない．

しかし，9 個以上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．本研究では，このような場合に小さいオーバーヘッドでコンテキストスイッチを実現するため

に，ハードウェアによるコンテキストスイッチを行うスレッド制御機構を設計，実装する．

また，ハードリアルタイム処理とソフトリアルタイム処理の両方を実現する場合，ソフトリアルタイム処理の大量のデータを演算する機能が必要となる．従来研究では，データ並列性を利用しているが，RMT PUでソフトリアルタイム処理の演算を実現する場合，RMT PUが優先度に従って複数のスレッドを実行していることを考慮する必要がある．本研究では，このような状況を考慮したベクトル演算機構を設計，実装する．

次章では，スレッド制御機構の詳細について述べる．そして6章ではベクトル演算機構の詳細について述べる．

第5章

スレッド制御機構

本章では，スレッドの切り替えを小さいオーバーヘッドで実現するために，ハードウェアで優先度に従ってスレッドの切り替えを行う，スレッド制御機構について述べる．そして本手法について従来の手法と比較を行い，本手法の有効性を示す．

5.1 基本方針

3.1節で述べたように，マルチスレッドアーキテクチャを用い，ハードウェアコンテキストに割り当てたスレッドを切り替える場合はコンテキストスイッチが発生しない．よってハードウェアコンテキスト数を増やすことにより，より多くのスレッドを切り替えて実行した場合でも，コンテキストスイッチのオーバーヘッドを削減することができる．しかし，ハードウェアコンテキスト数を増やすということは，その分だけコンテキストを保持しておくレジスタを用意しなければならない．そのため回路規模が増加する．また，RMT PUは優先度に従って各スレッドに命令スロットを割り当てる．ハードウェアコンテキストが増加すると，命令スロットを割り当てるための比較器が増加し，回路規模の増加，動作周波数の減少につながる．よって，ハードウェアコンテキスト数を増やすという方法は有効でない．

ソフトウェアによるコンテキストスイッチでは，Load命令を用いてコンテキストを1つ1つメモリに退避し，Store命令を用いてコンテキストを1つ1つ復帰するため，大量のメモリアクセスが発生し，オーバーヘッドが大きくなる．よってメモリアクセスのスループットを改善することにより，コンテキストスイッチのオーバーヘッドを削減できると考えられる．本研究ではメモリアクセスのスループットを改善するために，コンテキストを格納するための専用メモリをオンチップに用意する．そしてハードウェアで自動的にコンテキストスイッチを行うことにより，コンテキストスイッチのオーバーヘッドを削減する．

オンチップメモリにより，外部メモリへのアクセスがなくなるため，低レイテンシでメモリにアクセスすることが可能となる．ハードウェアで自動的にコンテキストスイッチを

行うことにソフトウェアで1つ1つ転送せずに，一度にまとめてコンテキストを転送することができ，コンテキストスイッチにかかる時間を削減する．オンチップメモリはハードウェアコンテキストを保持するためのレジスタを用意するよりも回路規模を抑えることができる．そのため，実行するスレッドの数が増加した場合でも十分対応することができると考えられる．本研究ではこのオンチップメモリをコンテキストキャッシュと呼ぶ．

本研究では，コンテキストキャッシュとRMT PUのハードウェアコンテキストの間のコンテキストスイッチをハードウェアで行う．リアルタイム処理では，各スレッドに付与された優先度に従ってスレッドを順番に実行することにより，各スレッドの時間制約を守る．よってハードウェアでコンテキストスイッチを行う場合においても，スレッドに付与された優先度に従って，コンテキストキャッシュ内のスレッドを順番にRMT PUのハードウェアコンテキストに割り当てる．コンテキストスイッチをハードウェアだけで行うことにより，ソフトウェアによるオーバーヘッドを削減する．

2.4節で述べたように，リアルタイム処理では多くの場合，割り込みによりスレッドが起動する．例えば，周期スレッドはタイマ割り込みによって起動する．リアルタイム処理では割り込みに対して素早く応答することが重要である．従来の手法では，割り込みが発生すると，割り込みハンドラが起動し，割り込み要因により，該当するスレッドを起動する．ソフトウェアにより，割り込み要因を調べるための処理が入るため，割り込み応答性が低下する．本研究では，割り込みによるスレッドの起動をハードウェアで行うことにより，割り込み応答性を向上する．

5.2 設計

5.2.1 コンテキストキャッシュ

本研究では，コンテキストスイッチを行う場合にメモリアクセスのスループットを向上するために，コンテキストキャッシュを用意する．コンテキストキャッシュはオンチップに存在するため，コンテキストキャッシュのアクセスは，RMT PUと同じ周波数で行うことができる．また，オンチップにメモリがあるため，外部のメモリに比べて低レイテンシでアクセスすることができ，メモリアクセスのスループットを向上することができると考えられる．

コンテキストスイッチで入れ替えるデータは，汎用レジスタ，浮動小数点レジスタ，制御レジスタである．それぞれのレジスタは独立しているため，それぞれ並列にデータを入れ替えることにより，一度に複数のデータを入れ替えることができ，メモリアクセスのスループットが向上する．よって，図5.1に示すように，これらのレジスタに対応した3種類のコンテキストキャッシュを用意し，それぞれのレジスタと専用バスで接続する．RMT

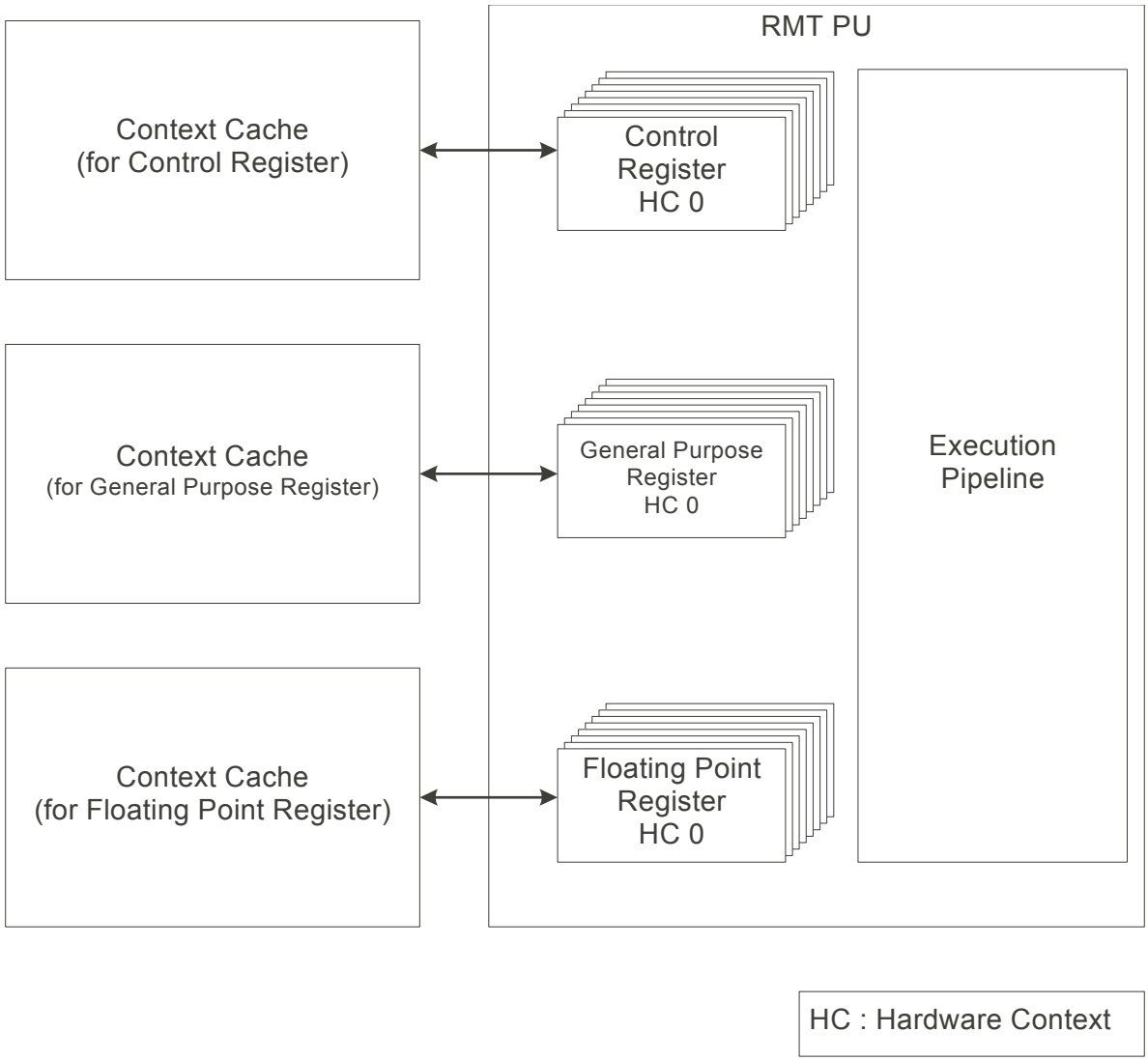

図 5.1: コンテキストキャッシュ

PUでは8つのハードウェアコンテキストを持つため，コンテキストキャッシュとこれら8つのレジスタの間で専用バスを介してデータを転送する．

コンテキストスイッチはコンテキストの退避（メモリへの書き込み）と復帰（メモリからの読み出し）を行う．チップ内では通常，双方向バスは用いないため，コンテキストキャッシュとRMT PUのハードウェアコンテキストをつなぐバスは，退避用のバスと復帰用のバスで別々に用意される．よって，コンテキストキャッシュをデュアルポートメモリで構成することにより，退避用のバスと復帰用のバスを有効に使用することができ，コンテキストスイッチにかかる時間を半分に短縮することができる．デュアルポートメモリは回路規模が増加するが，コンテキストスイッチにかかる時間が半分になるため，本研究では，デュアルポートメモリを用いてコンテキストの退避と復帰を並列に行う．

チップの外での配線では，パッケージのピン数の制約により，メモリとの間のバス幅を広げることはできない．しかし，チップ内の配線では，ピン数の制約がないため，コンテキストキャッシュと RMT PU のハードウェアコンテキストの間のバス幅を広げることができる．バス幅を大きくすることにより，一度に複数のデータを転送することができるため 1 つ 1 つデータを入れ替える場合に比べて，メモリアクセスのスループットが向上する．よって，コンテキストキャッシュと RMT PU のハードウェアコンテキストをつなぐバスのバス幅ついて検討する必要がある．

コンテキストキャッシュと RMT PU のハードウェアコンテキストを 1 対 1 で接続することにより，1 クロックでコンテキストスイッチが完了する．この場合，汎用レジスタでは 2048bit（32bit × 32 ワードで読みと書きの 2 つ）のバス幅が必要になる．チップ外の配線に比べてバス幅の制限が少ないが，このようなビット幅の大きいバスが多く存在すると ASIC に実装する場合に，配線面で問題になる．本研究ではこの問題を解決するために，コンテキストキャッシュと RMT PU のハードウェアコンテキストを接続するバスのバス幅を全データのビット幅の 1/4 にし，4 クロックで全てのレジスタの入れ替えを完了するようにする．コンテキストスイッチにかかる時間が増加するが，RMT PU のマルチスレッディングにより，レイテンシを隠蔽することができると考えられる．

コンテキストキャッシュ内に保持するコンテキストの数が多いほど，多くのスレッドを小さいオーバーヘッドで切り替えて実行することが可能となる．しかし，コンテキスト数を増すためにはオンチップメモリの容量を増やす必要があるため，回路規模が増加する．また，メモリの容量が増えれば増えるほど，オンチップメモリの動作周波数は低下していく．本研究ではオンチップメモリは RMT PU と同じ周波数で動作するため，動作周波数の低下は問題となる．また，コンテキスト数が増加すると，次に述べるスレッド切り替え機構において，優先度を比較する回数が増加する．そのため，コンテキスト数の増加はオンチップメモリの動作周波数だけでなく，スレッド切り替え機構の動作周波数にも影響する．本研究では回路規模と動作周波数を考慮し，コンテキストキャッシュに保持するコンテキストの数は 32 個とする．

5.2.2 スレッドの状態

コンテキストスイッチを行う場合，どのスレッドとどのスレッドを入れ替えるかを選択する必要がある．コンテキストキャッシュにより，コンテキストの入れ替え自体は 4 クロックで終了する．そのため，入れ替えるスレッドの選択をソフトウェアで行うと，コンテキストスイッチよりもスレッドの選択に大幅な時間を要し，コンテキストキャッシュを用いた入れ替えを有効に使用することができない．コンテキストキャッシュを用いた入れ替え

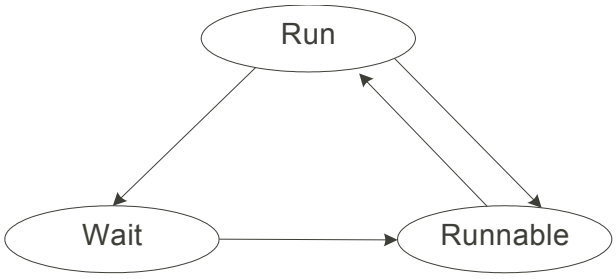

図 5.2: 一般的な OS のスレッドの状態

を有効に使用するためには，入れ替えるスレッドの選択も短かい時間で行う必要がある．よって本研究では，入れ替えるスレッドの選択をハードウェアで行う．つまり，RMT PU のハードウェアコンテキストとコンテキストキャッシュの間のコンテキストスイッチをソフトウェアを用いず，ハードウェアで自動的に行う．ハードウェアでコンテキストスイッチを行うために，各スレッドの状態をハードウェアで管理する必要がある．

図 5.2 に一般的な OS で使用されているスレッドの状態を示す．

実行（Run）状態はスレッドに CPU が割り当てられていて，スレッドが実行されている状態である．最大で CPU の数だけ実行状態のスレッドが存在する．他のスレッドによりプリエンプションが起った場合，スレッドの実行が中断され，実行可能状態へと遷移する．スレッドが I/O など他の処理の結果を待たなければならず，これ以上実行を続けられなくなると，待ち状態へと遷移する．また，周期的に起動するスレッドでは，一周期分の処理を終え，次の周期まで実行を停止する場合も待ち状態（OS の実装によっては休止状態）へと遷移する．

実行可能（Runnable）状態はスレッドが実行可能になった状態である．現在実行状態のスレッドが待ち状態へと遷移した場合，実行可能状態のスレッドの中から最も優先度の高いスレッドが CPU を割り当てられて実行状態に遷移する．また，プリエンプションを許している OS では，現在実行状態のスレッドよりも優先度の高いスレッドが実行可能状態になった場合，現在実行状態のスレッドを実行可能状態に遷移させ，優先度の高いスレッドを実行状態に遷移させる．

待ち（Wait）状態はスレッドが実行できない状態である．I/O や他のスレッドの処理結果を待つ場合や，一周期分の処理が完了し，次の周期まで実行を待つ場合にこの状態になる．OS によっては次の周期まで待つ場合は休止状態として状態を分ける場合がある．待ち状態のスレッドは実行できない理由が解決した場合，実行可能状態へと遷移する．具体的には I/O や他のスレッドの処理が完了した場合や，次の実行周期が来た場合，待ち状態から実行可能状態へと遷移する．

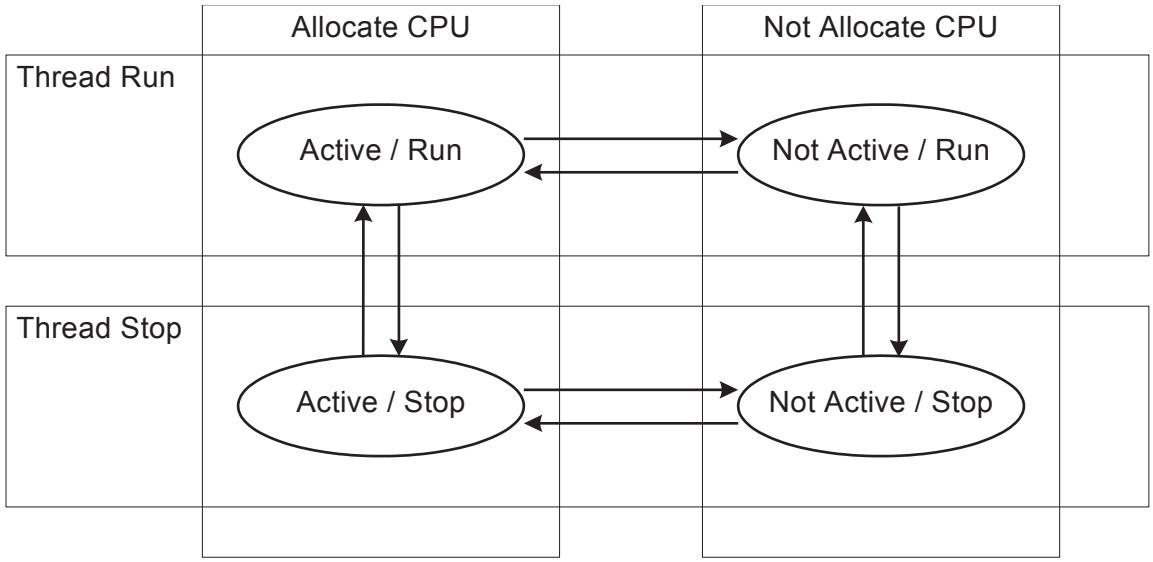

図 5.3: スレッド制御機構におけるスレッドの状態の定義

本研究では，プロセッサ内で保持されているスレッドは，RMT PU のハードウェアコンテキストを割り当てられているか，コンテキストキャッシュに保持されているかの二つである．また，リアルタイム処理では，実行可能なスレッドの中から優先度の高い順番にスレッドを実行していく．よって，スレッドが実行できるのか，実行できない（停止している) のかを判断する必要がある．以上のことから，本研究ではスレッドの状態を RMT PU のハードウェアコンテキストに割り当てられているか（Active），割り当てられていないか（Not Active）という情報と，スレッドが実行可能であるか（Run），I/O 待ちや次の実行周期を待っているために停止しているか（Stop）という情報の組合せで定義する．

図 5.3 にスレッド制御機構で定義したスレッドの状態について示す．スレッド制御機構ではハードウェアでコンテキストスイッチを行うスレッドを選択するために，各スレッドの状態をハードウェアで遷移させる．

Run / Stop はスレッドの実行状態を示す．スレッド制御機構は，スレッドが実行可能になるとスレッドの状態を Run 状態へと遷移する．I/O や他のスレッドの処理を待たなければならず，スレッドが実行できない場合はスレッドの状態を Stop 状態へと遷移する．また，一周期分の処理が完了し，次の実行周期が来るまで待つ場合もスレッドの状態を Stop 状態へと遷移する．

スレッド制御機構がコンテキストスイッチを行うことにより，スレッドを RMT PU のハードウェアコンテキストに割り当てると，スレッドの状態を Active 状態へと遷移する．スレッドがコンテキストスイッチにより，RMT PU のハードウェアコンテキストからコンテキストキャッシュに退避されると，スレッドの状態を Not Active 状態へと遷移する．Run

/ Stop の状態と Active / Not Active の状態を示すために，2ビットを用いてスレッドの状態を表現する．

図 5.2 の一般的な OS のスレッドの状態と比較すると，Active / Run 状態はスレッドが実行可能であり，RMT PU のハードウェアコンテキストに割り当てられている（CPU が割り当てられている）ため，実行状態に対応する．Not Active / Run 状態はスレッドが実行可能であるが，RMT PU のハードウェアコンテキストが割り当てられていない（CPU が割り当てられていない）ため，実行可能状態に対応する．Active / Stop 及び Not Active / Stop はスレッドが実行できない状態のため，待ち状態に対応する．

スレッド制御機構ではハードウェアでコンテキストスイッチを行うために，スレッドの状態はハードウェアで自動的に遷移する．しかしソフトウェアによってスレッドの実行を明示的に制御する場合が考えられる．よって，本研究ではソフトウェアでスレッドの制御を行うための専用命令を用意する．以下に述べるスレッド制御命令を用いることにより，ソフトウェアでスレッドの実行を明示的に制御する．

RMT PU では，各スレッドを識別するために，スレッドの生成時にそのスレッドにソフトウェアでスレッド ID を付与する．スレッド制御用命令はスレッド ID を用いることにより各スレッドを識別する．スレッドの状態を制御する命令を以下に示す．

- mkth rd, rs, rt

  スレッドを新しく生成する．rs に生成するスレッドに付与するスレッド ID，rt に開始アドレスを指定する．スレッドの生成に成功した場合は rd に 1 が，失敗した場合は 0 が返る．

- delth rd, rs

  スレッドを削除する．rs に削除するスレッドのスレッド ID を指定する．スレッドの削除に成功した場合は rd に 1 が，失敗した場合は 0 が返る．

- runth rd, rs

  スレッドを Run 状態にする．rs に Run 状態にするスレッドのスレッド ID を指定する．スレッドの実行に成功した場合は rd に 1 が，失敗した場合は 0 が返る．

- stopth rd, rs

  スレッドを Stop 状態にする．rs に Stop 状態にするスレッドの ID を指定する．スレッドの停止に成功した場合は rd に 1 が，失敗した場合が 0 が返る．

- stopslf rd

自身を Stop 状態にする．スレッドの停止に成功した場合は rd に 1 が，失敗した場合は 0 が返る．

- bkupth rd, rs

  スレッドをコンテキストキャッシュに退避し，Not Active 状態にする．スレッドの退避に成功した場合は rd に 1 が，失敗した場合は 0 が返る．

- bkupslf rd

  自身をコンテキストキャッシュに退避し，Not Active 状態にする．スレッドの退避に成功した場合は rd に 1 が，失敗した場合は 0 が返る．

- rstrth rd, rs

  スレッドをコンテキストキャッシュから RMT PU のハードウェアコンテキストに復帰して，Active 状態にする．スレッドの復帰に成功した場合は rd に 1 が，失敗した場合は 0 が返る．

- swapth rd, rs, rt

  実行中のスレッドとコンテキストキャッシュ内のスレッドを入れ替える．rs に実行中のスレッドのスレッド ID，rt にコンテキストキャッシュ内のスレッドのスレッド ID を指定する．実行中のスレッドはコンテキストキャッシュに退避され，Not Active 状態になる．コンテキストキャッシュ内のスレッドは RMT PU のハードウェアコンテキストが割り当てられ，Active 状態となる．スレッドの入れ替えに成功した場合は rd に 1 が，失敗した場合は 0 が返る．

- swapslf rd, rt

  自分自身とコンテキストキャッシュ内のスレッドを入れ替える．rt にコンテキストキャッシュ内のスレッドのスレッド ID を指定する．自身はコンテキストキャッシュに退避され，Not Active 状態になる．コンテキストキャッシュ内のスレッドは RMT PU のハードウェアコンテキストが割り当てられ，Active 状態となる．スレッドの入れ替えに成功した場合は rd に 1 が，失敗した場合は 0 が返る．

5.2.3 スレッド入れ替え機構

本研究では，コンテキストキャッシュを用いたコンテキストスイッチを効果的に行うために，ハードウェアで入れ替えるスレッドを選択する．リアルタイム処理では，各スレッドの優先度に従って，優先度の高い順番にスレッドを実行する．よって，本研究では各ス

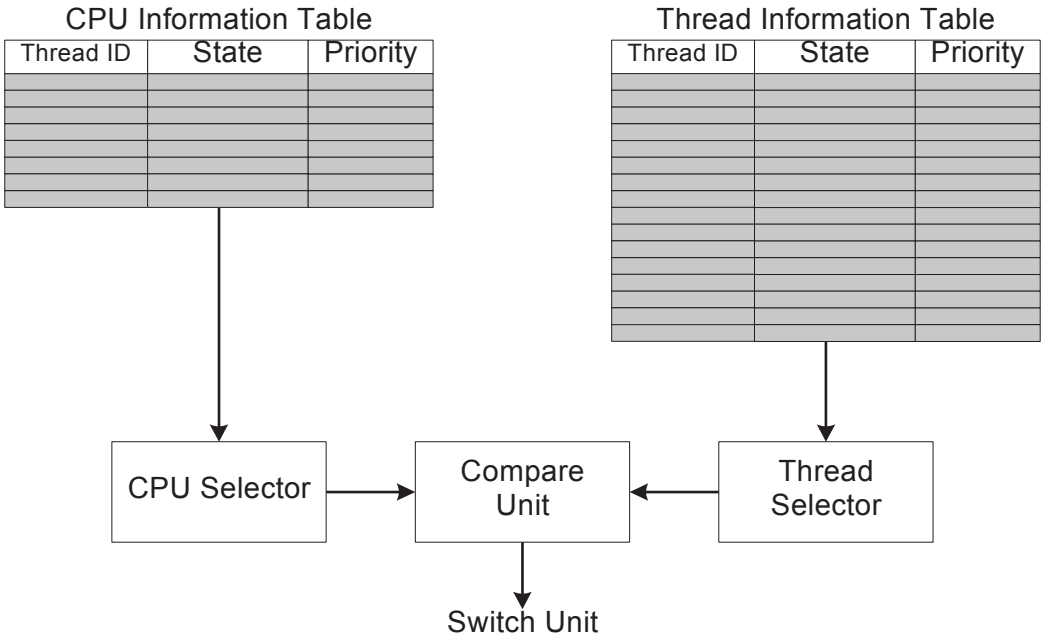

図 5.4: スレッド入れ替え機構のスレッド選択部

レッドに付与されている優先度と 5.2.2 節で述べたスレッドの状態を用いてコンテキストスイッチを行うスレッドを選択する．

図 5.4 にスレッド入れ替え機構のスレッド選択部を示す．コンテキストキャッシュに保持されている全てのスレッドの情報は Thread Information Table で管理する．Thread Information Table はハードウェアで入れ替えるスレッドを選択するための情報として，5.2.2 節で述べたスレッドの状態（State）と優先度（Priority）を保持する．本研究ではコンテキストキャッシュは 32 個のスレッドを保持するため，Thread Information Table は 32 エントリとなる．

RMT PU は 8 つのハードウェアコンテキストを持つ．コンテキストスイッチを行うためには，8 つの中から退避するスレッドを選択する必要がある．退避するスレッドの選択をハードウェアで行うために，RMT PU のハードウェアコンテキストに割り当てられているスレッドの状態を CPU Information Table で管理する．CPU Information Table には，RMT PU のハードウェアコンテキストに割り当てられているスレッドの状態（State）と優先度（Priority）を保持する．本研究では RMT PU のハードウェアコンテキストは 8 つのため，CPU Information Table は 8 エントリとなる．

本研究ではコンテキストスイッチを行うスレッドをハードウェアで選択し，コンテキストキャッシュを用いて自動的にコンテキストスイッチを行うことにより，コンテキストスイッチのオーバーヘッドを削減する．しかし，スレッドによっては，ソフトウェアで実行を

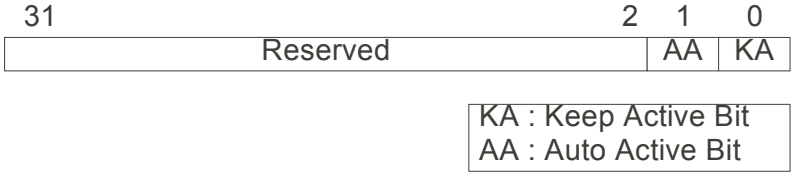

図 5.5: スレッド制御レジスタ（1）

制御するために，ハードウェアによる制御を行いたくない場合が考えられる．よって，スレッド制御機構の動作を制御するために，各スレッドに制御レジスタを用意する．

スレッドの状態遷移として，スレッドを RMT PU のハードウェアコンテキストに割り当てる（Active 状態に遷移する）場合と RMT PU のハードウェアコンテキストからコンテキストキャッシュに退避する（Not Active 状態に遷移する）場合がある．そこで，それぞれの状態遷移を制御するためのフラグを用意する．図 5.5 にスレッド制御レジスタを示す．

各ビットの機能は以下のとおりである．

- Keep Active Bit（KA）

  Stop 状態になったスレッドはこれ以上実行を続けることができない．そのため別のスレッドを実行するために， スレッド入れ替え機構は Stop 状態のスレッドをコンテキストキャッシュに退避する．しかし，Stop 状態のスレッドに CPU を割り当てておくことにより，次にスレッドが起動したときに直ちに実行を開始することができる．このビットに 1 をセットすることにより，そのスレッドは RMT PU のハードウェアコンテキストからの退避の対象にはならず，Stop 状態や，より高い優先度のスレッドが実行可能になった場合でも，RMT PU のハードウェアコンテキストに割り当てられたままになる．

- Auto Active Bit（AA）

  このビットに 1 をセットすることにより，そのスレッドはハードウェアによる制御の対象となり，ハードウェアが自動的にスレッドの状態を制御して，コンテキストスイッチを行う．0 がセットされている場合は，そのスレッドはハードウェアによる制御が行われないため，スレッドの実行はソフトウェアで制御する必要がある．

リアルタイム処理では，実行可能なスレッドの中から最も高い優先度のスレッドを選択して実行することにより，各スレッドの時間制約を守る．よって，スレッド入れ替え機構のスレッド選択部では，実行可能なスレッドの中で優先度の高いスレッドから順番に RMT PU のハードウェアコンテキストに割り当てられるように，コンテキストスイッチを行うスレッドを選択する．

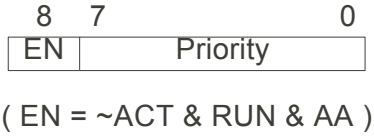

図 5.6: Thread Information Table のポイントの生成

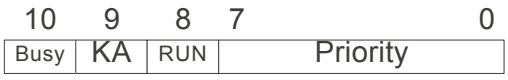

図 5.7: CPU Information Table のポイントの生成

Thread Selector はコンテキストキャッシュの中から RMT PU のハードウェアコンテキストに復帰するスレッドを選択する．実行可能なスレッドの中で，優先度の高い順に RMT PU のハードウェアコンテキストを割り当てるために，Thread Selector は毎クロック Thread Information Table を調べ，Not Active / Run（実行可能）状態のスレッドの中から最も優先度の高いスレッドを選択する．実際にはハードウェアが複雑になることを防ぐために，図 5.6 に示すポイントをコンテキストキャッシュに保持しているスレッド全てに対して生成し，比較器を用いて最も高いポイントを持つスレッドを選択する．

EN はそのスレッドが入れ替えの対象になるかどうかを示している．ACT は 5.2.2 節で述べたスレッドの状態のうち Active / Not Active 状態を示すビット，RUN はスレッドの状態のうち Run / Stop 状態を示すビット，AA はスレッド制御レジスタの Auto Active Bit，Priority はスレッドに付与されている優先度を示す．このようにして生成したポイントのうち，最大のものを選択することにより，Not Acrive / Run 状態のスレッドの中で最も優先度の高いスレッドが選択される．Thread Selector は選択したスレッドのスレッド ID と優先度を Compare Unit へと送る．

CPU Selector はハードウェアコンテキストに割り当てられているスレッドの中からコンテキストキャッシュに退避するスレッドを選択する．実行可能なスレッドの中で，優先度の高い順に RMT PU のハードウェアコンテキストを割り当てるために，CPU Selector は毎クロック CPU Information Table を調べ，優先度の最も低いスレッドを選択する．実際にはハードウェアが複雑になることを防ぐために図 5.7 に示すポイントを RMT PU のハードウェアコンテキストに割り当てられているスレッド毎に生成し，比較器を用いて最もポイントの低いエントリを選択する．

Busy はそのハードウェアコンテキストにスレッドが割り当てられているかどうか，KA はスレッド制御レジスタの Keep Active Bit，RUN はスレッドの状態のうち Run / Stop 状

態を示すビット，Priorityはそのハードウェアコンテキストに割り当てられているスレッドの優先度を示す．

このようにして生成したポイントのうち，最小のものを選択することにより，まず，スレッドが割り当てられていないハードウェアコンテキストがある場合はそのハードウェアコンテキストが選択される．全てのハードウェアコンテキストにスレッドが割り当てられている場合，Stop状態のスレッドが使用しているハードウェアコンテキストが選択される．複数のスレッドがStop状態の場合は，優先度の最も低いスレッドが使用しているハードウェアコンテキストが選択される．全てのスレッドがRun状態の場合は，最も優先度の低いスレッドが使用しているハードウェアコンテキストが選択される．CPU Selectorは選択したハードウェアコンテキストの番号と，そのハードウェアコンテキストで実行されているスレッドの状態，優先度をCompare Unitへと送る．

Compare UnitはThread SelectorとCPU Selectorから送られてきた情報をもとに，最終的にコンテキストスイッチを行うかどうかを判断する．Compare UnitはCPU Selectorが選択したハードウェアコンテキストの状況により，次のように動作する．

CPU Selectorが選択したハードウェアコンテキストに何もスレッドが割り当てられていない場合，そのハードウェアコンテキストにスレッドを割り当てることができるため，Thread Selectorの選択したスレッドをCPU Selectorが選択したハードウェアコンテキストに割り当てる．Compare UnitはThread Selectorの選択したスレッドのスレッド IDと復帰（Restore）命令をSwitch Unitに対して送る．

CPU Selectorが選択したハードウェアコンテキストを使用しているスレッドがStop状態の場合，ハードウェアコンテキストに割り当てられているスレッドは実行が完了しているため，別の実行可能なスレッドと入れ替える必要がある．Compare UnitはThread Selectorの選択したスレッドのスレッド IDとCPU Selectorの選択したハードウェアコンテキストを使用しているスレッドのスレッド ID，および入れ替え（Swap）命令をSwitch Unitに対して送る．ただし，CPU Selectorが選択したスレッドのスレッド制御レジスタのKeep Active Bitに1がセットされている場合，Switch Unitへの命令発行は行わない．

CPU Selectorが選択したハードウェアコンテキストを使用しているスレッドがRun状態の場合，ハードウェアコンテキストに割り当てられてるスレッドはまだ実行中であるため，優先度による比較を行う必要がある．Compare UnitはCPU Selectorから送られてきたスレッドの優先度とThread Selectorから送られてきたスレッドの優先度の比較を行う．Thread Selectorから送られてきたスレッドの優先度の方が高ければ，より優先度の高いスレッドが実行可能になったと判断する．よってCompare UnitはThread Selectorの選択したスレッドのスレッド IDとCPU Selectorの選択したハードウェアコンテキストを使用しているスレッドのスレッド ID，および入れ替え（Swqap）命令をSwitch Unitに対して送

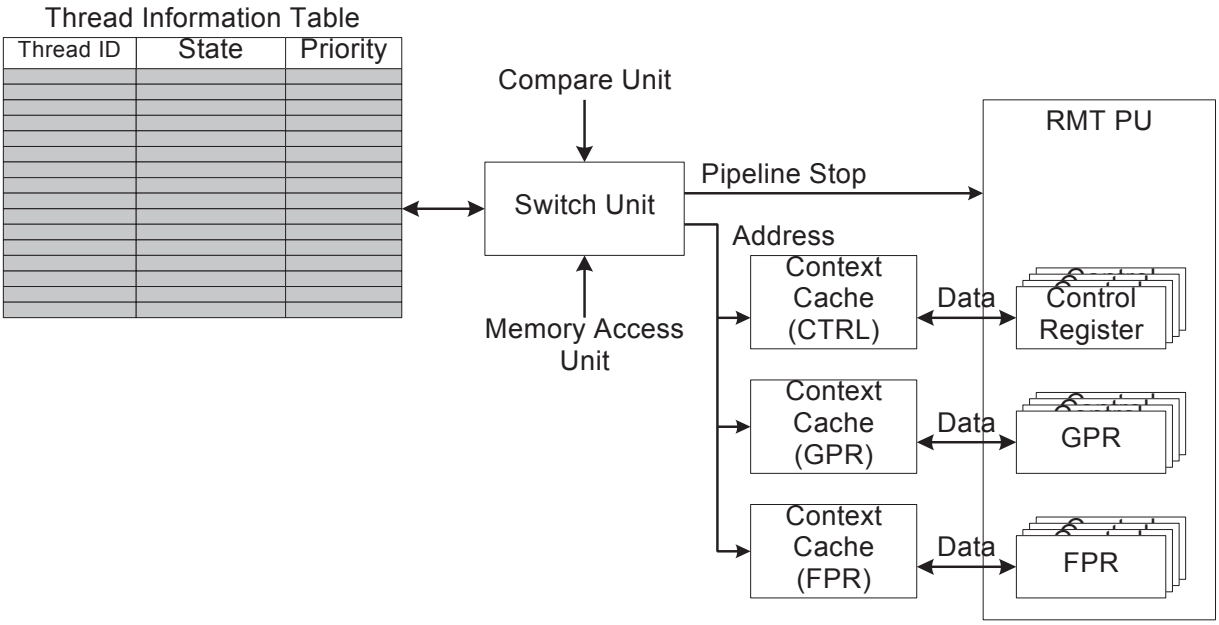

図 5.8: スレッド入れ替え機構の入れ換え制御部

る．Thread Selectorから送られてきたスレッドの優先度の方が低ければ，優先度の高い順にハードウェアコンテキストにスレッドが割り当てられているため，スレッドを入れ替える必要はない．よってCompare Unitはなにも命令を発行しない．ただし，CPU Selectorが選択したスレッドのスレッド制御レジスタのKeep Active Bitに1がセットされている場合，Thread Selectorから送られてきたスレッドの優先度の方が高い場合でも，Switch Unitへの命令発行は行わない．

以上の機構により，スレッド入れ替え機構のスレッド選択部はRun状態のスレッドの中で優先度の高いスレッドから順番にRMT PUのハードウェアコンテキストに割り当てる．

コンテキストキャッシュとRMT PUのハードウェアコンテキストの間のコンテキストの入れ替えはSwitch Unitが制御する．図5.8にスレッド入れ替え機構の入れ換え制御部を示す．

Switch UnitはCompare Unitからスレッド制御（入れ替えまたは復帰）命令，および対象とするスレッドのスレッドIDを受け取ると，該当ハードウェアコンテキストのパイプラインを停止する．送られてきたスレッドのスレッドIDから，そのスレッドが格納されているコンテキストキャッシュ上のアドレスを計算する．図5.1に示した専用バスを介してコンテキストキャッシュと各レジスタのデータを入れ替える．復帰命令の場合はコンテキストキャッシュから各レジスタにデータが送られるが，各レジスタからコンテキストキャッシュへのデータ転送は行わない．データの転送が終了すると，該当ハードウェアコンテキストのパイプラインの停止を解除し，スレッドの実行を開始する．また同時にThread

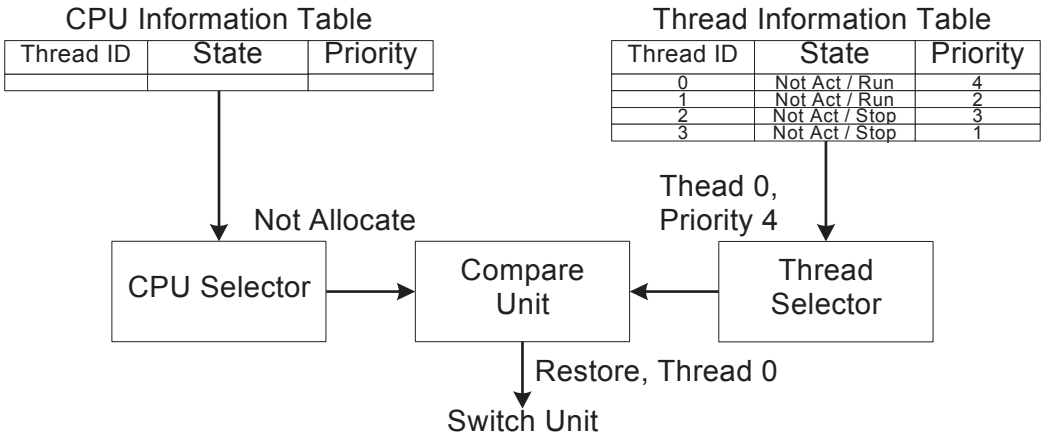

図 5.9: スレッド切り替え機構によるスレッド制御（1）

Information Table，CPU Information Tableの該当するスレッドの状態を変更する．

Switch Unitは Compare Unitからのスレッド制御命令の他に，5.2.2 節で述べたソフトウェアによる制御命令を処理する．ソフトウェアによる制御命令は RMT PU のメモリアクセスユニット（Memory Access Unit）から送られてくる．Compare Unit とメモリアクセスユニットから同時に命令が送られた場合，ソフトウェアによる明示的な命令の方が重要度が高いと考え，Switch Unitはメモリアクセスユニットからの命令，つまりソフトウェアによるスレッド制御命令を先に処理する．

図 2.2 を例にスレッド入れ替え機構の動作を説明する．ただし RMT PU は 8 つのハードウェアコンテキストがあるが，話を簡略化するためにここではハードウェアコンテキストを 1 つとする．

図 2.2 では，EDF スケジューリングが行われているため，デッドラインまでの時間が短かいスレッドから順番に高い優先度が与えられる．よって，Thread 0，Thread 2，Thread 1，Thread 3 の順番で高い優先度が与えられる．

最初に Thread 0 と Thread 1が起動し，Not Active / Run（実行可能）状態となる．Thread 2 と Thread 3 は Release Time がきていないため Not Active / Stop（待ち）状態となる．Thread Selectorは Not Active / Run 状態のスレッドのうち優先度の最も高いスレッドを選択するため，Thread 0 が選択されて，Compare Unit へと送られる．最初の状態ではハードウェアコンテキストにはスレッドが割り当てられていないため，CPU Selector はハードウェアコンテキストにスレッドが割り当てられていないことを Compare Unit へ伝える．Compare Unitは CPU Selectorからハードウェアコンテキストにスレッドが割り当てられていないことが伝えられるため，Thread 0 をハードウェアコンテキストに復帰するように Switch Unit に復帰命令を発行する（図 5.9）．これにより，Thread 0 は Active / Run（実

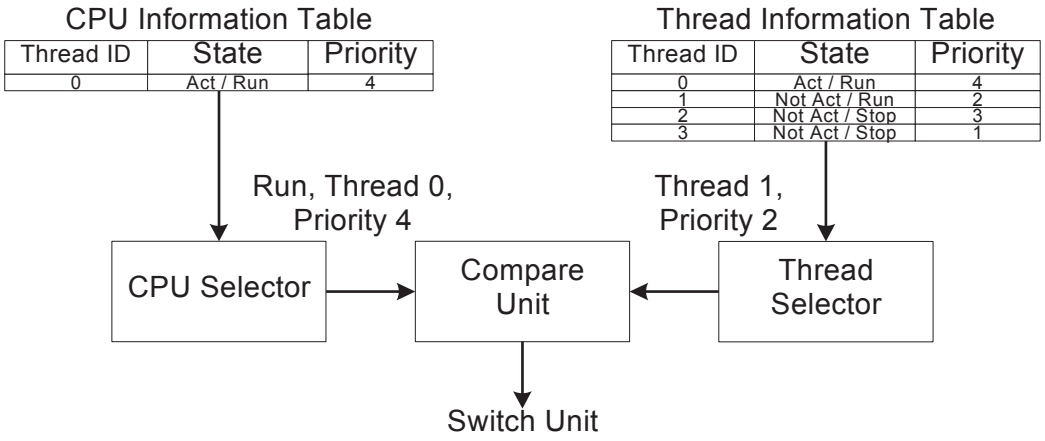

図 5.10: スレッド入れ替え機構によるスレッド制御（2）

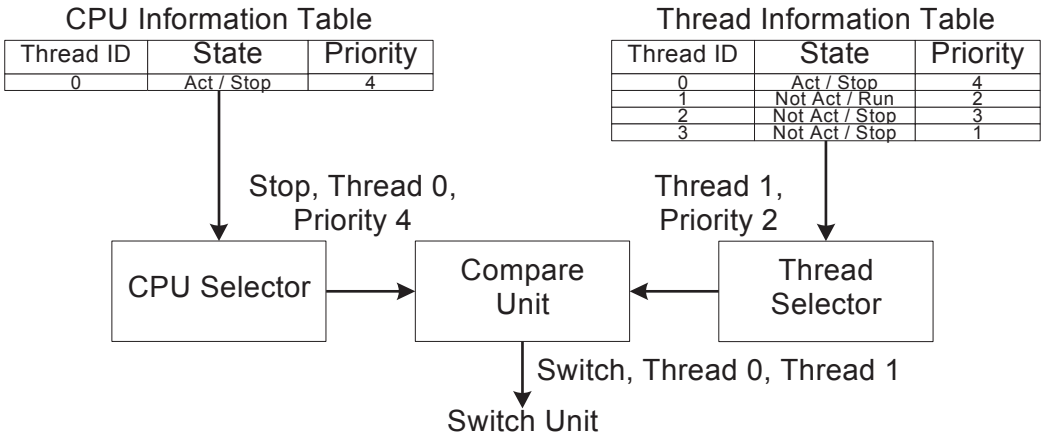

図 5.11: スレッド入れ替え機構によるスレッド制御（3）

行）状態へと遷移し，ハードウェアコンテキストが割り当てられて処理を開始する．

Thread 0 の実行を開始すると，CPU Selector は（ここではハードウェアコンテキストが 1 つとしているため）Thread 0 の情報を Compare Unit に送る．Thread Selector は Not Actrive / Run（実行可能）状態のスレッドのうち，最も優先度の高い Thread 1 の情報を Compare Unit に送る．CPU Selector から送られてきたスレッドは Run 状態のため，Compare Unit は CPU Selector から送られてきたスレッド（Thread 0）の優先度と Thread Selector から送られてきたスレッド（Thread 1）の優先度を比較する．Thread 1 の優先度は Thread 0 の優先度よりも低いため，Compare Unit は何も行わない（図 5.10).

Thread 0 の実行が完了すると，Thread 0 の状態は Active / Stop 状態へと遷移する．CPU Selector はスレッドが Stop 状態であることを Compare Unit に伝える．ハードウェアコンテキストに割り当てられているスレッドが Stop 状態のため，Compare Unit は優先度の比

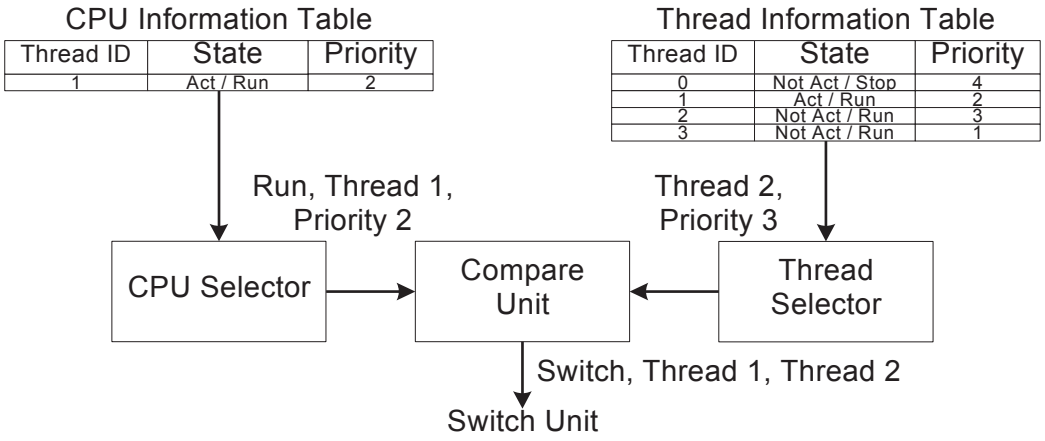

図 5.12: スレッド入れ換え機構によるスレッド制御（4）

較を行わずに，CPU Selector から送られてきたスレッド（Thread 0）と Thread Selector から送られてきたスレッド（Thread 1）を入れ替えるように Switch Unit に入れ換え命令を発行する．Thread 0 と Thread 1 が入れ替えられ，Thread 1 の処理が開始される．このとき Thread 0 は Not Active / Stop 状態，Thread 1 は Active / Run 状態へと遷移する（図 5.11）．Thread 1 の実行を開始すると，CPU Selector は Thread 1 の情報を Compare Unit に送る．

次に Thread 3が起動し，Not Active / Run 状態になる．Thread Selector は Thread 3 の情報を Compare Unit へと送る．CPU Selector の選択したスレッド（Thread 1）は Run 状態のため，Compare Unit は優先度の比較を行う．CPU Selector の選択したスレッド（Thread 1）よりも Thread Selector の選択したスレッド（Thread 3）の優先度の方が低いため，Compare Unit は何もしない．

Thread 2 の Release Time になると，Thread 2が起動し，Not Active / Run 状態になる．Thread Selector は Not Active / Run 状態のスレッドの中で最も優先度の高い Thread 2 を選択し，Compare Unit へと送る．CPU Selector の選択したスレッド（Thread 1）は Run 状態のため，Compare Unit は優先度の比較を行う．Thread 2 の優先度は Thread 1 の優先度よりも高いため，Compare Unit は Thread 1 と Thread 2 を入れ替えるように Switch Unit に指示を出す（図 5.12）．これに Thread 1 と Thread 2が入れ替えられて，Thread 2 の処理が開始される．このとき Thread 1 は Not Active / Run 状態，Thread 2 は Active / Run 状態へ遷移する．スレッド入れ替え機構により，より優先度の高いスレッドが起動するとハードウェアにより直ちにスレッドの入れ替えが行われる．

Thread 2 の実行が開始されると，CPU Selector は Thread 2 の情報を Compare Unit に送る．Thread Selector は Not Active / Run 状態のスレッドの中で最も優先度の高い Thread

1を選択してCompare Unitに送る．Compare Unitは優先度を比較し，Thread 1の優先度の方がThread 2よりも低いため，Compare Unitは入れ替えを行わない．

Thread 2の実行が完了するとThread 2はActive / Stop状態へと遷移する．よってCompare Unitは優先度の比較を行わずにThread 2とThread 1を入れ替える．これにより，Thread 1の実行が再開される．

以上のようにスレッド入れ替え機構はRun状態のスレッドの中から優先度の高い順にRMT PUのハードウェアコンテキストを割り当てて実行するように制御を行う．より優先度の高いスレッドが起動すると，Thread Selectorがそのスレッドを選択し，直ちにコンテキストスイッチが行われるため，優先度の高いスレッドは直ちにハードウェアコンテキストに割り当てる．スレッドのコンテキストはコンテキストキャッシュに保持されているため，低レイテンシで入れ替えられる．スレッドの入れ替えはハードウェアで自動的に処理されるため，ソフトウェア（スケジューラ）の介入無しに優先度に従ってスレッドを実行することができる．

5.2.4 割り込み制御機構

リアルタイム処理では，スレッドの起動（待ち状態から実行可能状態への遷移）は，主に割り込みによって実現する．例えば，I/Oの処理を待っているスレッドはI/Oからの割り込みによって，I/Oの処理が完了したことを知る．周期的に起動するスレッドは，タイマ割り込みによって一定間隔で起動する．

2.4節で述べたように，リアルタイムシステムでは割り込みに対して素早く応答する必要がある．従来の手法では，割り込みが発生した場合，ソフトウェア（割り込みハンドラ）により該当スレッドを起動する．割り込みハンドラは割り込み要因からどのスレッドを起動するか調べ，コンテキストスイッチを行うことによりスレッドを起動するため，オーバーヘッドが生じ，割り込み対して素早く応答することができない．本研究では，ハードウェアでスレッドの状態を管理しているため，割り込みによるスレッドの起動を，ソフトウェアを介さずハードウェアで実現することにより，割り込みに対する応答性を向上する．

図5.13に割り込み制御機構のブロック図を示す．RMT Processorでは，割り込みの要因は割り込みレベル（IRQ）によって区別される[伊藤 他]．スレッド毎に起動要因となる割り込みの種類は異なるため，各スレッドでどの割り込みレベルを受けつけるかを指定するレジスタ（Mask）を用意する．このレジスタに起動する要因となる割り込みのレベルをビットマップで指定する．

割り込みが発生した場合，Interrupt Controllerはコンテキストキャッシュ内の全てのスレッドに対して以下を実行する．まず，Thread Information TableのMaskレジスタの値

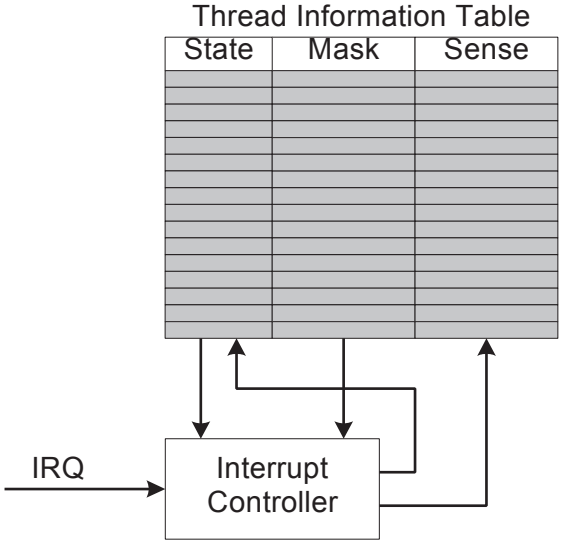

図 5.13: 割り込み制御機構

と割り込みレベル（IRQ）と比較し，受け付けるべき割り込みかどうかを判断する．受け付けるべき割り込みの場合，Interrupt Controllerはスレッドの状態（State）を調べる．スレッドがActive状態の場合，そのスレッドはハードウェアコンテキストが割り当てられているため，発生した割り込みを直ちにRMT PUへ伝える．スレッドがNot Active状態の場合，そのスレッドはハードウェアコンテキストが割り当てられていないため，直ちに割り込み処理を開始することができない．この場合，Interrupt Controllerはそのスレッドの Senseレジスタに発生した割り込みのレベルを保存する．Senseレジスタの内容は，そのスレッドがハードウェアコンテキストを割り当てられたときにRMT PUへ伝えられ，割り込み処理を開始するために使われる．Interrupt Controllerはさらにスレッドの Run / Stop状態を調べる．スレッドがStop状態の場合はRun状態へと遷移させる．これにより，スレッドの状態はNot Active / Run状態となりスレッドが起動する．スレッドが起動すると，5.2.3節で述べたスレッド入れ替え機構により，優先度に従って適切なタイミングでコンテキストスイッチが行われる．割り込みにより起動したスレッドの優先度がRMT PUで実行しているスレッドの優先度よりも高い場合，スレッド入れ替え機構により直ちにコンテキストスイッチが行われ，起動したスレッドにRMT PUのハードウェアコンテキストが割り当てられる．起動したスレッドの優先度がRMT PUで実行しているスレッドの優先度よりも低い場合，スレッド入れ替え機構はすぐにはスレッドを入れ替えず，優先度の高いスレッドの実行が終了した後にコンテキストスイッチを行って，割り込みによって起動したスレッドを実行する．

スレッドによっては，ソフトウェアによる実行の制御を行うために，ハードウェアによ

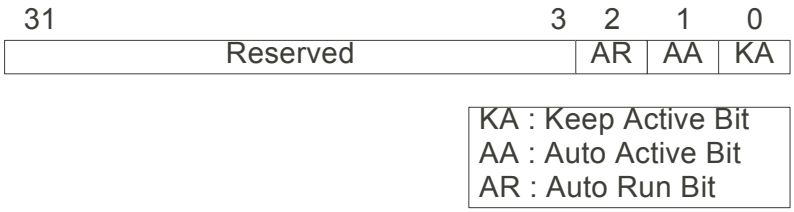

図 5.14: スレッド制御レジスタ（2）

るスレッドの割り込み制御を抑制したい場合が考えられる．よってスレッド制御レジスタに 1 ビット付加し，ハードウェアによるスレッドの割り込み制御の有効 / 無効を制御する．

図 5.14 にスレッド制御レジスタを示す．KA bit および AA bit は 5.2.3 節で述べたとおりである．

- Auto Run Bit (AR)

  割り込みにより，スレッドの自動起動を行うか行わないかを制御する．このビットに 1 をセットすることにより，Mask レジスタで指定した割り込みが発生した場合にスレッドを Run 状態へと遷移させる．0 がセットされている場合は，割り込みによる自動起動を行わない．Mask レジスタで指定した割り込みが発生した場合，Interrupt Controller はスレッドの状態を変更しないが，割り込みが発生したことは記録しておかなければならないため，Sense レジスタに割り込みがかかったことを記録する．

5.2.5 タイマ

リアルタイム処理では時間を扱う．ハードウェアで時間を扱う場合，タイマを用いる．例えば周期スレッドは一定時間毎に発生するタイマ割り込みによって起動する．

従来のプロセッサでは，タイマの数は周期スレッドの数よりも少ないため，複数の周期スレッドでタイマを共有して使用する．この場合，それぞれのスレッドで起動する周期が異なるため，タイマ割り込みが発生する度に，タイマハンドラがスレッド毎に起動するかしないかを判別する必要がある．本研究では割り込み機構により，割り込みが発生した場合にハードウェアで自動的にスレッドを起動する機能を持つ．しかし，タイマが共有された場合，この割り込み制御機構を有効に活用することができないと考えられる．よって本研究では全てのスレッドに対して 1 つづつタイマを用意する．

図 5.15 にタイマの設定レジスタを示す．このタイマは指定した時間によってスレッドを自動的に起動する役割を持つ．1 つのスレッドに対して 1 つのタイマを持つことにより，スレッドの実行周期が異なる場合においても，ソフトウェアを介さずに割り込み制御機構

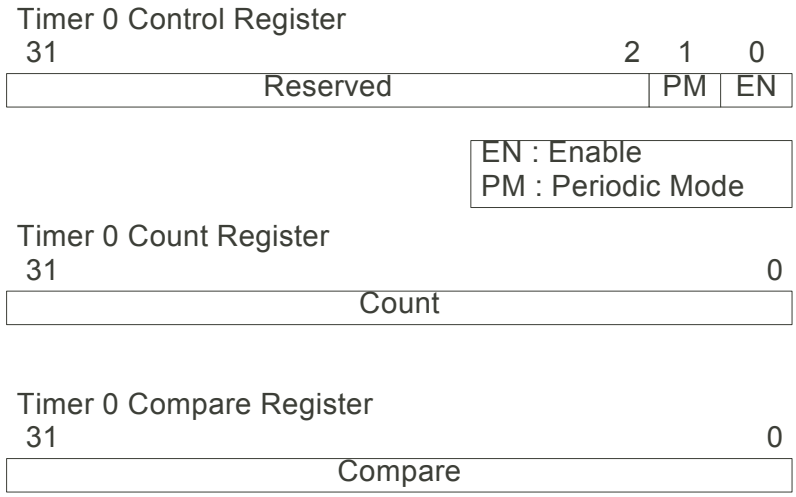

図 5.15: タイマ（Timer 0）の設定レジスタ

により，スレッドを起動することが可能になる．タイマは一定時間毎に割り込みを発生することにより，周期スレッドを一定時間毎に起動する．また，相対的な起動時間（Release Time）がわかっているスレッドに対して，指定した時間後に割り込みを発生することにより，Release Time にスレッドを自動的に起動する．

Control Register の各ビットの機能は以下のとおりである．

- Enable（EN）

  1 をセットするとタイマを起動し，カウントを開始する．

- Periodic Mode（PM）

  1 をセットすると Periodic Timer となり，指定された時間毎に割り込みを発生させる．0 をセットすると One Shot Timer となり一度割り込みを発生させた後，タイマは停止する．周期スレッドの場合，Periodic Timer を使用することにより，一定時間毎に割り込みを発生させて，スレッドを周期的に起動する．Releas Time を指定してスレッドを起動する場合，One Shot Timer を用いる．

Compare Register には，割り込みを発生させるまでの時間を指定する．周期スレッドの場合は実行周期，Release Time に自動的に起動する場合は Release Time までの相対時間を指定する．タイマが起動する（Control Register の Enable Bit に 1 がセットされる）と，Count Register の値がカウントアップされる．Count Register と Compare Register の値が等しくなったときに割り込みが発生する．各スレッド毎にタイマを用意しているため，割り込みは対応するスレッドのみにかかる．割り込みが発生すると，5.2.4 節で述べた割り込み制御機構がスレッドを自動的に起動する．

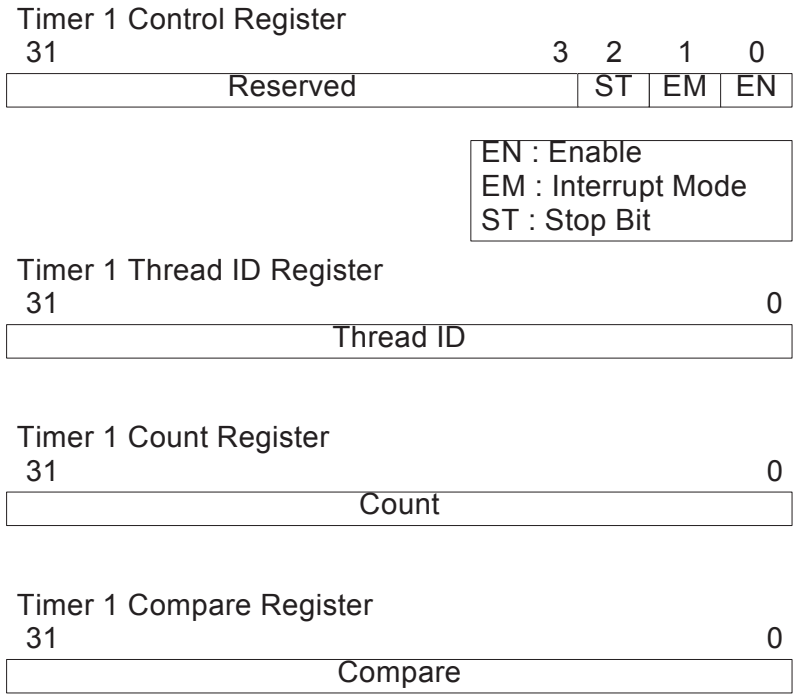

図 5.16: デッドラインタイマ（Timer 1）の設定レジスタ

Control Register の Periodic Mode に 1 がセットされている場合，割り込みが発生すると同時に Count Register の値がクリアされ，再びカウントアップを開始する．Periodic Mode に 0 がセットされている場合，Count Register の値がクリアされ，Control Register の Enable がネゲートされ，タイマが停止する．

リアルタイム処理では，あるスレッドがデッドラインミスを起こした場合，そのスレッドを適切に処理しないと，他のスレッドの実行に影響を与えてしまう．よってデッドラインミスを早期に検知し，適切に処理を行う機構が必要となる．例えばインプリサイス計算モデル [Lin *et al.* 87] では，デッドラインミスが発生する場合，付加部分の実行を切り捨てることより，デッドラインミスによる影響を防ぐ．従来のリアルタイム処理では，カウンタやタイマを用いてソフトウェアでデッドラインミスが起ったかどうかを判断する．ソフトウェアで処理を行うため，処理のオーバーヘッドが生じ，直ちにデッドラインミスを検出することができない．よって本研究ではデッドラインミスを検出するタイマを設計し，早期にデッドラインミスを検出する．

デッドラインミスを検出するタイマは，スレッドが Run 状態になると自動的にカウンタをカウントアップして計測を始め，スレッドが Stop 状態になると自動的にカウンタをリセットする．指定した時間が経過してもスレッドが Stop 状態に遷移しない場合，デッドラインミスが発生したと判断し，割り込みを発生させる．

図5.16にデッドラインタイマの設定レジスタを示す．このタイマはデッドラインミスを検出するために用いる．Control Registerの各ビットの機能は以下のとおりである．

- Enable（EN）

  1をセットするとタイマが有効になる．その状態でスレッドがRun状態になると計測を開始する．

- Interrupt Mode（IM）

  デッドラインミスが発生した場合，デッドラインミスを起こしたスレッド自身がデッドラインミスの処理を行う場合と，OSなどの他のスレッドがデッドラインミスの処理を行う場合が考えられる．Interrupt Modeにより，どのスレッドに割り込みを発生させるかを指定する．Interrupt Modeが0の場合，自分自身に対して割り込みを発生させる．1がセットされている場合，Thread ID Registerで指定したスレッドに割り込みを発生させる．

- Stop Bit（ST）

  デッドラインミスが発生した場合，Interrupt Modeに1がセットされていると，他のスレッドに対して割り込みが発生する．この場合，デッドラインミスを起こしたスレッドの実行を続けるかどうかを設定する．0がセットされている場合，デッドラインミスが発生した後もスレッドの実行を続ける．1がセットされている場合，デッドラインミスが発生すると，スレッドをStop状態へと遷移させる．Interrupt Modeに0がセットされている場合は，このビットは無視される．

Thread ID RegisterにはControl RegisterのInterrupt Modeが1の場合に割り込みを発生させるスレッドのスレッドIDを指定する．Compare Registerには，割り込みを発生させるまで（デッドラインまで）の時間を指定する．

Control RegisterのEnable Bitが1の場合，スレッドがRun状態に遷移すると自動的にCount Registerの値をカウントアップする．スレッドがRun状態からStop状態に遷移するとCount Registerの値をクリアし，カウントアップを停止する．Count RegisterとCompare Registerの値が等しくなった場合（設定した時間Run状態が続いた場合），デッドラインミスが発生したことになるため，割り込みを発生させる．Control RegisterのInterrupt ModeとThread ID Registerによって，自分自身または指定したスレッドに対して割り込みが発生する．これにより，デッドラインミスを直ちに検出し，後処理を開始することが可能となる．

表 5.1: スレッド制御機構の論理合成結果

	回路規模（μm^2）	最大遅延（ns）
Thread Unit	1,287,968.375000	3
Context Cache（Control Register）	151,995.343750	3
Context Cache（General Purpose Register）	274,499.968750	3
Context Cache（Floating Point Register）	137,334.000000	3
Total	1,851,797.687500	3

5.3 評価

本研究で設計したスレッド制御機構を Verilog-HDL を用いて実装した．本節では本研究で設計したスレッド制御機構の評価を行う．スレッド制御機構はまだ実機には搭載されていないため，RTL を用いた論理シミュレーションにより評価を行う．

5.3.1 論理合成

RMT Processor は ASIC への実装を予定している．そこで 0.13μm プロセスを用い，Synopsys 社の Design Compiler でスレッド制御機構の論理合成を行った．RMT Processor は 300MHz の動作を目標としている．この条件を満たすために回路のタイミング制約を 3ns とした．また，回路規模は小さければ小さいほど良いため，回路規模の制約条件は大きさ 0 とした．以上の制約条件を与えて論理合成を行った．

表 5.1 に論理合成の結果を示す．Thread Unit はスレッドの制御を行うユニットで，スレッド入れ替え機構や割り込み機構の他にソフトウェアによるスレッド制御命令を実行するための回路を含む．Context Cache は各レジスタ用のコンテキストキャッシュで主に SRAM 回路で構成されている．

回路規模はセルの総面積を表している．RMT Processor は 0.13μm プロセスで 1cm 角のチップ面積で実装を行う．1cm 角の中に I/O パッドなどを配置しなければならないため，実際に使用できる面積は 8.5mm 角程度となる．通常，配線などを考えるとチップ内にセルを 100%配置することはなく，50%から 60%程度しか配置することはできない．配置の割り合いを 60%とすると，スレッド制御機構の大きさはチップ全体の 2.5%となり，回路規模の面では十分実装可能である．

最大遅延は制約条件通り 3ns となった．よってスレッド制御機構は 333MHz で動作することができるため，目標の周波数を達成している．

表 5.2: コンテキストスイッチのオーバーヘッド

入れ換え方法	入れ換え時間
スレッド制御機構	7 クロック
ソフトウェア	1558 クロック

5.3.2 コンテキストスイッチのオーバーヘッド

スレッドを入れ替える場合に発生するコンテキストスイッチのオーバーヘッドを評価する．RMT PU はパイプライン処理を行っているため，命令がオーバーラップして実行される．よってコンテキストスイッチにより入れ替える前のスレッドの最後の命令がコミットされてから，次に実行するスレッドの命令フェッチが始まるまでをコンテキストスイッチのオーバーヘッドとする．

本研究で提案したスレッド制御機構を用いてコンテキストスイッチを行った場合と，ソフトウェア（スケジューラ）により，Load / Store 命令を用いてコンテキストスイッチを行った場合について，コンテキストスイッチにかかる時間を測定した．

RMT Processor は 300MHz の動作周波数を目標としているが，現在，全体の論理合成の結果 100MHz での動作が可能となっている．よって，RTL によるシミュレーションでも RMT Processor の動作周波数は 100MHz とする．

表 5.2 に実行結果を示す．ソフトウェアによるコンテキストスイッチでは，Load / Store 命令を用いて外部のメモリに対して 1 つ 1 つコンテキストを入れ替える．一方，スレッド制御機構では，オンチップメモリであるコンテキストキャッシュに対して複数のデータを並列に退避，復帰するため，ソフトウェアによるコンテキストスイッチに比べて 0.4%の時間でコンテキストスイッチが完了している．

ソフトウェアによるコンテキストスイッチでは，コンテキストスイッチを行う前にどのスレッドと入れ替えを行なうかを優先度に従って決定する必要がある．スケジューラの実装により，次に実行するスレッドを決定するためのオーバーヘッドは異なるが，実際にリアルタイム処理を行う場合，コンテキストスイッチの時間にスレッド選択のオーバーヘッドが加算される．スレッド制御機構ではハードウェアが自動的に最も優先度の高いスレッドを選択してコンテキストスイッチを行うため，スレッド選択のオーバーヘッドは加算されない．よって実際にリアルタイム処理を行う場合はさらにオーバーヘッドの差が大きくなると考えられる．

シミュレーションの結果より，一回のコンテキストスイッチにかかる時間は，スレッド制御機構では 70ns，ソフトウェアでは 15.18μs となる．これをスレッドの実行時間（スレッ

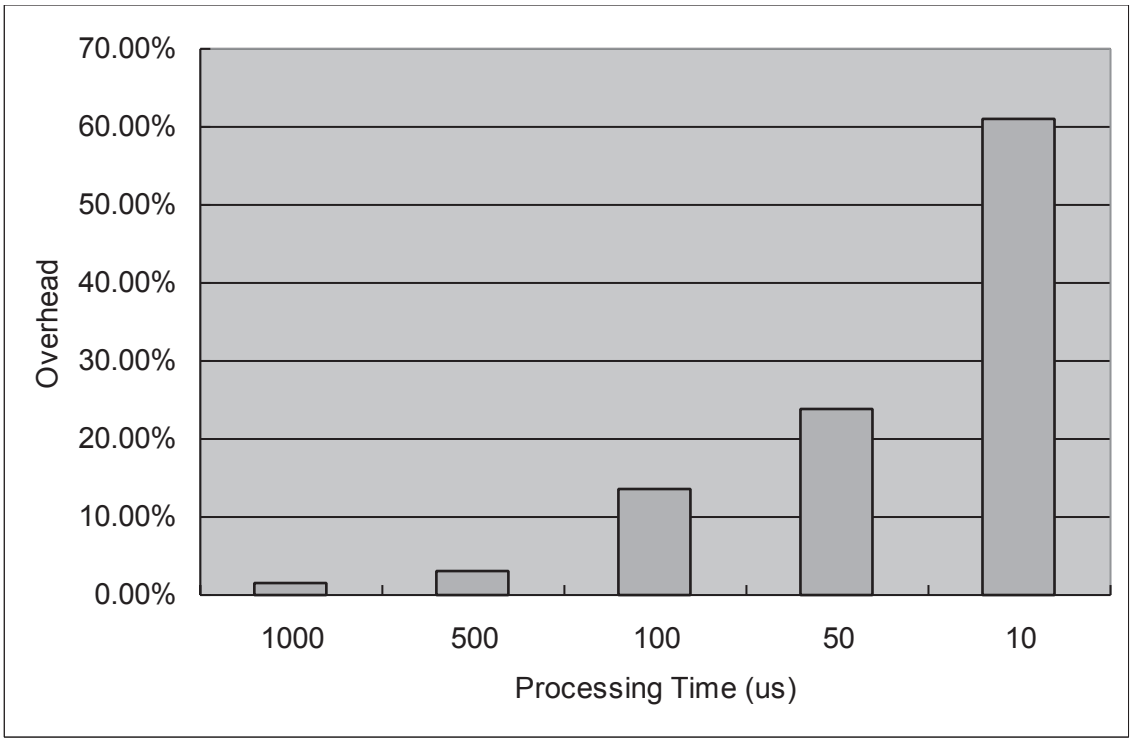

図 5.17: スレッドの実行時間に対するコンテキストスイッチの時間の割り合い（ソフトウェア）

ドが実際に処理を行うための時間にコンテキストスイッチの時間を加えた時間）に対する割り合いであらわすと，図5.17，5.18のようになる．

図5.17はソフトウェアでコンテキストスイッチを行った場合に，コンテキストスイッチにかかる時間の割り合いを示している．ソフトウェアでコンテキストスイッチを行った場合，スレッドの処理時間が1000μsと長い場合はコンテキストスイッチにかかる時間は全実行時間の1.36%と小さい．しかしスレッドの処理時間が10μsと短かくなるとコンテキストスイッチにかかる時間は実行時間全体の57.93%となり，実行時間の半分以上をコンテキストスイッチに費すことになる．このようなスレッドを実行すると，コンテキストスイッチに費す時間の割り合いが大きくなり，システム全体のスループットが低下する．

図5.18はスレッド制御機構を用いてコンテキストスイッチを行った場合のコンテキストスイッチにかかる時間の割り合いを示している．スレッド制御機構を用いてコンテキストスイッチを行った場合，スレッドの処理時間が10μsと短かい場合でもコンテキストスイッチにかかる時間は全実行時間の0.70%となり，オーバーヘッドは無視できるほど小さい．よってこのような短かい処理時間のスレッドを実行したとしても，コンテキストスイッチの費す時間の割り合いを抑えることができ，システム全体のスループットが向上する．

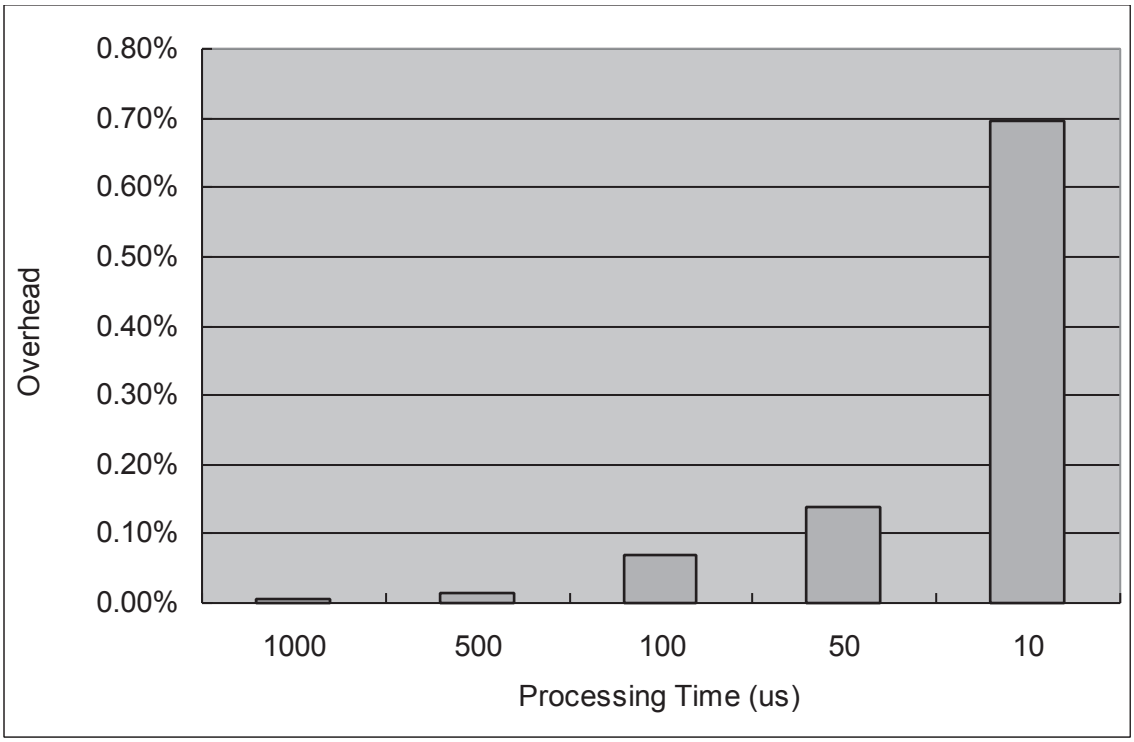

図 5.18: スレッドの実行時間に対するコンテキストスイッチの時間の割り合いスレッド制御機構）

表 5.3: 割り込み応答時間

入れ換え方法	応答時間
スレッド制御機構	6 クロック
ソフトウェア	5970 クロック

5.3.3 割り込み応答時間

リアルタイム処理では外部からの割り込みに対して素早く応答することが重要である．よって割り込みに対する応答時間を評価する．RMT PU はパイプライン処理を行っているため，割り込みが発生してから割り込み処理を行うスレッドの命令フェッチが始まるまでを割り込み応答時間とする．

スレッド制御機構を用いてスレッドを起動して割り込みに応答する場合と，ソフトウェアによりスレッドを起動して割り込みに応答する場合について，応答時間を測定する．

表5.3に実行結果を示す．ソフトウェアで割り込み応答を行う場合，割り込みが発生すると割り込みハンドラが起動し，割り込み要因が調べられ，その割り込みに応答するスレッ

ドを起動した後にコンテキストスイッチを行ってスレッドの実行を開始する．ソフトウェアによる処理が増えるため，コンテキストスイッチを行うだけの場合に比べて処理時間が増加している．

スレッド制御機構を用いて割り込み応答を行った場合，ソフトウェアで割り込み応答を行った場合に比べて0.1%の時間で割り込みに対して応答した．スレッド制御機構では割り込みが発生した瞬間に，ハードウェアが割り込みレベルに応じてスレッドを自動的に起動し，スレッド入れ替え機構が優先度に従って直ちにコンテキストスイッチを行う．割り込み制御機構は1クロックでスレッドを起動するため，ソフトウェアによる割り込み応答に比べ，割り込みに対して素早く応答している．

5.3.4 スケジューリング可能性

複数のスレッドをRMT PUのハードウェアコンテキストとコンテキストキャッシュの間で入れ替えながら実行した場合のスケジューリング可能性について評価する．

処理時間が100μsのスレッドを実行した場合

処理時間が100μsのスレッドを優先度に従ってRMT PUのハードウェアコンテキストに割り当てて実行する場合を評価する．スレッドは先に生成しておき，スレッドIDを1から順番に付与する．スレッドIDが小さいスレッドに高い優先度を与える．生成したスレッドをスレッド制御機構を用いて入れ替えを行った場合と，ソフトウェアにより入れ替えを行った場合について評価を行った．

図5.19にソフトウェアによりスレッドを入れ替えた場合の実行結果，図5.20にスレッド制御機構を用いてスレッドを入れ替えた場合の実行結果を示す．図の横軸は時間経過，縦軸は各ハードウェアコンテキストにおける単位時間当たりの命令コミット数をあらわしている．それぞれのハードウェアコンテキストでの色の違いは，異なるスレッドを実行していることをあらわしている．

ソフトウェアによりコンテキストスイッチを行う場合，Load / Store命令によりコンテキストをメモリに対して退避，復帰する．メモリアクセスのレイテンシはRMT PUの優先度制御により隠蔽されているため，コンテキストスイッチを行いながら，より優先度の低いスレッドが実行されている．しかし，コンテキストスイッチを行うために大量のLoad / Store命令を実行する必要があり．このLoad / Store命令が命令スロットを占有するため，優先度の低いスレッドが命令スロットを割り当てられず，実行に影響が出ている．

スレッド制御機構を用いた場合，コンテキストスイッチはハードウェアにより自動的に行われるため，ソフトウェアでLoad / Store命令を実行する必要はない．よって，コンテキ

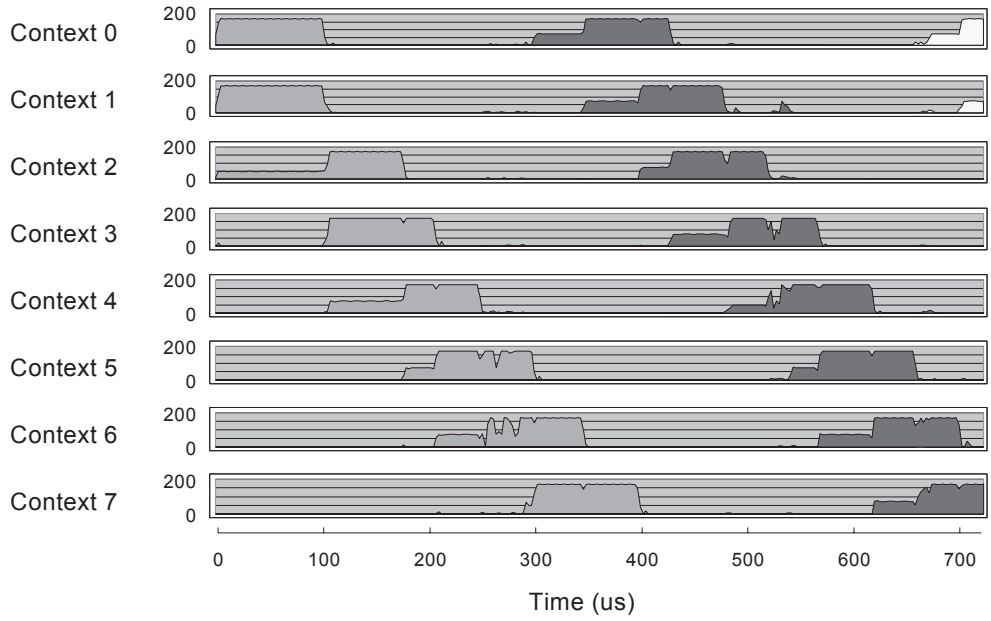

図 5.19: スレッドの処理時間が 100μs の場合の実行結果（ソフトウェア）

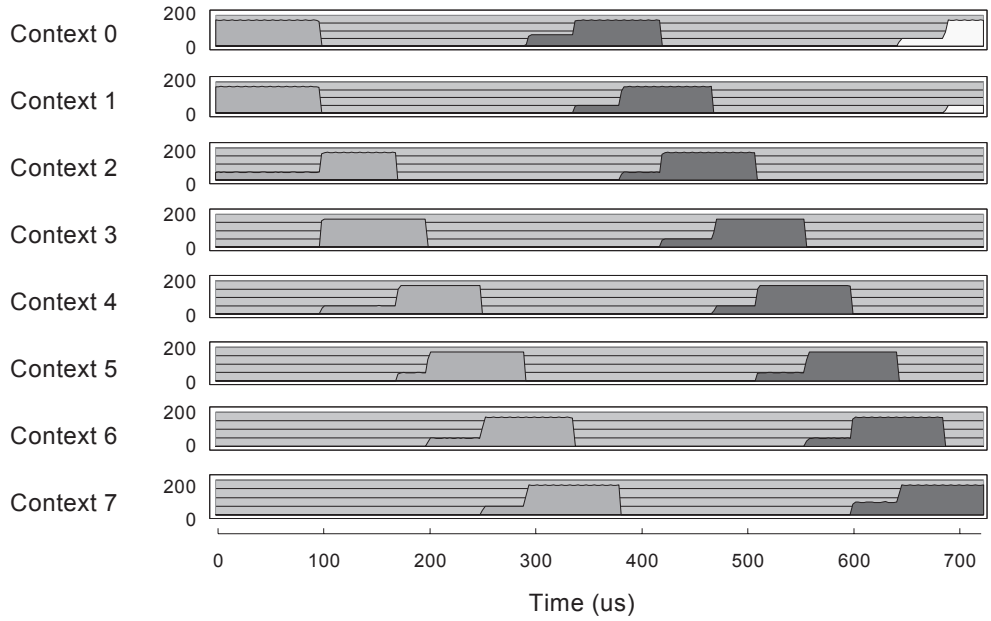

図 5.20: スレッドの処理時間が 100μs の場合の実行結果（スレッド制御機構）

ストスイッチにより高い優先度のスレッドが命令スロットを占有することがないため，ソフトウェアによるコンテキストスイッチと異なり，優先度の低いスレッドが影響を受けずに実行されている．

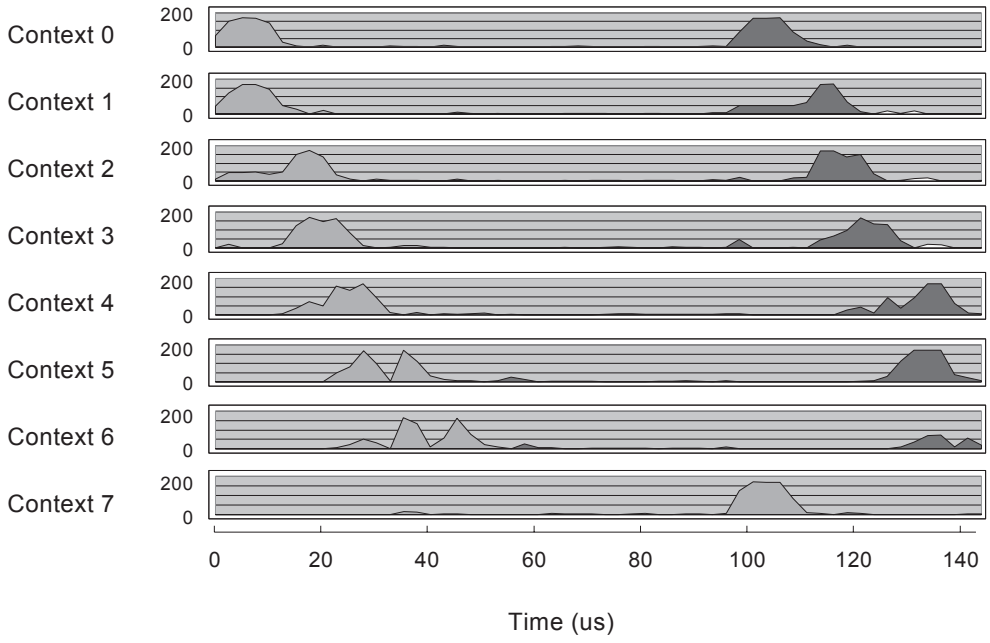

図 5.21: スレッドの処理時間が 10μs の場合の実行結果（ソフトウェア）

処理時間が 10μs のスレッドを実行した場合

処理時間が 10μs のスレッドを優先度に従って RMT PU のハードウェアコンテキストに割り当てて実行する場合を評価する．スレッド制御機構を用いて入れ替えを行った場合と，ソフトウェアにより入れ替えを行った場合について評価を行った．

図 5.21 にソフトウェアによりスレッドを入れ換えた場合の実行結果，図 5.22 にスレッド制御機構を用いてスレッドを入れ替えた場合の実行結果を示す．

処理時間が 10μs の場合，ソフトウェアによる入れ替えでは，スレッドの処理時間に対してコンテキストスイッチに費やす時間の割り合いが大きくなる．そのため，多くのスレッドがコンテキストスイッチを開始すると Load / Store 命令のレイテンシを隠蔽できなくなる．また，コンテキストスイッチに 15μs かかるため，スレッドが終了した後，RMT PU の優先度機構により別のスレッドを実行中に，次のスレッドを準備できず，スレッドが演算を行うことができない時間帯が存在している．

スレッド制御機構を用いた場合，コンテキストスイッチはハードウェアにより 7 クロックで行われる．RMT PU の優先度機構により別のスレッドを実行中にコンテキストスイッチを行って次のスレッドをハードウェアコンテキストに割り当てることができるため，ソフトウェアによる入れ替えと異なり，常にいずれかのスレッドが演算を行っている．これにより，ソフトウェアによる入れ替えに比べて同じ時間内に実行できるスレッドの数が増加している．つまり，ソフトウェアによる入れ替えに比べてスケジューリング可能性が増

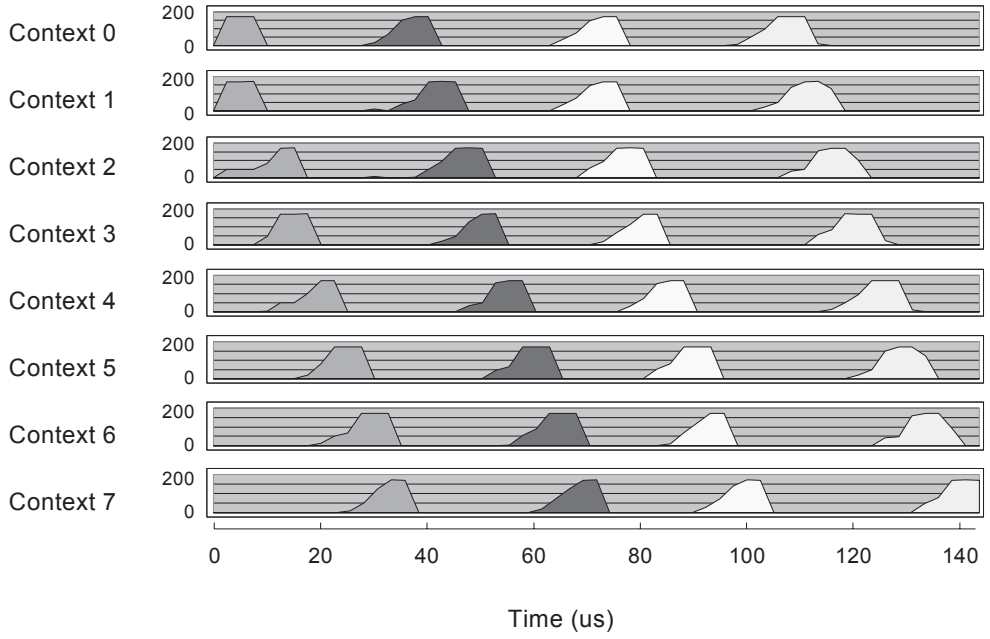

図 5.22: スレッドの処理時間が 10μs の場合の実行結果（スレッド制御機構）

```
for( ; ; ){
    a = input();
    e = a - x;          /* x : Target Value */
    ei += e;
    ed = e - e1;

    b = Kp * ( e + Ki * ei + Kd * ed );
    output(b + x);
    e1 = e;

    sleep();
}
```

図 5.23: PID 制御のプログラム例

加している．このようにスレッド制御機構では，より短かい時間粒度でスレッドを切り替える場合に効果が大きくなる．

例えばロボットシステムにおいて，PID 制御でモータを制御する場合，図 5.23 のような

プログラムを用いる．このようなプログラムは一周期分の処理時間が 1μs とさらに短かくなる．このようなスレッドを複数実行する場合，ソフトウェアによるコンテキストスイッチでは，コンテキストスイッチのオーバーヘッドの割り合いが大きくなる．しかし，スレッド制御機構を用いることにより，コンテキストスイッチのオーバーヘッドの割合を小ざくすることができ，ソフトウェアによるコンテキストスイッチに比べてより多くのスレッドをスケジューリングして実行することが可能となる．また，コンテキストスイッチにかかる時間が 70ns のため，RMT PU の優先度制御機構により，1μs の処理を行っている間にコンテキストスイッチが完了し，次に実行するスレッドの準備ができるため，RMT PU で絶えず演算を行うことが可能となる．

5.4 本章のまとめ

本章では，スレッドの切り替えを小さいオーバーヘッドで実現するための，スレッド制御機構について述べた．

スレッド制御機構では，コンテキストスイッチのオーバーヘッドを削減するために，コンテキストキャッシュ（オンチップメモリ）を用意し，ハードウェアで RMT PU のハードウェアコンテキストとコンテキストキャッシュの間でスレッドを入れ替える．スレッド入れ替え機構は，ハードウェアでスレッドの状態を管理し，優先度に従って入れ替えるスレッドを選択し，自動的にコンテキストキャッシュを行う．割り込み制御機構は，割り込み要因によってハードウェアがスレッドを自動的に起動する．また，二種類のタイマを全てのスレッドに用意し，周期スレッドの起動，およびデッドラインミスの検知をハードウェアのみで行う．

従来，主にソフトウェア（スケジューラ）によってスレッドの起動と実行が制御されていた．スレッド制御機構では，ハードウェアが自動的にスレッドを起動し，優先度に従って自動的にコンテキストスイッチを行い，スレッドを切り替えて実行する．つまり，スレッド制御機構を用いることにより，ソフトウェアはスレッドの起動や入れ換えなどの制御を行う必要がなくなる．ソフトウェアはスレッドに対して適切な優先度を与えるだけでリアルタイム処理を行うことが可能となる．例えば，RM スケジューリングのように，システムの起動前に静的にスケジューリングされ，実行時にスレッドの優先度が変わらないようなシステムでは，実行時にスケジューラを用いずにシステム内のスレッドを実行することが可能となる．

スレッド制御機構により，ソフトウェアでスレッドの制御を行う場合に比べてコンテキストスイッチのオーバーヘッドを 0.1%に削減した．コンテキストスイッチのオーバーヘッドが削減した分，より多くのスレッドを実行することができるようになり，スケジューリ

ング可能性が向上した．特にスレッドの実行時間が短かい場合でも，RMT PU の優先度機構により，他のスレッドが実行されている間にコンテキストスイッチを完了することができ，スケジューリング可能性が向上することを示した．本研究で提案したスレッド制御機構により，今まででは実現できなかった短い時間粒度でスレッドの制御を行うことが可能となった．

第6章

ベクトル演算機構

4.3節において，RMT PUがハードウェアコンテキストに保持しているスレッドに対して，優先度に従って命令スロットを割り当てていく機構について述べた．この機構のためにRMT PUでは，優先度の高いスレッドの命令が優先的に実行される．よって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割り当てられる命令スロットの数が減少する．本章では，このような状況において，ソフトリアルタイム処理のデータを演算するためのベクトル演算機構について述べる．そして本手法について評価を行い，本手法の有効性を示す．

6.1 基本方針

3.2節で述べたように，ソフトリアルタイム処理の大量のデータを演算するために，従来の研究では，主にデータ並列性に注目した機構が提案されている．それらは大きく分けるとSIMD演算とベクトル演算がある．

SIMD演算は図6.1に示すように，1つのレジスタに複数のデータを詰め込み，それらを並列に演算する．既存のレジスタを用いることにより回路規模の増加を抑えつつデータの並列演算を行うことができる．しかし，1つのレジスタに詰め込むことのできるデータは限られている（通常は2個から8個程度）ため，並列度を上げることはできない．

ベクトル演算は図6.2に示すように，ベクトルレジスタに複数のデータを保持し，それらのデータをパイプライン的に演算する．ベクトルレジスタのベクトル長（Length）を増やすことにより，並列度を上げることができる．しかし新たにベクトルレジスタを追加しなければならないため，回路規模は大きくなる．また，ベクトル演算ではベクトル要素をパイライン的に演算するため，ベクトル長が増えると演算にかかるレイテンシが増加する．

以上，SIMD演算とベクトル演算をまとめると表6.1のようになる．

4.3節で述べたように，RMT PUでは優先度に従って各スレッドに対して命令スロットを割り当てる．優先度の高いスレッドに対して命令スロットを割り当てられない場合に，よ

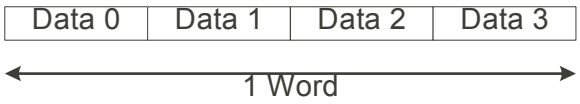

図 6.1: SIMD 演算

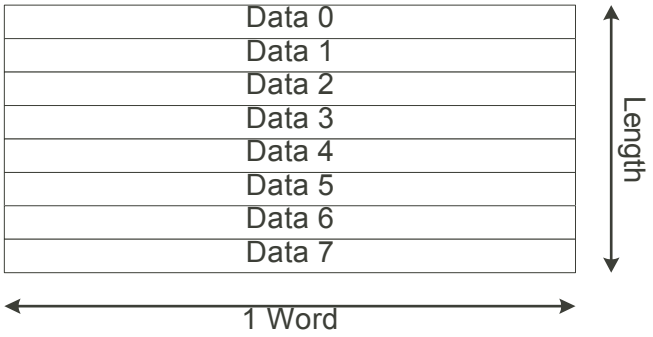

図 6.2: ベクトル演算

表 6.1: SIMD 演算とベクトル演算の比較

	SIMD 演算	ベクトル演算
回路規模	小	大
並列度	小	大
レイテンシ	短い	長い

り優先度の低いスレッドに対して命令スロットが割り当てられる．そのため優先度の高いスレッドが実行されている間は，優先度の低いスレッドは割り当てられる命令スロットの数が減少するため，実行できる命令数が減少する．ソフトリアルタイムスレッドは 4.1 節で述べたとおり，ハードリアルタイムスレッドに比べて優先度が低くなる場合が多い．そのため，ハードリアルタイムスレッドが実行中は，ソフトリアルタイムスレッドの実行できる命令数が減少する．このような場合でも一命令で多くの演算を行うことにより，効率良く命令スロットを使用することができ，演算性能を維持できると考えられる．

SIMD 命令は一命令で演算できるデータは最大でも 8 個程度である．演算を続けるためには絶えず命令を発行し続ける必要がある．一方，ベクトル演算ではベクトル長を長くすることにより，より多くのデータを一命令で演算することができる．演算はパイプライン的に行われるため，一つの命令で演算に数クロック要する．よって絶えず命令を発行しなくても演算を続けることができる．本研究では以上のことから，少ない命令スロットを効

率良く使用するために，データ並列性の高いベクトル演算を用いる．ベクトル演算を用いることにより，SIMD 演算に比べて少ない命令数で演算を行うことができるため，少ない命令スロットで演算を続けることができると考えられる．ベクトル演算では演算を行うためにベクトルレジスタを用意する必要があるため，回路規模が増加するが，1 チップに集積できるゲート数が増加しているため，問題はない．

RMT PU はマルチスレッドアーキテクチャであり，複数のスレッドが並列に実行される．また，5 章で述べたとおり，コンテキストスイッチのオーバーヘッドを削減するために，RMT PU は複数のスレッドをコンテキストキャッシュ内に保持する．これらのスレッドが同時にベクトル演算を行う場合が考えられる．また，3.2.2 節で述べた Multithreaded Vector Architecture のように，複数のスレッドでデータを分割して処理をすることにより，スループットを向上することができる．よって，複数のスレッドでベクトル演算を行う機構が必要となる．

ベクトル演算を行うためにはベクトルレジスタが必要になる．Multithreaded Vector Architecture では全てのスレッドに対してベクトルレジスタを用意している．RMT PU ではコンテキストキャッシュ内にもスレッドを保持しており，プロセッサ内のスレッド数が多いため，全てのスレッドにベクトルレジスタを用意すると回路規模が問題となる．また，本研究ではハードリアルタイム処理とソフトリアルタイム処理の両方を同時に実現することを目的としているため，ハードリアルタイム処理のように全くベクトル演算を行わないスレッドが多数存在することが考えられる．このようなスレッドに対してベクトルレジスタを用意することは非効率的である．よって本研究では，ベクトルレジスタを共有して使用する．システムによって，ベクトル演算を行うスレッドの数や必要となるレジスタの個数，ベクトル長が異なることが考えられるため，ベクトルレジスタの構成を柔軟に変更する機構により，ベクトルレジスタを効率良く共有する．

ソフトリアルタイム処理では，同じ演算を繰り返す場合が多い．従来のプロセッサにおいても，積和演算や Sum of Absolute Difference（SAD）演算のように，繰り返し実行される複数の演算を一命令で行う命令が用意されている．本研究では一命令でより多くの演算を行うことにより，スレッドの優先度が低く，命令スロットの割り当てが減少した場合でも演算性能を維持することを考えている．よってこのような複数の演算を一命令で実行する命令は有効であると考えられる．本研究では，より多くの演算を一命令で実行するために，プログラマが自由に定義した複合演算を一命令で演算する複合演算機構を設計する．

6.2 設計

6.2.1 ベクトルレジスタ制御機構

ベクトル演算を行うためにはベクトルデータを格納するためのベクトルレジスタが必要になる．Multithreaded Vector Architecture のように，プロセッサ内の全てのスレッドに対してベクトルレジスタを用意する方法がある．本研究ではハードウェアコンテキストだけでなくコンテキストキャッシュにもスレッドを保持し，また全てのスレッドがベクトル演算を行うわけではないため，全てのスレッドに対してベクトルレジスタを用意する方法は回路規模の面から非効率的である．よって，本研究ではベクトル演算を行うスレッドの間でベクトルレジスタを共有して使用する．ベクトル演算命令はベクトルレジスタが割り当てられてから実行する．

ベクトルレジスタのサイズを大きくすると，多くのベクトルデータを格納することができ，一度に演算できるデータの個数が増加する．しかし，レジスタの個数が増加した分，回路規模も増加する．また，レジスタ数が大きくなるとレジスタアクセスにかかるレイテンシが増加する．本研究では回路規模とのトレードオフからベクトルレジスタのサイズを 4096 ワード × 32bit とする．これをベクトルレジスタを行うスレッド間で共有する．

ベクトルレジスタの構成（ベクトル長，必要なレジスタの個数，何スレッドで共有するか）は各々のシステムによって異なる．ベクトルレジスタの構成を固定にした場合，システムで要求するレジスタの構成と合わない場合，ベクトルレジスタの共有に無駄が生じる．本研究では，複数のスレッドでベクトルレジスタを共有するため，無駄なくレジスタを割り当てる必要がある．そのためには様々なレジスタ構成の要求に対応する必要がある．よって本研究では，ベクトルレジスタの構成を柔軟に変更する機構を設計する．

ベクトルレジスタの構成を任意に指定できるようにするとハードウェアでの制御が複雑になる．本研究ではベクトルレジスタの構成に制限を設けることにより，ハードウェアが複雑化することを防ぐ．RMT PU は 8 つのハードウェアコンテキストがあるため，ハードウェアコンテキストに割り当てられている全てのスレッドでベクトル演算を行うことができるように，ベクトルレジスタを 8 つの領域（512 ワード）に分割（図 6.3）し，この単位でレジスタを共有する．

ベクトルレジスタを 8 つの領域に分割したため，8 つまでのスレッドがベクトルレジスタを同時に共有することができる．しかし，それ以上の数のスレッドがベクトル演算を行う場合，動的にベクトルレジスタを割り当てる必要がある．よって，ベクトルレジスタの予約命令（VRES: Vector Register Reserve）と開放命令（VREL: Vector Register Release）を用意し，これらを用いてベクトルレジスタを動的にスレッドに割り当てる．予約命令は，スレッドに対してベクトルレジスタを割り当てるための命令である．ベクトル演算を行う

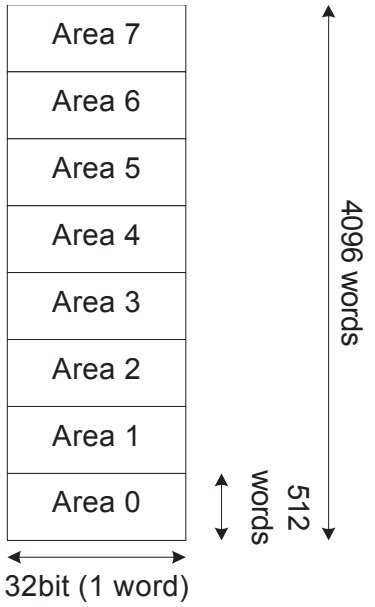

図 6.3: ベクトルレジスタの分割

スレッドは，最初に予約命令によりベクトルレジスタの一部（あるいは全部）を確保する．予約命令では必要となるベクトルレジスタのサイズ（領域の個数）を指定する．予約命令が実行されると，ベクトルレジスタのまだスレッドに割り当てられていない領域の中から，指定された分の領域がスレッドに割り当てられる．割り当てられていない領域が指定されたサイズよりも少なければ，ベクトルレジスタの割り当てを行わず，例外を発生させる．開放命令は割り当てられているベクトルレジスタを開放し，他のスレッドが使用できる状態にする命令である．開放された領域は予約命令により再度割り当てが可能となる．

例えば，システムでベクトル演算を行うスレッドが 1 つしかない場合，図 6.4（a）に示すように全てのベクトルレジスタを 1 つのスレッドに割り当てる．システムで複数のスレッドがベクトル演算を行う場合，図 6.4（b）のように，ベクトルレジスタを分割して割り当てる．システム内でベクトル演算を実行するスレッドの数により，レジスタの構成を変更することができるため，効率良くベクトルレジスタを共有することが可能となる．

ベクトルレジスタの割り当て情報はビットマップ形式で管理する．これにより，図 6.4（c）の Thread 1 のように，連続していない領域に対してスレッドを割り当てることが可能となる．実際のレジスタへのアクセスは後で述べるアドレス変換をハードウェアで行うことにより，ソフトウェアからはベクトルレジスタの共有を意識させず，連続した領域が割り当てられているように見える．

アプリケーションによって扱うデータは様々なので，スレッド毎に最適なベクトル長は異なる．ベクトルレジスタのベクトル長を固定すると，アプリケーションによっては使用

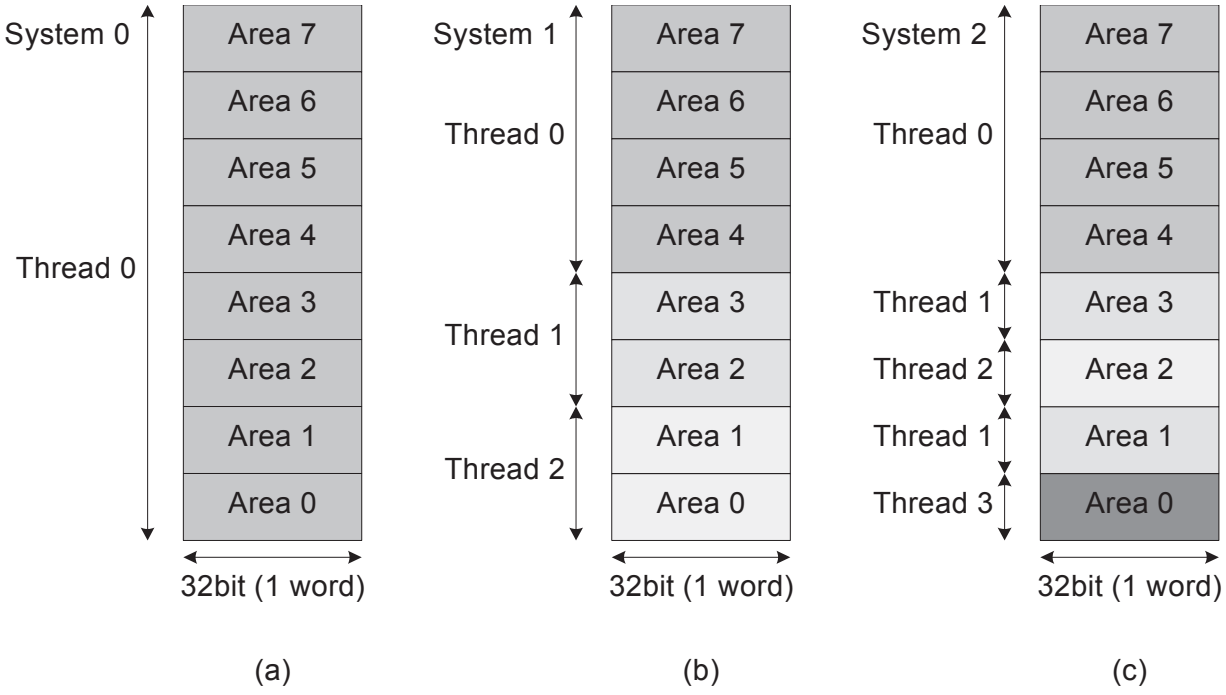

図 6.4: スレッドへのベクトルレジスタの割り当て例

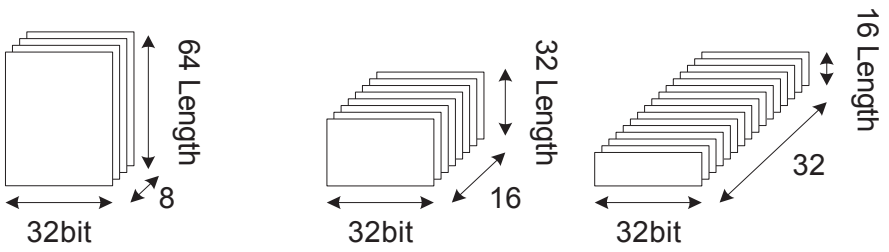

図 6.5: ベクトルレジスタの構成（512 ワード）

されないレジスタができ，無駄が生じる．本研究では効率良くベクトルレジスタを共有するために，ベクトル長とレジスタ ID の構成をスレッド毎に変更する機構を設計する．

RMT PU の命令セットはオペランドの指定が 5bit となっている [伊藤 他]．そのため，レジスタ ID は 32 個まで指定することができる．よって予約したレジスタの個数に応じて，レジスタ ID が 8，16，32 となるようにベクトル長を決定する．

図 6.5 に 1 個分の領域（512 ワード）が割り当てられた場合のレジスタ構成を示す．512 ワードのレジスタが割り当てられた場合，64 Length × 8 個，32 Length × 16 個，16 Length × 32 個といった構成の中から 1 つを選択する．

図 6.6 に 2 個分の領域（1024 ワード）が割り当てられた場合のレジスタ構成を示す．1024

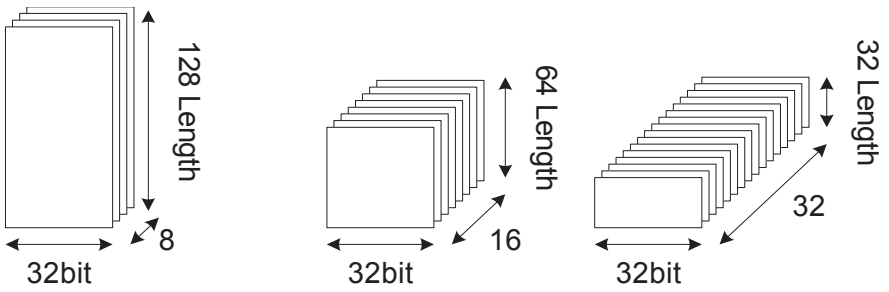

図 6.6: ベクトルレジスタの構成（1024 ワード）

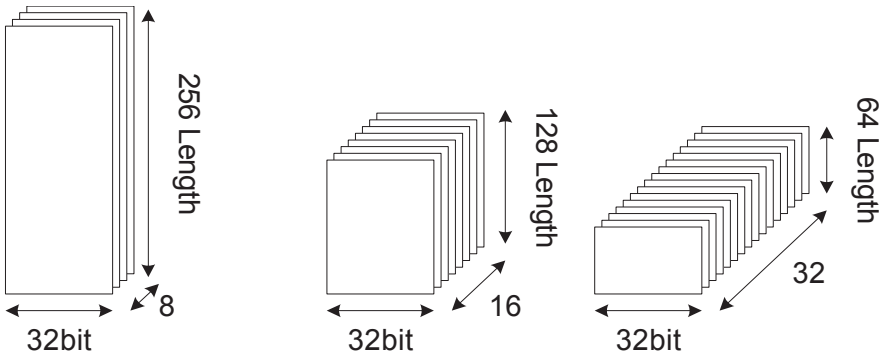

図 6.7: ベクトルレジスタの構成（2048 ワード）

ワードのレジスタが割り当てられた場合，128 Length × 8 個，64 Length × 16 個，32 Length × 32 個といった構成の中から 1 つを選択する．

図 6.7 に 4 個分の領域（2048 ワード）が割り当てられた場合のレジスタ構成を示す．2048 word のレジスタが割り当てられた場合，256 Length × 8 個，128 Length × 16 個，64 Length × 32 個といった構成の中から 1 つを選択する．

図 6.8 に全ての領域（4096 ワード）が割り当てられた場合のレジスタ構成を示す．4096 word のレジスタが割り当てられた場合，512 Length × 8 個，256 Length × 16 個，128 Length × 32 個といった構成の中から 1 つを選択する．

レジスタの構成は，予約命令でサイズを指定する際に一緒に指定する．スレッド毎にベクトル長を変えることができ，ベクトル長が短かくて良い場合はレジスタ ID の個数が増える．これにより，多くのデータを保持することができ，効率良くベクトルレジスタを共有する．

ベクトルレジスタの構成や割り当ての情報をハードウェアで管理するために，図 6.9 に示す Register Control Table に情報を保持する．Size は各スレッドが予約命令で指定した領域の大きさ，Length は図 6.5，6.6，6.7，6.8 で示したベクトルレジスタの構成情報が保持

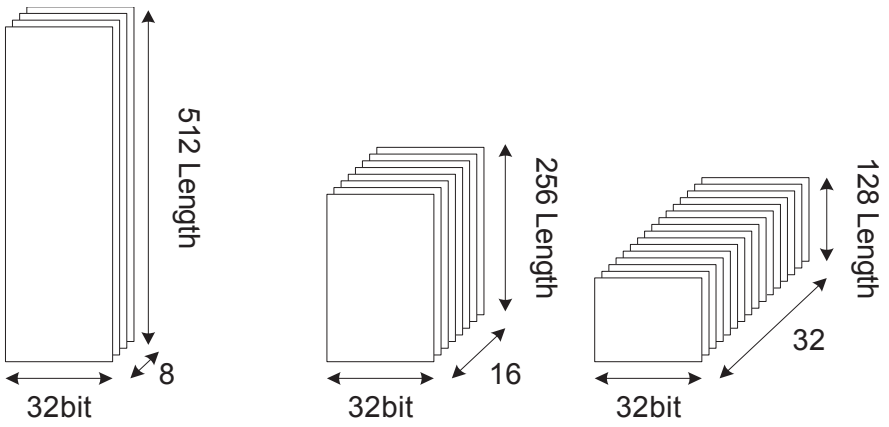

図 6.8: ベクトルレジスタの構成（4096 ワード）

Thread ID	Size	Length	Allocation

図 6.9: Register Control Table

される．Allocation はベクトルレジスタ制御機構がそのスレッドに割り当てたベクトルレジスタの領域をビットマップで保持する．ビットマップで保持することにより，連続していない領域に対してもベクトルレジスタを割り当てることが可能となる．ベクトルレジスタは最大で 8 つのスレッドで共有されるため，Register Control Table は 8 エントリとなる．

ベクトルレジスタの予約命令が実行されると，ベクトルレジスタ制御機構は予約命令で指定されたサイズの分だけベクトルレジスタの領域をスレッドに割り当て，どの領域が割り当てられたかという情報を Register Control Table に書き込む．開放命令が実行された場合，ベクトルレジスタ制御機構は Register Control Table からそのスレッドに割り当てられているベクトルレジスタの領域を調べ，その領域を未割り当て領域に変更し，Register Control Table の内容を更新する．

ソフトウェアからはベクトルレジスタの共有を意識させないために，実際にベクトル演算を行う場合に，オペランドとして指定されたレジスタ ID からベクトルレジスタの実効アドレスを計算する必要がある．ベクトルレジスタ制御機構はベクトル演算命令中のオペランドと，Register Control Table の情報を用いて実効アドレスを計算する．まずレジスタ ID を Length によって左シフトする．シフトする量は Length が 16 の場合は 4bit，32 の場合は 5bit，64 の場合は 6bit，128 の場合は 7bit，256bit の場合は 8bit，512bit の場合は 9bit

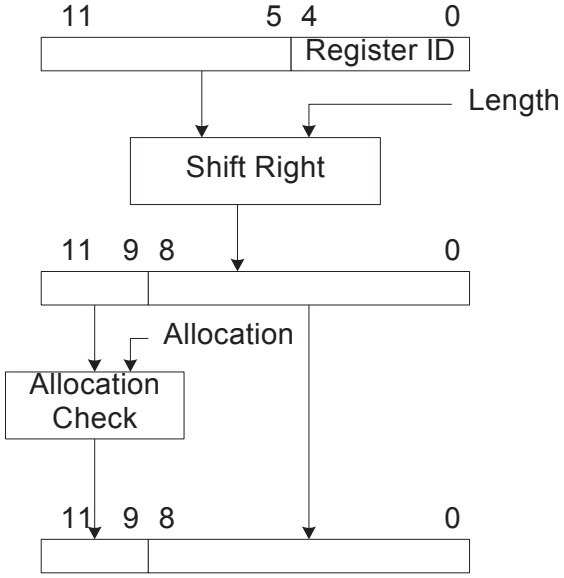

図 6.10: ベクトルレジスタの実効アドレスの計算

となる．シフトした後，上位 3bit と Allocation 情報を調べ，対応する領域に変換する（図 6.10）．例えばシフトした結果，上位 3bit が 2 となった場合，割り当てられたベクトルレジスタの領域のうち 3 番目（0 orign のため）が実効アドレスであることを意味する．このとき例えば Allocation Table が 0b11110000 である（図 6.3 において Area 4 から Area 7 が割り当てられている）場合，このスレッドにとって 3 番目の領域は Area 6 となるため，上位 3bit は 6 に変換される．

ベクトルレジスタ制御機構は計算した実効アドレスを演算パイプラインに送る．演算パイプラインでは渡された実効アドレスから順番にベクトル長の分だけベクトルレジスタにアクセスすることにより，適切なデータを取得する．以上の機構により，ソフトウェアはベクトルレジスタのどの領域が割り当てられたかを意識せずにプログラミングを行うことができる．

6.2.2 複合演算機構

ソフトリアルタイム処理では，図 6.11 に示す例のように，for 文によって同じ演算が繰り返し実行される．従来の研究においても，3.2 節で述べたように，積和演算や Sum of Absolute difference 演算など繰り返し実行される一連の命令を専用命令として用意している．このような一連の演算を一命令で実行することにより，優先度の高いスレッドが実行中であっても，命令スロットをより効率良く利用することがでると考えられる．そこで，複数の演算を 1 命令で実行するための複合演算機構を設計する．既存の研究の多くはマルチ

```
for( i = 0; i < N; i++ ){
  for( j = 0; j < 8; j++ ){
    for( k = 0; k < 8; k++ ){
      sad[i] += abs( x[j][k] - y[j][k] );
    }
  }
}
```

図 6.11: ソフトリアルタイム処理ののプログラム例

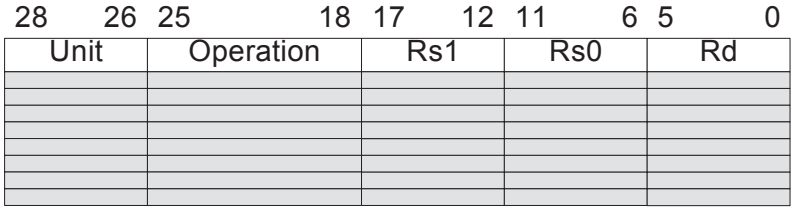

図 6.12: 複合演算命令バッファ

メディアアプリケーションで使用されるアルゴリズムの中で使用頻度の高い複合演算を新たに命令として用意している．このように複合演算を新たな命令として追加した場合，アルゴリズムが複合演算に合わない部分では，複合演算機構を有効に使用することができない．そこで，本研究ではプログラマが複合演算を定義し，それを一命令で実行することにより，どのようなアルゴリズムでも効率良く複合演算機構を用いて実行できるようにする．

プログラマが任意の複合演算を定義するために，専用のバッファ（複合演算命令バッファ）を用意する．プログラマは使用するアルゴリズムに応じて，複合演算命令バッファに複合演算を定義することにより，柔軟に複合演算を変更する．

図 6.12 に複合演算命令バッファを示す．複合演算命令バッファの 1 エントリには，複合演算の 1 つ分の演算を定義する．Unit フィールドには演算パイプラインの種類，Rs0，Rs1，Rd にはそれぞれ演算で使用するオペランドを指定する．Operation フィールドには演算パイプラインで使用する命令が入る．複合演算命令バッファのエントリを複数用いて 1 つの複合演算を定義する．

例えば積和演算を行う場合，図 6.13 に示すように一連の演算を定義する．複合演算命令バッファが空いていれば，複数の複合演算を定義することができる．

RMT PU では複数のスレッドでベクトル演算を行うことを想定しているため，複合演算

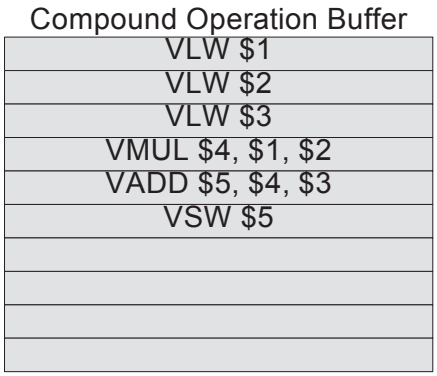

図 6.13: 複合演算命令バッファへの命令定義の例

命令バッファはベクトルレジスタと同様に各スレッドで共有する．ベクトルレジスタの割り当てと情報を共有するために，割り当てられたベクトルレジスタの領域に比例して，複合演算命令バッファを割り当てる．

複合演算命令バッファへ命令を定義するために，専用の定義命令（VDCO : Vector Define Compound Operation）を用意する．また，複合演算命令バッファに定義した複合演算を実行するために複合演算実行命令（VECO : Vector Execute Compound Operation）を用意する．

複合演算定義命令ではオペランドとして，複合演算命令バッファのアドレスと，定義する命令を指定する．複合演算機構は，指定されたアドレスを Register Control Table の Allocation フィールドを使用して実効アドレスに変換し，実効アドレスに対して命令を書き込む．複合演算実行命令では，複合演算命令バッファの中のどの命令を実行するかを，開始アドレスと終了アドレスで指定する．また，for 文を含めて一命令で実行するために，演算回数を指定する．

図 6.14 に複合演算機構の命令実行部分ブロック図を示す．Instruction Buffer から複合演算実行命令が送られてくると，Issue Unit はオペランドとして送られてくる開始アドレス，終了アドレス，繰り返し回数をレジスタに保持する．Issue Unit は指定された開始アドレスとベクトルレジスタの割り当て情報（Allocation）から複合演算命令を読み出すための実効アドレスを計算する．計算した実効アドレスをもとに複合演算命令バッファをアクセスし，最初の命令を読み出す．読み出された命令は，レジスタ ID の変換を行ってベクトルレジスタの実効アドレスを計算し，適切な演算パイプラインへと送る．アドレスをインクリメントし，次の命令を複合演算命令バッファから読み出す．これを指定されて終了アドレスまで繰り返す．

読み出すアドレスが終了アドレスまできた場合，カウンタをインクリメントし，開始ア

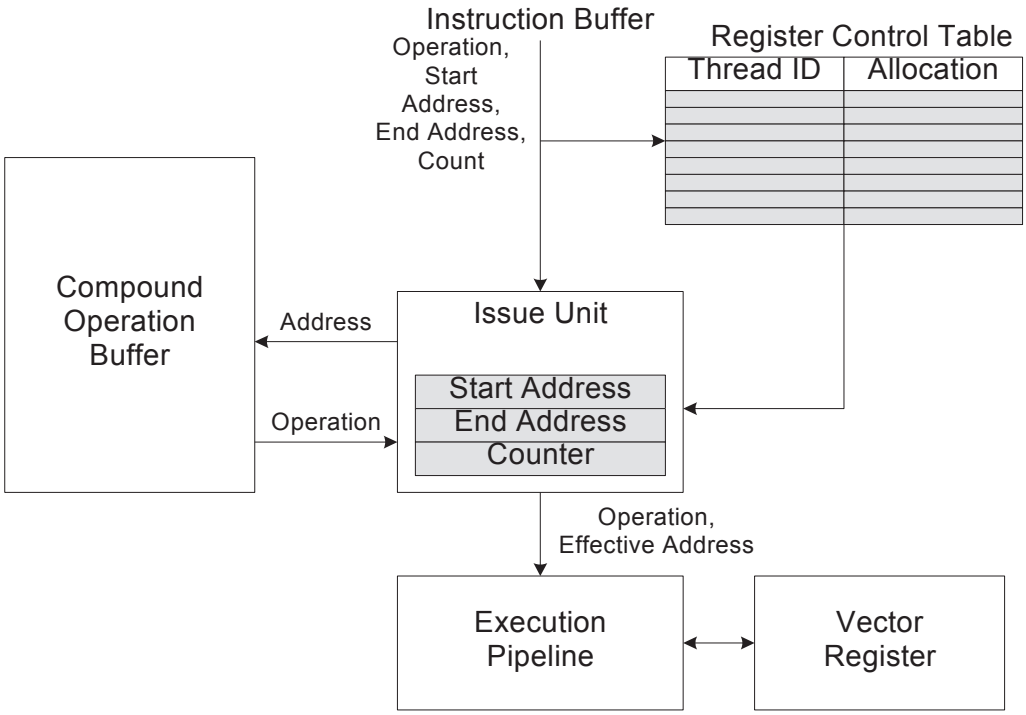

図 6.14: 複合演算機構

ドレスから再び命令を読み出して実行する．これを指定された回数分だけ繰り返すと複合演算実行命令は終了する．複合演算実行命令により，一命令で繰り返しを含めて複数の演算を実行するため，RMT PU の優先度機構により，命令スロットが割り当てられず，命令が発行されない場合でも Issue Unit はレジスタに保持している情報をもとに，ベクトル演算を続けることができる．

6.2.3 ベクトル演算器の構成

図 6.15 にベクトル演算器の構成を示す．また，図 6.16 にベクトル演算器のパイプラインを示す．

本研究ではベクトルレジスタを共有するために，ベクトル演算を行う前にベクトルレジスタが割り当てられている必要がある．よってスレッドに対してベクトルレジスタが割り当てられているかどうかを最初にチェックする必要がある．ベクトル演算命令は Reservation Station から最初に Check Unit に送られる．Check Unit ではスレッドにベクトルレジスタが割り当てられているかを調べる．ベクトルレジスタが割り当てられている場合，ベクトル演算命令は Instruction Buffer へと送られる．ベクトルレジスタが割り当てられていない場合は，例外を発生させる．Check Unit はパイプラインの Check Instruction（CI）ステー

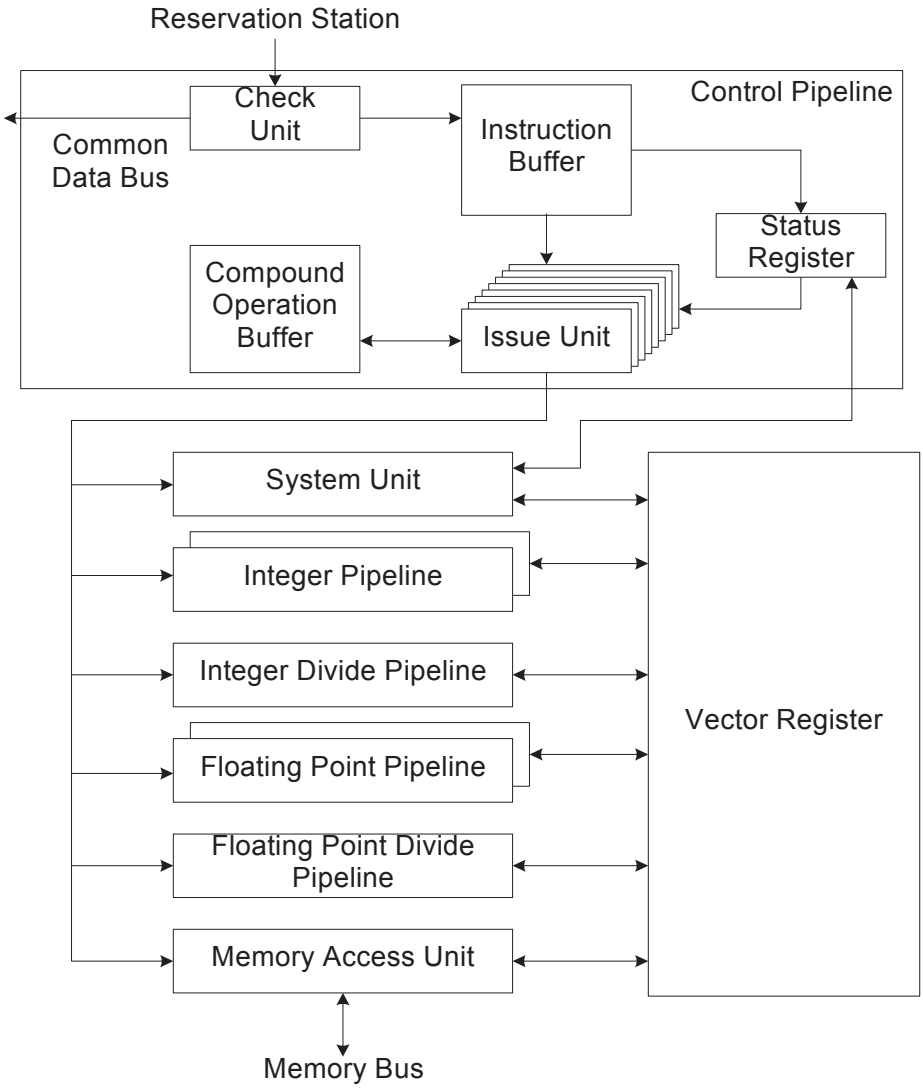

図 6.15: ベクトル演算器の構成

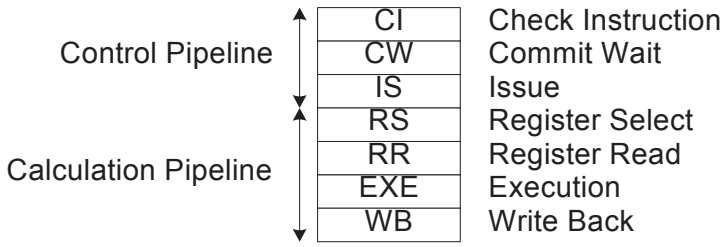

図 6.16: ベクトル演算器のパイプライン

ジに相当する．

RMT PU は Reservation Station と Reorder Buffer を用いて命令を投機実行する．通常の

演算では演算結果を命令コミット時までレジスタに書き込まず，別に保持しておくことによって，投機実行が失敗した場合でも正しく処理を行うことができる．しかしベクトル演算では一度に演算するデータが多いため，命令コミット時まで演算結果を別に保持しておくことは非効率的である．よって本研究ではベクトル演算命令の投機実行は行わない．投機実行を行わないためには，命令がコミットされるまで，ベクトル演算命令を保持しておき，命令がコミットしてから処理を開始する必要がある．

Instruction Buffer はコミットされるまでベクトル演算命令を保持し，コミットされた命令から順番に Issue Unit へと送る．複数のベクトル演算命令が同時にコミットされた場合，Instruction Buffer は優先度の高いスレッドの命令から順番に Issue Unit へ送る．同時に Status Register をアクセスし，スレッドに対するベクトルレジスタの割り当て情報，レジスタの構成を調べ，Issue Unit へ送る．Instruction Buffer はパイプラインの Commit Wait（CW）ステージに相当する．

Issue Unit はハードウェアコンテキストの数（8 個）あり，ベクトル演算命令の種類により使用する演算パイプラインを選択する．演算パイプラインが空いていればそのパイプラインに命令を発行する．演算パイプラインが空いていなければ，パイプラインが空くまで命令を待たせる．複数のスレッドが同時に同じ演算パイプラインを使用しようとした場合，優先度の高いスレッドの命令から先に演算パイプラインを使用するように制御する．Issue Unit はパイプラインの Issue（IS）ステージに相当する．

Register Select ステージ以降が演算パイプラインとなる．ベクトル演算の種類によって，処理を分けるために以下の 6 種類の演算パイプラインを設計する．

- System Unit
- Integer Pipeline
- Integer Divide Pipeline
- Floating Point Pipeline
- Floating Point Divide Pipeline
- Memory Unit

System Unit は汎用レジスタとベクトルレジスタ間のデータ転送，制御レジスタへのアクセス，複合演算命令バッファへの命令定義を行う．

Integer Pipeline は除算以外の整数演算を行う．ベクトルデータはデータ並列性があるため，1 クロックで複数のベクトル要素を演算することにより性能が向上すると考えられる．よって Integer Pipeline は演算器を複数持つことによりベクトルデータを並列演算する．ベ

クトルデータは Memory Bus に合わせて 256bit 長でベクトルレジスタに保持されているため，この幅に合わせて整数演算器を 8 つとする．1 つの整数演算器は 1 つの演算に 1 クロックかかる．

本研究では複数のスレッドがベクトル演算を行うことを想定している．演算パイプラインが 1 つしかない場合，スレッド間で演算パイプラインを使用する際に競合が起こる．演算パイプラインの個数を増やすことにより，並列に演算できるスレッドの数が増えるが，その分回路規模が増加する．また，演算パイプラインはベクトルレジスタからベクトルデータを供給してもらわなければならないため，演算パイプラインを増やす場合，ベクトルレジスタのポート数を増やさなければならない．ベクトルレジスタのポート数の増加は大幅に回路規模を増加させるため，本研究では Integer Pipeline の個数を 2 とする．

Integer Divide Pipeline は整数除算を行う．除算器は回路規模が大きいため，256bit に合わせて除算器を 8 つ持つと回路規模が増加する．よって Integer Divide Pipeline は除算器を 1 つとする．除算器は 1 つの演算に 11 クロックかかるが，完全にパイプライン化されているため 1 クロック毎に演算を完了する．Integer Divide Pipeline は回路規模が大きく，除算命令は他の整数演算命令よりも使用頻度が低い [Hennessy *et al.* 95] ため，Integer Divide Pipeline の個数は 1 とする．

Floating Point Pipeline は除算以外の浮動小数点演算を行う．浮動小数点演算器は回路規模が大きいが，三次元画像処理などを行う場合，浮動小数点演算が頻発する．よって Floating Point Pipeline では演算器を複数持つことにより，ベクトルデータを並列演算する．Floating Point Pipeline では 2 つの単精度浮動小数点演算または 1 つの倍精度浮動小数点演算を行う演算器を 4 つ持つ．演算器は 1 つの演算に 3 クロックかかるが，完全にパイプライン化されているため 1 クロック毎に演算を完了する．

Floating Point Divide Pipeline は浮動小数点除算を行う．浮動小数点除算器は回路規模が大きいため，Floating Point Divide Pipeline の演算器を 1 つとする．演算器は単精度浮動小数点演算に 7 クロック，倍精度浮動小数点演算に 12 クロックかかり，パイプライン化されていない．

Memory Unit はベクトルレジスタとメモリ間のデータ転送を行う．RMT PU は 32KB の Data Cache を持っているが，ベクトルデータは 1 つのデータのサイズが大きく，また多くの場合，一度しか使用されない．このようなデータを Data Cache に入れると Data Cache ミスが増加する恐れがあり，効率が良くない．従来研究においても，3.2 節で述べたように，キャッシュのポリューションを防ぐために L1 キャッシュにデータを置かない手法がある．よって本研究でもキャッシュのポリューションを防ぐために，ベクトルデータは Data Cache を通さず直接メモリから転送する（RMT PU は 2 次キャッシュを持たないため 1 次キャッシュの次はメモリとなる）．Memory Unit は Memory Bus に接続し，直接メモリと

表 6.2: ベクトル演算機構の論理合成結果

	回路規模（μm^2）	最大遅延（ns）
Control Unit	1,871,704.00	3
Integer Pipeline	884,623.75	3
Integer Divide Pipeline	255,911.91	3
Floating Point Pipeline	1,322,488.50	3
Floating Point Divide Pipeline	186,876.95	3
System Unit	20,256.77	3
Memory Unit	24,442.56	3
Register Unit	3,310,430.50	3.08
Total	10,083,847.19	3.08

データの転送を行う．Memory Bus は 256bit 幅のため，ベクトルレジスタの幅を 256bit 長にし，効率良く転送を行う．

6.3 評価

本研究で設計したベクトル演算機構を Verilog-HDL を用いて実装した．本節では本研究で設計したベクトル演算機構の評価を行う．ベクトル演算機構はまだ実機に搭載されていないため，RTL を用いた論理シミュレーションにより評価を行う．

6.3.1 論理合成

RMT Processor は ASIC への実装を予定している．そこで 0.13μm プロセスを用い，Synopsys 社の Design Compiler でベクトル演算機構の論理合成を行った．RMT Processor は 300MHz の動作を目標としている．この条件を満たすために回路のタイミング制約を 3ns とした．また，回路規模は小さければ小さいほど良いため，回路規模の制約条件を大きさ 0 とした．以上の制約条件を与えて論理合成を行った．

表 6.2 に論理合成の結果を示す．スレッド制御機構と同様に，セルの使用率を 60%として配置を行うと，ベクトル演算機構の大きさはチップ全体の 23.3%となり，面積の面においては十分実装可能である．演算パイプラインでは Floating Point Pipeline が大きな面積を占めている．これは浮動小数点演算器のサイズが大きいにも関わらず，ベクトル要素を並列演算するために，複数の浮動小数点演算器を持たせたためと考えられる．ベクトルレジスタはポート数が多く，4096 ワード分のデータを保持するため，サイズが大きくなっていると考えられる．

最大遅延は3.08nsとなり，与えた制約よりは少し大きくなっているが，目標である300MHzは達成している．

6.3.2 基本性能

離散コサイン変換

本研究で設計したベクトル演算器の基本性能を評価する．評価には画像データの圧縮やMPEGエンコード使用される離散コサイン変換プログラムを用いる．

離散コサイン変換では8×8のブロックを行列演算する．ベクトル演算機構を用いる場合と，ベクトル演算機構を用いない（スカラ演算を行う）場合，SIMD演算器を用いる場合について性能を評価する．SIMD演算器は3.2.1節で述べた，浮動小数点レジスタに複数のデータを保存し，これらを並列演算する演算器である．

RMT PUはマルチスレッドアーキテクチャを用いている．3.2節で述べたように，データを複数のスレッドで分割して，マルチスレッドアーキテクチャを用いてスレッドを並列に実行することにより，演算性能を向上することができる．

そこで，ベクトル演算，スカラ演算，SIMD演算に対して，1スレッド，2スレッド，4スレッド，8スレッドで実行した場合について評価する．複数のスレッドで実行する場合は，データの個数をスレッド数で分割して演算を行う．

図6.17に実行結果を示す．性能は演算にかかった時間の逆数とし，図では1スレッドでスカラ演算を行った場合を1とした相対性能であらわしている．1スレッドでベクトル演算を行った場合，1スレッドでスカラ演算を行った場合に比べて4.47倍性能が向上した．8スレッドでベクトル演算を行った場合では，1スレッドでスカラ演算を行った場合に比べて10.0倍演算性能が向上した．ベクトル演算機構を用いない場合，整数演算器を用いて演算を行うことになる．整数演算器は4つのため，最大でも4並列でしか演算できないのに対し，ベクトル演算機構では8並列で演算を行うことができる．また，ベクトル演算ではfor文による分岐命令を削減できるため，分岐ミスのペナルティを減らすことができる．そのため，性能が向上したと考えられる．

1スレッドでSIMD演算を行った場合，1スレッドでスカラ演算を行った場合に比べて1.84倍性能が向上した．8スレッドでSIMD演算を行った場合，1スレッドでスカラ演算を行った場合に比べて7.65倍性能が向上した．SIMD演算も一度に複数のデータを演算することができるため，性能が向上したと考えられる．ベクトル演算による性能向上が，SIMD演算による性能向上よりも高いのは，ベクトル演算では並列度を上げることができ，また，メモリアクセスを一度にまとめてできるためであると考えられる．

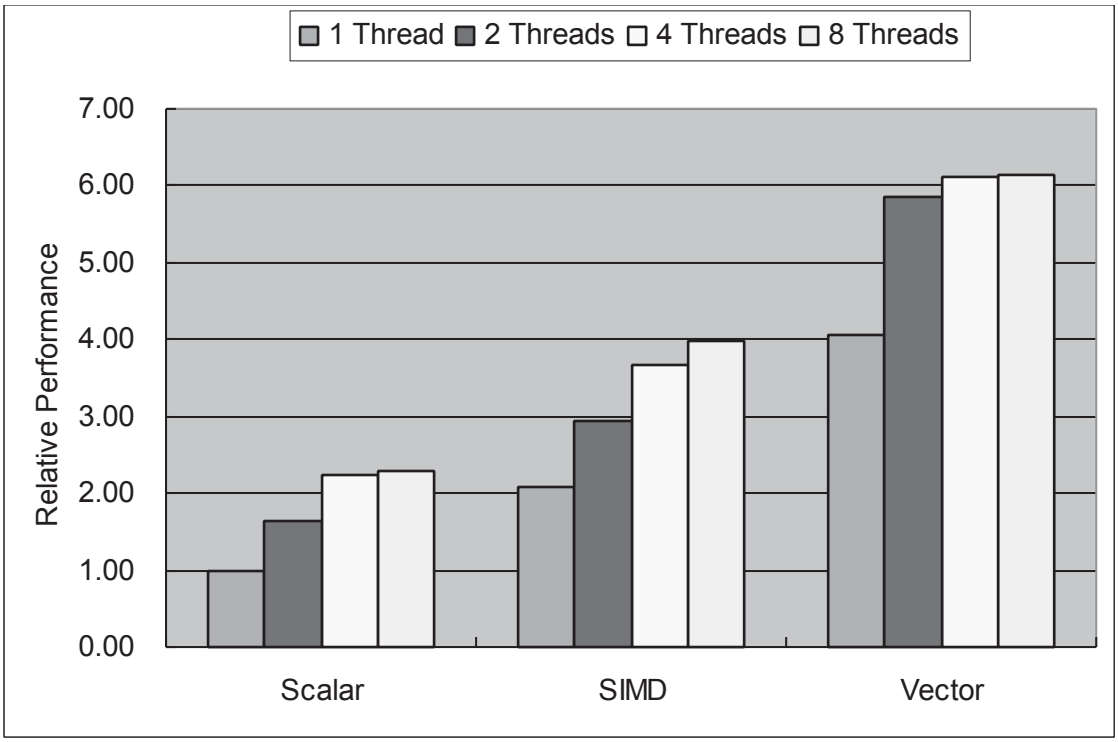

図 6.17: 離散コサイン変換の実行結果

他のアーキテクチャと性能を比較するために，スカラ演算を行うプログラムを Xeon 2.80GHz のプロセッサを用いて Linux 上で実行した．実行結果を図 6.18 に示す．図は RMTPU が 1 スレッドでベクトル演算を行った場合の性能を 1 とした相対性能で示している．2.80GHz の Xeon プロセッサを用いた場合，1 スレッドでベクトル演算を行った場合と比較すると 7.47 倍，8 スレッドでベクトル演算を行った場合と比較すると 4.92 倍の性能となった．

ブロックマッチング

ブロックマッチングは MPEG エンコードの中で動き補償を行うための処理であり，16 × 16 のブロック同士の絶対差の総和（Sum of the Absolute Differences : SAD）からブロック同士の一致度計算するものである．

$$SAD = \sum_{i=0}^{15}\sum_{j=0}^{15} |Block_n[i][j] - Block_m[i][j]|$$

ブロックマッチングを行うプログラムを，スカラ演算で行う場合，ベクトル演算で行う場合，SIMD 演算で行う場合について性能を評価する．それぞれの演算に対して，1 スレッ

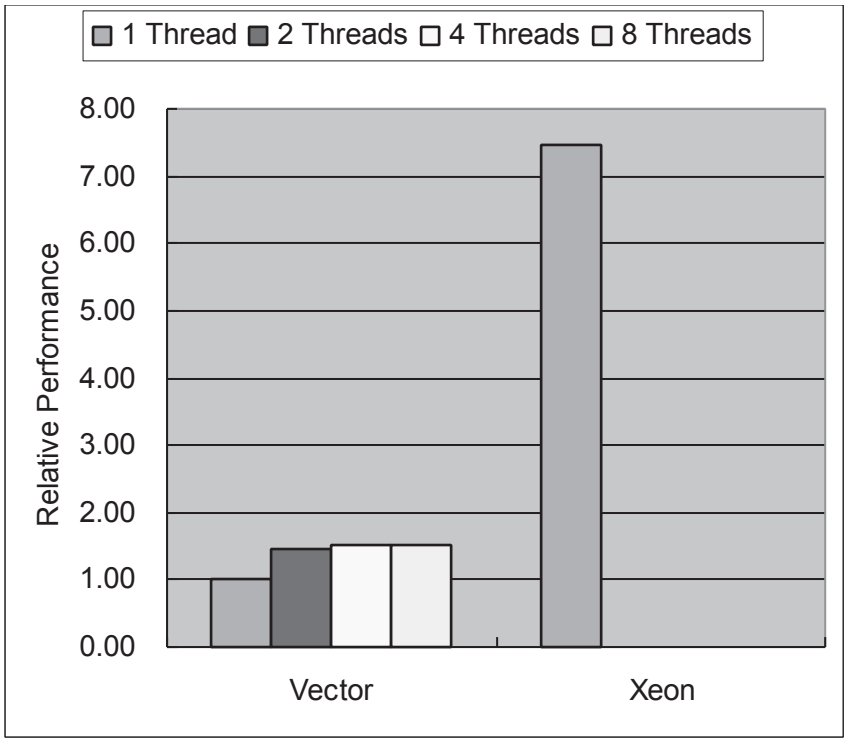

図 6.18: 離散コサイン変換における性能の比較

ド，2 スレッド，4 スレッド，8 スレッドで実行した場合について評価する．複数のスレッドで実行する場合は，データの個数をスレッド数で分割して演算を行う．

図 6.19 に実行結果を示す．性能は演算にかかった時間の逆数とし，図では 1 スレッドでスカラ演算を行った場合を 1 とした相対性能であらわしている．1 スレッドでベクトル演算を行った場合，1 スレッドでスカラ演算を行った場合に比べて 1.67 倍性能が向上した．8 スレッドでベクトル演算を行った場合，1 スレッドでスカラ演算を行った場合に比べて 1.98 倍性能が向上した．

1 スレッドで SIMD 演算を行った場合，1 スレッドでスカラ演算を行った場合に比べて 3.27 倍性能が向上した．8 スレッドで SIMD 演算を行った場合，1 スレッドでスカラ演算を行った場合に比べて 3.54 倍性能が向上した．SIMD 演算はベクトル演算に比べて性能を向上している．ブロックマッチングのプログラムではメモリアクセスが性能に影響を与えており，SIMD 演算ではデータキャッシュを用いるのに対し，ベクトル演算ではメモリからデータを取ってこなければならないため，SIMD 演算の方が性能が向上したと考えられる．よって今後，バス（Memory Bus）の性能改善，ローカルメモリを持たせるなど，メモリアクセス性能の改善を行う必要があると考えられる．

他のアーキテクチャと性能を比較するために，スカラ演算を行うプログラムを Xeon

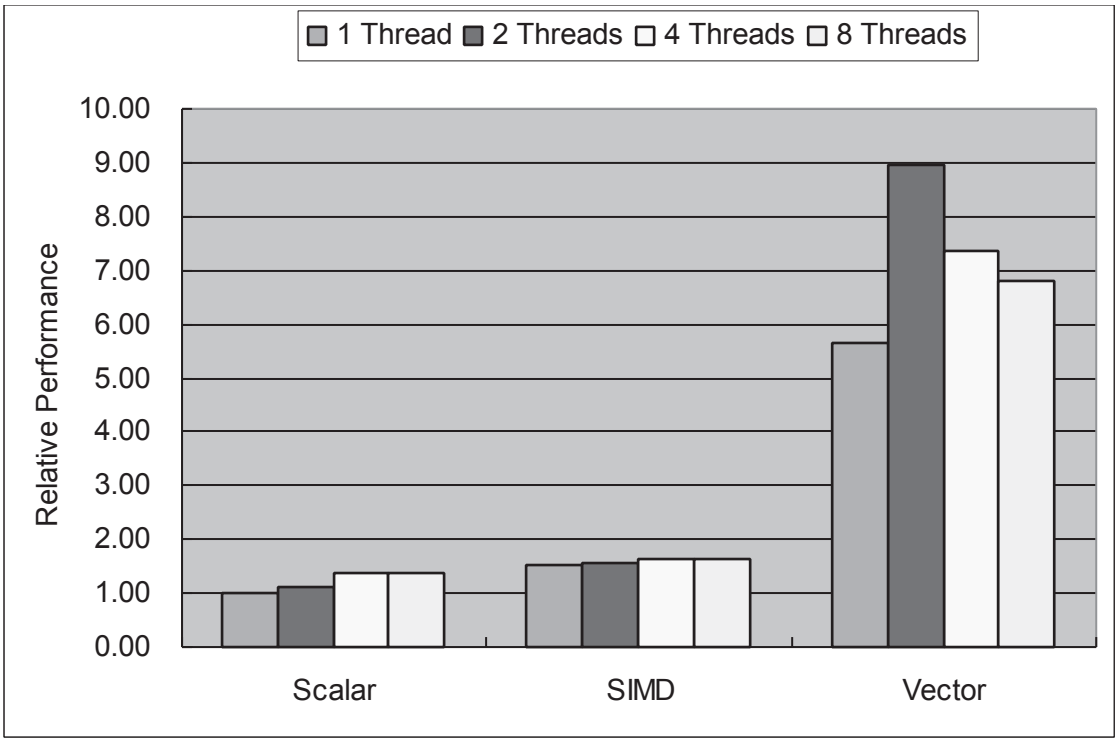

図 6.19: ブロックマッチングの実行結果

2.80GHz のプロセッサを用いて Linux 上で実行した．実行結果を図 6.20 に示す．図は RMT PU が 1 スレッドでベクトル演算を行った場合の性能を 1 とした相対性能で示している．2.80GHz の Xeon プロセッサを用いた場合，1 スレッドでベクトル演算を行った場合と比較すると 5.82 倍．2 スレッドでベクトル演算を行った場合と比較すると，3.68 倍の性能となった．

6.3.3 優先度による影響

RMT PU では複数のスレッドを優先度に従って切り替えられながら実行している．そこで，複数のスレッドが優先度に従って実行されている状況での評価を行う．一定周期毎に起動する周期スレッド（Dhrystone ベンチマークプログラム）を 6 個と離散コサイン変換を行うスレッドに対して優先度を設定する．周期スレッドはそれぞれ 50μs，100μs，200μs，400μs，800μs の周期で起動し，RM スケジューリングにより優先度を与える．周期スレッドの実行時間は単体で実行した場合，28.9μs である．離散コサイン変換を行うスレッドが，スカラ演算を行う場合，ベクトル演算を行う場合，SIMD 演算を行う場合について評価する．それぞれの演算に対して，離散コサイン変換を行うスレッドの優先度を全スレッドの中間にした場合と，最低にした場合について評価する．基本性能の評価において，スカラ

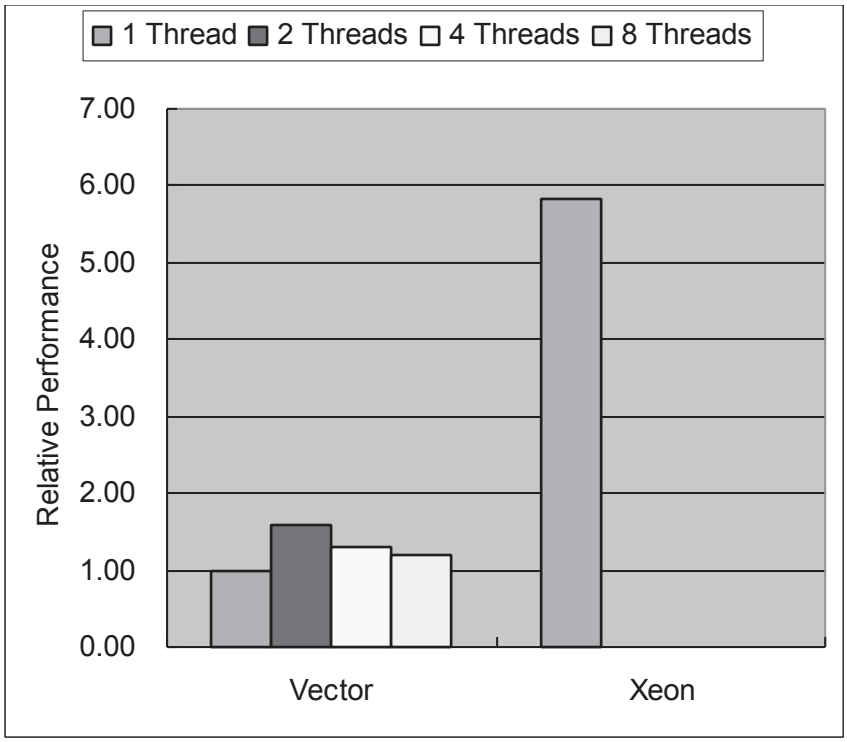

図 6.20: ブロックマッチングにおける性能の比較

演算に比べて SIMD 演算では 2.07 倍，ベクトル演算では 4.05 倍性能が向上している．演算性能の差による影響をなくすため，SIMD 演算を行う場合は 2 倍，ベクトル演算を行う場合は 4 倍のデータを演算する．

図 6.21 に実行結果を示す．図ではそれぞれの演算において，単体で実行した場合の実行時間を 1 とし，優先度によりどの程度実行時間が増加したかを示している．

離散コサイン変換を行うスレッドがスカラ演算を用いる場合，離散コサイン変換を行うスレッドの優先度を全スレッドの中間にした場合，実行時間が 1.61 倍に増加した．離散コサイン変換を行うスレッドの優先度を全スレッド中最も低くした場合，実行時間は 2.03 倍となった．RMT PU の優先度制御により，優先度の低い離散コサイン変換を行うスレッドの実行が，優先度の高い周期スレッドの実行によって中断され，離散コサイン変換を行うスレッドが演算を続けられないためと考えられる．そのため，優先度を中間にした場合と優先度を最低にした場合を比べると，優先度を最低にした場合の方が離散コサイン変換を行うスレッドよりも優先度の高いスレッドが多いため，より実行時間が増加している．

離散コサイン変換を行うスレッドが SIMD 演算を用いる場合，離散コサイン変換を行うスレッドの優先度を全スレッドの中間にした場合，実行時間が 1.59 倍に増加した．離散コサイン変換を行うスレッドの優先度を全スレッド中最も低くした場合，実行時間は 1.97 倍

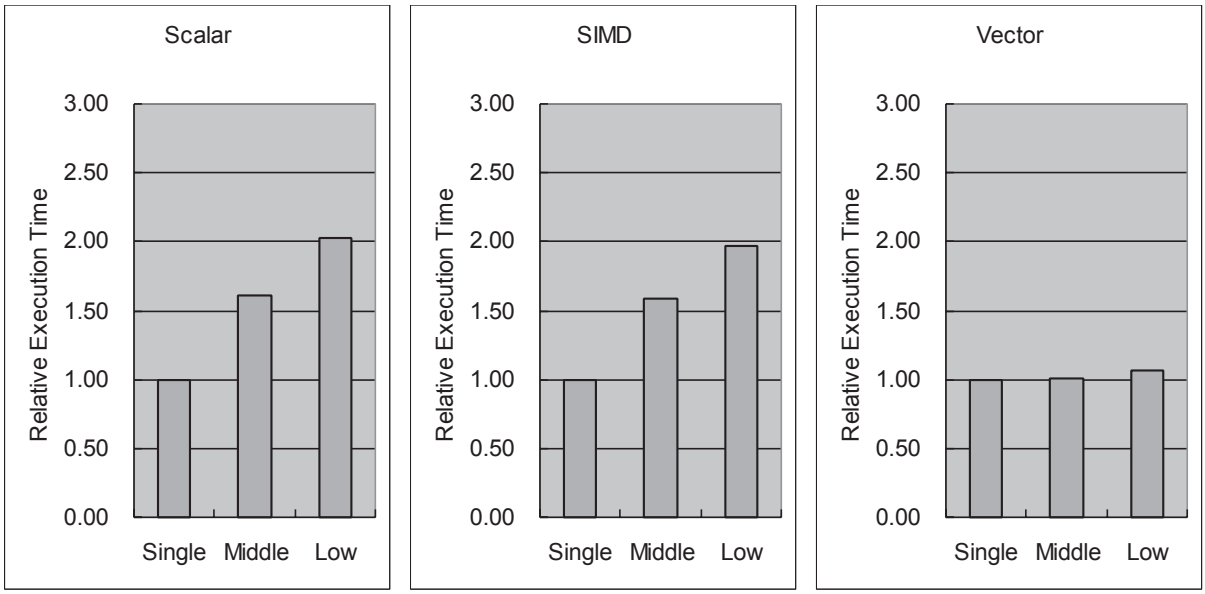

図 6.21: 優先度を設定した場合の実行結果

となった．SIMD 演算は一度に複数のデータを演算することができるが，演算を続けるためには絶えず命令を発行しなければならない．しかし，RMT PU の優先度制御機構のように優先度に従ってスレッドの実行が制御されている場合，優先度の高い周期スレッドが実行中は，SIMD 演算命令を発行することができない．そのため，スカラ演算と同じように優先度が低くなるほど実行時間が増加している．

離散コサイン変換を行うスレッドがベクトル演算を用いる場合，離散コサイン変換を行うスレッドの優先度を全スレッドの中間にした場合，実行時間が 1.01 倍に増加した．離散コサイン変換を行うスレッドの優先度を全スレッド中最も低くした場合，実行時間は 1.07 倍となった．ベクトル演算機構を用いることにより，スカラ演算を用いる場合や SIMD 演算を用いる場合に比べて実行時間の増加を抑えることができている．これは，ベクトルの複合演算により，1 命令で多くのデータを演算するため，スカラ演算や SIMD 演算に比べて少ない命令数で離散コサイン変換を行うことができ，優先度の高い周期スレッドの実行により，離散コサイン変換を行うスレッドの命令発行が一時的に中断されたとしてもベクトル演算を続けることができるため，実行時間の増加を抑えることができたと考えられる．

離散コサイン変換を行うスレッドの優先度を中間にした場合の各スレッドの実行シーケンスを図 6.22，図 6.23 に示す．図 6.22 は離散コサイン変換を行うスレッドがスカラ演算を用いた場合，図 6.23 はベクトル演算を用いた場合において，横軸は時間経過，縦軸は単位時間当たりの命令コミット数を示している．Periodic Thread 0 から Periodic Thread 6 は周期スレッド，DCT Thread は離散コサイン変換を行うスレッドを示している．

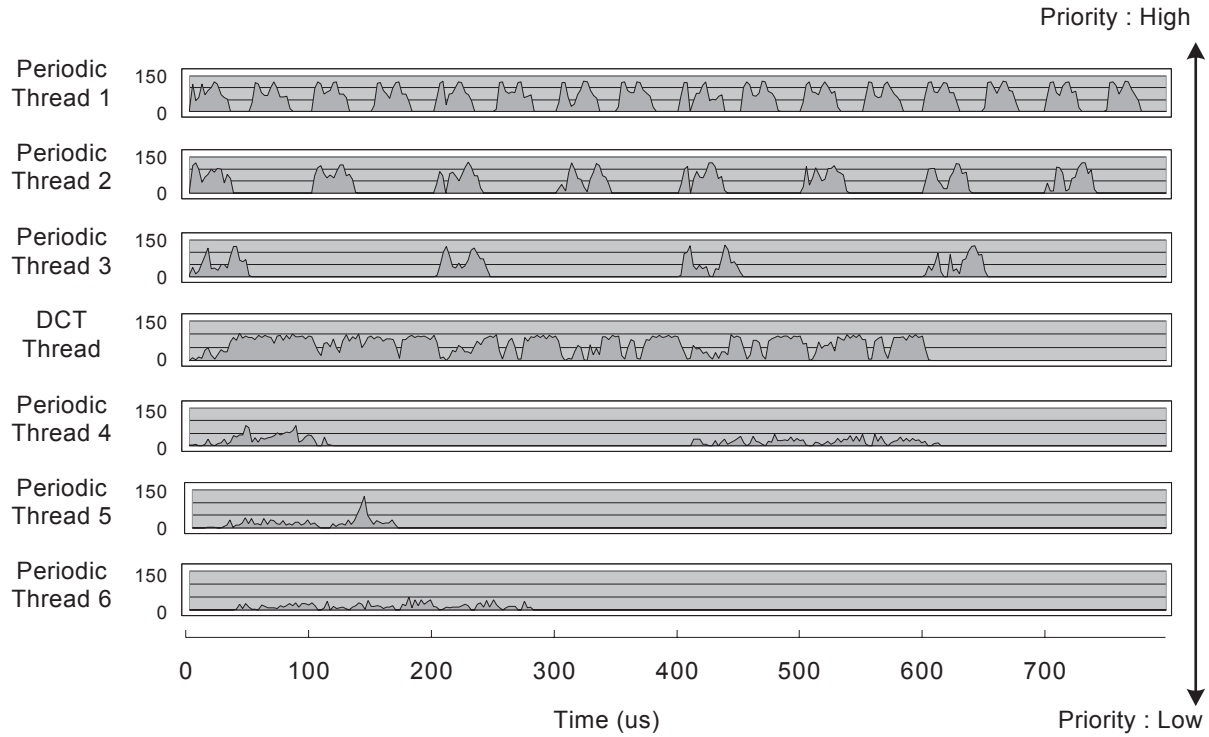

図 6.22: 各スレッドの実行シーケンス（スカラ演算）

スカラ演算を用いる場合，整数演算器を用いて離散コサイン演算を行う．整数演算器の多くが DCT Thread に使われるため，DCT Thread よりも優先度の低い周期スレッドが演算ユニットが使用できないため，実行時間が増加している．一方，ベクトル演算機構を用いる場合，離散コサインの演算はベクトル演算器を用いて行われる．空いている整数演算器を用いて，優先度の低い周期スレッドを実行することができるため，スカラ演算を行う場合に比べて実行時間の増加が抑えられている．つまりベクトル演算機構を用いることにより，優先度の低いスレッドのスループットが向上している．

6.4 本章のまとめ

本章では，優先度に従ってスレッドが実行されている状況においてソフトリアルタイム処理のデータを演算するための，ベクトル演算機構について述べた．

RMT PU では複数のスレッドをプロセッサ内に保持するが，全てのスレッドに対してベクトルレジスタを用意することは非効率的である．よって，本研究では，ベクトル演算を行うスレッドの間でベクトルレジスタを共有する．構築するシステムにより，ベクトル演算を行うスレッドの個数，必要なベクトル長は異なるため，効率良くベクトルレジスタを共有するために，ベクトルレジスタ制御機構では，柔軟にベクトルレジスタの構成を変更

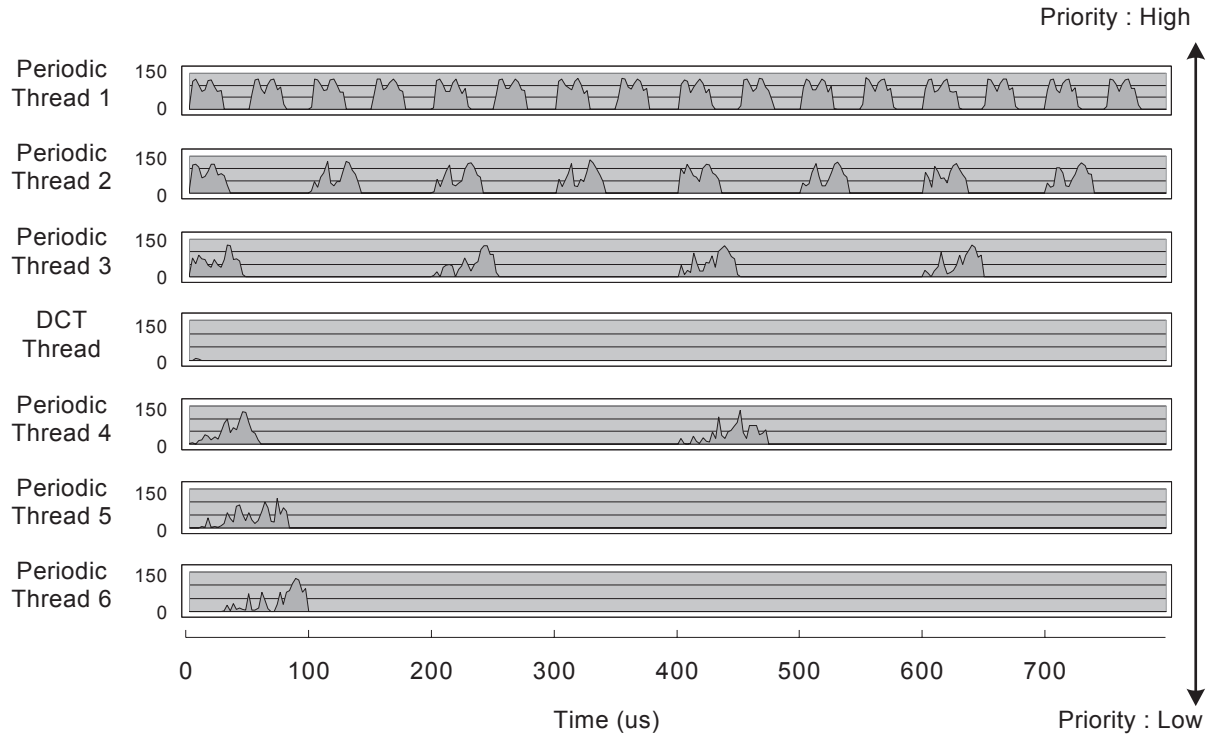

図 6.23: 各スレッドの実行シーケンス（ベクトル演算機）

する機能を持つ．これにより，複数のスレッドで効率良くベクトルレジスタを共有する．

RMT PUでは，スレッドは優先度に従って実行されるため，優先度の高いスレッドが実行可能になると，優先度の低いスレッドに命令スロットが割り当てられず，命令を実行できなくなる．これによりスループットが低下する．本研究では，複合演算機構により，1命令で多くの演算を行うことにより，命令が発行できなくなった場合でもベクトル演算を続ける機構を設計した．これにより，優先度による制御により，ベクトル演算を行うスレッドの命令発行が行われない場合でも，スループットを維持した．

ベクトル演算機構では，本来整数演算器を用いて行っていた大量の演算をベクトル演算器で処理する．整数演算器が空くため，それを利用してより優先度の低いスレッドを実行することができ，優先度の低いスレッドのスループットが向上した．

第7章

議論

5章ではコンテキストスイッチにかかるオーバーヘッドを削減するための，スレッド制御機構について述べた．また，6章ではRMT PUの優先度制御機構を考慮してソフトリアルタイム処理の大量のデータを演算する，ベクトル演算機構について述べた．

スレッド制御機構では，コンテキストスイッチをハードウェアで実行することにより，ソフトウェアでコンテキストスイッチを行った場合に比べて，スレッドの切り替え時間が短縮した．これにより多くのスレッドを切り替えて実行することができ，とくに実行時間の短かいスレッドを切り替えて実行する場合に有効であることを示した．ベクトル演算機構では，複合演算機構により一命令で多くのベクトル演算を行うことにより，スレッドの優先度が低く，RMT PUの優先度制御機構により命令スロットが割り当てられない場合でも，ソフトリアルタイム処理の演算性能を維持し，その有効性を示した．

本章では，スレッド制御機構とベクトル演算機構の両方を用いて，ハードリアルタイム処理とソフトリアルタイム処理を同時に実行する場合において，それぞれの機構の有効性について検討する．

ハードリアルタイムスレッドとソフトリアルタイムスレッドを同時に実行した場合について，まずハードリアルタイムスレッドの時間制約を守るために，ハードリアルタイムスレッドに高い優先度を与えた場合を考える，図7.1にハードリアルタイムスレッドとソフトリアルタイムスレッドの実行例を示す．ハードリアルタイムスレッドに高い優先度を与えた場合，ソフトリアルタイムスレッドはハードリアルタイムスレッドが実行されていないときに実行される．P_hはハードリアルタイムスレッドの実行周期，E_hはハードリアルタイムスレッドの処理時間，E_cはスレッドの入れ替え時間をあらわす．

図のように二つのスレッドを切り替えて実行した場合，ハードリアルタイムスレッド一周期の間にソフトリアルタイムスレッドの実行できる時間は，ハードリアルタイムスレッドの実行周期からハードリアルタイムスレッドの実行時間と二回のコンテキストスイッチの時間を差し引いた分となる．つまり，$E_{s1} = P_h - E_h - E_c \times 2$となる．

ソフトウェアでコンテキストスイッチを行う場合，5.3.2節よりスレッドの切り替えにか

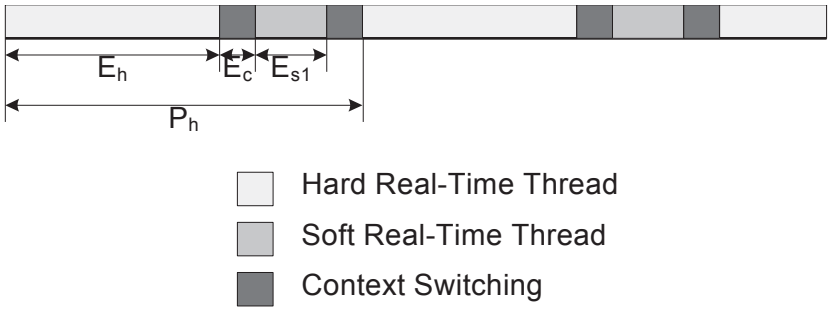

図 7.1: ハードリアルタイムスレッドとソフトリアルタイムスレッドの実行例（1）

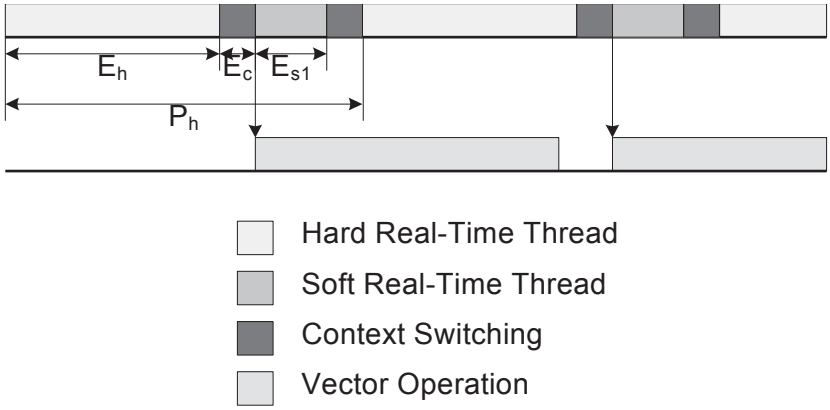

図 7.2: ハードリアルタイムスレッドとソフトリアルタイムスレッドの実行例（2）

かる時間は 15.18μs となる．一方，スレッド制御機構でコンテキストスイッチを行う場合，スレッドの切り替えにかかる時間は 70ns となる．よってスレッド制御機構を用いることにより，ソフトリアルタイムスレッドは $15.18\mu\text{s} - 70\text{ns} = 15.11\mu\text{s}$ だけコンテキストスイッチにかかる時間が削減され，その分多く実行時間を得ることができる．ハードリアルタイム処理では，処理時間が 1μs 程度と短かい場合が多いため，スレッド制御機構による実行時間の増加の影響は大きい．しかし，ソフトリアルタイム処理では大量のデータを演算するため，スレッド制御機構による実行時間の増加の影響は小さい．

図 7.2 にベクトル演算機構を用いてソフトリアルタイムスレッドとハードリアルタイムスレッドを実行した場合の実行例を示す．ベクトル演算機構では 6 章で述べたように，複合演算機構により一命令で多くのベクトル演算を行う．この場合，ベクトル演算命令が発行されなくても複合演算機構により，ベクトル演算を続けることができる．そのため，図 7.2 に示すように，ソフトリアルタイムスレッドが複合演算実行命令を発行した後，コンテキストスイッチによりハードリアルタイムスレッドに実行が切り替わると，ベクトル演算器を用いてソフトリアルタイム処理を，通常の整数演算器を用いてハードリアルタイム処

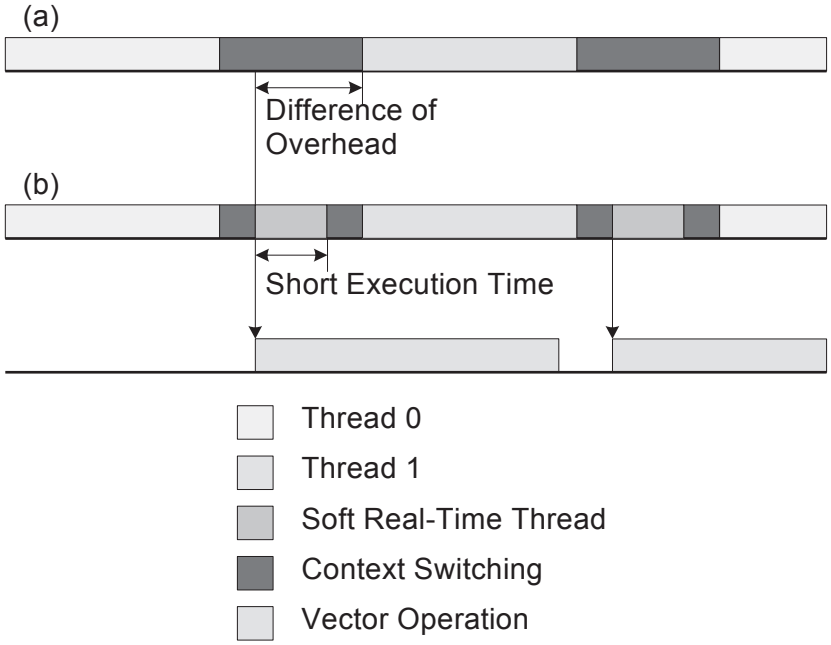

図 7.3: 従来のプロセッサと本研究のプロセッサの比較（1）

理を並列に行うことができる．

ベクトル演算機構では複合演算命令により，一命令で多くの演算を行う．逆に言うと少ない命令数でソフトリアルタイム処理を実現することができる．つまり，短かい実行時間があればソフトリアルタイム処理のデータを演算することができる．実行時間の短かいスレッドを実行する場合，ソフトウェアによりコンテキストスイッチを行うと図 5.17 に示したように，コンテキストスイッチのオーバーヘッドの割り合いが増加する．しかし，スレッド制御機構を用いることにより，図 5.18 に示したようにオーバーヘッドの割り合いを抑えることができる．また，スレッド制御機構によって増加した実行時間を用いて複合演算命令を発行することができる．

従来のプロセッサでは図 7.3（a）に示すようにソフトリアルタイムスレッドを実行することができないところを，スレッド制御機構とベクトル演算機構により図 7.3（b）に示すように実行することが可能となる．つまり，ベクトル演算機構により，ソフトリアルタイム処理の大量のデータ演算は実行時間の短かい処理に置き換えられる．スレッド制御機構はコンテキストスイッチにかかる時間を削減し，削減した分の時間で他のスレッドを実行することができる．この時間を使ってソフトリアルタイムスレッドを実行することにより，ハードリアルタイムスレッドとソフトリアルタイムスレッドを同時に効率良く実行することが可能となる．

次にハードリアルタイムレッドとソフトリアルタイムスレッドを同時に実行した場合に

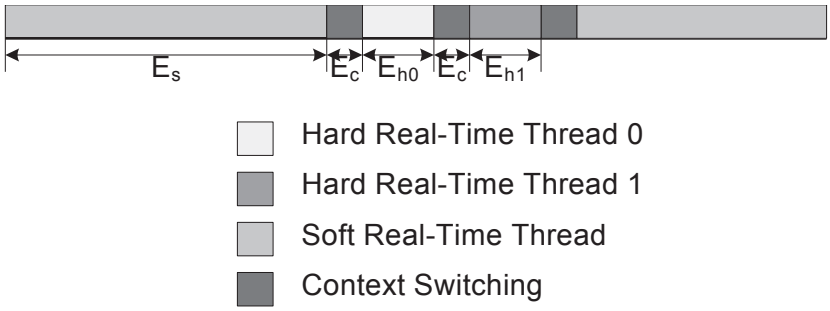

図 7.4: ハードリアルタイムスレッドとソフトリアルタイムスレッドの実行例（3）

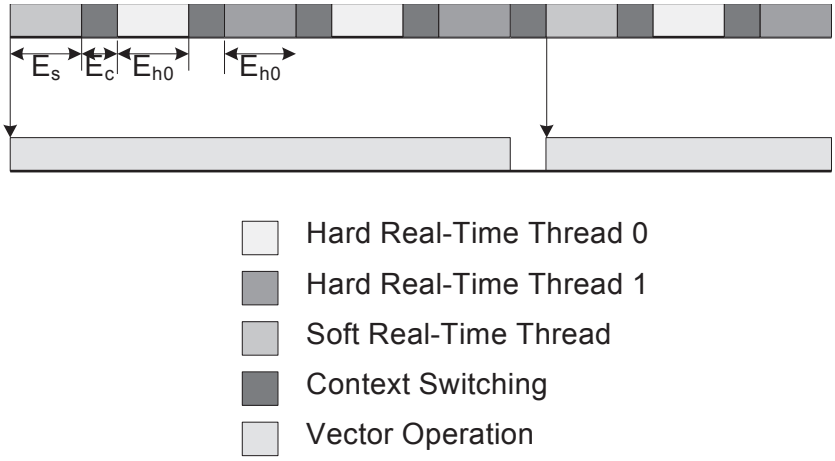

図 7.5: ハードリアルタイムスレッドとソフトリアルタイムスレッドの実行例（4）

ついて，ソフトリアルタイムスレッドの優先度が高い場合を考える．図 7.4 にソフトリアルタイムスレッドの優先度が高い場合の実行例を示す．E_s はソフトリアルタイムスレッドの実行時間，E_{hi} はハードリアルタイムスレッドの実行時間を示す．ソフトリアルタイムスレッドの優先度が高い場合，ハードリアルタイムスレッドはソフトリアルタイムスレッドが実行されていないときに実行される．ソフトリアルタイムスレッドは実行周期が長く，演算量が多いため，演算資源がソフトリアルタイムスレッドに占有され，ハードリアルタイムスレッドが実行できず，ハードリアルタイムスレッドの時間制約を守ることができなくなる．

スレッド制御機構を用いた場合，E_c が 15.11μs 減少する．この場合もソフトリアルタイムスレッドが演算資源を占有し，ハードリアルタイムスレッドを実行できないため，ハードリアルタイムスレッドの時間制約を守ることができなくなる．

図 7.5 にベクトル演算機構を用いてソフトリアルタイムスレッドとハードリアルタイムスレッドを実行した場合の実行例を示す．ベクトル演算機構を用いた場合，ソフトリアル

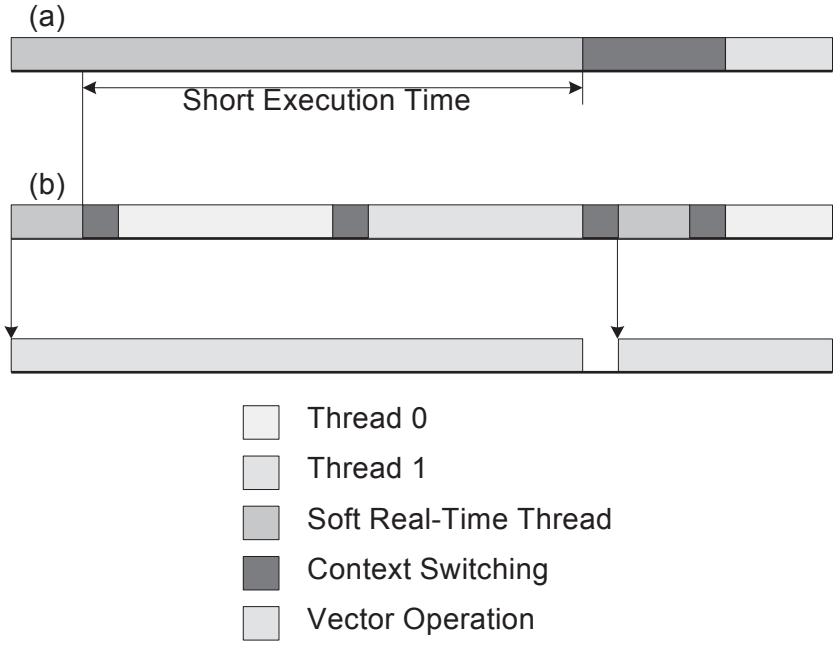

図 7.6: 従来のプロセッサと本研究のプロセッサの比較（2）

タイムスレッドは少ない命令数で処理を行うことができる．よって，ベクトル演算機構を用いない場合に比べてE_sが減少する．また，6.3.3節で述べたように，ソフトリアルタイムスレッドの演算はベクトル演算器で行われるため，空いている整数演算器を用いて別の演算を行うことができる．よって，この演算器を用いて優先度の低いハードリアルタイムスレッドを実行することが可能となる．つまり，ソフトリアルタイムスレッドの命令はベクトル演算器で演算し，ハードリアルタイムスレッドの命令は整数演算器で演算することにより，ソフトリアルタイムスレッドとハードリアルタイムスレッドを並列に実行することができる．さらにスレッド制御機構を用いることにより，E_cが減少し，より多くのスレッドを実行することが可能となる．

従来のプロセッサでは図7.6（a）に示すようにソフトリアルタイムスレッドが演算資源を占有し，優先度の低いスレッドを実行することができないところを，スレッド制御機構とベクトル演算機構により（b）に示すように実行することが可能となる．つまり，ベクトル演算機構により，ソフトリアルタイムスレッドが演算資源を占有することがなくなり，その演算資源を使用してハードリアルタイムスレッドを実行することにより，ハードリアルタイムスレッドとソフトリアルタイムスレッドを同時に効率良く実行することが可能となる．

以上まとめると，ハードリアルタイム処理とソフトリアルタイム処理を同時に実行した場合，ハードリアルタイム処理はスレッド制御機構により，コンテキストスイッチの時間

が短縮され，その空いた時間にベクトル演算機構を用いてソフトリアルタイム処理を実行することが可能となる．また，ソフトリアルタイム処理はベクトル演算機構により，演算資源を占有することがなくなるため，その空いた演算資源を用いてより優先度の低いハードリアルタイム処理を行うことができるため，ハードリアルタイム処理とソフトリアルタイム処理を同時に効率良く実行することができる．

第8章

まとめ

リアルタイム処理は，大きくハードリアルタイム処理とソフトリアルタイム処理に分けられる．ハードリアルタイム処理は時間制約が厳しく，演算量が少ない．ソフトリアルタイム処理はハードリアルタイム処理に比べて時間制約は厳しくないが，大量のデータを演算する．リアルタイム処理では，このように性質の異なる処理を同時に扱う必要がある．

リアルタイム処理では，各タスクの時間制約を守るために，タスクの時間制約によって優先度を決定し，優先度に従ってタスクを切り替えながら実行する．実行するタスクを切り替える場合，コンテキストスイッチが発生する．ソフトウェアでコンテキストスイッチを行う場合，大量のメモリアクセスが発生するため，オーバーヘッドが大きい．

より高度なリアルタイムシステムを構築する場合，実行するタスクの増加や，より短かい時間周期でタスクの切り替えを行うことが考えられる．このような場合，タスクの切り替え回数が増加するため，ソフトウェアによるコンテキストスイッチではオーバーヘッドが問題となる．また，ソフトリアルタイム処理では大量のデータを演算する．ハードリアルタイム処理とソフトリアルタイム処理を同時に実行する場合，大量のデータを演算するために，ソフトリアルタイム処理が演算資源を多く占有することになる．この場合，他のタスクが実行できなくなり問題となる．

よって，今後はソフトウェアだけでなくハードウェアレベルからリアルタイム処理を支援することが必要である．ハードリアルタイム処理とソフトリアルタイム処理の両方を実現するために，ハードウェアは小さいオーバーヘッドでコンテキストスイッチを行う機能と，ソフトリアルタイム処理のデータを演算する機能が必要であると考えられる．

RMT Processorはリアルタイム処理用プロセッサとして設計，実装が行われている．RMT Processorのプロセッシングユニットである RMT PU は，8way SMT アーキテクチャに優先度によるスレッドの制御を行う．優先度制御機構により，8 個までのスレッドをコンテキストスイッチを行わずに優先度の高いスレッドから実行する．しかし 9 個以上のスレッドを実行する場合，従来通りソフトウェアでコンテキストスイッチを行ってスレッドを切り替える必要がある．また，RMT PU の優先度機構では，優先度の高いスレッドに対して命

令スロットが優先的に割り当てられる．そのため，優先度の高いスレッドが実行されている間は，優先度の低いスレッドに対して割り当てられる命令スロットの数が減少する．そのため，ハードリアルタイム処理とソフトリアルタイム処理を同時に実行した場合，ソフトリアルタイム処理の優先度が低いと，大量のデータを演算することができない．逆にソフトリアルタイム処理の優先度が高いと，ソフトリアルタイム処理に演算資源が占有され，他の処理を行うことができない．

本研究は，以上の問題を解決するために，小さいオーバーヘッドでスレッドを切り替えるスレッド制御機構と，ソフトリアルタイム処理のデータを効率良く演算するベクトル演算機構を設計，実装した．

ソフトウェアによるコンテキストスイッチでは外部のメモリに対して Load / Store 命令を用いて 1 つづつコンテキストを転送するため，オーバーヘッドが大きくなる．スレッド制御機構ではコンテキストキャッシュと呼ばれるオンチップメモリを用意し，ハードウェアでコンテキストスイッチを行うことにより，小さいオーバーヘッドでコンテキストスイッチを実現する．コンテキストキャッシュに保持されているスレッドは優先度に従って RMT PU のハードウェアコンテキストと入れ替える．また，割り込み要因をハードウェアで識別し，スレッドをハードウェアで自動的に起動する機能を持つ．スレッド制御機構により，ソフトウェアでコンテキストスイッチを行う場合に比べてコンテキストスイッチにかかる時間を 0.4%に削減した．コンテキストスイッチに費やす時間が減少した分，より多くのスレッドを実行できるようになり，スケジューリング可能性が向上した．特にスレッドの処理時間が短かい場合でも，RMT PU の優先度制御機構により，他のスレッドの実行中にコンテキストスイッチを完了することができ，スレッド制御機構が有効であることを示した．

スレッド制御機構は，スレッドの起動から優先度に従った切り替えをソフトウェアなしで実現する．よって，ソフトウェアは適切な優先度を各スレッドに設定するだけでリアルタイム処理を実現することが可能となる．そのため RM スケジューリングのようにシステムの実行前に優先度を静的に設定可能なアルゴリズムを用いると，システムの実行時にはスケジューラを用いずにリアルタイム処理を行うことが可能となる．

RMT PU は複数のスレッドをプロセッサ内に保持する．ベクトル演算機構では，これらのスレッドで効率良くベクトルレジスタを共有するために，柔軟にベクトルレジスタの構成を変更する機構を設計した．命令スロットを効率良く利用するために，より多くの演算を一命令で実行する複合演算機構を設計した．複合演算機構ではプログラマが定義した複合演算を一命令で実行する．for 文を含めた一連のベクトル演算を一命令で行うことにより，ハードリアルタイムスレッドとソフトリアルタイムスレッドが同時に実行されている場合，RMT PU の優先度機構によってソフトリアルタイムスレッドに命令スロットが割り当てられない場合でもベクトル演算を続けることができ，優先度の低いスレッドのスルー

プットの低下を抑えた．ソフトリアルタイムスレッドの演算をベクトル演算器で行うことにより，空いている演算資源を用いてより優先度の低いスレッドの実行を行うことができ，優先度の低いスレッドのスループットが向上することを示した．

スレッド制御機構とベクトル演算機構により，ハードリアルタイム処理とソフトリアルタイム処理の両方を同時に実行した場合，それぞれの処理を効率良く実行することができることを示した．本研究により，ハードウェアレベルからリアルタイム処理を支援することができるため，より高度なリアルタイムシステムを構築することが可能となる．

謝辞

本研究を進めるにあたり，適切なご指導を頂きました慶應義塾大学助教授 山崎信行先生に厚く御礼申し上げます．先生には学部時代から研究，論文に至るまで大変お世話になりました．論文執筆にあたり，有益なアドバイスを頂いた慶應義塾大学教授 天野英晴先生，慶應義塾大学教授 寺岡文男先生，慶應義塾大学教授 野寺隆先生には大変お世話になりました．先生方にはお忙しい中，論文を見て頂き，大変感謝しております．

本研究を行う機会を頂いた慶應義塾大学教授 安西祐一郎先生に心より感謝を申し上げます．また，共に博士課程で研究生活を送り，色々と助言を頂いた小林秀典君に感謝いたします．安西・山崎・今井研究室で共に過ごしたメンバーには色々と助けて頂き，感謝しております．

最後に，私の進む道を暖かく見守ってくれた父，母に深く感謝の意を表します．

2006 年 2 月

伊藤 務

参考文献

[A. 88] Stankovic J. A. Misconceptions about real-time computing. *IEEE Computer*, Vol. 21, pp. 10–19, 1988.

[Arakawa *et al.* 98] Fumio Arakawa, Osamu Nishii, Kunio Uchiyama, and Norio Nakagawa. SH4 RISC multimedia microprocessor. *IEEE Micro*, Vol. 18, No. 2, pp. 26–34, 1998.

[Benveniste *et al.* 92] A. Benveniste, P. L. Cuernic, Y. Sorel, and M. Sorien. A denotational theory of synchronous reactive systems. *Information and Computation*, Vol. 99, pp. 192–230, 1992.

[Biswas *et al.* 00] Prasenjit Biswas, Atsushi Hasegawa, Srinivas Mandaville, Mark Debbage, Andy Sturges, Fumio Arakawa, Yasuhiko Saito, and Kunio Uchiyama. SH5: The 64-bit SuperH architechture. *IEEE Micro*, Vol. 20, No. 4, pp. 28–37, 2000.

[Brinkschulte *et al.* 99] U. Brinkschulte, C. Karowski, J. Kreuzinger, and T. Ungerer. A multithreaded java microcontroller for thread-oriented real-time event-handling. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT 99)*, pp. 34–39, 1999.

[Buttazo 97] Giorgio C. Buttazo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[Corbal *et al.* 99a] J. Corbal, R. Espasa, and M. Valero. MOM: a matrix simd instruction set architecture for multimedia applications. In *SC'99 Supercomputing Conference*, 1999.

[Corbal *et al.* 99b] Jesus Corbal, and Mateo Valero. Exploiting a new level of dlp in multimedia applications. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 72–79, 1999.

[Diefendorff *et al.* 00] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. Altivec extension to powerpc accelerates media processing. *IEEE Micro*, Vol. 20, No. 2, pp. 85–95, 2000.

[Eggers *et al.* 97] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack K. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading : A platform for next-generation processors. *IEEE Micro*, Vol. 17, No. 5, pp. 12–19, 1997.

[Espasa *et al.* 02] Roger Espasa, Federico Ardanaz, Joel Emer, Stephen Felix, Julio Gago, Roger Gramunt, Isaac Hernandez, Toni Juan, Geoff Lowney, Matthew Mattina, and Andre Seznec. Tarantula: A vector extension to the alpha architecture. In *Proceedings of the 29th Annual International Symposium of Computer Architecture*, pp. 281–292, May 2002.

[Hennessy *et al.* 95] John L Hennessy, and David A Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, second edition, 1995.

[Kreuzinger *et al.* 00] J. Kreuzinger, A. Shulz, M. Pfeffer, TH. Ungerer, U. Brinkshulte, and C. Krakowski. Realtime scheduling on multithreaded processors. In *Proceedings of the Seventh International Conference on Real-Time Computing Systems and Applications*, pp. 155–159, 2000.

[Kunimatsu *et al.* 00] Atsushi Kunimatsu, Nobuhiro Ide, Toshinori Sato, Yukio Endo, Hiroaki Murakami, Takayuki Kamei, Masashi Hirano, Fujio Ishihara, Haruyuki Tago, Masaaki Oka, Akio Ohba, Teiji Yutaka, Toyoshi Okada, and Masakazu Suzuoki. Vector unit architecture for emotion synthesis. *IEEE Micro*, Vol. 20, No. 2, pp. 40–47, 2000.

[Lehiczky *et al.* 89] J. P. Lehiczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavio. In *Proc. IEEE 10th Real-Time Systems Symp*, pp. 166–171, Dec 1989.

[Lin *et al.* 87] K. Lin, S. Natarajan, and J. S. Liu. Imprecise results: Utilizing partial computations in real-time systems. In *Proceedings of the IEEE 8th Real-Time Systems Symposium*, pp. 210–217, 1987.

[Liu *et al.* 73] C. L. Liu, and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, Vol. 20, No. 1, pp. 46–61, 1973.

[MIPS Technologies, Inc.97] MIPS Technologies, Inc., http://www.mips.com/arch/ISA5/MDMXindx. *MIPS Extension for Digital Media with 3D*, 1997.

[Oehring *et al.* 99] Heiko Oehring, Ulrch Sigmund, and Theo Ungerer. Simultaneous multithreading and multimedia. In *MTEAC 99*, 1999.

[Oka *et al.* 99] Masaaki Oka, and Masakazu Suzuoki. Design and programming the emotion engine. *IEEE Micro*, Vol. 19, No. 6, pp. 20–28, 1999.

[Peleg *et al.* 97] Alex Peleg, Sam Wilkie, and Uri Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, Vol. 40, No. 1, pp. 24–38, 1997.

[Raman *et al.* 00] Srinivas K. Raman, Vladimir Pentkovski, and Jagannath Keshave. Implementing streaming simd extensions on the PentiumIII processor. *IEEE Micro*, Vol. 20, No. 4, pp. 47–57, 2000.

[Suzuoki *et al.* 99] Masakazu Suzuoki, Ken Kutaragi, Toshiyuki Hiroi, Hidetaka Magoshi, Shin'ichi Okamoto, Masaaki Oka, Akio Ohba, Yasuyuki Yamamoto, Makoto Furuhashi, Masayoshi Tanaka, Teiji Yutaka, Toyoshi Okada, Masato Nagamatsu, Yukihiro Urakawa, Masami Funyu, Atsushi Kumnimatsu, Harutaka Goto, Kazuhiro Hashimoto, Nobuhiro Ide, Hiroaki Murakami, Yukio Ohtaguro, and Akira Aono. A microprocessor with a 128-bit CPU, ten floating-point MAC's, four floating-point dividers, and an MPEG-2 decoder. *IEEE Journal of Solid-State Circuits*, Vol. 34, No. 11, pp. 1608–1618, 1999.

[Tremblay *et al.* 96] M. Tremblay, J. O'Conner, V. Narayanan, and L. He. Vis speeds new media processing. *IEEE Micro*, Vol. 16, No. 4, pp. 10–20, 1996.

[Tullsen *et al.* 95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading : Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium in Computer Architecture*, pp. 392–403, 1995.

[Tullsen *et al.* 96] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Levy Henry M, Jack K. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 191–202, 1996.

[Yamasaki 01] Nobuyuki Yamasaki. Design and implementation of responsive processor for parallel/distributed control and its development environments. *Journal of Roboticsand Mechatronics*, pp. 125–133, 2001.

[Yamasaki 05] Nobuyuki Yamasaki. Responsive multithreaded processor for distributed real-time systems. *Journal of Robotics and Mechatronics*, Vol. 17, No. 2, pp. 130–141, 2005.

[伊藤 他] 伊藤務, 薄井弘之, 松浦克彦, 一ノ瀬信征, 山崎信行. Responsive Multithreaded Processor 仕様書. http://www.ny.ics.keio.ac.jp/research/rmt/.

[矢向 他 94] 矢向高弘, 菅原智義, 安西祐一郎. μ-pulser: パーソナルロボットを構築するためのオペレーティングシステム. 電子情報通信学会論文誌, Vol. J77-D-1, No. 2, pp. 207–214, 1994.

論文目録

主論文に関する公刊論文

- 伊藤 務，山崎 信行，“Responsive Multithreaded Processor の命令実行機構”，情報処理学会論文誌 Vol.44，No. SIG 11 (ACS 3)，pp.226-235，2003 年.

- Tsutomu Itou, and Nobuyuki Yamasaki, “Design and Implementation of the Multimedia Operation Mechanism for Responsive Multithreaded Processor”, Journal of Robotics and Mechatronics Vol. 17, No. 4, pp. 456-462, 2005.

国際会議発表

- Tsutomu Itou, and Nobuyuki Yamasaki, “The Instruction Execution Mechanism for Responsive Multithreaded Processor”, 19th International Conference on Computers and Their Applications (CATA), pp. 252-255, 2004.