修 士 論 文

面積効率を指向するプロセッサ「雷上動」の設計と実装

2014年2月6日

指導教官

坂井 修一 教授

情報理工学系研究科 電子情報学専攻 修士2年

学籍番号：48-126437

藤田 晃史

目 次

図 目 次

表 目 次

概要

我々の研究グループでは，プロセッサの面積効率を向上させるアーキテクチャ技術をいくつか提案してきた．これらは out-of-order スーパースカラ・プロセッサの性能を下げずに，回路面積を削減する技術である．具体的には，マトリクス・スケジューラ，分離リオーダ・バッファなどがあげられる．しかし，ハードウェアとしての動作が未検証な技術がある，異なる技術を組み合わせると不整合が発生する可能性がある，という懸念が存在した．面積効率の高いプロセッサを実現するうえでの問題を明らかにするため，これらの技術を組み合わせたプロセッサを開発したいと我々は考えていた．

本論文では，面積効率を向上させる技術を採用した out-of-order スーパースカラ・プロセッサ「雷上動」の設計と実装について述べる．雷上動は FPGA 上で動作する段階まで開発が進んでいる．FPGA の資源消費量を評価した結果，面積効率を向上させる技術を用いることで使用 LUT 数を抑えられることが確認できた．

第1章

序論

本研究の背景と目的

高性能な out-of-order スーパースカラ・プロセッサの面積は非常に大きい．Out-of-order スーパースカラ・プロセッサでは，命令レベル並列処理によって高い性能を実現するが，それを実現する回路には多数のマルチポートメモリが含まれる．より性能を向上させるために，各部分の処理幅や同時に扱う命令数を大きくしようとすると，マルチポートメモリのポート数・容量も大きくなり回路面積が増大する．回路面積の増大は，近年問題視されている消費電力の増加につながる．また，マルチコアの時代においては，1 つのコアの面積が大きいと搭載できるコア数が少なくなり，システム全体の性能が制限されてしまう．

そこで，我々の研究グループでは，高い命令レベル並列処理性能を保ちつつ回路面積を削減するアーキテクチャ技術をいくつか提案してきた．ウェイクアップ・ロジックの面積を削減するマトリクス・スケジューラ [1,2]，リオーダ・バッファの面積を削減する分離リオーダ・バッファ[3]，レジスタ・ファイルの面積を削減する非レイテンシ指向 レジスタ・キャッシュ・システム [4,5] などである．我々は，高性能と小面積を両立させるこれらの技術を総称して，面積効率を向上させる技術と呼んでいる．

我々は，面積効率を向上させる技術の有用性を実証するため，これらの技術を統合したプロセッサを作りたいと考えていた．実際に動作するプロセッサを設計・実装すれば，技術に存在する欠陥や技術同士の組み合わせで生じる不整合を洗い出すことができる．また，実際に面積効率の高いプロセッサが完成すれば，我々の提案してきた技術の実用性をより強くアピールできる．

そこで本研究では，面積効率を向上させる技術を採用したプロセッサ「雷上動」

の設計と実装を行った．雷上動は 32bit ARM 命令セットをサポートする out-of-order スーパースカラ・プロセッサである．Out-of-order スーパースカラ・プロセッサとして基本と思われる機能はすべて実装したうえで，面積効率を向上させる技術を適用している．実装は，ハードウェア記述言語 SystemVerilog の記述などにより行っている．

雷上動は現在，FPGA 上では動作するがチップとしての開発は進んでいない，という段階である．そこで本論文では，回路面積の代わりに FPGA の資源消費量を用いて評価を行う．面積効率を向上させる技術を用いれば，命令レベル並列処理性能を低下させずに FPGA の資源消費量を削減できると考えられる．

本稿の構成

次章以降の構成は以下のようになっている．2 章，3 章では雷上動を理解するのに必要な予備知識を説明する．2 章では out-of-order スーパースカラ・プロセッサの基礎について，3 章では面積効率を向上させるアーキテクチャ技術について述べる．4 章，5 章では本稿の主題である雷上動について記す．4 章では設計と実装について，5 章では FPGA で行った評価について述べる．6 章では関連研究を紹介する．最後に、7 章で結論を述べる．

第2章

Out-of-order スーパースカラ・プロセッサの基礎

本章では，雷上動および面積効率を向上させる技術を理解するのに必要な，out-of-order スーパースカラ・プロセッサの基礎を説明する．

2.1 Out-of-order スーパースカラ・プロセッサのアーキテクチャ

Out-of-order スーパースカラ・プロセッサの基本的な構造を図 2.1 に示す．Out-of-order スーパースカラ・プロセッサの全体はフロントエンドとバックエンドに大きく分かれており，両者は命令ウィンドウと呼ばれるバッファにより緩く結合している．フロントエンドは命令の供給や解析を行う部分で，ここでの処理は in-order に行われる．フロントエンドでの処理が終わった命令は命令ウィンドウに書き込まれ，依存する命令を待ってから out-of-order にバックエンドに送られる．バックエンドでは演算を行い，結果をレジスタ・ファイルに書き込む．

Out-of-order スーパースカラ・プロセッサは，レジスタ・ファイルをフロントエンド/バックエンドのどちらで読み出すかで大きく 2 種類に分けることができる．雷上動はレジスタ・ファイルをバックエンドで読み出す方式を採用するので，以後それを前提として説明を行う．

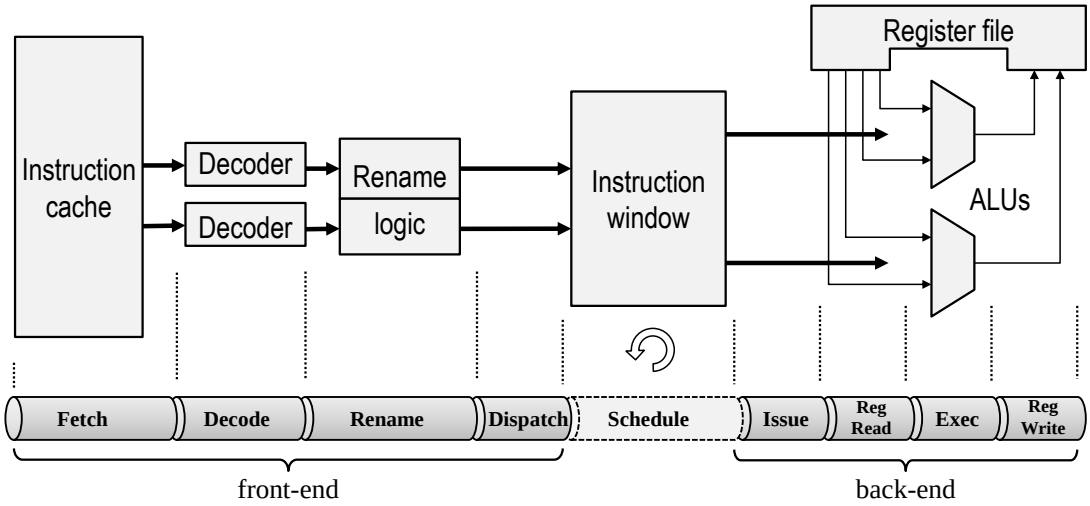

図 2.1: Out-of-order スーパースカラ・プロセッサ の基本構成

2.1.1 フロントエンド

フロントエンドは以下のフェーズに分かれている．

1. フェッチ
2. デコード
3. リネーム
4. ディスパッチ

それぞれについて簡単に説明する．

フェッチ

フェッチ・フェーズでは PC (Program Counter) の値に従って命令キャッシュにアクセスし，命令を読み出す．分岐命令がある場合は分岐予測器によって分岐方向や分岐先を予測し，それに応じて PC を更新する．

デコード

デコード・フェーズでは命令を解析し，命令の種類やオペランドといった情報を得る．Intel x86 のような複雑な命令セットを持つアーキテクチャでは，命令をマイクロ命令に分解する処理を行う [6]．

リネーム

リネーム・フェーズでは，命令間の偽の依存を取り除くためにレジスタ・リネーミングを行う．レジスタ・リネーミングでは，命令セットで定義された論理レジスタ番号から，プロセッサ内部の物理レジスタ番号へ変換を行う．この変換を行うために論理レジスタ番号と物理レジスタ番号の対応表が存在し，それを RMT (Register Map Table) という．

ディスパッチ

ディスパッチ・フェーズでは，命令を命令ウィンドウに書き込む．ディスパッチは，フロントエンドの最後の処理となる．

2.1.2 バックエンド

バックエンドは以下のフェーズに分かれている．

1. スケジューリング
2. 発行
3. レジスタ読み出し
4. 演算実行
5. レジスタ書き込み

それぞれについて簡単に説明する．

スケジューリング

スケジューリング・フェーズでは，バックエンドにおける命令の実行タイミングを決定する．あるタイミングで命令を実行するには，以下の 2 つの条件が満たされている必要がある．

- 依存関係が解決され，オペランドが利用可能であること．
- 演算器などのハードウェア資源が利用可能であること．

前者が満たされた状態を，実行可能と呼ぶ．オペランドが利用可能であること，ならびに命令が実行可能であることは，ともにレディと呼ばれる．レディとなった命令は，後者の条件が満たされるタイミングで実行される．

依存関係が解決されて実行可能となった命令を”起こす”処理をウェイクアップという．依存関係のうち，偽の依存についてはリネーム・フェーズで解決しているので，ここでは真の依存についてのみ考えればよい．つまり，デスティネーション・オペランドが A である命令が実行された時に，ソース・オペランドが A である命令がウェイクアップされる．ただし，ソース・オペランドを複数持つ命令は，全てのソース・オペランドが利用可能となった時点で実行可能となる．

ウェイクアップされて実行可能となった命令の中から，次に実行する命令を選択する動作をセレクトという．セレクトが必要になるのは，演算器などのハードウェア資源に限りがあるためである．例えば，演算器が 2 つしかないのに 3 命令が実行可能であった場合は，次に実行する 2 命令を選ぶ必要がある．セレクトの結果はウェイクアップ・ロジックに伝えられ，セレクトされた命令に依存する命令をウェイクアップするために用いられる．

スケジューリング・フェーズでは，ウェイクアップとセレクトを交互に繰り返すことで命令の実行タイミングを決定する．ウェイクアップとセレクトは互いの結果を利用するので，スケジューリングを行う回路はウェイクアップ・ロジックとセレクト・ロジックがフィードバック・ループを形成する構造になっている．

発行

セレクト・フェーズで選択された命令は，続く発行フェーズにおいて命令ウィンドウから読み出される．命令を命令ウィンドウから読み出す動作を発行という．発行フェーズで読み出した命令の情報は，以後命令ウィンドウに保持しておく必要が無いものも多い．そこで，ディスパッチ・フェーズから発行フェーズまでのみ命令を保持するハードウェアがあり，それを発行キューという．

演算実行

演算実行フェーズでは，ALU やシフタなどの演算器で演算を行う．

レジスタ・アクセス

レジスタ読み出しフェーズおよびレジスタ書き込みフェーズでは，物理レジスタ・ファイルにアクセスを行う．なお，物理レジスタ・ファイルの読み書きには通常 1 サイクル以上の時間がかかる．そこで，物理レジスタ・ファイルへの書き込みを待たずに先行命令の結果を利用できるようにして，命令の実効レイテンシを削減するのが普通である．これを実現する回路をバイパス・ネットワークという．

2.1.3 正確な例外

逐次実行における正確なプロセッサ状態を確保するような例外を，正確な例外という．つまり，ある命令が例外を起こした時に，例外を起こした命令まではすべて実行が終了し，それ以降の命令は実行されていないという状態を作り出せるときに，その例外は正確であるという．正確でない例外が発生した場合は正しいプログラムの実行が継続できなくなるが，分岐予測ミスなどもここでいう例外の一種にあたるため，正確な例外をサポートすることは必須となる．

Out-of-order スーパースカラ・プロセッサでは正確な例外を実現するために，命令を実行した後も特定の時点までキャンセルすることを可能としている．命令を out-of-order に実行すると，例外を発生させる命令よりもその後続の命令が先に実行されることがある．正確な命令を実現するには，例外が発生した時点で後続の命令をキャンセルする機能が必要である．

プロセッサ状態を非可逆的に更新することをコミットといい，命令はコミットされた時点でキャンセルできなくなる．コミットは in-order に行うものとし，例外の検出はコミット時に行う．こうすると，例外を検出した時点では後続の命令がコミットされておらずキャンセル可能であることが保証される．

例外を検出した際に，後続の命令をキャンセルしてプロセッサの状態を巻き戻すことをリカバリという．リカバリ時には，RMT をキャンセルされた命令をリネームする前の状態に戻すなど，プロセッサ内のハードウェアを復元する処理を行う．この処理が終了したら，例外の復帰先の PC から命令のフェッチを再開する．

リカバリに関する情報は，アクティブ・リストというハードウェアにおいて保持される．具体的には，例外の有無や復帰先の PC，および RMT の復元に必要なレジスタ番号などである．一般的に，アクティブ・リストは命令のリネーム時からコミット時までこれらの情報を保持する．命令がコミットされるとアクティブ・リストのエントリが解放され，リカバリに関する情報が失われるため，命令をキャンセルできなくなる．

なお，アクティブ・リストに相当するハードウェアはリオーダ・バッファと呼ばれることもあるが，雷上動では一貫してアクティブ・リストという呼称を用いる．ただし，面積効率を向上させる技術のひとつである分離リオーダ・バッファについては，手法の名称自体に含まれているためリオーダ・バッファという語を用いる．

2.1.4 ロード/ストア命令の out-of-order 実行

ロード/ストア命令には，メモリのデータ依存という他には無い依存関係がある．すなわち，同じメモリ・アドレスにアクセスするロード/ストア命令には依存関係があり，ロード命令は先行するストア命令の実行結果が得られるのを待たねばならないというものである．

これに関するトピックとして，ロード/ストア命令のスケジューリングとストア–ロード・フォワーディングをとりあげる．

ロード/ストア命令のスケジューリング

ロード/ストア命令は，実行時に得られるメモリ・アドレスによって依存関係が決まるので，スケジュール・フェーズの時点では依存関係が曖昧である．保守的な実装では，ロード命令はプログラム・オーダ上で先行するストア命令の実行を待たないとメモリのデータ依存を無視してしまう可能性があるので，ロード命令が先行ストア命令を追い越すようなスケジューリングは行わない．しかし，2 つのロード/ストア命令のメモリ・アドレスは異なる場合が大半である．そのため，保守的なスケジューリングでは out-of-order 実行可能な命令を in-order に実行するケースが増えてしまう．

そこで，ロード/ストア命令を投機的に out-of-order 実行し，あとからメモリのデータ依存を検証するという手法がある [7]．この方法では，ストア・セット [8] のようなメモリ依存予測器によってメモリのデータ依存を予測し，それに従ってスケジューリングを行う．実行時にメモリ・アドレスを計算したあと，他のロード/ストア命令とメモリ・アドレスを比較するなどして，メモリのデータ依存を無視した実行が行われていないか検証を行う．予測ミスにより依存を無視した実行を行っていた場合は，メモリアクセス順序違反という例外を発生させる．

ストア–ロード・フォワーディング

一般に，データ・キャッシュ書き込みは非可逆的な動作なので，正確な例外を実現するためにストアのデータ・キャッシュ書き込みはコミット後に行う必要がある．よって，ストア命令にメモリ経由で依存しているロード命令は，ストア命令のコミットを待たないとデータ・キャッシュから正しい値を得ることができない．しかし，ストア・データは既にプロセッサ内に存在するため，ストア命令のコミットを待たなくてもロードすべき値を読み込むことは可能である．

そこで，ロード命令がデータ・キャッシュではなく依存するストア命令のストア・データを直接読み込むことがあり，これをストア–ロード・フォワーディングという．ストア–ロード・フォワーディングを用いるとロードの実行タイミングを早めることができるため，プログラム実行速度の短縮につながる．

2.2 Out-of-order スーパースカラ・プロセッサの回路面積

Out-of-order スーパースカラ・プロセッサでは多数の命令を同時に扱うため，それらの情報を管理して制御を行うためのマルチポートメモリがいくつも存在する．例として，物理レジスタ・ファイルやアクティブ・リストなどが挙げられる．これらはパイプラインの特定のステージで読み書きを行うものが多く，その際は処理幅に比例する数のポートが必要となる．また，処理幅が増えると同時に扱う命令の数も増えるため，マルチポートメモリの容量もほぼ処理幅に比例する．

このようなマルチポートメモリの回路面積はポート数の 2 乗と容量に比例する [1]．そのため，マルチポートメモリの回路面積はおおむね処理幅の 3 乗に比例する．

よって，高い命令レベル並列処理性能を得るために処理幅を大きくすると，マルチポートメモリのポート数が増えて回路面積が劇的に増加してしまう．回路面積の増加は，消費電力の増大など多くの問題をもたらす．これを解決するために提案されてきたのが，次章で説明する面積効率を向上させる技術である．

第3章

面積効率を向上させる技術

本章では，out-of-order スーパースカラ・プロセッサの面積効率を向上させるアーキテクチャ技術である，マトリクス・スケジューラと分離リオーダ・バッファについて説明する．我々の研究グループでは他にも面積効率を向上させる技術を開発しているが，本論文で扱うのはその 2 つに限る．

3.1　マトリクス・スケジューラ

マトリクス・スケジューラ [1,2] を使うと，ウェイクアップ・ロジックの面積を削減できる．従来の CAM 式ウェイクアップ・ロジックでは，命令間の依存の表現にマルチポートの CAM を必要とする．そのポート数は発行幅に応じて増えるため，発行幅の大きいスーパースカラ・プロセッサではウェイクアップ・ロジックの面積が非常に大きくなってしまう．一方，マトリクス・スケジューラでは命令間の依存の表現にマトリクスを用いるが，それに必要なハードウェア量は発行幅に無関係である．以下では，2 つの手法の内容について説明する．

CAM 式ウェイクアップ・ロジック

2.1 節で述べたように，ウェイクアップ・ロジックの役目はデスティネーション・オペランドが A である命令がセレクトされた時にソース・オペランドが A である命令をウェイクアップすることであった．この動作を CAM の検索による実現するのが，CAM 式ウェイクアップ・ロジックである．

CAM 式ウェイクアップ・ロジックを用いるスケジューラの構成を図 3.1 に示す．ソース CAM とデスティネーション RAM にセレクト・ロジックが加わり，ウェ

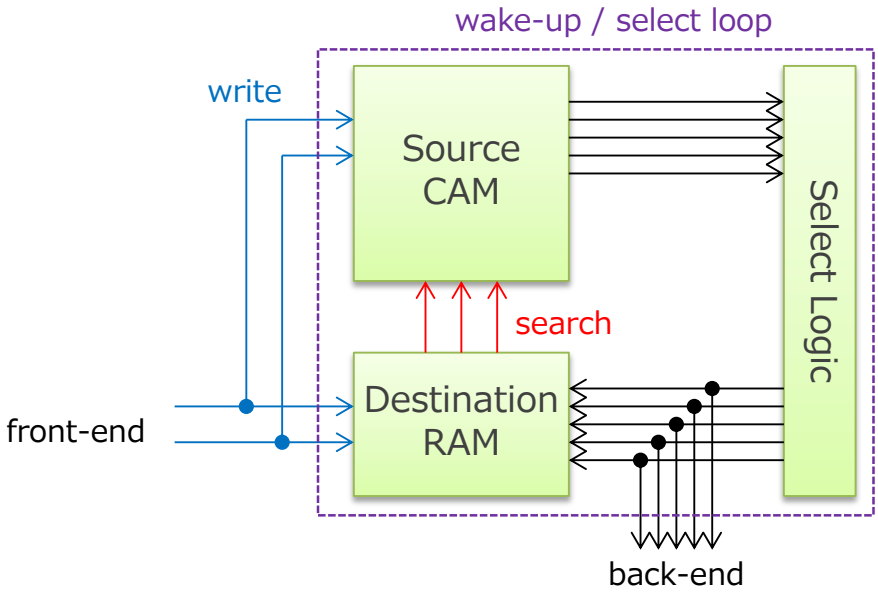

図 3.1: CAM 式ウェイクアップ・ロジックを用いたスケジューラの構成

イクアップ・セレクトのフィードバック・ループを形成している．

ウェイクアップ・ロジックの中心となるのはソース CAM で，ソース・レジスタ番号をキーとし，レディ・ビットを値とする．デスティネーション・レジスタ番号が A である命令がセレクトされたら，ソース・レジスタ番号が A であるエントリを検索してヒットしたエントリのレディ・ビットを立てる．すべてのオペランドのレディ・ビットが立ったら，その命令は実行可能となる．実行可能となった命令はセレクト・ロジックに入力され，その中から発行される命令を選択する．

デスティネーション RAM は，発行された命令のデスティネーション・レジスタ番号を読み出すために使われる．セレクト・ロジックからは発行された命令を示す信号が出力されるが，ソース CAM の入力はそのデスティネーション・レジスタ番号になっているので，デスティネーション RAM でその変換を行ってやる必要がある．

ソース CAM とデスティネーション RAM の読み出しポート数は発行幅に等しいため，発行幅の増大に応じて面積が増加することになる．近年のプロセッサは発行幅がどんどん大きくなる傾向にあるが，CAM 式ウェイクアップ・ロジックによってこれを支え続けるのは難しい．

マトリクス・スケジューラ

マトリクス・スケジューラの構成を図 3.2 に示す．プロデューサ・マトリクスとセレクト・ロジックにより，ウェイクアップ/セレクトのフィードバック・ループが形成されている．その前段に，WAT (Wake-up Allocation Table) というハードウェアが存在する．以下，これらの説明を行う．

マトリクス・スケジューラでは，レジスタ番号で表現されている依存関係を，命令同士の関係に変換して依存解決を行う．具体的には，依存元の命令の発行キューのエントリ番号を保持し，それによって命令間の依存を表現する．

そのため，レジスタ番号を発行キューのエントリ番号に変換するハードウェアが必要となり，これを WAT という．WAT はレジスタ番号をインデックスとし発行キューのエントリ番号を値とするテーブルである．各エントリはレジスタに最後に書き込んだ命令の発行キューのエントリ番号を保持する．命令はソース・オペランドをインデックスとして WAT を読み出し，依存元となる命令の発行キューのエントリ番号を得る．同時に，デスティネーション・オペランドをインデックスとして自身の発行キューのエントリ番号を WAT に書き込む．

マトリクス・スケジューラのウェイクアップ・ロジックでは，プロデューサ・マトリクスによって命令同士の依存関係を表現する．プロデューサ・マトリクスは一辺が発行キューのエントリ数 WS の正方行列になっている．(a, b) がアサートされているときは，エントリ b の命令がエントリ a の命令に依存していることを表す．b の命令が発行されて依存が解消したらディアサートされる．命令がレディかどうかは，対応する行の要素がすべてディアサートされているがどうかで判断できる．命令を発行した場合は，対応する列の要素をすべてディアサートしてウェイクアップを行う．ウェイクアップの際のマトリクスの入力には，セレクト・ロジックの出力をそのまま利用できるため，ウェイクアップ/セレクトのフィードバック・ループからデスティネーション RAM を除去できる．

このため，プロデューサ・マトリクスは行単位および列単位で読み書きできる必要があるが，実はこのような回路は 1 ポートの RAM と同規模の回路で構成することができる [1]．これは，複数の行や列を一度に読み書きする場合でも変わらない．よって回路面積は，同時にウェイクアップする命令数，すなわち発行幅が増加しても変わらない．

以上から，発行幅が大きいスーパースカラ・プロセッサでは，マトリクス・スケジューラでは CAM 式ウェイクアップ・ロジックより回路面積を小さくできる．また，CAM 式ウェイクアップ・ロジックの面積はソース・オペランドの数に比例

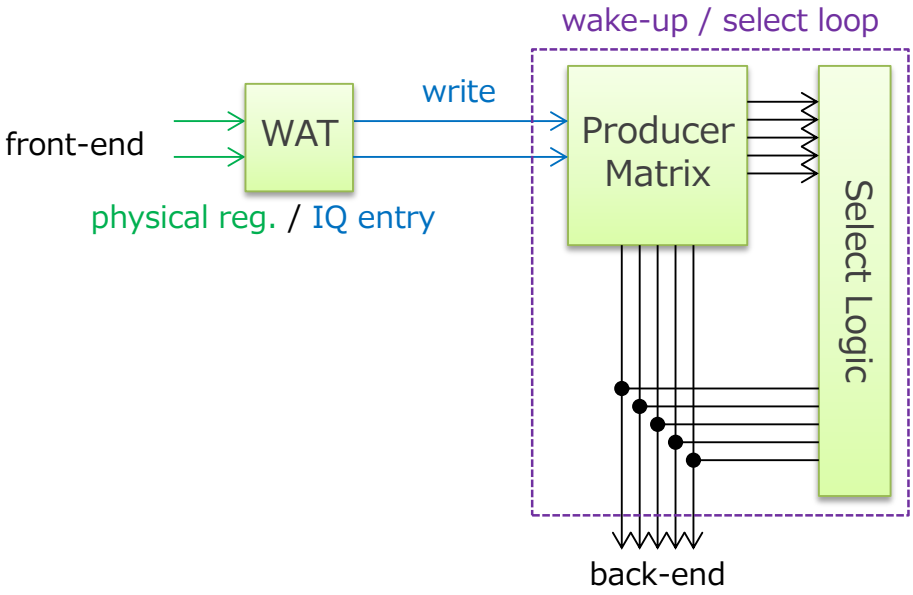

図 3.2: マトリクス・スケジューラの構成

するため，フラグ・レジスタやストア・セット [8] の採用によって命令の依存関係が増加すると面積が増えてしまうが，マトリクス・スケジューラはそれによらず面積が一定であるという点も有利である．ただし， WAT という新しいハードウェアが追加されているため，その面積も考慮に入れる必要がある．

3.2 分離リオーダ・バッファ

分離リオーダ・バッファは，リオーダ・バッファないしアクティブ・リストのハードウェア量を削減する技術である [3]．雷上動は正確な例外のためにアクティブ・リストを実装しており，リオーダ・バッファは使用していないので，以下ではアクティブ・リストを念頭に説明を行う．

アクティブ・リストには以下のフィールドが含まれる．

- PC: プログラム・カウンタ
- ecode: 例外原因コード
- dreg: 論理デスティネーション・レジスタ番号
- ppreg: dreg に以前割り当てられていた物理レジスタ番号

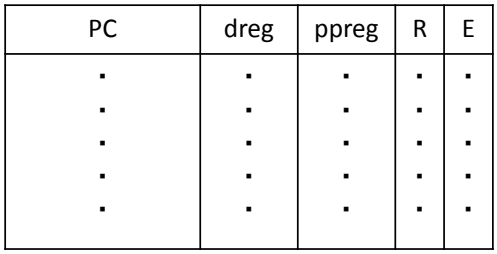

図 3.3: 分離リオーダ・バッファを採用しないアクティブ・リスト

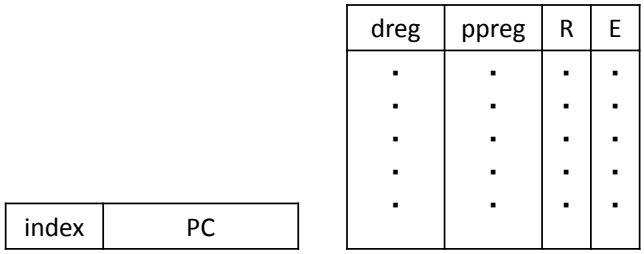

図 3.4: 分離リオーダ・バッファを採用するアクティブ・リスト

- R: 命令がリタイアしたことを示すフラグ
- E: 例外の有無を示すフラグ

実装に応じて他のフィールドが加えられることもあるが，基本的なものは以上である．通常のアクティブ・リストでは図 3.3 のように，アクティブ・リストのすべてのエントリがこれらのフィールドを保持する．

しかし，例外の内容に関する情報を保持するエントリは 1 つで十分である．なぜなら，例外が発生した時点でそれ以後の命令はキャンセルされてしまうため，複数の例外についての情報が同時に利用されることはありえないからである．つまり，例外の内容に関する情報は，例外を発生した命令のうち最も古いものについてのみ保持すればよい．

図 3.4 に分離リオーダ・バッファの模式図を示す．例外の情報を保持するフィールドである PC と ecode は，エントリ数を 1 に制限している．なお， PC は例外発生時に復帰すべき PC として使われる．例外の有無を示す E フラグは，すべてのエントリについて必要である．

3.3　非レイテンシ指向 レジスタ・キャッシュ・システム

非レイテンシ指向 レジスタ・キャッシュ・システム [4,5] は，レジスタ・ファイルにキャッシュを設けることで面積を削減する技術である．

ふつうの out-of-order スーパースカラ・プロセッサでは，レジスタ・ファイルは発行幅の 3 倍という非常に大きなポート数を持つ．1 つのバックエンドの命令パイプラインが，2 つのレジスタ・ファイル読み出しポートと 1 つのレジスタ・ファイル書き込みポートを必要とするためである．マルチポートメモリの面積はポート数の 2 乗に比例するため，発行幅が増えるに従ってレジスタ・ファイルの面積は非常に大きくなる．

非レイテンシ指向 レジスタ・キャッシュ・システムでは，メイン・レジスタ・ファイルとレジスタ・キャッシュという 2 つのハードウェアにより，レジスタ・ファイルを構成する．レジスタ・ファイルを読み出す際はまずレジスタ・キャッシュにアクセスし，キャッシュ・ミスが発生した場合のみメイン・レジスタ・ファイルにアクセスする．レジスタ・ファイルを書き込む際は，レジスタ・キャッシュとメイン・レジスタ・ファイルの双方に書き込みを行う．すなわち，このレジスタ・キャッシュの書き込みポリシーはライトスルー方式となる．

レジスタ・キャッシュのエントリ数はもとのレジスタ・ファイルのそれより小さくすることができる．これは，レジスタ・ファイルにもメモリと同様に参照の局所性が存在するためである．レジスタ・ファイルのごく一部の内容を保持していれば，ほとんどのレジスタ・ファイル・アクセスに対応することができる．塩谷らの評価 [5] では，128 エントリのレジスタ・ファイルに対して 8 エントリのレジスタ・キャッシュがあれば十分だとしている．

また，メイン・レジスタ・ファイルのポート数はもとのレジスタ・ファイルのそれより小さくすることができる．読み出しポートを小さくできるのは，メイン・レジスタ・ファイルを読み出す命令がレジスタ・キャッシュにミスした命令に限られるためである．一方，レジスタ・キャッシュは書き込みポリシーにライトスルー方式を採用するので，メイン・レジスタ・ファイルに書き込む命令は少なくならない．しかし，メイン・レジスタ・ファイルにライト・バッファを設けてやることで，書き込みの時間的な集中を避けることでポート数を削減してやることができる．なお，メイン・レジスタ・ファイルのポート数が不足した場合は，命令の再実行など何らかの対処が必要となる．そのため，非レイテンシ指向 レジスタ・キャッシュ・システムを採用すると性能はわずかに低下する．

レジスタ・キャッシュとメイン・レジスタ・ファイルの両方がもとのレジスタ・ファイルよりかなり小さいために，それらを合計しても全体としての面積は小さくなる．塩谷らの評価 [5] では，2.1% の小さな性能低下でレジスタ・ファイルの面積を 24.9% に抑えることができるとしている．

なお,「非レイテンシ指向」という接頭辞は，この技術がレジスタ・ファイルのアクセス速度向上ではなく，面積の削減を目的としていることに由来する．レジスタ・キャッシュのヒット時のレイテンシをミス時より短くすることも可能だが，ミスのたびにパイプライン処理が乱れるため，かえって性能の低下を招く．非レイテンシ指向 レジスタ・キャッシュ・システムでは，ヒット/ミス時のレイテンシを同じにすることで，ミス時もパイプライン処理を乱さない構造となっている．非レイテンシ指向 レジスタ・キャッシュ・システムでパイプライン処理が乱れるのは，メイン・レジスタ・ファイルのポートが不足した場合のみである．よって非レイテンシ指向 レジスタ・キャッシュ・システムはレジスタ・ファイルのアクセス速度向上には貢献しないが，最小限の性能低下で回路面積を大きく削減することができる．

第4章

雷上動の設計と実装

4.1　命令セット・アーキテクチャ

雷上動では32bit ARM 命令セットのサブセットを実行可能である．表 4.1 に示す通り，実行可能な命令は整数演算命令やロード/ストア命令などである．浮動小数点命令や OS の制御に関わる命令は実装していない．

あとで詳しく述べるように，これらの命令はプロセッサ内部で複数のマイクロ命令に分解されて実行されるが，アーキテクチャの都合上でそれが困難な場合がある．このうち，複数レジスタのロード/ストア命令については，アセンブリ・プログラムの段階で複数のロード/ストア命令に変換することにより対応している．その他の場合は，雷上動では未対応となる．表 4.1 で実行可能と分類されていても，条件コードやアドレッシング・モードによって未対応なこともある．

図 4.1 に，雷上動の論理レジスタの一覧を示す．32bit の汎用レジスタが 15 本，32bit の PC が 1 本，4bit のフラグ・レジスタが 1 本存在する．

雷上動では，オプションで整数 SIMD 命令をサポートできる．サポートする整数 SIMD 命令は，32bit × 4 の加算，乗算，ロード/ストアである．このオプションが有効な場合，論理レジスタとして 128bit の SIMD 用レジスタが 16 本追加される．

4.2　マイクロアーキテクチャ

本節では雷上動のマイクロアーキテクチャの概要について説明する．図 4.2 に雷上動のブロック図を，図 4.3 に命令パイプラインの構成をそれぞれ示す．

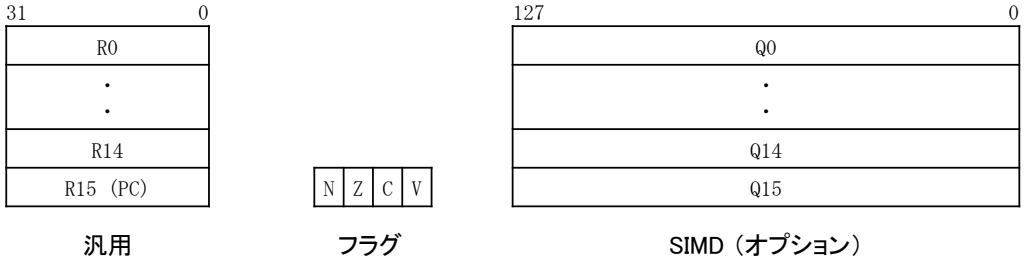

図 4.1: 雷上動の論理レジスタ一覧

雷上動の実装では，以下の要素の ON / OFF を切り替えることができるようになっている．

- マトリクス・スケジューラ
- 分離リオーダ・バッファ
- 非レイテンシ指向 レジスタ・キャッシュ・システム
- SIMD

本節ではこれらの要素をまったく用いない場合をベースラインとし，各要素を採用する場合の変更点についてはそのつど説明を行う．特に非レイテンシ指向 レジスタ・キャッシュ・システムと SIMD については 4.2.12 節と 4.2.13 節でそれぞれ詳しく説明を行う．

4.2.1 フェッチ

フェッチ・ステージでは，PC が指し示す命令アドレスの命令を命令キャッシュから読み込む．PC の更新は分岐予測に基づいて行われる．分岐すると予測した場

表 4.1: 雷上動の命令対応状況

対応	非対応	アセンブル時に変換
整数演算命令 ロード/ストア命令 プリフェッチ命令 整数 SIMD 命令	OS 関連の命令 浮動小数点演算命令	複数レジスタのロード/ストア命令

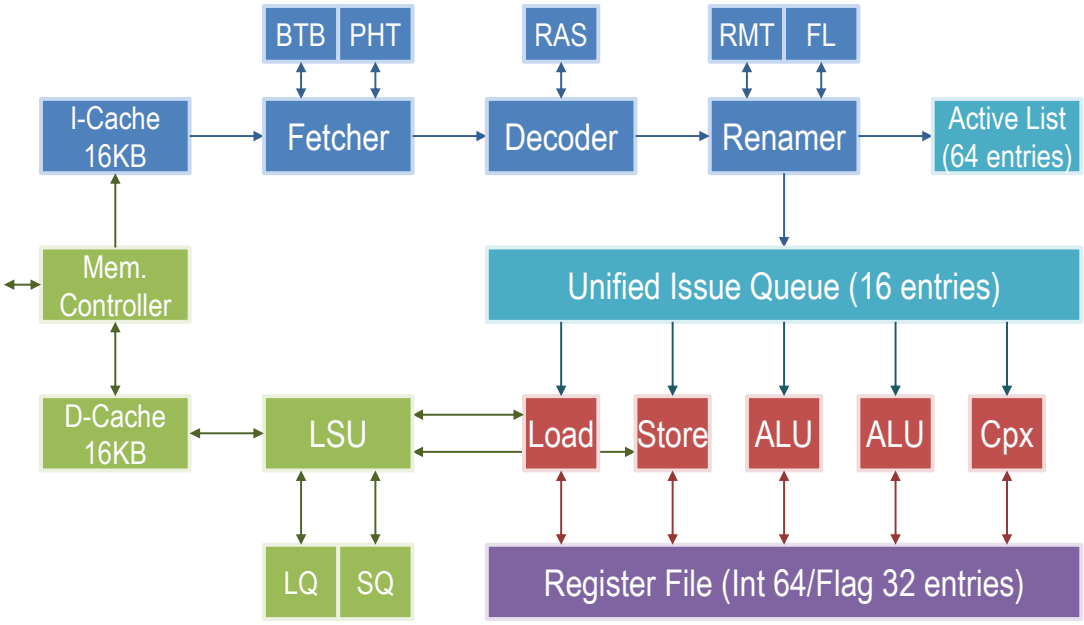

図 4.2: 雷上動のブロック図

合，フェッチ・グループ内で分岐より後にある命令をフラッシュし，次のサイクルの PC を予測された分岐先アドレスにセットする．そうでない場合は，フェッチした命令の数に応じて PC をインクリメントする．

命令キャッシュ

命令キャッシュはダイレクト・マップ方式で実装されている．キャッシュ・ミスが発生した場合，メモリから該当するキャッシュ・ラインを読みだして命令キャッシュに書き込み，そのあいだ下段のステージに対してバブルを送出する．命令キャッシュは読み出し専用なので，ミス時にライト・バックは行わない．

命令キャッシュからは一度に複数の命令を読みだすが，常に連続したアドレスでアクセスされるため，バンク分けが可能である．よって，実質的な読み出しポート数は 1 となる．

分岐予測

フェッチ・ステージに置かれる分岐予測器は，1 サイクルの間に結果が得られることが望ましい．なぜなら，分岐先アドレスが次のサイクルのフェッチに用いられるため，1 サイクルで結果を得られないとフェッチを毎サイクル行うことが不可能

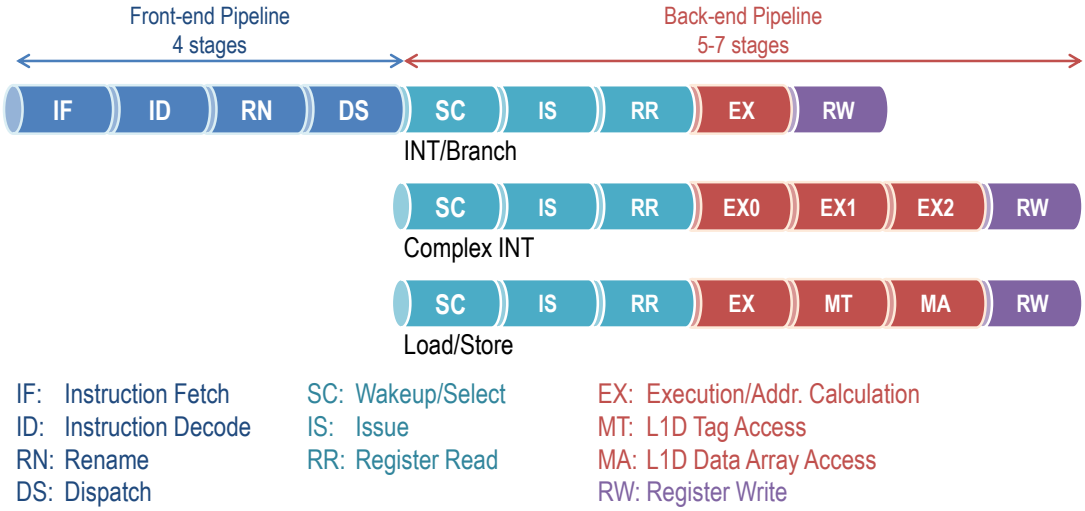

図 4.3: 雷上動の命令パイプライン

になり，性能が大きく低下するからである．

よって，高い周波数でも 1 サイクルで分岐結果が得られるよう，方向分岐予測器には単純な 2 ビット・カウンタ方式を用いた．g-share [9] などの高い予測精度を持つ方式は，回路が複雑になりクリティカル・パスになる可能性があると考えたため，採用を見送った．

2 ビット・カウンタの値は PHT (Pattern History Table) に，分岐先アドレスは BTB (Branch Target Buffer) に保持される．BTB の各エントリには分岐先アドレスの他に valid ビットが存在し，valid ビットが 1 の場合はその命令が分岐命令であると予測される．命令が分岐であり，かつ 2 ビット・カウンタの値から taken と予測された場合は，次の PC を BTB の示す分岐先アドレスにセットする．

PHT, BTB はともに命令アドレスの下位ビットをインデクスとしてアクセスするが，その値は常に連続するためバンク分け可能である．よって，実質的な読み出しポート数は 1 となる．

4.2.2 デコード

デコード・ステージではフェッチした ARM 命令をマイクロ命令に分解するほか，分岐予測の一部検証や， RAS (Return Address Stack) へのアクセスを行う．マイクロ命令分解は回路が複雑で遅延も大きいため，PD, ID の 2 ステージにわたって

行う (図 4.3)．マイクロ命令分解は PD ステージで行い，分岐予測の検証や RAS へのアクセスは ID ステージで行う．

マイクロ命令分解

32bit ARM 命令セット は複雑な命令を数多く含むため，デコード・ステージでマイクロ命令分解を行う.「複雑な命令」の具体例とその分解方法をを以下に示す.

- 3 つ以上のオペランドを持つ演算命令.
  - ハードウェアの都合からマイクロ命令のオペランドは最大 2 つ（フラグ・レジスタを除く）としているので，複数の 2 オペランド演算命令に分解する.
- 複雑なアドレッシング・モードを持つロード/ストア命令.
  - アドレス計算命令とロード/ストア命令に分解する.
- PC をデスティネーションとする演算およびロード命令.
  - 演算およびロード命令とレジスタ間接分岐命令に分解する.

分解したマイクロ命令は PD/ID ステージ間のパイプライン／レジスタに保持され，ID ステージで次のステージに送るマイクロ命令が選択される．1 サイクルですべてのマイクロ命令を送れない場合は，パイプラインの前段をストールする．

1 つの ARM 命令は最大 3 命令までのマイクロ命令に分解される．3 命令である理由は，以下の 2 点である.

- 多くの ARM 命令は 3 つ以下のマイクロ命令で処理可能.
- 1 度に生成されるマイクロ命令の数を増やすと，ID ステージのマイクロ命令選択回路が非常に大きくなる.

しかし，最大で 16 のレジスタをロード/ストアする命令など，3 つ以内のマイクロ命令に分解するのが困難な ARM 命令も存在する．そのような命令はアセンブリ言語の段階で他の命令列に変換している．それすらも困難な場合，雷上動では未サポートとしている.

デコード・ステージにおける分岐予測の検証

デコード・ステージでは，分岐と予測された命令が実際に分岐であったかの検証を行う．フェッチ・ステージの分岐予測では命令が分岐か否か自体を予測していたが，その結果はデコード時に判明するため，分岐予測の検証の一部をこのステージで行うことが出来る.

Taken な分岐と予測されたがそもそも分岐では無かった場合は，前段のパイプラインをフラッシュして再度フェッチを行う．Untaken な分岐と予測されたが分岐では無かった場合は，特に不正な処理を行っていないことになるので，パイプラインはフラッシュせずに処理を続行する．

デコード・ステージで検出できるのは一部の分岐予測ミスであるが，出来る限り早くミスを検出することが性能向上につながる．分岐方向や分岐先の予測ミスは実行ステージまで判明しないが，その時点では多くの後続命令がフェッチされているため，それをすべてキャンセルするとパイプラインが大幅に乱れてしまう．デコード・ステージで判明するのは命令が分岐か否かのみであるが，この場合はフェッチおよびデコード段の命令のみをキャンセルすればよく，パイプラインの乱れは比較的小さい．

RAS

RAS (Return Address Stack) とは，サブルーチンのリターン・アドレスを格納するスタックである．サブルーチンからのリターンはレジスタ間接分岐命令となるが，その分岐先はサブルーチンの呼び出し元によって変わるため，BTB を用いた分岐先予測では精度が上がらないことが多い．しかし，大半の場合において分岐先はサブルーチンのリターン・アドレス，すなわち呼び出し元の次の命令である．そこで，通常の分岐予測器とは別にサブルーチンのリターン・アドレスを記憶しておけば，サブルーチンからのリターン時に高い精度を持つ分岐先として使用でき，性能向上に役立てることができる．

RAS へのアクセスは以下の要領で行われる．サブルーチン呼び出しに該当する分岐命令をデコードした際に，その PC を RAS にプッシュする．サブルーチンからのリターンに該当するレジスタ間接分岐命令をデコードした際は，RAS から値をポップして新しい分岐先アドレスの予測値として用いる．すなわち，パイプラインの前段をフラッシュし，RAS からポップした値を PC にセットして再度フェッチを行う．この値が正しい分岐先であったかの検証は，通常の分岐予測の検証と同じように実行ステージで行う．

4.2.3 リネーム

リネーム・ステージではレジスタ・リネーミングに加え，各種資源の確保とアクティブ・リストへの書き込みを行う．

レジスタ・リネーミング

リネームを行うリネーム･ロジックは主に RMT (Register Map Table) とフリー･リストからなる．

RMT は論理レジスタ番号をインデクスとし，物理レジスタ番号を格納するテーブルで，各論理レジスタがどの物理レジスタに割り当てられているかを示している．RMT はマルチポート RAM として構成されるが，あるサイクルで書き込んだ値を同じサイクルで読み出さねばならない場合があるので，専用のバイパス・ネットワークが必要となる．RMT のバイパス・ネットワークはレジスタ番号の比較器とセレクタで構成される．

フリー・リストは論理レジスタに割り当てられていない物理レジスタ番号のリストである．物理的には RAM として実装され，head と tail のポインタによってリストの先頭と末尾を表現する．フリー・リストを読み出す際はリストの先頭から，書き込む際はリストの末尾から，それぞれ順番にアクセスする．

これらを用いたレジスタ・リネーミングは以下のように行われる．

1. ソース・レジスタの論理レジスタ番号を使用して RMT を参照し，物理レジスタ番号に変換する．
2. デスティネーション・レジスタの論理レジスタ番号に現在割り当てられている物理レジスタ番号を得るため，RMT を参照する．この時得られた物理レジスタ番号は物理レジスタ解放の際に使用される [10].
3. フリー・リストから物理レジスタ番号を取り出し，デスティネーション・レジスタに割り当てる．また，割り当てられた物理レジスタ番号を RMT に書き込み，更新する．

なお，32bit ARM 命令セットにはフラグ・レジスタも存在するため，これについても別途レジスタ・リネーミングを行う．RMT とフリー・リストは，汎用レジスタとフラグ・レジスタ用にそれぞれ用意される．論理フラグ・レジスタは 1 つしか存在しないため，フラグ・レジスタ用の RMT も 1 エントリとなり，テーブルよりむしろレジスタと呼ぶべきものになる．

リネーム・ロジックの回路はリネーム幅に増加に伴って大きくなる．RMT の読み出しポート数はリネーム幅の 3 倍，書き込みポート数はリネーム幅に等しく，リネーム幅が大きくなると RMT の面積はその 2 乗に比例して大きくなる．RMT のバイパス・ネットワークも，リネーム幅が増加すると非常に複雑なロジックとなる．ただし，フリー・リストは常に連続した領域にアクセスするため，バンク分

けが可能である．よって，リネーム幅が増加してもフリー・リストの面積はそれほど増加しない．

資源の確保

リネーム・ステージでは以下の資源の確保を行う．

- アクティブ・リストのエントリ．
- 発行キューのエントリ．
- ロード・キューのエントリ（ロード命令のみ）．
- ストア・キューのエントリ（ストア命令のみ）．

これらの資源が不足していて確保出来なかった場合，パイプラインの前段をストールさせて，後段にバブルを送出する．

上記の資源のうち，アクティブ・リスト，ロード・キュー，ストア・キューは head と tail のポインタによって空き領域を管理し，発行キューはフリー・リストによって空き領域を管理する．アクティブ・リスト，ロード・キュー，ストア・キューはコミット・ステージにて in-order にエントリが解放されるため，必ず確保した順番で解放される．よって，使用領域と空き領域を常に連続させておき，head と tail のポインタで空き領域を管理することが可能である．一方，発行キューは発行ステージで out-of-order に資源の解放が行われるため，確保した順番で解放されるとは限らない．よって，使用領域と空き領域が不連続になり head と tail のポインタのみで空き領域を確保するのは困難なので，フリー・リストを用いて空き領域の管理を行っている．

アクティブ・リストへの書き込み

アクティブ・リストのエントリの確保に成功したら，命令の情報を書き込む．書き込む情報は，以下のようなコミット時に必要な情報である．

- dreg: 論理デスティネーション・レジスタ番号
- ppreg: dreg に以前割り当てられていた物理レジスタ番号

また，以下のフラグをクリアする．

- R: 命令がリタイアしたことを示すフラグ
- E: 例外の発生を示すフラグ

アクティブ・リストの書き込み幅はリネーム幅に等しくなるが，リネーム・ステージで確保したアクティブ・リストのエントリは常に連続しているので，書き込みについてはバンク分けが可能である．

4.2.4 ディスパッチ

ディスパッチ・ステージでは，レディ・ビット・テーブルを読み出してオペランドがレディか調べた後，命令を発行キューに書き込む．発行キューに書き込まれた命令は，ウェイクアップ/セレクトの対象となる．このステージが，フロントエンドの最後のステージとなる．

レディ・ビット・テーブルの読み出し

レディ・ビット・テーブルは，各物理レジスタがレディか否かを保持するテーブルである．命令は各ソース・オペランドに対応するレディ・ビットを読み出す．すべてのオペランドがレディであった場合は，依存関係が解決済みなので命令は即座にウェイクアップされる．命令にレディでないソース・オペランドがある場合は，依存元によってウェイクアップされるのをスケジューラで待つことになる．

各命令のソース・オペランドは最大 2 つ存在するため，レディ・ビット・テーブルの読出しポート数はディスパッチ幅の 2 倍となる．レディ・ビット・テーブルのビット幅は 1 と小さいが，ポート数は比較的大きくなるので，その面積にも注意する必要がある．

32bit ARM 命令セットにはフラグ・レジスタも存在するため，フラグ・レジスタ用のレディ・ビット・テーブルも用意する必要がある．フラグ・レジスタのソース・オペランドは最大 1 つであるため，フラグ・レジスタ用のレディ・ビット・テーブルの読出しポート数はディスパッチ幅と等しくなる．

発行キューへの書き込み

発行キューへの書き込みとひとくちにいっても，書き込み対象となるハードウェアは複数存在する．大きく分けて，ウェイクアップ/セレクトに必要なものと，命令の実行に必要なものである．

ウェイクアップ/セレクトに必要な情報は，デスティネーション RAM やソース CAM ないしプロデューサ・マトリクスに書き込まれる．書き込まれる情報は，

オペランドのレディや物理レジスタ番号のような命令の依存関係に関するものである．

命令の実行に必要な情報は，ペイロード RAM に書き込まれる．その内容は，演算の種類やアクティブ・リストのエンドリ番号など多岐にわたる．ペイロード RAM はディスパッチ幅に等しい書き込みポート数を持つことになる．

4.2.5 スケジューリング

スケジューリング・ステージでは，発行キューにディスパッチされた命令に対してウェイクアップ/セレクトを行う．

ウェイクアップ・ロジックには CAM 式ウェイクアップ・ロジックとマトリクス・スケジューラのどちらかを使うことができる．回路の内容は，3.1 節で説明したとおりである．マトリクス・スケジューラを採用する場合は，WAT を追加してリネーム・ステージでアクセスする．

セレクト・ロジックは単純な組み合わせ回路で構成され，ここでセレクトされた命令は次のステージとウェイクアップ・ロジックの双方に送られる．後者は依存する命令をウェイクアップするために用いられるが，このときセレクトされた命令の実行レイテンシだけ待つ必要がある．よって，セレクト・ロジックからウェイクアップ・ロジックに向かう信号には，命令の実行レイテンシに等しい段数のパイプライン・レジスタを挿入する．

命令をウェイクアップした際には，その命令のデスティネーション・オペランドがレディになるため，前節で述べたレディ・ビット・テーブルを更新する必要がある．このとき，デスティネーション RAM から読みだした値をレディ・ビット・テーブルのインデクスとして用いる．図 3.2 ではデスティネーション RAM が描かれていないが，実際はマトリクス・スケジューラでもデスティネーション RAM を用意する必要がある．

命令ウィンドウの非集中化

整数演算，浮動小数点演算，ロード/ストア命令というように，命令の種類ごとに発行キューを分けることを命令ウィンドウの非集中化という．それぞれの命令ウィンドウにおいては見かけ上発行幅が少なくなるため，スケジューリングに関するいくつかのハードウェアのポート数を減らし，回路面積を小さくすることができる．

雷上動では全ての種類の命令を単一の発行キューに格納する構成をとっており，命令ウィンドウの非集中化は行っていない．その理由は，単純に未実装であるからである．よって，命令ウィンドウの非集中化により面積を削減する余地が残っている．

ロード/ストア命令のスケジューリング

ロード/ストア命令も，他の命令と同じようにスケジューリングが行われる．ロード/ストア命令はメモリによるデータ依存が存在する可能性があるが，それらはまったく存在しないものとしてスケジューリングし，投機的に out-of-order 実行する．ストア・セット [8] のようなメモリ依存予測器は用いない．

4.2.6 発行

発行ステージでは，セレクトされた命令の発行キュー から読み出し，次段以降のステージに送り出す．同時に，発行キューのエントリを解放する．命令の情報は発行キューのペイロード RAM から読み出すので，ペイロード RAM には発行幅に等しい読出ポートが必要となる．

再発行

雷上動には，キャッシュ・ミスしたロード命令とそれに依存する命令を再発行する機能がある．普通，命令が正しく実行できなかった場合はパイプラインをフラッシュしてフェッチをやり直す必要があるが（例：分岐予測ミス），キャッシュ・ミスの場合はリフィルを待って再発行すれば正しい結果を得ることが出来る．再発行は再フェッチに比べてパイプラインの乱れが小さいため，可能な場合は再発行を用いるほうが性能上有利となる．

再発行の対象となる命令の判別は，データ・パスに対して valid ビットを設けることによって行う．実行結果の valid ビットがリセットされている場合，その実行結果は不正であるので再発行の対象となる．実行結果の valid ビットがリセットされるのは以下の場合である．

- ロードがキャッシュ・ミスを起こした場合．
- ソース・オペランドのどれか 1 つでも valid ビットがリセットされていた場合．

再発行の対象となった命令は，リプレイ・キューに格納される．リプレイ・キューで保持されるデータの内容は，発行キューのペイロード RAM と同じものである．

特定のサイクル数が経過すると，リプレイ・キューに格納されていた命令が再び発行ステージに送り出される，すなわち再発行が行われる．この時，バックエンド・パイプラインの前段のステージをストールさせる．再発行までのサイクル数は，データ・キャッシュのリフィルにかかる時間を目安とする．再発行が早過ぎると，データ・キャッシュのリフィルが終わっていないために，再度キャッシュ・ミスを起こしてしまう[1]．再発行が遅すぎると，実質的にリフィルのレイテンシが延びるのと同じであるため，性能低下につながる．

リプレイ・キューはバックエンド・パイプラインの各レーンごとに用意することで，読み出しおよび書き込みのポート数をそれぞれ1にすることができる．

4.2.7 レジスタ読み出し

レジスタ読み出しステージでは，物理レジスタ・ファイルの読出しと，オペランド・バイパスのためのレジスタ番号の比較を行う．

レジスタ読み出し

レジスタ読み出しステージでは物理レジスタ・ファイルからソース・オペランドを読み出す．命令のソース・オペランドは最大で2つなので，物理レジスタ・ファイルの読み出しポート数は発行幅の2倍となる．

フラグ・レジスタについては，汎用レジスタとは別に専用の物理レジスタ・ファイルを設けている．フラグについてはソース・オペランドが最大で1つなので，読み出しポート数は発行幅と等しくなる．

オペランド・バイパスためのレジスタ番号の比較

雷上動のオペランド・バイパスでは，レジスタ読み出しステージでバイパス元を特定し，実行ステージでバイパス・ネットワークにアクセスしてオペランドを読み込む．バイパス元は，バックエンド・パイプラインの種類とレーン番号およびステージの組み合わせで表現される．例えば,「INT パイプラインの 2 レーン目の RW ステージ」という形になる．

バイパス元を特定するために，レジスタ読み出しステージではレジスタ番号の比較を行う．具体的には，各レーン・各ステージの命令のデスティネーション・レ

[1]二度目のキャッシュ・ミスを起こした命令は，再びリプレイ・キューに格納され，三回目の発行を行うことになる．実はこの方法は，キャッシュ・アクセスのパターン次第で再発行の無限ループに陥る危険性がある．

ジスタ番号をシフトレジスタに格納しておき，それと自命令のソース・レジスタ番号を比較して一致したものがバイパス元となる．一致するものが無ければオペランド・バイパスは行わず，物理レジスタ・ファイルから読みだしたデータをオペランドとして使用する．

レジスタ・キャッシュ読み出し

非レイテンシ指向 レジスタ・キャッシュ・システムを採用する場合，このステージでレジスタ・キャッシュとメイン・レジスタ・ファイルの双方にアクセスする．つまり，レジスタ・キャッシュのヒット/ミスを判定し，ヒットならレジスタ・キャッシュの値を読み出し，ミスならメイン・レジスタ・ファイルの値を読み出す．なお，レジスタ番号の比較によってオペランド・バイパスを行うと決定した場合は，レジスタ・キャッシュとメイン・レジスタ・ファイルへのアクセスは行わない．

4.2.8 演算の実行

整数演算パイプラインでは ALU/シフタを用いる整数演算命令と，分岐命令を実行する．実行ステージは 1 段である．

コンプレックス整数演算パイプラインは 3 段にパイプライン化されていて，1 サイクルでは実行できないような複雑な演算を実行する．具体的には，乗算とビット・カウントである．ビット・カウントとは，ソース・オペランドの上位から連続しているビット 0 の数を数える命令である．32bit ARM 命令セットにおいて除算命令は比較的新しいバージョンにしか含まれていないため，雷上動では除算器の実装を省略している．

4.2.9 ロード/ストア命令の実行

ロード・パイプラインはアドレス計算，タグ・アクセス，アレイ・アクセスの 3 ステージからなる．タグ・アクセス・ステージではデータ・キャッシュのタグにアクセスし，ヒット・ミス判定を行う．また，このステージではロード・キュー/ストア・キューへのアクセスを行い，ストア–ロード・フォワーディングやアクセス順序違反の検出を行う．そして続くアレイ・アクセス・ステージでロード結果を得る．

ストア・パイプラインも，ロードと同じ3ステージからなる．ただし，アレイ・アクセス・ステージではキャッシュへの書き込みは行わない．ストアの実行結果はストア・キューに書き込まれ，後のコミット時にデータ・キャッシュに書き戻される．

雷上動は1バイトおよび2バイト単位のロード/ストアにも対応している．そのため，データ・キャッシュにはバイトごとにライト・イネーブルを設け，ストア・キューでも書き込むバイトの情報を保持している．また，ロード結果のシフタも用意している．ただし，4バイト境界をまたぐ場合については対応していない．

データ・キャッシュ

データ・キャッシュはダイレクト・マップ方式で実装されている．キャッシュ・ミス時には同じインデクスのキャッシュ・ラインをメモリにライト・バックして，ミスしたキャッシュ・ラインをメモリから読み込みリフィルを行う．

データ・キャッシュはノンブロッキング・キャッシュとなっているため，キャッシュ・ミスが発生した場合もその他の命令はデータ・キャッシュにアクセスが可能である．これを実現するため，キャッシュ・ミスを MSHR (Miss Status Handling Register) によって管理している．MSHR にはキャッシュ・ミスしたアドレスや，ライト・バックするアドレスおよびデータなどが格納される．

キャッシュ・ミス時の動作を以下に示す．MSHR は複数エントリ存在し，資源の空き具合に応じて各段階の処理が並行して行われる．

1. MSHR のエントリを確保する．
2. ライト・バックの対象となるキャッシュ・ラインを読み出し，MSHR に格納する．
3. ライト・バックの対象となるキャッシュ・ラインのデータをメモリに書き込む．
4. ミスしたキャッシュ・ラインをメモリから読み込む．
5. 新しいキャッシュ・ラインをデータ・キャッシュに書き込む．
6. MSHR のエントリを解放する．

メモリ階層

雷上動のキャッシュは命令キャッシュとデータ・キャッシュのみであり，2次キャッシュは存在しない．そのため，命令キャッシュとデータ・キャッシュは簡単なアービタを通じてメモリ・コントローラにつながっており，キャッシュ・ミス時は直接メ

モリを読み書きする構造になっている．なお，簡単のために命令キャッシュとデータ・キャッシュの間でコヒーレンシを維持する機構は省略している．

ストア–ロード・フォワーディング

ストア–ロード・フォワーディングはストア・キューを利用して行う．ストア・キューには，ストア命令の実行時にストア・アドレスとストア・データが格納される．ロード命令は実行時にストア・キューを検索し，同じアドレスの先行ストアを発見した場合は，フォワーディングを行ってそのストア・データをロード結果とする．正しくフォワーディングを実行できなかった場合はフォワード・ミスとなり，例外を発生させる．フォワード・ミスとなるのは例えば，ロード命令がワード・アクセスでストア命令がバイト・アクセスであった場合などである．

ストア–ロード･フォワーディングの際にアドレス比較を行うため，ストア･キューは CAM として構成される．ストア・アドレスは CAM におけるキーとなり，ストア・データは値となる．

メモリ・アクセス順序違反の検出

メモリ・アクセス順序違反の検出は，ロード・キューを利用して行う．ロード・キューには，ロード命令の実行時にロード・アドレスが格納される．ストア命令は実行時にロード・キューを検索し，同じアドレスの後続ロードを発見した場合は，メモリ・アクセス順序違反となって例外を発生させる．

4.2.10　レジスタ書き込み

レジスタ書き込みステージでは物理レジスタ・ファイルとアクティブ・リストに書き込みを行う．

レジスタ書き込み

レジスタ書き込みステージでは，物理レジスタ・ファイルに演算結果を書き込む．物理レジスタ・ファイルには汎用レジスタ/フラグ・レジスタに対応する 2 種類があるが，両者ともこのステージで書き込みを行う．命令のデスティネーションは汎用レジスタ/フラグ・レジスタそれぞれについて最大 1 つずつなので，物理レジスタ・ファイルの書き込みポート数は発行幅に等しくなる．

レジスタ・キャッシュ書き込み

非レイテンシ指向 レジスタ・キャッシュ・システムを採用する場合は，このステージでレジスタ・キャッシュとメイン・レジスタ・ファイルの双方に書き込みを行う．なお，メイン・レジスタ・ファイルにライト・バッファを設けて書き込みポートを減らすという実装は行っていない．よって，メイン・レジスタ・ファイルの書き込みポート数は，もとのレジスタ・ファイルと同じになる．

アクティブ・リストへの書き込み

レジスタ書き込みステージでは，アクティブ・リストの以下のフィールドをセットする．

- R: 命令がリタイアしたことを示すフラグ
- E: 例外の有無を示すフラグ

このフィールドはリネーム・ステージでも書き込みを行うため，書き込みポート数はリネーム幅と発行幅の和となる．

例外が発生した場合は，以下のフィールドを適切にセットする．

- PC: 復帰すべきプログラム・カウンタ

このフィールドはレジスタ書き込みステージでしか書き込みを行わないので，書き込みポート数は発行幅に等しくなる．

なお，分離リオーダ・バッファの採否によって，アクティブ・リストへの書き込み動作が一部異なる．分離リオーダ・バッファを採用しない場合は，アクティブ・リストのすべてのフィールドについて各命令固有の領域が確保されているのでそこに書き込みを行う (図 3.3)．分離リオーダ・バッファを採用する場合は，例外を発生させたインフライト命令の中で自命令が最も古い場合のみ例外関係のフィールドを更新する．命令の古さの比較は，命令に対応するアクティブ・リストのエントリ番号を比較することで実現できる．そのため，PC のフィールドに対応するアクティブ・リストのエントリ番号も記録する (図 3.4).

4.2.11 コミット

コミット・ステージでは命令をコミットする．また，資源の解放やストア・データのデータ・キャッシュへの書き戻し，RRMT (Retirement RMT) の更新を行う．さらに，例外が発生した場合は適切なリカバリ処理も行う．

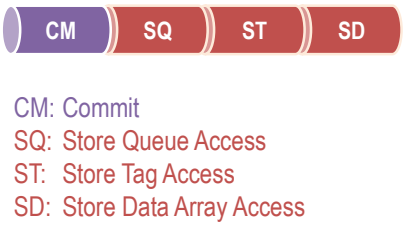

図 4.4: ストア命令のコミット・パイプライン

命令のコミットはアクティブ・リストを先頭を読み出して行われる．アクティブ・リストには，コミットに必要な命令の情報が記録されている．アクティブ・リストの読み出しはコミット・ステージでしか行わないため，読出ポート数はコミット幅に等しくなる．

資源の解放

命令をコミットした際，それによって不要になった資源を解放する．解放される資源には以下のものがある．

- アクティブ・リストのエントリ
- デスティネーションに以前割り当てられていた物理レジスタ・ファイルのエントリ
- ロード・キューのエントリ（ロード命令のみ）
- ストア・キューのエントリ（ストア命令のみ，キャッシュ書込後）

ストア命令のコミット

ストア命令はコミットの後にデータ・キャッシュへの書き込みを行う．2.1.3 節で述べたように，コミットの前にデータ・キャッシュへの書き込みを行うと，正確な例外を実現できなくなるからである．

データ・キャッシュへの書き込みは図 4.4 に示すパイプライン処理で実現される．このパイプラインは，ストア命令のみが in-order に通過する 1 レーンのものである．各ステージの処理を以下に記す．

- CM: コミット・ステージ．
- SQ: ストア・キューを読み出し，ストア・アドレスとストア・データを得る．

- ST: データ・キャッシュのタグ・アレイにアクセスする．キャッシュ・ミスが発生した場合はリフィルを行い，その間パイプラインをストールする．
- SD: データ・キャッシュのデータ・アレイに書き込みを行う．同時に，ストア・キューのエントリを解放する．ストア命令の処理はこれで全て終了する．

RRMT の更新

RRMT (Retirement RMT) は，RMT と同様に論理レジスタ番号をインデクスとして物理レジスタ番号を保持するテーブルである．ただし，常に最後にコミットした命令に対応している点が RMT とは異なる．すなわち， RRMT には最後にコミットした命令にとっての論理レジスタと物理レジスタの対応が記録されている．

RRMT は命令のコミット時に更新する．具体的には，コミットする命令のデスティネーションの論理レジスタ番号 *dreg* と物理レジスタ番号 *preg* の対応を上書きする．すなわち， RRMT の *dreg* に対応するエントリに *preg* の値を書き込む．

RRMT の値は，すぐ後で述べるようにリカバリの際に使われる．

例外発生時のリカバリ処理

例外が発生した際は，後続の命令をキャンセルしてプロセッサの状態を巻き戻すリカバリ処理を行う．リカバリ処理が終わったら，例外ごとに定められた復帰先から再フェッチを行う．雷上動で発生する例外と，その復帰先の一覧を表 4.2 に示す．

リカバリ処理の内容は以下の通りである．

- パイプラインをフラッシュする．
- アクティブ・リスト，発行キュー，ロード・キューのすべてのエントリを解放する．
- ストア・キューのエントリのうち，対応するストア命令がコミット済でないものを解放する．
- RMT に RRMT の値をコピーする．

RMT に RRMT の値をコピーすることによって，論理レジスタの値は最後にコミットした命令まで実行した時のものになる．

コミットに関する制約

雷上動ではリカバリを正しく行うため，コミットに関して以下の制約が存在する．

1. 同じ ARM 命令に属するマイクロ命令は，同時にコミットする．
2. コミット幅は 3 以上とする．

まず，1 つ目の制約の理由を述べる．フェッチの時点では命令はマイクロ命令に分解されていないため，リカバリ後の再フェッチは ARM 命令単位で行われる．よって，リカバリ前も ARM 命令単位でコミットが終了している必要がある．ARM 命令に属する一部のマイクロ命令だけがコミットされた状態でリカバリが行われ，別の ARM 命令が再フェッチされてしまうと，不正な結果となる可能性がある．一方，一度マイクロ命令をコミットしてしまったら，それを取り消すのは不可能である．よって，同じ ARM 命令に属するマイクロ命令は，それらがすべてコミット可能であることを確認したうえで，同時にコミットを行う必要がある．

2 つ目の制約は，1 つ目の制約から導かれるものである．1 つのマイクロ命令は最大 3 マイクロ命令に分解されるため，最低でも 1 サイクルの間に 3 マイクロ命令がコミットできないと，コミット不可能な命令が出てきてしまう．よって，コミット幅は 3 以上にしなければならない．

4.2.12 雷上動における非レイテンシ指向 レジスタ・キャッシュ・システム

非レイテンシ指向 レジスタ・キャッシュ・システムについて述べた既存の論文 [5] では，レジスタ・キャッシュを多ポートの RAM とし，キャッシュの置き換えアルゴリズムを LRU としているが，このような回路を効率よく構成することは困難である．そのため，雷上動ではレジスタ・キャッシュの構成を工夫することで面積

表 4.2: 雷上動で発生する例外の一覧

例外の種類	命令の種類	復帰先の命令
分岐予測ミス	分岐命令	分岐先
フォワードミス	ロード命令	自命令
アクセス順序違反検出	ストア命令	次の命令
未定義命令実行	未定義命令	-

削減を図っている．

まず，レジスタ・キャッシュをバックエンド・パイプラインのレーンごとに設けている．つまり，あるレーンでレジスタ・キャッシュに書き込んだデータは，そのレーンでしか読みだすことができない．これにより，レジスタ・キャッシュのポート数を削減している．

さらに，レジスタ・キャッシュを設置するのは整数演算パイプラインに限定している．ロード，コンプレックス整数演算の結果をキャッシュしても，同じパイプラインで再利用する可能性は低いと判断したためである．ストアについては，そもそもキャッシュすべき実行結果が存在しない．

また，キャッシュの置き換えアルゴリズムに NLU (Not Last Used) を用いている．これにより，LRU を用いた場合より回路面積を少なくできる．

以上の構成のレジスタ・キャッシュでも，十分な面積効率向上効果を持っていると筆者は考えている．しかし，標準的なベンチマークで評価して確認する必要があり，それは今後の課題となる．

4.2.13 SIMD

雷上動では，オプションとして整数 SIMD 命令に対応する．サポートする命令は，32bit × 4 の加算，乗算，ロード/ストア命令である．SIMD 加算/乗算命令は，ともにコンプレックス整数演算パイプラインで実行される．SIMD ロード/ストア命令は，32bit × 4 = 16 byte にアラインされたメモリ・アドレスにのみ対応する．

SIMD を有効にする場合の主なハードウェアの変更点を，以下に列挙する．

- SIMD 用の物理レジスタ・ファイルおよびバイパス・ネットワークを追加．
- リネーム・ロジックのフリー・リストについて，SIMD 用のものを追加する．
- RMT を拡張し，SIMD レジスタのリネーミングに対応させる．
- データ・キャッシュやストア・キューのデータ・アレイのビット幅を，32bit から 128bit に拡張する．

4.3 FPGA 上での実装

本節では雷上動を FPGA に実装する上でのトピックである，マルチポートメモリの構成法について説明する．4.3.1 節で FPGA の内蔵 RAM について説明し，

4.3.2 節で FPGA の内蔵 RAM を使ってマルチポートメモリを構成する方法をいくつか述べる．それらの方法のうち，雷上動内のマルチポートメモリはいずれを用いるのかという話を 4.3.3 節で述べる．

4.3.1 FPGA の内蔵 RAM

多くの FPGA は内蔵 RAM を持っており，ユーザーのデザインにおいて使用できる．メモリの機能を持つ回路は組み合わせ回路と D-FF を使っても実現できるが，内蔵 RAM に比べると回路面積や消費電力の面で大きく劣る．そのため，メモリの実装には可能な限り内蔵 RAM を用いるのが普通である．

FPGA の内蔵 RAM には Block RAM と LUT-RAM の 2 種類が存在する．FPGA ベンダによって呼称や分類の仕方が異なることがあるが，基本的にはどのベンダの製品にもこの 2 種類に該当する内蔵 RAM が存在する．以下，それぞれを詳しく説明する．

BlockRAM

FPGA 上に 専用のブロックとして存在する RAM を，Block RAM と呼ぶ．Block RAM は一般に数 KB から数十 KB の容量を持ち，ポート数は高々2 程度であることが多い．

LUT-RAM

一部の LUT は RAM として利用できる場合があり，これによって構成する RAM を本論文では LUT-RAM と呼ぶ．LUT-RAM は，Xilinx 社の FPGA の場合は分散 RAM，Altera 社の FPGA の場合は MLAB という呼称となる．

LUT が RAM として利用できるのは，n 入力論理関数の実装に SRAM を利用する方式が一般的だからである．論理関数の入力を SRAM のアドレス入力に対応させ，出力を SRAM のデータ出力に対応させると．ビット幅が 1 でエントリ数が 2^n の SRAM を用いれば任意の n 入力論理関数を実現できることがわかる．このような場合， FPGA ベンダは簡単な回路を付加するだけで，LUT を RAM としても使えるようにできる．

LUT-RAM の容量は，LUT の入力数 n に対して 2^n ビットとなる．n はおおむね 4 から 8 程度なので，1 つの LUT-RAM の容量は数十ビット程度となり，Block RAM と比べると大変小さい．ポート数は Block RAM と同様，高々2 程度である．

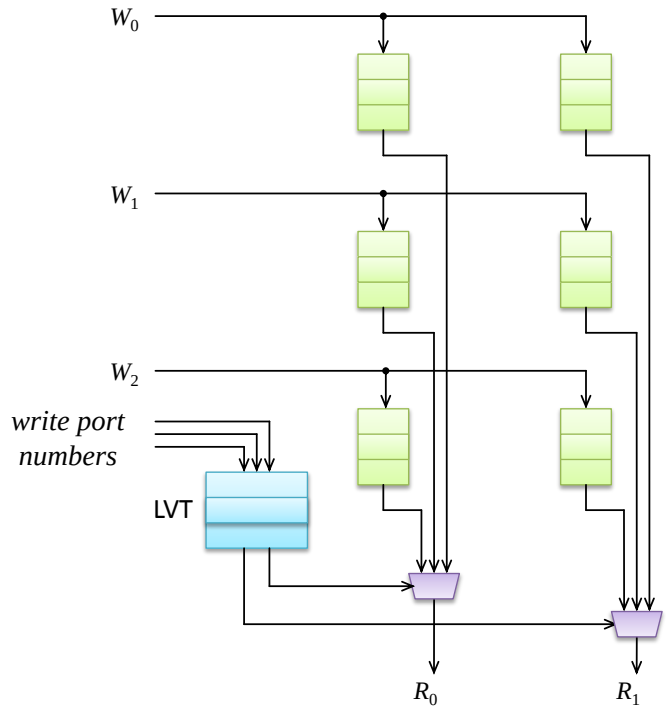

図 4.5: Live Value Table によるマルチポート RAM の構成法

4.3.2 FPGA におけるマルチポートメモリの構成法

前節で述べたように内蔵 RAM のポート数は高々2 ポートであるので，これをそのままマルチポート RAM として用いることは不可能である．組み合わせ回路と D-FF によってマルチポート RAM と同等の機能を持つ回路を構成することは難しくないが，非常に多くの資源を消費することになってしまう．

本節では，1R1W の内蔵 RAM を用いて任意のポート数のマルチポート RAM を実現する手法を説明する．

LVT による手法

任意のポート数のマルチポート RAM を実現する 1 つ目の方法として，LVT (Live Value Table) による手法 [11] を紹介する．

図 4.5 に，この手法で 3W2R のマルチポート RAM を実装した場合の回路を示す．中心的なハードウェアとして，マトリクス状に並んだメモリ・バンクと LVT

(Live Value Table) というテーブルがある．LVT という単語はテーブルと手法の両方を示しうるので，以後テーブルについては LVT （テーブル）と表記する．

メモリ・バンクはマルチポート RAM に書き込まれた値を保持するもので，メモリの実体となるハードウェアである．各メモリ・バンクは 1R1W の内蔵 RAM で構成され，そのビット幅とエントリ数はマルチポート RAM に要求されるものと同じである．メモリ・バンクはマトリクス状に配置されていて，任意のライト・ポートとリード・ポートの組について対応するメモリ・バンクが存在する．そのため，あるライト・ポートで書き込んだ値を，どのリード・ポートでも読み出すことができる．

LVT （テーブル）では，最新の値が存在するメモリ・バンクの場所を保持する．あるライト・ポートから入力した値は，それが接続されているメモリ・バンクにしか書き込むことができないため，残りのメモリ・バンクには古い値が記憶されていることになる．そこで値の書き込み時に，書き込みを行ったライト・ポートの番号を記憶しておく．読み出し時には，その情報を用いてメモリ・バンクから読み出した値のうち最新のものを選択する．最新の値がどのメモリ・バンクに存在するかはエントリごとに異なるため，LVT （テーブル）はマルチポート RAM に要求されるものと同じエントリ数を持つ必要がある．ポート数もマルチポート RAM のそれと等しくなるため，LVT （テーブル）は内蔵 RAM ではなく組み合わせ回路と D-FF で構成する．ただし，LVT （テーブル）のビット幅はライト・ポートを特定するために高々数ビットあればよいので，マルチポート RAM 全体を組み合わせ回路と D-FF で構成する場合と比べると資源の消費は小さくなる．

いま一度，書き込み時と読み出し時の回路の動作をまとめる．

書き込み　ライト・ポートに接続されているすべてのメモリ・バンクに値を書き込む．同時に，書き込んだライト・ポートの番号を LVT （テーブル）に書き込む．

読み出し　リード・ポートに対応するすべてのメモリ・バンクから値を読み出し，セレクタに入力する．LVT （テーブル）から読み出したライト・ポートの番号に基づいて，最新の値を選択する．

この回路の資源消費において支配的となるのは，ふつうはメモリ・バンクである．マルチポート RAM のリード・ポート数を p_r，ライト・ポート数を p_w とすると，メモリ・バンクの数は $p_r p_w$ である．一方，メモリ・バンク 1 つ 1 つの構成はポート数と相関が無い．よって，ポート数に対する資源消費のオーダーは $O(p_r p_w)$

となる．リード・ポート数とライト・ポート数が比例するときは，総ポート数 p に対して資源消費のオーダーは $O(p^2)$ となり，チップにおけるマルチポートメモリの面積と同様の性質を示す．

ただし，メモリ・バンクのビット幅が小さくなり，LVT （テーブル）のそれに近くなってくると，徐々に LVT （テーブル）の資源消費が支配的となる．LVT（テーブル）は組み合わせ回路と D-FF で構成されるので，こちらの資源消費が支配的となる場合は全体として資源の使用効率が悪くなる．

XOR による手法

マルチポート RAM を実現する方法の 2 つ目は，XOR によるもの [12] である．この方法は，基本は LVT による手法と変わらず，メモリ・バンクをマトリクス状に配置する．ただし，LVT （テーブル）や読み出し用のセレクタは存在しない．その代わりに，ビットごとの XOR 演算を行う回路が存在する．

読み込みおよび書き込みの際の具体的な動作を示す．

書き込み　書き込むデータと，自分以外のライト・ポートに対応するメモリ・バンクに書き込まれているデータとで，ビットごとの XOR 演算を行う．得られた値を，そのライト・ポートに対応するメモリ・バンクに書き込む．

読み出し　リード・ポートに対応するすべての メモリ・バンクから値を読み出し，ビットごとの XOR 演算を行う．得られた値が読み出し値となる．

書き込みの際，XOR 演算用にメモリ・バンクを読み出す必要があるため，そのためのメモリ・バンクを新たに追加する必要がある．

この方法がうまく機能する理由を，例を用いて説明する．図 4.6 は，3W2R のマルチポート RAM を実装した時のものである．W_0, W_1, W_2 の 3 つのライト・ポートが存在し，それぞれに対応するメモリ・バンクに w, x, y という値が書き込まれていたとする．いま，W_0 から z というデータを書き込むと，W_1, W_2 に対応するメモリ・バンクに書き込まれている値と XOR 演算を行うので，$x \oplus y \oplus z$ という値が得られる．この値が，W_0 に対応するメモリ・バンクに書き込まれる．ここで，リード・ポートから値を読み出そうとすると，W_0, W_1, W_2 に対応するメモリ・バンクに書き込まれている値について XOR 演算を行うことになる．ここで，XOR

演算の基本性質[2] [3] [4] を用いると，以下のように式変形できる．

$$(x \oplus y \oplus z) \oplus x \oplus y = (x \oplus x) \oplus (y \oplus y) \oplus z = 0 \oplus 0 \oplus z = z$$

つまり，過去に書き込み行ったライト・ポートに対応するメモリ・バンクに対応する値が 2 度 XOR 演算を行うことで打ち消されるので，必ず最新の書き込み値を得ることができるのである．

XOR による手法と LVT による手法のどちらの資源消費が小さくなるかは，おもにビット幅によって変わってくる．先ほど述べたとおり，LVT による手法ではビット幅が小さくなると LVT （テーブル）の資源消費が支配的になって非効率になる．一方，XOR による手法では LVT （テーブル）が存在しないためそのようなことは起こらない．しかし，ビット幅が大きい場合は，より多くのメモリ・バンクを必要とする XOR による手法が不利となる．よって，ビット幅が小さいと XOR による手法が有利に，ビット幅が大きいと LVT による手法が有利になる．

Live Value Table と XOR の組み合わせ

FF と組み合わせ回路で構成されていた LVT （テーブル）を，XOR による手法で構成すると，さらに資源消費を少なくすることができる．LVT （テーブル）の資源消費が大きくなるという LVT による手法の欠点と，メモリ・バンクが増えるという XOR による手法の欠点の両方をカバーできるからである．

原則として，この手法はもとの 2 つの手法よりも資源消費の面で優れているので，マルチポート RAM を構成する際はこれを用いればよい．ただし，マルチポート RAM のビット幅がごく小さい場合に限り，XOR による手法を用いる方が有利となる．これは，マルチポート RAM のビット幅が LVT （テーブル）のビット幅と同程度まで小さくなると，LVT （テーブル）の資源消費が全体の半分以上となり，無いほうがましという状態になるためである．

マルチポート CAM

本節ではこれまでマルチポート RAM について述べてきたが，最後にマルチポート CAM についても触れる．

最近の FPGA では，ハードウェアの CAM プリミティブは存在しないことが多い．雷上動の評価で使用した Spartan-6 [13] や Virtex-6 [14] でも，内蔵 RAM

[2] XOR 演算は交換可能: $A \oplus B = B \oplus A$
[3] 値 0 と XOR 演算を行っても，元の値から変化しない: $A \oplus 0 = A$
[4] 同じ値どうしで XOR 演算を行うと，結果は 0 になる: $A \oplus A = 0$

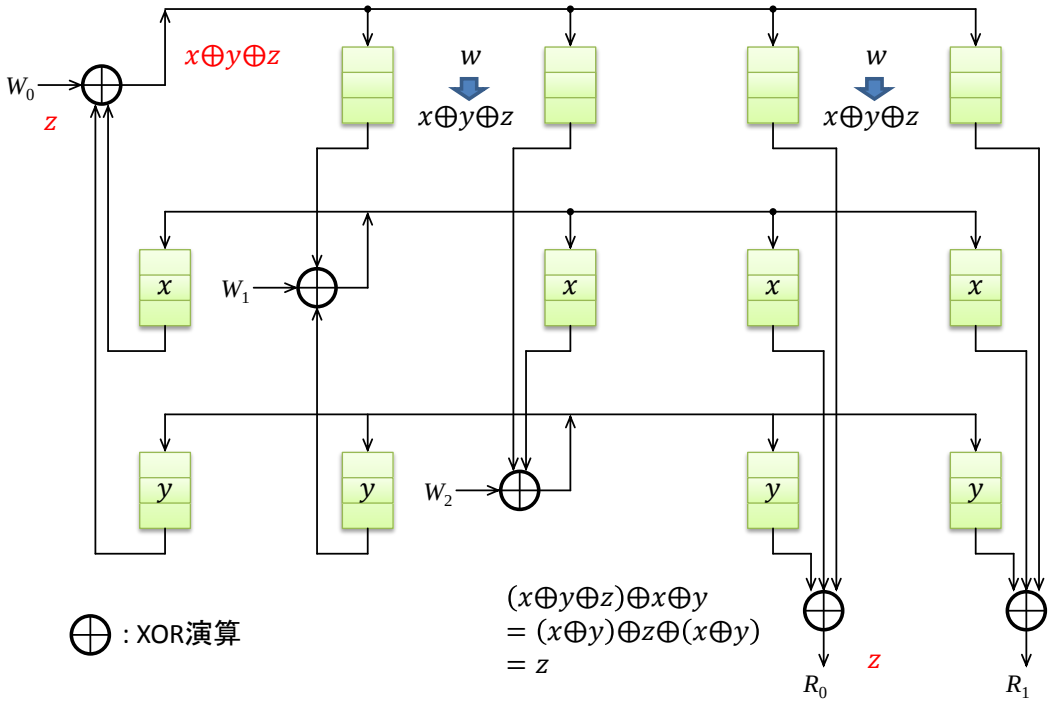

図 4.6: XOR によるマルチポート RAM の構成法

に対応する内蔵 CAM のようなものは無い．そのため，本節で取り上げたマルチポート RAM の構成法を CAM に応用して資源消費を抑える，ということは不可能である．

雷上動においてマルチポート CAM が必要な箇所では，組み合わせ回路と D-FF を用いて同等の回路を構成している．キーの値はすべて D-FF に格納されている．それを組み合わせ回路によって入力の値と比較している．D-FF の入力はセレクタ回路につながっていて，書き込むライト・ポートの値が選択される．D-FF の出力は比較回路につながっていて，入力された検索値と比較される．

このように構成されるマルチポート CAM の資源消費は，LVT による手法を用いたマルチポート RAM などとは異なる性質を持つ．書き込み側のセレクタ回路は，書き込みポート数 p_w に対し $O(p_w \log p_w)$ の資源を消費する．読み出し側の比較回路は，読み出しポート数 p_r に対し $O(p_r)$ の資源を消費する．D-FF の数自体はポート数に依存しないが，比較回路との配線は読み出しポート数が増えるにつれて大きくなる．

4.3.3 各マルチポートメモリの構成

表 4.3 と表 4.4 に，雷上動に含まれるマルチポートメモリの構成を示す．各表のポート数の欄は，バンク分け可能なものについてはバンク分け後のものを記している．たとえば，フリーリストの読み出し幅はリネーム幅 W_{rn} と等しいが，バンク分け可能であるため読出ポート数を 1 としている．ビット幅とエントリ数は，プロセッサのパラメータにより細かく変動するため，目安の値となっている．実装方法の欄は，4 章で述べた FPGA 上のマルチポートメモリの実装方法のうち，どれを用いたのかを示している．具体的な内容は以下の通りである．

- FF : 組み合わせ回路と D-FF を用いている．
- LUT : LUT-RAM を用いている．
- Block : Block RAM を用いている．
- XOR : XOR によるマルチポート RAM の構成法を用いている．
- MIX : LVT と XOR の組み合わせによるマルチポート RAM の構成法を用いている．

条件の欄は，プロセッサの構成に応じて有無が変わることを示している．具体的な内容は以下の通りである．

- MXS : マトリクス・スケジューラを採用する場合のみ存在する．
- 非 MXS : マトリクス・スケジューラを採用しない場合のみ存在する．
- 非 SROB : 分離リオーダ・バッファを採用しない場合のみ存在する．
- NORCS: 非レイテンシ指向 レジスタ・キャッシュ・システムを採用する場合のみ存在するもの．
- 非 NORCS: 非レイテンシ指向 レジスタ・キャッシュ・システムを採用しない場合のみ存在するもの．
- SIMD : SIMD を採用する場合のみ存在するもの．

各マルチポートメモリの実装方法は，ポート数やビット幅などの条件を鑑みてふさわしいものを選択している．RAM の場合，エントリ数が多いものは Block RAM を採用し，そうでないものは LUT-RAM を採用している．これは， Block RAM の方が LUT-RAM よりエントリ数が多いことをふまえたうえで，FPGA の資源を効率よく利用しようと意図した結果である．たとえば Xilinx 社の Virtex-6 [14]

では，1 つの Block RAM プリミティブは最低 512 エントリであるのに対し，1 つの LUT-RAM プリミティブは最高 64 エントリである．マルチポート RAM を構成する場合，ビット幅が数 bit 以下と非常に小さい場合は XOR ベースの手法を利用し，それ以外の場合は LVT と XOR を組み合わせた手法を採用している．

4.4 開発環境

本節では雷上動の開発環境について説明する．雷上動の開発環境の特徴は大きく分けて 2 つある．1 つ目は，ハードウェア記述言語として Verilog の後継言語である SystemVerilog を採用し，高品質なコード記述を実現したことである．2 つ目は，命令パイプラインの可視化ツールをデバッグに活用したことである．4.4.1 節と 4.4.2 節でこれらについて説明する．

開発に使用したソフトウェアを表 4.5 に示す．シミュレーションや合成を行うソフトウェアには，SystemVerilog への対応が進んでいる点を重視し Mentor Graphics QuestaSim と Synopsis Synplify を使用している．最終的に Xilinx 社の FPGA で動作させるため，配置配線ソフトウェアには同社の ISE Design Suite を使用している．また，命令パイプライン可視化に Kanata を，テスト・プログラムのコンパイルに GNU GCC を使用している．

4.4.1 SystemVerilog による可読性/再利用性の高いコーディング

雷上動 は Verilog から SystemVerilog にバージョンアップする過程で導入された新しい構文を活用し，高品質なコード記述を実現している．構造体や共用体，インターフェースなどが新しい構文の例として挙げられる．

筆者が特に強調したいのが，インターフェース構文の効用である．インターフェース構文とは，複数のモジュール間を接続する信号線群をまとめて扱う構文である．インターフェースを定義するときは，信号線群を宣言した後，それらが各モジュールに対してどう接続されているか（入力/出力など）を指定する．図 4.7 にモジュール間接続の模式図を示す．インターフェース構文を利用することで，モジュール間接続を整理して表現できることがわかる．

インターフェース構文がモジュール間の接続を抽象化するという点が，雷上動のコード記述を簡潔にする上で重要な役割を果たしていると筆者は考えている．スーパースカラ・プロセッサは多数のモジュールが密結合した構造をとっており，本質

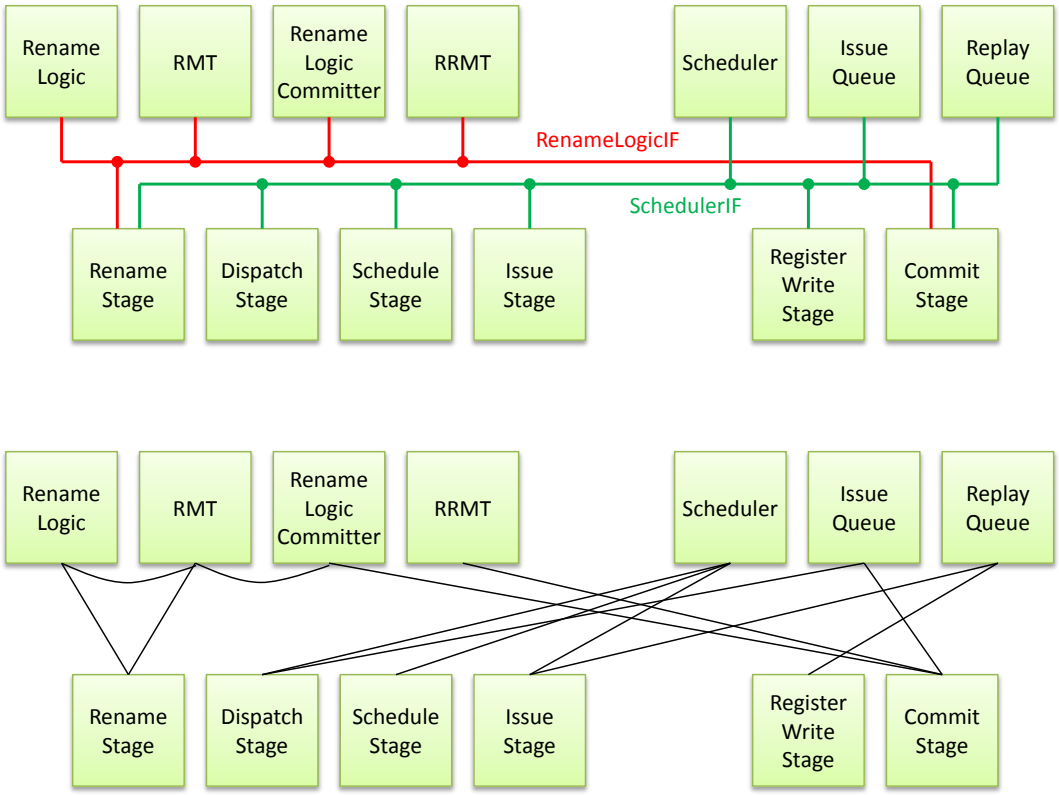

図 4.7: モジュール間の接続の模式図

インターフェースを利用する場合（上）と，利用しない場合（下）

的に複雑なハードウェアである．モジュール同士が密結合する場合は，信号線の種類がおおよそモジュール数の 2 乗に比例するため，構造体や共用体の利用ではコードの複雑さを抑え切ることが困難である．一方，インターフェース構文はモジュール間の接続という単位で抽象化を行う．インターフェースを用いれば，モジュール間を接続するコードは簡潔になる．各インターフェースの定義は，同じグループに属する信号線の集合についての記述なので，理解は容易である．よってインターフェースを用いると，多数のモジュールが密結合した複雑なハードウェアを，理解しやすいコードの集合で表現できる．

4.4.2 パイプライン可視化による柔軟性の高い開発スタイル

雷上動の開発時には，プロセッサ・シミュレータ鬼斬 [15] 用に開発されたパイ

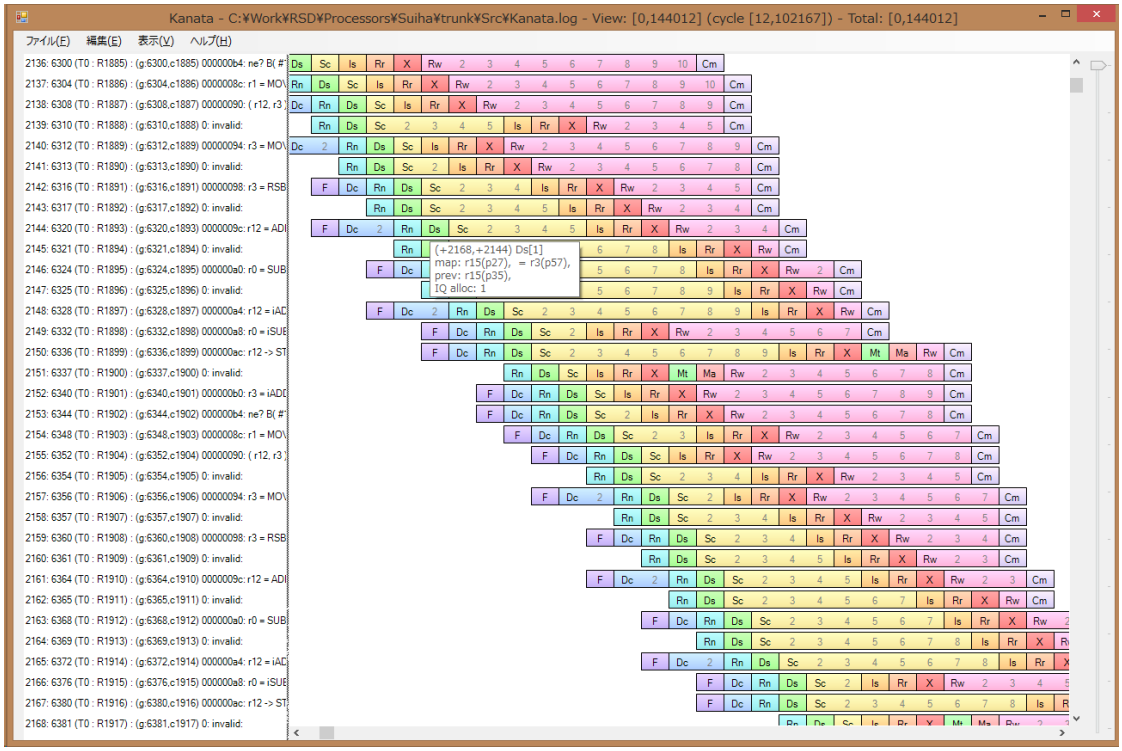

図 4.8: パイプライン可視化ツール Kanata

プライン可視化ツール Kanata を移植して使用した．図 4.8 に Kanata による雷上動の実行の様子を示す．Kanata では命令パイプラインの状態を可視化し，各種資源の確保や解放，リネーミングの結果などをオーバーレイして表示可能である．

Kanata はハードウェアのデバッグ速度を大きく向上させた．Out-of-order スーパースカラ・プロセッサの開発中は，特定の命令列を特定の順序で実行した場合にのみ起こるバグに頻繁に遭遇する．Kanata を用いると複数の命令がパイプライン処理されている様子を俯瞰できるため，すばやくバグの原因にあたりをつけることができる．

雷上動では新しいアーキテクチャを次々と導入するため柔軟性の高い開発スタイルが求められたが，Kanata によりそれが実現できたと筆者は考えている．

表 4.3: プロセッサ内のマルチポート RAM の構成

名称	ポート数 (R/W)	エントリ幅	エントリ数	実装方法	条件
命令キャッシュ	1/1	80-160 bit	1K 程度	Block	
BTB	1/1	28-36 bit	1K 程度	Block	
RMT	$3W_{rn}/W_{rn}$	5-7 bit	32	LUT, MIX	
フリーリスト	1/1	5-7 bit	32-256	LUT	
アクティブ・リスト					
データ	1/1	80-120 bit	32-256	LUT	
実行状態	$W_{cm}/W_{rn}+W_{is}$	2 bit	32-256	LUT, XOR	
分岐先	W_{cm}/W_{int}	32 bit	32-256	LUT, MIX	非 SROB
WAT	$2W_{rn}/W_{rn}$	4 bit	32	LUT, MIX	MXS
スケジューラ					
Payload (int)	W_{ds}/W_{int}	100-200 bit	16-32	LUT, MIX	
Payload (cpx)	W_{ds}/W_{cpx}	100-200 bit	16-32	LUT, MIX	
Payload (mem)	W_{ds}/W_{mem}	100-200 bit	16-32	LUT, MIX	
レディ・ビット	$2W_{ds}/W_{ds}+W_{is}$	1 bit	64-256	LUT, XOR	
Dest. RAM	W_{is}/W_{ds}	10-14 bit	16-32	LUT, MIX	
Prod. matrix	1/1	8-32 bit	16-32	FF	MXS
リプレイ・キュー	1/1	400-800 bit	16-32	LUT	
レジスタ・ファイル					
汎用	$2W_{is}/W_{is}$	32 bit	32-128	LUT, MIX	非 NORCS
SIMD	$2W_{is}/W_{is}$	128 bit	32-128	LUT, MIX	SIMD
Main reg. file	$2/W_{is}$	32 bit	32-128	LUT, MIX	NORCS
Reg. cache	2/1	32 bit	8-16	LUT, MIX	NORCS
ストア・キュー					
データ・アレイ	$W_{ld}+1/W_{st}$	128-144bit	16-32	LUT, MIX	
データ・キャッシュ					
タグ・アレイ	2 (Shared)	32-48 bit	1K 程度	Block	
データ・アレイ	2 (Shared)	128 bit	1K 程度	Block	

W_{rn}: リネーム幅 / W_{ds}: ディスパッチ幅 / W_{is}: 発行幅 / W_{cm}: コミット幅
W_{int}: 整数演算命令発行幅 / W_{cpx}: 乗算および SIMD 命令発行幅
W_{mem}: メモリアクセス命令発行幅 / W_{ld}: ロード命令発行幅 / W_{st}: ストア命令発行幅

表 4.4: プロセッサ内のマルチポート CAM の構成

名称	ポート数 (R/W)	エントリ幅	エントリ幅	実装方法	条件
スケジューラ					
ソース CAM	1/1	80-160 bit	24-96	FF	非 MXS
ロード・キュー	W_{st}/W_{ld}	32-48bit	16-32	FF	
ストア・キュー					
タグ・アレイ	W_{ld}/W_{st}	32-48bit	16-32	FF	
MSHR	1/1	5-7 bit	2-4	FF	

記号の意味は、表 4.3 の注釈参照

表 4.5: 開発に使用したソフトウェア

用途	ソフトウェア
HDL シミュレーション	Mentor Graphics QuestaSim 10.2c
論理合成	Synopsys Synplify Premier H-2013.03-SP1-1
配置配線	ISE Design Suite 14.6
パイプライン可視化	Kanata 1.24
コンパイラ	GNU GCC 4.8.2

第5章

評価

本章では雷上動の評価について述べる．5.1 節では，面積効率を向上させる技術の効果を確認するため，回路面積の代わりに FPGA で使用した LUT の数を評価する．5.2 節では，雷上動で動作するプログラムについて述べ，プロセッサとして動作することが確認できていることを示す．

5.1 使用 LUT 数の評価

5.1.1 評価環境

評価に用いたプロセッサのパラメータを表 5.1 に示す．パラメータ・セットとして 5-issue と 8-issue の 2 種類が存在するが，これは発行幅が 5 および 8 のプロセッサにおいて平均的に用いられる値を念頭に設定したものである．

使用 LUT 数は，論理合成ソフトウェアである Synplify および ISE のログファイルより求めている．論理合成時の設定を表 5.2 に示す．ここでターゲットとしている FPGA は使用可能資源が非常に多いハイエンド品であり，資源不足によって合成が停止するという事態は今回は発生しなかった．

5.1.2 モジュール単位での使用 LUT 数の評価

マトリクス・スケジューラに関するモジュールの使用 LUT 数

図 5.1 に，ウェイクアップ・ロジックおよび WAT の使用 LUT 数を示す．マトリクス・スケジューラを採用する/しない場合と，発行キューのエントリ数を 8, 16,

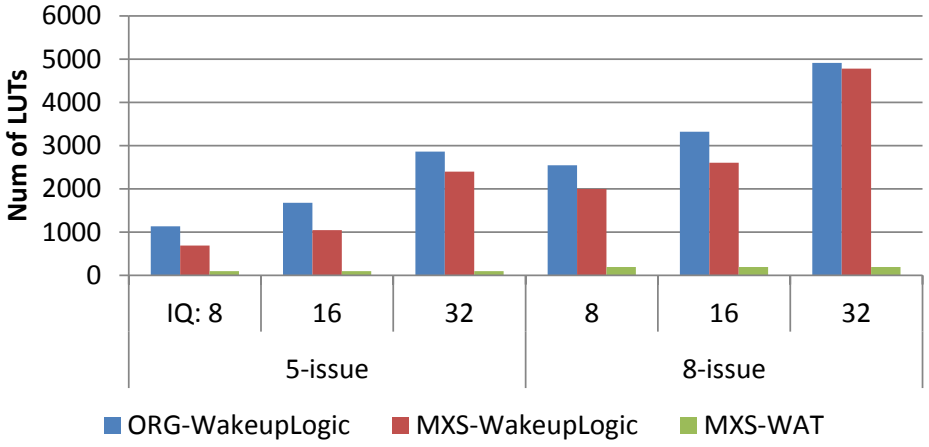

図 5.1: マトリクス・スケジューラに関係するモジュールの使用 LUT 数

32 と変化させた場合，その他のパラメータを表 5.1 の 5-issue および 8-issue に設定した場合について比較を行っている．略称の意味は以下のとおりである．

- ORG-WakeupLogic : CAM 式ウェイクアップ・ロジックの使用 LUT 数
- MXS-WakeupLogic : マトリクス・スケジューラのウェイクアップ・ロジックの使用 LUT 数
- MXS-WAT : WAT の使用 LUT 数

WAT はマトリクス・スケジューラを採用した場合のみ必要なモジュールなので，マトリクス・スケジューラを採用した場合は WAT の使用 LUT 数を加算して比較する必要がある．

図 5.1 より，マトリクス・スケジューラの採用によって使用 LUT 数が削減できていることが確認できる．ただし，8-issue / 32 entries の場合については，マトリクス・スケジューラのウェイクアップ・ロジックと WAT の使用 LUT 数の合計が，CAM 式ウェイクアップ・ロジックの使用 LUT 数とほぼ変わらない．これは，CAM 式スケジューラの回路規模は発行キューのエントリ数に比例するのに対し．

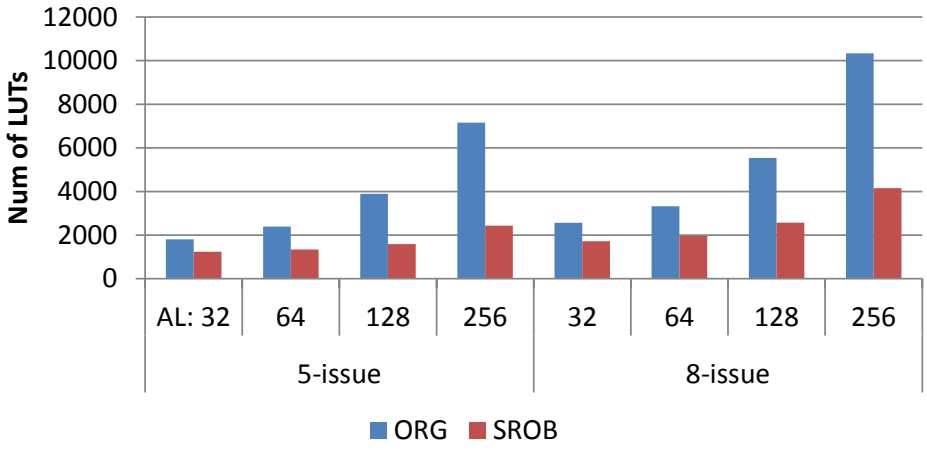

図 5.2: 分離リオーダ・バッファに関係するモジュールの使用 LUT 数

マトリクス・スケジューラ のマトリクスの回路規模が発行キューのエントリ数の 2 乗に比例するからである．発行キューのエントリ数がより増えるとマトリクス・スケジューラが不利になると考えられるが，実際は命令ウィンドウの非集中化によって発行キューのエントリ数を低く抑えることができる．

分離リオーダ・バッファに関するモジュールの使用 LUT 数

図 5.2 に，アクティブ・リスト の使用 LUT 数を示す．分離リオーダ・バッファを採用する/しない場合と，アクティブ・リストのエントリ数を 32, 64, 128, 256 と変化させた場合，その他のパラメータを表 5.1 の 5-issue および 8-issue に設定した場合について比較を行っている．略称の意味は以下のとおりである．

- ORG : 分離リオーダ・バッファを採用しない場合
- SROB : 分離リオーダ・バッファを採用した場合

いずれの場合も，分離リオーダ・バッファの採用によって使用 LUT 数が大きく減少することがわかる．これは，期待した通りの面積削減効果が表れているため

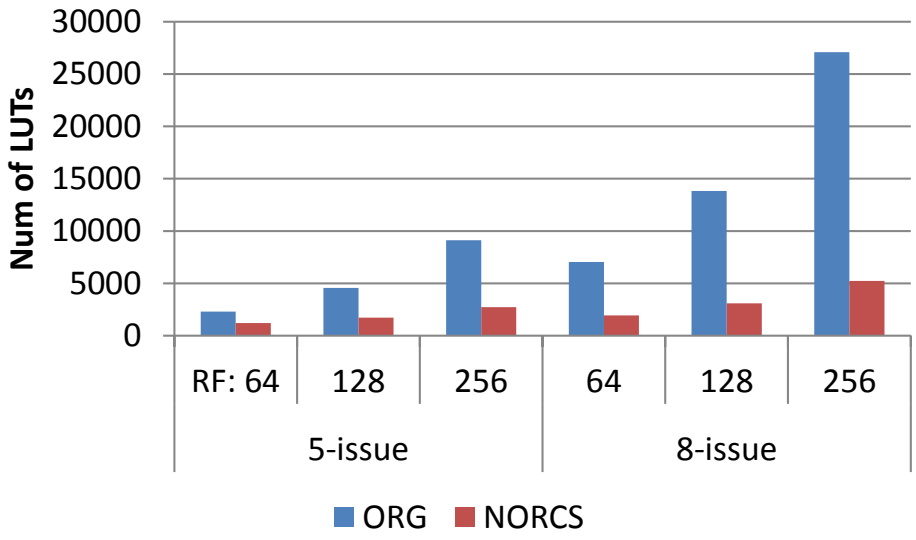

図 5.3: 非レイテンシ指向 レジスタ・キャッシュ・システムに関係するモジュールの使用 LUT 数

と考えられる．

非レイテンシ指向 レジスタ･キャッシュ･システムに関するモジュールの使用 LUT 数

図 5.3 に，非レイテンシ指向 レジスタ・キャッシュ・システムに関するモジュールの使用 LUT 数を示す．非レイテンシ指向 レジスタ・キャッシュ・システムを採用する/しない場合と，レジスタ・ファイルのエントリ数を 64, 128, 256 と変化させた場合，その他のパラメータを表 5.1 の 5-issue および 8-issue に設定した場合について比較を行っている．略称の意味は以下のとおりである．

- ORG : 非レイテンシ指向 レジスタ・キャッシュ・システムを採用しない場合の，レジスタ・ファイルの使用 LUT 数
- NORCS : 非レイテンシ指向 レジスタ・キャッシュ・システムを採用した場合の，レジスタ・キャッシュ・システムの使用 LUT 数
  - レジスタ・キャッシュ・システムには，レジスタ・キャッシュとメイン・レジスタ・ファイルおよびそれらのアクセスを調停する回路を含む．

非レイテンシ指向 レジスタ・キャッシュ・システムの採用によって使用 LUT 数が大きく減少することがわかる．ただし，4.2.12 節で述べたように，今回の実装では既存の評価 [5] とは異なる構成を用いて面積の削減を図っている．そのため，この構成で大きな性能低下が発生しないか，標準的なベンチマークを用いて確かめる必要がある．

5.1.3 プロセッサ全体の使用 LUT 数の評価

図 5.4 と図 5.5 に，雷上動プロセッサ全体の使用 LUT 数を示す．図 5.4 と図 5.5 は，プロセッサのパラメータを表 5.1 の 5-issue と 8-issue にそれぞれ設定している．SIMD 用のレジスタ・ファイルとバイパスネットワーク，および演算器は省略している．各図において，面積効率を向上させる技術を ON/OFF した場合で，使用 LUT 数がどう変化するのか比較を行っている．略称の意味は以下のとおりである．

- ORG : 面積効率を向上させる技術を一切適用しない場合
- MXS : マトリクス・スケジューラのみを適用した場合
- SROB : 分離リオーダ・バッファのみを適用した場合
- NORCS : 非レイテンシ指向 レジスタ・キャッシュ・システムのみを適用した場合
- FULL : 上記の面積効率を向上させる技術をすべて適用した場合

面積効率を向上させる技術によって，プロセッサ全体の使用 LUT 数を削減できていることが確認できる．マトリクス・スケジューラは図中の Scheduler の部分，分離リオーダ・バッファは図中の CommitLogic の部分，非レイテンシ指向 レジスタ・キャッシュ・システムは図中の RF and Bypass の部分に対し，それぞれ使用 LUT 数を削減する効果を発揮している．相対的には非レイテンシ指向 レジスタ・キャッシュ・システムの効果が大きく，マトリクス・スケジューラの効果は小さいことがわかる．

5.1.4 使用 LUT 数と回路面積の関係についての考察

FPGA における使用 LUT 数が，プロセッサをチップにした場合の面積とどのような関係にあるかについて考察を行う．FPGA とチップでは回路の構成方法が

大きく違う．そのため，使用 LUT 数の削減量とチップ上での面積削減効果は比例しないが，両者がおおむね正の相関を持つのは確かである．

全体として，FPGA ではロジック部分の面積が過大に評価され，メモリ部分の面積は過小に評価される．その理由は，ロジック部分は論理関数のコンフィギュレーションで実現されるのに対し，メモリ部分は既にハードウェアとして FPGA 上に実装されているものを使用可能だからである．よって，チップではメモリ部分の面積が相対的に大きくなる．これは，マルチポートメモリの面積に着目した面積効率を向上させる技術の効果が，より大きくなることを意味する．

マルチポート RAM の構成方法も FPGA とチップで大きく異なるが，使用 LUT 数と回路面積はマルチポート RAM のパラメータに対して似たような関数となる．すなわち，どちらもおおむねポート数の 2 乗とエントリ数の積に比例する．よって，マルチポート RAM の構成方法が異なるといえど，分離リオーダ・バッファや非レイテンシ指向 レジスタ・キャッシュ・システムなどの効果はチップでも同様に現れると言える．

マトリクス・スケジューラに関係するモジュールはやや複雑である．マトリクス・スケジューラに含まれるプロデューサ・マトリクスと，CAM 式ウェイクアップ・ロジックに含まれるソース CAM は，FPGA においてはメモリではなく組み合わせ回路と D-FF で実現される．そのため，FPGA においてこれらの面積は過大に評価されており，チップ上ではより小さくなると考えられる．ただし，ソース CAM の使用 LUT 数は発行幅に比例するのに対し，チップ上での面積は発行用ポートを含むポート数の 2 乗に比例するため，この点でソース CAM の面積は FPGA 上で過小評価されている．ゆえに，チップ上ではマトリクス・スケジューラが相対的に優位に立つ．

まとめを述べる．FPGA の使用 LUT 数とチップ上での面積は比例するものではないので，評価結果からチップ上の面積を定量的に予測することは困難である．しかし，面積効率を向上させる技術の効果は FPGA よりもチップ上での方が大きくなると予測できる材料が多い．面積効率を向上させる技術の有用性は，これまで以上に説得力のある形で示されたと考えられる．

5.2 プロセッサとしての動作確認

筆者は，雷上動にメモリや IO を組み合わせたシステムを構築し，プログラムを実行できることを確認している．システムは Digilent Atlys Spartan-6 FPGA Developement Board 上に構築した．このボードの写真を図 5.6 に，スペックを表 5.3 に示す．このシステムは USB ケーブルで PC と接続して通信を行うことができ，PC からシステムへのプログラムの転送と，システムから PC への実行結果の送信が可能になっている．

なお，このシステムはもともと情報処理学会が主催したプロセッサ・コンテストである The 1st IPSJ SIG-ARC High-Performance Processor Design Contest に参加するために構築したものである．使用するボードや PC との通信プロトコルがコンテスト側で規定されていたため，それに合わせてシステムを構築している．

このシステムで利用した FPGA は比較的安価なミドルレンジ製品にあたり，LUT などの資源の数はそれほど多いわけではない．そのような FPGA に out-of-order スーパースカラ・プロセッサを実装できたということ自体が，雷上動の面積効率の高さを示していると考えられる．

以下にこのシステムで動作を確認したプログラムの例を示す．

- Dhrystone ベンチマーク
- ソート（クイックソートおよび基数ソート）
- 密行列積
- ステンシル計算
- 最短経路問題（ダイクストラ法）

Dhrystone ベンチマーク以外は，プロセッサ・コンテストの課題として出題されたものである．4.1 節で述べたように雷上動は浮動小数点演算に対応していないが，上記のプログラムはすべて浮動小数点変数を用いない整数プログラムである．浮動小数点演算の他にも現在の雷上動には未対応命令が多く，それゆえ動作するプログラムも制限される．上記のように多様な整数プログラムが動作しているとはいえ，SPEC CPU2006 のようなある程度大規模なベンチマークは動作を確認できていない．

表 5.1: 評価に用いたプロセッサのパラメータ

Name	Parameter (5-issue)	Parameter (8-issue)
fetch width	2 inst.	3 inst.
rename width	2 micro-ops.	3 micro-ops.
issue width	5 micro-ops.	8 micro-ops.
commit width	3 micro-ops.	6 micro-ops.
exec units	2 int / 1 complex int	3 int / 3 complex int
	1 load / 1store	1 load / 1store
issue queue	16 entries	16 entries
active list	64 entries	128 entries
load / store queue	16 / 16 entries	32 / 32 entries
miss status handling register	2 entries	4 entries
physical register file (general)	64 entries	128 entries
physical register file (flag)	32 entries	32 entries
branch pred.	2-bit saturating counter,	
	8K entries PHT, 1K entries BTB	
L1C(I)	8 KB, direct-mapped, 16 B/line	
L1C(D)	32 KB, direct-mapped, 16 B/line	

表 5.2: 論理合成時の設定

Technology	Xilinx Virtex-6
Device	XC6VLX760
Package	FF1760
Speed Grade	-2
Optimization	Default

表 5.3: 動作確認に使用した FPGA ボードのスペック

FPGA	Xilinx Spartan-6 LX45
Memory	DDR2-SDRAM 128 MB, 333 MHz
IO	USB-UART etc.

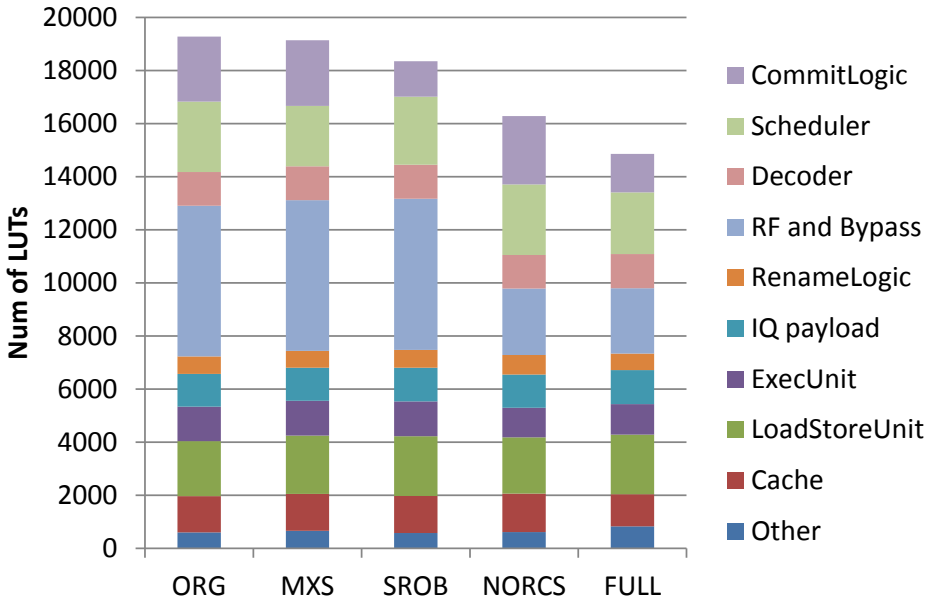

図 5.4: プロセッサ全体の使用 LUT 数 (5-issue)

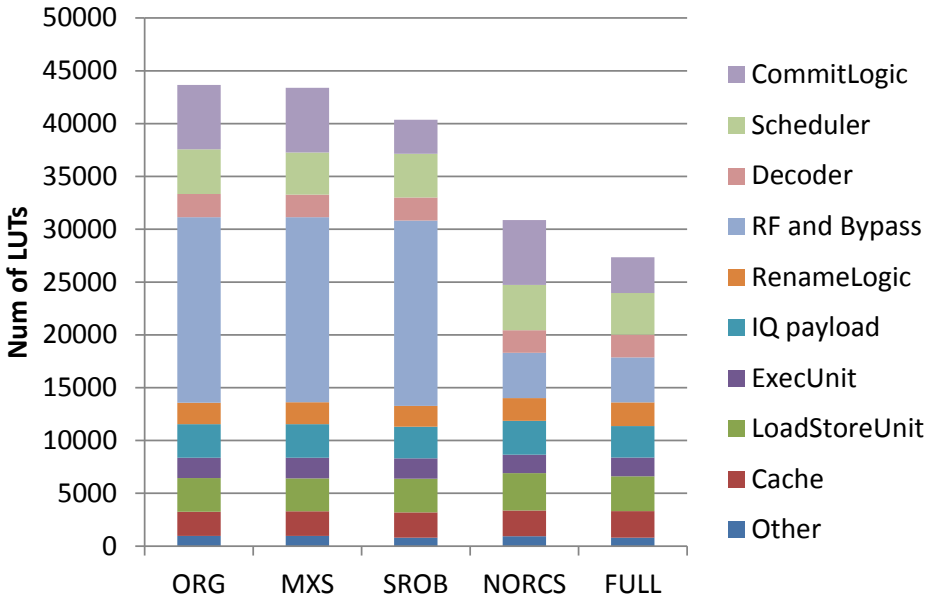

図 5.5: プロセッサ全体の使用 LUT 数 (8-issue)

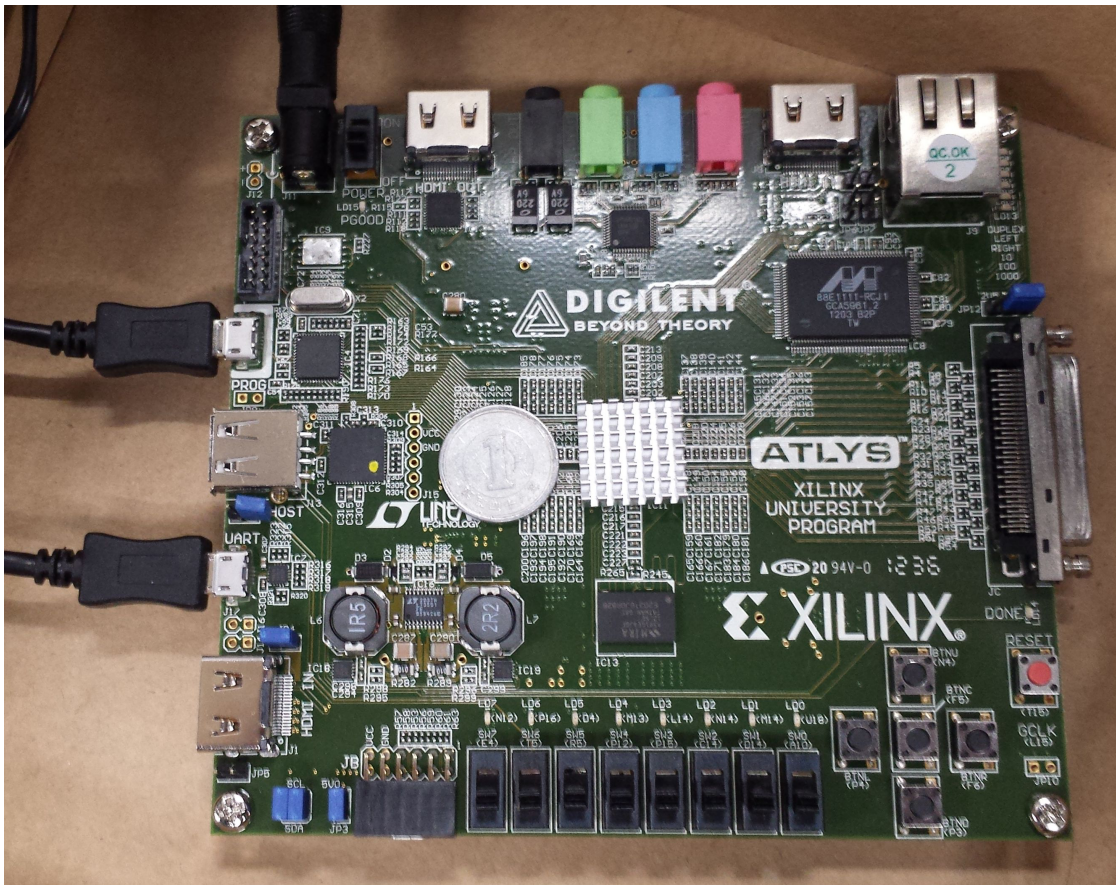

図 5.6: 動作確認に使用した FPGA ボード

第6章

関連研究

本章では雷上動に関連する研究について述べる．まず，雷上動の先行研究について述べる．面積効率を指向するプロセッサの設計と実装に関する研究は，本研究とこの先行研究を除いて存在しない．続いて， FPGA 上で動作するソフトコア・プロセッサの研究について述べる．これらの研究はいずれも面積効率という観点には立っていない．しかし，雷上動も FPGA 上で動作するプロセッサの一種であるため，そのようなプロセッサの研究は比較対象として重要である．

先行研究

堀尾は，マトリクス・スケジューラ [1,2] や非レイテンシ指向 レジスタ・キャッシュ・システム [4,5] などの技術を統合し，面積効率を指向するプロセッサの設計を行った [16]．堀尾は上記の 2 つの他に，以下の 3 つの技術の統合を試みた．

リネームド・トレース・キャッシュ　リネーム済の命令をキャッシュすることで，リネーム・ロジックに必要な面積を大幅に縮小する技術 [5]．

ツインテール・アーキテクチャ　バックエンドに単純な in-order 実行系を設け，通常の out-of-order 実行系の消費エネルギーを減らしつつ，性能を向上させる技術 [17]．

NoSQ　従来の CAM 式ロード・ストア・キューを用いずにロード/ストア命令の out-of-order 実行を実現することで，消費エネルギーを削減する技術 [18]．

堀尾は，リネームド・トレース・キャッシュとツインテール・アーキテクチャを単純に統合することは不可能であると論じ，そのためにフューチャ・バッファというハードウェアを新たに提案した．ただし，堀尾の研究は実装を進めるまでには至

らなかった．

雷上動では堀尾の設計とは独立に設計を行っており，実装と評価を行った点でも新規的である．

堀尾が提案したフューチャ・バッファは本研究では扱わないが，今後雷上動の設計と実装で用いられる可能性がある．現在，雷上動に以下の 2 つの技術を組み込む研究が，それぞれ進行中である．

ディスパッチト・イメージ・キャッシュ　ディスパッチ済の命令をキャッシュすることで，リネーム・ロジックとディスパッチ・ネットワークに必要な面積を大幅に縮小する技術 [19]．

フロントエンド実行　フロントエンドに設けた in-order 実行系で演算を実行することで，バックエンドの out-of-order 実行系の消費エネルギーを減らしつつ，性能を向上させる技術 [20]．

ディスパッチト・イメージ・キャッシュはリネームド・トレース・キャッシュから，ツインテール・アーキテクチャはフロントエンド実行から派生したそれぞれ研究であり，共通点も多い．将来的には，雷上動においてディスパッチト・イメージ・キャッシュとフロントエンド実行を統合する予定であり，その際にフューチャ・バッファを用いる可能性がある．

FPGA 上で動作するソフトコア・プロセッサ

FPGA 上で動作するソフトコア・プロセッサについては，多くの製品や研究が存在する．

FPGA ベンダが，各社の FPGA 上で動作するソフトコア・プロセッサを提供している．Altera 社の Nios II [21] や Xilinx 社の MicroBlaze [22] がそれに該当する．これらはいずれも in-order スカラ・プロセッサである．

チップの形で製造されているプロセッサを，ソフトコア・プロセッサの形にして FPGA 上で動作させたという研究も存在する．Intel 社の グループが，in-order の Atom プロセッサのコアや out-of-order の Nehalem プロセッサのコアを FPGA 上に実装している [23, 24]．Atom のコアは 1 つの Virtex-4 [25] 上に実装されているが，Nehalem のコアは Virtex-4 と Virtex-5 [26] を計 5 つ使用する巨大なものになっている．

FabScalar [27, 28] は out-of-order スーパースカラのソフトコア・プロセッサで，プロセッサのパラメータを簡単に設定できるという特徴を持つ．このような機能

は雷上動でもサポートしており，現存するソフトコア・プロセッサの中で最も雷上動に近いものであるといえる．

第7章

結論

本論文のまとめ

本論文では，面積効率を指向した out-of-order スーパースカラ・プロセッサ「雷上動」の設計と実装について述べた．雷上動は，我々の研究グループで提案してきた面積効率を向上させる技術の効果を実証するために開発されているものである．本論文では，それらの技術のうちマトリクス・スケジューラと分離リオーダ・バッファおよび非レイテンシ指向 レジスタ・キャッシュ・システムを選択して雷上動の上に実装した．

FPGA 向けに合成した結果，面積効率を向上させる技術によって使用 LUT 数を削減できることがわかった．FPGA における資源消費量の削減という効果は，チップにおいては回路面積の削減という形で現れると期待できる．

今後の課題

筆者は，今後の課題が大きく分けて 2 つ存在すると考えている．

1 つ目は，面積や性能の評価をより充実させることである．本論文では FPGA の資源消費量によって評価を行ったが，実際に即した評価を行うためにはチップとして実装することが不可欠である．また，現状の雷上動には未対応命令が多く，SPEC CPU2006 のような標準的なベンチマークによる性能比較を行うことが出来ない．これらに対応していくことは，評価に説得力を持たせるという点で重要である．

2 つ目は，雷上動に新たなアーキテクチャ技術を導入していくことである．面積効率を向上させる技術は今回採用したもの以外にも存在する．それらをすべて組み込んだ非常に面積効率の高いプロセッサを実現することが，我々のグループで

行ってきた一連の研究のゴールである．また，我々は面積効率に関するもの以外にも様々なアーキテクチャ技術を提案しており，それらも雷上動で実装することを計画している．将来的には，様々なアーキテクチャ技術を試験するプラットフォームとして，雷上動を活用したい．

参考文献

[1] 五島正裕: Out-of-Order ILP プロセッサにおける命令スケジューリングの高速化の研究, 博士論文, 京都大学 大学院情報学研究科 (2004).

[2] Sassone, P. G., Rupley, II, J., Brekelbaum, E., Loh, G. H. and Black, B.: Matrix Scheduler Reloaded, *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 335–346 (2007).

[3] 岩原佑磨, 安藤秀樹: リオーダ・バッファのハードウェア量と消費エネルギーの削減, 先進的計算基盤システムシンポジウム SACSIS 2010, pp. 37–44 (2010).

[4] Shioya, R., Horio, K., Goshima, M. and Sakai, S.: Register Cache System not for Latency Reduction Purpose, *Proceedings of the International Symposium on Microarchitecture*, pp. 301–312 (2010).

[5] 塩谷亮太: 面積効率を指向するプロセッサの研究, 博士論文, 東京大学大学院情報理工学系研究科 (2011).

[6] Krewell, K.: Intel's Haswell Cuts Core Power, *Microprocessor Report 9/24/12*, pp. 1–5 (2012).

[7] Kessler, R.: The Alpha 21264 microprocessor, *IEEE Micro*, Vol. 19, No. 2, pp. 24–36 (1999).

[8] Chrysos, G. and Emer, J.: Memory dependence prediction using store sets, *Proceedings of the International Symposium on Computer Architecture*, pp. 142 –153 (1998).

[9] McFarling, S.: Combining Branch Predictors, Technical report, Digital Equipment Corporation Western Research Lab (1993).

[10] Yeager, K.: The MIPS R10000 Superscalar Microprocessor, *IEEE Micro*, Vol. 16, No. 2, pp. 28–41 (1996).

[11] LaForest, C. E. and Steffan, J. G.: Efficient multi-ported memories for FPGAs, *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '10, New York, NY, USA, ACM, pp. 41–50 (2010).

[12] Laforest, C. E., Liu, M. G., Rapati, E. R. and Steffan, J. G.: Multi-ported Memories for FPGAs via XOR, *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, New York, NY, USA, ACM, pp. 209–218 (2012).

[13] Xilinx Inc.: *Spartan-6 Family Overview v2.0* ***http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf***. Accessed: 2013-09-08.

[14] Xilinx Inc.: *Virtex-6 Family Overview v2.4* ***http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf***. Accessed: 2013-09-08.

[15] 塩谷亮太, 五島正裕, 坂井修一: プロセッサ・シミュレータ「鬼斬弐」の設計と実装, 先進的計算基盤システムシンポジウム SACSIS 2009, pp. 120–121 (2009).

[16] 堀尾一生: 面積効率を指向するプロセッサの設計, 修士論文, 東京大学大学院情報理工学系研究科 (2010).

[17] 堀尾一生, 亘理靖展, 塩谷亮太, 五島正裕, 坂井修一: ツインテール・アーキテクチャの評価, 情報処理学会研究報告 2008-ARC-171, pp. 7–12 (2008).

[18] Sha, T., Martin, M. and Roth, A.: NoSQ: Store-Load Communication without a Store Queue, *Proceedings of the International Symposium on Microarchitecture*, pp. 285–296 (2006).

[19] 伊達三雄: フロントエンド・パイプラインを最小化する命令キャッシュ・アーキテクチャ, 修士論文, 東京大学大学院 情報理工学系研究科 (2013).

[20] 鷹見怜, 塩谷亮太, 安藤秀樹: フロントエンドで命令を実行するプロセッサにおけるエネルギー効率の評価, 情報処理学会研究報告 2013-ARC-206, pp. 1–10 (2013).

[21] Altera Corporation: *Nios II Processor Reference Handbook Version 11.0.0* *http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf*. Accessed: 2013-09-08.

[22] Xilinx Inc.: *MicroBlaze Processor Reference Guide v9.0* *http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf*. Accessed: 2013-09-08.

[23] Wang, P. H., Collins, J. D., Weaver, C. T., Kuttanna, B., Salamian, S., Chinya, G. N., Schuchman, E., Schilling, O., Doil, T., Steibl, S. and Wang, H.: Intel atom processor core made FPGA-synthesizable, *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '09, New York, NY, USA, ACM, pp. 209–218 (2009).

[24] Schelle, G., Collins, J., Schuchman, E., Wang, P., Zou, X., Chinya, G., Plate, R., Mattner, T., Olbrich, F., Hammarlund, P., Singhal, R., Brayton, J., Steibl, S. and Wang, H.: Intel nehalem processor core made FPGA synthesizable, *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '10, New York, NY, USA, ACM, pp. 3–12 (2010).

[25] Xilinx Inc.: *Virtex-4 FPGA User Guide v2.6* *http://www.xilinx.com/support/documentation/user_guides/ug070.pdf*. Accessed: 2013-09-08.

[26] Xilinx Inc.: *Virtex-5 FPGA User Guide v5.4* *http://www.xilinx.com/support/documentation/user_guides/ug190.pdf*. Accessed: 2013-09-08.

[27] Choudhary, N. K., Wadhavkar, S. V., Shah, T. A., Mayukh, H., Gandhi, J., Dwiel, B. H., Navada, S., Najaf-abadi, H. H. and Rotenberg, E.: FabScalar: composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template, *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, New York, NY, USA, ACM, pp. 11–22 (2011).

[28] Choudhary, N., Wadhavkar, S., Shah, T., Mayukh, H., Gandhi, J., Dwiel, B., Navada, S., Najaf-abadi, H. and Rotenberg, E.: FabScalar: Automating Superscalar Core Design, *IEEE Micro*, Vol. 32, No. 3, pp. 48–59 (2012).

著者発表論文

口頭発表

[1] 藤田晃史, 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: 面積効率を指向するプロセッサ「雷上動」の設計と実装, 組込み技術とネットワークに関するワークショップ ETNET 2014, (発表予定).

[2] 藤田晃史, 早水光, 中島潤, 塩谷亮太: スーパースカラ・プロセッサ「雷上動」の設計と実装, 情報処理学会報告 2014-ARC-208, No. 15, pp. 1–4 (2014).

[3] 藤田晃史, 田浦健次朗, 近山隆: 大域アドレス空間における GC とリージョンによるメモリ管理, 情報処理学会第 74 回全国大会, Vol. 1, pp. 191–121 (2012).

受賞

- チーム「雷上動」(藤田晃史, 早水光, 中島潤, 塩谷亮太): The 1st IPSJ SIGARC High-Performance Processor Design Contest プロフェッショナル部門優勝.
- 藤田晃史 , 早水光 , 中島潤 , 塩谷亮太 : スーパースカラ・プロセッサ「雷上動」の設計と実装, 情報処理学会報告 2014-ARC-208, 情報処理学会 コンピュータサイエンス領域奨励賞(CS 領域奨励賞).

謝辞

本論文の執筆にあたり，多くの方々から多大な御支援を頂きました．

指導教員である 坂井修一 教授からは，毎週の相談会にて御指導をいただいたほか，研究生活全般をバックアップして頂きました．

研究室を共同運営する 五島正裕 准教授は，プロセッサ・アーキテクチャの基礎から論理的な思考方法まで，研究に必要な多くのことを教えてくださいました．

研究室の OB である名古屋大学の 塩谷亮太 助教からは，雷上動の設計と実装を進めるにあたり，言葉では表現しきれないほど多くの御指導・御協力を頂きました．塩谷先輩と共に開発ができたことを，とても幸運に思っています．

研究室の博士課程生である 倉田成己 氏と 山田淳二 氏には，研究・開発で行き詰っている時にたびたび助言を頂きました．

その他，研究室メンバーの皆様からは，研究についての議論や日々の生活を通じて様々な御支援・御協力を頂きました．

また，The 1st IPSJ SIG-ARC High-Performance Processor Design Contest は，雷上動を発表する絶好の機会となりました．東京大学 田浦研究室の 中島潤 氏と 早水光 氏は，多忙にもかかわらずチームメンバーとして開発に参加して，雷上動を優勝に導いてくれました．その他，多くの関係者がコンテストを支えてくださいました．

最後に，私の学生生活をとても多くの面から支えてくれた両親に感謝し，本論文を締めくくりたいと思います．