戦略的創造研究推進事業
発展研究（SORST）

研究課題
「超高性能並列分散アーキテクチャに関する研究」

研究期間：平成13年3月1日～平成15年3月31日

研究代表者
山崎 信行
慶應義塾大学理工学部情報工学科　専任講師

１．研究実施の概要

コンピュータの用途には、Personal ComputerやWork Stationのように入力されたデータに対して処理結果をデータとして返す用途の他に、センサ等のデバイスから実世界のデータを取り込んで処理を行い、処理結果を実世界への行動として反映する用途がある。そして、今後コンピュータが人間の生活により密着していくと後者の用途の重要性が増してくると考えられる。そこで必要となってくるのが、リアルタイム性を持つシステムである。

リアルタイム性とは、処理の真偽が結果の真偽と共に、時間にも依存する性質である。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。リアルタイム性はその時間制約により大きくハードリアルタイム性とソフトリアルタイム性に分かれる。ハードリアルタイム性とは、与えられたデッドラインに必ず処理が完了しなければならない性質であり、デッドラインまでに処理が完了しない場合、その価値がただちに0になる性質である。制御アプリケーションなどはハードリアルタイム性を持つ。一方ソフトリアルタイム性とは、与えられたデッドラインまで処理が完了しなくても、その価値がただちに0にならず、結果の品質が時間経過と共に低下する性質である。マルチメディアアプリケーションなどはソフトリアルタイム性を持つ。リアルタイムシステムにおけるタスクの多くは周期的で、ハードリアルタイム性を持つタスクは周期の時間粒度が100usから10msと比較的短く、CPUの処理時間も短いといった特徴を持つ。また、ソフトリアルタイム性を持つタスクの周期は時間粒度が10msから1s程度と比較的長く、データ演算量が多いといった特徴を持つ。

リアルタイムシステムの研究はOS、ネットワーク、プロセッサ等の様々な分野で行われている。今後リアルタイムシステムがより一般的となり、ロボットやホームオートメーション等多くシステムに採用されることになれば、特定の用途に限定せずにリアルタイム処理全般に適用できることが求められてくる。その場合、リアルタイムシステムの基本となるのは、リアルタイムオペレーティングシステム(RT-OS)によって複数のスレッドに優先度を与え、その優先度に従ってスケジューリングを行うシステムである。このようなシステムでは与えられた優先度に従って、優先度の高いスレッドから順番にプロセッサ資源が割り当てられて、実行が行われる。プロセッサ資源が割り当てられているスレッドの実行が完了した場合や、より優先度の高いスレッドが実行可能になった場合、コンテキストスイッチにより、実行するスレッドを入れ替える必要がある。コンテキストスイッチでは現在実行しているタスクのコンテキスト(レジスタセット、プログラムカウンタ、ステータスレジスタ等)をメモリに退避し、

次に実行するタスクのコンテキストをメモリからプロセッサ内に復帰しなければならないため、大きなオーバーヘッドが生じる。このオーバーヘッドを削減するにはソフトウェアよりハードウェア、特にプロセッサによって支援することが効果的であると考えられる。

そこで本研究では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のプロセッサを設計する。

複数のスレッドを並列に実行するプロセッサアーキテクチャにマルチスレッディングがある。マルチスレッディングでは、プロセッサ内に複数のレジスタセットを用意し、それらを用いて複数のコンテキストを同時に保持する。実行中のスレッドがレイテンシの長い命令を実行した場合、別のスレッドに切り替えて実行を続けることにより、レイテンシが隠蔽され、プロセッサのスループットが向上する。この時、ハードウェアにより実行するスレッドを切り替えるため、コンテキストスイッチによるオーバーヘッドを削減することができる。細粒度マルチスレッディングでは、1クロックごとに実行するスレッドを切り替えることにより、依存性の少ない命令を並列実行し、プロセッサのスループットを向上している。また、Simultaneous Multithreadingでは、さらに1クロックごとに複数のスレッドから命令を選択することにより、さらに、依存性の少ない命令を並列実行し、プロセッサのスループットを改善している。

本研究では細粒度マルチスレッディングに、スケジューリングで用いられる優先度を取り入れることにより、リアルタイム処理をハードウェアレベルで支援するResponsive Multithreaded (RMT) Processorを設計・実装する。
最粒度マルチスレッディングでは、複数のスレッドがプロセッサ内で並列に実行されるため、それらのスレッド間で演算ユニット、キャッシュなどの演算資源において競合が起こる。マルチスレッディングでは、資源の競合が起こった場合、ラウンドロビンにより調停を行う。一方RMT Processorでは、演算資源の競合が起こった場合、優先度を用いて調停を行う。これにより、優先度の高いスレッドから優先的に実行される。そのため、スケジューラにより適切に優先度を設定し、RMT Processorを用いてスレッドを実行することにより、コンテキストスイッチをせずに、優先度の高いスレッドから順番に実行することが可能となる。

また、マルチメディア処理などのソフトリアルタイム処理では、高い演算性能が必要となる。これらの処理では、データの並列性が高いという特徴を利用して演算性能を高めることができる。多くの汎用プロセッサではデータの並列性を利用するためにSIMD演算を行う。SIMD(Single Instruction Stream Mult

iple Data Stream)演算は1つの命令で複数のデータを演算する。汎用プロセッサでは1つのレジスタを複数の領域に分割して、それぞれの領域に対して同じ演算を並列に行う。既存のレジスタファイルを利用することによりハードウェア量の増加を防ぐことができ、演算にかかるレイテンシも小さいが、演算の並列度は小さい。一方、データの並列性を利用した別の演算にベクトル演算がある。ベクトル演算はベクトルレジスタを用いてベクトル要素をパイプライン的に演算する。ベクトルレジスタに対してデータのLoad / Storeを一度に行うため、メモリアクセスによるオーバーヘッドが小さい。ベクトルレジスタのベクトル長を長くすることにより並列度を上げることができる一方、演算にかかるレイテンシが増加し、ハードウェア量も増加する。しかしマルチスレッドアーキテクチャではレイテンシが大きい命令の実行を、スレッドを切り替えることにより隠蔽することができるため、本研究では演算性能を向上させるためにベクトル演算を行うVector Unitを設計する。

ベクトル演算は専用のベクトルレジスタを用いて演算を行う。RMT Processorでは全てのスレッドにベクトルレジスタを持たせるとゲートサイズが大きくなってしまうため、これらのスレッドでベクトルレジスタを共有して使用する機構を設計する。そのため基本的な演算命令の他にVector Unitを共有して使うための命令を追加する。Vector Unitの使用はベクトルレジスタの確保から始まる。プログラムにより必要なベクトルレジスタの大きさは異なるため、演算に必要な分だけベクトルレジスタを確保することにより、ベクトルレジスタを効率良く共有する。さらに、ユーザが定義することができる複合演算命令によりVector Unitの使用率を向上させる。

２．研究構想

現在、単一プロセッサの性能は十分高速であり、今後も速くなり続けることが予想されるが、その性能を持ってしても処理しきれないような大規模システムや、空間的に広がっているようなシステムを実現しようとすると、高性能・高機能な並列分散処理が必要になってくる。

そこで、本研究では、非常に高性能かつ高機能の通信及び演算の両方を実現する並列分散アーキテクチャの研究開発を行い、それらを融合する。具体的には、1チップ当り10[GFlops]以上の演算性能を引き出すことを狙った超高性能プロセッサアーキテクチャの研究開発を行う。同時に10[Gbps]以上の通信性能を引き出すことを狙った超高速ネットワークアーキテクチャの研究開発を行う。そして、2年後にはそれらをシステムオンチップとして1チップに集積した1チップスーパーコンピュータのアーキテクチャを実現する。その際、それらのチップを複数個接続することによって、スケーラブルな超高性能並列分散コンピューティングシステムを構築可能にするように設計を行う。

本研究開発によって、安価かつ小さなサイズでスーパコンピュータ並の演算性能を得ることができるようになるため、科学技術計算分野から組込、制御、アミューズメント分野に至るまでの様々な分野において大きなインパクトを与えることが期待でき、本チップを応用することによって多岐にわたる技術革新が期待できる。

例えば本チップを科学技術計算用途に応用すれば、価格性能比の非常に高いスーパコンピュータが実現可能であると考えられる。また、例えばヒューマノイドロボットに応用したとすれば、オンボードでのリアルタイム画像処理、音声認識、音声合成から、オンラインでの複雑なプランニングまで実現可能にすると考えられる。

本研究は、研究代表者が単独で研究を行う

３．研究内容

(1)実施の内容

本研究では、細粒度マルチスレッディングを基本として、スケジューリングにより各スレッドに付与される優先度を、プロセッサ内の演算資源の競合を調停するために使用することにより、リアルタイム処理をハードウェアレベルで支援するResponsive Multithreaded (RMT) Processorの設計及び実装を行う。細粒度マルチスレッディングアーキテクチャを採用した理由は以下の2点である。

1つ目の理由は、複数のスレッドをプロセッサ内にハードウェアコンテキストとして保持しハードウェアでスレッドを切り替えることにより、コンテキストスイッチを行うことなく別のスレッドを実行することができるため、リアルタイムシステムにおけるスレッドスケジューリングに伴う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可能である。2つ目の理由は、リアルタイムシステムにおける様々なスレッドを1リードポートの命令キャッシュで並列実行可能でありかつシングルスレッドでの高い実行スループットを実現することができるためである。

マルチスレッドプロセッサをリアルタイム処理用として応用する際に最も重要で基本となるのは、リアルタイムシステムのスケジューリングで用いられる優先度をプロセッサ内に保持するスレッドにも導入するというアイデアである。プロセッサ内で生じるスレッド間でのあらゆる演算資源の競合の解決を優先度に基づいて行い、高い優先度を持つスレッドの実行が低い優先度を持つスレッドによって実行が阻害されることを防ぎ、高い優先度を持つスレッドの実行を保証する。

図1に、通常のプロセッサにおいて優先度が付与された8つのスレッドをリアルタイム実行した場合の例を示す。例では、Task0により高い優先度が与えられ、Task7に最も低い優先度が与えられている。横軸は時間を示す。青い矢印はそのスレッドが実行可能になる時間(Release Time)を示し、赤い矢印はそのスレッドのデッドラインを示す。図1の例では最初にTask1、2、3が実行可能になっている。そのため、最も優先度の高い、Task1にプロセッサ資源が与えられ、Task1が実行される。

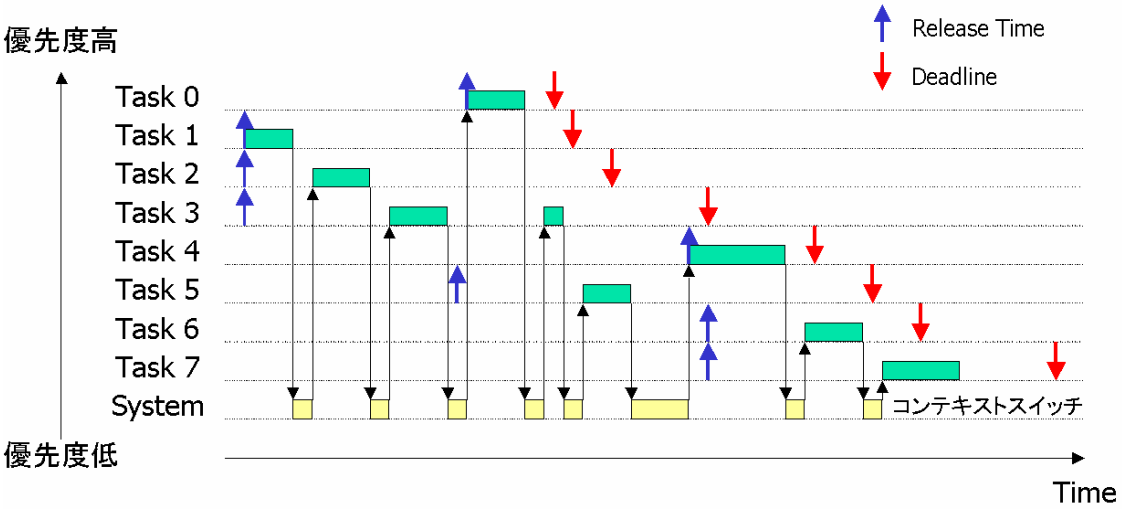

図1 通常のプロセッサによるリアルタイム実行

Task1の実行が完了すると、次に優先度の高いTask2の実行を行うためにコンテ

キストスイッチが生じる。コンテキストスイッチにより、現在実行しているTask1のコンテキストをメモリに対比し、次に実行するTask2のコンテキストをプロセッサ内に復帰する。その後、Task2の実行を開始する。さらに、Task2の実行が完了するとコンテキストスイッチが起こり、Task3の実行が開始される。ここで、より優先度の高いTask0が実行可能になると、Task3の実行が中断され、コンテキストスイッチを行ってTask0の実行を先に行う。Task0の実行が完了すると、コンテキストスイッチを行い、中断していたTask3の実行を再開する。このように、実行するスレッドを変更する度に、コンテキストスイッチが生じる。コンテキストスイッチでは千サイクル近くの大きなオーバーヘッドとなる。

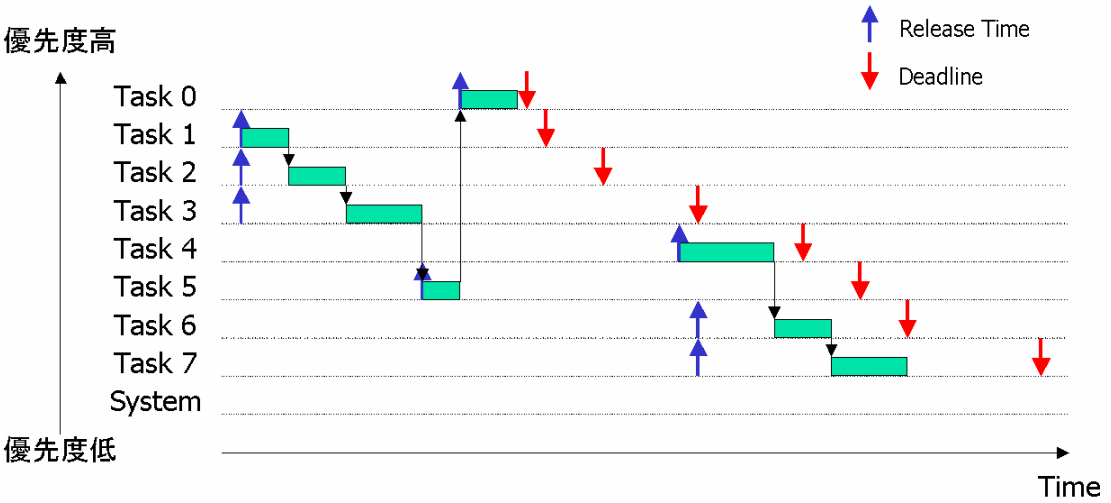

図2 RMT Processorによるリアルタイム実行

図2にRMT Processorによるリアルタイム実行の例を示す。図1の例と同様に、Task0により高い優先度が与えられ、Task7に最も低い優先度が与えられている。横軸は時間を示す。青い矢印はそのスレッドが実行可能になる時間(Release Time)を示し、赤い矢印はそのスレッドのデッドラインを示す。

図2の例では、最初にTask1、2、3が実行可能になっている。そのため、最も優先度の高い、Task1にプロセッサ資源が与えられ、Task1が実行される。タスク1の実行が完了すると、Task2の実行が開始する。この時、マルチスレッディングにより、ハードウェアで実行するスレッドが変更されるため、コンテキストスイッチは起こらず、直ちにTask2の実行を開始することが可能である。また、Task5を実行中に、より優先度の高いTask0が実行可能になると、マルチスレッディングにより、直ちにTask0の実行が開始される。

このように、スケジューリングに用いられる優先度を用いて、プロセッサ内の演算資源の調停を行い、優先度の高いスレッドを優先的に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いながら実行した場合と動揺の効果を得る。

実際には、メモリアクセスやデータ依存により、優先度の高いスレッドがストールする場合がある。その場合、直ちに次に優先度の高いスレッドが実行される。そのため、優先度の高い順に1スレッドずつ実行されるのではなく、実際には優先度の高い順に、数スレッドずつ並列に実行される。

RMT Processorでは8つのコンテキストを同時に保持するため、Rate Monotonicなどの静的スケジューリングによってスケジューリング(優先度が付与)されたスレッドが8スレッド以内であれば、スケジューラ無しでリアルタイム実行を行うことが可能である。この場合、コンテキストスイッチのオーバーヘッドは生じない。また、スケジューリングされたスレッドが8スレッドより多い場合や、EDF(Early Deadline First)などの動的スケジューリングを行う場合は、ソフトウェアによるスケジューリング(スケジューラ)が必要となる。

しかし、RMT Processorのリアルタイム実行の効果と、次に述べるコンテキストキャッシュを利用することにより、スケジューラはより短い周期でスケジューリングを行う、もしくはより多くのスレッドをスケジューリングすることが可能となる。また、細粒度マルチスレッディングにより、優先度の高いスレッドが長いレイテンシの命令を実行した場合は、空いている計算資源を優先度の低いスレッドが使用できるため、優先度の高いスレッドの実行を妨げることなく全体のスループットを向上することが可能である。

先に述べた通り、RMT Processorではプロセッサ内に保持することのできるスレッドの数は8スレッドに限られているため、それより多くのスレッドを実行する場合、コンテキストスイッチが起こる。コンテキストスイッチでは、今まで実行されていたスレッドのコンテキスト(汎用レジスタやプログラムカウンタ、ステータスレジスタ等)の退避を行い、新しく実行されるスレッドのコンテキストの復帰を行わなければならないため、大きなオーバーヘッドとなる。特にリアルタイムシステムでは頻繁にコンテキストスイッチが発生するため、このオーバーヘッドが大きな問題となる。この問題を解決するために、RMT Processorではコンテキストを格納するための専用キャッシュをオンチップに用意し、レジスタファイルとの間をバンド幅の広い専用バス(GPR:256bit、FPR:128bit)で接続している。コンテキストの退避と復帰を高速なオンチップメモリとバンド幅の広い専用バスを用いてハードウェアで行うことにより、4クロックサイクルでコンテキストスイッチを行うことができ、大幅にオーバーヘッドを削減している。

一方、ストリーミング等のソフトリアルタイム処理で扱われるデータは並列性が高く、大量のデータに対して繰り返し同じ演算を行うといった特徴がある。データ並列性を利用して演算性能を向上させる方法として、SIMD演算とベクトル演算がある。SIMD(Single Instruction Stream Multiple Data Stream)演算は1つの命令で複数のデータを演算する。汎用プロセッサでは1つのレジスタを複数の領域に分割して、それぞれの領域に対して同じ演算を並列に行う。既存のレジスタファイルを利用することによりハードウェア量の増加を防ぐことができ、演算にかかるレイテンシも小さいが、演算の並列度は小さい。一方、ベクトル演算はベクトルレジスタを用いてベクトル要素をパイプライン的に演算する。ベクトルレジスタに対してデータのLoad / Storeを一度に行うため、メモリアクセスによるオーバーヘッドが小さい。ベクトルレジスタのベクトル長を長くすることにより並列度を上げることができる一方、演算にかかるレイテンシが増加し、ハードウェア量も増加する。

先に述べたように、RMT Processorでは優先度を用いて計算資源の調停を行う。この時、命令キャッシュへのアクセスが優先度による調停の影響を大きく受ける。通常は優先度の高いスレッドが命令キャッシュをアクセスし、長いレイテンシの命令などで優先度の高いスレッドが命令キャッシュをアクセスしない時に、より優先度の低いスレッドが命令キャッシュをアクセスして命令を実行する。ハードリアルタイム性を持つスレッドの時間制約を保証するために、時間制約の厳しくないソフトリアルタイム性を持つスレッドはハードリアルタイム性を持つスレッドよりも低い優先度で実行される。そのため、ハードリアルタイムのスレッドが実行されている間は、ハードリアルタイムのスレッドが命令キャッシュをアクセスしない時にのみ、ソフトリアルタイムのスレッドの命令がフェッチされる。この命令フェッチを有効に使用することにより、ソフトリアルタイム処理の性能を向上することができると考えられる。そこでRMT Processorでは、ソフトリアルタイム処理に必要とされる演算性能を達成するために、少ない命令数で高い並列度を実現することができるベクトル演算を用いる。これにより少ない命令フェッチで高い演算性能を実現する。また、優先度の高いスレッドの実行が完了し、命令フェッチが多く行われるようになった場合でも、細粒度マルチスレッディングにより、ベクトル演算にかかるレイテンシは、別のスレッドを実行することで隠蔽される。

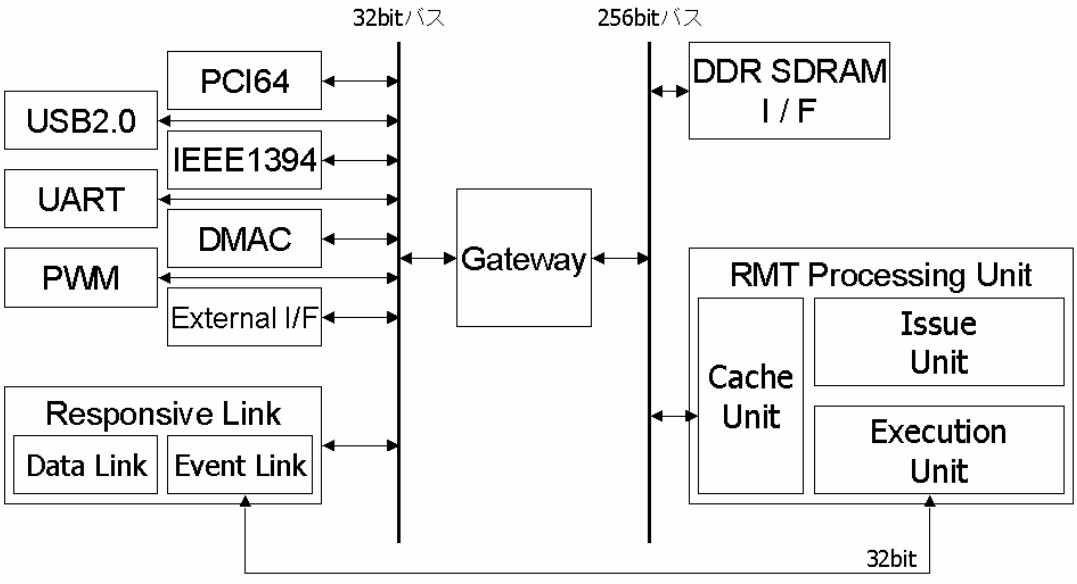

図3 RMT Processorのブロック図

RMT Processorのブロック図を図3に示す。また、RMT Processorのレイアウト図を図4に示す。RMT Processorのプロセッシングユニットであ るRMT Processing Unit(RMT PU)は、256bitのバスを介してDDR SDRAM I/Fと接続している。バンド幅の広いバスを用いてプロセッシングコアとメインメモリを接続することにより、命令フェッチやベクトル演算において、メモリアクセスのスループットを改善している。

レスポンシブリンク(Responsive Link)、PCI I/F、USB I/F、IEEE1394 I/F、シリアル I/F(UART)、DMAコントローラ(DMAC)、PWMジェネレータとパルスカウンタ(PWM/PC)、クロック制御ユニット(CLK CTRL)、外部バス I/F(EXT I/F)の各種I/Oは32bitバスに接続されている。32bitバスと256bitバスはゲートウェイ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、レスポンシブリンク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いる。プロセッシングコアから、バスを介さず、制御レジスタの一部としてレスポンシブリンクをアクセスすることにより、高速にイベントリンクにアクセスすることが可能である。

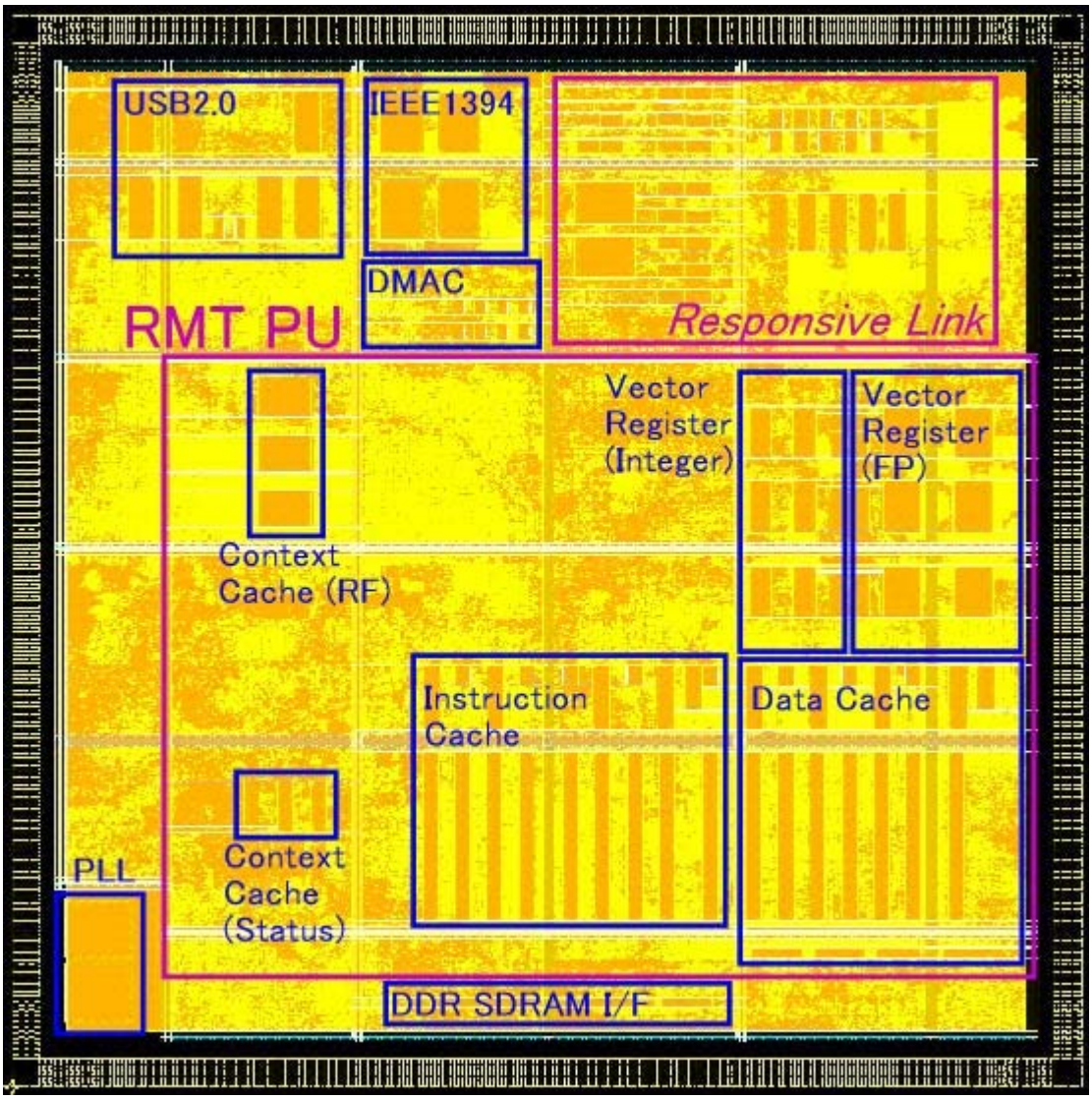

図4 RMT Processorのレイアウト図

RMT Processorの命令セットは、組み込み用プロセッサ等に多く用いられているMIPS命令セットアーキテクチャ互換で設計を行う。MIPSアーキテクチャはMIPSIからMIPSIVまで定義されているが、MIPSIとMIPSIIは32bitの命令セットのみで、MIPSIIIとMIPSIVは64bitの命令セットを持つ。本研究で設計するプロセッサは複数のスレッドをプロセッサ内に保持するために、複数のレジスタセットを持つ。データ幅を大きくすると、このレジスタセットが、全体のトランジスタ数に非常に大きく関わってくる。よってRMT Processorでは汎用レジスタは32bitとして、MIPSII命令セットアーキテクチャを基本とし、本プロセッサ固有の命令を追加する形で設計を行う。ただし、浮動小数点レジスタを用いることにより、64bit演算を行うことができるため、MIPSIIIの64bit命令を一部使用する。

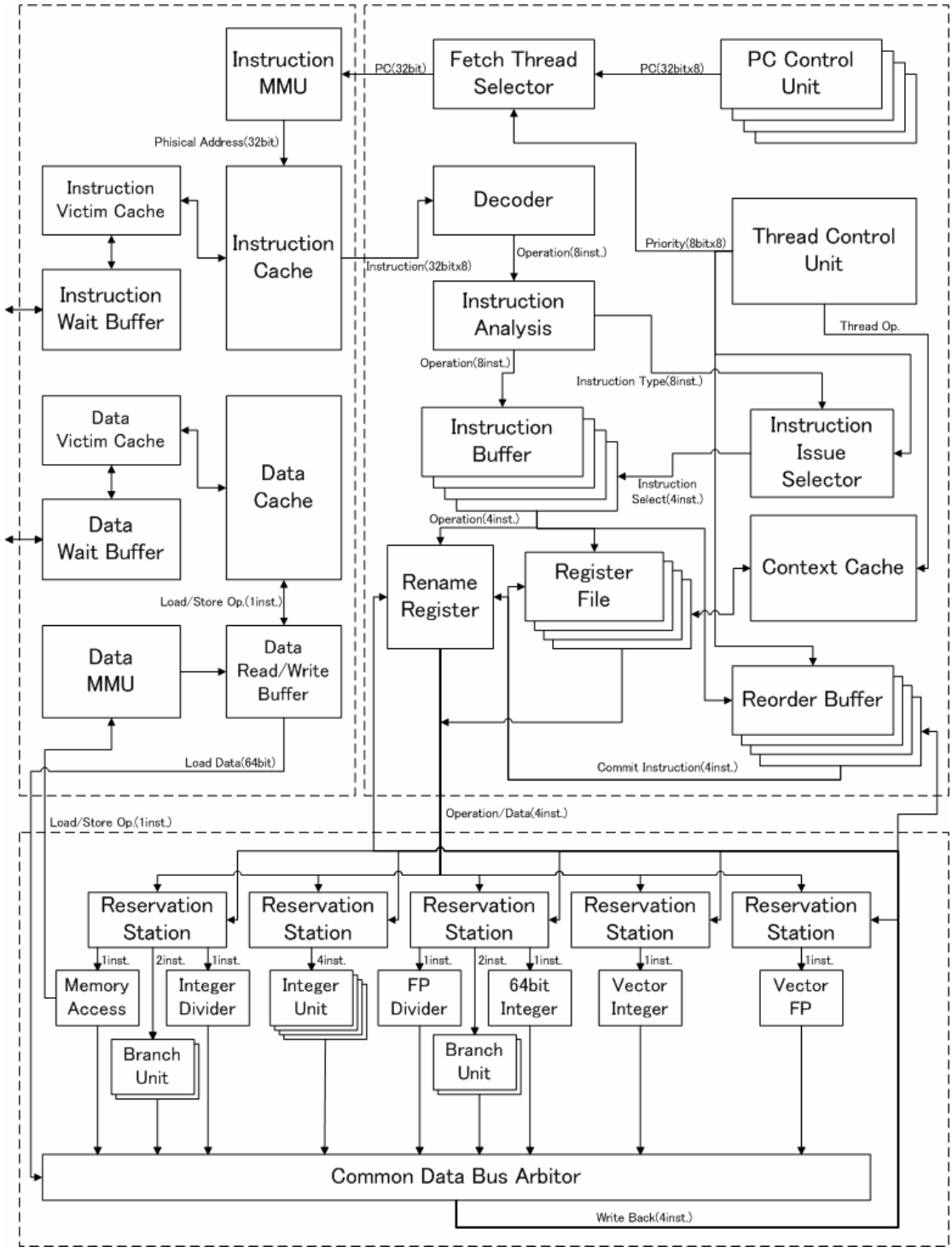

図5 RMT Processing Unitのブロック図

また、エンディアンも既存のMIPS命令セット用コンパイラの使用を考慮してビッグエンディアンとする。

図5にRMT PUのブロック図を示す。RMT PUは命令発行ユニット(Issue Unit)、命令演算ユニット(Execution Unit)、キャッシュユニット(Cache Unit)の大きく3つに分かれる。命令発行ユニットは各スレッドの実行を制御し、優先度に従って命令演算ユニットに対して各スレッドの命令を送る。命令演算ユニット

は命令発行ユニットから送られてきた命令を演算する。キャッシュユニットは命令発行ユニットからの命令フェッチ要求、命令演算ユニットからのデータアクセス要求を処理する。

RMT Processorのパイプラインステージは、次の通りである。

1. FS(Fetch Thread Select)ステージ
   命令フェッチユニット内でフェッチするスレッドが選択される。フェッチ可能な状態でありながら選択されなかったスレッドは次のクロックまで待機する。
2. IF1(Instruction Fetch 1)ステージ
   Instruction-MMUでアドレス変換を行う。
3. IF2(Instruction Fetch 2)ステージ
   命令キャッシュ(I-Cache)にアクセスする。キャッシュミスを生じた場合はメモリアクセスを行って命令が到着するまで待機する。
4. IA(Instruction Analysis)ステージ
   命令デコーダで4命令が並列にデコードされ、命令解析ユニットで分岐などの解析を経て命令バッファに格納される。命令でコーダ、命令解析ユニット、命令バッファはすべて命令ユニット内に位置する。
5. IS(Issue Instruction Select)ステージ
   命令バッファ内の命令から発行する命令が選択される。発行可能な状態でありながら選択されなかった命令は、次のクロックまで待機する。
6. REG(Register Renaming and Read)ステージ
   リネームバッファ(GP Rename Buffer、FP Rename Buffer)でデスティネーションレジスタのリネームとソースレジスタアクセスが行われる。同時にリオーダバッファ(Reorder Buffer)のエントリを命令に割り当てる。
7. RS1(Reservation Station 1)ステージ
   命令の種類に対応したリザベーションステーション(Reservation Station)に命令が格納される。
8. RS2(Reservation Station 2)ステージ
   命令が演算可能かどうかを判断され、実行可能となった命令の中から演算を行う命令が選択される。演算可能状態でありながら選択されなかった命令は次のクロックまで待機する。
9. EXE(Execution)ステージ
   整数演算ユニット(INT)や浮動小数点演算ユニット(FPU)といった演算ユニットで演算が行われる。演算の内容によってEXEステージのサイクル数は

様々である。

10. WB(Write Back)ステージ

演算を終了した命令がCDBアービタ(Common Data Bus Arbiter)を通ってリネームバッファとリオーダバッファにライトバックされる。同時にCDBの幅以上の命令がライトバック可能となった場合は、多くなった分の命令が次のクロックまで待機する。

11. CS(Commit Instruction Select)ステージ

リオーダバッファ内でコミット可能となった命令の中から実際にコミットされる命令が選択される。コミット可能で状態でありながらコミットされなかった命令は次のクロックまで待機する。

12. COM(Instruction Commit)ステージ

命令をコミットする。レジスタへのデータの書き込みを伴う命令の場合はリネームレジスタからレジスタファイル(GPR、FPR)へ値が書き込まれる。また、コミットした命令のPCが命令ユニット内のPC制御ユニットに送られる。

IAステージとISステージ、RS1ステージとRS2ステージ、CSステージとCOMステージ等は必ずしも切り分ける必要のないステージである。しかし、RMT PUではクロックサイクルタイムを短くする為に切り分けている。

次に命令発行ユニットについて述べる。

スーパースカラプロセッサは同時に複数の命令を発行することにより、プロセッサの性能を改善している。同時発行命令数を多くすることにより、IPCを向上させることが可能だが、複雑性が増してコストがかさみ、クロック周波数を上げられなくなるといった問題が生じる。同時発行命令数を制限する大きな要因としては次のようなものがある。

- レジスタセットの読み出しポート数
- 同時発行命令間の依存関係の解析

MIPSアーキテクチャではソースレジスタを2つ指定する命令があるので、毎クロックサイクルに同時発行命令数×2の読み出しアクセスが生じる。よって、レジスタセットの読み出しポートは同時発行命令数×2ポートが必要となる。レジスタポートが同時発行命令数に線形的に増加するのに対して、命令間の関係の解析は命令の組み合わせが組み合わせ関数的に増加する為に、急激に複雑さを増す。また、SMTで命令を発行するため、発行命令を複数スレッドから組み合わせることを考えると、さらに複雑さを増してクロックサイクルタイムに非常に大きな影響を与えると考えられる。

1、2、3ビットで表現できる2、4、8命令を候補として考慮した結果、2命令で

はIPCの向上が抑えられ、8命令では複雑さの面で無理があると考えられる。よって、RMT PUの同時発行命令数は4命令とした。以下、同時発行4命令を基準として各部位の命令数を決定していく。

- ハードウェアコンテキスト数

  通常、シングルスカラのプログラムを実行した場合、IPCは0.5程度となる。プロセッサの同時命令発行数である4命令を最大限有効に使用するためには、8スレッドを切り替えながら実行する必要がある。よって、ハードウェアコンテキストは8スレッド分設ける。

- フェッチバンド幅

  基本的にパイプラインの各ステージにおけるIPCはステージ毎に減少していき、増加することはありえない。発行命令数が4であれば、フェッチ命令数も4以上が必要である。本プロセッサではフェッチバンド幅は8命令分の256 bitとしている。これ以上大きくなると、命令をデコードし分岐の解析を行って次のフェッチアドレスを求める処理が複雑になるため、8命令フェッチとした。

- ライトバック命令数

  ライトバック命令数を大きくすると、リザベーションステーションやリオーダバッファのライトポート数が増え、トランジスタ数が増加する。よって、RMT PUでは同時発行命令数と等しく4命令とすした。

- コミット命令数

  同時発行命令数である4命令とする。ただし、リオーダバッファはプログラム順に命令を整列させる必要があるのでスレッド毎に別々に実装する。各スレッドに4命令ずつを対象とすると8スレッド分の合計32命令からスレッドの優先度に従ってコミットする4命令を選択するという処理が必要になる。コミット命令選択の部分を実装して検討した結果、32スレッドからの選択処理は遅延が大きすぎるという結論に達した。そこで、各スレッド2命令ずつの合計16命令からコミットする4命令を選択する。

- リネームバッファサイズ

  32レジスタとする。

- リオーダーバッファサイズ

  各スレッド16命令分とする。

次にスレッドの制御機構について述べる。

スレッドはプロセッサ内ではハードウェアコンテキストとして保持される。前述のようにハードウェアコンテキストは8スレッド分で、さらにコンテキスト

キャッシュを用いてより多くのスレッドに対して高速なコンテキストスイッチを提供する。スレッドの状態には大きく分けて次のようなものが考えられる。

- ハードウェアコンテキストとして保持(アクティブスレッド)
- コンテキストキャッシュに格納(キャッシュスレッド)
- 通常のメモリ内に格納

RMT PUでは、ハードウェアとしてスレッドがどの状態にあるかを管理する必要があり、それをスレッド制御ユニット(Thread Control Unit)で行う。

通常のメモリ内に格納されているスレッドに関してはソフトウェアでメモリの番地に書き込むという形で変更や管理をしたとしても、ハードウェア的に状態を管理する必要はない。以下に、ハードウェア的に管理する必要があるコンテキキャッシュ内に格納されているキャッシュスレッドとハードウェアコンテキストとして保持されているアクティブスレッドに関する設計の
詳細を述べる。

オンチップキャッシュの利点は高速なアクセス速度とチップ外に配置した場合よりもはるかに広いバンド幅でアクセスできる点である。レジスタセットの内容を広いバンド幅を用いて数メモリサイクル(オンチップキャッシュのサイクル)でバックアップやリストアすることが可能となる。

キャッシュスレッドに関するアプローチは次の3つが考えられる。

- ハードウェアで自動的にハードウェアコンテキストとの入れ替えを行う方法
- ソフトウェアで明示的にハードウェアコンテキストとの入れ替えを行う方法
- ソフトウェアとハードウェア両方から入れ替えを可能とする方法

ソフトウェアで明示的に入れ替えを行うことに問題はない。対してハードウェアで自動的に入れ替えを行う方法はメモリに対してバックアップを行う際等にソフトウェアで明示的に制御できない問題がある。そして、ハードウェアとソフトウェアの両方で入れ替えが可能という方法もあるが、ハードウェアが自動でスレッドの位置(ハードウェアコンテキストの番号やコンテキストキャッシュ内での番号)を変更してしまうと、ソフトウェアで制御しようにも例えば対象スレッドをコンテキスト番号で指定したとしても、タイミングによっては突然番号が変更されてしまうなど場合によっては致命的な問題を発生させる可能性がある。また、スレッドに一意の長いIDを割り当ててそのIDのみでアクセスをするとしても、今度は同期等の際に長いIDを比較に用いる必要があるなど、問題が多い。スレッドはOSによってハードウェアリソースを割り当てると

いう形で制御することが問題のない制御方法であると考え、コンテキストキャッシュとハードウェアコンテキストの間での入れ替えはソフトウェアによって明示的に行う形に限定する。

一方で、スレッドIDも必要である。コンテキストキャッシュ内のスレッドをIDによって制御できないと、OSはスレッドIDとコンテキストキャッシュ内の位置を完全な形で保持しておかなければならず、さらにIDの照合にも余計なレイテンシがかかってしまう。OS内で用いられるIDは通常、データ長の32ビットとなる。OSの制御に用いるIDを付け替える必要がないようにスレッドには32ビットのIDを付加し、それを用いて制御することを可能とする。キャッシュスレッドのIDはキャッシュテーブルに保持され、入力と比較して一致した場合はエントリ番号を返し、それを用いてコンテキストキャッシュへのアクセスが行われる。

ソフトウェアでコンテキストキャッシュに対して行う処理の内容を挙げる。

- バックアップ

  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へスレッドを移動させる。ただし、通常のデータとは異なり、上書きを許すとシステムに致命的な問題を引き起こす可能性があるため上書きはできず、コンテキストキャッシュに空きがないために処理を失敗する状況が生じた場合は、例外を引き起こす。

- リストア

  コンテキストキャッシュからハードウェアコンテキストへスレッドを移動させる。ハードウェアコンテキストに空きがない場合は例外を引き起こす。

- コピー

  スレッドをコピーするソース・デスティネーションともに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どちらかに限定する必要はない。プログラムにおいてスレッドを作成する場合、C言語のfork()関数に相当する関数をバックアップとリストアだけで実現するにはメインメモリを介する必要があるので非常に長いクロックサイクルを必要とする。ハードウェアでスレッドのコピーを実装することでコピーのレイテンシを大きく削減することが可能である。

- スワップ

  アクティブスレッドとキャッシュスレッドをスワップする。コンテキストキャッシュやレジスタの実装には双方向バスは用いず、リードポートとライトポートを別々に用意する。よって、読み出しと書き込みは同時に行うことが

可能であり、スワップ処理が容易に実現可能である。コンテキストスイッチの際のレイテンシを削減するためにスワップを実装する。

アクティブスレッドはスレッド制御ユニット内でスレッドテーブルによって管理され、テーブルの内容に従ってプロセッサ全体へ対する制御信号を生成する。

スレッドテーブルには優先度を示すフィールドがある。優先度はプロセッサ内の多くの部分で競合の調停に使われるが、遅延的にもゲート数的にも調停処理の大部分を占めるのが優先度の比較を行う比較器であり、優先度のビット幅が増えると比較器のサイズと遅延が増大し、調停機構のサイズと遅延もそれに従って増大する。逆に利用可能な優先度が少なすぎると、ハードウェアコンテキスト間の相対的な優先度の変化によって複数スレッドに対して優先度を変更するという処理が多く必要になるという問題がある。動的なシステムで性能を保証するだけの優先度数の指標はないが、静的に優先度をつけてRate Monotonicスケジューリングを行う場合には256レベルの優先度で十分なプロセッサ使用率を達成可能であるという証明がなされており、それを1つの基準として、RMT PUにおいても8ビットで256レベルの優先度を用いる。

また、スレッドテーブルにはアクティブスレッドの状態を示すフィールドがある。アクティブスレッドの状態は命令スケジューリングとの関係で大きく次の3つに分けられる。

1. 命令スケジューリングの対象となっている状態
2. 命令スケジューリングの対象とはなっていないが、いくつかの命令がパイプラインのREGステージ以降のいずれかのステージに存在している状態
3. 命令スケジューリングの対象となっておらず、発行されている命令もない状態

1の状態は通常の実行状態である。3の状態は停止状態であり、実行状態へ遷移する要求が到着した場合は遅延なしですぐに実行状態に移行することが可能であるほか、キャッシュスレッドへの移行もすぐに行うことができる。2の状態は1の状態から3の状態への移行途中の状態である。移行状態の必要性はスレッド制御命令の実装方法に関わってくる。自分自身のスレッドをコピーする場合を考えると、コピーするタイミングはスレッドをコピーする命令を実行し終わった時点である。ここでスレッドのコピーがコミット時に行われるとすると、自分自身の命令がライトバックしてコミットされるまでは他のスレッドからのスレッド関連のアクセスを拒否する必要があり、2の移行状態が必要になる。

各スレッドはバックアップ、リストア、コピー、スワップといった形でキャ

ッシュスレッドとアクティブスレッドの状態を移動する。スレッドのストップやバックアップやスワップの操作を自分自身に対して行う際にはスレッド状態を遷移させる命令終了後の状態を確実に保持する必要があり、専用の命令を用いて他スレッドに対する制御とは別の状態をとる。

以上で述べた機能を実現するためにMIPS命令セットに次のような命令を追加する。

- MKTH

  スレッドを生成する。スレッドIDと開始PCを指定する。

- DELTH

  スレッドを削除する。スレッドIDを指定する。

- CHPR

  スレッドの優先度を設定する。スレッドIDと新しい優先度指定する。

- CHSTATE

  HOLD、KEEPの設定を行う。スレッドIDと設定の内容を指定する。

- RUNTH

  スレッドの実行を開始する。スレッドIDを指定する。

- STOPTH

  スレッドを停止する。スレッドIDを指定する。

- STOPSLF

  命令を実行したスレッドを停止する。

- BKUPTH

  スレッドをコンテキストキャッシュにバックアップする。スレッドIDを指定する。

- BKUPSLF

  命令を実行したスレッドをコンテキストバックアップメモリにバックアップする。

- RSTRTH

  スレッドをコンテキストキャッシュからリストアする。スレッドIDを指定する。

- SWAPTH

  アクティブスレッドとキャッシュスレッドを入れ替える。スレッドIDを2つ指定する。

- SWAPSLF

  命令を実行したスレッドとキャッシュスレッドを入れ替える。スレッドID

を指定する。

- CPTHTOA

スレッドをハードウェアコンテキストの空いているエントリにコピーする。コピー元のIDと新しいスレッドに付けるスレッドIDを指定する。

- CPTHTOM

スレッドをコンテキストキャッシュの空いているエントリにコピーする。コピー元のIDと新しいスレッドに付けるスレッドIDを指定する。

- GETTT

スレッドテーブルの内容をレジスタに読み出す。スレッドIDを指定する。

- GETTID

アクティブスレッドのスレッドIDをレジスタに読み出す。アクティブスレッドの番号を指定する。

- GETOTID

命令を実行したスレッドのスレッドIDをレジスタに読み出す。

- GETMTID

コンテキストキャッシュ内のスレッドのスレッドIDをレジスタに読み出す。コンテキストキャッシュのアドレスを指定する。

- GETCNUM

コンテキスト番号をレジスタに読み出す。スレッドIDを指定する。

GETTT、GETTID、GETOTID、GETMTID、GETCNUM以外の命令は成功すると0、失敗すると0以外の値がデスティネーションレジスタに書き込まれる。

次に命令発行までの流れについて述べる。

最初に命令キャッシュから命令をフェッチする。スーパースカラやSMTでは1クロックサイクルに複数の命令をフェッチするので、シングルスカラのパイプラインプロセッサと異なり分岐命令をフェッチした時にディレイスロットでパイプラインストールを無くすことが出来ない。また、同時フェッチ命令が多くなるほど1度にフェッチしてきた命令中に分岐命令を含む可能性が高くなる。正確なアドレスを用いてフェッチするためにはデコード結果を受けてフェッチを行う必要がある。それを補う為のテクニックとしてBranch Target Buffer(BTB)を用いる方法がある。BTBは分岐命令の跳び先アドレスをバッファに保持し、フェッチを行ったアドレスを元に次にフェッチするアドレスを予測することができる。BTBを用いると分岐予測が当たっている限り毎クロックサイクルフェッチを行うことが可能である。

RMT PUはマルチスレッディングを行うため、フェッチするスレッドを選択す

る必要がある。選択に1ステージ(FSステージ)かかるとすると命令フェッチに3クロックサイクルかかる。さらに、RMT PUではメモリ管理を行っており、MMUを介して物理アドレスでキャッシュをアクセスするため、MMUによるアドレス変換のためにさらに1ステージ(IF1ステージ)が必要となり、普通に設計を行うと合計4クロックサイクルとなる。4クロックサイクル毎のフェッチではあまりにもIPCが低すぎるため、キャッシュから応答が帰ってくるIF2ステージと重複して投機的にフェッチを行う。フェッチスレッドは毎クロックサイクル1スレッドのみで、スレッド選択は単純に優先度に従う。

フェッチスレッド選択ユニット(Fetch Thread Selector)は、スレッド制御ユニットから送られてくる優先度に従って、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、命令キャッシュにプログラムカウンタ(PC)を送る。また、BTBの参照結果を基にしたプリフェッチ時のPCの生成やフェッチ要求したPCが無効になった場合にフェッチ命令を無効化する。

命令キャッシュから送られてきた命令は命令デコーダ(Decoder)でデコードされる。デコードはIAステージの前半で行われる。命令解析ユニット(Instruction Analysis Unit)はフェッチした8命令間の関係を解析する。デコード結果と分岐予測の結果を合わせて次にフェッチするアドレスとフェッチした命令毎の有効無効を決定する。解析結果は命令タイプテーブルに格納され、発行命令選択に使用される。

命令タイプテーブル(Instruction Type Table)はデコード後に解析された命令の情報を保持する。テーブルの内容は有効な命令であるかどうか、分岐予測の結果フェッチされてきた命令であるかどうか、命令の現時点での予測の深さ、特殊な命令であるかどうかとなっている。命令が発行されればそのテーブルエントリを無効化し、分岐の結果が判明すれば結果を命令の情報に反映させるといった処理を行う。

発行命令選択ユニット(Issue Instruction Selector)は優先度をもとに、発行可能な命令の中から4命令を選択し、命令バッファに発行指示を送る。
命令バッファ(Instruction Buffer)はデコードした命令を保持する。発行命令選択ユニットからの指示で命令を選択して発行する。

命令バッファから発行された命令は、レジスタファイルおよび、リネームレジスタをアクセスし、命令実行ユニットに送られる。また、命令の実行はOut of Orderに行われるため、リオーダバッファに命令が格納される。
次に命令実行について述べる。
RMT PUは以下の演算ユニットを持つ。

- Integer Unit

  32bit整数の論理演算、算術演算、比較、論理シフト、算術シフト、ローテーションを行う。プログラム中で整数演算は多く出現するためInteger Unitの数が少ないと資源の競合が多く発生してしまう。またこのユニットのゲートサイズは小さいので、Integer Unitは多い方が望ましい。しかし命令発行が1サイクルに4つのため多すぎても使用されないInteger Unitがでてきて無駄になる。よってRMT PUではInteger Unitを命令発行数と同じ4つとする。

- Complex Integer Unit

  整数の除算、剰余演算を行う。プログラム中で除算はそれほど多く発生せず、またパイプライン化することにより1サイクルに1つの演算を開始することができるので、RMT PUではComplex Integer Unitを1つとする。

- Branch Unit

  条件により分岐やトラップが発生するかどうかを判定する。分岐はプログラム中で多く発生し、またプロセッサの性能を向上させるためにはできるだけ早く分岐結果を求めなければならないため、競合を避けるためにBranch Unitは多く必要になる。しかし分岐結果を命令発行部へ返し、その結果によりPCの制御を行わなければならないため、1度に複数の結果を返すとPCの制御が複雑になる。よってRMT PUではBranch Unitを2つとする。

- Memory Access Unit

  メモリにアクセスするためのアドレス計算を行う。また、Thread Control Unitや各Control Registerのアクセスを行う。メモリやThread Control Unit、各Control Registerへのアクセスはin orderで行わなければならず、またData Memory Management Unitを1つしか持たないので、メモリアクセスを1サイクルで1つしかできないためにMemory Access Unitを複数持たせても無駄になってしまう。よってRMT PUではMemory Access Unitは1つとする。

- Synchronization Unit

  共有レジスタへのアクセスを行う。共有レジスタへのアクセスはリネームレジスタを用いていないため、in orderで行わなければならない。よってRMT PUではSynchronization Unitを1つとする。

- Floating Point Unit

  浮動小数点の算術演算、比較、整数と浮動小数点の変換を行う。浮動小数点のフォーマットはIEEEの単精度浮動小数点フォーマットと倍精度浮動小数点フォーマットを用いる。浮動小数点の演算を行うプログラムではこの演算

ユニットを多く使用する。よって浮動小数点演算のプログラムが多く実行されている場合はこのユニットが少ないと競合が多く発生してしまう。しかしこのユニットはゲートサイズが大きいため多くのFloating Point Unitを持たせることができない。よってRMT PUではFloating Point Unitを2つとする。

- Complex Floating Point Unit

浮動小数点の除算を行う。このユニットはゲートサイズが大きく、プログラムでは除算はそれほど多く発生しないため、RMT PUではComplex Floating Point Unitを1つとする。

- 64bit Integer SIMD Unit

64bit整数の算術演算、論理演算、比較、算術シフト、論理シフト、算術シフト、ローテーション、および 8bit×8、16bit×4、32bit×2の整数SIMDの算術演算、算術シフト、論理シフトを行う。このユニットで使うオペランドは浮動小数点レジスタを用いる。このユニットはゲートサイズが大きいので、ゲートサイズを抑えるために、RMT PUでは64bit Integer SIMD Unitを1つとする。

- Vector Integer Unit

整数のベクトル演算を行う。詳細は後に述べる。このユニットはゲートサイズが大きいので、RMT PUではVector Integer Unitを1つとする。

- Vector Floating Point Unit

浮動小数点のベクトル演算を行う。詳細は後に述べる。このユニットはゲートサイズが大きいので、RMT PUではVector Floating Point Unitを1つとする。

命令発行ユニットから送られてきた命令はリザベーションステーションに格納される。リザベーションステーションには次の3つのタイプがある。

- 演算ユニット1つに対して1つのリザベーションステーションを持たせるタイプ

この場合リザベーションステーションのエントリ数は1から2程度とすることが多い。

- 1つのリザベーションステーションで全ての演算ユニットにディスパッチを行うタイプ
- いくつかの演算ユニットがリザベーションステーションを共有するタイプ

RMT PUは15個の演算ユニットを持つ。1つのリザベーションステーションで15個の演算ユニットへ出力を行うとリードポートが多くなるため、遅延が大き

くなる。一方1つの演算ユニットに1つのリザベーションステーションを持たせるとゲートサイズが大きくなる。そこでRMT PUでは演算ユニットを複数のカテゴリに分け、各カテゴリに1つのリザベーションステーションを持たせる。各リザベーションステーションのリードポート数を小さくするために1つのリザベーションステーションに最大で4つの演算ユニットを接続する。各リザベーションステーションのエントリのビット幅はそのリザベーションステーションに格納される命令の最大幅になる。よってゲートサイズを抑えるためにはなるべく同じビット幅を必要とする演算ユニット同士を同じカテゴリにしたほうが良い。デコードされた命令のビット幅は各演算ユニットでほとんど差はないので、各演算ユニットで使われるオペランドによりカテゴリを分ける。ただし、Memory Access UnitとSynchronization Unitはin orderで命令をディスパッチする必要があるので、この2つのユニットで1つのポートを共有する。以下にリザベーションステーションへの割り当てを示す。

- Integer Reservation Station
  32bitオペランドを2つ必要とする演算ユニットのカテゴリ。4つのInteger Unitを接続する。
- Floating Point Reservation Station
  64bitオペランドを2つ必要とする演算ユニットのカテゴリ。2つのFloating Point Unit、1つのComplex Floating Point Unit、1つの64bit Integer SIMD Unitを接続する。
- Memory / Branch Reservation Station
  2オペランド以上を必要とする残りの演算ユニットのカテゴリ。2つのBranch Unit、1つのMemory Access Unit、1つのSynchronization Unit、1つのComplex Integer Unitを接続する。Memory Access UnitとSynchronization Unitはリザベーションステーションからの出力を共有する。
- Vector Integer Reservation Station
  1つの32bitオペランドを必要とするカテゴリ。1つのVector Integer Unitを接続する。
- Vector Floating Point Reservation Station
  1つの64bitオペランドを必要とするカテゴリ。1つのVector Floating Point Unitを接続する。

RMT PUでは複数のスレッドが並列に実行されているため、各演算器においてスレッド間で競合が起こる。命令演算ユニットではリザベーションステーションにおいて、優先度による制御を行う。リザベーションステーションでは演算

に必要なオペランドがそろうまで命令は保持される。演算に必要なオペランドがそろい命令の実行が可能になると、各演算器に対して命令が発行される。RMT PUでは複数の命令が実行可能になった場合、リザベーションステーションは各命令の優先度を調べ、優先度の高い命令から先に演算器に発行する。これにより優先度の高いスレッドの命令に対して、先に演算器を割り当てる。

一方、マルチメディア処理のようなソフトリアルタイム処理では多くのデータを繰り返し演算しなければならないため、高い演算性能が要求される。このような処理ではデータの並列性を利用して演算性能を高めることができる。

データ並列性を利用して演算性能を向上させる方法として、SIMD演算とベクトル演算がある。

SIMD(Single Instruction Stream Multiple Data Stream)演算は1つの命令で複数のデータを演算する。汎用プロセッサでは1つのレジスタを複数の領域に分割して、それぞれの領域に対して同じ演算を並列に行う。既存のレジスタファイルを利用することによりハードウェア量の増加を防ぐことができ、演算にかかるレイテンシも小さいが、演算の並列度は小さい。ベクトル演算はベクトルレジスタを用いてベクトル要素をパイプライン的に演算する。ベクトルレジスタに対してデータのLoad / Storeを一度に行うため、メモリアクセスによるオーバーヘッドが小さい。ベクトルレジスタのベクトル長を長くすることにより並列度を上げることができる一方、演算にかかるレイテンシが増加し、ハードウェア量も増加する。

先に述べたように、RMT PUでは優先度を用いて計算資源の調停を行っている。この時、命令キャッシュへのアクセスが優先度による調停の影響を大きく受ける。通常は優先度の高いスレッドが命令キャッシュをアクセスし、長いレイテンシの命令などで優先度の高いスレッドが命令キャッシュをアクセスしない時に、より優先度の低いスレッドが命令キャッシュをアクセスして命令を実行する。ハードリアルタイム性を持つスレッドの時間制約を保証するために、時間制約の厳しくないソフトリアルタイム性を持つスレッドはハードリアルタイム性を持つスレッドよりも低い優先度で実行される。そのため、ハードリアルタイムのスレッドが実行されている間は、ハードリアルタイムのスレッドが命令キャッシュをアクセスしない時にのみ、ソフトリアルタイムのスレッドの命令がフェッチされる。この命令フェッチを有効に使用することにより、ソフトリアルタイム処理の性能を向上することができると考えられる。

そこでRMT PUでは少ない命令数で高い並列度を実現することができるベクトル演算を用いる。これにより少ない命令フェッチで高い演算性能を実現する。

また、優先度の高いスレッドの実行が完了し、命令フェッチが多く行われるようになった場合でも、細粒度マルチスレッディングにより、ベクトル演算にかかるレイテンシは、別のスレッドを実行することで隠蔽される。
RMT PUでは細粒度マルチスレッディングにより、複数のスレッドが並列に実行される。ベクトル演算でマルチスレッディングを用いることによりメモリアクセスの効率を向上し、ベクトル演算の性能が向上する。しかしベクトル演算の並列度を大きくするとベクトルレジスタに必要なハードウェア量も増加するため、スレッド毎に大量のベクトルレジスタを持たせるとことはできない。一方、リアルタイムシステムでは複数のプログラムが並列に実行されるために、ベクトル演算を行うスレッドを1スレッドとし、他のプログラムを並列に実行する場合も考えられる。このようにシステムやプログラムにより、ベクトル演算に必要となるスレッド数やベクトルレジスタの構成は異ってくる。このような場合、ベクトルレジスタの構成を固定してしまうとベクトルレジスタを効率良く利用できない。そこでRMT PUではベクトルレジスタを共有して使用し、動的にベクトルレジスタの構成を変化させることにより複数のスレッドで柔軟なベクトル演算を可能とする。

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

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

ベクトルレジスタの確保は、Vector Reserve(VRES)命令で行う。この時、ベクトル演算に必要なベクトルレジスタ数と、その構成を指定する。また演算を行い、これ以上ベクトル演算器を使用しなくなった場合は、Vector Release(VREL)命令で確保していたベクトルレジスタを開放することにより、別のスレッドが新たにベクトルレジスタを確保し、ベクトル演算を行えるようにする。

実際にベクトル演算を行う場合、デコードされたレジスタIDとテーブル内に

保持されている割り当て情報とベクトル長からデータが格納されているベクトルレジスタの実アドレスを計算し、ベクトルレジスタにアクセスする。

整数ベクトル演算器の演算パイプラインでは算術、論理、シフト、比較および積和演算を行う。パイプラインはベクトルレジスタのread、演算に2段、ベクトルレジスタへのwriteの計4段で構成され、完全にパイプライン化されている。また1つの演算パイプラインで8つの整数演算器を持つことにより1クロックで8つのベクトル要素を並列に演算する。浮動小数点ベクトル演算器の演算パイプラインは算術、比較、積和演算および整数フォーマットとの変換を行う。パイプラインはベクトルレジスタからのread、演算に5段、ベクトルレジスタへのwriteの計7段で構成され、完全にパイプライン化されている。また1つの演算パイプラインで4つの浮動小数点演算器を持つことにより1クロックで4つのベクトル要素を並列に演算する。さらに、整数ベクトル演算器、浮動小数点ベクトル演算器共に、2つの演算パイプラインを持つことにより、複数のスレッドでベクトル演算を行った場合に、並列に演算を行うことが可能である。割り算は使用頻度が低いが、演算パイプラインで割り算を行うことができない場合、全てのベクトル要素をスカラの割り算器を用いて演算しなければならないため、処理効率が悪くなる。よってRMT PUでは2つある演算パイプラインのうち、片方のみ割り算器を持たせる。整数割り算器はパイプライン化され、1クロックに1つのベクトル要素の割り算を開始する。一方浮動小数点割り算器はパイプライン化されていない。

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

次にキャッシュシステムについて述べる。

キャッシュシステムの役割は命令発行ユニットから送られてくる命令フェッチ要求と、命令実行ユニットから送られてくるデータアクセス要求を処理することである。

キャッシュユニットはMMU(Memory Management Unit)を持ち、ハードウェアでアドレス変換を行うため、各スレッドは仮想アドレスを用いてプログラミングを行うことが可能である。MMUにはアドレス変換を行うためのTLB(Translation Look-aside Buffer)が64エントリある。MMUが置かれる場所により、仮想アドレスでキャッシュをアクセスするか物理アドレスでキャッシュをアクセ

スするかが決まる。仮想キャッシュを用いるとアドレス変換に必要となる時間を省略できるため、より高速なキャッシュアクセスが可能となる。しかしTLBエントリに設定されているページ単位での各種設定情報を、キャッシュタグと共にキャッシュライン単位で保持しなければならないためオーバーヘッドが大きくなる。また、コンテキストスイッチにより、実行されているスレッドが変更された場合にキャッシュ全体のフラッシュが必要となる。これには最低でもキャッシュの深さと同じだけのクロック数を要するため、オンチップのコンテキストキャッシュからわずか4クロックでスレッドの切替えが可能であるという特徴を完全に損ねてしまう。また複数スレッドでメモリ領域を共有する場合に、スレッド毎に異なる仮想アドレスを共有メモリ領域に割り当ててしまうと、同じ物理メモリのデータが複数キャッシュされてしまう問題(synonym)が発生する。これはキャッシュメモリの利用効率を低下させるだけではなく、それら並存するデータ間でのデータ一貫性の維持が必要となる。更にI/Oのデータをキャッシュした場合にもデータ一貫性の問題が発生する。RMT PUは多数のI/Oインタフェースを内蔵しており、I/Oから直接読み出したデータや、メモリを介して読み出したI/Oのデータをキャッシュすることがある。このような場合、I/O側のデータが更新されるとキャッシュされているデータは無効化されなければならない。しかしI/Oバスのデータは物理アドレスを用いて転送されている。つまり通常のアドレス変換とは逆となる物理アドレスから仮想アドレスへの変換を行わなければ、仮想キャッシュのデータを無効化することはできない。I/Oのアドレスを変換するためのTLBエントリを特別に設けるか、全TLBエントリで逆変換を可能にするような機能が必要となる。

一方、物理キャッシュを用いると、アドレス変換に必要な時間だけキャッシュアクセスが遅くなり性能を低下させてしまう。しかしキャッシュメモリや下位メモリのアクセス遅延を隠蔽することがマルチスレッドの目的の1つであり、out-of-order実行なども含めて完全ではないにしろアドレス変換によるアクセス時間の増加分はある程度吸収できると考えられる。またアドレス変換と同時に保護情報の確認を行うことが可能となるため、ページ単位でのメモリ保護や、キャッシュでのスレッド間のメモリ保護が容易となる。このようにタグが物理アドレスになることは、ある物理メモリのデータは常に1つしかキャッシュされないことになる。つまり共有メモリ領域のsynonym問題は起こらない。そのためデータの一貫性が崩れることはなく、その維持のための機構も必要がない。そしてI/Oのデータの無効化もその物理アドレスを直接用いて容易に行うことができる。

以上より、RMT PUではMMUはキャッシュよりも前に置かれ、キャッシュアクセスを行う前にアドレス変換を行い、キャッシュは物理アドレスを用いてアクセスする。

RMT PUのキャッシュシステムの特徴を以下に示す。

- 32KB 8-way set-associative方式
- ブロックサイズ、ラインサイズともに32byte
- Look Through
- ノンブロッキング
- 下位メモリとのデータ一貫性の維持はライトバック方式
- 書き込み要求ミスの処理はライトアロケート方式
- キャッシュポートは1ポート
- 物理タグでデータを保持
- 転送ブロック数が可変
- キャッシュのロックが可能
- 3サイクルのアクセス遅延
- マルチタグ、シングルデータ方式
- 16エントリのvictim buffer
- 最大16個のキャッシュミスを同時に保持
- 入れ換えを行うブロックの選択方法はLRUと優先度の2通り
- バス待ちキューでの優先度による要求の追い越しが可能

RMT PUでは命令やデータキャッシュのアクセス要求は必ずキャッシュシステムを通るようになっている。このような構成はLook Through方式と呼ばれる。命令やデータの要求がキャッシュでヒットしている限りは内部バスに要求が出ることはなく、高速なキャッシュアクセスが可能となる。逆に内部バスへ直接出す要求には数クロック無駄な時間が増えてしまうが、要求を処理する流れを複数に分けるとその制御が複雑になってしまう。また内部バスはアクセス自体に時間がかかるため、数クロック程度ならば大きなオーバーヘッドにならないと考えられる。ノンブロッキングキャッシュとは、あるキャッシュ要求がキャッシュミスを起しても、そのミス処理と並行して後続のキャッシュ要求を許可する方式である。キャッシュシステムでは最大16個のミスを処理することが可能である。

キャッシュメモリの容量はチップ面積を考慮して32KBとした。従って8つのスレッドが同時に使用した場合でも、平均して各コンテキスト1ページ分のデータをキャッシュすることが可能である。キャッシュポートの数は1ポートと

する。これは命令・データキャッシュ共に1クロックで1つの要求しか出さないためである。

キャッシュブロックとキャッシュラインのサイズは共に32byteとした。これはメモリコントローラとの内部バスの帯域幅と同じであり、ブロックの転送とキャッシュへの配置を低遅延で行うことができる。またブロックサイズとラインサイズを一致させることで、複数のスレッドの同時実行により頻繁にキャッシュラインの入れ換えが行われても容易に管理を行うことが可能である。一方、キャッシュからの追い出しをブロック単位で行うことも考えられる。その場合、特にデータキャッシュでは変更されたデータをメモリに書き戻さなければならないため、数クロックに渡りキャッシュが利用不可能になってしまう。逆にラインサイズよりもブロックサイズを小さくすることは実質的には不可能である。これは1ラインにつき1組のタグで管理しているためである。つまり、現在のラインサイズよりも小さいブロックを複数そのラインに割り当てるためには、その数だけタグを用意しなければならない。結局それはキャッシュのラインサイズを小さくして連想度を高めているのと同じことになる。また命令フェッチ要求やベクトルレジスタからのデータ要求は 32byte単位で行われているため、小さなキャッシュラインでは複数のキャッシュラインを読み出す必要があり、数クロックキャッシュが利用不可能になってしまう。逆にもっと大きなキャッシュラインでは読み出したデータから必要なデータを選択しなければならない。このようにメモリバスからキャッシュライン、命令要求、データ要求までのバス幅とブロックサイズを統一することで、より高速で容易なデータ転送を実現することができる。

キャッシュの連想度はハードウェアコンテキスト数と同じ8とした。これにより各スレッドが平均して1-wayを利用可能となるため、ある程度のサイズまでのデータは確実に競合ミスを減少させることができると考えられる。ただしキャッシュ容量自体は32KBのため、1-way当たりの割り当て容量は4KBとなる。

キャッシュブロックの入れ換え方式は、従来のLRU(Least Recently Used)方式に加え、優先度を用いて入れ換えることができる。優先度を基に入れ換えるブロックを選択する場合、より優先度の低いスレッドが使用しているブロックから先にキャッシュを追い出される。これにより、優先度の高いスレッドのキャッシュブロックが追い出されることを防ぎ、高優先度のスレッドの実行を優先させる。

キャッシュ入れ換えにより、キャッシュを追い出されたブロックは1度Victim Cacheに格納される。Victim Cacheは、追い出されたブロックをfull asso

ciative方式で保持する。キャッシュミスを起した場合、Victim Cacheにデータ(ブロック)が残っていれば、そのブロックをキャッシュに戻すことにより、キャッシュミスによる内部バスへの要求を減らし、メモリアクセスのレイテンシを減少させる。

キャッスミス等によりバスを介して下位メモリをアクセスする場合にも優先度を用いた制御を行う。メモリアクセスはキャッシュよりも低速なため、待ち行列が発生する。この場合、より優先度の高いスレッドからバスを使用して下位メモリにアクセスすることにより、高い優先度のスレッドの実行を優先する。

(2)得られた研究成果の状況及び今後期待される効果

評価は実チップの設計・実装に用いたRTLによるシミュレーションによって行った。まず優先度を用いた計算資源の調停による評価を行うために、8スレッドで逆離散コサイン変換を行うプログラムを作成し、それぞれのスレッドに優先度を付与した。これらのスレッドを従来のシングルスレッドのプロセッサを用いて実行した場合、細粒度マルチスレッディングによる実行を行った場合、RMT Processorで優先度用いて実行した場合についての評価を行った。

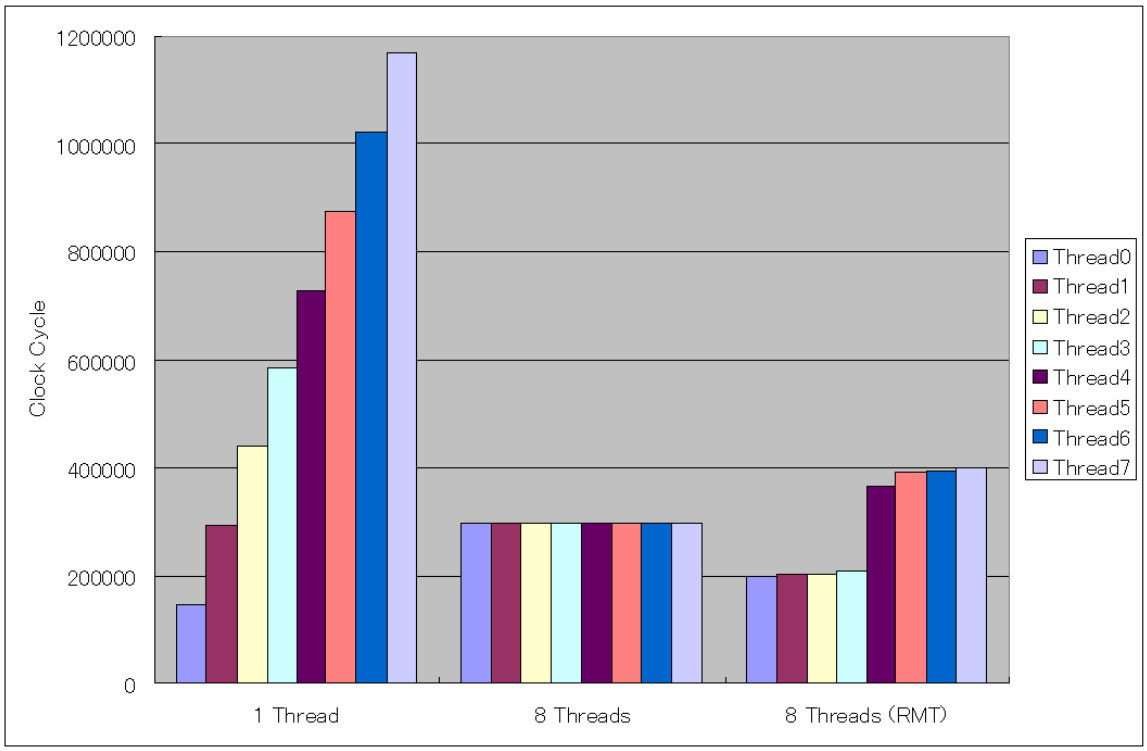

図6 逆離散コサイン変換を実行した場合の実行時間

図6に実行結果を示す。1 Threadはシングルスレッドのプロセッサで実行した結果、8 Threadは細粒度マルチスレッディングを行った場合の結果、8 Threa

d(RMT)はRMT Processorを用いて行った結果を示している。また、Thread0の優先度が最も高く、Thread7の優先度が最も低い。シングルスレッドのプロセッサで実行した場合、図1に示したように、優先度の高いスレッドから1つずつ順番に実行されるため、優先度の順に段階的に実行時間が増加している。細粒度マルチスレッディングを用いて実行した場合、依存関係の少ない命令を実行でき、ILPを向上させられるため、プロセッサ全体のスループットが向上している。しかし、最も優先度の高いスレッドと、最も優先度の低いスレッドの実行時間に差がなく、最も高いスレッドの実行時間が増加してしまっている。一方、RMT Processorを用いて実行した場合、優先度による計算資源の調停を行っているため、優先度の高いスレッドから順番に実行が完了している。また、実行するスレッド数が8スレッドなので、コンテキストスイッチなしに、実行することができるため、シングルスカラのプロセッサで実行した場合よりもスループットが向上している。

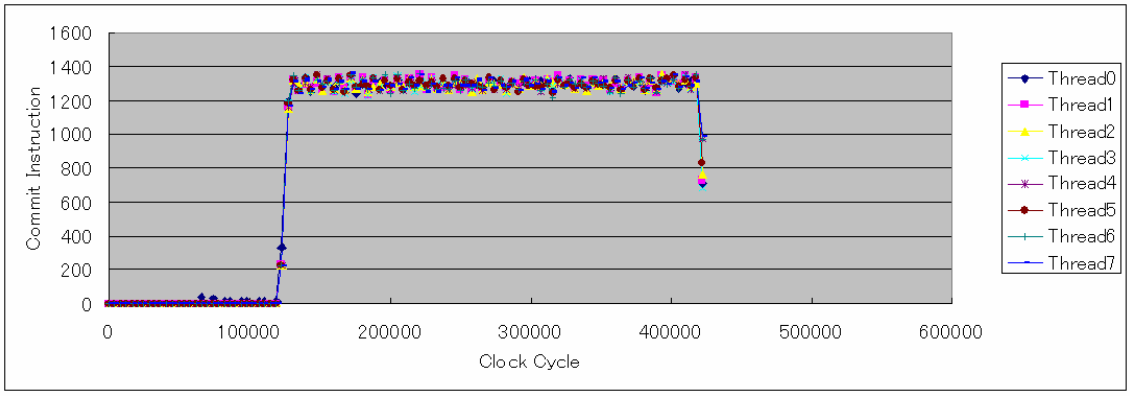

図7 細粒度マルチスレッディングにおける単位時間当たりの命令実行数

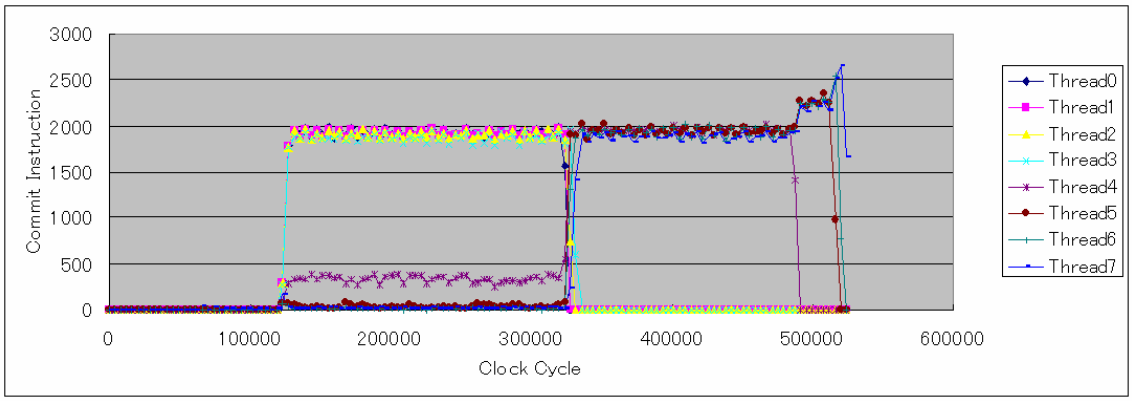

図8 RMT Processorにおける単位時間当たりの命令実行数

図7に細粒度マルチスレッディングを用いた場合の、各スレッドの単位時間当たりの命令実行数、図8にRMT Processorを用いた場合の単位時間当たりの命令実行数を示す。細粒度マルチスレッディングの場合は、各スレッドはラウン

ドロビンで実行されるため、全てのスレッドに対して同じ数だけ命令が実行される。一方RMT Processorでは、優先度の高いスレッドに対して命令が多く実行されている。またThread4のように優先度の低いスレッドでも、高い優先度のスレッドがストールしている場合には、細粒度マルチスレッディングにより、直ちに実行されるため、少しずつ命令が実行されている。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Thread6やThread7は実行されず、優先度の高いスレッドの実行が完了すると、次第に優先度の低いスレッドの実行が開始されている。

次にコンテキストキャッシュの評価を行うために、割り込みにより起動し、8スレッド分のコンテキストを入れ換えるプログラムを作成した。これを用いて、コンテキストキャッシュを用いた場合と、コンテキストキャッシュを用いない場合について、割り込みがかかってから8スレッド分のコンテキストスイッチが完了するまでの時間を測定した。コンテキストキャッシュを用いない場合は、Load / Store命令を用いてメモリとの間でコンテキストの入れ換えを行い、コンテキストキャッシュを用いた場合は、SWAP命令を用いてコンテキストキャッシュとの間でコンテキストの入れ換えを行った。

コンテキストキャッシュを用いない場合、7770クロックサイクルかかったのに対し、コンテキストキャッシュを用いた場合、390クロックサイクルでコンテキストスイッチが完了した。よって、コンテキストキャッシュを用いることにより、コンテキストキャッシュにかかるオーバーヘッドを大幅に削減した。

次に、ベクトル演算ユニットの評価を行うために、逆離散コサイン変換をベクトル演算で行うプログラムを作成した。また、RMT Processorの優先度付実行の影響を評価するために、スカラ演算で逆離散コサイン変換を行うプログラムと並列に実行した。この時、ベクトル演算を行うプログラムとスカラ演算を行うプログラムでは、命令フェッチ(命令キャッシュアクセス)、リザベーションステーションへの命令発行スロット、メモリアクセス(データキャッシュアクセス)、リオーダバッファからのコミット処理において資源競合が起きる。

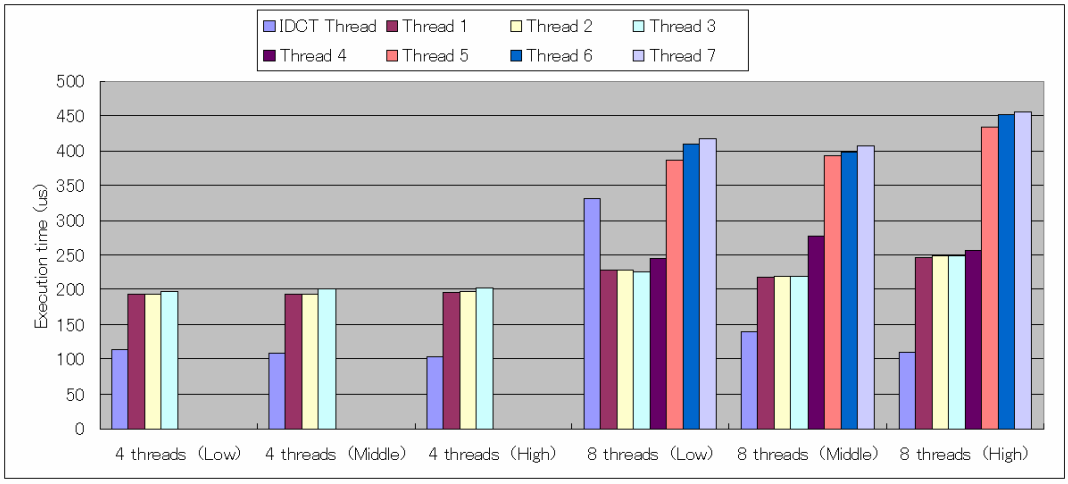

図9 ベクトル演算の実行結果

図9に実行結果を示す。IDCT Threadはベクトル演算を用いて逆離散コサイン変換を行うスレッドでThread 1から7はスカラ演算を行うスレッドを示している。またスカラ演算を行うスレッドはThread1が最も優先度が高く、Thread 7が最も優先度が低くなるように設定した。LowはIDCT Threadの優先度を全スレッド中最低にした場合、Middleは優先度を全スレッドの中間にした場合、Highは最も優先度を高くした場合を示している。

実行しているスレッドが4スレッドの場合、ベクトル演算を行うと優先度が低い場合でも1スレッド単体で実行したときとほぼ同じ実行時間を実現している。これはベクトル演算により、少ない命令フェッチを有効に利用することができたためだと考えられる。一方、8スレッドが並列に実行されている場合、ベクトル演算を行うスレッドの優先度が低い場合は、優先度による計算資源の調停のために、逆離散コサイン変換を行うスレッドの実行があとになるため、単独で実行した場合に比べて実行時間が増加している。また、優先度を中間にした場合においても、実行するスレッド数が多いため、優先度の高いスレッドが実行されている間は、ベクトル演算の命令フェッチが4スレッドの時ほど頻繁に行われず、単独で実行した場合に比べて実行時間が増加している。優先度が高い場合は、優先的にベクトル演算の命令フェッチが行われるため、スレッド数が増加した場合でも実行時間は増加しなかった。

これらの研究は、JSTにおいて世界に先駆けて行った研究であり、ハードウェアレベルで実時間処理・通信をμ秒オーダでサポートするプロセッサは、現在においても他に存在しない。また、アカデミックにおいて、1000万ゲート規模のLSIを設計・実装できるところは他に例を見ない。

Responsive Linkは、現在、国内では学会試行標準として標準化を行ってい

る。国外においては、ISO/IEC SC25 WG4において標準化作業を行っている。リアルタイム通信における国内および国際標準を目指している。標準化が実現されると、異なるリアルタイムシステム間をシームレスに接続可能になり、大規模な分散リアルタイムシステムを構築可能になる。また、分散リアルタイムシステムを構築するための、基盤プロセッサ、基盤通信リンクとなりうる。低消費電力と共に、QoS(リアルタイム性)、多機能、高い演算性能及び通信性能を実現しているので、非常に様々な分野で利用できると考えられる。

４．研究実施体制

(1)体制

慶應義塾大学理工学部
代表者 山﨑 信行

(2)メンバー表

氏　名	所　属	役　職	研究項目	参加時期
山﨑 信行	慶應義塾大学理工学部	専任講師	全ての研究項目	平成13年3月～平成15年3月

５．研究期間中の主な活動

(1)ワークショップ・シンポジウム等

なし

(2)招聘した研究者等

なし

６．主な研究成果

(1)論文発表　（国内 2件、海外 1件）

1. 山﨑信行, 松井俊浩,「並列分散リアルタイム制御用レスポンシブプロセッサ」, 日本ロボット学会誌, Vol.19, No.3, pp.68-77, 2001.
2. Nobuyuki Yamasaki, “Design and Implementation of Responsive Processor for Parallel/Distributed Control and Its Developing Environment,” Journal of Robotics and Mechatronics, Vol.13, No.2, pp.125--133, 2001.
3. 伊藤 務, 山﨑信行,「Responsive Multithreaded Processorの命令実行機構」, 情報処理学会ACS論文誌, No.3, 2003年7月掲載予定

(2)口頭発表

①招待、口頭講演　（国内 4件、海外 2件）

1. Nobuyuki Yamasaki, “Responsive Processor for Parallel/Distributed Real-Time Control,” IEEE/RSJ International Conference on Intelligent Robots and Systems, pp.1238-1244, 2001.
2. Masato Uchiyama, Tsutomu Ito, Jun-ichi Sato, Nobuyuki Yamasaki, and Yuichiro Anzai, “A New Processor Architecture for Real-Time Systems,” IFAC Conference on New Technologies for Computer Control, pp.377-382, 2001.
3. 伊藤 務, 内山 真郷, 佐藤 純一, 薄井 弘之, 松浦 克彦, 山﨑 信行, 「Responsive Multithreaded Processorの設計・実装」, 情報処理学会研究報告 2003-ARC-145, pp.31-36, 2003.
4. 薄井 弘之, 内山 真郷, 伊藤 務, 山﨑 信行, 「Responsive Multithreaded Processorの同期機構の設計と実装」, 情報処理学会研究報告 2003-ARC-145, pp.37-42, 2003.
5. 松浦 克彦, 伊藤 務, 山﨑 信行, 「マルチスレッド技術を用いたマルチメディア処理向けベクトルユニットの設計と実装」, 情報処理学会研究報告 2003-ARC-145, pp.49-54, 2003.
6. 一ノ瀬 信征, 佐藤 純一, 山﨑 信行, 「Responsive Multithreaded Processor用バス機構の設計と実装」, 情報処理学会研究報告 2003-ARC-145, pp.43-48, 2003.

②ポスター発表　　（国内 1件、海外 0件）

1. 山崎信行, 伊藤務, 内山真郷, 安西祐一郎, 「柔軟なマルチメディア処理機構を有したリアルタイムプロセッサアーキテクチャ」, ロボティクス・メカトロニクス講演会’01講演論文集, No.2P1-N7, pp.1-2, 2001.

③プレス発表

なし

(3)特許出願（国内 2件、海外　　件）

① 国内

1. 山﨑 信行, 「コンテキスト切り替え方法及び装置、中央演算装置、コンテキスト切り替えプログラム及びそれを記憶したコンピュータ読み取り可能な記憶媒体」, 特願2003-3038, 2003年1月9日

2. 山﨑 信行, 「命令発行方法及び装置、中央演算装置、命令発行プログラム及びそれを記憶したコンピュータ読み取り可能な記憶媒体」, 特願2003-83001, 2003年3月25日

②海外

1. 上記1を出願予定

(4)新聞報道等

①新聞報道

なし

②受賞

1. 山﨑 信行, 日本機械学会 ロボティクス・メカトロニクス部門 ベストプレゼンテーション賞, 2001年6月
2. 山﨑 信行, 日本ロボット学会論文賞, 2002年10月

③その他

なし

(5)その他特記事項

なし

7．結び

なし