東海大学大学院 2002 年度 修士論文

コヒーレンシ制御不要なバッファを用いた
レイテンシ隠蔽アーキテクチャを適用したプロセッサ及び
スループット向上を目的としたシステムコントローラの設計

指導 清水 尚彦 助教授

東海大学大学院 工学研究科
電気工学専攻

孕石 裕昭

目 次

図 目 次

表 目 次

第1章　序論

コンピュータシステムの性能の中で演算性能の向上が注目されている. 現在, プロセス技術の発展と様々な並列演算アプローチによって演算性能は順調に向上している. しかし, コンピュータシステム全体の性能としてみるとそれほど急激な性能向上とはならない. 特に数値計算分野や大規模なデータベースを扱うプログラムなどでは利用するメモリ量, 要求される演算性能が重要となる. この分野で要求性能を満たすシステムを構築するためには演算性能とともにメモリ性能が重要となってくる.

メモリ性能は大きく 2 種類に分類される. 1 つはプロセッサからのデータ要求が発行されてからそのデータがプロセッサに到着するまでの時間であるメモリレイテンシ. もう 1 つはプロセッサに対して単位時間あたりどれくらいデータが転送できるかの指標であるスループットである. まず, メモリレイテンシは近年メインメモリの主流である DRAM(SDRAM,DDR-DRAM,RDRAM 等) のレイテンシが 40〜70ns であり, システムとして構築した場合のレイテンシは 80〜400ns ほどとなる. 例えば動作周波数 2GHz のプロセッサで, システムのメモリレイテンシが 200ns とすると, プロセッサは要求したデータを利用できるようになるまで 400 命令の間このデータに対する演算を行うことができず, その間プロセッサは有効な演算を行うことができないため, 事実上停止してしまうことになる. そのため, メモリレイテンシに対する耐性を持たせた技術であるキャッシュメモリ,Load/Store バッファ,Out-of-Order 等の技術が現在のシステムでは広く用いられている. これらの技術に関しては次節で述べる.

次にスループットであるが, メモリシステムからプロセッサへの単位時間当たりのデータ転送量であり, 理論スループットは式 1.1 で表される.

$$Throuput = \text{バス幅} * \text{動作周波数}\ [MB/S] \quad (1.1)$$

式 1.1 は理論的な限界であり, プログラムのデータ構造の扱いにより, 実行スループットが変化する. ほとんどのコンピュータシステムにおいて連続的なメモリアクセスについては高い実効スループットを出すことができるが, 不規則なメモリアクセスについての対応は従来の機構ではあまりなされていない.

これはプログラムのデータの参照が連続的, または (時間的, 空間的に) 局所性の高いものであるためであり, このような場合の性能向上がまず第一に検討されるからである.

しかし, 今日のプログラムは大規模化, 複雑化されており, データアクセスのパターンを抽出することが困難である.

そこでまず, 従来から用いられてきたレイテンシ耐性, スループット向上技術に注目し, 新たな試みについて検討することにする.

1.1　メモリレイテンシ隠蔽技術

メモリレイテンシ隠蔽技術はプロセッサにおいて適用されており, 代表的なものとしてキャッシュメモリ,Load/Store バッファ, Out-of-Order 機構,Prefetch 技術について述べる

1.1.1　キャッシュメモリ

キャッシュメモリはプログラムのメモリ参照が時間的, 空間的に局所性を持つという特性を利用したレイテンシ隠蔽技術である. 小規模だが高速なメモリを用いて局所的で, 再利用性の高いメモリアクセスに対して効果を発揮する. キャッシュメモリの構成を図 1.1 に示す.

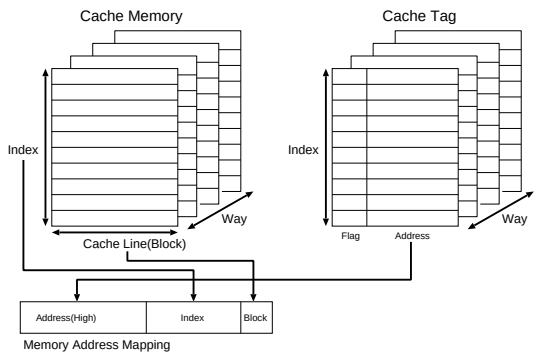


図 1.1: キャッシュメモリの構成

キャッシュメモリは容量や連想度を大きくすると高速な動作が困難となるため, キャッシュメモリを複数の階層で持つようなシステム構成をとっている. 図 1.1 のようにキャッシュ内のデータはキャッシュタグというものにアドレスが保持されており, ハードウェアによって管理されている. このため, キャッシュメモリの構成が変更されてもプログラム上で意識することなくプログラムを作成できるという利点がある. これは重要な技術であり, 同一アーキテクチャにおいて, キャッシュメモリの構成, 容量が変化した場合においても, プログラムを変更することなくバイナリを実行することが可能である.

キャッシュメモリはメインメモリのコピーであり, メモリリクエストの一貫性の保証をハードウェアにより行っている. この一貫性の保証のことをコンシステンシまたはコヒーレンシと呼ぶ.

一貫性保証プロトコルとマルチプロセッサ

キャッシュメモリはメインメモリのコピーを保持し, 過去に参照されたアドレスとその周辺のデータを 1 ブロック (キャッシュライン) 単位で保持している. このブロックにコピーされているアドレスに対して他のプロセッサや DMA コントローラからの書き込みが行われる場合, メインメモリとキャッシュメモリに保持されたコピーとの間で整合性がなければならない, そのためにプロセッサはメモリシステムから書き込まれたアドレスに対するキャッシュラインの更新または無効化を行わなければならない. この処理をインバリデートまたはパージ処理と呼ぶ. キャッシュメモリはメインメモリのコピーであるため, このキャッシュパージ処理を行うことができなければならない.

このキャッシュパージ処理はマルチプロセッサの際に大変重要なものである. プロセッサのメモリ要求は他のすべてのプロセッサがキャッシュメモリの更新または無効化を行わなければ整合性が無くなり, 予期した動作ができないという状況に陥るからである. また, 各プロセッサ間で同期を取るような場合, メモリシステムへ正しい順序でメモリ要求が発行されることを保証するために Memory Barriar 命令というものをマルチプロセッサ構成の可能なプロセッサでは必ず存在する.

現在のマルチプロセッサシステムにおいて, このキャッシュパージを各プロセッサに伝播する機構と順序保証を行う操作であるメモリバリアの 2 点は必ず実装されており, この 2 点についての概念であるコンシステンシモデルというものがある. このコンシステンシモデルというものが存在するため, メモリ対称型マルチプロセッサシステムは現実にそれほど多くのプロセッサを搭載できない. それはプロセッサの搭載数に比例してコンシステンシ保証プロトコルが複雑になるためである. そのため, 接続数によって SMP,NUMA, 分散メモリと様々な接続形態が存在する.

1.1.2 Load/Store バッファ

キャッシュメモリによりキャッシュ内に存在するデータに対するメモリアクセスは高速に実現可能となり, それにより演算性能を向上することができるが, キャッシュ内にデータが存在しない場合 (キャッシュミス) も必ず起こる. キャッシュミスした命令以降のメモリアクセス命令はプロセッサ外に発行されているメモリアクセス命令を知ることができない場合, 制御依存関係を検証することができず, 命令が停滞してしまう. このために現在プロセッサ外に発行されているメモリリクエストを管理し, 後続のメモリリクエストが同一アドレスに対してメモリ参照中であるかどうかを検証し, プロセッサ外へ複数のメモリリクエストの順序保証された発行を管理するものが Load/Store バッファである.Load/Store バッファによる依存関係の検証を図 1.2 に示す.

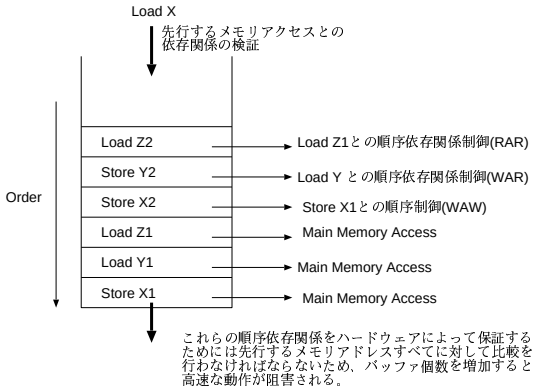


図 1.2: Load/Store バッファによる依存関係の検証

もし,Load/Store バッファが存在しない場合,Out-

of-Order のような命令追い越しを行うことはできず, また, キャッシュミスした Load 命令以降のキャッシュアクセスを行うための機構であるノンブロッキングキャッシュも実現できなくなる. これはリクエスト履歴が無いため, 依存関係を検証することができないためである.

1.1.3 Out-of-Order

Out-of-Order 技術とはプログラムの命令を順番に実行していくのではなく, 制御命令やメモリ操作命令等長大なペナルティを持つ命令が発行された際に, その命令の制御依存を順序依存関係に置き換え, レジスタをリネーミングすることでプログラムの順序を入れ替える技術である. たとえば,Load 命令がキャッシュミスした場合においてその Load 命令以降の命令で要求した値を用いない命令は Load 命令の完了を待たずとも, 演算結果が確定する. このため, メモリアクセスと演算とをオーバーラップさせることでレイテンシを隠蔽する. また, 分岐命令をフロー依存関係に置き換え, 分岐予測ペナルティを低減することも可能である. このため, 分岐後の Load 命令をあらかじめ発行することが可能であるため, 分岐が確定し, その Load が無効化されたとしても, レジスタにライトバックされることがなくなるだけで, キャッシュ内にはデータが格納され, 次にその Load 命令が実行されるときには Load 命令のレイテンシが低減される. その点においてもレイテンシを隠蔽することが可能となる.

1.1.4 Prefetch

Prefetch 命令とはその名の通りあらかじめデータの取り出しを行っておく操作のことである. このような考え方を (Memory Access) Decoupled Architecture と呼ぶ. 現在, 実現されている Prefetch 操作はキャッシュメモリに無いアドレスのデータをそれを取り出す Load 命令の数百命令前に Prefetch 命令によってキャッシュ内に格納しておくことで,Load 命令時のレイテンシを削減することものである.Prefetch 命令の格納先はキャッシュメモリとなっているため, プロセッサの命令実行パイプラインに影響を及ぼすこと無く並列な動作を行うことが可能である. Prefetch は連続したアドレスに対してのブロック転送が主であり, ベクトルコンピュータなどではバースト転送の他にストライド転送が用意されているものもある.(図 1.3)

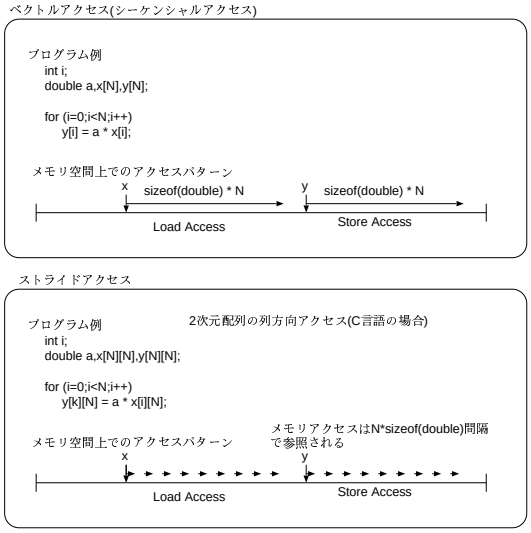


図 1.3: バースト転送とストライド転送

キャッシュを用いた Prefetch では, データの転送先がキャッシュメモリになっており, ほとんどが連続したアドレス上の大きなデータブロックを読み込む, データ転送が完了するまで, メモリバスを占有することや, 先読みされたデータによって現在利用しているデータブロックがキャッシュメモリから追い出されることもある. ただし, キャッシュプリフェッチは同一アーキテクチャ上の異なるシステム構成においてもバイナリ互換性が保証される.

1.2 スループット向上技術

スループット向上技術はプロセッサ, メモリシステム (システムコントローラ,DRAM) 間での転送性能向上のための技術であり, DRAM の技術について述べる.

1.2.1 DRAM 技術

コンピュータシステムの主記憶装置として現在主流となっている DRAM(Dynamic Random Access Memory) は SRAM とは異なり, メモリレイテンシ

が大きく DRAM モジュールのアクセス時間は 45ns ～70ns ほどである.

バースト転送

このレイテンシによるペナルティを解消するため, 粒度の大きなデータブロックのスループットを向上する技術であるバースト転送技術が用いられている. また, 複数のメモリバンクで構成することでスループットを向上するバンクインターリーブ技術を用いることもできる.DRAM はデータスループットを向上するために EDO-DRAM,SDRAM,DDR-SDRAM 等複数のものがあるが, これらはメモリセルの技術はあまり変わらず, バースト転送効率が異なる.

バースト転送とはメモリリクエスト (アドレス, コマンド, マスク) を発行後, 一定の間隔で連続した複数のアドレスのデータを転送する技術である. このとき, メモリリクエストは先頭の 1 つだけであり, それ以降のデータは内部で一定間隔のアドレス加算を行うことでデータが送られる.

図 1.4 に SDRAM によるバースト転送と 1 回ごとメモリリクエストを発行する場合のタイミングチャートを示す.

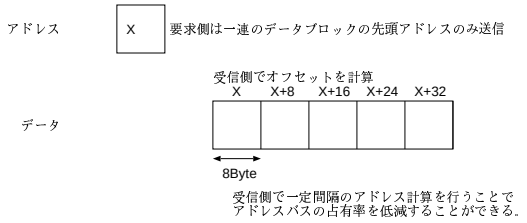


図 1.4: バースト転送によるスループットの向上

図 1.4 に示すように,2 個目以降のデータに対してのデータ送出間隔が短く, バスの利用率を高めることが可能となる. ただし, バースト転送は先頭アドレスとバースト転送長 (要求するデータの個数) が必要であり, ほとんどの場合固定長のデータ転送となる. そのため, ブロック転送時には有効であるが, ブロック内の有効なデータの割合が低くなると性能は低下する.

マルチバンク

SDRAM 以降の DRAM では 1 チップ内に複数のメモリバンクが存在する. メモリバンクが複数存在する場合, 各メモリバンクを交互にアクセスするバンクインタリーブという技術が利用できる. バンクインタリーブはリクエストからデータの送出までの遅延時間を他のバンクの転送時間で埋め合せ, スループットを高める技術である. 特に DRAM では 2 次元のアドレスデコード, アクセス後のプリチャージ等, データ転送以外の遅延が多く発生するため, 重要な技術となっている.

このように現在のコンピュータシステムにおいて, メモリに対する技術は大変重要な意味を持ち, また, 様々な技術が検討されてきた. それでもなお, プログラムによってはこれらの技術が適用困難であり, 予想以上に性能が出ないということがある.

そこで我々はコンピュータシステムの高速化, 大規模化, 複雑化の進む時代における, レイテンシ隠蔽とスループット向上技術について, 問題点の抽出と新しい技術の提案を行い, それを実現するための構造設計について述べる.

第2章　従来技術についての問題点

前章にて, コンピュータシステムにおけるメモリレイテンシ, スループットに対する現在のアプローチを述べた. ここではそれらの技術においての問題点を挙げる.

2.1　キャッシュメモリの限界

キャッシュメモリは時間的, 空間的局所性を利用した, 近接アドレスへの複数回参照に対してのレイテンシ隠蔽技術である. そのため, 空間的局所性の低いアクセス, 例えば大規模な配列への順次操作というような簡単な問題に対しても, 効果が薄れてしまう. また, 参考文献 [6] で述べられているようにキャッシュメモリ内のデータの配置をソフトウェア上から制御できないという問題がある. 複数の配列をキャッシュに載せる際の配列間の干渉 (cross-interference) の排除は上記の理由により困難である. 等, ソフトウェアによる高速化技法を困難とさせる問題がある.

キャッシュメモリのデータ配置で問題となる点はもう 1 つあり, キャッシュベース Prefetch におけるデータの格納先であるキャッシュメモリが, 自由に配置ができないこともあり, キャッシュの容量不足, 連想度不足による現在利用中のデータの追い出しとそれに伴うキャッシュミスによる性能低下という点も解決することが困難な問題である.

2.2　Load / Store バッファの限界

Load/Store バッファはキャッシュベースのシステムにおいての Load / Store 命令間での順序制御を行うハードウェアであるが, この Load/Store バッファの個数は通常 2〜16 程度である. これはエントリ数の増加は依存関係の検証要素数の増加でもあるため, 高速性が要求されるプロセッサのメモリパスを複雑化することになりうるからである.

この Load/Store バッファ個数の限界は Out-of-Order 機構の限界ともなってしまうため, レイテンシ隠蔽機構自体の限界を決めてしまう. [15]

2.3　アドレス変換機構に起因する問題点

商用のスカラプロセッサにおけるアドレス変換機構サポートはマルチタスク, マルチユーザ環境の実現において大変重要な技術である. アドレス変換は仮想アドレス, 物理アドレスの対応表をオペレーティングシステムが作成し, ソフトウェア側からはプロセスごとの独立したメモリ空間を実現している. 仮想アドレスから物理アドレスへの変換は一般的に 2 段から 3 段のアドレス変換テーブルを引くことで目的の物理アドレスが参照可能となる. つまり,1 回の Load 命令で複数回のメモリアクセスが必要となり, 大変効率が悪い. そのため, プロセッサは仮想アドレス, 物理アドレスのペアを高速なバッファに記録しておき, アドレス変換を高速化する機構を持っている. これをアドレス変換バッファと呼ぶ.

仮想アドレス, 物理アドレスともにページというサイズごとに管理される. このサイズは 4kB から数十 KB 程度であり, アドレス変換バッファの個数は 32〜128 エントリ程度となっている. アドレス変換バッファ内に登録されているアドレス空間内へのメモリアクセスはオペレーティングシステムを経由することなく, メモリアクセスが可能となる.

しかし, このページ単位での管理のために, 仮想アドレス上では連続したメモリ空間であったとしても, 物理アドレス空間上では連続していないということがある. そのため, ベクトルアクセスのような連続的なメモリアクセスに関してもページ単位という限界が発生する. これを回避するためにはあらかじめアドレス変換バッファへ全てを登録するか, 転送途中でのアドレス変換フォールトを行うことのできる機構が必要となり, メモリアクセス機構を複雑化する要因となる. 命令を先読みするための Prefetch ではこの問題が影響するため,1 命令で転送できるブロックのアドレスはページ境界内となるという制約がある. キャッシュベースの Prefetch は Load バッファ数以上の転送要求が発行できないことと 1 回の要求サイズがページサイズ以下という制約からスループット

があまりでない.

このような様々な問題が現在のコンピュータシステムでは存在しており, 科学技術計算分野においては [35] で述べられているようなキャッシュブロッキング, ループ展開, 入換え, ソフトウェアパイプライニング等メモリアクセス最適化手法が提案され, これに適したデータ再構築手法として [17] のように Data Restructuring 等も研究されている.

ポインタアプリケーションにおけるソフトウェア最適化については [19] のような最適化が行われており, ポインタアプリケーションに対する Prefetch の効果を評価を行っている.

これらのソフトウェア手法はあくまで上記の問題点に対するソフトウェアレベルでの対応策であり, ハードウェアによるサポートがあればより効果的に利用できると考えられる. 次章ではこれらの問題点の提示からハードウェアの複雑化を防ぎつつ, これらの問題点を解決するためのメモリアーキテクチャ SCALT について述べ, 原理を説明する.

第3章　メモリレイテンシ耐性アーキテクチャ (SCALT)

コンピュータシステムの理論性能と実効性能の差は性能のボトルネックとなるメモリ性能,I/O 性能が演算性能ほどの性能向上をなされていないことが原因である. メモリ性能はレイテンシ, スループットの 2 点で性能が決定される. 前章で述べた従来までの技術はメモリレイテンシ隠蔽の面で相対的にレイテンシが増大する現在, このメモリレイテンシを隠蔽しきることが困難である. これからの時代, いままで以上に相対的なメモリレイテンシが増加することが容易に想像出来る. そのためにはレイテンシ隠蔽技術にも利用できる資源に対してスケーラビリティを持たせたものが必要である. また, 大規模なコンピュータシステムの長大なレイテンシや NUMA システムのようにレイテンシが変動するようなシステムにおいても, レイテンシの隠蔽, 高いスループットの実現でき, リソースに対して性能がスケーラブルなアーキテクチャが必要であり, そして, 従来の技術と組み合わせた場合においても互いの技術による性能向上を阻害することの無いものである. たとえば, プロセスの微細化, 長いパイプライン化による高速化の阻害となるものやキャッシュメモリなどの従来より用いられているレイテンシ隠蔽技術との親和性が高いものでなければならない.

そこで我々はアドレス空間に SCALT バッファという高速 SRAM を配置し, このバッファ領域とメインメモリ間の転送に演算実行パスとは独立した動作を行う転送命令を持つアーキテクチャ(SCALT) を提案している. 以下に SCALT アーキテクチャの原理, 構成を述べる.

3.1　アーキテクチャ検討

現在の複雑化したコンピュータシステムの中で, ハードウェアアーキテクチャという概念だけでは問題を解決することが困難である. コンピュータシステムという考え方はハードウェア, ソフトウェア双方の技術を協力することが重要であり, なおかつ, オペレーティングシステムの稼働するようなシステムでも効果が現れるようなものであるべきである. また, メモリの高スループット化を実現する技術はほとんどが連続的なアクセスにおいて効果を表し, ランダムなアクセスに対してはあまり効果が期待できないものである.

ソフトウェアが大規模, 複雑複雑化されている現在, プログラムは様々なデータ構造を持ちハードウェアで解決するにはアクセスパターンが複雑である. このメモリアクセスパターンをソフトウェアにより解決できないかと考え, それを実現するためにソフトウェアから管理可能な高速 SRAM 領域, シンプルなハードウェア機構とメインメモリからのデータ転送性能を十分に引き出すプロセッサーメモリ間のインターフェースにより, 大規模なメモリ利用, ランダムなメモリアクセスにおいても高い性能を出すことができると考えられる.

従来の技術は基本的にデータ参照の局所性つまり,” 過去に利用したアドレスに対する高速なデータ参照, 連続的なアクセスの性能向上” という点で効果的なアクセスが実現できている. しかし, それぞれの技術が密接な関係に有り, どれか 1 つがボトルネックとなった場合, または技術が有効に機能しない場合にそれがレイテンシの隠蔽できる限界を決めてしまうことが問題であった.

特にメモリアクセスに関してハードウェアによるアドレスの前後関係の検証を行う Load/Store バッファは前述したように複雑であり, それほど大きくできない, 検証できるアドレス個数が減るということは,Out-of-Order による投機的なメモリアクセス数も Load/Store バッファ個数以上のメモリレイテンシを隠蔽できないことを意味する.

もし, データの局所性のないアクセスが多発するようなプログラムを実行した場合, システムはメインメモリのスループットに支配される. この場合, 実効スループットは式 3.1 で表すことができ, これを用いてレイテンシ隠蔽機構の有効性を示す不等式 3.2

を満たすことができればレイテンシを隠蔽できることになる.

$$Throuput = LoadBufferEntry * CacheLineSize \tag{3.1}$$

$$Latency \leq Throuput + (OoO * freq) \tag{3.2}$$

式 3.2 において,Latency はコンピュータシステムのメモリレイテンシ,OoO は Out-of-Order による投機的な命令実行数 freq はプロセッサの動作周波数である.

参照性の無いデータに対するレイテンシ隠蔽機構である Prefetch についても同様で, 大きなブロック転送を行った際にも連続的でないメモリアクセスについてはデータブロック中の有効なデータが少くなるため, オーバーヘッドが大きくなる.

3.2 SCALT アーキテクチャ

SCALT アーキテクチャはプロセッサアーキテクチャの 1 部ではあるが, 命令セットアーキテクチャ, キャッシュアーキテクチャに依存せず, 従来までの技術と共存できるような構成である. 図 3.1 に SCALT アーキテクチャにおけるプロセッサ, メモリ間の構成を示す.

SCALT アーキテクチャにおいて従来までのシステムモデルとの間で構造的に異なる点は SCALT バッファというアドレッシング可能な SRAM 領域とプロセッサーメモリシステム間のインターフェースに'tag'というリクエスト識別情報が追加された点である. まず,SCALT バッファと tag について説明する.

3.2.1 SCALT バッファ領域と tag

SCALT バッファ領域はキャッシュメモリと同様にデータ記憶素子に SRAM を用いており, キャッシュメモリと同階層に位置する記憶領域である. プロセッサーバッファ間はキャッシュメモリのシステムと同様に Load/Store 命令によってアクセスされるが, バッファーメモリ間のデータ転送についてはキャッシュメモリとは異なり明示的な転送命令で処理が行われる.

このような観点から考えると,“Load/Store によってアクセスされる大規模なレジスタ” とも考えられる. それならばレジスタを大きくすることで解決できるのかというとそうではない. レジスタは演算パスの視点と終点に位置し, 高速性が要求される. また, レジスタである限り, 演算パスとメモリアクセスパスを分離することが不可能である. その点でレジスタとは大きく異なる.

図 3.2,3.3 に SCALT バッファの構成とメモリ空間モデルを示す.

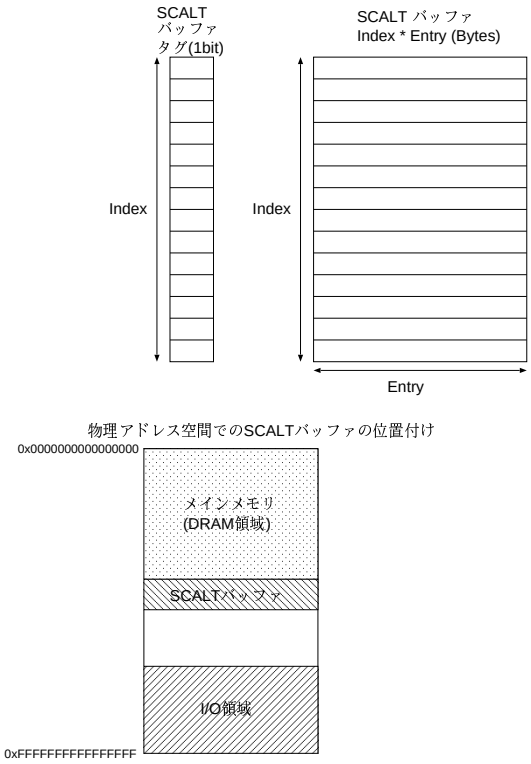


図 3.2: SCALT バッファの構成

図 3.2 に示すように SCALT バッファは 1bit の Valid(有効) ビットのみを保持するバッファタグとブロック単位 (以下エントリと呼ぶ) でデータを格納するデータ記憶域で構成される. バッファはキャッシュのようにアドレスやフラグなどハードウェアによっての管理を必要としないため, 物理的な構成が簡素化される.

3.3 に SCALT におけるメモリモデルにおいての SCALT バッファの位置付けが示してあるが, SCALT バッファは物理メモリ上にマッピングされ, メインメモリのようにオペレーティングシステム下で管理さ

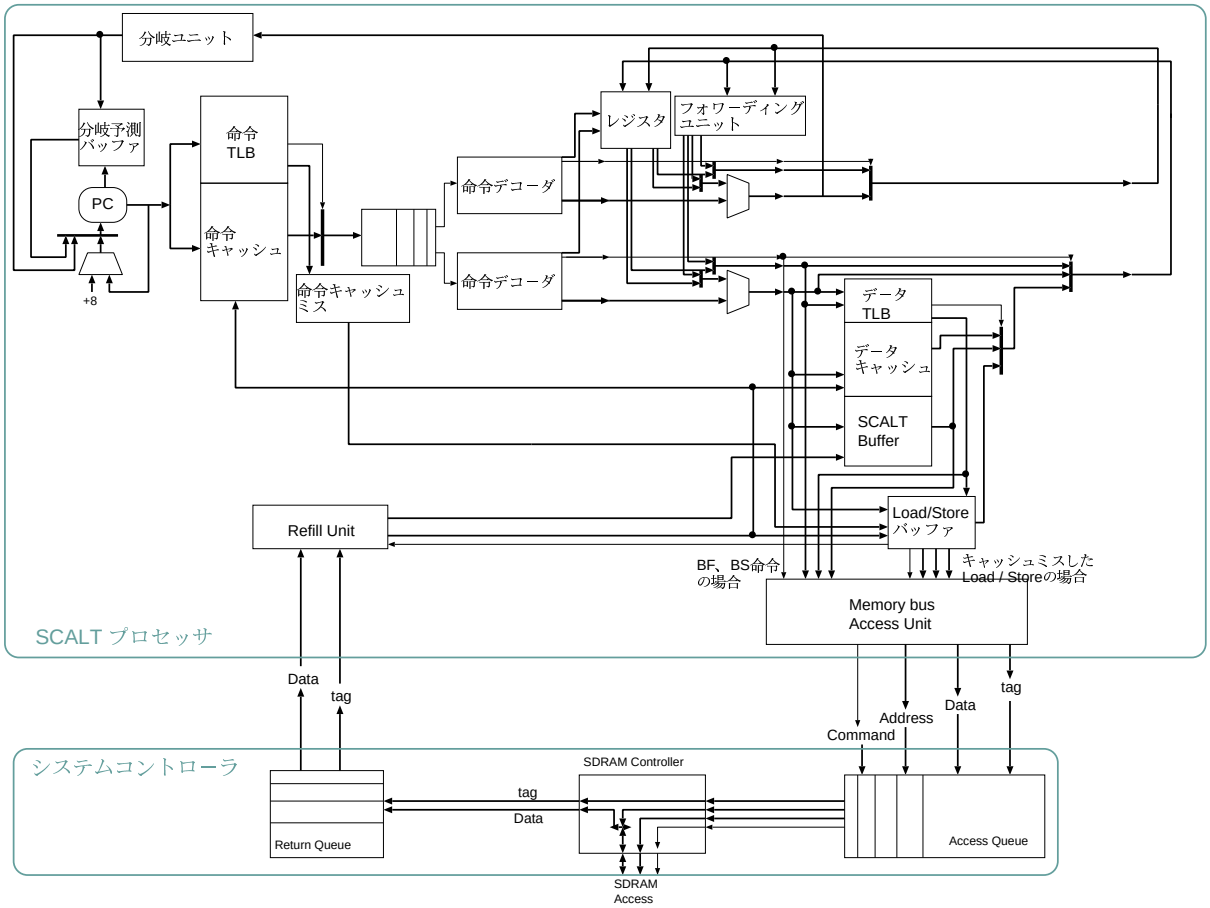


図 3.1: SCALT アーキテクチャにおけるシステムモデル

れる. つまり, プログラムは SCALT バッファを利用する際, メインメモリのようにオペレーティングシステムに malloc システムコールのような要求を出し, 仮想メモリ空間にマッピングされた SCALT バッファへのアドレスを得ることで初めて SCALT バッファを利用することができるようになる. このため, ある SCALT バッファ領域は 1 つのプログラムによってのみアクセス可能である. つまり, キャッシュのように他のプログラムから書き換えられることの無いようにオペレーティングシステムによるメモリ保護を受ける. このように仮想アドレス上にマッピングされたバッファ中のデータ領域に関してはプログラマが自由にデータレイアウトを構築することが可能となる. プロセッサー SCALT バッファ間では Load/Store 命令で SCALT バッファのマッピングされている仮想アドレスを指定してアクセスされる. そのためアドレス変換バッファ(TLB) で SCALT バッファへアクセスするのか, キャッシュメモリへアクセスするのかを識別するための制御ビットを持つ必要があり,TLB を通常の TLB と別に持つか, 統一した形を取るかは実装者が自由に決定することができる.

そして, バッファーメモリ間の転送は SCALT で規定される転送命令を用いてエントリと呼ばれるブロック単位転送を行う. 専用命令については次節で詳細を述べる. 転送命令が発行されるとき, プロセッサからはメモリアドレス. コマンド, データとともに,'tag' と呼ばれる識別符号をシステムコントローラへ発行する. この'tag' のフォーマットは図 3.4 のような構成を取る. 図 3.4 のように'tag' は $\log_2$(バッファのエントリ数) + 1 ビット幅で実現する.

タグの内容は最上位ビットがキャッシュ/バッファの選択ビットであり, それ以下のビットが SCALT バッファのエントリ番号となっている. この'tag' はプロセッサからメモリリクエストが発行されたとき, その

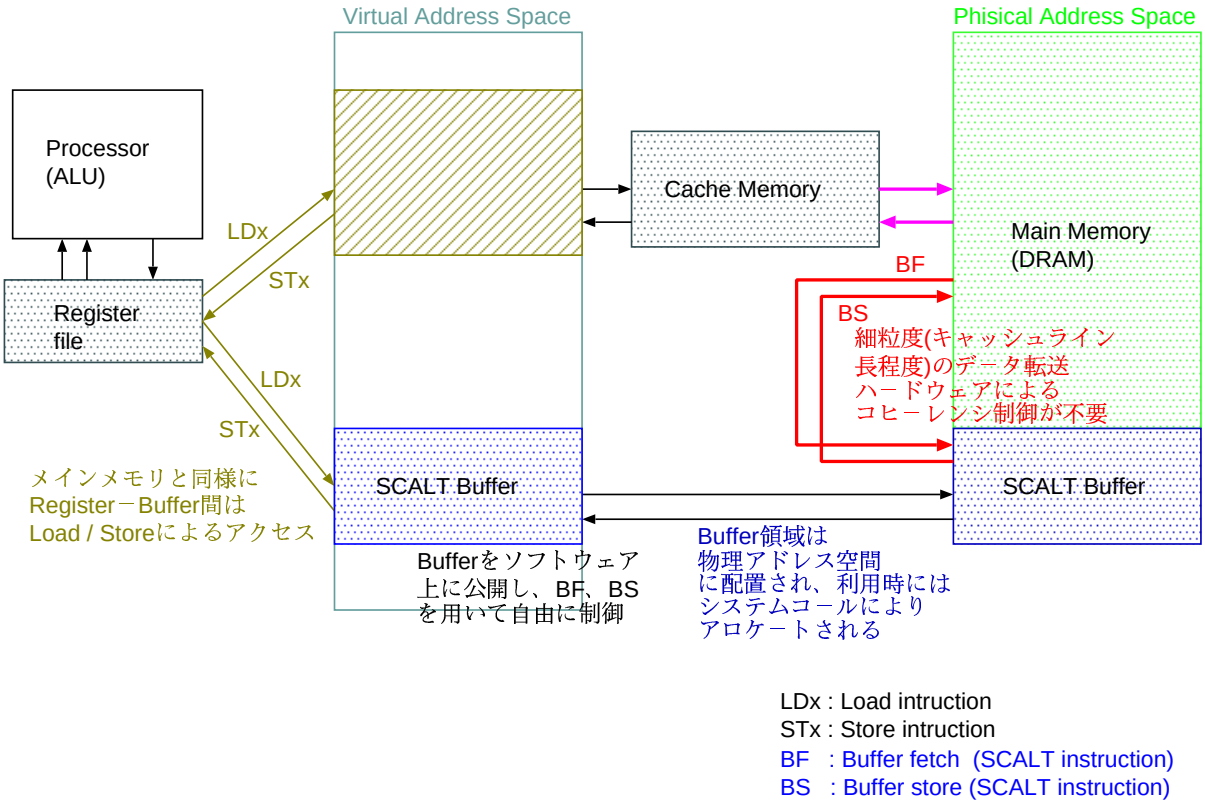


図 3.3: SCALT アーキテクチャのメモリモデル

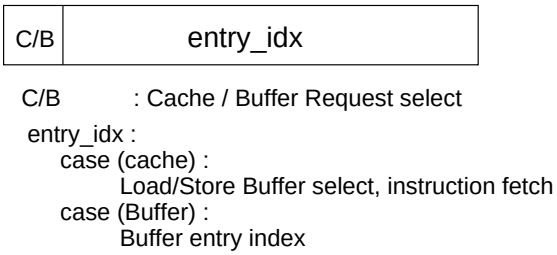


図 3.4: SCALT タグフォーマット

バッファエントリの有効ビットを無効化してシステムコントローラへメモリリクエストを発行する. システムコントローラはアドレス, コマンド, データにより, 要求されたリクエストを処理し, 読み出しリクエストの際にはメモリから取り出されたデータと共に'tag' をプロセッサに送る. プロセッサはデータの到着時にシステムコントローラから'tag' を受け取り, データの格納先を識別する. そして到着したデータを SCALT バッファに書き込み, そのエントリの有効ビットをセットする.

SCALT アーキテクチャにおいて'tag' は重要な存在であり, この中にデータの格納先が入っている. そのため, プロセッサ内部に情報を保持する必要が無く, バッファエントリの指定 (つまり, アドレスの管理) はソフトウェアで行うことから Load/Store バッファの必要が無くなり, バッファーメモリ間転送命令では命令のハードウェアによる依存関係のチェック無しでシステムコントローラへリクエストを大量に発行できる. リクエストの最大発行数は式 3.3 で表すことができる.

$$MaxReq = Min(Entry, Queue) \tag{3.3}$$

式 3.3 において,MaxReq は SCALT アーキテクチャを適用した場合の最大リクエスト数, Entry は SCALT バッファのエントリ数, Queue はメモリシステムのキューの本数である.

従来の方式では Load バッファ個数までのリクエストしか発行できず, 最大個数は 2〜数十個程度である. これは Load バッファがアドレスでリクエストを

管理しているためであり, 先行するメモリリクエストとの関係を全てチェックしなければならないため, あまりこの個数を増加してしまうと大きな遅延が発生し, また回路構成が複雑になってしまう. しかし,1k エントリのバッファを持つ SCALT システムでは最大 1024 リクエストまでプロセッサ外へリクエストを発行可能であり, バッファ容量を増加するだけでそれ以上にすることも容易である.

また,'tag' の存在により, プロセッサによってシステムコントローラへ要求したデータは'tag' (格納先) がデータと共にプロセッサに送られる. そのため, システムコントローラはプロセッサのリクエスト発行順序と異なる順序でデータを送ったとしてもプロセッサは'tag' を見るだけで一意に格納先が識別できるため, プロセッサは複雑な機構を持たずにリクエスト順序とデータの到着順序が異なる場合でも問題無く処理することができる. これはシステムコントローラとしては大変重要なことで, 現在のコンピュータシステムではメモリデバイスは複数バンク構成を取ることが多く, リクエストの順序を入れ換えることによってスループットを高めることが可能となる. これについての評価は次章で述べる.

SCALT アーキテクチャでは固有の命令として SCALT バッファーメモリ間転送命令である BufferFetch(BF), BufferStore(BS) とデータ到着検証命令である Buffer Check(BC) 命令の 3 命令を基盤となるプロセッサに追加する. SCALT アーキテクチャ固有の命令である 3 命令 (BF,BS,BC) について説明する.

3.2.2 SCALT 転送命令

SCALT ではアドレッシング可能な SRAM 領域とメインメモリ間のデータ転送命令が 2 つと先ほど述べたバッファエントリの有効ビットの確認を行う命令で構成されている. この 3 命令について説明する.

Buffer Fetch(BF) 命令

Buffer Fetch 命令は図 3.5 のような形で命令が指定される.

BF 命令は命令で指定されたバッファエントリへメモリからエントリ (1 ブロック) 単位でデータを読み込む命令である. BF 命令の動作を図 3.6 に示す.

図 3.6 のように,BF 命令は必ずプロセッサ外へメモリリクエストを発行する. この際の転送粒度は SCALT バッファの 1 エントリとなる. このエントリ長はキャッシュメモリのライン長と同様に命令セットで規定されている転送粒度より大きいブロックである. 命令で用いるデータパスはキャッシュミスした場合の Load 命令と同じであるため, 従来のデータパスがそのまま利用でき, 特殊な実装をせずに SCALT バッファ,TLB の制御ビットの追加だけで実現可能である. BF 命令のキャッシュベースの Load 命令と異なる点はレジスタへのライトバックが行われないこと,SRAM 領域を明示的に指定することと Load バッファに登録されることはなく, 前後の Load/Store 命令との依存関係をハードウェアによって検証されない. の 3 点が大きく異なる. レジスタへのライトバックが行われないということは, プロセッサの演算パスからメモリパスを切り離すことができるということであり, 命令依存関係の検査パスが緩和される. そして,Load バッファへの登録がされないということはハードウェアによる依存関係のチェックが行われないことであり, これは格納先が SCALT バッファであるからである.SCALT バッファと演算パスの間にはもう 1 つ Load 命令による転送が存在するが, レジスターバッファ間のレイテンシは 1 サイクルであり, 非常に小さい. また,SCALT バッファの各エントリはソフトウェアによる明示的な指定を受けているため, 各エントリに対する操作はソフトウェア側で命令順序が保証されており, ハードウェアはその検証操作から開放される.

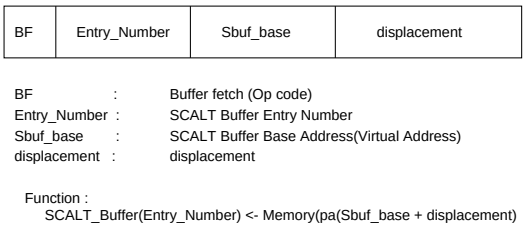


図 3.5: Buffer Fetch(BF) 命令

メモリの要求と使用を分離した形で実現される Decoupled な命令の中で重要な問題となる高速なデータ格納先を SCALT バッファとすることで予期しないデータの追い出しが発生するキャッシュベースプリフェッチのようなことを意識せずにすむ.

前述した 'tag' はバッファアクセスビットとエント

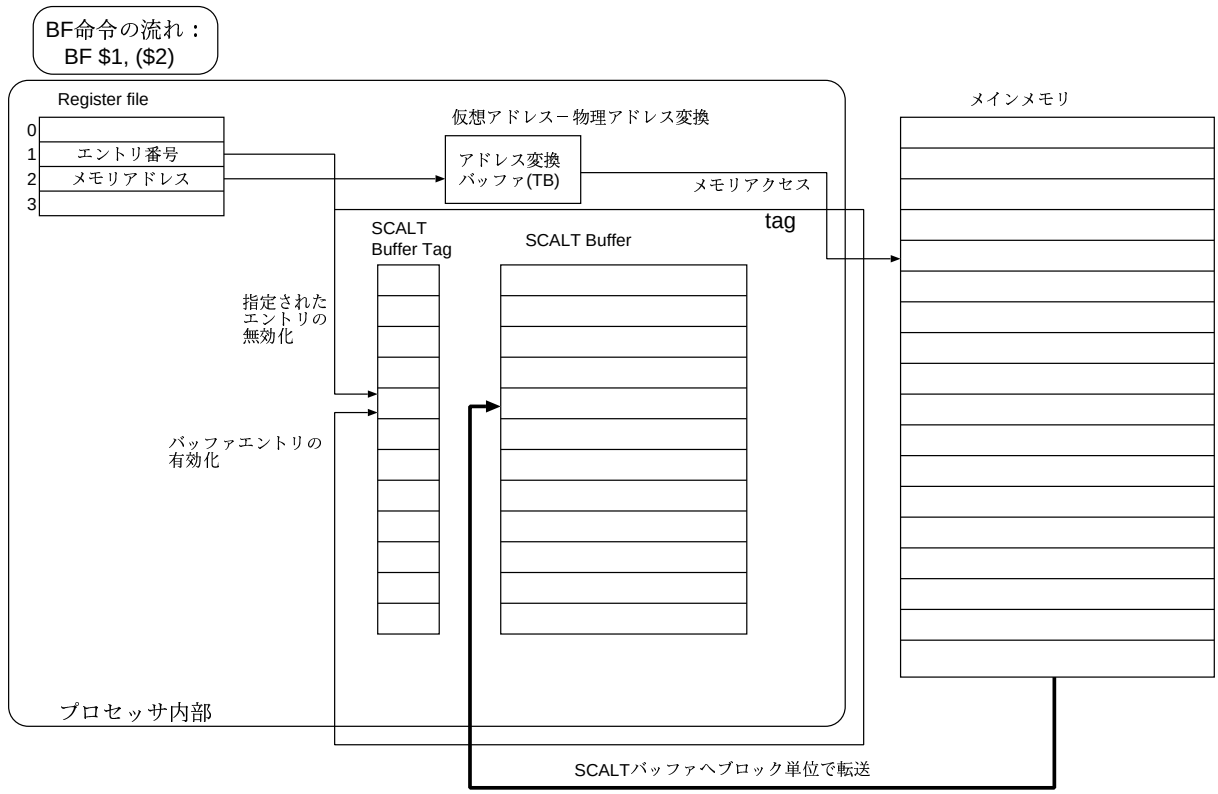


図 3.6: BF 命令の動作

リ番号により構成され,‘tag’ の存在によりメモリアクセス時間の異なるシステム内において, プロセッサ外でデータ到着順序が変更されたとしてもプロセッサは ‘tag’ 内で指定されたエントリに対してデータを格納するだけですむ. 格納される際にそのエントリに対して有効ビットをセットすることでそのエントリに対する Load 命令が可能となる.

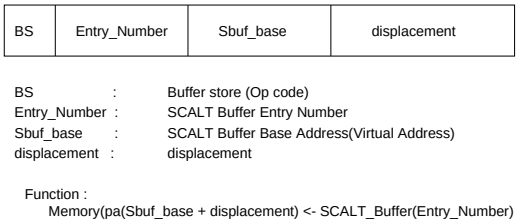


図 3.7: Buffer Store(BS) 命令

Buffer Store(BS) 命令

Buffer Store 命令は図 3.7 のような形で命令が指定される.

BS 命令は命令で指定されたバッファエントリからメモリへエントリ (1 ブロック) 単位でデータを書き出す命令である. BS 命令の動作を図 3.8 に示す.

BF 命令においても,BF 命令と同様にキャッシュメモリアクセス時のデータパスをそのまま利用可能である. そして, 前後の Load/Store 命令との依存関係を検証することなく処理される.BS 命令の依存関係はあくまで他の BF,BS 命令との間でのみ検証され, これはプログラムの命令列で既に検証が完了しているのでハードウェアで特に検証を行う必要はない.

BF 命令と異なる点はメモリへの格納操作であるため, プロセッサはシステムコントローラへリクエストを発行した段階で,BS 命令の操作は完了する. そのため,BS 命令時の’tag’ はプロセッサへ返却されることは無い. また,’tag’ と同様に SCALT バッファのエントリ有効ビットも特に操作する必要は無い. あえて必要だと考えられる時間はアドレス変換されてからシステムコントローラがリクエストを受け付けたことが確認されるまでの時間である.

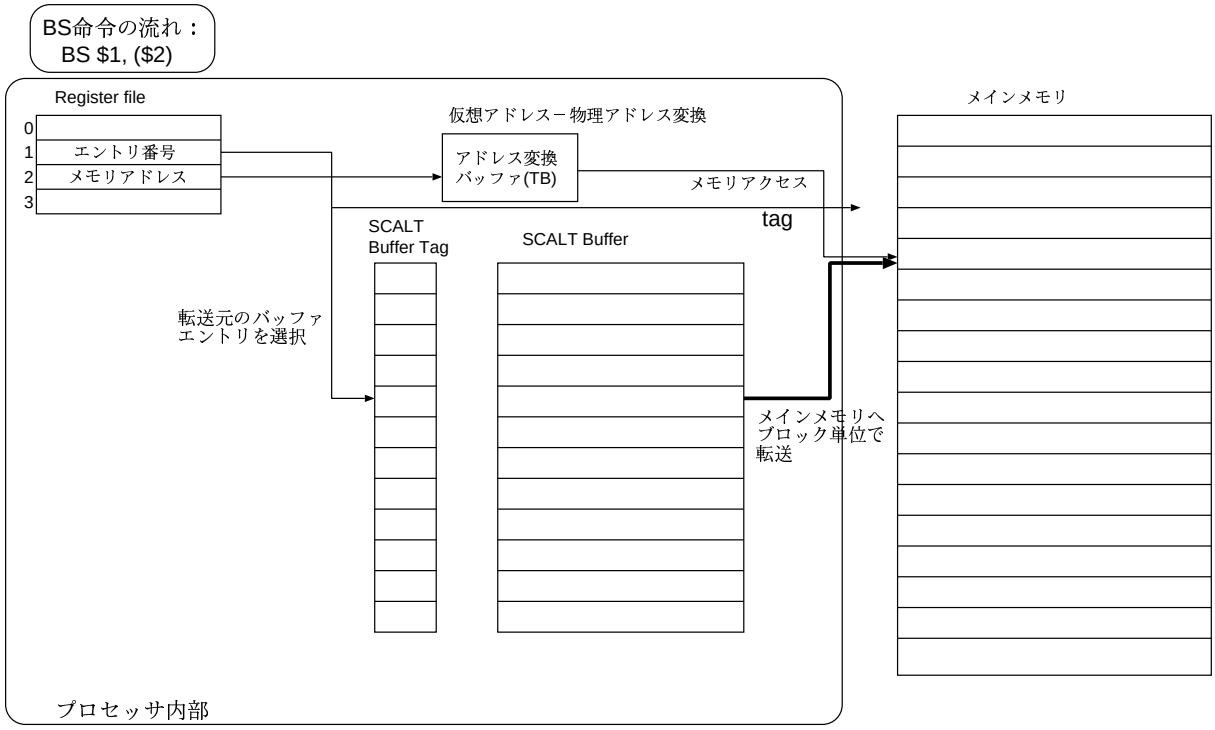


図 3.8: BS 命令の動作

Buffer Check(BC) 命令

Buffer Check 命令は図 3.9 のような形で命令が指定される.

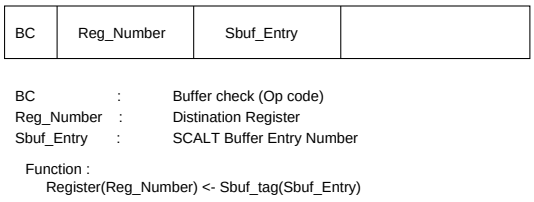


図 3.9: Buffer Check(BC) 命令

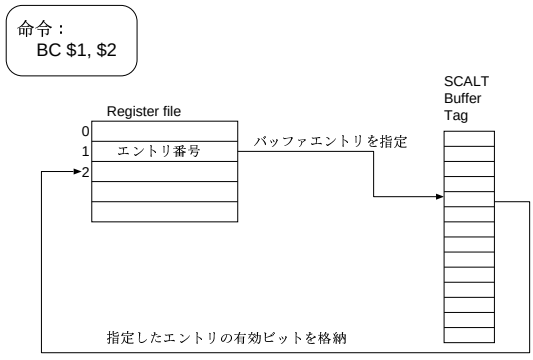


図 3.10: BC 命令の動作

BS 命令は命令で指定されたバッファエントリの有効ビットをディスティネーションで指定されるレジスタへ格納する命令である. この命令を用いることで BF 命令で要求された命令が到着しているかどうかを確認する. BS 命令の動作を図 3.10 に示す.

メモリアクセスがアドレスによって変化する NUMA のようなシステムにおいて, この命令を用いることで, バッファに対する Load 命令のストールを回避することができる. またシステムのレイテンシを把握する際のパフォーマンス指標のために用いることで, システムに特化したチューニングを可能とする.

BC 命令は SCALT アーキテクチャの中でレイテンシの隠蔽とは直接関係しない命令ではあるが, この命令を利用することによってレイテンシの把握が可能となり,BF 命令の発行数を決定する際の指標を求めることができるという点で重要な意味を持つ.

SCALT アーキテクチャとして規定されているも

のはこれまでに述べたものだけであり, ハードウェア構成としては非常に簡潔にまとめられている. 特に他のハードウェア技術との依存が少ない点は重要であり, これは複雑化するハードウェアの中で依存の発生するものが増加すると, 性能のボトルネックとなる部分の切り離しが非常に困難となるからである.

これらのことより,SCALT における性能のスケーラビリティは SCALT バッファの容量のみによって変化するといえる.

第4章　適用領域の検討と関連研究

ここでは本研究で用いているSCALTアーキテクチャと関連する研究のアプローチと現在の高速化手法において比較的困難である問題への対応について述べる.

4.1　適用領域

これまでに述べたSCALTアーキテクチャが実際にどのようなプログラムにおいて有効であるのかを検討する. これはキャッシュベースのレイテンシ隠蔽機構の不得意な問題である, 大規模数値計算における大きな配列に対するアクセスやデータベースやオペレーティングシステムにおけるポインタ指向プログラムが主にあげられる.

4.1.1　数値計算分野におけるSCALTの利用

大規模数値計算分野ではプログラム中で利用するメモリ量が大きく, 大きな配列に対してのベクトル演算の場合, 再利用性の無いメモリアクセスが大量に発行される. この場合, メモリアクセスが時間的にも空間的にも局所性が無く, これまでに述べたようなメモリレイテンシ隠蔽機構でのハードウェアによる解決が困難となる.

そのため, 数値計算時のメモリ性能はメインメモリまでのレイテンシと, スループットが重要となる. 数値計算におけるメモリ性能を測定するためのベンチマークとして,STREAMベンチマーク [29] というものがあり, これは大きな配列に対する4つの演算カーネルを用いてシステムのメモリ性能を測定するプログラムである. STREAMベンチマークの各カーネルについての説明を表4.1に示す.

これらの4つのカーネルは線形代数で用いられる基本的な操作を行っており, 計算量自体はあまり多くない. そのため, システムのメモリ性能がそのまま反映される. また4つのカーネルはループ回数Nが1000000となっており, また再利用性が低いため, キャッシュメモリは事実上意味を為さない.

しかし, これらの演算カーネルは基本的に配列の連続アクセスを行っているこのため,Prefetchによる先行アクセスはアドレス計算が容易であることもあり, 有効であると考えられる. SCALTにおいてはBuffer Fetch命令がこれに相当し, 効果的なメモリアクセス実現できると考えられる.

4.1.2　ポインタ指向プログラムにおけるSCALTの利用

ポインタ指向アプリケーションでは基本的に値と次の要素を指定するアドレスをセットでもち, これを繋げることで図4.1のようにデータ構造を構築している. このセットの接続構成により Linked-List, 木等の構造となる.

ポインタ指向プログラムはメモリ性能があまりよくないといわれている. これは各セットはメモリ中のどこに存在してもよいという自由度の高さのため, 各要素がメモリアドレス上で連続していない場合が多いため, リストの操作はメモリアクセスにおけるランダムアクセスとなることと, 各要素へのアクセスはその1つ前の要素を取得できず, 次の要素へのアクセスができないため, メモリアクセス命令が発行できないという2点においてである.

そのため, 通常, データ構造としてはリストより, 木構造が多く用いられている. 図4.2にB-Treeと呼ばれる2分木の構造を示す.

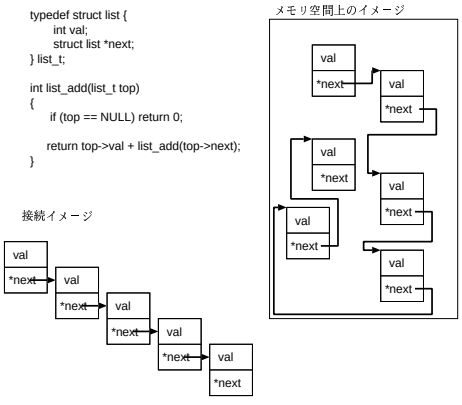


図 4.1: リスト構造

カーネル	演算	Load(Byte)	Store(Byte)
Copy	b[i] = a[i]	8	8
Scale	b[i] = s * a[i]	16	8
Add	c[i] = a[i] + b[i]	16	8
Triad	c[i] = s * a [i] + b[i]	24	8

表 4.1: STREAM ベンチマークカーネル

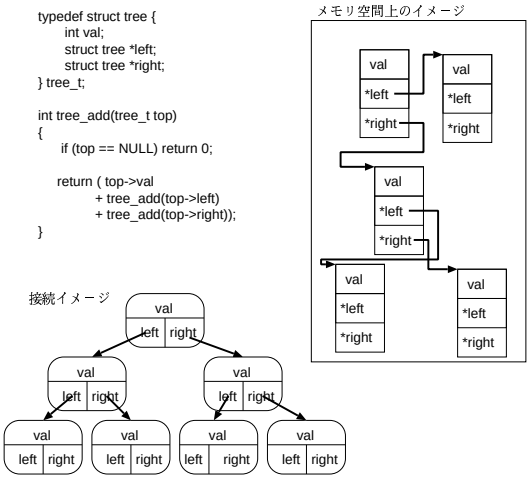


図 4.2: 木構造 (B-木)

ポインタ指向プログラムのベンチマークプログラムとして,Olden ベンチマーク [32] があり, ベンチマークカーネルとして TreeAdd と呼ばれるプログラムがある. このベンチマークは 2 分木の全ての要素の総和を求めるベンチマークであり, ポインタアクセス, ランダムアクセス性能についてのシステムの特性を得ることができる. 二分木の特徴として階層が深くなるほど, 次の要素へアクセス可能な要素が多くなる. キャッシュベースのプログラムの場合,B-Tree の総和を求めるとき, アクセス可能な要素, つまりアドレスがわかっている要素に対するアクセス数が Load / Store バッファ数以上にはできない. しかし,SCALT では Buffer Fetch をアクセス可能な枝全てに発行可能であるため, 幅優先アクセスを行えばこのようなプログラムに関してかなり高速化を期待できる.

4.1.3 シミュレータによる評価

これまでに述べたプログラムに対して CPU, メモリシステムを合わせたサイクルレベルシステムシミュレータによって SCALT がどのていど有効であるかを調べるために性能検証を行った. システムシミュレータは表 4.2 のパラメータを用いた.

STERAM ベンチマーク

STEREAM のそれぞれのカーネルにおいて性能をレイテンシを 40 サイクル,50 サイクルと変化させ, 各カーネルごとにキャッシュメモリを利用した場合と BF,BS 命令を利用した場合のそれぞれの性能を比較した. その結果を図 4.3,4.4 に示す.

図 4.3,4.4 からわかるように,BF,BS 命令による Prefetch の効果が大きく, 大幅に実行時間が減少できている. カーネルごとの比較を行うと, やはり演算量が少ないほど性能差が大きくなるということがわかる. これは演算量が多くなるほど命令中の Load/Store 命令比率が小さくなるためであり, データ転送と演算を同時に行うことでレイテンシを隠蔽する Out-of-Order が効果的に機能しているということがわかる. ただし,Triad カーネルの結果を見てもわかるようにキャッシュメモリ+Out-of-Order でも隠蔽できるレイテンシには限界があり, これは Load/Store バッファにより限界が発生している. BF,BS 命令は Load/Store バッファよりも多くのメモリアクセスが発行できるため, メモリアクセスの点で Out-of-Order の限界を拡張できると考えられる. つまり,SCALT の効果は他のメモリレイテンシ隠蔽技術を生かす技術でもあるとも考えられる.

パラメータ名	値
プロセッサパイプライン	5 段
整数命令	1 Cycle
浮動小数点演算	3 Cycle
メモリアクセス / Cycle	1 リクエスト
システムレイテンシ	40, 50 Cycle
メモリバンク数	4
キャッシュライン長	32 Byte
キャッシュメモリ	32kByte
SCALT バッファ	32kByte
Load / Store バッファ	2/2

表 4.2: シミュレーションパラメータ

TreeAdd ベンチマーク

ポインタ指向プログラムに対して SCALT がどの程度効果が期待できるのかを検証するため,TreeAdd ベンチマークカーネルを用いてシミュレータによる検証を行った. 問題となる 2 分木を Level 5(31 要素) 〜 Level 8(257 要素) まで変化させ, 測定した. TreeAdd ベンチマークの測定結果を図 4.5 に示す.

図 4.5 からわかるように, キャッシュベースのシステムの場合, 要素数が増加するにしたがって実行時間が指数関数的に増加する. これは Load 命令が発行可能要素数に対して,Load バッファ個数が不足しているために性能が低下していると考えられる. 次に,SCALT を幅方向優先でアクセスした場合の結果については要素数に比例した時間となっている. これはアクセス可能な要素数が増加しても Buffer Fetch 命令が有効に機能しているからだと考えらえる. SCALT バッファエントリ以上の Level の場合は SCALT バッファエントリ不足による性能低下が考えられるが, バッファエントリ数を増加することが容易であるため, そのような場合においても十分に対応可能であるといえる.

4.2 関連研究

本研究と同様に現在のレイテンシ隠蔽技術では効果の薄い科学技術計算における性能向上を目指した研究は様々な組織で研究されている. ここでは, 本研究で提案している SCALT と同様にアドレッシング可能な SRAM 領域を用いたレイテンシ隠蔽機構を研究している東京大学先端科学技術センター 中村 宏助教授のグループによるプロジェクトである SCIMA(Software Controlled Integrated Memory Architecture) を挙げ, この研究の特徴について説明する.

4.2.1 SCIMA

SCIMA は本研究で提案している SCALT とかなりの共通点がある. ただし, 適応領域が多少異なるため, メモリアクセス部については大きく異なる. まずはじめに SCIMA の概要を説明する.

4.2.2 概要

SCIMA はキャッシュメモリの明示的な操作ができないことを問題点の主眼として挙げており, 大域的なメモリアクセスを時間的に局所的になるようなプログラミング技法であるキャッシュブロッキング [11], [12] を効果的に利用するために SRAM 領域をソフトウェア操作可能にしている. SCIMA の命令は離散したメモリ領域を On-ChipSRAM 領域上で連続的に扱い, 演算器への高いスループットを実現することを目的としており, この離散領域はブロック－ストライド転送のみで補えるようなものに限られる.

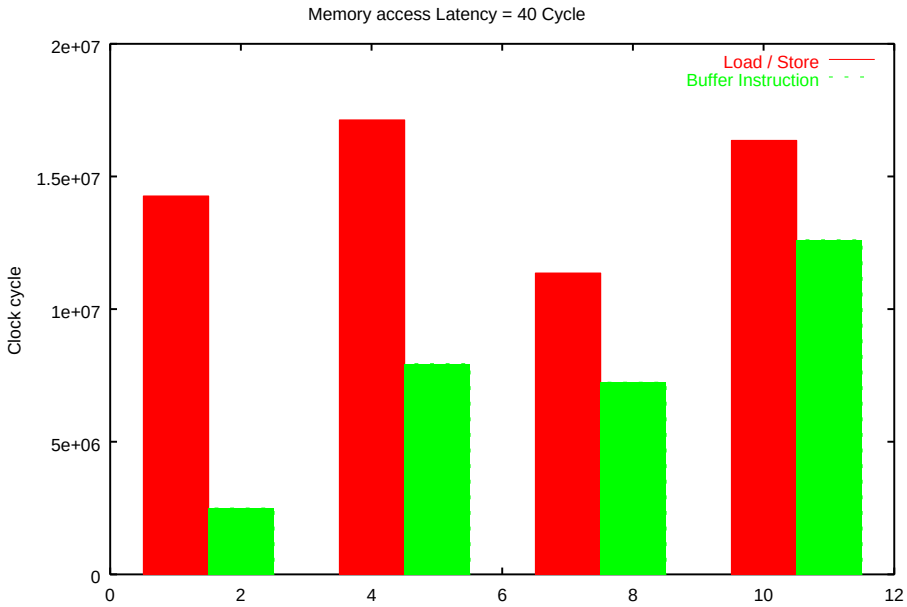


図 4.3: STREAM ベンチマーク (Latency 40 Cycle)

On-Chip SRAM 領域

On-ChipSRAM 領域はプロセッサにオンチップに配置され, アドレス空間上にマッピングされる点で SCALT と同様である. ただし, この On-Chip SRAM を管理するための専用ハードウェア機構があり, 特殊な構成となっている. この SRAM 領域に送られるデータは Uncacheable 属性として扱い, このアドレスに対するコンシステンシを無効化している.

On-Chip SRAM 領域は専用ハードウェアで制御されており, 次のようなものがある.

- Way Lock register(WLR)
  SCIMA はこの WLR レジスタによってキャッシュメモリ動作をするか,On-Chip SRAM 領域として利用するかを変更することで適用領域を変更可能としている. この WLR はキャッシュメモリの Way 数分用意され, ビットがセットされた Way は On-Chip SRAM 領域として Lock される.
- On-Chip address start register(ASR)
  オンチップメモリとしてメモリ空間にマッピングされた領域の先頭アドレスを保持している. ASR のメモリアラインメントは On-Chip メモリのサイズにアラインメントされる.
- On-Chip address mask register(AMR)
  オンチップメモリ領域のサイズを示すためのマスクレジスタであり,Way サイズ (キャッシュ総容量/連想度) のマスクである.

On-chip SRAM 領域へのデータ転送は page-load,page-store,load-multiple,store-multiple という命令を用いて行う. これらの命令の操作について説明する.

page-load, page-store

メインメモリとオンチップメモリ間でデータ転送を行う命令である. この転送は page という単位での大きな粒度の転送を行う. この命令操作は DMA 転送によって実現される. この命令では 4 つのパラメータを指定して転送される.

- source address
  データ転送を行うソース側の先頭番地であり,page-load の場合はメインメモリ,page-store の場合は On-Chip SRAM 領域を示す.
- distination address
  データ転送を行う対象側の先頭番地であり,page-load の場合は On-ChipSRAM 領域,page-store の場合はメインメモリとなっている.

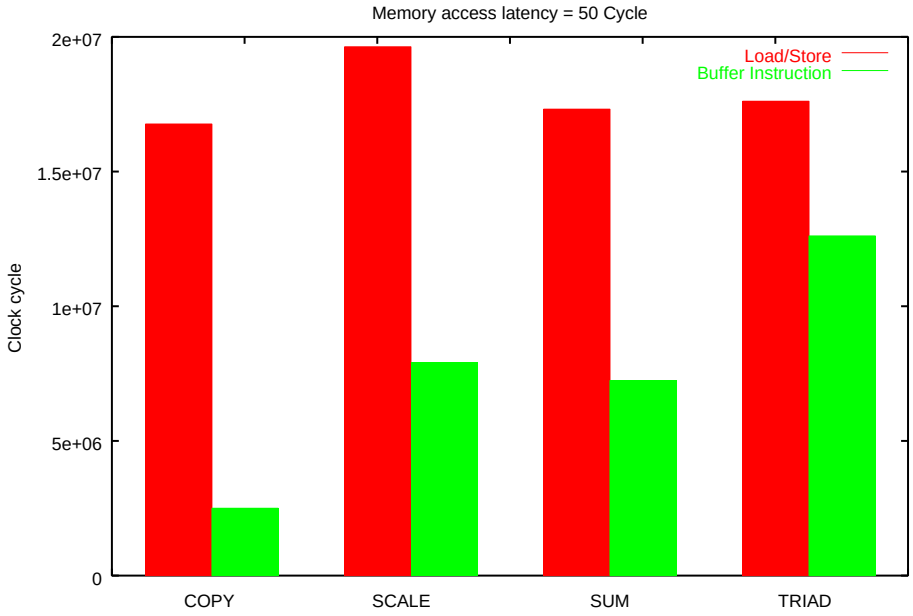


図 4.4: STREAM ベンチマーク (Latency 50 Cycle)

- size
  転送サイズ
- block, stride
  ブロック幅, ストライド幅. 転送サイズは page 内で可変となっており, ページサイズは 128kB と現在のシステムよりかなり大きなサイズとなっている.

load-multiple, store-multiple

オンチップメモリとレジスタ間でのデータ転送を行う命令である.On-Chip SRAM 領域とレジスタ間では通常の load/store も利用できるようであるが, 連続した領域へのスループット向上のためにこれらの命令を追加している. これらは SIMD 演算器への利用が考えられ, ベクトルプロセッサのような高いスループットを実現することを目的としているようである.

本研究との比較

SCIMA の研究と本研究との違いは主にデータ転送粒度の違いにある. SCIMA ではサイズ可変なデータ転送命令を持つが,SCALT では粒度固定のデータ転送命令である Buffer Fetch/Store のみである. この 2 つの違いは 1 命令で大量のデータを取得するか, 1 命令は細かくとも複数命令で大量のデータを取得するかの違いである. 本論文の 1,2 章で述べたとおり, キャッシュベースアーキテクチャでは複数命令で大量のデータを取得することは困難であった. ただし,SCALT では'tag' の存在により, 複数のメモリアクセスの順序保証を行っているため, このような方式で高いスループットが実現できているが,SCIMA ではあくまで,Load/Store バッファに依存するデータ転送命令であるために,1 操作で大量のデータを取得する以外に方法がない.

そのため,SCALT の適用領域である, ポインタ指向プログラムやランダムアクセスを行うようなプログラムには向かない. しかし,SCIMA ではあくまで大規模科学技術計算向けという目的を持っているため, その点において十分効果的なアプローチといえる.

SCIMA の詳細については [6],[7],[8],[9],[10] にアーキテクチャ詳細, 評価等が述べられている.

4.2.3 その他の関連研究

SCIMA 以外にもレイテンシ隠蔽のアプローチとして, ソフトウェアにおける研究, ハードウェアにおける

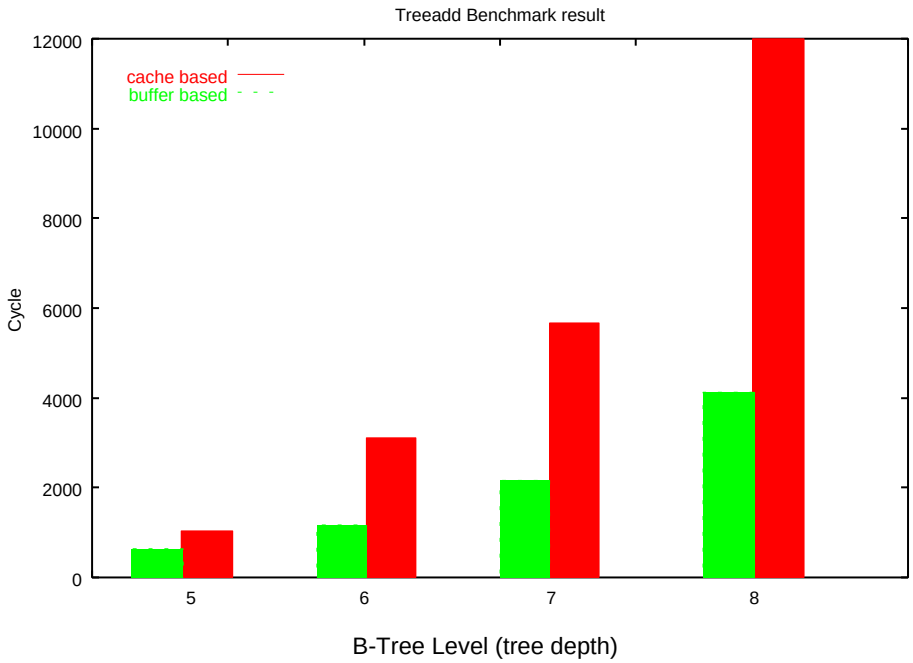


図 4.5: TreeAdd ベンチマーク結果 (Latency 40 Cycle)

研究をおこなっている機関は複数ある.[13],[14],[16],

ベンダとして SCALT,SCIMA と同様のアプローチをとっているベンダは Cray Inc. があり, 商用的に実現しているシステムには T3E,X1 等が挙げられる.

第5章　アーキテクチャ実装

本章ではこれまでに述べた SCALT アーキテクチャを適用したコンピュータシステムの実現について述べる. SCALT コンピュータシステムは大きく分けて 2 つのモジュールで構成している.1 つは演算を行うためのプロセッサ, もうひとつはメモリ,I/O 機能をもつシステムコントローラである. これらのモジュールへ SCALT アーキテクチャで必要な機能を適用し, アーキテクチャの実装を実現するためにはどのようにすればよいのかを述べる. その際, アーキテクチャの特徴を十分に生かすための技術についても検討する.

まず, プロセッサ実装について述べる.

5.1　SCALT プロセッサ (SPK)

まず,SCALT アーキテクチャを実装するにあたり, 現在のプロセッサアーキテクチャを検討した際,Hewlett-Packard(旧 Compaq,DEC) 社の Alpha AXP アーキテクチャの命令セット, メモリパスを持ったプロセッサを設計することとした. これは Alpha AXP アーキテクチャがもつ命令セットが簡潔であり, 柔軟性が高く, また, メモリパスに対して高い技術を有しているからである.

5.1.1　プロセッサアーキテクチャ

Alpha AXP アーキテクチャより規定されている命令セット, データ形式等に合わせてプロセッサを設計する. 現在利用されているレイテンシ隠蔽技術も含めたプロセッサとして設計を検討した. 以下プロセッサの構成について述べる.

5.1.2　データ形式

データ形式は図 5.1 の様に主に 4 つの粒度で扱う. また,1 語内のバイトアドレッシングはリトルエンディアンで格納される.

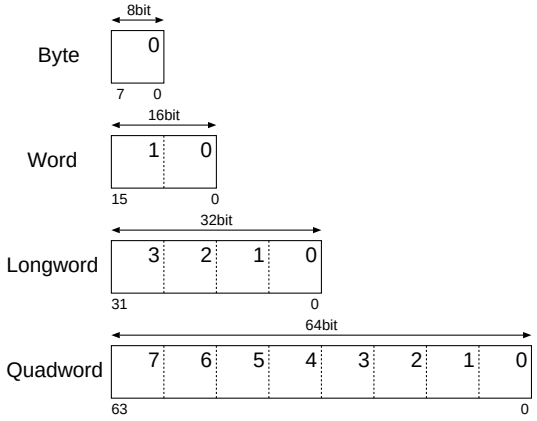


図 5.1: データ形式

図 5.1 内の右上の添字は各バイトのアドレスを示している. バイトアドレッシングはリトルエンディアンである. また,SCALT において用いられる転送ブロック単位である Entry は 256Bit とし, キャッシュライン長も同一とする.

5.1.3　プロセッサ構成

SPK プロセッサは 64bit,2-issue スーパースカラプロセッサであり,7 段のパイプライン, 分岐予測ユニット, 命令, データそれぞれ 32 本のフルアソシアティブなアドレス変換バッファ(ITB,DTB) , 命令, データそれぞれに 32kByte のダイレクトマップ方式のキャッシュッメモリ, 2/2 本の Load/Store バッファを有している. レイテンシ隠蔽機構として Hit-Under-the-Miss 機構 (Load ミス時の Out-of-Order 機構) を有している.

このような構成のプロセッサに 32kByte の SCALT バッファ,SCALT 専用命令を追加した形でプロセッサを実現する. 現在の実装では浮動小数点命令については実装しておらず, 浮動小数点用レジスタファイル, 浮動小数点演算パスはないが, 浮動小数点命令のデコードは行うことができる. プロセッサの論理設計には PARTHENON[25] 環境, ハードウェア記述言語には SFL を用い, プロセステクノロジライブラリは NEC CMOS9 を利用する.

以下, 内部構成について述べる.

プロセッサパイプライン

SPK プロセッサは 7 段パイプラインの命令パイプラインとメモリへのリクエストパス, リフィルパスによって構成されている. プロセッサのパイプライン構成を図 5.2 に示す.

5.1.4 レジスタ

プロセッサ内のデータ記憶素子として, 汎用レジスタ, プロセッサ内部レジスタ, パイプラインレジスタの 3 つで構成される. それぞれについて説明する.

5.1.5 汎用レジスタ

汎用的に用いることのできるレジスタファイルが 32 個あり,0〜31 まで番号付けされている. 汎用レジスタはソフトウェアによって可視であり, 命令コード中にレジスタ番号を指定することで, これらのレジスタを操作することができる. ただし,31 は読み出しが 0 であり, 書き込みが無視される.*Longword* 単位の命令の場合, 汎用レジスタには *Longword* の最上位ビット (31 ビット目) を符号拡張し, 64*bit* データとして扱う.31 をディスティネーションとして指定した命令は Nop(No OPeration) として動作する.JMP 命令の場合は分岐制御のみが行われる (戻りアドレスが格納されない).

5.1.6 プロセッサ内部レジスタ

プロセッサ固有の管理情報である Internal Processor Registers(IPRs) をプロセッサ内部レジスタと呼ぶ. プロセッサ内部レジスタは特にオペレーティングシステムによるリソース管理情報, 割り込み制御情報, エラー, 例外処理情報等を管理するレジスタである.IPRs はソフトウェアによる直接の操作はできず, 特権命令 (PAL 命令) でのみ操作することが可能である. 例えば, アドレス空間番号の変更, アドレス変換フォールト, 割り込み, コンテキストスイッチ等であり, これらの操作は Privileged Architecture Library(PAL) コードによって処理される.IPRs ので指定するアドレスを表 5.1 に示す.

ビット	例外発生原因
0	Software completion(SWC)
1	Invalid Operaton(INV)
2	Division by zero(DZE)
3	Overflow(OVF)
4	Underflow(UNF)
5	Inexact Result(INE)
6	Integer Overflow(IOV)
7	Page fault(Translation Not Valid)
8	Access Violation
9	Fault on Read
10	Fault on Write
11	Illigal Alignment

表 5.2: 例外発生要因レジスタ

例外処理用レジスタ

例外処理用レジスタとして IPRs 内では例外復帰 PC, 例外発生原因, 例外エントリポイントが存在する. それぞれについて説明する.

- 例外復帰 PC
  割り込み, または PAL 呼出命令, メモリアクセス等による例外が発生した場合, 例外復帰 PC に例外を発行した命令のアドレスが格納される. 外部割り込み等の場合は命令フェッチユニットの PC が格納される.
- 例外発生原因
  例外ルーチンへ移行した際, どのような例外が発生したのかを検証するためのレジスタである. 例外発生要因は表 5.2 のようになる.
- 例外エントリポイント
  例外処理ルーチン呼出し時のエントリポイントのベースアドレスである. このアドレスと例外発生原因によって目的のルーチンの指定を行う.

PAL ベースアドレス

Privileged Architecture Library(PAL) コード呼出し時の PAL コードのベースアドレスを指定する. このアドレスと PAL 命令で指定された即値の OR を取ることで目的のライブラリを指定する.

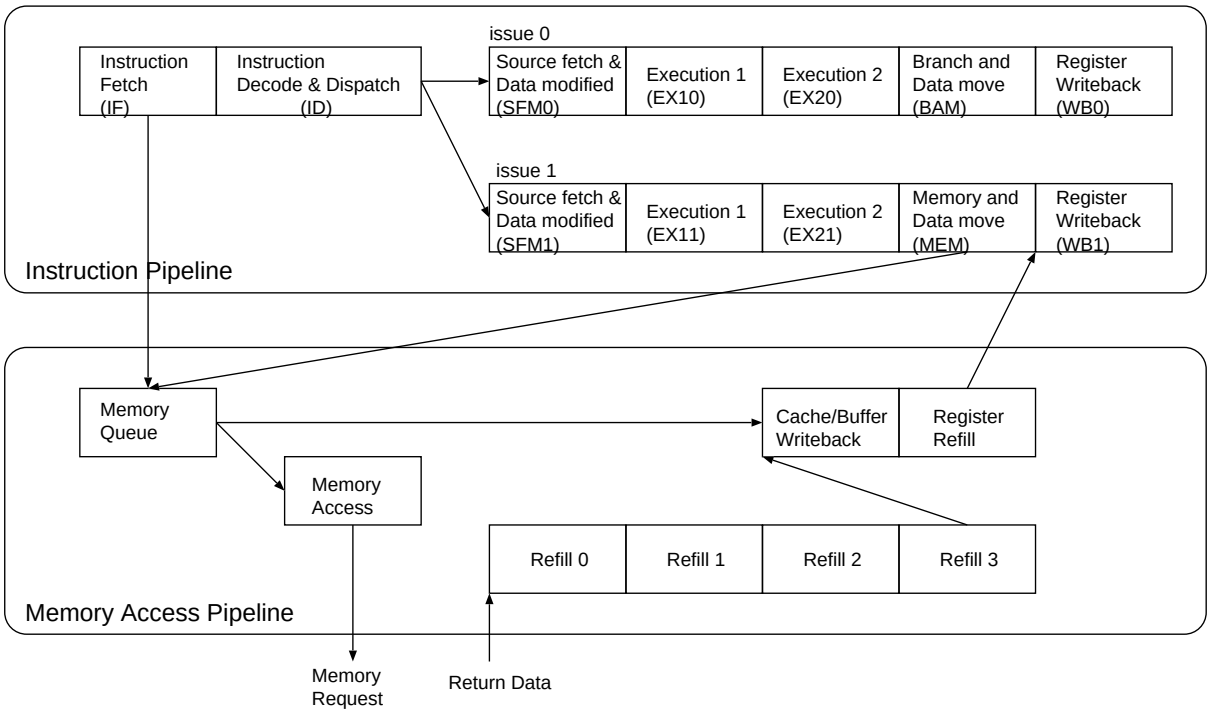


図 5.2: パイプライン構成

5.1.7 プロセッサステータスと割り込み

PS はプロセッサモードを指定する.PS は 3bit の空間であり, カーネル, スーパーバイザ, ユーザのいずれかの状態をとる. プロセッサモードによっては機能しない命令がある.

割り込み制御については割り込み優先レベル, 割り込み ID の 2 つのレジスタを持つ, 割り込み優先レベルは 16 レベルの割り込みを制御するマスクであり, 割り込み ID はソフトウェア割り込み識別子である.

機能制御操作レジスタ

アドレス変換制御, 浮動小数点制御の 2 つは共に 1bit のフラグであり, これらのビットが 1 の時機能が有効となる. 浮動小数点制御ビットが無効となっている場合, 浮動小数点命令はデコードステージにてこのビットを確認し, 浮動小数点例外 (無効命令例外) を発行し, 例外ルーチンへ分岐する. アドレス変換制御ビットが 0 の時はアドレス変換を行わずに, システムへメモリリクエストを発行する.

アドレス管理ユニット操作レジスタ

仮想アドレスサポートがされるプロセッサにはアドレス変換バッファ(Translation Buffer:TB) というものがあり, オペレーティングシステムが管理する仮想アドレス空間と実アドレス空間の対応表であるアドレス変換テーブルのキャッシュを持っている. このアドレス変換バッファはオペレーティングシステムによってコピーの整合性が保たれるが, プロセスが破棄された場合の対応するすべてのエントリに対するページテーブルの無効化操作を行うためのプロセッサ側で用意する制御用レジスタが, このアドレス管理ユニット操作レジスタである.

SPK では命令用, データ用にそれぞれアドレス変換バッファを有しているため, それぞれを操作する目的で ITB_FL,DTB_FL を持っている. 各エントリは 4bit を持っており特権命令である ARCH_ST 命令でこれらのレジスタにフラグを立てることで無効化操作を行う.

Address	Name	Access	Comment
0x00	EXCP_PC	Read	例外復帰 PC
0x01	EXCP_SUM	Read	例外発生原因
0x02	EXCP_BASE	Read/Write	例外エントリポイント
0x03	PAL_BASE	Read/Write	PAL ベースアドレス
0x04	IPL	Read/Write	割り込み優先レベル
0x05	PS	Read/Write	プロセッサステータス
0x06	INT_ID	Read/Write	割り込み ID
0x07	CC_h	Read/Write	CycleCounter(High)
0x08	CC_l	Read	CycleCounter(Low)
0x09	F_VA	Read	例外発生アドレス
0x0a	ASN	Read/Write	命令アドレス空間番号
0x0b	ASN	Read/Write	データアドレス空間番号
0x0c	ITB_FL	Write	命令 TB 制御
0x0d	DTB_FL	Write	データ TB 制御
0x0e	ATC	Read/Write	アドレス変換制御
0x0f	FPC	Read/Write	浮動小数点制御
0x10	SB_BASE	Read	SCALT バッファベースアドレス
0x11	SB_SIZE	Read	SCALT バッファ容量

表 5.1: Internal Processor Registers(IPRs)

SCALT バッファサポートレジスタ

SPK プロセッサ内に存在する SCALT バッファの物理アドレス上の先頭アドレスを SB_BASE に格納してあり, SCALT バッファ容量を SB_SIZE に格納している. オペレーティングシステムはこれらの情報を用いて SCALT バッファの利用を管理する.

5.1.8 パイプラインレジスタ

パイプライン化するにあたり, 各パイプラインステージの間で中間結果をラッチするためのレジスタである. パイプラインレジスタはソフトウェアからは完全に不可視であり, デコードされた命令の制御信号によってのみレジスタは操作される. パイプラインレジスタ以外にも Lock フラグ,VAX フラグ等の制御レジスタについても特定の命令でのみ設定されるレジスタには直接操作できない.

5.1.9 命令形式

SPK プロセッサでは Alpha AXP 命令セットアーキテクチャに規定される命令に, SCALT アーキテクチャ専用命令 (BF,BS,BC),IPRs 操作命令 (ARCH_LD,ARCH_ST,ARCH_
の 6 命令を追加した形で命令セットを構成した.

5.1.10 命令フォーマット

1 命令は 32bit 固定長であり, 命令の格納形式は図 5.3 に示すように機能毎に分類される.

IPRs 操作命令

特権アーキテクチャライブラリ (PAL) コード中でのみ利用可能な命令として ARCH_LD,ARCH_ST,ARCH_REI の 3 命令を追加した. これらの命令は PAL モード (PAL コード以外) ではデコード時に違法命令例外を起こす. それぞれの命令について説明する.

- ARCH_LD
  ARCH_LD 命令は IPRs をよりデータを読み

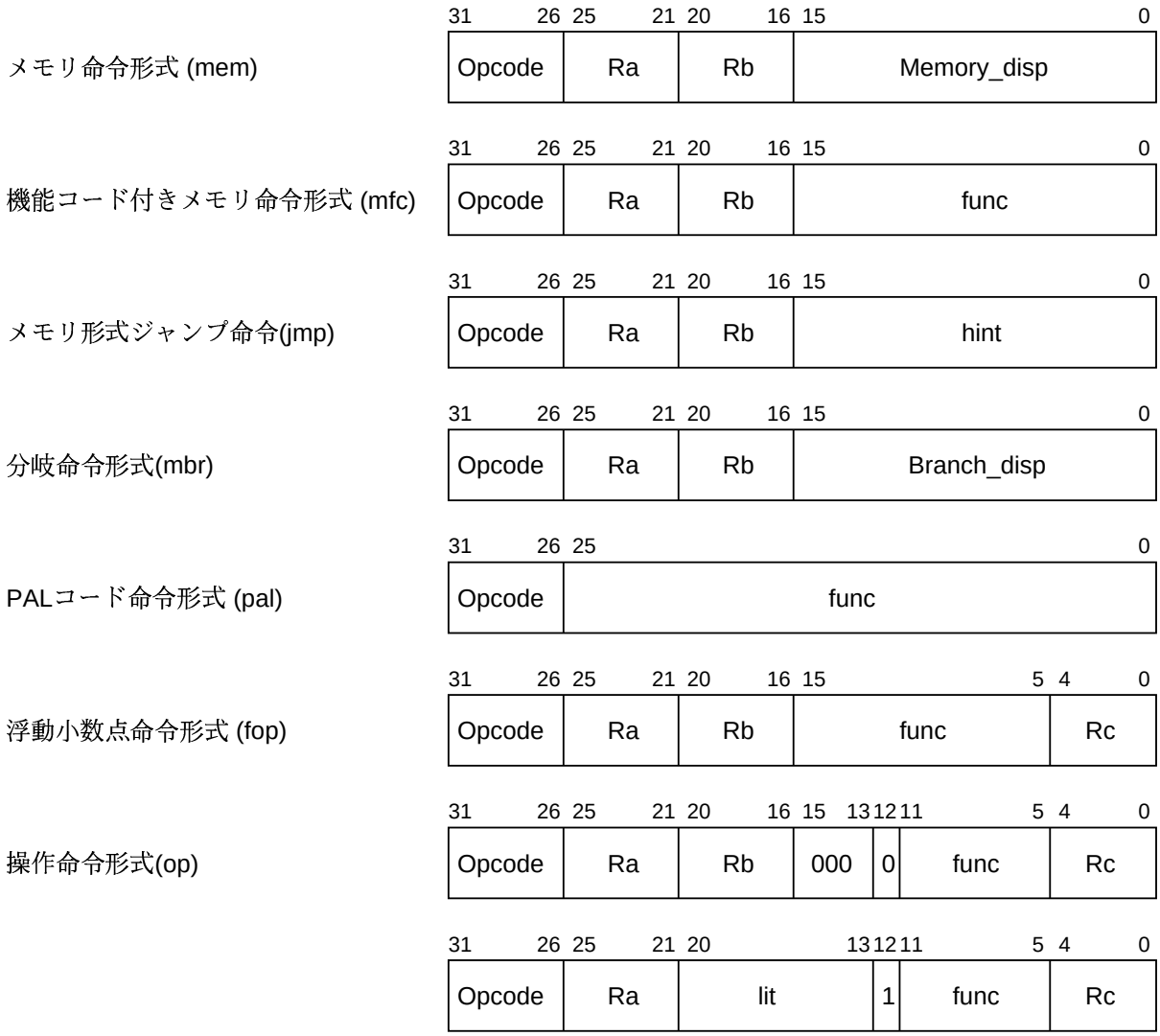


図 5.3: SPK 命令フォーマット

出す命令であり,mem フォーマットである.

- ARCH_ST
  ARCH_ST 命令は IPRs に対してデータを書き込む命令であり,mem フォーマットである.
- ARCH_REI
  ARCH_REI 命令は PAL ライブラリモード, や例外処理ルーチンから復帰するための命令である. この命令では IPRs から EXCP_PC を取り出し, 命令フェッチユニットのプログラムカウンタに PC を書き込み,PAL モードから復帰する.

5.1.11 命令セット

図 5.3 に命令一覧を示す. 各項目について説明する.

- Mnemonic
  プロセッサの命令のアセンブラ表記上の名前を示している.
- Opcode
  命令の op フィールド中のビットを 16 進数で表現している. ファンクションフィールドを持たない命令は oo(16 進数) ファンクションフィールドを持つ命令では oo.ffff(oo は op フィールド,ffff はファンクションフィールド) のように

表現している.

- Format
  命令形式を示している.Format の名前は図 5.3 中の名前を用いている.

- Function
  各命令で行われる操作を示している.Alpha AXP の命令セットでは Londword という記述が無い場合, 演算は 64bit 単位でしか行われない. バイト操作命令 (EXTxx, INSxx, MSKxx, ZAPx, CMPBGE) では 64bit のデータを 8 バイト単位に区切り, 演算を行う.

- issue[0/1]
  SPK プロセッサの命令実行パイプラインである issue0 と issue1 のどちらにディスパッチされるのかを示している.0,1 の場合, デコードされた命令はいずれの issue に対しても命令を発行可能である. issue 0 のみの命令は分岐命令またはモード変更を伴う命令や特権命令である. また,issue 1 のみの命令はメモリアクセス命令である.

5.1.12 分岐予測バッファ

SPK プロセッサの命令パイプラインは比較的長い構成であるため, 分岐命令時のパイプライン内に最大 10 命令が破棄されることになる. 分岐命令は一般的に命令の 10%程度の出現頻度であるため, 分岐命令ごとにこれだけの命令を破棄するのでは, 命令実行効率が極端に低くなる. そのため,SPK プロセッサも分岐予測バッファを持つ, 分岐予測バッファはフルアソシアティブな構成で 10 エントリの履歴バッファを持つ, 分岐予測バッファの構成を図 5.4 に示す. 分岐予測バッファの各エントリは命令アドレスと分岐先アドレスのペアを保持しており, PC と命令アドレスが一致した場合に PC に分岐先アドレスをセットすることで, 過去に分岐のあった命令については分岐するということで, パイプラインの命令破棄を緩和することが目的である. 分岐予測が行われた命令は予測に関係なく実行される. ただし, 分岐予測が行われたかどうかのフラグも共にパイプラインに流れ分岐が本当に行われるかを検証する.

分岐予測バッファ内に保持される命令アドレスと分岐先アドレスは共に仮想アドレスである.

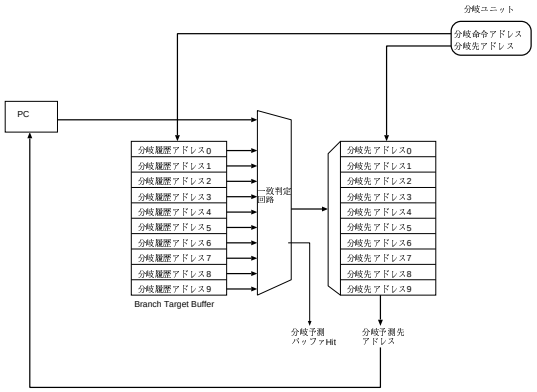


図 5.4: 分岐予測バッファ

5.1.13 アドレス変換バッファ(ITB,DTB)

コンピュータシステムははマルチユーザ, マルチタスクを実現するために実アドレス空間と仮想アドレス空間との間でアドレス変換を行う. アドレス空間の変換はオペレーティングシステムが行うが, プログラムがメモリアクセスをするたびにオペレーティングシステムのアドレス変換ルーチンを実行するのではあまりに効率が悪く, 現実問題として, 実用的なものではない. そのため, オペレーティングシステムを稼動するプロセッサはアドレス変換高速化機構として, アドレス変換バッファ(Translation Buffer:TB) を有している. アドレス変換バッファは命令, データを独立したバッファ用意するシステムが多く, それぞれ, 演算器 (アドレス計算機構) とキャッシュメモリの間に置かれる. アドレス変換バッファの構成はオペレーティングシステムのアドレス変換履歴のコピーを保持しており, ほとんどの場合, フルアソシアティブな構成をとっている.SPK プロセッサにおいてもこのフルアソシアティブな構成をとっており, 命令, データ用のバッファを有する. それぞれアドレス変換バッファを ITB,DTB と呼ぶこととし, 各 TB は 32 エントリ, リプレースメントアルゴリズムを FIFO とし, 構成した.TB の構成は図 5.5 のようになる.

SPK システムにおいては最小物理ページサイズを 64kB としているため, 各 TB によって $64kB * 32 = 2MB$ となり,2MB 以内のアドレスへのアクセスに

表 5.3: 命令一覧

Mnemonic	Opcode	Format	Function	issue[0/1]
LDL	28	mem	Load Longword	1
LDQ	29	mem	Load Quadword	1
LDL_L	2A	mem	Load Longword Locked	1
LDQ_L	2B	mem	Load Quadword Locked	1
LDQ_U	0B	mem	Load Quadword Unaligned	1
STL	2C	mem	Store Longword	1
STQ	2D	mem	Store Quadword	1
STL_C	2E	mem	Store Longword Conditional	1
STQ_C	2F	mem	Store Quadword Conditional	1
STQ_U	0F	mem	Store Quadword Unaligned	1
LDA	08	mem	Load Address	0,1
LDAH	09	mem	Load Address High	0,1
FETCH	18.8000	mfc	Prefetch Data	0,1
FETCH_M	18.A000	mfc	Prefetch Data Modify Intent	0,1
MB	18.4000	mfc	Memory Barrier	0,1
RC	18.E000	mfc	Read and Clear	0,1
RPCC	18.C000	mfc	Read Process Cycle Counter	0,1
RS	18.F000	mfc	Read and Set	0,1
TRAPB	18.0000	mfc	Trap Barrier	0,1
JMP	1A.0	mbr	Jump	0,1
JSR	1A.1	mbr	Jump to Subroutine	0,1
JSR_Coroutine	1A.3	mbr	Jump to Subroutine Return	0,1
RET	1A.2	mbr	Return from Subroutine	0,1
BEQ	39	bra	Branch if (Ra eq 0)	0,1
BGE	3E	bra	Branch if (Ra ge 0)	0,1
BGT	3F	bra	Branch if (Ra gt 0)	0,1
BLBC	38	bra	Branch if Ra Low Bit Is Clear	0,1
BLBS	3C	bra	Branch if Ra Low Bit Is Set	0,1
BLE	3B	bra	Branch if (Ra le 0)	0,1
BLT	3A	bra	Branch if (Ra lt 0)	0,1
BNE	3D	bra	Branch if (Ra ne 0)	0,1
BR	30	bra	Unconditional Branch	0,1
BSR	34	bra	Branch to Subroutine	0,1
ADDL	10.00	op	Add Longword	0,1
ADDL/V	10.40	op	Add Longword /IOV	0,1
ADDQ	10.20	op	Add Quadword	0,1
ADDQ/V	10.60	op	Add Quadword /IOV	0,1
SUBL	10.09	op	Subtract Longword	0,1
SUBL/V	10.49	op	Subtract Longword /IOV	0,1
SUBQ	10.29	op	Subtract Quadword	0,1
SUBQ/V	10.69	op	Subtract Quadword /IOV	0,1

表 5.4: 命令一覧 (続き)

Mnemonic	Opcode	Format	Function	issue[0/1]
CMPEQ	10.2D	op	Compare Signed Quadword (Ra == Rb)	0,1
CMPLT	10.4D	op	Compare Signed Quadword (Ra lt Rb)	0,1
CMPLE	10.6D	op	Compare Signed Quadword (Ra le Rb)	0,1
CMPULT	10.1D	op	Compare Unsigned Quadword (Ra ult Rb)	0,1
CMPULE	10.3D	op	Compare Unsigned Quadword (Ra ule Rb)	0,1
CMPBGE	10.0F	op	Compare Byte	0,1
S4ADDL	10.02	op	Scaled Add Longword by 4	0,1
S4ADDQ	10.22	op	Scaled Add Quadword by 4	0,1
S4SUBL	10.0B	op	Scaled Subtract Longword by 4	0,1
S4SUBQ	10.2B	op	Scaled Subtract Quadword by 4	0,1
S8ADDL	10.12	op	Scaled Add Longword by 8	0,1
S8ADDQ	10.32	op	Scaled Add Quadword by 8	0,1
S8SUBL	10.1B	op	Scaled Subtract Longword by 8	0,1
S8SUBQ	10.3B	op	Scaled Subtract Quadword by 8	0,1
AND	11.00	op	Logical Product	0,1
BIS	11.20	op	Logical Sum (OR)	0,1
XOR	11.40	op	Logical Difference	0,1
BIC	11.08	op	Logical Product with Complement	0,1
ORNOT	11.28	op	Logical Sum with Complement	0,1
EQV	11.48	op	Logical Equivalence (XORNOT)	0,1
CMOVEQ	11.24	op	Conditional Move Integer if (Ra eq 0)	0,1
CMOVLT	11.44	op	Conditional Move Integer if (Ra lt 0)	0,1
CMOVLE	11.64	op	Conditional Move Integer if (Ra le 0)	0,1
CMOVNE	11.26	op	Conditional Move Integer if (Ra ne 0)	0,1
CMOVGE	11.46	op	Conditional Move Integer if (Ra ge 0)	0,1
CMOVGT	11.66	op	Conditional Move Integer if (Ra gt 0)	0,1
CMOVLBS	11.14	op	Conditional Move Integer if (lbs)	0,1
CMOVLBC	11.16	op	Conditional Move Integer if (lbc)	0,1
SLL	12.39	op	Shift Left Logical	0,1
SRA	12.3C	op	Shift Right Arithmetic	0,1
SRL	12.34	op	Shift Right Logical	0,1
EXTBL	12.06	op	Extract Byte Low	0,1
EXTWL	12.16	op	Extract Word Low	0,1
EXTLL	12.26	op	Extract Longword Low	0,1
EXTQL	12.36	op	Extract Quadword Low	0,1
EXTWH	12.5A	op	Extract Word High	0,1
EXTLH	12.6A	op	Extract Longword High	0,1
EXTQH	12.7A	op	Extract Quadword High	0,1

表 5.5: 命令一覧 (続き)

Mnemonic	Opcode	Format	Function	issue[0/1]
INSBL	12.0B	op	Insert Byte Low	0,1
INSWL	12.1B	op	Insert Word Low	0,1
INSLL	12.2B	op	Insert Longword Low	0,1
INSQL	12.3B	op	Insert Quadword Low	0,1
INSWH	12.57	op	Insert Word High	0,1
INSLH	12.67	op	Insert Longword High	0,1
INSQH	12.77	op	Insert Quadword High	0,1
MSKBL	12.02	op	Mask Byte Low	0,1
MSKWL	12.12	op	Mask Word Low	0,1
MSKLL	12.22	op	Mask Longword Low	0,1
MSKQL	12.32	op	Mask Longword Low	0,1
MSKWH	12.52	op	Mask Word High	0,1
MSKLH	12.62	op	Mask Longword High	0,1
MSKQH	12.72	op	Mask Quadword High	0,1
ZAP	12.30	op	Zero Bytes	0,1
ZAPNOT	12.31	op	Zero Bytes Not	0,1
MULL	13.00	op	Multiply Longword	0
MULL/V	13.40	op	Multiply Longword /IOV	0
MULQ	13.20	op	Multiply Quadword	0
MULQ/V	13.60	op	Multiply Quadword /IOV	0
MULH	13.30	op	Multiply Quadword High	0
CALL_PAL	00	pal	Call PAL routine	0
ARCH_LD	1D	mbr	Load from IPRs	0
ARCH_ST	1E	mbr	Store from IPRs	0
ARCH_REI	1F	mbr	Return from PAL mode	0
BF(OPC01)	01	mem	SCALT Buffer Fetch	1
BS(OPC02)	02	mem	SCALT Buffer Store	1
BC(OPC03)	03	mem	SCALT Buffer Check	1

略称	操作
Reserved	予約
V	Valid
FOR	Fault on Read
FOW	Fault on Write
ASM	Address Space Match
GH	Page Granuality Hint
KRE	Kernel Read Enable
ERE	
SRE	Supervisor Read Enable
URE	User Read Enable
KWE	Kernel Write Enable
EWE	
SWE	Supervisor Write Enable
UWE	User Write Enable
SBA	SCALT Buffer Allocated
SBS	SCALT Buffer セグメント分割

表 5.7: 略称の説明

関してはオペレーティングシステムを介在することなくメモリへアクセスすることが可能となる.

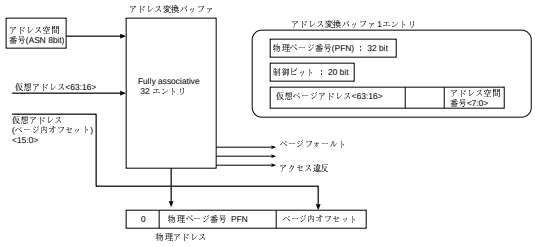


図 5.5: アドレス変換バッファ

このアドレス変換バッファはあくまでオペレーティングシステムのアドレス変換テーブルのコピーであり, コピーの制御はオペレーティングシステムが行う, オペレーティングシステムが管理するための制御ビットが 20 ビット用意されており, 各ビットについての意味を表 5.6 に示す. 命令 TB, データ TB によって各ビットの意味が異なる. また, 各ビットの意味については略称を用いている. 略称の説明は表 5.7 に示す.

Alpha AXP アーキテクチャにおいて, ページ粒度ヒントビット (GH) というものが 2bit 用意されており, 連続した物理ページ領域をアドレス変換バッファの 1 エントリにマッピングすることが可能となっている. GH ビットによって 4 つのページサイズサポートが実現でき,64kB,512kB,4MB,32MB の 4 つを指定することができる. これにより, 最大 1GB までの連続アドレスをオペレーティングシステムの介入なしで行うことができる.

TB 制御ビットの SBA ビットが有効な場合, そのエントリは SCALT バッファを利用していることを示す. アドレス変換要求したアドレスの TB エントリのこのビットが有効な場合〉CALT バッファへアクセスする. また 2bit の SBS ビットは現在利用していないが, バッファ領域の分割またはページサイズ以上のバッファ領域へのアクセス時のコントロールパス制御ヒントとして利用することが考えられる.

5.1.14 キャッシュメモリ

SPK プロセッサは命令, データそれぞれ 32kB のキャッシュメモリを持ち,SCALT バッファと同階層に位置している. キャッシュメモリはダイレクトマップなキャッシュメモリで,1 キャッシュブロック 32B, インデックス 10bit(1024 エントリ) の構成を採っている.

キャッシュメモリは SCALT バッファとは異なりハードウェアによるアドレス管理をしているため, 各エントリに存在するデータブロックのアドレスをキャッシュタグへ保持している.

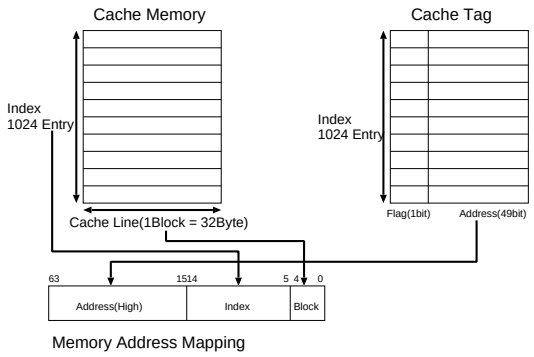


図 5.6: キャッシュメモリ

制御ビット	命令 TB	データ TB
bit<19 >	Reserved	Reserved
bit<18 >	Reserved	SBS
bit<17 >	Reserved	SBS
bit<16 >	Reserved	SBA
bit<15 >	Reserved	URE
bit<14 >	Reserved	SWE
bit<13 >	Reserved	EWE
bit<12 >	Reserved	KWE
bit<11 >	URE	URE
bit<10 >	SRE	SRE
bit<9 >	ERE	ERE
bit<8 >	KRE	KRE
bit<7 >	Reserved	Reserved
bit<6 >	GH	GH
bit<5 >	GH	GH
bit<4 >	ASM	ASM
bit<3 >	Reserved	Reserved
bit<2 >	Reserved	FOW
bit<1 >	Reserved	FOR
bit<0 >	V	V

表 5.6: アドレス変換バッファ制御ビット

5.1.15 SCALT バッファ

SCALT バッファはキャッシュメモリと同階層に位置し, 容量は 32kB としている. 1 エントリは 32B インデックス 10bit(1024 エントリ) としている. 図 5.7 に SCALT バッファの構成を示す. SCALT バッファにも SCALT バッファタグという TAG ビットをそれぞれのエントリで 1bit 有しており, このビットは該当するエントリに対する BF 命令が発行された際に無効となり, データが SCALT バッファに書き込まれると有効となる. このバッファタグへアクセスする命令として BC 命令がある. この BC 命令を用いることでシステムのレイテンシの指標を測ることも可能となる.

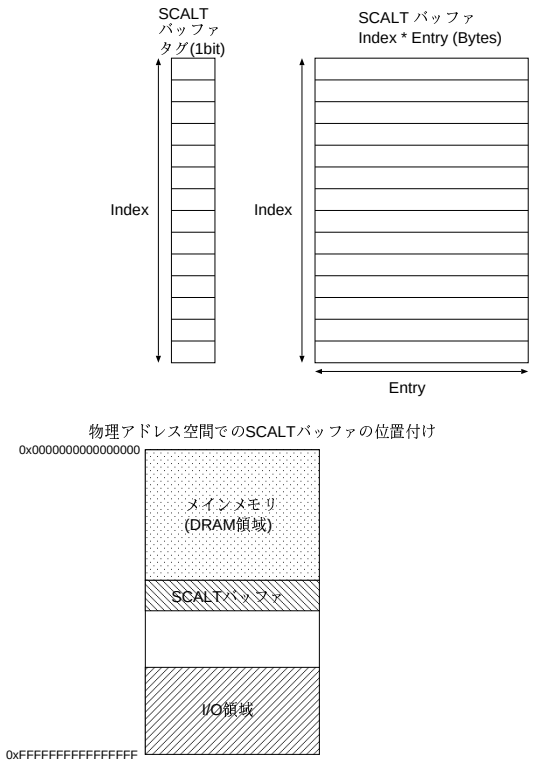


図 5.7: SCALT バッファ

5.1.16 プロセッサ内部構成

これまでに SPK プロセッサの概観とレジスタ, メモリに関連するモジュールを説明した, ここからはそれらのモジュールを組み合わせたプロセッサのコントロールパスとデータパスの設計について述べる.

命令フェッチ (IF)

命令フェッチユニット (IF) は PC で指定されるアドレスを命令 TB へ送り, それと並列にページ内オフセットであるアドレスの <14:5 >のビット 10 で命令キャッシュの該当ラインを取り出す. 取り出したキャッシュライン中より 64 ビット分の領域 (2 命令) を選択する.ITB から出力された実アドレスとキャッシュタグの該当したラインの実アドレスを比較し, 命令キューへ格納する.

これと並行して PC のインクリメントと分岐予測バッファの検証を行う. 分岐予測がヒットした場合, 更新された PC はではなく, PC へは予測されたアドレスを格納する.

外部割り込み信号線から割り込み要求がある場合は命令キューへの書き込みは行わず,Trap カウンタ (パイプラインを流れている命令の内, 例外を発行する可能性の有る命令数を保持している) が 0 となり, かつ割り込みマスクの状態を満たすのであれば割り込みアドレスへ PC を更新する. 命令キャッシュミスの場合, メモリユニットへアドレスを送り, キャッシュへリフィルされるまで停止する.

命令デコード, ディスパッチ (ID)

命令デコーダを 2 つ用意しており, いずれのデコーダも同一のものを利用している. 命令キューよりデコードユニット A,B へ取り出された命令を送り, 各デコーダは命令デコードと同時に Inter-Lock check Unit へ実行パスへデータを送出することが可能かを検証すると共にデータフォワーディングをする際のフォワーディングユニットのエントリを持つ, また, 実行パスの各ステージで利用する制御ビットの生成を行う. 制御ビットについては各ステージで説明する.

命令デコード時に規定されていない命令, 浮動小数点操作無効時の浮動小数点命令等が検出された場合, 命令デコーダは違法命令ビットを立て, 実行ステージへ送出する. このステージから直接例外処理に移行するのではなく, 分岐パスで違法命令ビットを検出することで例外処理へ移行する.

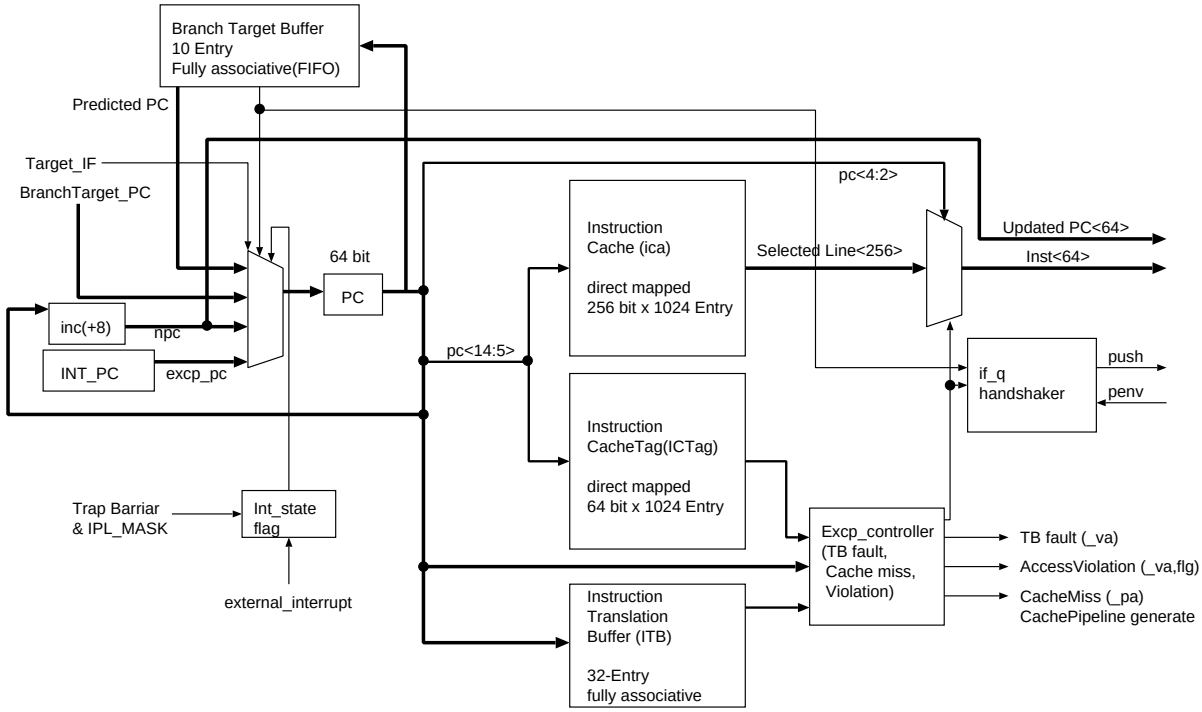


図 5.8: 命令フェッチユニット

ソースフェッチ (sfm)

ソースフェッチは演算器へセレクタ無しでデータを転送するためのデータ成形ユニットであり, 図 5.10 のような構成をとる. 命令パイプラインにディスパッチされた命令は, 命令デコーダによって指定されたリソースである, レジスタ, フォワーディングユニット, 即値,PC のいずれかにアクセスする.

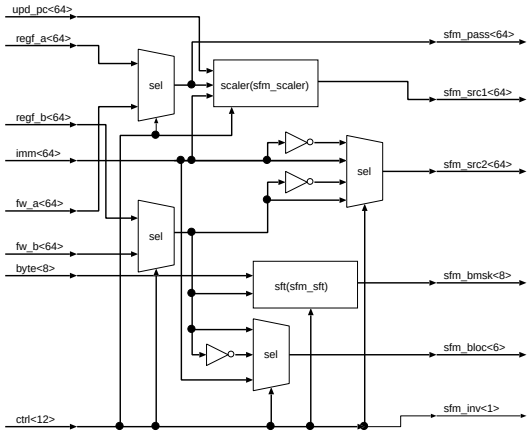


図 5.10: ソースフェッチパス

ビット	シフタ	論理操作
00	SLL	AND(BIC)
01	SLL	BIS(ORNOT)
10	SRL	XOR(EQV)
11	SRA	0

表 5.9: 実行 1 パス制御ビット

実行 1(EX1)

SPK は 64bit プロセッサということもあり,64bit 演算を行う必要がある, しかし,64bit 演算器は 32bit 演算器と比べ, 遅延時間が長い, そのため, 演算を 2 つに分割し, 実行 1 パス, 実行 2 パスで 1 つの演算を行うこととした.

実行 1 パスは図 5.11 のような構成をとる. ソースフェッチパスで成形されたデータに対して, 各演算器が並列に動作を行う. 各演算器の入力までにセレクタが存在しない. 実行ステージにおける制御ビットは表 5.9 に示す通り 3bit である.

実行パス 1 の結果はそのまま実行 2 パスへ転送される.

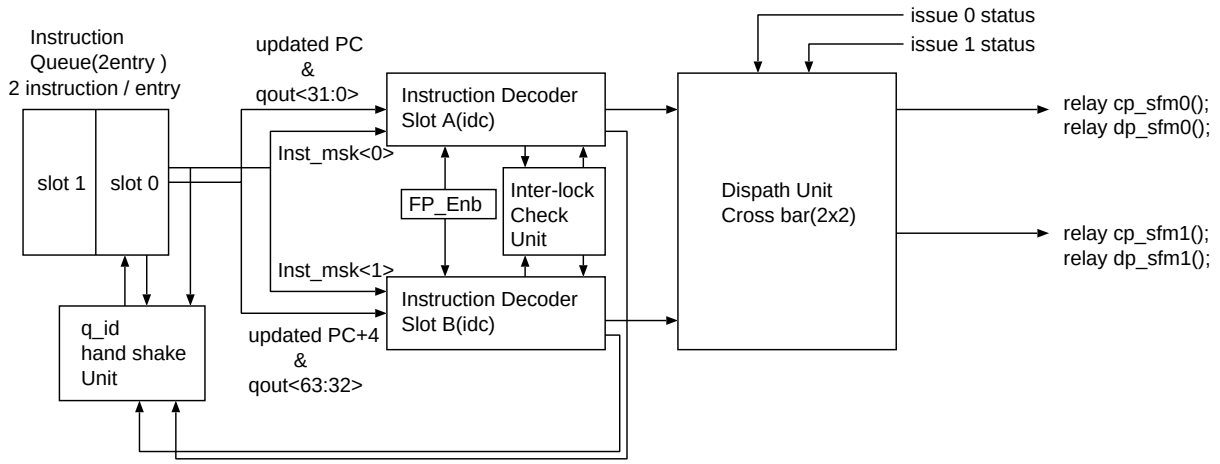


図 5.9: 命令デコード, ディスパッチユニット

ビット	名称	操作
0	SFM_INV_RB	Rb 反転
1	SFM_SCALE4	Rb を 2bit 左シフト
2	SFM_SCALE8	Rb を 3bit 左シフト
3	SFM_USE_FW_A	フォワーディングユニット利用
4	SFM_USE_FW_B	フォワーディングユニット利用
5	SFM_USE_PC	PC を利用
6	SFM_BYTE_OPR	バイトマスク演算
7	SFM_BYTE_LOW	ByteLow 操作
8	SFM_USE_RB	Rb を利用
9	SFM_USE_IMM	即値を利用
10	SFM_BLOC	バイト操作ビットシフト
11	SFM_SAVE_PC	PC をディスティネーションパスへ転送

表 5.8: ソースフェッチパス制御ビット

実行 2(EX2)

実行 2 パスでは実行 1 パスより送られた演算の中間結果を受け, 演算操作の残りの処理を行う. 実行パス 2 の構造を図 5.12 に示す.

実行 2 パスの制御は実行 1 と比較して複雑になっている. また分岐, メモリパスと操作上の類似点が多いためデコードビットをまとめている. 表 5.10 に示す.

デコードビット <11:7 >は命令選択を行うビット郡であり, ビット<6:0 >はそれらの命令グループ内での操作ビットとなっている.

分岐 (BAM)

分岐パスでは実行 2 パスより送られてくるデータであるアドレス, データ, 比較ビットとパイプラインを流れる図 5.10 制御ビットによって目的の演算操作を行う. 分岐パスの構成を図 5.13 に示す.
分岐の確定は制御ビットと実行 2 パスから送られる有効ビットで判定される. 分岐が確定し, 分岐予測がミスした場合はメモリパス, 分岐ユニットに対して分岐予測ミス (分岐信号) を送信する. その際に分岐ステージ内の命令は木ビットをセットし, 後続の命令を破棄する.

分岐パスから分岐ユニットへは命令アドレス, 分岐

ビット	分岐命令	比較命令	条件転送命令	メモリ命令	演算命令	特権命令
演算パス	issue 0	issue 0/1	issue 0/1	issue 0	issue 1	issue 0
11	0	0	0	0	0	1
10	0	0	0	0	1	0
9	1	0	0	1	0	0
8	0	1	0	0	0	0
7	0	0	1	0	0	0
6	UC	USGN	0	Buffer	USGN	CALL_PAL
5	GT	CMPBGE	GT	Fetch	MUL_H	ARCH_REI
4	LT	SLT	LT	Lock/Cond	MUL_L	IPR
3	LBS	0	LBS	Unaligned	0	VAX
2	LBC	0	LBC	Quad	Quad	RPCC
1	NE	0	NE	ST	ZAPNOT	ARCH_ST
0	EQ	SEQ	EQ	LD	/V	ARCH_LD

表 5.10: EX2, 分岐, メモリパス制御ビット

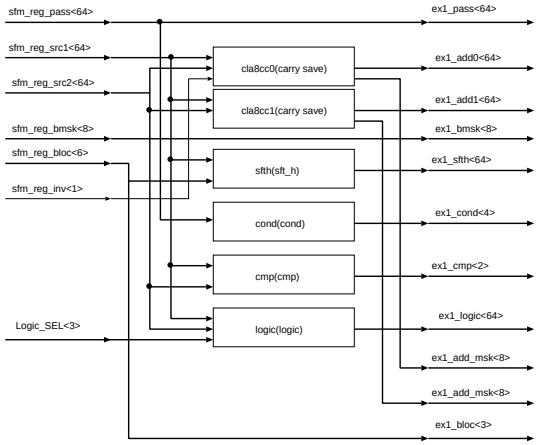


図 5.11: 演算実行 1 パス

先アドレス, 制御情報が転送される. この制御情報は分岐,PAL, 違法命令, 整数命令例外トラップ, IPR アクセス例外がある. 分岐ユニットはそれを受け, 命令キューの破棄, フェッチパスへの PC 変更操作を行う.

分岐パスには IPR レジスタがあり, 制御ビットにより特権命令アクセス可能となっている場合, IPR へのデータ操作を行う. 特権命令アクセスモードは命令アドレスの 0 ビット目が 1 となっているかどうかという情報で検証される.

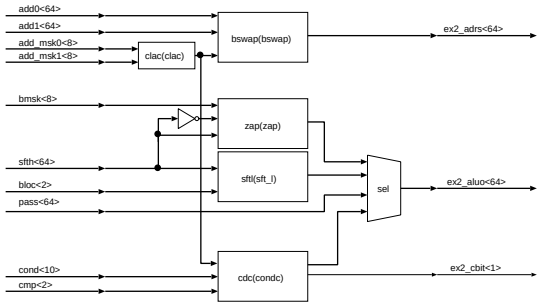


図 5.12: 演算実行 2 パス

メモリ (MEM)

メモリパスでは実行 2 パスより送られてくるデータであるアドレス, データ, 比較ビットとパイプラインを流れる制御ビットによって目的の演算操作を行う. 図 5.14 にメモリパスの構成を示す.

メモリアクセスユニットへデータを転送する際は Load / Store バッファの空きを確認し, データを転送する必要がある. メモリアクセス命令でメモリパスへの転送を必要とする際,Load / Store バッファに空きが無い場合は ID ステージのリトライキューへアドレス, データ, 操作信号を転送し, その Load 命令はもう一度命令をやり直す. これをリトライ操作と呼び, これが Hit under the miss という Load ミス時の Out-of-Order となる.

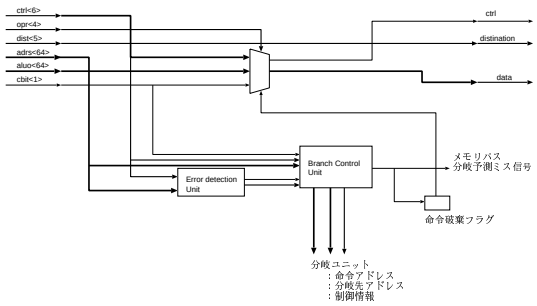


図 5.13: 分岐パス

メモリパスは分岐パス, リフィルパスとの間で依存があり, 分岐パスからの分岐予測ミス信号, リフィルパスからのキャッシュミスした Load 命令のライトバック要求のそれぞれをを監視している. 分岐予測ミス信号がきた場合には命令は破棄され, メモリパス内のフラグを立てる. このフラグが有効である場合, デコード, ソースフェッチ, 実行 0,1 パスを流れている命令がメモリステージに到着した際, 命令を破棄する. また, ライトバック要求がきた場合にはメモリパスの命令を停止させる.

メモリパスにおいてメモリ操作がアドレス変換フォールト, アクセス例外が発生した場合, 分岐ユニットへ命令アドレス, メモリアドレス, 制御情報を転送する. これらは分岐ユニットによって IPR 内に制御情報が格納され, 例外ルーチン内では IPR を確認することで, 例外発生要因を確認することができる.

ライトバック (WB)

ライトバックパスが有効となっている場合, 前のステージまたはリフィルユニットより送られてくるレジスタ番号へデータを書き込みが行われる. 図 5.15 にライトバックパスの構成を示す. 分岐パス, メモリパスからのライトバック要求信号をレジスタファイル, フォワーディング有効信号としてそれぞれに対して演算結果を格納する.

Load/Store バッファ, キュー

MEM ステージ, または IF ステージからの要求はここでキューへ入れられ, 発行順序が確定する. このキューは Load/Store バッファの番号だけが保持さ

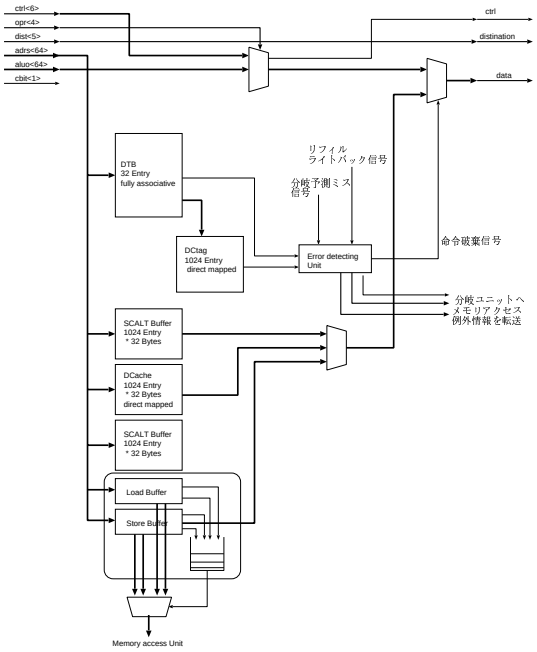


図 5.14: メモリパス

れている. Load/Store バッファの各エントリのビットによってメモリアクセスを行うかどうかを決定する. プロセッサ外へのリクエスト発行順序はキューから取り出すことで確認する. キューから取り出された Load リクエストはメモリアクセスユニットによってリクエストを発行した後も Load バッファ内でアドレスが保持される. BF 命令もこのキューに登録されるが, リクエスト発行後はエントリを解放する. Store リクエストはキャッシュにヒットしているのであればキャッシュリフィルを行うまで,Store バッファ内に留まるが, キャッシュミス時はリクエスト発

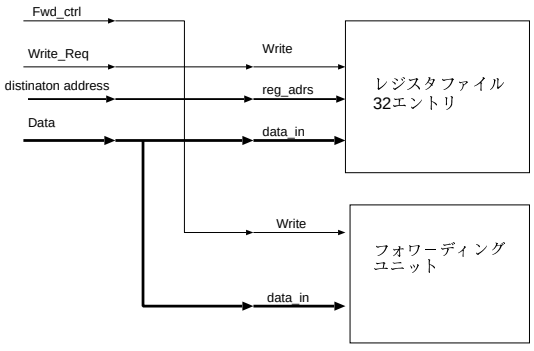


図 5.15: ライトバックパス

行後, ただちに解放される.

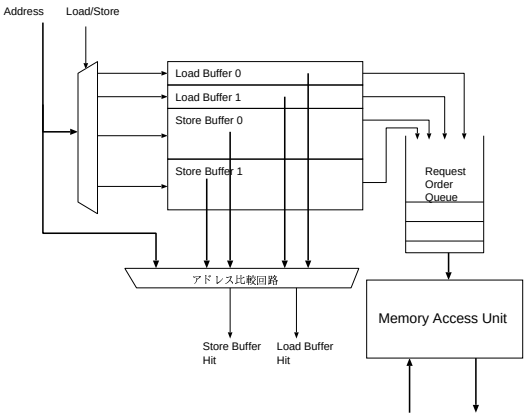


図 5.16: Load/Store バッファとキュー

リフィル

リフィルユニットはプロセッサモジュールの中でただ 1 つ, プロセッサ外部からの信号によって起動されるステージである. リフィルユニットは図 5.17 のような構成をとる.

リフィルユニットの起動はシステムコントローラからの Load 完了信号によって起動され,64bit*4 サイクルでバースト転送される Load データを受信する.Load 完了信号と共に'tag' も受信し,tag の内容によって操作が異なる.

- MSB が 1 のとき
  この場合,Buffer Fetch 命令であることが判別でき, 下位 10 ビットが格納先の SCALT バッファエントリである. リフィルユニットは 64bit*4 のデータをバッファに格納する.

- MSB が 0 のとき
  この場合, キャッシュベースのメモリ要求であることが判別でき, 下位 2 ビットで Load / Store バッファを指定し, Load / Store バッファ内の情報によって命令キャッシュ, データキャッシュの選択, レジスタへのライトバックを行うかどうかを決定する.

リフィルユニットにはもう 1 つ重要な操作である, キャッシュパージ操作がある. キャッシュパージは外部からパージ要求, アドレスが転送され, データキャッシュに対してキャッシュパージを行う. キャッシュパージ要求はキューに格納され, 他のメモリアクセス操作の空き時間を利用してキャッシュパージ処理が開始される.

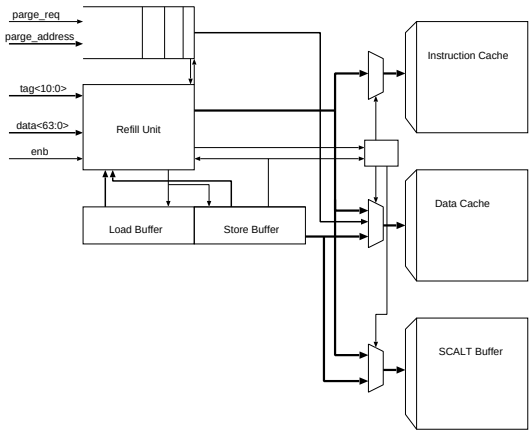


図 5.17: リフィルパス

5.2 システムコントローラ (SSC)

SCALT アーキテクチャの実現に必要な設計コントローラを設計した. ここでは SCALT メモリコントローラ (SSC) についての説明をする.

5.2.1 システムコントローラ概要

SCALT システムコントローラ (以下SSC) はSCALT CPU のメモリアクセス, また外部バスからのメモリ,I/O リクエストを処理するものである.

SCALT アーキテクチャの優位性であるバッファ転送命令によるメモリスループットの向上が期待できるよう 'tag' を効果的に利用した構成となっている.

SSC はシステムバス,I/O ノードからのリクエストを SSC 独自のインターフェースに変換し, また複数のデバイスから同時に発行されるリクエストをメモリバスを有効に活用できるようにアクセス最適化を行う機能を実現する. そして,SSC 同士の通信のためのインターコネクトチャネルを 2 つ用意し,SSC 同士の通信を行うことで NUMA 構成を実現できるようになっている.

SSC は 5 つのモジュールから構成される. この 5 つのモジュール間でメッセージの送受信を行うことでメモリリクエストを処理する.

モジュールを分散することによる利点は次のようなものが挙げられる.

- 機能を分散したことによりシステム構成が変更する場合であっても, SSC 自体の変更をリクエスト部のみに留めることができる.
- 同様に現在は SDRAM に限定したメモリコントローラであるが,VC-DRAM,DDR-DRAM, などへの対応も 1 つの機能モジュールを変更するだけで可能である.
- SCALT の BF 命令のようにプロセッサから大量に発行されるメモリリクエストと, 各インターコネクト間のリクエストを同時に受けることができるようなキュー構成を追加することが容易である.

設計対象となっている SCALT CPU はメモリに対して CPU と非同期にメモリリクエストが発行されるものである. データの到着を待たずに複数のメモリリクエストがプロセッサより発行される. SSC は SSC 内部のキューにあるリクエストを依存関係が保たれる範囲でアクセス最適化を行う. アクセス最適化については後に説明する.

5.2.2 メモリキュー数の予備評価

SCALT アーキテクチャを適用したプロセッサからは大量のメモリリクエストが発行される. そのため, システムコントローラがメモリリクエストを受け付けるキュー数が少ない場合, プロセッサから発行できるメモリリクエストが極端に低下してしまう. システムコントローラを設計するに当たり, キュー数の検討はメモリ性能のなかでも重要な要素であるため, メモリキュー数を決定するために予備評価を行った.

ランダムアクセス耐性

従来の Load / Store バッファを用いたプロセッサのメモリ要求はそれほど多くすることはできなかった. そのため, スループットを向上させるために 1 リクエスト当たりのデータ転送量を増加することでスループットを向上させていた.

しかし, プログラムによっては必ずしも連続的なメモリアクセスということは無く, また, システムコントローラへリクエストが到着するような場合というのはキャッシュ等のプロセッサ内のレイテンシ隠蔽機構で対応できなかったリクエストであるとも考えられる. これまでに従来のレイテンシ隠蔽機構, スループット向上技術の特徴を述べてきたが, そのなかでスループット向上を阻害する要因として, ランダムアクセスを挙げた.

SCALT システムコントローラは SCALT の特徴である大量に発行されるメモリリクエストと'tag' を用いた順不同メモリアクセスが可能という特徴を受け, キュー内で最適化を行う. 最適化については後に述べるが, その最適化を行う際, 候補となるリクエストは多いほどよいため, 実際にどの程度の要素数で最適化の効果が現れるかを検証した.

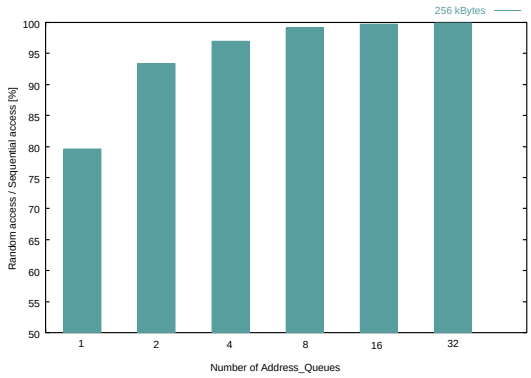


図 5.18: キュー数の変化によるランダムアクセス性能の違い

キュー数の選定

キュー数の決定のために第 4 章で述べたシステムシミュレータを利用して, キュー数を 2〜64 の間で変化し, シーケンシャルアクセス性能, ランダムアクセス性能を行った. 図 5.18 に最適化キュー数ごとのランダムアクセス/シーケンシャルアクセス比を示す.

このような形でランダムアクセス性能が最適化キューの増加に伴い, シーケンシャルアクセス性能に近づく. この結果よりシステムコントローラの設計においてキュー数を 32 と決定した.

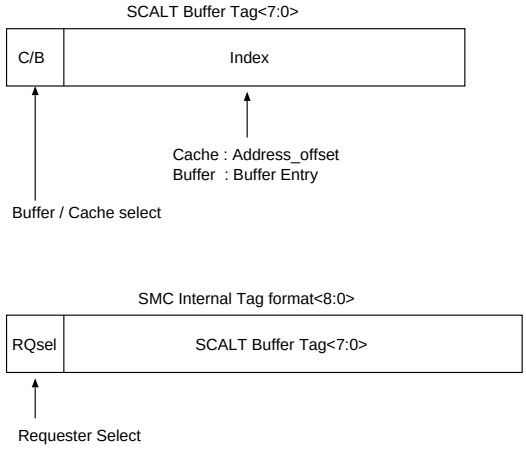
5.2.3 SSC における'tag' の扱い

SCALT アーキテクチャではバッファ, キャッシュからのリクエストの識別符号としてプロセッサから tag と呼ばれる識別符号を受ける.tag は SCALT バッファのインデックスとバッファ, キャッシュ識別フラグから構成される. 現在の実装では 11bit となり,SSC 内部では最適化のヒント, プロセッサ内部ではリクエストの到着順序の変更に対する管理情報として用いられる.tag はアドレスと比較しても十分小さいため, 管理は容易である.

また,SSC 内部ではプロセッサからの tag にリクエスタ識別符号を付加し, データフローを制御する.

5.2.4 SSC の構成

SSC は大きく 5 つの機能モジュールから構成される. メモリコントローラの構造図を図 5.19 に示す.



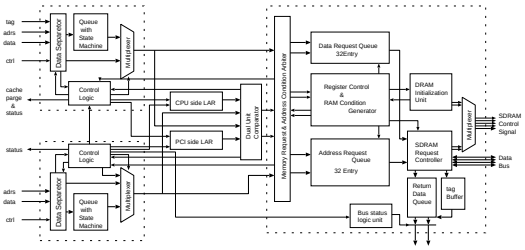


図 5.19: SSC 構成図

現在はメモリには SDRAM を用い,CPU,I/O ノードからのメモリリクエストを処理することを目的としている.

次に各機能モジュールについて説明する.

CPU side Requester (CPUrq)

CPUrq は CPU からのメモリリクエストを受けるモジュールである. CPUrq の構成を図 5.20 に示す.

CPUrq は CPU-SSC 間のバスプロトコルから SSC 内部バスプロトコルへの変換を行い SDRAM Access Queue (SAQ) へメモリリクエストを転送する.

CPU からのメモリリクエストには次のようなものがある

- 制御情報
  この制御情報には Read/Write/Cache Line Write の信号の他に Lock / Conditional 信号がある.
- アドレス
  プロセッサ, または I/O からの物理アドレスで指定される.

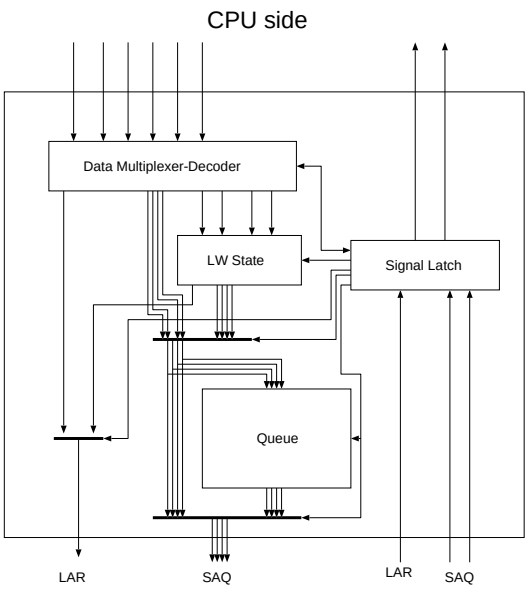


図 5.20: CPU side Requester 構成図

- tag
  SCALT バッファタグが 10bit で格納される.SSC ではリクエストの処理にこの情報を参照することは無いが,SSC 内部でヘッダを付加し, データを返却する際に利用する.

また PCI からのストアリクエストは CPU 内部のキャッシュがメモリとの一貫性を崩してしまうため CPU(キャッシュ) へパージ操作も行う.

SSC 内部のリクエストキューが埋まってしまった場合は現在のメモリリクエストを CPUrq 内部に保持し,CPU に対して, リクエスト停止を伝える.

Lock 信号がアサートされた場合,Locked Address Register (LAR) に Lock 処理を転送し, そのサイクルでのリクエストを保持する. Lock が失敗するとリクエストを停止させ, 保持しておいたリクエストを LAR に発行し続ける.

I/O Requester (PCIrq)

PCIrq の構成を図 5.21 に示す.

PCIrq は PCI バスからのメモリリクエストをバスインターフェースを経由して接続される.

PCI バスからのメモリリクエストは PCI バスインターフェースで PCI デバイス内部のプロトコルに変換し, それを PCIrq (SSC) に転送する.PCI デバイス

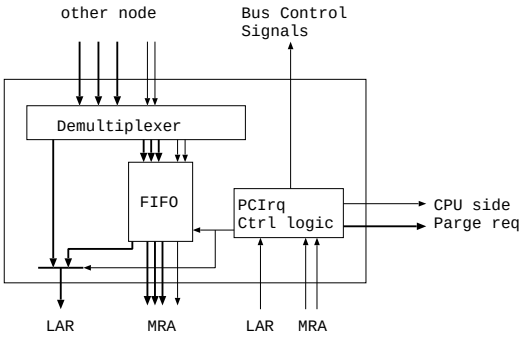


図 5.21: PCI side Requester 構成図

内部プロトコルは SSC 内部プロトコルとほぼ同一のプロトコルにしているため PCI バスからのリクエストに tag を付加して SAQ へ転送する.

識別用タグを付加した信号を SDRAM access queue(SAQ) へ転送する. この識別用タグは PCIrq 内部で生成し, PCI バスサイクルごとに異なるタグを生成する.

PCI バスサイクルの中の Memory Read Multiple コマンドでプリフェッチされたデータのうち必要とされないデータは PCIrq でタグの比較により破棄される.

また CPU 内部キャッシュへのキャッシュパージも CPUrq と連動して行う.

Lock 信号の処理は CPUrq と同様に LAR に対して信号を転送する. Lock の失敗は 1 クロック後に LAR から通知され,I/O バス側に通知される.

Locked Address Register (LAR)

LAR の構成を図 5.22 に示す.

LAR はハードウェアによる小規模なリソースロックを実現するためのモジュールである. PCIrq,CPUrq と接続し, お互いのデバイスからの排他制御を行う.

PCIrq,CPUrq 用にそれぞれ 1 エントリずつアドレス管理レジスタが用意されこのレジスタとセットされた以降の Lock 付きリクエストとの間で排他制御をする. 比較に時間がかかり SSC のクリティカルパスとなることを回避するため,Lock の失敗, リクエストの失敗はラッチされ,1 クロック後に PCIrq, CPUrq,SAQ それぞれに通知される. そのため CPUrq,PCIrq は LAR の結果が通知されるまでリクエストを保持する必要がある.

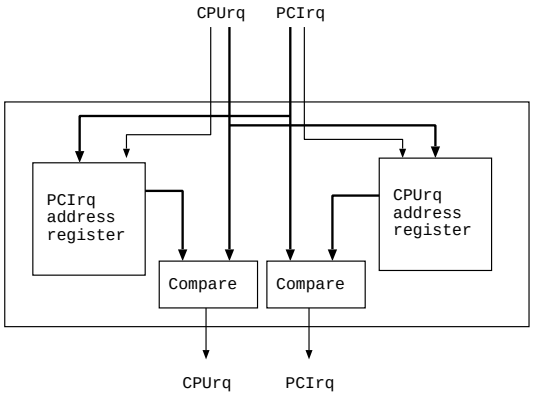


図 5.22: Locked Address Register 構成図

Lock の失敗を通知された CPUrq,PCIrq はそれぞれのリクエスト受付を一時停止して, 保持していたリクエストを再度 LAR,SAQ に発行する. これはリクエストが受付されるまで繰り返し行う.

SDRAM Access Queue (SAQ)

SAQ の構成を図 5.23 に示す.

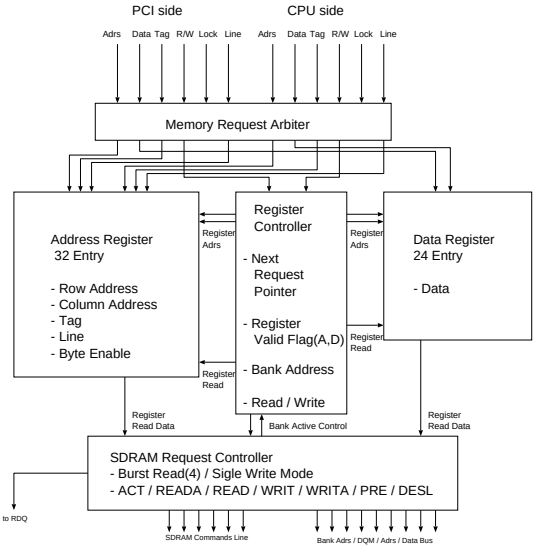


図 5.23: SDRAM Access Queue 構成図

SAQ は 5 つの機能モジュールにより実現されている.

SAQ 内部の機能モジュールは次のようなものである.

- Address Register (a_reg)

  アドレスリクエストをメモリへの発行まで保持する.

  32 エントリで 2 Read,2 Write 型のレジスタファイル

  レジスタの 1 エントリは図 5.24 のような構成になっている.

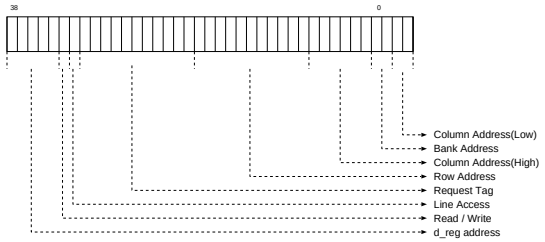


図 5.24: Address Register Entry

- Data Register (d_reg)

  ストアリクエストにより書き込まれるデータを保持する.

  32 エントリで 1 Read,2 Write 型のレジスタファイル

  レジスタの 1 エントリは図 5.25 のような構成になっている.

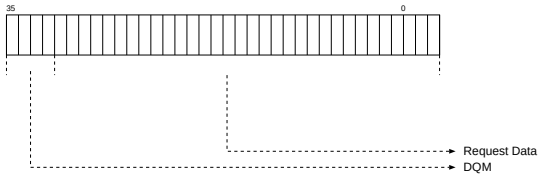


図 5.25: Data Register Entry

- Register Control Unit (RCU)

  RCU は大きく分けて 2 つの機能部から構成される.MRA から送られてきたリクエストの受け付け, レジスタファイル部のアドレス管理を行う Request 部,SDRAM へのアクセス管理. リクエスト発行のタイミング管理を行う Access 部の 2 つを独立して動作させることでシステムとしてのパフォーマンスを向上させる.

  RCU の構成を図 5.26 に示す.

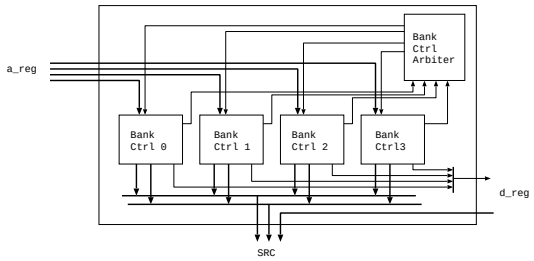


図 5.26: Register Control Unit 構成図

- Memory Request Arbiter (MRA)

  CPUrq,PCIrq からのリクエストを LAR からの信号により判断し,a_reg,d_reg へのリクエストを調停する.(図 5.27)

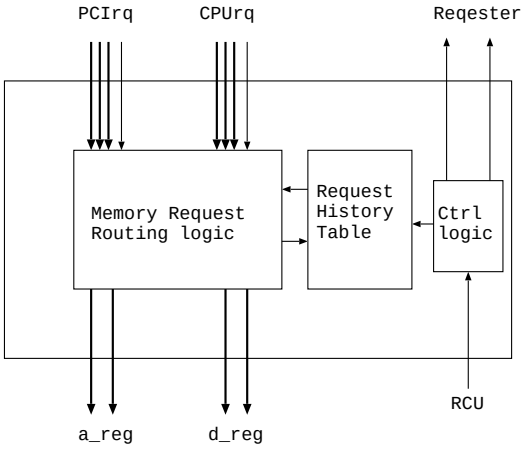


図 5.27: Memory Request Arbiter 構成図

- SDRAM Request Controller (SRC)

  RCU と a_reg,d_reg からのメモリリクエストを SDRAM バスコマンドに変換し,SDRAM へ発行する. また SDRAM の初期化動作を行う. 初期化動作時の外部へのリクエスト中止信号の通知や RDQ へのデータ,tag の転送も行う.(図 5.28)

これらの５つの機能モジュールによりメモリリクエストのパフォーマンスの向上を実現する.

RCU 内部の具体的な動作は次のようになっている.

SAQ はまず MRA が CPUrq,PCIrq からのメモリリクエストを LAR の信号で破棄するかどうかを判断し,SDRAM へリクエストを発行するまでの間アドレス情報, データ情報に分け各レジスタファイル（a_reg,d_reg）に保持する.

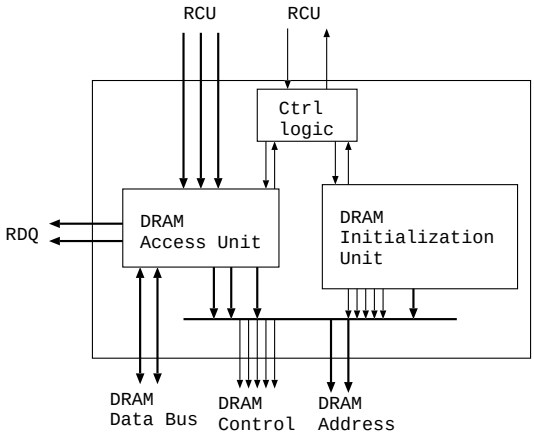


図 5.28: SDRAM Request Controller 構成図

MRA のアービトレーションアルゴリズムは基本的にはラウンドロビンであるが, 性能を向上させるためにバンクアドレスの履歴で判断する.

各リクエストは SAQ 内部の RCU がアドレスレジスタ, データレジスタのアドレスを決定し, 統計情報からエントリの依存関係を崩さないようにアクセス最適化を行う.

アクセス最適化ファクターは主にバンクアドレスである.

アドレスレジスタ, データレジスタは各 32 エントリ用意されている.

どちらかのレジスタが完全に埋まった時,RCU は CPUrq,PCIrq に対してリクエストが受け付けられないことを通知する.

RCU はバンク数分のステートマシンを持つ. 各ステートマシンは RCU_arbiter という管理ユニットに管理され, 各バンク独立したリクエストを SDRAM へ発行する. SRC にリクエストを発行する.

バンクステートマシンの状態遷移図を図 5.29 に示す.

バンクごとのステートマシンの状態とステートマシン同士の通信によりで SDRAM のバスの状態が判断できるためこれらの情報を基に RCU がアクセス最適化を行う.

SRC は RCU からメモリリクエストを受けとると SDRAM バスコマンドに変換し,SDRAM へ転送する. 識別用タグはコマンド発行と同じタイミング内

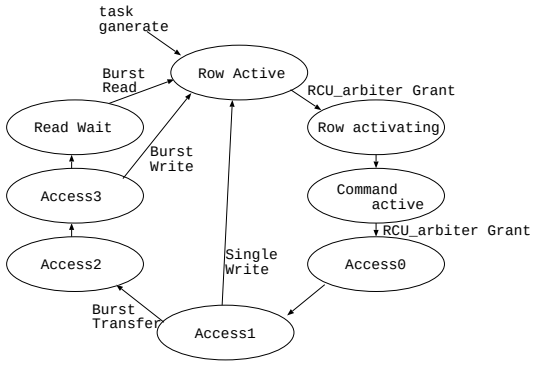


図 5.29: バンクステートマシン状態遷移図

で Return Data Queue （RDQ）へ転送する.

以上の動作を行うことによりメモリへのリクエストが完了する.

Return Data Queue (RDQ)

RDQ の構成を図 5.30 に示す.

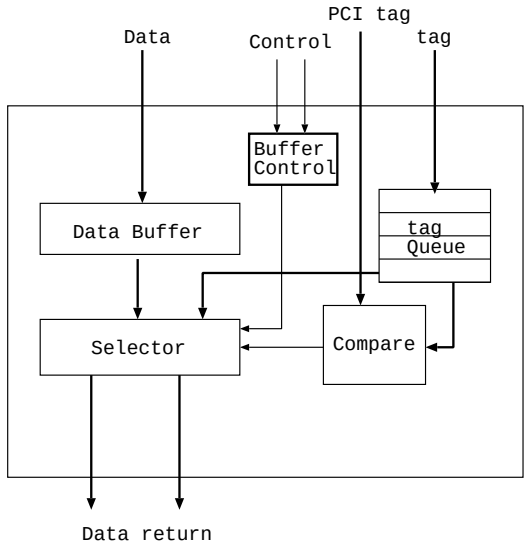


図 5.30: Return Data Queue 構成図

Return Data Queue(以下 RDQ) は SAQ から転送されたリクエスト識別タグ (tag) と SDRAM から読み出されたデータとを対応させ,tag の情報をもとに要求元へデータを送信する.

tag は $\log_2 x + 1$ bit で表され, 最上位ビットにより転送先を決定する.

RDQ に関してもキューが存在し, プロセッサまたは I/O へのデータの転送時の速度差を緩衝する. もし, システムバスのスループットより SDRAM のスループットが大きい場合, キュー数を増加し, Write アクセスを隠蔽することが可能となる.

PCI バスインターフェース側へのデータの転送は一度 PCIrq へ tag とデータを転送し,tag が有効であるかのチェックをする. これは PCI バスの Memory Read Multiple によるプリフェッチによるデータの有効期限を確認するためである.

CPU(SCALT Buffer) へのリクエストは tag <9:0 >が SCALT Buffer tag となるためそのままシステムバスへ tag, データを送信する.

5.2.5 アクセス最適化

SCALT アーキテクチャでは CPU はメモリからのデータの到着を待たずにバッファエントリ数分のメモリリクエストを発行することができる.

SSC は大量に発行されるメモリリクエストをキュー (a_reg) に格納し, メモリビジー率を上げるようにアクセス最適化を行う.

5.2.6 メモリアドレス

パフォーマンスの向上のためシステムのメモリアドレスは図 5.31 のような構成をとった.

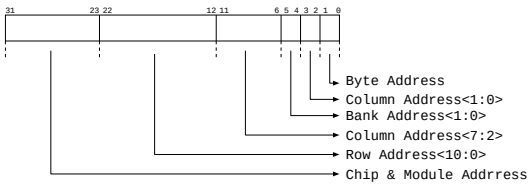


図 5.31: SSC Memory Address

SDRAM が EDO-DRAM 以前の DRAM より優れている点として

- 1 チップ内でのマルチバンク化
- バースト転送効率の向上

という 2 点が挙げられる.

この利点を生かしたメモリアドレスのためこのような構成をとった. まず SDRAM の各バンクは独立

しているためバンクアドレスを低位ビットに割り当てることで各バンクのビジー率を平均して向上することができる. 上の図ではカラムアドレスを分割して割り当てているが, これは CPU の 1 キャッシュラインと同一にしている. SDRAM のバースト長も同様に 4 としている. これはメモリリクエストが主にキャッシュライン単位での転送を目的としているためである.

このようなメモリアドレスの割り当てにより各バンクのビジー率を平均化することとバースト転送機能を生かすことができる.

SSC のアクセス最適化機能

アクセス最適化は主に SAQ 内の機能モジュールである MRA と RCU によって実現される.

- MRA

  MRA は CPU,PCI から発行されるメモリリクエストを a_reg,d_reg に格納するための優先度を決定する. 優先度は基本的にはラウンドロビンによる優先度の決定を行うが,SDRAM の各バンクへのアクセスの履歴 (2bit) 情報を用いて各バンクへのリクエストが分散されるように優先度を決定する.

  同一バンクに割り当てられたリクエストに対してはそのまま a_reg に転送するため順序依存を破壊することは無い.

  MRA でのアクセス最適化は CPU,PCI から同時に発行されたリクエストに対してのアクセス最適化であるためこれだけでは不完全である. そのため MRA 以降の段階でのアクセス最適化は RCU が担当する.

- RCU

  RCU は内部のバンクステートマシンに空きがあるとき,a_reg から空きのあるバンクに対するリクエストを取得して SDRAM への処理を行う.a_reg に対応するバンクに対してのリクエストが存在しない場合は待ち状態となる.

  バンクステートマシンが複数空き状態であるときは RCU_arbiter により決定される優先度の高い上位 2 つのバンクマシンが a_reg を検索できる.

このように簡単ではあるが時間差で複数回のアクセス最適化を行うことでメモリバンクビジー率を向上できる. また動的にアクセス最適化はハードウェアで行う際, クリィティカルパスとなる部分であるが, これを各モジュールに分散することで動作周波数の向上を阻害しないよう考慮している.

5.2.7 SDRAM バスコマンド

SDRAM には次のようなバスコマンドがあり, コマンドと各バンクの状態遷移によりデータの読み書きを実現する.

Mode Register set command

(/CS,/RAS,/CAS,/WE = Low)

このコマンドは A0 - A10, BA0, BA1 の信号を入力とし, デバイスの動作モードを選択する.

SDRAM は電源投入後, 初期か動作を行った後このコマンドを必ず入力しなければならない. またモードレジスタへの値の入力は全てのバンクが idle 状態でなければならない.

Active command

(/CS,/RAS = Low, /CAS,/WE = High)

uPD4564323 では 4096 本の Row アドレスをもつ 4 つのバンクが存在する.

BA0,BA1 によりバンクを決定し,A0 - A10 で Row アドレスを決定する. このコマンドは汎用 DRAM の/RAS の立ち下がりに対応する.

Precharge command

(/CS,/RAS,/WE = Low,/CAS = High)

このコマンドは BA によって指定されたバンクのプリチャージ操作を開始する.

A10 が High の時は BA に関係なく全てのバンクに対してプリチャージ操作を開始する. A10 が Low の時は BA によって指定されたバンクのみに対してプリチャージ操作を行う.

このコマンドが入力された後, t_{RP}(precharge to active command period) の間 active コマンドの入力を受け付けない.

このコマンドは汎用 DRAM の/RAS の立上りに対応する.

Write command

(/CS,/CAS,/WE = Low,/RAS = High)

メモリへの書き込みを行う. もしモードレジスタによってバーストライトモードであるのならば, コマンドと同時に入力されたカラムアドレスを開始アドレスとして SDRAM 内部でアドレス計算を行う.

書き込む最初のデータはコマンドの入力と同時に行う.

Read command

(/CS,/CAS = Low,/RAS,/WE = High)

読みだしデータは/CAS レイテンシ後,DQ 線にドライブされる. バーストモードの場合は要求するデータの開始アドレスをコマンドと同時に入力する.

SDRAM 内部のアドレス計算方法はモードレジスタに従う.

CBR(auto) refresh command

(/CS,/RAS,/CAS = Low,/WE,CKE = High)

このコマンドの入力によって CBR(auto)refresh 操作が開始される. リフレッシュされるアドレスは SDRAM 内部で生成される. CBR(auto)refresh コマンドを実行するためには全てのバンクがプリチャージされていなければならない. そしてこのコマンドが実行されると全てのバンクが idle 状態に遷移し active コマンドの入力待ちとなる. CBR(auto)refresh コマンド入力後,t_{RC}(from refresh command to refresh or activate command) の間, 他の全てのコマンドは受け付けられない.

Self refresh entry command

(/CS,/RAS,/CAS,CKE = Low,/WE = High)

このコマンド入力後,CKE が Low の間,self refresh コマンドは継続される. セルフリフレッシュモードの間はリフレッシュ間隔, リフレッシュ動作を内部で行うため, 外部から特別な操作をする必要はない.

このコマンドを入力する前に, 全てのバンクをプリチャージしなければならない.

Burst stop command

(/CS,/WE = Low,/RAS,/CAS = High)

このコマンドは現在行われているバーストサイクルを終了する操作である. このコマンドによってバンクがプリチャージされることはない. このコマンド実行後,Row active 状態へ遷移する.

No operration

(/CS = Low,/RAS,/CAS,/WE = High)

このコマンドは何も実行されない.

5.2.8 SDRAM バスコマンド一覧

SDRAM バスコマンドの一覧を表 5.11 に示す.

(L = Low,H = High,V = Valid,x = Don't care)

また SDRAM はメモリの省電力化も考慮して設計されており, 次のような機能も実現されている.(表 5.12)

(L = Low,H = High,V = Valid,x = Don't care)

5.2.9 モードレジスタ

モードレジスタは SDRAM をどのように動作させるかを設定するレジスタである. 設定項目のほとんどはバースト転送に関係するものである.

モードレジスタは図 5.32 のような構成をしている.

図の WT は Wrap Type と呼びバースト転送時のアドレッシングを設定する. シーケンシャルモードとインターリーブモードがありそれぞれのアドレス

表 5.11: SDRAM バスコマンド表

Function	Symbol	CKE(n-1)	/CS	/RAS	/CAS	/WE	BA(0,1)	A10	A9 - A0
Device deselect	DESL	H	H	x	x	x	x	x	x
No operation	NOP	H	L	H	H	H	x	x	x
Burst stop	BST	H	L	H	H	L	x	x	x
Read	READ	H	L	H	L	H	V	L	V
Read with auto precharge	READA	H	L	H	L	H	V	H	V
Write	WRIT	H	L	H	L	L	V	L	V
Write with auto precharge	WRITA	H	L	H	L	L	V	H	V
Bank active	ACT	H	L	L	H	H	V	V	V
Precharge select bank	PRE	H	L	L	H	L	V	L	x
Precharge all banks	PALL	H	L	L	H	L	x	H	x
Mode register set	MRS	H	L	L	L	L	V	V	V

表 5.12: クロック操作系バスコマンド一覧

Current state	Function	Symbol	CKE(n-1)	CKE	/CS	/RAS	/CAS	/WE	Address
Activating	Clock suspend entry		H	L	x	x	x	x	x
Any	Clock suspend		L	L	x	x	x	x	x
Clock suspend	Clock suspend exit		L	H	x	x	x	x	x
Idle	CBR(auto) refresh	REF	H	H	L	L	L	H	x
Idle	Self refresh entry	SELF	H	L	L	L	L	H	x
Self refresh	Self refresh exit		L	H	H	x	x	x	x
Idle	Power down entry		H	L	x	x	x	x	x
Power down	Power down exit		L	H	H	x	x	x	x

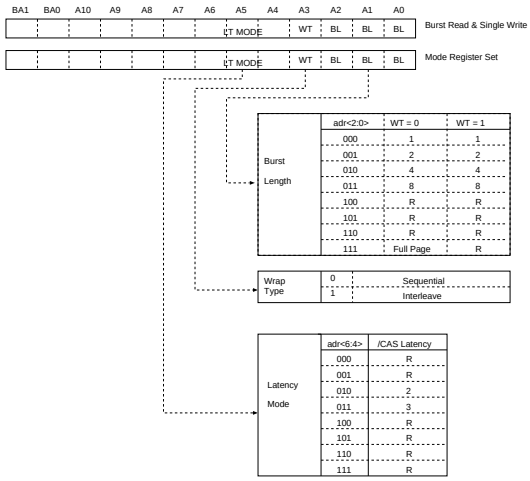


図 5.32: モードレジスタ

の計算は表 5.13 のように行う.(例としてバースト長 8)

図の LTMODE は CAS レイテンシの値を指定する.CAS レイテンシの値は uPD4564323 の場合 2,3 が選択できる.

図の BL はバースト長であり,1,2,4,8,Full Page が設定できる. Full Page は製品によって異なるが,uPD4564323 では 256 となっている.

また Write back キャッシュを考慮して Burst Read & Single Write モードというものも選択可能である.

5.2.10 SDRAM READ/READA コマンド

SDRAM の基本的な Read サイクルについてタイミングチャートを図 5.33,5.34 に示す.

READA はバンクプリチャージを自動的に行うコマンドでプリチャージを自動的に発行できるためコマンド線の効率が良く, その間に他のバンクを操作できる. しかし同一の Row アドレスに対して再度アクティブにする必要があるためそのような場合にはパフォーマンスが低下する.

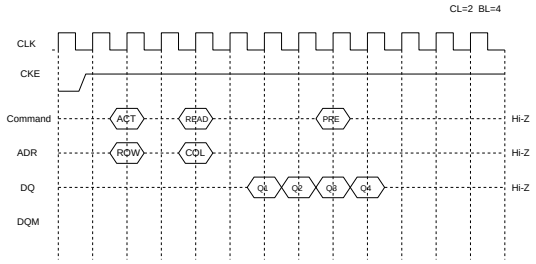


Fig.3 Precharge Timing

図 5.33: READ タイミングチャート

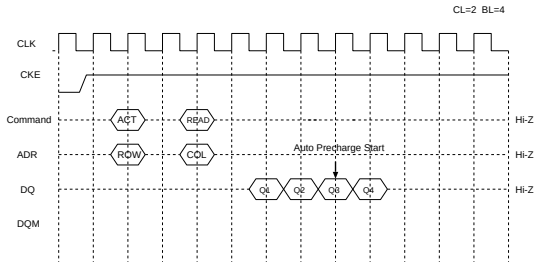


Fig.4 Auto Precharge Timing

図 5.34: READA タイミングチャート

5.2.11 SDRAM WRIT/WRITA コマンド

SDRAM の基本的な Write サイクルについてのタイミングチャートを図 5.35,5.36 に示す.

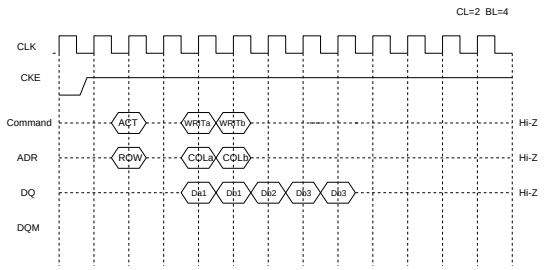


Fig.7 Write to Write Command Interval Timing

図 5.35: WRIT タイミングチャート

WRITA に関しても READA と同様プリチャージコマンドが不要であるというメリットがある. デメリットに関しても同様である.

表 5.13: Wrap Type

Starting address (Address A2 - A0)	Sequential addressing sequence	Interleave addressing sequence
000	0, 1, 2, 3, 4, 5, 6, 7	0, 1, 2, 3, 4, 5, 6, 7
001	1, 2, 3, 4, 5, 6, 7, 0	1, 0, 3, 2, 5, 4, 7, 6
010	2, 3, 4, 5, 6, 7, 0, 1	2, 3, 0, 1, 6, 7, 4, 5
011	3, 4, 5, 6, 7, 0, 1, 2	3, 2, 1, 0, 7, 6, 5, 4
100	4, 5, 6, 7, 0, 1, 2, 3	4, 5, 6, 7, 0, 1, 2, 3
101	5, 6, 7, 0. 1, 2, 3, 4	5, 4, 7, 6, 1, 0, 3, 2
110	6, 7, 0, 1, 2, 3, 4, 5	6, 7, 4, 5, 2, 3, 0, 1
111	7, 0, 1, 2, 3, 4, 5, 6	7, 6, 5, 4, 3, 2, 1, 0

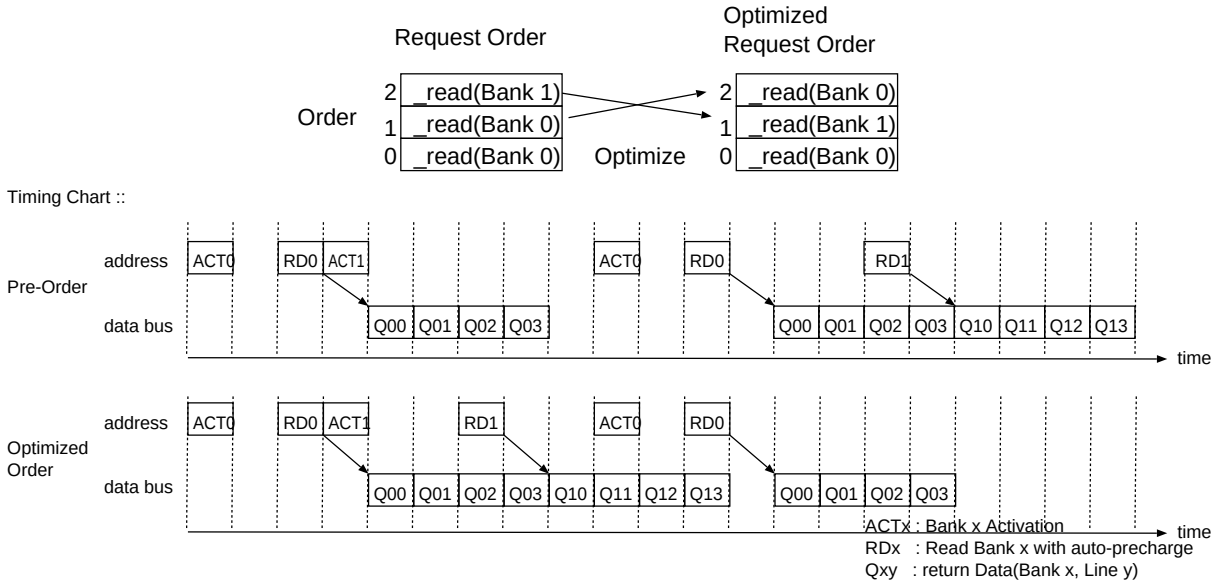


図 5.37: SSC により最適化されたメモリアクセス

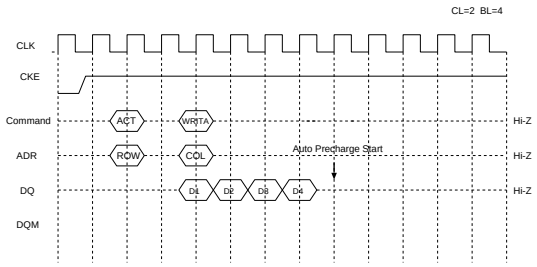


Fig.5 Write with Auto Precharge Timing

図 5.36: WRITA タイミングチャート

5.2.12 最適化された SDRAM アクセス

SSC によって最適化された SDRAM へのアクセスを図 5.37 に示す.

このように同一バンクへのメモリリクエストが連続した場合においても, 最適化キュー内でその他のバンクへのアクセス要求を追い越すことで,SDRAM で性能のオーバーヘッドとなるバンクアクティベーション, バンクプリチャージ時間を隠蔽し,SDRAM からの高いスループットを実現している.

第6章　検討

コンピュータシステムとしての構築を行うに必要であろうと思われる, ハードウェア構成, オペレーティングシステムとプロセッサ間のインターフェース等似ついて検討する.

6.1　Write Back 型キャッシュ

現在のシステムモデルにおけるキャッシュメモリ,SCALT バッファは同階層に位置し, キャッシュメモリに関しては Write Through 型キャッシュを用いているため, キャッシュ, バッファ間のコンシステンシを考慮せずにシステム構築が行われているが, 図 6.1 のように多階層で Write Back 型のキャッシュメモリが配置された場合, キャッシュ－メモリ間での書き戻しタイミングが予測できない.

この場合に最後にキャッシュメモリ経由でアクセスしたメモリが書き戻されずに Buffer Fetch / Buffer Store が発行されると予期した動作を行うことができない.

この問題に対応するためには Buffer Fetch 命令発行の際に Write Back 型キャッシュに対してパージを行うか,SCALT バッファを利用する時点で Write Back キャッシュを書き戻す操作を行う必要がある.

6.2　排他制御操作

オペレーティングシステムは複数のプロセス間でメモリを共有する際, それぞれのプロセス間で排他的にメモリアクセスできるようにソフトウェアロックを用意する必要がある. ソフトウエアロックにはセマフォ等が挙げられるが, このセマフォを獲得するためにはハードウェアの Lock / Unlock 命令が必要である.

SPK プロセッサでは Lock / Unlock を LDx_L,STx_C という命令で行う. LDx_L 命令はプロセッサ内部の Lock フラグ,Lock アドレスを設定し, システムコン

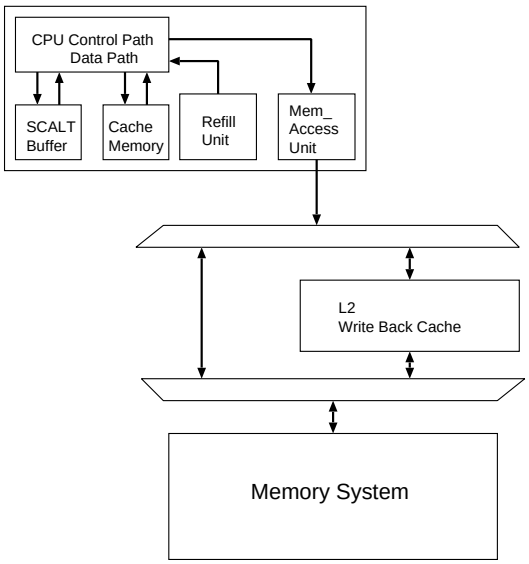


図 6.1: Write Back 型キャッシュが存在する場合

```
try_again: LDQ_L R1, x
           <modify R1>
           STQ_C R1, x
           BEQ   R1, try_again
```

図 6.2: クリティカルセクションにおける操作

トローラに対しても Lock 信号を送る. この Lock フラグはかなり弱く,STx_C 命令以外にも割り込みやモード変更が行われた際にも解除される.

次に STx_C 命令であるが, これは LDx_L 命令によってセットされた Lock フラグをレジスタに格納する命令である. この Lock / Unlock の操作例を図 6.2 に示す.

LDx_L, STx_C のプロセッサパイプラインにおける流れは図 6.3 のようになる.

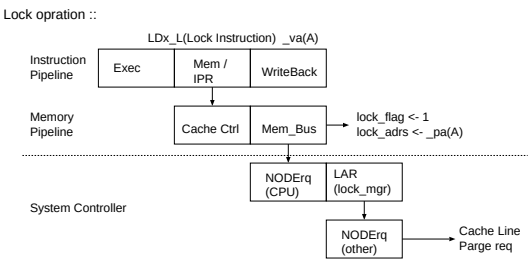


図 6.3: 排他制御操作のデータフロー

6.3 SCALT バッファのオペレーティングシステムへの対応

SCALT バッファをユーザが利用する際, 仮想アドレス上にバッファをマッピングするという特徴からオペレーティングシステムはバッファのアドレス, サイズ, 利用状況を把握する必要がある. また, ユーザーオペレーティングシステム間でのインターフェースであるシステムコールを用意する必要がある.

SPK プロセッサではプロセッサ内部レジスタにバッファ領域の先頭アドレス (SB_BASE), サイズ (SB_SIZE) が格納されているため, これらを用いてリソース管理を行う.

システムコールについては malloc,free と操作的に相似点があるため, これと同様のインターフェースがよいと考えている. ただし,Buffer Fetch / Buffer Store 共にバッファエントリ長を知る必要があるため, ソフトウェア側がエントリ長を知る必要が有り, ヘッダファイル等も必要となる. SCALT バッファを用いたプログラムは図 6.4 のようになる.

6.3.1 互換性

同一プロセッサファミリ間で SCALT バッファの容量変更した場合においてもプログラムの互換性がなければならない. システム内での SCALT バッファの容量が変化した場合においても図 6.4 のように SCALT バッファが利用できる場合, できない場合それぞれの処理を記述することで問題無く処理できる. これは malloc を用いたプログラムの記述と同様である.

前にも述べた通り, システム間でバッファエントリ長が異なるような場合もある, これはシステムによって利用分野がへんかするためであり, システムごとに隠蔽すべきメモリレイテンシが変わることも原因のひとつとしてある. SCALT を用いたシステムではそれぞれのシステムレイテンシ特性に特化した形でデータ転送ライブラリを BF 命令の組合せで作成することで, ユーザがエントリ長を特に意識すること無く SCALT バッファが利用できるようになるとも考えられる.

```
 #define ENTRY 32
 #define ELEMENT (ENTRY / sizeof(double))
 double x[N][N], y[N][N]; /* メインメモリ上のメモリ領域 */
 double *sbuf;            /* SCALT バッファへのポインタ */

 /* SCALT バッファの利用要求 */
 sbuf = (double *)scalt_alloc(ENTRY * 1024);
 if (sbuf != NULL) {
      /* バッファを利用する場合の処理 */
      scalt_free(sbuf); /* SCALT バッファの解放 */
 } else {
      /* バッファを利用できない場合の処理 */
 }
 return;
}
```

図 6.4: SCALT バッファを利用するプログラム

第7章　結論

現在, 広く用いられているレイテンシ隠蔽技術について述べ, それらの技術では補うことのできない問題の提示を行い, レイテンシ隠蔽アーキテクチャSCALT を提案した. SCALT を適用したシミュレータによる, 性能評価を行い,PARTHENON を用いて設計を行った.

近年のプロセッサ技術は高速化, 並列化が主流となっており, 要求演算量が増大していることがわかる. しかし, プログラムの大型化する傾向に有る現在, 演算量の増加はメモリ参照量の増加とも考えられ, メモリの高性能化はますます重要な技術である. SCALT を用いた, 広域参照型の先行的なメモリアクセス, ポインタアプリケーションへのレイテンシ隠蔽技術の対応はこれからのコンピュータシステムにおいて重要な問題である.

また, メモリ性能の高速化技術はハードウェアだけではなく, ソフトウェアにおいても積極的に研究されている. 特に, コンパイラにおいて積極的に SCALT を利用できるような技術を研究することは大変重要である. コンパイラによる Buffer Fetch(Prefetch) の自動的な挿入が可能となれば,SCALT を容易に利用でき, 有用である.

5,6 章ではプロセッサおよびシステムコントローラの設計について述べたが, 設計については完全な状態ではないため, 今後のシミュレーション等で構成を変更する必要があると考えられる.

謝辞

研究の過程に於いて指導教員である清水尚彦 助教授の熱心な御指導頂きました. 私自身後からが及ばず, 御指導に沿えない部分が多々あり, 御力をお借りするばかりでしたが, その中で得るものが多数あり清水尚彦助教授には大変感謝をしております.

また, 技術的, 精神的に支援して頂いた清水研究室の三竹氏, 津田氏, 宮坂氏, 近氏, 李氏, 神山氏, 早坂氏ならびに同研究室の諸学生の皆様, また, 他研究室の友人に大変感謝しております. 最後に, 本研究へ貴重なアドバイスを頂いた東海大学総合情報センター湘南計算機室, 伊勢原計算機室の職員の皆様にお礼を申し上げます.

業績

1. 孕石, 宮坂, 清水, **レイテンシ耐性を有するコンピュータシステムの研究**, **電子情報通信学会全国大会**, *2001*

2. **孕石**, **宮坂**, **清水**, レイテンシ耐性を有するコンピュータシステムの研究, 第 18 回パルテノン研究会, 2001

3. 孕石, 宮坂, 清水, **レイテンシ耐性を有するコンピュータシステムの研究**, *SWoPP* **沖縄**, *2001*

4. **孕石**, **宮坂**, **清水**, メモリレイテンシ耐性を有するコンピュータシステムの研究, デザインガイア 2001, 2001

5. 宮坂, 孕石, 清水, **メモリレイテンシ隠蔽アーキテクチャ***SCALT,* **デザインガイア** *2000, 2000*

6. **清水**, **宮坂**, **孕石**, Design of A Memory Latency Tolerant Processor(SCALT), MEDEA Workshop, 2001

7. 津田, 宮坂, 近, 孕石, 渡辺, **第** *6* **回** *ASIC* **デザインコンテスト** *(16 bit CPU),* **第** *16* **回 パルテノン研究会**, *2000*

8. **近**, **孕石**, CPLD ベンダライブラリサポートツール, 第 17 回 パルテノン研究会 , 2000

9. 孕石, 早坂, **第** *7* **回** *ASIC* **デザインコンテスト 規定課題** *B PCI* **バスインターフェース** *(K_tel),* **第** *18* **回 パルテノン研究会**, *2001*

10. **早坂**, **孕石**, 第 8 回 ASIC デザインコンテスト 規定課題 B PCI バスインターフェース (n_tele), 第 20 回 パルテノン研究会, 2002

11. 孕石, 早坂, *PARTHENON* **による** *PCI* **バスインターフェースの開発**, **第** *9* **回 パルテノン講習会**, *2001*

*12.***早坂**, **孕石**, **清水**, アマチュア電子工作に適したオープンな PCI インターフェースの設計ならびに仕様について, パルテノン研究会, 2002

13. H.Hayasaka,H.Haramiishi, N.Shimizu, *The Design of PCI Bus Interface, 6D-11, ASP-DAC2003, 2003*

14. **早坂**, **石川** *(***隆***)*, **石川** *(***慎***)*, **簗場**, **孕石**, **清水**, PARTHENON を用いたコンピュータシステムの設計及び Linux システムの実現, 第 20 回 パルテノン研究会, 2002

参考文献

[1] 清水尚彦, スケーラブル レイテンシトレラント アーキテクチャ, IPSJ Sig Notess Vol.97 No.21, 1997

[2] 三竹大輔 清水尚彦, 変動するメモリレイテンシに対応するプロセッサ, 東海大学紀要 (工学部), Vol.39 No.1, 1999

[3] 三竹大輔, レイテンシ耐性を有するプロセッサアーキテクチャの研究, 1999 年度 東海大学大学院 修士論文

[4] 宮坂和幸, メモリレイテンシを隠蔽する専用バッファ及び非同期転送命令を実装した RISC プロセッサの実現, 2001 年度 東海大学大学院 修士論文

[5] David Kroft, *Lookup-free Instruction Fetch/Prefetch Cache Organization, IEEE, 1981*

[6] **大河原 英喜**, **中村 宏**, **芳江 友照**, **金谷 和至**, **ハイパフォーマンスコンピューティングに適したメモリ階層の検討**, **情報処理学会** *ARC133-10, 1999*

[7] **近藤 正章**, **坂井 修一**, **朴 泰祐**, **中村 宏**, *HPC* **向けプロセッサのメモリアーキテクチャの基本構成**, **情報処理学会** *ARC134-1, 1999*

[8] **大河原 英喜**, **近藤 正章**, **中村 宏**, **朴 泰祐**, **ハイパフォーマンスに適したメモリアーキテクチャの予備評価**, **情報処理学会** *ARC136-3, 2000*

[9] **中村 宏**, **近藤 正章**, **大河原 英喜**, **朴 泰祐**, **ハイパフォーマンスコンピューティング向けアーキテクチャ** *SCIMA,* **情報処理学会論文誌** *Vol.41, No.SIG5(HPS1), pp.15-27, 2000*

[10] **近藤 正章**, **中村 宏**, **朴 泰祐**, *SCIMA* **における性能最適化手法の検討**, **情報処理学会論文誌** *Vol.42, No.SIG12(HPS4), pp.37-48, 2001*

[11] M. Lam, E. Rothberg, M.Wolf, The cache performance and optimizations of blocked algorithms, Proc. of ASPLOS-IV, pp.64-74, 1991

[12] P. Panda, H. Nakamura, N. Dutt, A. Nicolau, *Augmenting loop tiling with data alignment for improved cache performance, IEEE Trans. on Computers, Vol.48 No.2 pp.142-149, 1999*

[13] J.L. Baer etc. An Effective on-chip preloading scheme to reduce data access penalty, In Proceedings of Supercomputing '91, 1991

[14] Nakazawa, Nakamura, etc. *Pseudo Vector Processor based on Register-Windowed Superscaler Pipeline, Supercomputing '92, 1992*

[15] Todd C. Mowry, Tolerating Latency Through Software-Controlled Fata Prefetching,

[16] Chi-Keung Luk, Todd C. Mowry *Automatic Compiler-Inserted Prefetching for Pointer-Based Applications , IEEE TRANSACTIONS ON COMPUTER Vol. 48 No.2, Feb. 1999*

[17] Bharat Chandramouli, John B. Carter, Wilson C. Hsieh, Sally A. McKee, A Cost Framework for Evaluating Intergrated Restructuring Optimizations, IEEE Int'l Conference on Parallel Architecture and Compilation Techiniques, 2001

[18] Sally A. McKee, Wm. A, Wulf, *A Memory Controller for Improved Performance of Streamed Computations on Symmetric Multi-Processor, IPPS'96, 1996*

[19] Trishul M. Chilimbi, Mark D. Hill, James R. Larus, Cache-Conscious Structure Layout, Proceedings of ACM SIGPLAN'99,Comference on Programming Language Design and Implementation, 1999

[20] Cray Inc., *Preformance of the Cray T3E Multiprocessor Cray Research*

[21] Richard L. SITES, Richard T. Witek, Alpha AXP Architecture Reffernce Manual(SECOND Edition), Digital Press, Digital Equipment Corp. , 1995

[22] DEC, *Alpha 21164 Microprocessor Data Sheet, Digital Equipment Corp, 1994*

[23] AMD, AMD-751 : System Controller Data Sheet, AMD, 2000

[24] Elpida Memory Inc. , SDRAM の使い方

[25] PARTHENON, *http://www.kecl.ntt.co.jp/parthenon, PARTHENON* **研究会***, NTT*

[26] John L. Hennessy, David A. Patterson, Computer Archtecture: A Quantitative Approach MORGAN KAUFMANN

[27] 曽和 将容, コンピュータアーキテクチャ原理, コロナ社, 1993

[28] Harris. Tohamm, *The Scalability of Decoupled Multiprocessor, SHPCC'94, 1994*

[29] McCalpin, Sustainable Memory Bandwidth in High Performance Computers, http://www.cs.virginia.edu/stream/

[30] John D. McCalpin, *A Survey of Memory Bandwidth and Machine Balance in Current Hight Performance Computers, IEEE TCCA Newsletter, 1995*

[31] Trishul M. Chilimbi, Cache-Conscious Pointer Structures, Dagstuhl Semminar on Programs

[32] *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines, PhD thesis, Princeton University Department of Computer Science, 1996*

[33] Marchin C. Calisle, Anne Rogers, Supporting Dynamic Data Structures on Distributed Memory Machines, CS-TR-447-94, ACM Transactions on Programming Languages Vol. 17 No.2, 1995

[34] Thomas Alexander, Gershon Kedem, *Distributed Prefetch-Buffer/Cache Design for High Performance Memory Systems, Duke University*

[35] Kevin Dowd **著***,* **久良知 真子 訳***,* High Performance Computing, トムソンパブリッシング ジャパン, O'Reilly & Accociates, Inc.