NAIST-IS-DD0961025

博士論文

高性能と低消費電力を実現する プロセッサに関する研究

吉村 和浩

2012 年 3 月 16 日

奈良先端科学技術大学院大学
情報科学研究科 情報システム学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
博士 (工学) 授与の要件として提出した博士論文である.

吉村 和浩

審査委員：

中島 康彦 教授　　（主指導教員）
井上 美智子 教授　（副指導教員）
嶋田 創 准教授　　（副指導教員）

高性能と低消費電力を実現する プロセッサに関する研究*

吉村 和浩

内容梗概

近年の電子機器に搭載されているプロセッサは組込み機器と汎用計算機のそれぞれの用途で設計が大きく異なる．携帯情報端末を代表とする組込み機器用途のプロセッサは実装面積とバッテリ駆動の制約を受けるため，プロセッサは小型半導体チップに実装され，必要最低限の消費電力で要求される性能を実現するよう設計されている．今後も，大量のデータ処理とバッテリ駆動時間の延長のために，さらなる高性能と低消費電力が強く求められる．しかしながら，近い将来半導体プロセス技術の微細化の限界が懸念されており，単位面積に実装できる回路規模の増加を期待できない．そのため，回路規模増加を伴う性能向上手法はチップ面積の肥大化を招き，本手法に基づくプロセッサは組込み機器用途に不適となる．よって，省回路規模を前提とした高性能および低消費電力を実現するプロセッサが必要である．一方，汎用計算機用途を前提としているプロセッサは実装面積の制約が弱く，比較的大きな半導体チップに実装される．そこで，科学技術計算や画像処理の性能向上のために，膨大な計算量に見合う豊富なハードウェア資源を投入できる．しかしながら，汎用計算機用途でも消費電力は許容できない値に達しており，電力効率に優れたプロセッサが求められている．有望視されているものとして，既存プロセッサのコア数を増加させたメニコアアーキテクチャと，専用ハードウェアに近い構成で高性能と高電力効率を実現する粗粒度再構成アーキテクチャが挙げられるが，それぞれ電力効率とプログラマビリティの向上が課題である．よって，それぞれのアーキテクチャの長所を兼ね備えたプロセッサが求められている．

*奈良先端科学技術大学院大学 情報科学研究科 情報システム学専攻 博士論文, NAIST-IS-DD0961025, 2012 年 3 月 16 日.

そこで，本論文では 2 部構成を採り，組込み機器と汎用計算機それぞれの用途に対して高性能と低消費電力を実現するプロセッサを提案する．第 1 部では，ハードウェア資源を共有してプロセッサ全体の回路規模を削減する異種命令混在実行プロセッサ OROCHI を提案する．OROCHI はレジスタファイルと演算器を含むバックエンドを共有しつつ，異なる種類のプログラムを 2 つのスレッドで同時実行する．まず，OROCHI の命令発行機構を応用した命令フラッシュによるスレッド制御手法を提案し，提案手法と従来手法におけるヘテロジニアスマルチスレッド実行の性能を明らかにし，提案手法が全体性能を 2%改善することを示す．さらに，LSI 試作の結果より，OROCHI とマルチコアプロセッサとの比較を行い，OROCHI がマルチコアプロセッサの 79%の回路規模で 1.31 倍の電力効率を実現することを示す．第 2 部では，メニコアおよび粗粒度再構成アーキテクチャの長所を持つ線形アレイ型パイプラインプロセッサ LAPP を提案する．メニコアアーキテクチャと同様，既存機械語命令をそのまま使用することで高いプログラマビリティを維持し，粗粒度再構成アーキテクチャのように演算器アレイに命令を割り当て高速実行することで高い電力効率を実現する．まず，演算器ネットワークについて，すべての値を伝搬する比較モデルと必要な値のみを伝搬する提案モデルを示す．そして，既存機械語命令列を演算器アレイに割り当てるための命令写像手法を提案する．LAPP の論理合成および LSI 試作の結果より，提案モデルは比較モデルの 84%の回路規模になり，提案モデルを含むアレイ段は比較モデルを含むアレイ段の 88%の回路規模になることを示す．さらに，36 段構成の LAPP が 9.5 コアを搭載するメニコアプロセッサと同回路規模となることを示し，ソフトウェアシミュレータによる評価結果を踏まえ，LAPP がメニコアプロセッサの 10.9 倍の電力効率を実現することを示す．

キーワード

マイクロプロセッサアーキテクチャ，異種命令セット，マルチスレッド実行，粗粒度再構成，演算器アレイ

Studies on Processors with Features of High Performance and Low Power Consumption*

Kazuhiro Yoshimura

Abstract

Recently, processors implemented into electronic devices are designed for embedded systems and general-purpose computer systems. The processors in embedded systems such as mobile phones have strong constraints on an implementation area and a battery life. Therefore, the processors are implemented on a small semiconductor chip to meet the required performance with minimum power consumption. Much high-performance and low-power features will be required for processing a large amount of data and increasing the battery life. However, future semiconductor technologies will reach the limits of miniaturization. Performance advances in a processor with a method increasing the number of transistors will cause that the chip area of the processor enlarges. Therefore, high-performance and low-power consumption processors based on saving the number of transistors are necessary for the embedded systems. Meanwhile, processors for the general-purpose computer systems have weak constraints on an implementation area. They can be implemented into a larger semiconductor chip than the embedded systems. To increase the performance for image processing, a large number of hardware resources corresponding to the huge amount of calculation can be implemented into the chip. Therefore, Many-Core Architectures (MCAs) and Coarse-Grained Reconfigurable Architectures (CGRAs) have been proposed. MCAs are composed of general-purpose processors (GPPs) which are able to execute conventional instruction architecture based machine instructions and thus provide high-programmability with the support of existing compilers. Meanwhile, CGRAs achieve high performance and low-power consumption by reconfiguring the interconnection between many functional units (FUs). However, MCAs and CGRAs have problems

*Doctoral Dissertation, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD0961025, March 16, 2012.

about their power efficiency and programmability respectively. Therefore, a processor with advantages of both MCAs and CGRAs is absolutely imperative.

In this paper, a simultaneous multi-threading processor with heterogeneous instruction architectures called OROCHI is firstly proposed for the embedded systems. The OROCHI is capable of executing two threads simultaneously by resource sharing between two different processor units. Preliminary evaluation shows the performance of the heterogeneous SMT execution on the OROCHI is superior to the baseline model on total performance by average 2%. Based on an LSI implementation, the author compared the OROCHI with a heterogeneous multi-core processor. The evaluation results show that the power efficiency of the OROCHI is 1.31 times better, using only 79% of the chip area of the multi-core processor.

Secondly, LAPP (Linear Array Pipeline Processor) is proposed to improve energy efficiency for various workloads such as image processing and to maintain programmability by working on VLIW codes for general-purpose computer systems. The author proposed an instruction mapping scheme for the LAPP to fully exploit the array execution of FUs and bypass networks by a mapper to fit the VLIW codes onto the FUs. The instruction mapping can be finished within multi-cycles during a data prefetch before the array execution of FUs. The synthesized results show that the hardware required for mapping scheme is 84% of the cost introduced by a baseline method. The proposed mapper can further help to shrink the size of array pipeline stage, as our results show that their combination becomes 88% of the baseline model in area. The synthesized results also show that the area of a 36-stage LAPP is equal to 9.5 times that of a many-core processor. The evaluation results show that the LAPP achieves 10.9 times better power efficiency than the many-core processor.

Keywords:

Microprocessor architecture, Heterogeneous instruction set architecture, Simultaneous multithreading, Coarse grained reconfigurable architecture, Functional unit array

目 次

図目次

表 目 次

1. 緒論

本章では，本研究の背景と本研究の概要を述べ，最後に本研究の成果を述べる．

1.1 研究背景

近年，さまざまな機器にプロセッサ [1] が搭載されており，複数プログラムの実行や増加するデータ処理量への対処のための高性能化だけでなく，バッテリ駆動の延長や電力供給への配慮のための低消費電力化が求められている．

最近では，携帯情報端末でオペレーティングシステム（OS: Operating System）などの汎用プログラムだけでなく，動画や音楽を再生するためにマルチメディアプログラムを実行することが一般的になっている．半導体プロセス技術の微細化により携帯情報端末向けの半導体チップでも大規模な回路を実装できるようになった．例えば，携帯情報端末に搭載されている組込み機器向けプロセッサでは，用途に特化した拡張機能を実装したプロセッサや複数のプロセッサコアを持つマルチコアプロセッサにより高性能と低消費電力を実現している．しかしながら，近い将来には，半導体プロセス技術による微細化の限界が訪れることが懸念されており，さらなる高性能化のために単位面積に実装できる回路規模の増加を期待できなくなる．そのため，コア数増加による高性能化は回路規模を増加させ，チップ面積の肥大化を招いてしまう．同時に，リーク電流に起因する静的消費電力はチップ面積に比例して増加するため，消費電力の観点からもチップ面積の増大は許容できない．また，携帯情報端末よりも小型な組込み機器では依然として半導体チップ上に実装できる回路規模は限られており，組込み機器向けプロセッサには省回路規模を前提とした高性能と低消費電力の両立が求められている．

一方，大型の半導体チップを許容できる汎用計算機では，画像処理や科学技術計算のために膨大な計算量が要求されており，高い性能が求められている．さらに，本用途でも消費電力の増大が問題となっているため，電力当たり性能すなわち電力効率に優れたプロセッサが求められている．

図 1 に本研究の位置づけを示す．図 1 は文献 [2] から引用した既存ハードウェアの電力効率および面積効率を表し，縦軸が電力効率，横軸が面積効率を表す．

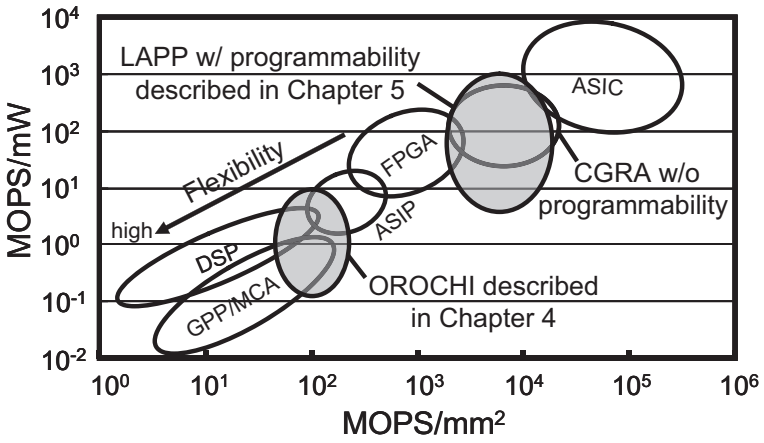


図 1 本研究の位置づけ

OROCHI と LAPP は本研究の提案するプロセッサの位置づけを表しており，それぞれの詳細は 4 章および 5 章で述べる．専用ハードウェア（ASIC: Application Specific Integrated Circuit）[3] は，実行対象となるアプリケーションに特化し最適化されたハードウェア構成を設計することで，優れた電力効率を実現している．しかし，実装すべき回路規模が増加し，開発費用の増加や開発期間の長期化が問題となっており，日々進歩するアプリケーションの新たな規格への対応が追いつかないため，専用ハードウェアの採用は困難となっている．そこで，専用ハードウェアよりも柔軟性の高いメニコアアーキテクチャ（MCA: Many-Core Architecture）[4] や粗粒度再構成アーキテクチャ（CGRA: Coarse-Grained Reconfigurable Architecture）[5, 6] が注目されている．メニコアアーキテクチャは汎用プロセッサ（GPP: General Purpose Processor）を複数並置した構成を採り，既存のソフトウェア資産の流用や並列プログラミング手法の利用ができる柔軟性を持つ．しかし，メニコアプロセッサを構成する汎用プロセッサの電力効率が ASIC よりも低いことが問題である．一方，粗粒度再構成アーキテクチャには専用ハードウェアに近い高電力効率を実現できるものの，コンパイラやプログラミング手法が成熟しておらず，プログラマビリティの向上が課題となる．以上より，メニコアアーキテクチャと粗粒度再構成アーキテクチャには一長一短があるため，両アーキテクチャの長所を併せ持つプロセッサが強く求められている．

1.2 論文概要

本論文では，組込み機器用途で高性能と低消費電力を実現するプロセッサとして異種命令混在実行プロセッサ OROCHI を提案する．また，汎用計算機用途で高性能と低消費電力を実現するプロセッサとしてアクセラレータ機構を持つ線形演算器アレイ型パイプラインプロセッサ LAPP（Linear Array Pipeline Processor）を提案する． 2 章では OROCHI と LAPP の関連研究を述べ，3 章では OROCHI および LAPP の開発に用いたプロセッサ開発手法について述べる．そして，第 1 部として 4 章では OROCHI について述べる．さらに，第 2 部として 5 章では LAPP について述べる．6 章では，OROCHI と LAPP の両者についてまとめを述べる．

1.2.1 OROCHI

OROCHI はスーパスカラプロセッサと VLIW（Very Long Instruction Word）プロセッサそれぞれのレジスタファイルと演算器を含むバックエンドを共有することでハードウェア資源を削減している．この共有バックエンドを用いて，異なる命令セット向けにコンパイルされた汎用プログラムとマルチメディアプログラムは独立したスレッドで同時実行される．このヘテロジニアスマルチスレッド実行は VLIW 型命令キュー [7] による命令スケジューリングおよび命令発行によって実現され，演算器使用率を高めている．また，OROCHI は回路規模を削減したことによりチップ面積を削減し，静的消費電力を削減する効果を得ることができる．よって，OROCHI は図 1 に示すように GPP よりも電力効率を向上できる． 以降では，ホモジニアスマルチスレッド実行で用いられている OS スケジューラによるソフトウェアレベルの従来スレッド制御手法を述べた後，VLIW 型命令キューと命令フラッシュの仕組みを応用したハードウェアレベルのスレッド制御手法を提案する．そして，予備評価としてソフトウェアシミュレータを用いて提案手法と従来手法のスレッド制御による OROCHI の性能を明らかにする．その後，全体評価として OROCHI の LSI（Large Scale Integration）試作結果を踏まえ，OROCHI とマルチコアプロセッサの回路規模，動作周波数，性能および消費電力を評価し，電力効率において OROCHI が優位であることを示す．

1.2.2 LAPP

LAPP は多数の演算器を用いて命令列を高速実行する高性能化に加えて，長期間計画的に低消費電力化技術を適用することで低消費電力化を実現する．LAPP のアクセラレータ部分は演算器を多数並置した演算器アレイから構成され，プログラム中の既存機械語命令列を演算器アレイに写像し，演算器ネットワークを構築した後，プログラムを高速実行する．既存コンパイラ技術と従来プログラミング手法を流用するために，高速実行される機械語命令列は命令セットレベルで従来プロセッサと互換性があり，LAPP のベースプロセッサである従来 VLIW プロセッサでも実行可能である．また，LAPP は高速実行するために必要最低限のハードウェア資源のみに電力を供給し，未使用ハードウェア資源には既存の低消費電力化技術を適用することで長期間計画的に停止させ低消費電力化を実現する．よって，LAPP は図 1 に示すように他 CGRA と同等の高い電力効率を実現しつつ，GPP と同等の高いプログラマビリティを維持できる．以降では，多数の演算器にデータを供給するために演算器ネットワーク上の値伝搬について評価モデルを示した後，既存機械語命令列を演算器アレイに割り当てるための命令写像手法を提案し，そのアルゴリズムとサンプルプログラムによる命令写像例を示す． そして，予備評価としてハードウェア設計した LAPP の各モジュールの論理合成結果より，値伝搬の評価モデルと命令写像機構の回路規模と遅延時間を評価する．その後，全体評価として LAPP の LSI 試作により実現性を証明した後，LAPP とメニコアプロセッサの回路規模，性能および消費電力を評価し，電力効率において LAPP が優位であることを示す．

1.3 研究成果

本研究で得られた主な成果は以下のとおりである．

1. 異種命令混在実行プロセッサの仕組みを確立した．

   従来 VLIW プロセッサと従来スーパスカラプロセッサのバックエンド部分を共有した上で，従来 VLIW プロセッサの余剰演算能力を利用し，スーパスカラ方式にて別スレッドの命令をアウトオブオーダ実行する仕組みを示

した．さらに，ヘテロジニアスマルチスレッド実行において，優先・非優先スレッドの制御が重要であることを示し，異種命令混在実行の仕組みを利用したハードウェアレベルのスレッド制御手法を提案した．

2. 異種命令混在実行プロセッサの有効性を実証した．

   異種命令混在実行の仕組みを有するプロセッサをソフトウェアシミュレータにより性能評価することに加え，本プロセッサにおける提案スレッド制御手法と従来スレッド制御手法によるスレッド実行性能を明らかにした．さらに，本プロセッサの LSI 試作により回路規模および消費電力を評価した結果，マルチコアプロセッサよりも回路規模を削減し，電力効率が優れていることを示した．

3. 線形演算器アレイ型パイプラインプロセッサにおける命令写像および高速実行の仕組みを確立した．

   メニコアアーキテクチャと粗粒度再構成アーキテクチャの長所を組み合わせ，多数の演算器から構成される演算器アレイに既存機械語命令を写像する仕組みと演算器にデータを供給する値伝搬モデルを示した．そして，写像された命令列を高速実行するとともに，既存低消費電力化技術を機能ユニットに長期間計画的に適用することで高性能と低消費電力を実現した．

4. 線形演算器アレイ型パイプラインプロセッサにおける命令写像および高速実行の仕組みの有効性を実証した．

   演算器アレイでの命令写像および高速実行の仕組みを有するプロセッサをソフトウェアシミュレータにより性能評価することに加え，本プロセッサの LSI 試作により回路規模および消費電力を評価した．その結果，本プロセッサは同回路規模のメニコアプロセッサよりも高性能かつ低消費電力を実現し，電力効率が優れていることを示した．

2. 関連研究

本章では，本研究で提案するOROCHIとLAPPの関連研究について述べる．まず，OROCHIが分類されるシングルプロセッサと，OROCHIの比較対象となるマルチコアプロセッサにおける関連研究について述べる．そして，LAPPの関連研究としてメニコアアーキテクチャと粗粒度再構成アーキテクチャについて述べる．

2.1 シングルプロセッサとマルチコアプロセッサ

プロセッサ性能を向上させる方法として命令レベル並列処理 [8] があり，例えば，スーパスカラ方式とVLIW方式が挙げられる．スーパスカラ方式はメモリから読み出した複数の命令の依存関係を動的に解析し，同時実行できる命令を見つけ出し，命令スケジューリングした後，演算器に発行する．スーパスカラプロセッサは既存のバイナリから命令レベル並列性を抽出して実行でき，新たに市場投入されるプロセッサに追加されたハードウェア資源を利用できるため，既存のソフトウェア資産をそのまま実行しても性能向上を実現できる．最近では，命令順を維持したまま命令レベル並列性を抽出するインオーダ型スーパスカラ方式から，命令順を並び替えてさらに多くの命令レベル並列性を抽出するアウトオブオーダ型スーパスカラ方式が汎用プロセッサに採用されている．しかし，より多くの命令レベル並列性を抽出するためには，ハードウェア機構が大幅に複雑化し，回路規模と消費電力の増大を招いてしまう．さらに，プログラムに内在する命令レベル並列性には限界があるため，投入したハードウェア資源に見合った性能向上を見込めないことが欠点である．

一方，VLIW方式はコンパイラにより静的に命令レベル並列性を抽出し，同時実行可能な複数の命令をパックすることで1つの大きな命令として扱う．この大きな命令をVLIW命令という．マルチメディアプログラムには高い命令レベル並列性が内在しており，コンパイル時の静的解析によって効率良く命令レベル並列性を抽出できる．VLIW方式のハードウェアはメモリから読み出したVLIW命令をそのまま演算器に発行するため，複雑な機構が不要となり消費電力を削減できる．しかし，既存のプログラムは新しく追加された演算器を活用できないため，

ソフトウェア資産が少ないことが欠点である．

さらに，プロセッサは汎用プログラムの実行に必要な命令（整数命令）だけでなく，マルチメディア処理に特化した命令（マルチメディア命令）をサポートすることで，マルチメディアプログラムの実行性能を向上させている．これらの整数命令とマルチメディア命令を同時に実行するために，整数命令を実行するハードウェアとマルチメディア命令を実行するハードウェアをそれぞれ実装することが一般的である．例えば，FR-V プロセッサ [9] は整数命令とマルチメディア命令を合わせて 8 並列実行を実現している VLIW プロセッサである．Intel 社や AMD 社の x86 プロセッサ [10, 11] や，ARM 社の ARM プロセッサ [12] では，汎用プロセッサコア内にマルチメディア向けの SIMD（Single Instruction Multiple Data）命令を実行するパイプラインステージを設け，スーパスカラ方式でマルチメディア処理を実行している．また，XC core[13] はコア内のプロセッサエレメント（PE: Processor Element）を動的再構成の技術を用いて再構成し，SIMD 実行モードと MIMD（Multiple Instruction Multiple Data）実行モードを切り替えてマルチメディア命令と整数命令を実行している．

そして，ハードウェア資源の使用率を高めるためにマルチスレッド実行（SMT: Simultaneous Multi Threading）[14] が研究されている．SMT は複数のスレッドでハードウェア資源を共有し，空きハードウェア資源で実行できる命令を複数スレッドの中から見つけ出し，ハードウェア資源を有効活用する．例えば，ハイパースレッディング [15] では，スーパスカラプロセッサのハードウェアスケジューラが空き演算器に他スレッドの命令を動的に発行することでプロセッサ全体の実行性能を向上させる．また，CSMT（Cluster-level Simultaneous Multi Threading）[16] は複数スレッドの VLIW 命令を分解し，動的に並び替え，再パックした VLIW 命令を演算器に発行することで，VLIW プロセッサでもハードウェア資源の使用率を高めている．しかし，マルチスレッド実行では，複数のスレッドのロード・ストア命令でキャッシュミスが同時に発生すると，各スレッドの実行はキャッシュミスが解決するまで停止する．複数のキャッシュミスは順番に解決されるため，各スレッドのストールサイクル数はシングルスレッド実行したときよりも増加し，各スレッドの実行性能はシングルスレッド実行したときよりも低下してしまう．

よって，マルチスレッド実行ではキャッシュミスによる性能低下を抑制することが重要であるといえる．そこで，マルチスレッド実行を制御する手法として大きくキャッシュ分割と命令フラッシュの 2 つが挙げられる．文献 [17, 18] はスレッドの優先度に応じてキャッシュ分割を行い，各スレッドが使用できるキャッシュの容量を動的あるいは静的に変更することで，キャッシュミス率を改善し，優先スレッドの性能を向上している． 文献 [19] は優先して実行するスレッドを決め，非優先スレッドが優先スレッドの実行を止めてしまう場合，パイプラインから命令をフラッシュし，非優先スレッドの命令フェッチを制御する手法を提案している．また，文献 [20] では，優先スレッドの実行を妨害する主な原因であるロード命令によるキャッシュミス [21] を解析し，予測したメモリレイテンシの値に基づいてフェッチおよびフラッシュする命令数を動的に変更することでスレッドを制御する手法が提案されている．さらに，文献 [22] はキャッシュミスを起こしたスレッドの命令を別の命令ウィンドウに退避させ，割り当てられていたハードウェア資源を一旦開放し，他スレッドに再度割り当てることでパイプラインストールを回避する手法を提案している．これらのスレッド制御手法はハードウェア資源を共有する上で問題となる共有資源の枯渇を回避し，マルチスレッド実行におけるハードウェア資源の使用率を高めつつ，優先スレッドの性能低下を防いでいる．

また，最近では，異なるプロセッサコアを複数個並置したヘテロジニアスマルチコアプロセッサが登場している．例えば，OMAP-L138[23] は ARM コアと VLIW 型 DSP コアを搭載し，MeP[24] は汎用プロセッサコアに加えて機能拡張部に DSP（Digital Signal Processor）コアまたは VLIW コプロセッサを搭載している．Cell Broadband Engine[25] では，汎用プロセッサコアとしての Power アーキテクチャの PPE（PowerPC Processor Element）と単精度浮動小数点演算を行う SPE（Synergistic Processing Element）を混載している．このようなプロセッサでは，OS を含む様々なソフトウェア資産を有効活用しつつ，命令レベル並列性が高い一部のアプリケーションについては専用ハードウェア向けの最適化を行うことによりプロセッサ全体の高性能化と低消費電力化を実現している．

以上より，ハードウェア資源を削減しつつ高性能および低消費電力を実現するためには，異なる用途のハードウェア資源を共有する仕組みだけでなく，その上

で実行される複数スレッドの優先・非優先を決定し，制御することが重要であるといえる．

2.2 メニコアアーキテクチャと粗粒度再構成アーキテクチャ

1.1 節で述べたように，画像処理や科学技術計算のように膨大な計算量を処理するために，その計算量に見合ったハードウェア資源を投入できる専用ハードウェアを採用することは困難となっている．そこで，プロセッサエレメントや機能ユニットを多数並置し，効率良くハードウェア資源を使用するさまざまなアクセラレータ方式が提案されている．例えば，メニコアアーキテクチャと粗粒度再構成アーキテクチャが有望視されている．

メニコアアーキテクチャはシングルスレッド性能の向上ではなく複数コアによるマルチスレッド性能の向上によりプログラムを高速実行する．より多くのスレッドを同時実行するために，VLIW プロセッサやインオーダプロセッサといった比較的簡素なハードウェア機構を持つプロセッサコアをチップ上に多数並置し，コア間ネットワークはメッシュ構造やリング構造が用いられている．例えば，Larrabee[26] は 2 命令発行のインオーダスーパスカラプロセッサをリングバスで接続した構造を採り，TILE64[27] は VLIW プロセッサ 64 個をメッシュネットワークで接続した構造を採用している．いずれも C 言語による並列プログラミング手法を用いてプログラムの開発が可能である．商用の汎用プロセッサでも，Intel 社や AMD 社からリング構造やクロスバースイッチ構造のコア間ネットワークを採用し，8 個のプロセッサコアを搭載するメニコアプロセッサが登場している．また，NVIDIA 社の GPGPU[28] は 128 個以上のプロセッサコアから構成され，多数のスレッドを同時実行することで性能向上を実現している．しかし，多数のスレッドからキャッシュメモリにアクセスが集中する状況では，期待した性能向上を実現することが困難な場合があり，さらに効果的なプログラムのチューニング手法が求められる．また，コア数の増加に伴い，外部メモリから供給しなければならないデータ量が増加するため，メモリバスの消費電力の増加が問題となる．さらに，プロセッサコアを並置する手法では，性能向上に比例して電力が増加するため，電力効率は単一プロセッサコアと同等以下になるが，プロセッサの単純化による電力効率の向上には限界がある．

一方，粗粒度再構成アーキテクチャは多数の機能ユニット（FU: Functional Unit）をアレイ状に並置し，実行するプログラムに応じて機能ユニットに命令を割り当て，機能ユニット間のネットワークを再構成する．そのため，専用ハードウェアに近いハードウェア構成になり，必要最低限の動作でプログラムを高速実行することで，電力効率を向上させている．例えば，VLDP[29] や LSRDP[30] はプログラム中のデータフローをプロセッサコア内の多数の演算器または PE に割り当て高速実行する．機能ユニット間のネットワークは上から下へデータが流れる単純な一方向ネットワークを採用している．しかし，1 回の実行において演算器アレイに大量のデータを供給するために，プロセッサコアと主記憶の間に高いメモリバンド幅を要求しており実現性が低いことが問題である．TRIPS[31] はデータフローを指示できる EDGE（Explicit Data Graph Execution）[32] 命令セットを用いて生成されたプログラムを実行する．このプログラム中の命令は 5×5 の機能ユニットとメッシュネットワークの上で実行される．しかし，機能ユニットに割り当てる命令を生成するために専用コンパイラが必要となる上，メッシュネットワークによるデータ供給が複雑になり，高い性能向上を見込むことが困難である．ADRES[33, 34, 35] は VLIW プロセッサ部分とアクセラレータ部分から成り，VLIW プロセッサは従来の機械語命令を実行し，アクセラレータ部分は 4×4 の FU アレイにループカーネルを写像し，パイプライン実行によりループカーネルを高速実行する．データは VLIW プロセッサのデータキャッシュからアクセラレータ部分に供給され，高速実行の結果は VLIW プロセッサに集約される．しかし，既存プロセッサとの互換性を考慮しているとはいえ，高速実行されるループカーネルを生成するには専用コンパイラが必要となる．PPA[36] は 2×2 の PE を持つ CGRA コアを 8 個並置した構成を採り，ループカーネルに応じて任意のコアを組み合わせてループカーネルをパイプライン実行できる．しかし，CGRA コアの個数だけでなく CGRA コアの組み合わせの数も考慮しなければならないため，複雑で高度なスケジューリングが必要となる．

以上で述べた CGRA の共通の問題として，単純に FU や PE を増加させても，高速実行の対象となるプログラムをそれらに割り当てるためのスケジューリングが大幅に複雑化してしまうことが挙げられる．さらに，FU や PE への割り当てとそ

れらの接続にはFPGA（Field Programmable Gate Array）を代表とする細粒度再構成アーキテクチャにおける論理合成の技術や新規の命令セットを用いたコンパイラ技術を必要とするため，技術が成熟しておらず商用化が困難である．

そこで，従来のスーパスカラプロセッサやVLIWプロセッサの演算器を2次元に拡張したCGRAが提案されている．文献[37]や文献[38]は従来のスーパスカラプロセッサのバックエンドを2次元に拡張したプロセッサを提案している．これらのプロセッサはSimpleScalar[39]で用いられている既存命令セットでコンパイルされたプログラム中のループカーネルを演算器アレイに割り当て高速実行する．しかし，演算器アレイに備えられたメモリアクセス機構は従来のメモリ階層を拡張したものであり，高速実行中でもキャッシュミスによるストールが発生するため，性能低下を招いている．この性能低下を補うためにはより多くの命令列を演算器アレイに供給する必要があり，膨大なハードウェア資源を投入しなければならない．また，2D-VLIW[40]は従来のVLIWプロセッサのバックエンドを2次元に拡張した構成をしており，ループカーネルを演算器アレイに割り当て，高速実行する．しかし，演算器アレイの規模に比例して，後段パイプラインステージに供給すべきVLIW命令の命令長が長くなってしまう．この問題を解決するためにコード圧縮手法[41]が提案されている．しかしながら，複数のロード・ストア命令を同時に実行するためのハードウェア構成の検討が不十分であり，ハードウェアの実現性に問題がある．

以上より，メニコアアーキテクチャとCGRAの長所を組み合わせることに加えて，機能ユニット，演算器ネットワークおよびメモリアクセスについて実現可能なものを検討することが重要であるといえる．

3. 実証的研究のためのプロセッサ開発手法

本章ではOROCHIとLAPPの開発に用いた開発手法と開発工程について述べる．

3.1 開発手法

近年では，迅速なハードウェア開発手法 [42, 43, 44, 45, 46, 47, 48] や開発環境 [49, 50] が多く提案され，大学研究室規模のLSI試作事例が文献 [51, 52, 53, 54] で報告されている．しかし，研究対象のプロセッサコアだけでなくその周辺回路の大規模化に伴い，設計，開発および検証にかかる人員および費用の兼ね合いが問題となっている．そこでOROCHIとLAPPの開発において，ソフトウェアシミュレータによる論理検証，FPGAによるハードウェア設計と検証，およびLSI試作後の実機評価を段階的に進めることができる開発手法を用いた．

表1にOROCHIとLAPPの開発手法で用いた諸元を示し，図2に段階的開発手法を示す．図2（a）はソフトウェアシミュレータのみを用いたソフトウェアモデルであり，図2（b）はプロセッサコアをFPGAに実装するモデルであり，図2（c）はプロセッサコアをLSI化したモデルである．そして，図2（d）は周辺回路も含めすべてをハードウェアに実装した実用システムである．ここで，大学研究室規模で図2（d）を試作することは人員および費用の兼ね合いから困難であると考えた．そこで，図2（a），図2（b）および図2（c）のいずれにおいてもI/Oやメモリシステムをホスト PC上でシミュレートし，OSのサポートが必要なシステムコールの実行をホスト PC上のOSに代理実行させることで，プロセッサコア

表1 OROCHIとLAPPの開発環境

	OROCHI	LAPP
Host PC	Intel Xeon 3.8GHz	Intel Core i7 3.33GHz
	Main memory 2GB	Main memory 6GB
FPGA	Xilinx XC2V8000	Xilinx XC5VLX330T
LSI	0.25μm process	0.18μm process

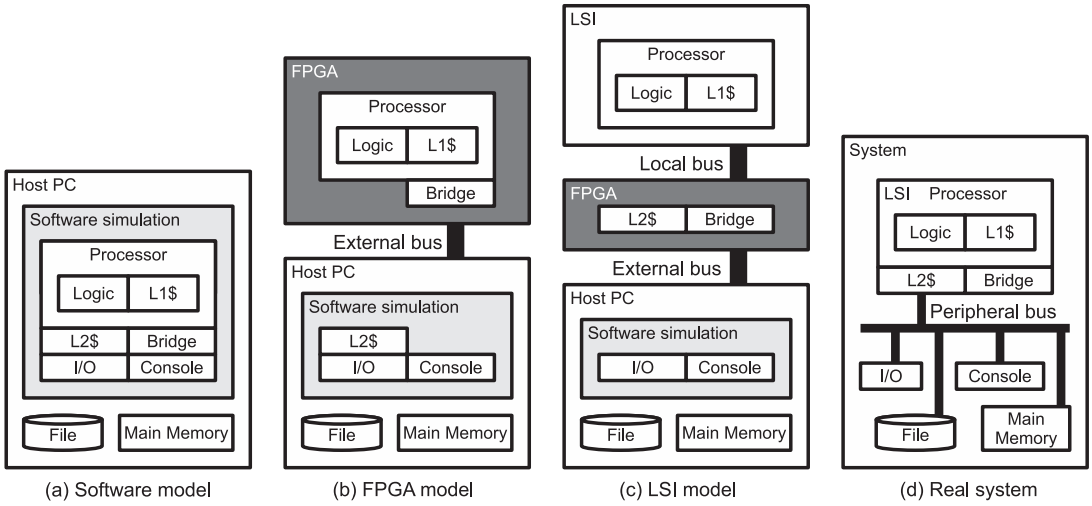


図 2 段階的開発手法

以外の周辺回路のハードウェア実装を省略し，費用の削減および開発期間の短縮を期待した．また，ホスト PC，FPGA および LSI の動作周波数などのスペックを調整することで図 2（c）における各機能ユニット間のレイテンシの比を図 2（d）と同等にでき，実用システムと同等の実行環境でプロセッサコアを評価できると考えた．

3.1.1 ソフトウェアモデル

図 2（a）のソフトウェアモデルでは，1 次キャッシュ（L1$）と論理回路を含むプロセッサコア（Processor），2 次キャッシュ（L2$），ブリッジバス（Bridge），入出力（I/O），コンソール，ファイルシステムおよび主記憶のすべての機能ユニットをソフトウェアシミュレータ上に実装するモデルである．ソフトウェアシミュレータは C 言語を用いて記述されている．図 1（a）に示したホスト PC 上で，ARM の μClinux[55] を従来の GDB ベースのシミュレータ（Armulator）で実行するとブート完了に約 2 秒かかる．一方，OROCHI のソフトウェアシミュレータではブート完了まで約 200 秒かかる．これは Armulator が命令レベルシミュレータであるのに対し，OROCHI のソフトウェアシミュレータはレジスタ転送レベル（RTL）でシミュレートしているからである．しかしながら，ソフトウェアシミュレータをレジスタ転送レベルで記述することで 3 つの利点を期待できる．1 つ目

はハードウェア記述言語（HDL: Hardware Description Language）による設計に近い精度で論理検証できる点であり，2 つ目は修正時の手間が HDL よりも C のほうが少ない点である．3 つ目はソフトウェアシミュレータと HDL 設計を検証する RTL シミュレータの間でパイプラインステージ単位の期待値比較ができる点である．以上の利点を得るために，OROCHI および LAPP の開発では，レジスタ転送レベルでソフトウェアシミュレータを開発した．

3.1.2 FPGA モデル

図 2（b）の FPGA モデルでは，プロセッサコア，1 次キャッシュおよびブリッジバスを優先して FPGA 上に実装する．プロセッサコアのフリップフロップを含む論理はロジックセルで実装され，1 次キャッシュはロジックセルとは別に用意されている専用メモリ（Xilinx 社製 FPGA では Block RAM）を使用する．また，図 2（a）のソフトウェアモデルを参考に HDL を用いて設計 [56] を行い実機検証に移ることで，RTL シミュレーションでは 1.3K サイクル/秒のシミュレーション速度が，25M サイクル/秒に向上し，開発期間の短縮を期待した．これにより，プロセッサコア単体の内部動作は図 2（d）の実用システムと一致させることができる．また，FPGA ボードとホスト PC の間にコマンドインタフェースとメモリインタフェースを実装する．これにより，ソフトウェアシミュレータはコマンドインタフェースを介してプロセッサコアを制御し，プロセッサコアはメモリインタフェースを介して 2 次キャッシュおよび主記憶にアクセスする．コマンドインタフェースとメモリインタフェースを組み合わせることで OS のサポートが必要となるシステムコールの実行をホスト PC 上の OS に代理実行させる．OROCHI の開発では FPGA ボードとホスト PC の間を外部バスとして PCI-X バスを用い，LAPP の開発では PCI-Express バスを用いてコマンドインタフェースおよびメモリインタフェースを実装した．なお，OROCHI および LAPP のプロセッサコアの論理が FPGA のハードウェア資源のほとんどを占めたため，2 次キャッシュおよび主記憶を含む周辺回路はソフトウェアシミュレータにシミュレートさせた．

3.1.3 LSI モデル

図 2（b）の FPGA モデルでは，FPGA ボードとホスト PC を接続する外部バスにより 2 次キャッシュアクセスを行うため，プロセッサコアと 2 次キャッシュのレイテンシが図 2（d）の実用システムよりも大きくなることが問題となる．この問題を解決するために，図 2（c）の LSI モデルでは，図 2（b）で FPGA に実装されていたプロセッサコアおよび 1 次キャッシュを LSI に実装した後，FPGA の空きハードウェア資源を利用して 2 次キャッシュを実装することで，プロセッサコアと 2 次キャッシュの間のレイテンシを削減し，図 2（d）の動作を再現できる．また，LSI には実行命令数やサイクル数などのパフォーマンスを計測する回路を実装することで，ソフトウェアシミュレータは主記憶および周辺回路のシミュレートだけでなくコマンドインタフェースを介して LSI からパフォーマンス情報を取得でき，LSI 試作後の評価にも利用できる．このようにソフトウェアシミュレータを開発初期から後期まで長期間使用することで，開発者は一貫して同じユーザインタフェースで開発することができ，不具合修正作業の効率化が期待できる．さらに，予算に応じて FPGA と LSI を変更できるだけでなく，開発状況や開発期間に応じてそれぞれに実装する機能ユニットを取捨選択できる．なお，OROCHI と LAPP のプロセッサコアおよび 1 次キャッシュが LSI の搭載ゲート数のほぼすべてを占めたことに加えて開発期間の都合により，2 次キャッシュとブリッジバスの実装をそれぞれソフトウェアシミュレータと FPGA に留め，プロセッサコアと 1 次キャッシュを優先して LSI に実装した．

3.2 開発工程および比較方法

図 2 に示した段階的開発環境を用いてプロセッサを実装するための開発工程を表 2 に示す．第 1 列の各工程の括弧内のアルファベットは，図 2 で示したモデルを用いることを表し，それ以外は CAD（Computer Aided Design）環境を用いることを表す．第 2 列および第 3 列はそれぞれ各工程の目的および評価項目を表し，第 4 列はシミュレーションおよび検証の実行速度を表す．本研究では，第 1 段階から第 6 段階までの開発工程を経て，OROCHI と LAPP の LSI 試作に至った．以

表 2 開発工程

工程	目的	評価項目	実行速度
1. 命令レベル シミュレーション （a）ソフトウェアモデル	命令列の動作検証	実行命令数	100M IPS
2. クロックアキュレート シミュレーション （a）ソフトウェアモデル	アーキテクチャ の妥当性検証	性能（IPC）	1M CPS
3. RTL 回路設計および RTL シミュレーション （-）CAD 環境	RTL 回路の論理検証	構成・機能	10K CPS
4. FPGA による実機検証 （b）FPGA モデル	FPGA-HostPC 間 の通信検証 大規模プログラム による検証	回路規模 遅延時間	25M CPS 40M CPS
5. GL 回路設計および GL シミュレーション （-）CAD 環境	物理レイアウト検証 実回路遅延検証 小規模プログラム による電力解析	回路規模 遅延時間 消費電力	1K CPS
6. LSI による実機評価 （c）LSI モデル	アーキテクチャ の実証 大規模プログラム による評価	性能（IPC） 消費電力	25M CPS 40M CPS

降，各工程について説明する．

第 1 段階では，プログラムの命令列を 1 サイクル当たり 1 命令実行できる命令レベルのソフトウェアシミュレータを C 言語で開発し，実行命令数を評価する．

シミュレート実行速度すなわち 1 秒あたりのシミュレート命令数（IPS: Instruction Per Second）は 100M であり，比較的容易に大規模プログラムを用いて評価できる．第 2 段階では，命令列の実行をさらに詳細に調査するために，トランジスタ転送レベル（RTL: Register Transfer Level）で命令列を実行するクロックアキュレートなソフトウェアシミュレータを C 言語で開発する．このソフトウェアシミュレータでは，毎クロックサイクルのパイプラインステージの動作をシミュレートするため，性能として 1 サイクル当たりの実行命令数（IPC: Instruction Per Cycle）を評価できる．しかし，シミュレート実行速度すなわち 1 秒あたりのシミュレートサイクル数（CPS: Cycle Per Second）は 1M となり，第 1 段階の 100 分の 1 に低下する．第 3 段階では，クロックアキュレートシミュレータを参考に，CAD 環境を用いて HDL による RTL 回路設計を行い，RTL シミュレーションによりプロセッサの論理検証を行う．本段階でパイプラインステージの構成および機能を回路として設計し，構成および機能の妥当性を評価できる．しかし，毎クロックサイクルの全信号のシミュレーションを行うため，シミュレート実行速度は 10K CPS となり，第 2 段階の 100 分の 1 に低下する．その結果，検証および修正に時間を要してしまう．そこで第 4 段階では，設計した回路を論理合成および配置配線により FPGA に実装し，実行速度を FPGA の動作周波数である 25M CPS または 40M CPS に改善する．また，FPGA 向けの論理合成および配置配線により，シミュレーションでは発見されなかった HDL の設計ミスを発見することができる．さらに，FPGA における回路規模および遅延時間の評価だけでなく，大規模プログラムによる FPGA とホスト PC の通信検証および論理検証を行うことができる．第 5 段階では，論理検証が完了した回路の物理レイアウト検証を行い，実回路の回路規模および遅延時間を評価する．また，LSI 試作のために物理レイアウトおよび実回路遅延を評価するゲートレベル（GL: Gate Level）の回路設計およびシミュレーションを行う．さらに，シミュレーションおよび電力解析ツールにより，消費電力の見積もることができる．しかし，これらの CAD ツールの実行速度は 1K CPS となり，第 1 段階の 100000 分の 1 まで低下する．その結果，実行には膨大な時間を要するため，大規模なプログラムを用いた評価を行うことは実用的ではなく，小規模なプログラムに限定される．そこで第 6 段階では，LSI 試作による実機評

価によって，実用的な実行速度で大規模プログラムによる性能および消費電力の評価を行う．以上より，本研究では第 1 段階から第 6 段階まで実施し，プロセッサの回路規模および遅延時間の評価だけでなく，大規模プログラムによるプロセッサの性能および消費電力の評価を実用的な速度で行う．

ところで，本研究では OROCHI と LAPP の比較対象として，マルチコアプロセッサとメニコアプロセッサを用いる．OROCHI と LAPP はそれぞれの関連研究のプロセッサと比較することは可能であるが，それぞれのプロセッサのパイプライン構成やキャッシュ構成が異なるため，提案部分の効果だけを評価することができない．そこで関連研究では，提案部分以外は同じ構成となる比較対象を用意して評価している．例えば，文献 [7] では，VLIW 型命令キューを用いたスーパスカラ方式を提案しており，提案スーパスカラ方式の ARM コア（ARMVLIW）と従来スーパスカラ方式の ARM コア（ARMSS）を比較した結果，ARMVLIW が ARMSS よりも小回路規模かつ高性能であることを実証している．文献 [14, 19] では，提案手法の適用部分以外は同じ構成をとるプロセッサモデルを用いて複数のモデルを評価している．ADRES[34] では，ADRES の一部である VLIW プロセッサ部分と同じ構成の VLIW プロセッサを用意し，ADRES がベンチマークプログラム全体の実行時間を VLIW プロセッサの 4.84 倍に高速化したことを報告している．2D-VLIW[40] では，2D-VLIW の一部である VLIW プロセッサと近い構成の VLIW プロセッサを別途用意し，性能比較を行っている．本研究では，提案部分の効果を明らかにするために，OROCHI と LAPP の提案部分のみが比較対象との違いとなるマルチコアプロセッサとメニコアプロセッサを用いる．

4. 異種命令混在実行プロセッサ OROCHI

本章では，スーパスカラプロセッサと VLIW プロセッサそれぞれのバックエンドを共有することでハードウェア資源を削減する OROCHI について概要を述べる．次に，OS スケジューラによるソフトウェアレベルの従来スレッド制御手法を述べ，VLIW 型命令キューと命令フラッシュの仕組みを応用したハードウェアレベルのスレッド制御手法を提案する．そして，予備評価として提案手法と従来手法のスレッド制御による OROCHI の性能を明らかにする．最後に，全体評価として OROCHI の LSI 試作結果を踏まえ，OROCHI とマルチコアプロセッサの回路規模，動作周波数，性能および消費電力を評価し，電力効率において OROCHI の優位性を示す．

4.1 緒言

1.1 節で述べた省回路規模を前提とした高性能と低消費電力を実現するプロセッサに対して，性能向上を目的とした追加ハードウェア資源の投入や，複数コアを並置することは回路規模の増大を招き，最適の解とはならない．本研究では，2 個のプロセッサコアを組み合わせ，ハードウェアを共有することで回路規模を削減する．ここで，実行するプログラムを組込み機器の代表として携帯情報端末に注目すると OS などの整数プログラムとマルチメディアプログラムの 2 種類に分類できる．それぞれのプログラムの実行に適した異なるプロセッサコアを用いることでそれぞれの実行で消費される電力を最低限に留め，プロセッサ全体の低消費電力化を図る．

まず，マルチメディアプログラムの実行に向いている VLIW プロセッサに注目する．できる限り多くの命令が同時実行できるように複数の命令を VLIW 命令にパックしようとしても，実際には並列性の抽出に限界があり，VLIW 命令が実行されたときに必ずしもすべての演算器が使用されていないことに注目する．次に，整数プログラムは潜在的な命令レベル並列性を期待することができないため，スーパスカラ方式で使用できるハードウェアを柔軟に変更できるようにする．すなわち，VLIW プロセッサの余剰演算能力を利用してスーパスカラ方式で整数プ

ログラムを実行することができれば，VLIW プロセッサとスーパスカラプロセッサを並置したヘテロジニアスマルチコアプロセッサよりもチップ面積を削減できる．

OROCHI では，VLIW 方式で実行するマルチメディアプログラムには FR-V 命令セット [57] を用いた．FR-V 命令セットは組込み用途向けのマルチメディアプロセッサである FR-V プロセッサに採用されている．一方，スーパスカラ方式で実行する汎用プログラムには ARM 命令セット [58] を用いた．ARM 命令セットは多くの組込み機器のプロセッサで採用され，業界標準となっている．これらの命令セットは GCC（GNU Compiler Collection）でサポートされており，コンパイラの新規開発が不要であること，命令セットの拡張が可能であること，既存の整数プログラムを使用できることが利点である．OROCHI では，FR-V プロセッサと ARM プロセッサのバックエンドを共有することでヘテロジニアスマルチコアプロセッサよりも省回路規模で高性能と低消費電力を実現することが目的である．

4.2 概要

ハードウェア資源を共有した 1 つのプロセッサで異なる命令セットを同時実行するためには，VLIW 方式の余剰演算能力を有効利用し，かつ回路規模を削減できるスーパスカラ実行方式を考えなければならない．本節では，この仕組みを実現する OROCHI の概要ついて述べる．

4.2.1 パイプラインモデル

OROCHI のパイプライン構成を図 3 に示す．OROCHI は FR-V と ARM それぞれのフロントエンドと共通バックエンドから構成される．灰色の長方形は FR-V のユニット，白色の長方形は ARM のユニットを表し，灰色兼白色の長方形は FR-V と ARM で共有するユニットを表す．また，太枠の長方形はキャッシュメモリを表し，灰色と白色の丸はそれぞれ FR-V と ARM の命令を表す．

各フロントエンドは，プログラムカウンタ，命令フェッチユニット，1 次命令キャッシュ，分岐予測器，命令デコーダおよびリターンアドレススタックを持つ．ARM デコーダと HOST デコーダは ARM 命令を内部命令に分解し，VLIW デコー

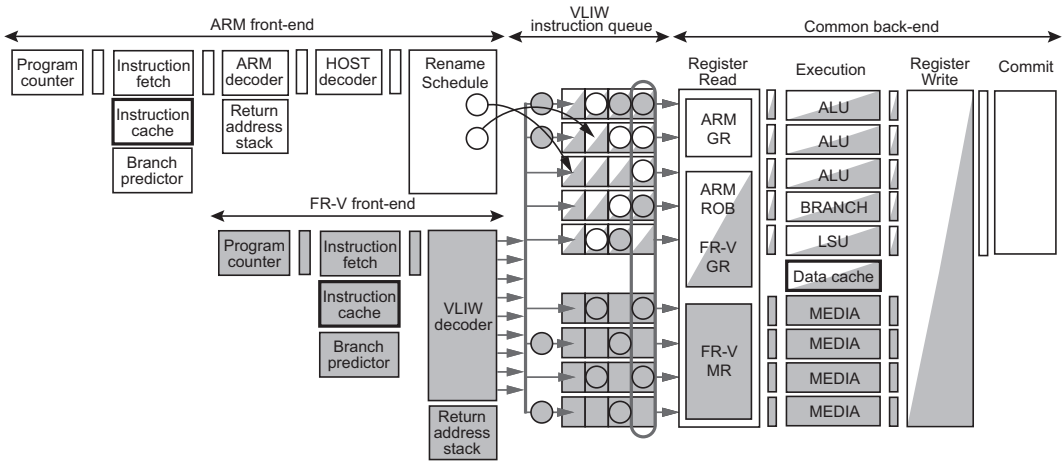
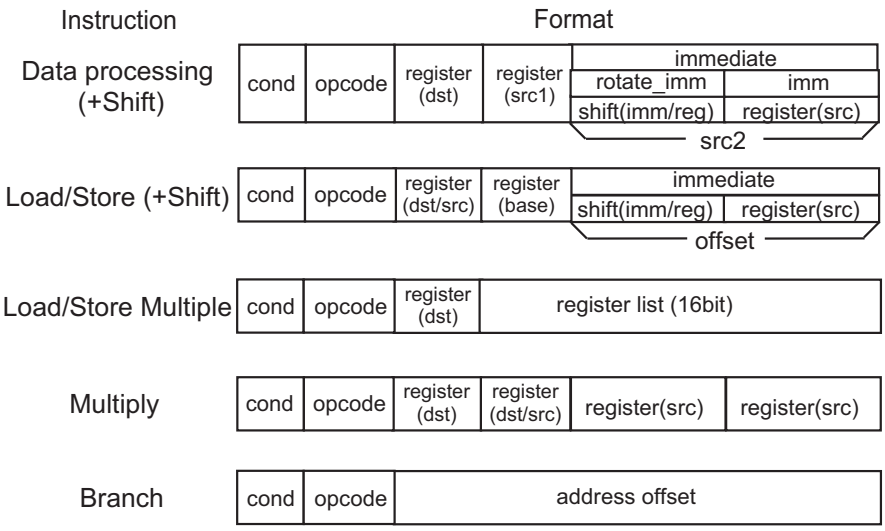


図 3 OROCHI のパイプライン構成

ダは FR-V 命令を内部命令に変換する．ここで，内部命令は FR-V 命令のサブセットに ARM のスーパスカラ実行に必要な命令を追加したものである．さらに，ARM フロントエンドは ARM のスーパスカラ実行のために，レジスタリネーミングと命令スケジューリングを行うリネーム・スケジューラを持つ．各フロントエンドと共通バックエンドは VLIW 型命令キューで接続されており，FR-V と ARM の内部命令は各フロントエンドから VLIW 型命令キューへ投入される．

共通バックエンドは，レジスタ読み出しステージ，実行ステージ，レジスタ書き込みステージおよびコミットステージから成る．レジスタ読み出しステージでは，回路規模を削減するために ARM のリオーダバッファ（ARM ROB）と FR-V の汎用レジスタファイル（FR-V GR）で共有されるレジスタファイルが設けられ，ARM の汎用レジスタファイル（ARM GR）と FR-V のマルチメディアレジスタファイル（FR-V MR）が独立して設けられている．実行ステージは算術論理演算器（ALU），分岐用演算器（BRANCH），ロード・ストアユニット（LSU），メディア用演算器（MEDIA）の合計 9 演算器から成る．VLIW 型命令キューから発行された命令はレジスタファイルから値を読み出し後，演算器で実行される．FR-V 命令はレジスタ書き込みステージにて実行が完了し，ARM 命令はコミットステージにてプログラム順に並び替えられた後，実行が完了する．



dst: destination, src: source, imm: immediate value, reg: register, cond: condition code

図 4 ARM 命令セットの命令形式

4.2.2 命令分解

ARM は命令形式が単純な RISC 型命令セットの特徴だけでなくコード効率の良い CISC 型命令セットの特徴を備えており，一般的な RISC 命令では複数の命令で表現する処理を 1 つの ARM 命令で表現できる．このような ARM 命令を共通バックエンドにおいて RISC 型命令セットである FR-V と同時実行するために，ARM 命令を複数の内部命令に分解する．

図 4 に ARM 命令セットの命令形式を示す．ARM 命令セットはデータ処理命令，シフト付ロード・ストア命令，複数ロード・ストア命令および分岐命令に分けられる．まず，データ処理命令とロード・ストア命令ではソースオペランドに対してシフト演算やローテート演算を適用した上で演算に使用される．従来の ARM プロセッサでは連結された複数の演算ステージによって実行されている．また，ロード・ストア命令では，ロード・ストア命令に示されたレジスタリストを使用して複数のレジスタとメモリ間のデータ転送を一つの命令で表現できる．しかし，この演算ステージやメモリアクセス機構を共通バックエンドに備えた場合，FR-V 命令では使用されないため，ハードウェア資源を共有することができない．そこで，OROCHI ではこれらのデータ処理命令とロード・ストア命令を単純な命令に

```
Target instruction (conditional execution)
 if(cond) ADD R1 R2 -> R3 else NOP

Internal instructions
 ADD R1 R2 -> T0
 SEL {if(cond) T0 else R3} -> R3
```

図 5 条件付き命令の分解例

分解する．例えば，シフト演算を含むデータ処理命令はシフト命令と主データ処理命令の 2 つに分解される．ロード・ストア命令は複数のアドレス計算と単純なメモリアクセスのみを行うロード・ストア命令に分解される．また，従来 ARM プロセッサでは実行に複数サイクルを必要とする乗算・積和命令をスーパスカラ実行するにあたり，部分積と加算を行う命令列に分解し，FR-V の空きスロットに挿入しやすくする．これにより，ARM 乗算・積和命令が実行中の間，FR-V 命令の実行を巻き込んでパイプラインストールが発生することを避けることができる．例えば，32bit×32bit の乗算は複数の 32bit×8bit の部分積と加算命令に分解される．FR-V の乗算命令についてはコンパイラによって同様の命令分解を行うようコンパイラおよび命令の拡張を行う．

さらに，ARM 命令セットではすべての命令に対して条件コード（cond）による条件付き実行を指定できる．条件付き実行を指定された命令を完了するためには，条件を生成する命令の実行が完了することを待たなければならないため，スーパスカラ実行による性能向上を期待できない．そこで，条件付き ARM 命令を条件なし命令と条件選択命令の 2 命令に分解する．図 5 に条件付き ARM 命令の分解例を示す．cond は条件コード，ADD は加算命令，R1 と R2 はソースレジスタ，R3 はデスティネーションレジスタを表す．条件コードの成立の場合，加算命令の結果が R3 に代入され，不成立の場合は，NOP（No OPeration）命令となり，R3 への代入は行われない．R3 への実行は ADD だけでなく条件コードを生成する先行命令にも依存することになる．OROCHI では，一時レジスタ（T0）を用いて条件なし加算命令と条件選択命令（SEL）に分解することで，R3 の実行は加算命令にのみ依存することになりスーパスカラ実行を効率的に行うことができる．

以上により，OROCHI は ARM 命令を命令分解し，VLIW 方式の空き演算器に発行できる形に変換する．

4.2.3 VLIW 型命令キュー

ハイパースレッディング [15] では，命令スケジューラが 2 スレッドの命令の依存関係を解析し，アウトオブオーダでスケジューリングする．そして，命令発行機構は発行可能な複数の命令のうち任意の命令を演算器に発行できる複雑な仕組みを備えることで，性能向上を実現している．しかし，発行検査の対象となる命令数が多く，回路規模および消費電力の増大が問題である．そこで，OROCHI では，図 3 に示された VLIW 型命令キューによって ARM と FR-V のマルチスレッド実行および ARM のスーパスカラ機構の簡素化を実現している．

図 3 のように FR-V 命令はデコード後，各命令の種類に応じて対応する VLIW 型命令キューの左端スロットに投入される．一方，ARM 命令は内部命令に分解された後，VLIW 型命令キューに既にスケジューリングされている ARM 命令との依存関係を調べた上で，VLIW 型命令キューの空きスロットにアウトオブオーダで投入される．VLIW 型命令キュー内の命令は毎サイクル 1 段ずつバックエンド方向へシフトされ，VLIW 型命令キューの右端から一斉に発行される．命令は VLIW 型命令キュー内で発行順に並んでおり，発行は右端スロットからのみに制限される．そのため，発行可能な命令を選択する機構が不要となるだけでなく発行検査の対象となる命令数を削減することで発行機構を簡素化する．

図 6 に VLIW 型命令キューによる ARM 命令のスケジューリングの例を示す．VLIW 型命令キューは 4 ステージから構成され，6 つの命令が既に VLIW 型命令キューにスケジューリング済みであることがわかる．ここで簡略化のためにすべての命令は ARM 命令に限定し，空きスロットにスケジューリングしていく．FR-V の VLIW 命令列は取り除いているが空きスロット数が異なるだけと考えることができる．S1 および S2 はソースレジスタ番号を表し，D はデスティネーションレジスタ番号を表す．まず命令 A（ALU 命令）は S1=5 より Stage0 の ALU0 命令（D=5）に依存し，S2=8 より Stage1 の ALU0 命令（D=8）に依存していることがわかる．そこで，命令 A は既にスケジューリングされている Stage3 の命令を追い

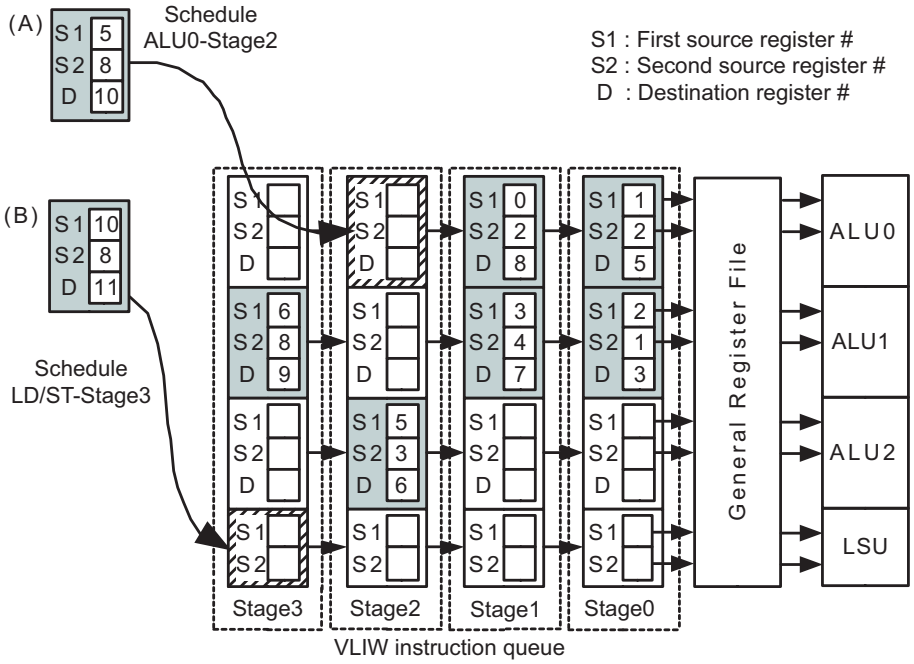


図 6 VLIW 型命令キューによる ARM 命令のスケジューリング

越して，Stage2 の ALU0 にアウトオブオーダでスケジューリングされる．また，命令 B（LD/ST 命令）は S1=10 および S2=8 より Stage3 の LD/ST にスケジューリングされる．ここで，命令 A の S=8 および命令 B の S=10 はそれぞれの命令が演算器に実行させるタイミングでそれぞれ先行命令から値がフォワーディングされる．このフォワーディングは ARM および FR-V の命令を 1 サイクルで実行可能な RISC 型内部命令に変換および分解することで実現される．

さらに，OROCHI は FR-V のために 32 本の汎用レジスタファイルと 32 本のメディア用レジスタファイルを備え，ARM のために汎用レジスタに加えて命令分解で使用する一時レジスタ 6 本，リオーダバッファ32 本を備える．FR-V の汎用レジスタファイルと ARM のリオーダバッファはポートを共有する．共有されたレジスタファイルには，多数の読み出しポートと書き込みポートが必要となる．従来プロセッサではマルチポートのメモリセルを用いてレジスタファイルを構成する方法が採用されている．しかし，マルチポートのメモリセルはシングルポートのメモリセルに比べて面積が大きく，アクセス速度の向上が困難である．OROCHI

では，バックエンドの回路規模削減のためにレジスタファイルに対してマルチバンク化を行った上で，読み出しバンク競合と書き込みバンク競合の回避を命令スケジューリング時に行う．バンクは Modulo-4 で行われているためレジスタ番号の下位 2bit の比較によって行われる．

以上により，OROCHI はマルチスレッド実行と共通バックエンドの回路規模削減を実現する．

4.3 スレッド制御手法

スレッドに優先・非優先が存在せず平等に命令スケジューリングを行うハイパースレッディング [15] と異なり，OROCHI では，VLIW 型命令キューにより FR-V 命令実行時の余剰演算能力を利用して ARM 命令を実行するため，FR-V のスレッドが優先スレッドとなり，ARM のスレッドは非優先スレッドとなる．そのため，非優先側の実行が優先側の実行を妨げ，両スレッドの性能を低下させてしまう問題がある．図 7 に OROCHI のスレッド実行における問題点を示す．図 7（a）のように，ARM と FR-V の命令は VLIW 型命令キューの右端から一括で発行されるため，ARM のロード命令が L1 キャッシュミスを起こすと，図 7（b）のように，そのロード命令に依存する ARM の後続命令が VLIW 型命令キューの右端に到達した時，ロード命令の L1 キャッシュミスが解決するまでストールが発生し，同時発行されるべき FR-V の命令実行もストールしてしまう．このようなキャッシュミスによるスレッド性能の低下を改善する従来手法として OS のプロセススケジューラによりスレッドの実行を意図的に一定期間停止することでキャッシュミスを抑制し，他スレッドの実行を優先するソフトウェアレベルの制御が挙げられる．また，ハードウェアによる制御ではキャッシュ分割 [17, 18] と命令フラッシュ [19, 22] が挙げられる．キャッシュ分割は各スレッドに応じてキャッシュ容量を静的あるいは動的に割り当てることで，キャッシュミスを抑制し性能を改善する．OROCHI においてもキャッシュ分割を適用することは有用であるが，VLIW 型命令キューの異種命令同時発行の制約によるストールを解決できない．そこで，VLIW 型命令キューと命令フラッシュを組み合わせ，VLIW 型命令キューの異種命令同時発行の制約によるストールを解決したスレッド制御手法を提案する．本節では，FR-V

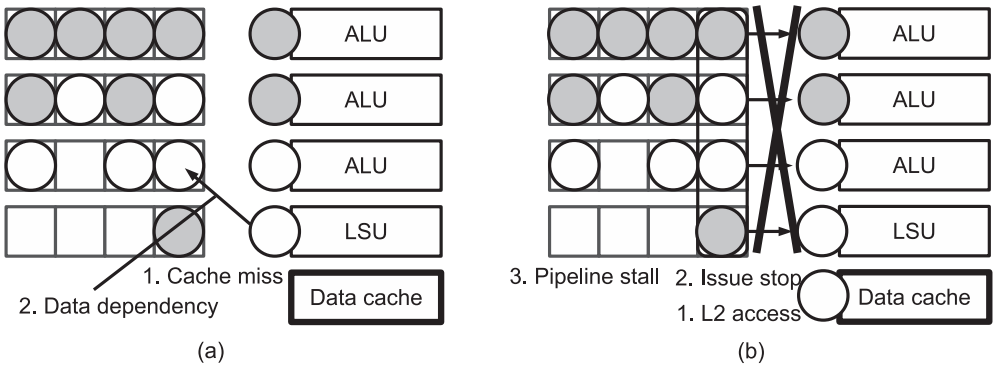


図 7 OROCHI のスレッド実行における問題点

のスレッドを優先スレッド，ARM のスレッドを非優先スレッドとして，従来手法の OS スケジューラによるスレッド制御について説明した後，パイプライン中の命令をフラッシュするハードウェアによる提案手法について述べる．

4.3.1 OS スケジューラによる制御

ARM の命令実行中に発生する L1 キャッシュミスが頻繁に発生してしまうほど，FR-V の命令実行がストールするサイクル数が増加する．そこで，ARM スレッド上で実行されている OS のスケジューラが意図的に一定期間だけプロセスの実行を停止すれば，その期間中は ARM の L1 キャッシュミス回数をできる限り抑制できる．

OS スケジューラはタイマーなどの割り込み発生時に実行され，次の実行すべきプロセスを決定している．あるプロセスが実行可能状態にあるとき，OS スケジューラによって実行状態に遷移され，プロセスは割り当てられた時間を使い切った後，または，入出力待ちになるまで実行された後，待機状態に遷移する．そして，OS スケジューラが実行可能状態を介してそのプロセスを待機状態から実行状態に再び遷移させる．ここで，そのプロセスが実行状態に遷移されることが複数回ある場合，そのプロセスの実行状態への遷移を回避することで ARM スレッドの命令は実行されず，FR-V スレッドの命令のみが実行される．例えば，OS スケジューラがあるプロセスに対して 5 回に 1 回の頻度で，実行状態への遷移を回避させれば，ARM 性能は 80%になる．

4.3.2 命令フラッシュによる制御

4.3.1 項で述べた OS スケジューラによる制御では，ミリ秒単位の粗粒度の制御となり，サイクルオーバヘッドが発生する．そこで，提案手法として 4.2.3 項で述べた VLIW 型命令キューの機構を利用してハードウェアによる細粒度の制御を提案する．図 8 に提案手法を示す．まず，図 8（a）のように OROCHI では ARM のロード命令による L2 キャッシュアクセス要求を残しつつ，VLIW 型命令キューとフロントエンドから ARM 命令のみをフラッシュする．その結果，図 8（b）のように ARM 命令が再度 VLIW 型命令キューの右端に到達するまで，パイプライン中には FR-V の命令のみが存在しストールすることなく実行される．ここで，L2 キャッシュアクセスが完了するまでこの ARM 命令のフラッシュを繰り返し，ARM 命令が FR-V 命令の実行を妨げることを回避する．また，ノンブロッキングキャッシュを LSU に実装することで，先行 ARM ロード・ストア命令の L2 リクエストが残っていても後続の FR-V ロード・ストア命令がキャッシュヒットする場合，OROCHI はマルチスレッド実行を継続できる．ただし，後続の FR-V ロード・ストア命令が L1 キャッシュミスした場合は先行 ARM ロード・ストア命令の L2 アクセスが完了するまでストールすることになる．この VLIW 型命令キューによる命令フラッシュ機構は，ARM 命令または FR-V 命令を判別する 1 ビット信号と従来のパイプラインフラッシュ機構を組み合わせるだけで容易に実装可能である．

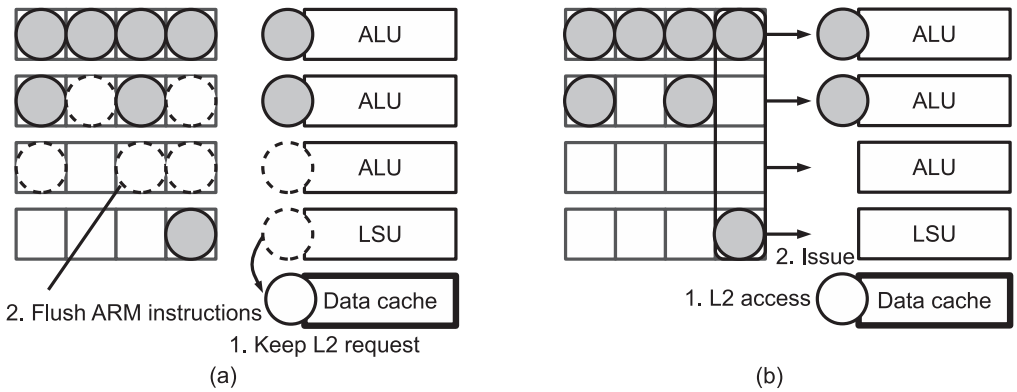


図 8 命令フラッシュによるスレッド制御

4.4 予備評価

本節では，OROCHIにおけるOSスケジューラと命令フラッシュの2つのスレッド制御手法による性能を明らかにするために予備評価を行う．まずシングルスレッド実行における性能を測定した後，マルチスレッド実行におけるARMとFR-Vそれぞれの性能を示す．そして，OROCHIの全体性能より各スレッド制御手法の効果を明らかにする．

OROCHIの評価パラメータを表3に示す．VLIW型命令キューの深さは文献[7]より4とする．ベンチマークプログラムは汎用の整数プログラムとしてMiBench[59]を使用し，ARMとFR-VのGCCクロスコンパイラでコンパイルしたバイナリを使用する．FFTとbasicmatchは浮動小数点演算を必要とするため，ソフトウェアライブラリを使用し整数演算で代替する．さらに，マルチメディアプログラムとしてステレオ画像処理（stereo）と輪郭検出（edge）のプログラムを手動で作成する．stereoとedgeはFR-Vのマルチメディア命令のひとつである差分絶対値の総和（SAD）命令を使用する．従来スレッド手法を実装したARM向けのμClinuxを使用する．

図9にOROCHIにおけるOSとベンチマークプログラムの実行モデルを示す．OROCHIのソフトウェアシミュレータ上でシングルシステムイメージとしてARMのμClinuxを構築し，その上でARMのベンチマークプログラムを実行する．一方，FR-Vのベンチマークプログラムは OROCHI上で直接実行され，ハンドシェイクによりOSに監視される．まず，OSがOROCHIに起動信号を通知することで，OROCHIはFR-Vプログラムの実行を開始する．FR-Vプログラムの実行中に発生するシステムコールはライブラリを介してOSによって実行される．そして，FR-Vプログラムの実行が完了すると，OROCHIがOSに終了信号を通知する．

本稿では，従来手法であるOSスケジューラによるスレッド制御をOSのプロセススケジューラに実装した．なお，対象となるプロセスはARMベンチマークプログラムのプロセスだけであり，その他のスケジューリングポリシーは変更せず，OS本来の機能を用いた．一方，提案手法である命令フラッシュによるスレッド制御はOROCHIのソフトウェアシミュレータに実装した．

表 3 OROCHI の評価パラメータ

Branch predictor	gshare, PHT: 2 bit×8K entry
Return Address Stack	8 entry
VLIW queue depth	4
Re-order buffer	32 entry
ARM general register	16 entry
FR-V general register	32 entry
FR-V multimedia register	32 entry
Store buffer	8 entry
Cache line size	64 byte
ARM instruction level 1 cache	4 way, 16 KB
FR-V instruction level 1 cache	4 way, 16 KB
Data level 1 cache	4 way, 16 KB
L1 miss latency	8 cycle
Unified L2 cache	4 way, 2 MB
L2 miss latency	40 cycle

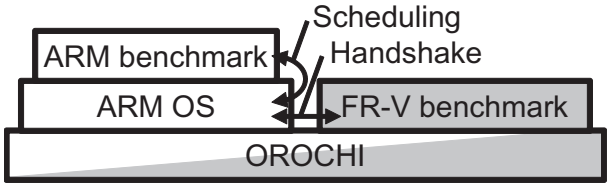


図 9 OROCHI における OS とプログラムの実行モデル

4.4.1 シングルスレッド実行における各スレッド性能

OROCHI のシングルスレッド実行における性能を評価するために，OROCHI 上で ARM および FR-V のベンチマークプログラムそれぞれを単独でシングルスレッド実行した結果を図 10 に示す．縦軸は 1 サイクル当たりの実行命令数（IPC）を表し，横軸は各ベンチマークプログラムを表す．FR-V の整数プログラムでは平均 IPC は 1 を超えていることがわかり，ARM の整数プログラムでは平均 IPC が

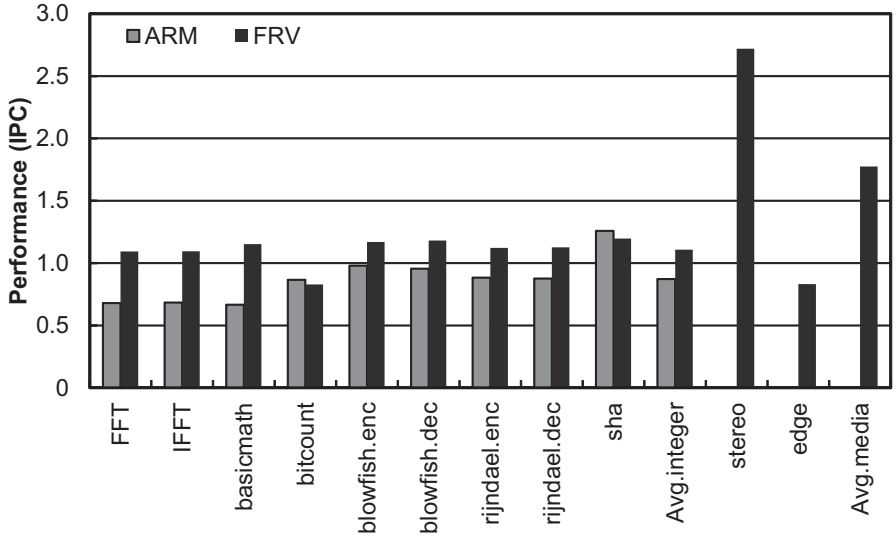


図 10 ARM と FR-V のシングルスレッド性能

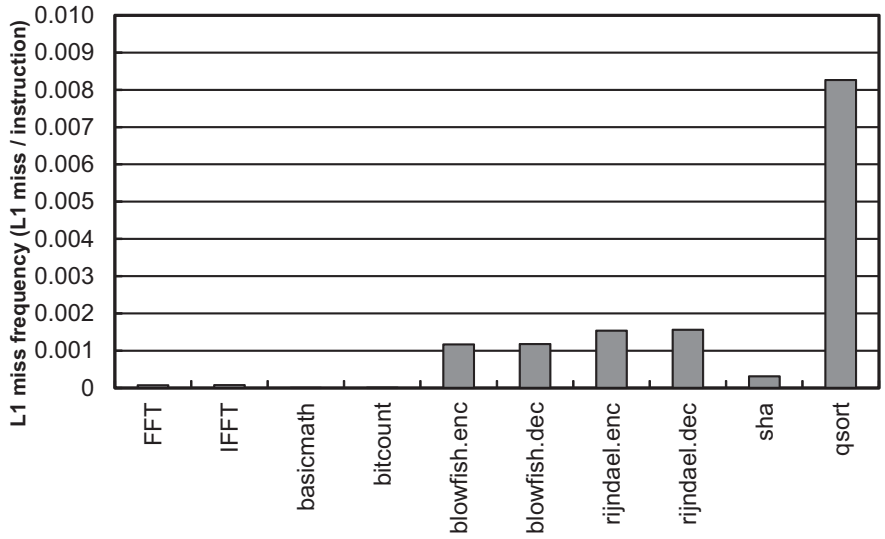


図 11 ARM ベンチマークプログラムの L1 キャッシュミス頻度

0.87 であることがわかる．また，2 つのマルチメディアプログラムのうち stereo は IPC が 2.72 であり，画像処理アプリケーションにおける高い命令レベル並列性を抽出できていることがわかる．

次に，FR-V スレッドの性能低下を引き起こす ARM の L1 キャッシュミス頻度を調べた結果を図 11 に示す．縦軸は 1 命令当たりの L1 キャッシュミス回数を表

し，横軸は ARM のベンチマークプログラムを表す．図 11 より qsort の L1 キャッシュミス頻度が最も高いことがわかる．したがって，以降で評価を行うマルチスレッド実行では ARM の qsort プログラム（ARM-qsort）を使用し，FR-V の各ベンチマークプログラム（FRV-Mibench）を使用する．

4.4.2 マルチスレッド実行における各スレッド性能

ARM の qsort プログラム（ARM-qsort）と FR-V の各ベンチマークプログラム（FRV-Mibench）のマルチスレッド実行において，ARM の OS スケジューラによるスレッド制御では ARM プログラムの実行を 80%，60%，40%，20%に抑制し，FR-V プログラムの実行性能を向上させる．OROCHI のベースライン，命令フラッシュおよび OS スケジューラによるマルチスレッド実行における FR-V 性能と ARM 性能をそれぞれ図 12 と図 13 に示す．図 12 の縦軸は FRV-Mibench を実行する FR-V スレッドの IPC を表し，図 13 の縦軸は ARM-qsort を実行する ARM スレッドの IPC を表す．図 12 と図 13 の横軸は FRV-Mibench の各ベンチマークを表す．SMT は OROCHI のベースラインモデルを表し，SMT+FLUSH はベースラインモデルに命令フラッシュを付加したモデルを表す．OS-80，OS-60，OS-40 および OS-20 はそれぞれ ARM の OS スケジューラにより ARM プログラムの実行を 80%，60%，40%，20%に抑制するモデルを表す．

図 12 と図 13 より，ARM プログラムの実行を抑制するにつれて，FR-V プログラムの実行性能が向上していることがわかる．FR-V の平均 IPC に注目すると，OS スケジューラによって ARM プログラムの実行を 20%まで抑制すると IPC は 1.05 まで向上し，図 10 の FR-V のシングルスレッド実行における平均 IPC が 1.11 であることから，シングルスレッド実行の 95%の性能まで向上できたことがわかる．また，FR-V の平均 IPC が SMT+FLUSH で 0.85 であり，SMT で 0.83 であることから，命令フラッシュは OS スケジューラによる制御なしで 2%の性能向上を実現したことがわかる．命令フラッシュによりシングルスレッド実行と同等の性能に向上できない原因として，ARM プログラムのメモリアクセスにより FR-V プログラムで使用するキャッシュラインを追い出していること，ARM が L2 アクセスしている間に FR-V の L2 アクセスが発生し，ARM の L2 アクセスが完了する

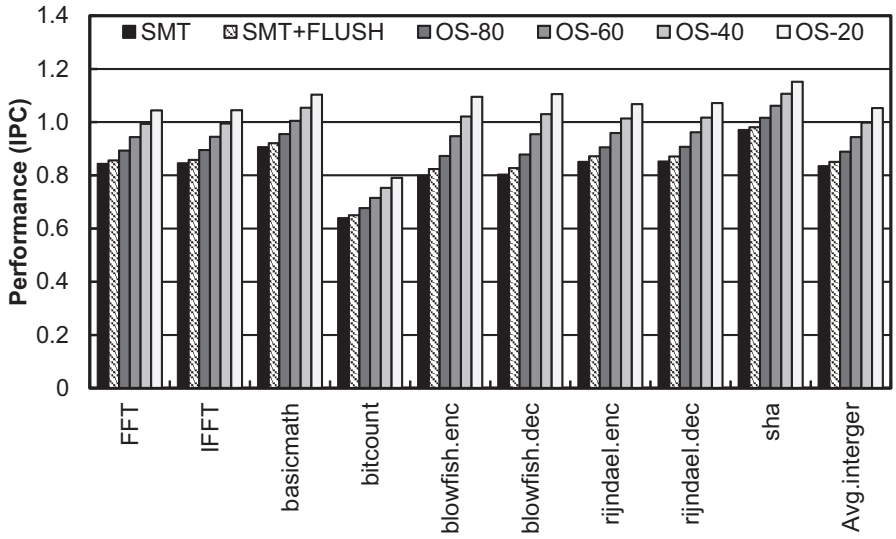


図 12 マルチスレッド実行における FR-V 性能

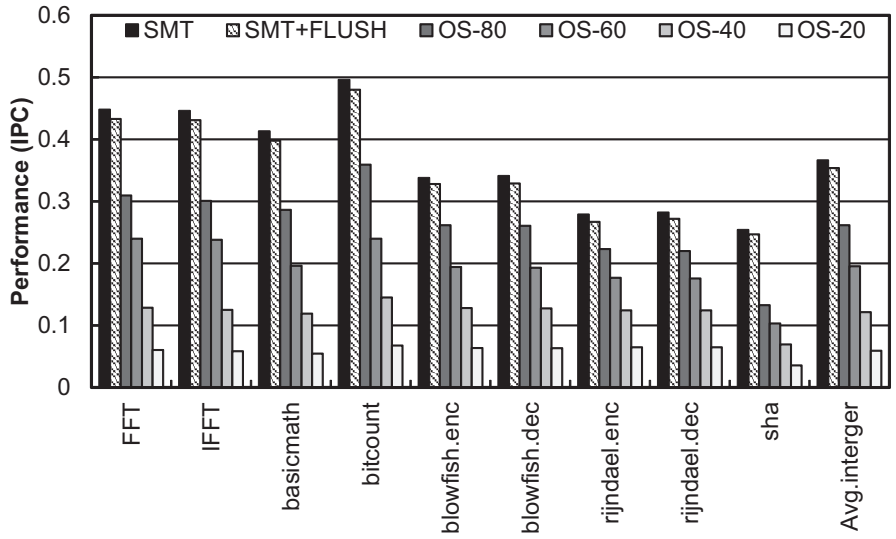


図 13 マルチスレッド実行における ARM 性能

まで FR-V の実行がストールしていることが考えられる.

4.4.3 マルチスレッド実行における全体性能

最後に ARM と FR-V の性能の和より全体性能を評価した結果を図 14 に示す. 図 14 より全体性能では SMT+FLUSH が最も高いことがわかる. これは OS スケ

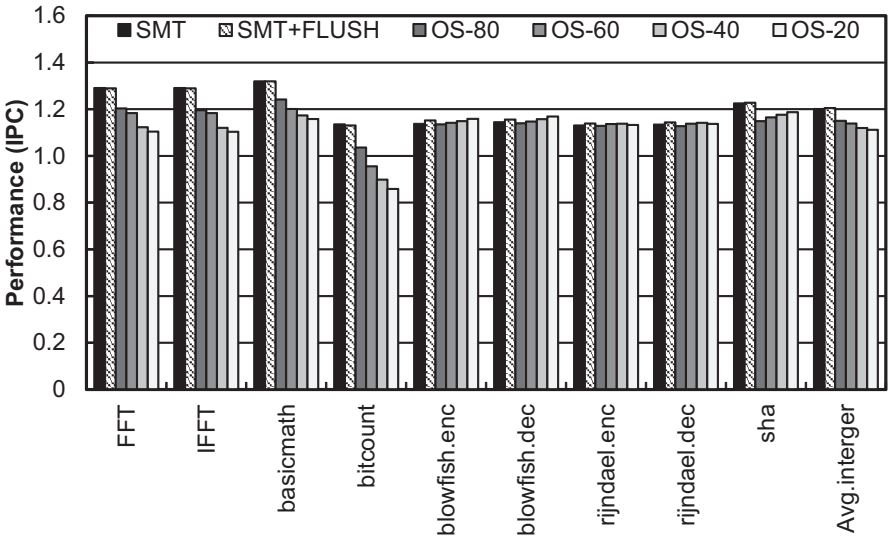


図 14 マルチスレッド実行における全体性能

ジューラがソフトウェアレベルによる数百サイクル単位の荒いスレッド制御よりも，命令フラッシュすなわちハードウェアによるサイクル単位の細かいスレッド制御のほうが，ARM の性能低下のオーバヘッドを低減できたためと考えられる．

以上より，OS スケジューラによるスレッド制御により優先スレッドをシングルスレッドの 95%まで向上させることができ，命令フラッシュにより全体性能を向上できることがわかった．したがって，OS スケジューラと命令フラッシュの両方のスレッド制御を実行状況に応じて使い分けることが有用であると考えられる．例えば，両スレッドに優先度を設けない状況では命令フラッシュのみでスレッド制御を行い，優先度がある場合は OS スケジューラによりスレッド制御を行うといったことができる．

4.5 全体評価

本節では，OROCHI のベースラインモデルを試作した結果である回路規模，遅延時間，性能および消費電力より，OROCHI とマルチコアプロセッサを比較し，電力効率で総合評価を行う．なお，マルチスレッド実行するベンチマークプログラムは 4.4 節と同じである．

4.5.1 評価モデルと LSI 試作結果

本項では，OROCHI の試作結果について述べ，比較対象となるマルチコアプロセッサについて述べる．3 章で述べた段階的開発手法を用いて 1 年当たり教員と学生を含め 8 名が開発に従事した結果，当初計画である研究期間の 3 年で OROCHI の LSI 試作を行い，図 2（c）の LSI モデルを用いた LSI の動作に成功した．

OROCHI の評価対象となるマルチコアプロセッサは高性能なスーパスカラ方式の ARM コアと一般的な VLIW 方式の FR-V コアを 1 コアずつ搭載する．そこで，OROCHI から ARM と FR-V のそれぞれを切り出し，Synopsys 社の Design Compiler を用いて ARM コアと FR-V コアの回路規模を見積り，その和を比較対象となるマルチコアプロセッサとする．ここで，マルチコアプロセッサはマルチコア構成で必要となるコア間調停回路を含まない理想的なモデルとした．マルチコアプロセッサを構成する ARM コアと FR-V コアのモデルを図 15 に示す．図 15（a）は図 3 の簡略図であり，OROCHI モデルを表し，白色は ARM 部分，灰色は FR-V 部分を表す．図 15（b）は OROCHI の ARM 部分より構成した ARM コアを表す．図 15（c）と図 15（d）は OROCHI モデルの FR-V 部分より構成した FR-V コアを表し，括弧内は VLIW 型命令キューの深さを表す．高性能なスーパスカラ方式の ARM コアでは，複雑な命令形式を持つ ARM 命令を単純な命令形式である内部命令に分解しスーパスカラ実行することが一般的であり，命令分解機構を持つ OROCHI の ARM フロントエンドを ARM コアに流用することができる．同様に，アウトオブオーダで実行した命令をプログラム順に並び替えるためのリオーダバッファおよびコミット機構を備えた ARM バックエンドは OROCHI の共通バックエンドの ARM 部分と同等である．さらに，文献 [7] は VLIW 型命令キューを用いたアウトオブオーダ型スーパスカラ方式が従来スーパスカラ方式よりも小回路規模かつ高性能であることを実証している．よって，マルチコアプロセッサに図 15（b）を採用した．一方，FR-V コアは一般的な VLIW プロセッサであり，FR-V フロントエンドおよび FR-V バックエンドは OROCHI の FR-V 部分と同等である．しかし，VLIW 型命令キューの深さは 1 で十分であるため，単純に OROCHI の FR-V 部分を切り出した図 15（c）ではなく図 15（d）をマルチコアプロセッサに採用した．以上より，マルチコアプロセッサを構成する ARM

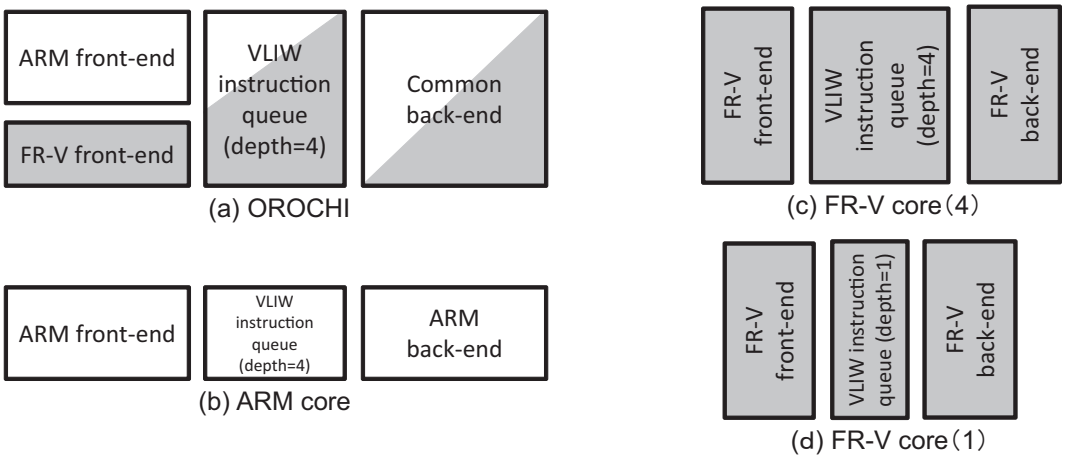


図 15 評価モデル

表 4 各プロセッサコアの回路規模および動作周波数

	Chip area [Kgates]	Clock frequency [MHz]	Area ratio
ARM core	879	95 (Logic base)	0.58
FR-V core (4)	652	96 (Logic base)	-
FR-V core (1)	643	96 (Logic base)	0.42
Multi-core	1,522	-	1.00
OROCHI	1,205	97 (Logic base)	0.79
		43 (Layout base)	

コアと FR-V コアの構成は妥当なものであるといえる．

OROCHI の試作結果を表 4 に示す．表 4 より，ARM コアは 879K ゲート，FR-V コアは VLIW 型命令キューの深さが 4 および 1 の場合でそれぞれ 652K ゲートと 643K ゲートであることがわかり，ARM コアが FR-V コアよりも回路規模が大きいことがわかる．マルチコアプロセッサの回路規模は ARM コアおよび FR-V（1）の和である 1,522K ゲートとした．一方，OROCHI の回路規模は 1,205K ゲートとなり，マルチコアプロセッサの 79%の回路規模であることがわかる．また，各コアの動作周波数を見積もった結果，論理合成の段階では，ARM コアと FR-V コアがそれぞれ 95MHz と 96MHz となり，OROCHI は 97MHz であることから OROCHI

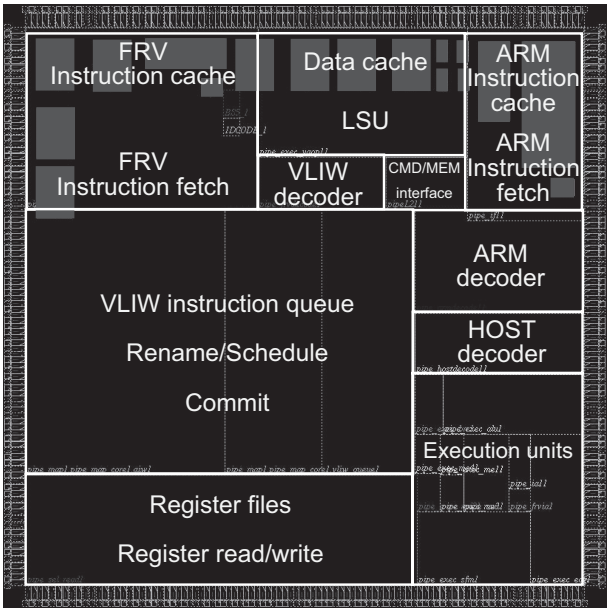


図 16 OROCHI のフロアプラン

による動作周波数の低下はないことがわかる．さらに，OROCHI の LSI 試作により物理設計を行った結果，動作周波数は 43MHz であることがわかった．これは設計段階で階層設計を取り入れなかったことにより配置配線が困難になったためと考えられる．

OROCHI のフロアプランを図 16 に示す．図 16 より VLIW 型命令キューとレジスタファイルがチップの中央部に配置され，他モジュールに囲まれていることがわかる．各モジュールの実装結果については文献 [60] を参照されたい．

4.5.2 性能評価

本項では，4.4 節の予備評価で測定した IPC と各モデルの論理合成の結果より，ARM と FR-V それぞれの IPC の和と各コアの動作周波数の積で表される 100 万実行命令毎秒（MIPS: Million Instructions Per Second）を総合性能として OROCHI とマルチコアプロセッサの比較を行う．ここで，マルチコアプロセッサの性能は，コア間通信およびメモリアクセスによる性能低下が無いものとして評価している．ベンチマークプログラムとして ARM-qsort と FRV-Mibench を同時実行したとき

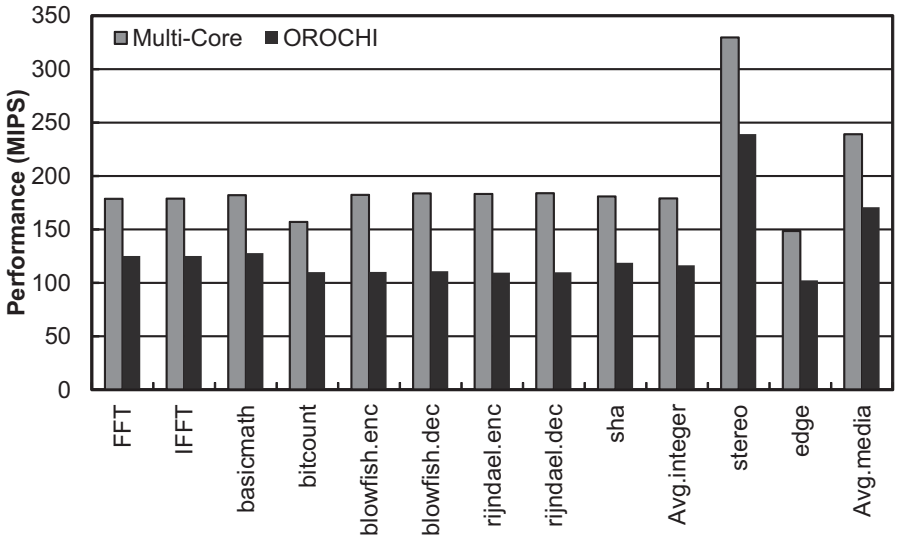


図 17 OROCHI とマルチコアプロセッサの総合性能

の OROCHI とマルチコアプロセッサの総合性能を図 17 に示す．汎用プログラムの平均（Avg.integer）に注目すると，OROCHI とマルチコアプロセッサの性能はそれぞれ 116MIPS と 179MIPS であることから，OROCHI はマルチコアプロセッサの 65%の性能を実現していることがわかる．また，マルチメディアプログラムの平均（Avg.media）ではそれぞれ 171MIPS と 239MIPS であることから，72%の性能を実現していることがわかる．

4.5.3 電力評価

本項では，OROCHI とマルチコアプロセッサの電力評価を行う．ベンチマークプログラムとして ARM-qsort と FRV-Mibench を同時実行したときの消費電力を図 2（c）で示した LSI モデルにて測定を行い，ARM と FR-V それぞれを実行したときの消費電力と回路規模の面積比からマルチコアプロセッサの電力を見積もり，OROCHI と比較する．

試作した OROCHI にはクロックゲーティングなどの低消費電力化技術は適用されていないため，電源とクロックのみを供給したアイドル時の消費電力は単純に回路規模に比例するものとした．まず，OROCHI のアイドル電力（*Pi*: $Power_{idle}$）

を Pi_{OROCHI} とし，表4より，OROCHIコア，ARMコア，FR-Vコア（1）およびFR-Vコア（4）の回路規模（A: $Area$）をそれぞれ A_{OROCHI}，A_{ARM}，A_{FRV1} および A_{FRV4} とし，ARMコアおよびFR-Vコア（1）のアイドル電力をそれぞれ Pi_{ARM} および Pi_{FRV1} とすると，

$$Pi_{ARM} = Pi_{OROCHI} \times (A_{ARM}/A_{OROCHI})$$
$$Pi_{FRV1} = Pi_{OROCHI} \times (A_{FRV1}/A_{OROCHI})$$

となる．次に，OROCHIでARMのプログラムを単体実行したとき，OROCHIのFR-V部分のスイッチングは発生せず，ARM部分のスイッチングのみ発生するため，スイッチングによる消費電力はARMコアと同等になる．同様に，OROCHIでFR-Vのプログラムを単体実行したときのスイッチング電力はFR-Vコア（4）と同等になる．全FR-V命令はVLIW型命令キューを通過するため，FR-Vコア（1）のスイッチング電力はFR-Vコア（4）とそれらの面積比から求められる．よって，OROCHIでARMおよびFR-Vのプログラムをそれぞれ単体実行したときの消費電力（Pt: $Power_{total}$）を $Pt_{OROCHI-ARM}$ と $Pt_{OROCHI-FRV}$ とし，ARMコアとFR-Vコア（1）のスイッチング電力（Psw: $Power_{switch}$）をそれぞれ Psw_{ARM} と Psw_{FRV1} とすると，

$$Psw_{ARM} = Pt_{OROCHI-ARM} - Pi_{ARM}$$
$$Psw_{FRV1} = (Pt_{OROCHI-FRV} - Pi_{FRV4}) \times (A_{FRV1}/A_{FRV4})$$

となる．最後に，ARMコアとFR-Vコア（1）の消費電力はそれぞれのアイドル電力とスイッチング電力の和となり，ARMコアとFR-Vコア（1）の和がマルチコアプロセッサの消費電力となる．よって，ARMコア，FR-Vコア（1）およびマルチコアプロセッサの消費電力をそれぞれ Pt_{ARM}，Pt_{FRV1} および $Pt_{Multi-Core}$ とすると，

$$Pt_{ARM} = Pi_{ARM} + Psw_{ARM}$$
$$Pt_{FRV1} = Pi_{FRV1} + Psw_{FRV1}$$
$$Pt_{Multi-Core} = Pt_{ARM} + Pt_{FRV1}$$

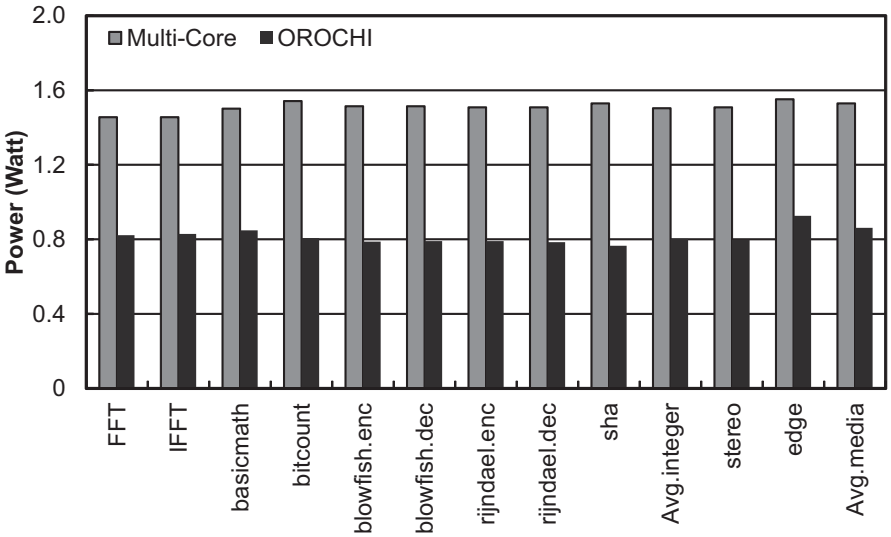


図 18 OROCHI とマルチコアプロセッサの消費電力

となる．

OROCHI とマルチコアプロセッサの消費電力を図 18 に示す．OROCHI の消費電力はマルチコアプロセッサよりも低いことがわかる．汎用プログラムの平均では，OROCHI とマルチコアプロセッサの消費電力はそれぞれ 0.80W と 1.50W であることから，OROCHI はマルチコアプロセッサの 53%の消費電力を実現していることがわかる．また，マルチメディアプログラムの平均ではそれぞれ 0.86W と 1.53W であることから，56%の消費電力を実現していることがわかる．

4.5.4 電力効率

本項では，ここまでの評価結果を踏まえて，総合評価として OROCHI とマルチコアプロセッサの電力効率を比較する．OROCHI とマルチコアプロセッサの電力効率を図 19 に示す．縦軸は電力効率として電力当たり性能（MIPS/W）を表す．OROCHI に注目すると，汎用プログラムとマルチメディアプログラムの平均がそれぞれ 145MIPS/W と 205MIPS/W であることがわかり，マルチメディアプログラム実行が汎用プログラム実行の 1.4 倍の電力効率を実現している．これは、前者では ARM と FR-V の両スレッドが整数演算器を使用するため，ARM スレッドが使

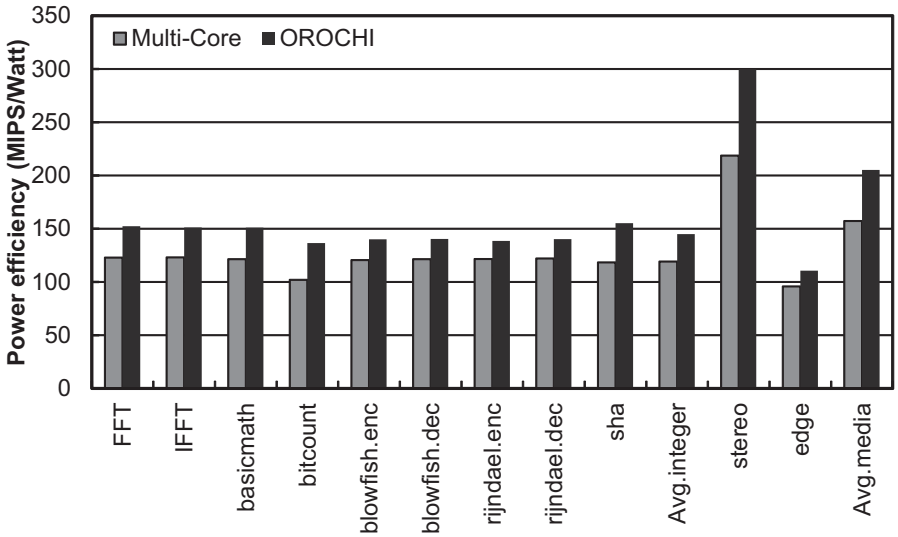


図 19 OROCHI とマルチコアプロセッサの電力効率

用できる空き整数演算器が少なく演算器使用率が低下する一方，後者では FR-V スレッドが整数演算器ではなくメディア用演算器を使用することに加えて ARM スレッドがより多くの空き整数演算器を使用できるため，演算器使用率が向上するからである．また，マルチコアプロセッサに注目すると，汎用プログラムとマルチメディアプログラムのそれぞれの平均が 119MIPS/W と 157MIPS/W であることがわかり，OROCHI の電力効率はマルチコアプロセッサの 1.22 倍と 1.31 倍であることがわかる．以上より，OROCHI はマルチコアプロセッサよりも高い電力効率を実現できたといえる．

4.6 結言

本章では，VLIW 型命令キューを用いて VLIW 方式の余剰演算能力を利用し，他スレッドをスーパスカラ実行することでヘテロジニアスマルチスレッド実行する異種命令混在実行プロセッサ OROCHI を提案した．そして，OROCHI において従来手法である OS スケジューラによるソフトウェア制御と提案手法である VLIW 型命令キューによるハードウェア制御の 2 つのスレッド制御手法についてソフトウェアシミュレータを用いた予備評価を行った．その結果，従来手法により優先

スレッドである FR-V の性能を 95%まで向上できることを示し，スレッド制御を適用しないベースモデルと比較して提案手法により 2%の性能を向上できたことを示した．さらに，OROCHI の開発ではソフトウェアシミュレータによる論理検証，ハードウェア設計・検証および LSI 試作後の評価を段階的に行う開発手法を用いて 8 人/年・研究期間 3 年の開発で LSI 動作に成功した．試作した OROCHI の回路規模，動作周波数，性能および消費電力より電力効率による総合評価を行った結果，OROCHI はマルチコアプロセッサと比較して 79%の回路規模で，汎用プログラムとマルチメディアプログラムを同時実行した場合に 1.31 倍の電力効率を実現した．

5. 線形演算器アレイ型パイプラインプロセッサLAPP

本章では，メニコアアーキテクチャと粗粒度再構成アーキテクチャの長所を組み合わせ，プログラマビリティと高い電力効率を実現する，線形演算器アレイ型パイプラインプロセッサ LAPP の概要について述べる．次に，LAPP でプログラムを高速実行するために必要な値伝搬について述べ，評価モデルを示す．さらに，既存機械語命令を演算器アレイに割り当てるための命令写像手法を提案し，そのアルゴリズムとサンプルプログラムによる命令写像例を示す．その後，予備評価としてハードウェア設計した LAPP の各モジュールの論理合成結果より，値伝搬の評価モデルと命令写像機構の回路規模と遅延時間を評価する．最後に，全体評価として LAPP の LSI 試作により実現性を証明した後，LAPP とメニコアプロセッサの回路規模，性能および消費電力を評価し，電力効率において LAPP が優位であることを示す．

5.1 緒言

本節では，プログラマビリティについて議論した後，CGRA の課題からプログラマビリティと高電力効率の両方を実現する LAPP の仕組みを述べる． CGRA はプログラム中のデータフローグラフを多数の演算器に割り当て高速実行するため，専用の命令セット [32] やコンパイラ [33] を必要とする．PipeRench[61] や PARS[62] では，大規模な CGRA に対してデータフローグラフ を用いたスケジューリングを適用することで高性能を目指している．しかし，大規模であるが故に既存プロセッサとは大きく異なる構成となっており，全体の回路規模増大を抑えるために演算器アレイ中に配置されるレジスタ数が限定され，高度なスケジューリング機能が要求される．そこで，LAPP では既存の VLIW 命令セットをそのまま使用し，既存のデータプリフェッチ命令によって，高速実行に必要なヒント情報をプロセッサに通知する方式を採ることにした．VLIW 命令セットを用いることで既存のコンパイラ技術を応用してプログラムの開発が可能であり，プロセッサにアクセラレータの機能が未実装の場合は，データプリフェッチ命令は NOP 命令扱いとすることにより，正しくプログラムを実行できる．このように，上位互換性だけでな

く，高速実行するプログラムを既存 VLIW プロセッサでも実行できる下位互換性を備えている．プログラム開発者にとって 5.2.1 項で後述する制約条件はあるものの，専用言語の習得や特別なプログラミングの知識は不要である．さらに，既存 VLIW プロセッサで動作しているプログラムの中から必要な箇所のみを順次高速実行に対応させるといった段階的なプログラム開発も可能となる．

CGRA がより複雑かつ大規模なプログラムを実行するためには，さらに多くの演算器を投入しなければならない．しかし，演算器数の増加により演算器ネットワークが複雑化するとともに，専用コンパイラによるスケジューリングやプログラムの生成の難易度が高くなることが予想される．そこで，LAPP では既存 VLIW プロセッサを初段として，VLIW プロセッサのレジスタファイル，演算器およびメモリアクセス機構から成るバックエンドをアレイ段として線形に拡張する．そして，ループカーネル中の VLIW 命令列を演算器アレイに写像し，パイプライン実行することで演算器アレイを専用ハードウェアと同様に毎サイクル動作させる．ここで，初段と最終段がデータ 1 次キャッシュ（L1 キャッシュ） に隣接するようなリング構造を想定すると，L1 キャッシュは初段および最終段のみと接続されている．実行するループカーネルの VLIW 命令数に応じて初段と最終段の間に任意のアレイ段数を挿入することができ，アレイ段数に関する拡張性が高いといえる．また，データが一方向に流れるため，段数によらず隣接する段間の接続は一定であり，演算器ネットワークの複雑化を回避している．

最後に，CGRA のメモリアクセス機構はメッシュネットワークを用いたもの [31] や，すべてのパイプラインステージにデータキャッシュを持つもの [37] があり，それぞれルーティングおよびキャッシュミス時にストールが発生し，高速実行の性能向上を阻んでいる．そこで，LAPP ではすべてのパイプラインステージが実現可能な個数のロード・ストアユニットに加えて，データキャッシュではなくローカルデータキャッシュ（L0 キャッシュ）を備える．そして，高速実行前にデータプリフェッチ命令によりデータキャッシュへ必要なデータをプリフェッチしておき，高速実行中は各パイプラインステージに必要なデータを供給することで，高速実行中のキャッシュミスを回避し，毎サイクル演算器が命令を実行できるようにした．

電力の観点からみたプロセッサの理想的状態は，必要最低限のデータメモリ，レジスタおよび演算器のみが動作する状態である．文献 [63] は，従来シングルプロセッサにおいて命令キャッシュやデータキャッシュの電力消費が全体の 54%を占めることを報告している．既存研究においては，クロックゲーティング（CG: Clock Gating）[64]，パワーゲーティング（PG: Power Gating）[65] および DVS（Dynamic Voltage Scaling）[66] を適用し，低電力化を図っている．しかし，高速化のために付加したキャッシュ等の電力を削減することは，性能低下に大きな影響がある．すなわち，パワーゲーティングによる低電力化のためには，長期間のユニット停止が必要である．LAPP では初段のみで実行する通常実行，高速実行前のアレイ設定，および，高速実行の 3 つの動作モードで未使用ユニットを長期間計画的に停止し，低消費電力化技術の適用が容易となる仕組みを持つ．

以上により，LAPP は高いプログラマビリティを持ち，既存の CGRA の主な問題点を解決した仕組みを有し，メニコアプロセッサよりも低消費電力化を実現できる．

5.2 概要

既存プロセッサ技術を応用しつつ高性能かつ低消費電力を実現するためには，5.1 節で述べたように多数の演算器をストールすることなく実行する仕組みと既存の低消費電力技術と組み合わせることができる実行の仕組みを考えなければならない．本節では，これらの仕組みを実現する LAPP の概要について述べる．

5.2.1 制約条件

本項では LAPP でプログラムを高速実行するために満たすべき制約条件について述べる．

アレイ段数

ループカーネル中の VLIW 命令数がアレイ段数よりも多いときは，そのままでは演算器アレイで高速実行できないため，事前にループを分割するなどの対応が必要である．もし，実行時にアレイ段数を超えていることが判明した場合は，演算器アレイでの実行を中止し初段のみにより通常実行することで互換性を維持で

きる．一般的なCGRAにおいても，構成情報を保存するメモリ容量などにより実行可能な命令数には制限があるものが多い．

依存関係

i回目のイタレーションの実行結果を$i+1$回目のイタレーションで参照する場合のような，イタレーションを越える依存関係が存在する場合は，基本的に演算器アレイで実行することはできない．これは配線の複雑化のような物理的な問題だけではなく，そもそも複数のループイタレーションがパイプライン実行により並行実行されているため，結果が必要となる時刻においてその結果を出力する演算が実行されていない可能性があるためである．これは多数の演算器を用いる場合には常に発生しうる問題であり，根本的な解決のためにはアルゴリズムレベルでの変更が必要となる．ただし，このような依存関係を全く考慮しない場合にはループカウンタの更新すらできなくなってしまうため特別な対応が必要である．具体的には$i += N$で表現される命令（自己更新命令）の実行をサポートする．このためには，演算器の出力を自分自身の入力にフォワーディングする回路（自己ループ）の追加を行えばよい．一斉演算時には初回のみ前段から初期値iを受け取りNを加える．次のサイクルからは追加したフォワーディング回路を通して自演算器の出力を受け取り，それにNを加えればよい．この構造からNはループ中で変更されない定数であることも必須条件である．

ループカーネル

ループカーネルはプリフェッチ命令と後方無条件分岐をそれぞれループの先頭および終端と定め，前方条件分岐によりループを脱出するものとし，ループカーネル中のVLIW命令列を演算器アレイに写像する．演算器アレイにデータを正確にプリフェッチするためには，ループの実行回数が既知である必要がある．もし，プリフェッチサイズがL1キャッシュの容量を超過する場合には，ループを複数回に分割する必要があるが，これは既存プロセッサにおけるブロック化のようなキャッシュ最適化と同等の難易度である．さらに，簡単な拡張によりループ途中での脱出をサポートすることは可能であるが，その場合もループ回数の最大値を保証する必要がある．また，この場合はプリフェッチデータが無駄になる可能性があるので，そのオーバヘッドを評価する必要がある．本研究では，定数回の

ループのみを評価対象とする．

ロード・ストア命令

分散配置したL0キャッシュによってロード命令に対してデータ供給を行うためには，ループ中で実行可能なロード命令に次のような制約がある．まず，データを正しくプリフェッチするためには，すべてのロード命令のアクセスパターンが解析可能である必要がある．具体的には連続アクセスか，定数幅のストライドアクセスの場合のみをサポートする．また容量の制約から，1つのイタレーション中でアクセスされるデータが各L0キャッシュに収まる必要がある．

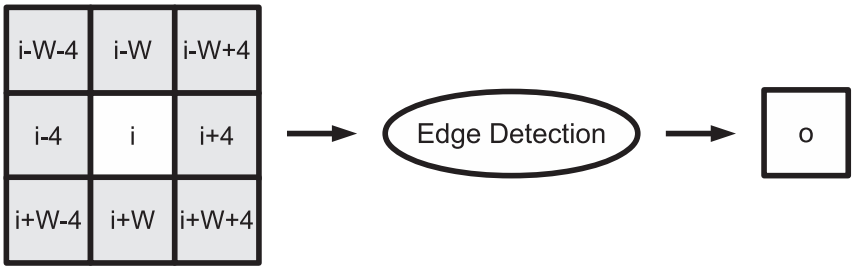
以上の制約条件をすべて満たしたループのみが演算器アレイによる高速実行の対象となる．

5.2.2 サンプルプログラムの実行例

5.2.1項で述べた制約を満たすプログラムのLAPPにおける実行例を図20に示す．図20（a）がサンプルコードである．このサンプルコードは図20（b）に示す3×3の画素に対して輪郭抽出を実行するプログラムである．周辺画素をロードし，対角画素の絶対値差分（SAD：Sum of Absolute Difference）を求め，閾値（T）との大小比較により輪郭情報として白（255）または黒（0）の画素値を中心画素にストアする．図20（c）にアセンブリコードを示し，図21に図20の命令写像およびネットワーク構築が完了した後の演算器アレイを示す．ここで，ロード・ストア 命令はアドレス計算と実際のデータ転送の2サイクルに分けて実行されるとした．図21では，図20（c）に示したVLIW 命令の各々が，各演算器に対応付けられている．初段のレジスタファイルから読み出した値を用いて，初段の演算器がVLIW0のkのデクリメント（$k--$）とロード命令（ld）の実行に必要なアドレス計算（$i-4$および$i+4$）を行う．アドレス計算（eag）結果に基づき，初段のロード・ストアユニットが内蔵するL0キャッシュを参照する．通常のVLIW動作では，ロード結果が初段のレジスタファイルに書き込まれ，必要に応じて初段の演算器にフォワーディングされる．一方，アレイ動作では，ロード結果がVLIW2のSAD命令（sad）に対応付けられた演算器にフォワーディングされる．同様に，VLIW1，VLIW2およびVLIW3のロード結果がそれぞれVLIW3，

```
for (k=W-2; k>0; k--) {
    if (SAD(*(i-1),*(i+1))+SAD(*(i-W-1),*(i+W+1))
        +SAD(*(i-W),*(i+W))+SAD(*(i-W+1),*(i+W-1))>T)
        *o = 255;
    else
        *o = 0;
    i++; o++;
}
```

(a) Program code



(b) Input / output data

```
loop:
VLIW0  ld @(i-4)    →1  ld @(i+4)    →2                  k-- →c
VLIW1  ld @(i-W-4)→3  ld @(i+W+4)→4                                  bz c, end
VLIW2  ld @(i-W)    →5  ld @(i+W)    →6  sad 1, 2 →x
VLIW3  ld @(i-W+4)→7  ld @(i+W-4)→8  sad 3, 4 →y     i+=4
VLIW4                                 sad 5, 6 →z     add x, y →p
VLIW5                                 sad 7, 8 →w     add z, p →p
VLIW6                                                 add w, p →p
VLIW7                                 cmp p, T →c
VLIW8                                 sel c →s
VLIW9  st s, @(o)                     o++                            bra loop
end:
```

(c) Assembly code

図 20 輪郭抽出プログラム

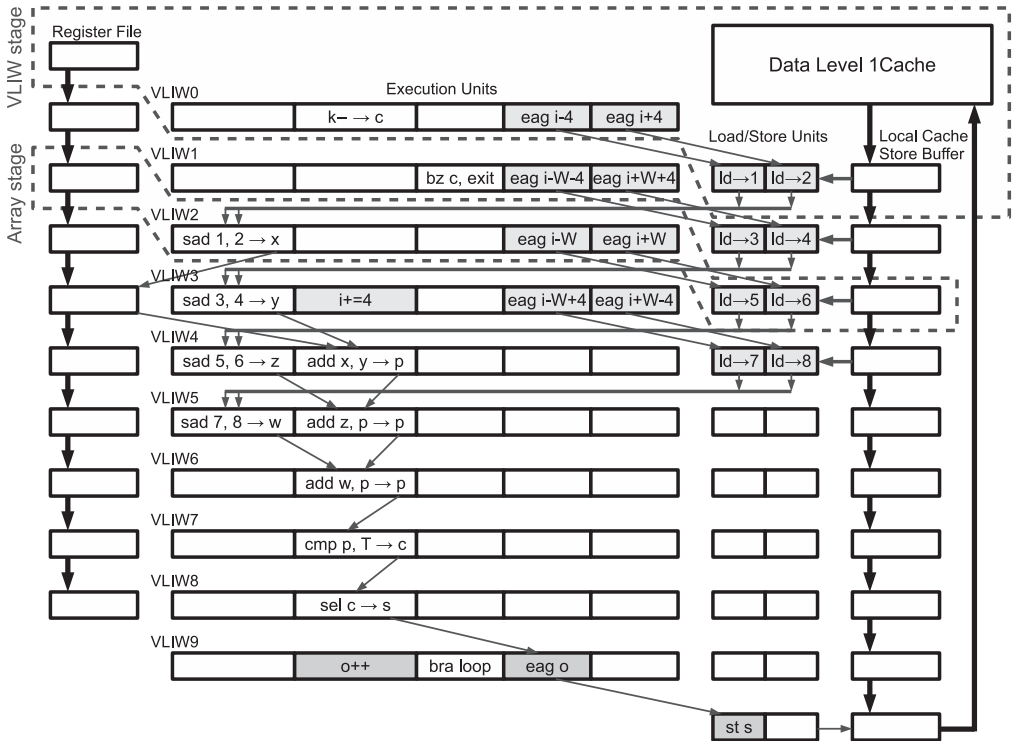


図 21 LAPP における輪郭抽出プログラムの実行

VLIW4 および VLIW5 の SAD 命令にフォワーディングされる．SAD 命令の結果は VLIW4，VLIW5 および VLIW6 の加算命令（add）にフォワーディングされ総和（p）が計算される．その後，VLIW7 の比較命令（cmp）で大小比較した結果（c）に基づき，選択命令（sel）で白（255）または黒（0）を画素値（s）に代入する．アレイ構造の 1 段を通過するのに 1 サイクルを要する場合，VLIW0 において最初のロードを開始してから，VLIW9 において結果（s）がストアされるまでに 10 サイクルを要する．この点では，従来の VLIW プロセッサによる実行と何ら変わらない．しかし，全段をパイプライン動作させることにより，各演算器に異なるイタレーションの同一命令を毎サイクル実行させ，最終的に，VLIW9 のストア命令を毎サイクル実行できる．その後，後方無条件分岐（bra）がループカーネルの終端を表しており，本ループカーネルを所定のループ回数分のループイタレーションをパイプライン実行する．所定のループ回数を実行すると前方条

件分岐（bz）が成立することで高速実行を上段から下段へ順次終了し，通常実行に復帰する．ここでアレイ実行中のロード・ストア命令を正しく実行するためには，VLIW3 のループカウンタの更新（$i += 4$）を VLIW0，VLIW1，VLIW2 および VLIW3 の先行ロード命令のアドレス計算（eag）に反映しなければならない．同様に，VLIW9 の加算計算（$o++$）を VLIW9 のストア命令のアドレス計算に反映しなければならない．具体的には，これらの命令を検出したときに先行ロード・ストア命令に対して加算および減算の即値（ここでは 4 および 1）を渡し，先行ロード・ストア命令のアドレス計算で同じ加算および減算をさせればよい．これらの反映を含め，命令写像およびネットワーク構築を実現する命令写像手法は 5.3 節で述べる．

5.2.3 モジュールモデルおよび実行モード

5.1 節，5.2.1 項および 5.2.2 項より図 22 に示す LAPP のモジュール構成に至る．LAPP は初段（VLIW stage）とアレイ段（Array stage）の 2 つから構成される．初段は従来 VLIW プロセッサと同様の機能を持ち，命令 1 次キャッシュ（I1 キャッシュ）およびデータ 1 次キャッシュ（L1 キャッシュ）を備え，プログラムカウンタ（PC: Program Counter），命令フェッチユニット（IF: Instruction Fetch），命令デコーダ（ID: Instruction Decoder），レジスタファイル（RF: Register Files），演算実行ユニット（EXEC: Execution Unit），ロード・ストアユニット（LSU: Load/Store Unit）から構成される．また，初段はアレイ段としても動作するため，高速実行に必要な命令写像ユニット（MAP: Instruction Mapper）とローカルデータキャッシュ（L0$: Level 0 Data Cache ）を持つ．アレイ段は演算実行ユニット，ロード・ストアユニット，命令写像ユニットおよびローカルメモリに加えて，初段レジスタファイルと演算器出力を後段へ伝搬するための段間セレクタ（SEL: Selector）を持つ．さらに，3 章で述べたコマンドインタフェース（CMD IF: Command Interface）およびメモリインタフェース（MEM IF: Memory Interface）を備え，それぞれホスト PC と，共有 2 次キャッシュ（L2 キャッシュ）および主記憶と通信を行う．

初段は通常実行において，既存機械語命令で生成されたプログラムを実行し，アレイ実行中はループカーネル中の最初の VLIW 命令が割り当てられるアレイ段

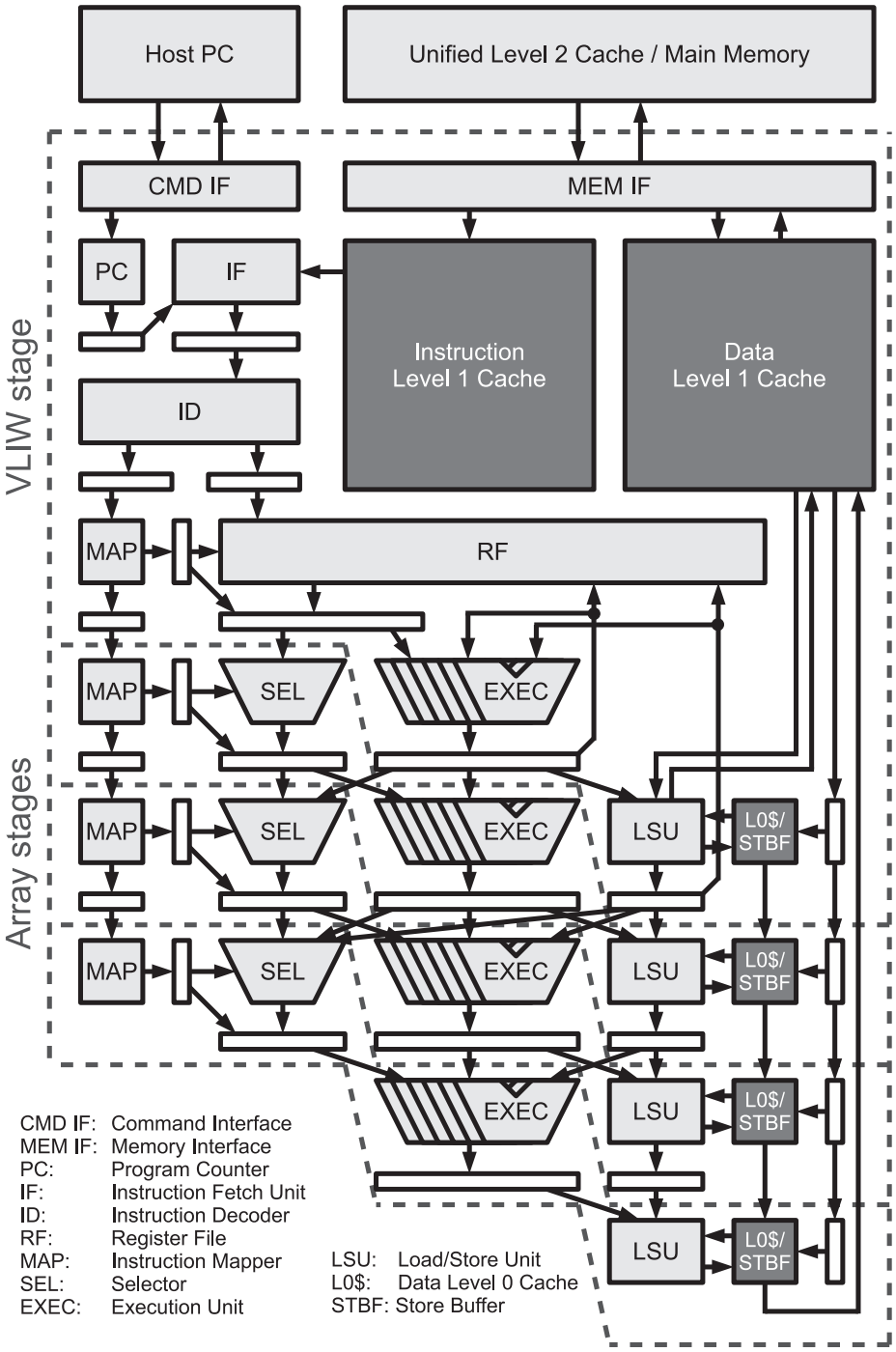


図 22 LAPP のモジュール構成およびその接続

として機能する．一方，アレイ段は初段の演算器およびメモリアクセスユニットから成り，ループカーネル中の VLIW 命令列が写像され，高速実行する．

LAPP は通常実行，アレイ設定およびアレイ実行の 3 つの動作モードを備える．まず，通常実行モードでは初段が従来 VLIW プロセッサと同様の仕組みによりプログラム中の VLIW 命令列を実行する．プログラム中にループカーネルを検出すると，アレイ設定モードに遷移し，ループカーネル中の VLIW 命令列をアレイ段に割り当てる．命令写像機構は VLIW 命令列を演算器アレイに写像し，VLIW 命令列中のレジスタ番号に基づいて段間セレクタを切り替えることで演算器ネットワークを構築する．命令写像と同時に，アレイ実行モードにて使用する入力データおよび出力データを L2 キャッシュおよび主記憶から L1 キャッシュにデータプリフェッチを行う．このとき，データプリフェッチが十分長ければ，命令写像によるサイクルオーバヘッドを隠蔽できる．命令写像およびデータプリフェッチが完了した後，アレイ設定モードからアレイ実行モードに遷移する．アレイ実行モードでは，演算器が毎サイクルパイプライン実行し，L1 キャッシュから必要なデータをアレイ段の上段から下段へ流し込む．ソースレジスタとデスティネーションレジスタの値は上段から下段へ伝搬され，ループカーネルの各イタレーションが並列実行される．これにより，LAPP が 1 サイクルに実行できる命令数（IPC: Instruction Per Cycle）の最大値はループカーネル中の命令数と同等になる．

5.2.4 低消費電力技術

本項では，LAPP に適用する低消費電力技術として，クロックゲーティング（CG: Clock Gating）[64]，パワーゲーティング（PG: Power Gating）[65] および DVS（Dynamic Voltage Scaling）[66] の 3 つと，それらを適用するモジュールについて述べる．

まず，クロックゲーティングは不要な回路のクロック供給を停止することで，レジスタの値の切り替わりを抑制し，動的消費電力を削減する手法である．クロックゲーティングによるクロック供給の停止は 1 サイクル以内で行うことができ，LAPP でも同様にすべての論理回路に対してクロックゲーティングを適用できる．また，I1 キャッシュおよび L1 キャッシュにおいてキャッシュミスした場合，

L2 キャッシュおよび主記憶にアクセスしている間，I1 キャッシュおよび L1 キャッシュにクロックゲーティングを適用できる．ただし，クロックゲーティングではこれらの論理およびメモリのリーク電流による静的消費電力を削減することはできない．

そこで，静的消費電力を削減する手法であるパワーゲーティングは機能ユニットごとに電源電圧回路にスイッチを設け，不要回路への電源供給を停止する．電源供給を停止することによりレジスタとメモリの内容は失われるものの，リーク電流を削減できるため，静的消費電力を削減できる．しかし，電源の切り替えには，時間および電力の切り替わりオーバヘッドが生じるため，損益分岐時間（BET: Break Even Time）を超える期間で切り替えなければ，かえって消費電力が大きくなってしまう．LAPP では，まず，通常実行モードにて未使用状態にあるアレイ段に対してパワーゲーティングを適用できる．また，アレイ実行モードの間，未使用ユニットに対してパワーゲーティングを適用できる．そのために，アレイ設定モードにてアレイ段の MAP を起動し，VLIW 命令列の演算器アレイへの写像が完了した時点で，PC，IF および ID を含むフロントエンドと VLIW 命令が割り当てられなかった機能ユニットはアレイ実行モードの間，未使用であることを確定すればよい．

一方，レジスタとメモリの内容を保持しつつ低消費電力化を図るために，内容を保持できる限界まで供給電圧を下げる DVS を適用する．LAPP では，アレイ実行モードにおいて I1 キャッシュおよびレジスタファイルには書き込みが行われないため，DVS を適用できる．

5.2.5 データプリフェッチおよびメモリアクセス

多数の機能ユニットを使用するプロセッサにおいて，高速実行中にメモリアクセスがキャッシュミスしストールが発生すると，全体の性能が低下するだけでなく演算器およびレジスタでストール中も一定の電力を消費してしまう．そこで，LAPP は高速実行する前に，書き出し先確保のためにデータキャッシュのフラッシュを行い，L2 キャッシュへデータを追い出した後，主記憶および L2 キャッシュから L1 キャッシュへ必要なデータをプリフェッチする．これにより高速実行中は

ストールしないことを保証する．

ただし，任意の VLIW 命令列を演算器アレイに写像するには，任意の段にロード命令を配置できる必要がある．しかし，数多くのロード命令が L1 キャッシュを直接参照することは，キャッシュメモリに必要なポート数の点から非現実的である．このため，論理的には全段が共通の L1 キャッシュの内容を参照できるものの，物理的には各ロード命令は同一段の小容量 L0 キャッシュのみを参照する仕組みが必要である．

ここで，イタレーション間に依存関係がないことに注目すると，各アレイ段が実行するイタレーションの必要なデータを供給できればよい．つまり，初段レジスタファイル，先行命令の演算結果のフォワーディングおよびロードされるデータをイタレーションの実行に間に合わせればよい．そこで，ロードされるデータを前段から後段へ流し込みながら，各アレイ段に配置された L0 キャッシュに順次キャッシュしていく．各アレイ段のロード命令がアクセスするデータに空間的局所性があることを利用すれば，各アレイ段のロード命令は必要とするデータを L0 キャッシュから読み出すことができる．

初段の L0 キャッシュのうち毎サイクル更新されるデータは，L1 キャッシュから送り込まれるデータのみである．このため，L0 の全内容を次段へコピーする必要はなく，L1 キャッシュ から送り込まれた内容のみを次段へ送れば十分である．すなわち，各イタレーションが 3×3 の画素を扱う場合，毎サイクル 3 ワードのみを次段の L0 キャッシュに転送すればよいため，実現は容易である．2 段目以降についても同様に，前段から送り込まれたデータのみを次段へ送ればよいため，段数の増加に対しても柔軟に対応できる．さらに，個々のワードについてはそれぞれがウェイに固定的に対応するため，段間での転送や L0 キャッシュへの取り込みにおいて 1 対 1 の単純な接続でよい．

また，任意段においてストア命令を実行可能とするために，各段にストアバッファを設ける必要があるものの，一般的な画像処理を考えた場合，L1 キャッシュに対するプリフェッチ性能以上にストア性能を確保する必要はない．すなわち，任意の段が生成したストアデータを各段が自由に L1 キャッシュへ書き戻せる必要はなく，次段へ順次伝搬させ，最終段のストアバッファから L1 キャッシュに毎サ

イクル 1 画素を書き出せればよい．このためには，各段につき 1 ワードのストアバッファがあれば十分であり，次段への全コピーは容易に実現できる．なお，この構成は，ストア命令の記述を 1 箇所に制限するものではない．前段から伝搬されるストアバッファに対し，異なるワードアドレスへのストアについては全体を上書きすることにより，前段のストアデータを後段においてロードするようなスピルアウト・スピルインや，汎用レジスタとメディアレジスタ間のデータ移動といった一時的なデータ転送に対応できる．また，同一ワードアドレスへの部分ストアについては，順にマージすることにより，バイト単位に生成した演算結果の逐次ストアに対応できる．すなわち，部分ストアを開始する前であれば，1 ワードのストアバッファを利用して任意回のストアとロードを実行できる．

図 23 に図 20 のロード・ストア命令の実行の様子を示す．4 つのウェイに分割された右端の L1 キャッシュのうち，3 つのウェイから初段の L1 キャッシュ読み出しバッファに対して，毎サイクル 3 ワードを読み出す．次のサイクルにおいて，初段の L0 キャッシュと次段の L1 キャッシュ読み出しバッファに転送する．以降，同様の手順により後段の L0 キャッシュと L1 キャッシュ読み出しバッファに順次転送する．このとき，各アレイ段のロード命令は L0 キャッシュから必要な値を読み出すことができる．ストアバッファについても，新たなストアデータを反映しながら後段に順次転送し，最終段から L1 キャッシュへ書き出すことができる．

5.2.6 演算器ネットワークおよび伝搬レジスタ

各アレイ段で実行するイタレーションに必要なデータを供給するためには，初段のレジスタファイルおよび前段までに実行された先行命令の結果をフォワーディングしなければならない．各アレイ段が必要とするデータが全て前段の演算器からのフォワーディングにより供給される理想的な命令列を前提とすることはできないため，各段の演算器がレジスタを直接参照できる仕組みが必要である．しかし，多くの演算器が 1 組のレジスタファイルを参照することは，L1 キャッシュと同様にポート数の増加を招き，任意の段間のフォワーディングパスを導入すると，配線が膨大になるため非現実的である．そのため，レジスタファイルについても論理的には同一レジスタ空間を参照できるものの，物理的には各段に分散配置す

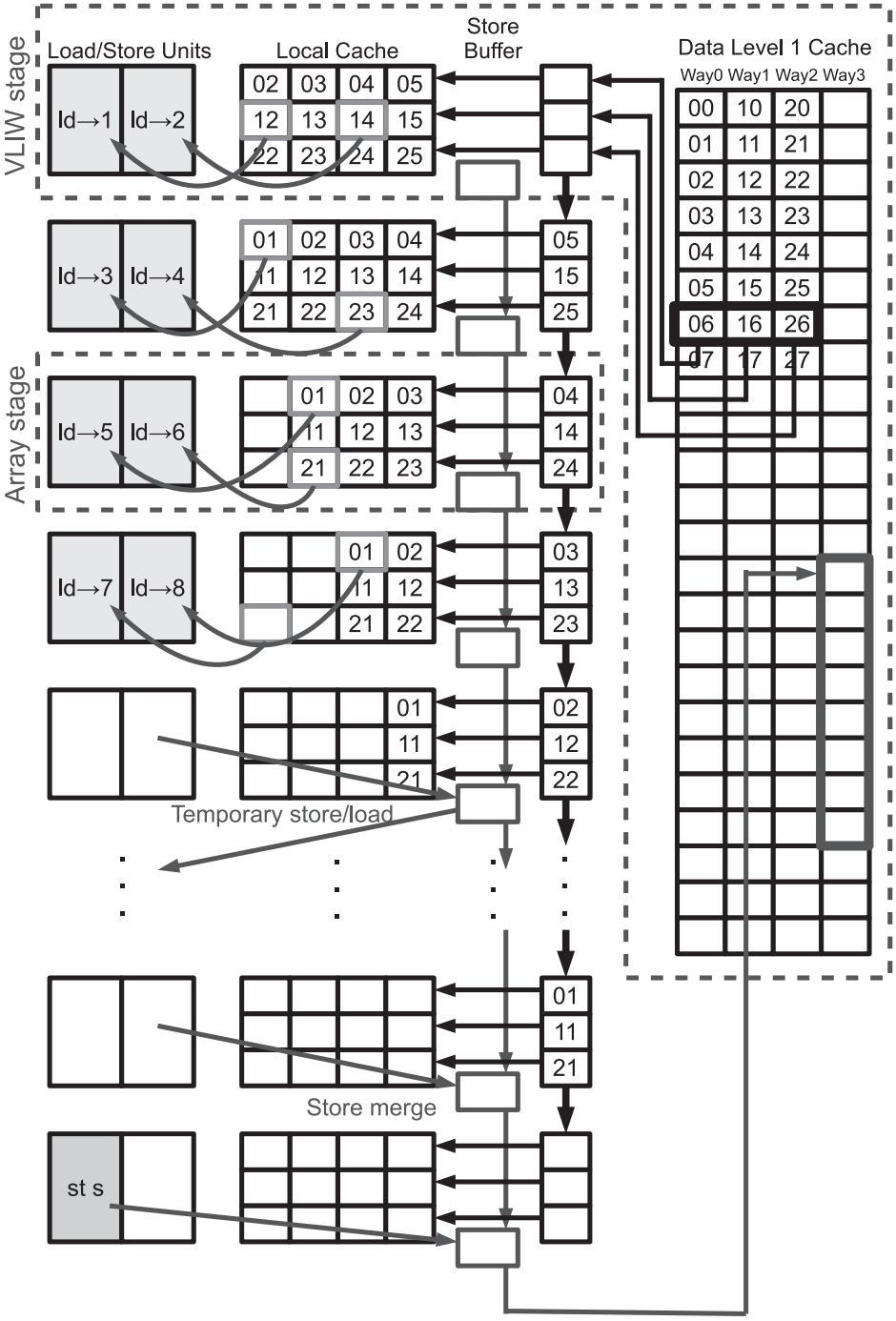


図 23 LAPP におけるロード・ストアの実行

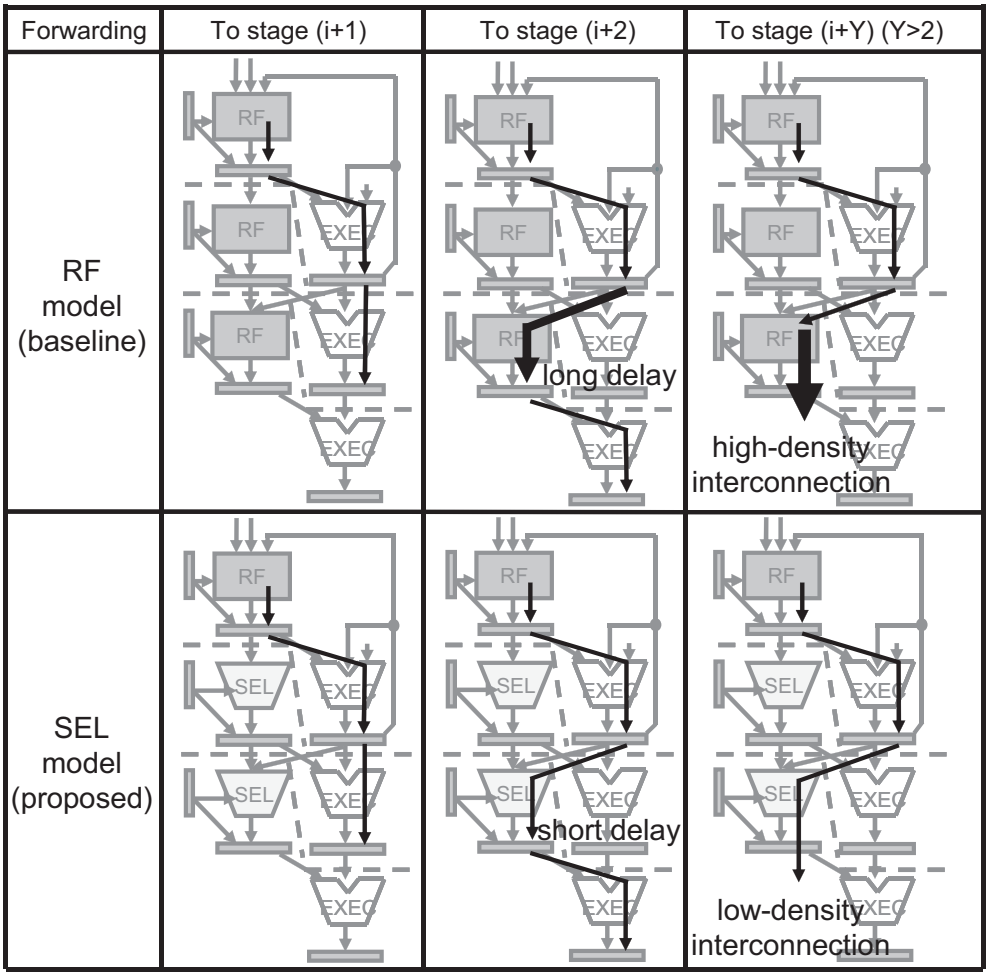


図 24 実行モデルとフォワーディング

る仕組みが必要となる．

そこで，初段レジスタファイルおよび先行命令の結果をイタレーションの実行速度に併せて各段が上段から下段へ伝搬し，その値を必要とするアレイ段まで届ければよい．この伝搬を実現するモデルを図 24 に示す．あるループカーネルで使用するレジスタの最大個数は命令セットで定義された論理レジスタ数であり，すべての論理レジスタの値を伝搬しなければならない．すべての論理レジスタを伝搬するためには，図 24 の RF モデルのように各アレイ段にレジスタファイルを配置するモデルとなる．i 段のレジスタファイルから値を読み出し演算された結果を $i+1$ 段の演算器で使用する場合は，i 段目演算器出力から次段演算器へフォワー

ディングすればよい．しかし，i段の演算結果を$i+2$段の演算器に供給するためには，$i+2$段のレジスタファイルに値を代入し，$i+2$段の演算器が使用する値を読み出さなければならないため，書き込み値を読み出しポートへバイパスする論理も含め，回路規模および遅延時間が増大してしまう．さらに，i段目の演算出力から$i+3$段以降の演算器に値を供給するためには，レジスタファイルに書き込み後，毎サイクル全ての値を次段レジスタファイルへ複製しなければならない．

ここで，すべての論理レジスタを読み出すループカーネルを持つプログラムは存在しないと仮定する．各段の空き演算器を利用して，それらの演算器入力レジスタをレジスタファイルのレジスタとして兼用することができれば，図24のSELモデルのように必要なレジスタ番号の値のみを伝搬することで回路規模および遅延時間を削減できる．まず，RFモデルと同様に，i段目の演算器から$i+1$段目の演算器へは演算結果をフォワーディングし，$i+2$段目の演算器への値の供給では，レジスタ数が減少したことでバイパス論理が削減され，遅延時間を削減できる．次に，$i+3$段目以降へも必要な値のみを選択して伝搬するため，レジスタファイルの値を複製するための配線を削減できる．このレジスタファイルのレジスタと演算器入力レジスタを兼用したレジスタを伝搬レジスタと呼ぶ．しかしながら，SELモデルを実現するためには，命令写像において論理レジスタ番号を伝搬レジスタに対応づけさせるレジスタリネーミングが必要となる．この演算器へのVLIW命令の割り当ておよびレジスタリネーミングについては5.3節で述べる．

ところで，先行研究[67]は画像処理ベンチマークを対象に必要な伝搬レジスタの個数を調査したところ，表5に示すレジスタファイルおよび演算器の構成にて伝搬レジスタが不足しないことを明らかにしている．初段には整数演算で使用する汎用レジスタファイルとメディア向けSIMD演算で使用するメディアレジスタファイルを配置する．汎用レジスタファイルは32ビットのレジスタが32個と，読み出し用に11ポートと書き込み用に5ポートを備える．一方，メディアレジスタファイルは32ビットのレジスタが32個と，読み出し用に7ポートと書き込み用に4ポートを備える．整数演算器群は，論理演算ユニット（ALU: Arithmetic and Logic Unit）が3個，アドレス計算ユニット（EAG: Effective Address Generator）が1個，ロード・ストアユニット（LSU: Load/Store Unit）が1個，分岐演算器（BRC:

表 5 画像処理ベンチマーク向けの LAPP 構成

General register file (GR)	32 entry 11R5W port
Media register file (MR)	32 entry 7R4W port
Function unit	3 ALU, 1 BRC, 1 EAG
	1 LSU, 4 MEDIA
Propagation register for GR	11
Propagation register for MR	9

Branch Unit）から成る．一方，メディア演算器群は，メディア演算器（MEDIA: Media Calculation Unit）が 4 個から成る．そして，伝搬レジスタは整数演算器群で 11 個，メディア演算器群で 9 個から成る．本研究では，先行研究で用いた画像処理ベンチマークを使用し，命令写像時に伝搬レジスタが不足するプログラムを検出した場合は初段のみで通常実行することとする．

以降，整数演算器群および汎用レジスタファイルについて，RF モデルと SEL モデルを適用し，配線数，遅延時間および回路規模を議論する．図 25 に評価モデルを示す．図 25（a）は 32 個すべての論理レジスタを有する RF32 モデルである．図 25（b）は 11 個の物理レジスタを持ち，32 個の論理レジスタから必要なレジスタの値だけを選択し，物理レジスタのすべての値を後段へ複製する RF11 モデルである．図 25（c）は 11 個の伝搬レジスタを持ち，32 個の論理レジスタから必要なレジスタの値だけを選択し，演算器入力レジスタを兼用して後段へ伝搬する SEL11 モデルである．

図 25 より，各評価モデルでアレイ段間の配線数を $WIRE_{RF32}$, $WIRE_{RF11}$ および $WIRE_{SEL11}$ としたとき次のように求めることができる．

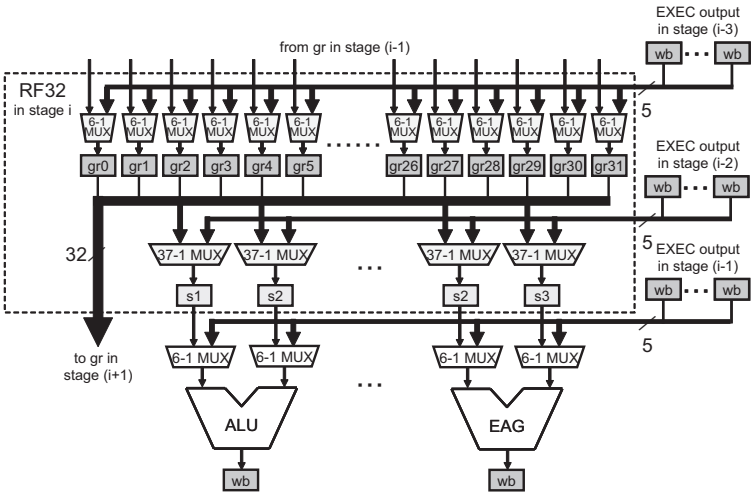
$$WIRE_{RF32} = 32\text{bit} \times 32(\text{No. of gr}) + 32\text{bit} \times 10(\text{No. of EXEC output}) = 1344$$

$$WIRE_{RF11} = 32\text{bit} \times 11(\text{No. of gr}) + 32\text{bit} \times 10(\text{No. of EXEC output}) = 672$$

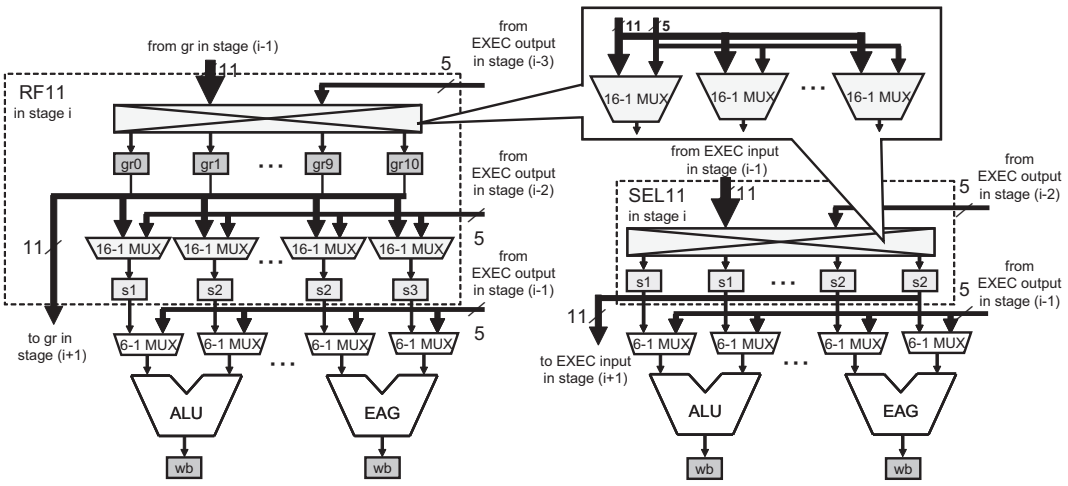
$$WIRE_{SEL11} = 32\text{bit} \times 11(\text{No. of EXEC input}) + 32\text{bit} \times 5(\text{No. of EXEC output}) = 512$$

よって，各評価モデルのアレイ段間の配線数の関係は次のようになる．

$$WIRE_{RF32} > WIRE_{RF11} > WIRE_{SEL11}$$



(a) RF32 model (baseline)



(b) RF11 model (preliminary)　　(c) SEL11 model (proposed)

図 25 評価モデル

同様に，各評価モデルの遅延時間を $DELAY_{RF32}$, $DELAY_{RF11}$ および $DELAY_{SEL11}$ とすると次のように求めることができる．まず，図 25（a）. より $DELAY_{RF32}$ は 32–1MUX の遅延時間となり，図 25（b）より $DELAY_{RF11}$ は 16–1MUX となり，図 25（c）より $DELAY_{SEL11}$ は 16–1MUX となる．よって，各評価モデルの遅延時間の関係は次のようになる．

$$DELAY_{RF32} > DELAY_{RF11} = DELAY_{SEL11}$$

次に各評価モデルの回路規模を求めるにあたり，RF32 モデルではすべての論理レジスタの値を読み出すことができる．一方，RF11 モデルおよび SEL11 モデルでは論理レジスタのうち最大 11 個の必要なレジスタを選択する論理が必要となる．そこで，必要なレジスタを選択する論理の回路規模を $AREA_{MAP}$ とし，6–1MUX, 16–1MUX, 37–1MUX, 汎用レジスタ（gr）および伝搬レジスタ (pr) の回路規模を $AREA_{6-1MUX}$, $AREA_{16-1MUX}$, $AREA_{37-1MUX}$, $AREA_{gr}$ および $AREA_{pr}$ とすると次のように求めることができる．

$$AREA_{RF32} = 32AREA_{6-1MUX} + 32AREA_{gr} + 11AREA_{37-1MUX} + 11AREA_{pr}$$

$$AREA_{RF11} = 22AREA_{16-1MUX} + 11AREA_{gr} + 11AREA_{pr} + AREA_{MAP}$$

$$AREA_{SEL11} = 11AREA_{16-1MUX} + 11AREA_{pr} + AREA_{MAP}$$

よって，RF11 モデルと SEL11 モデルの回路規模より，

$$AREA_{RF11} = AREA_{SEL11} + 11AREA_{16-1MUX} + 11AREA_{gr}$$

となり，RF11 モデルと SEL11 モデルの回路規模の関係は次のようになることがわかる．

$$AREA_{RF11} > AREA_{SEL11}$$

また，RF32 モデルと SEL11 モデルの回路規模より，

$$AREA_{RF32} = AREA_{SEL11} + 32AREA_{6-1MUX} + 32AREA_{gr}$$
$$+11AREA_{37-1MUX} - 11AREA_{16-1MUX} - AREA_{MAP}$$

となり，RF32 モデルと SEL11 モデルの回路規模の関係は MAP の回路規模によって決定されることがわかる．

以上を踏まえて，予備評価として 5.4 節で RF32 モデル，RF11 モデルおよび SEL11 モデルの回路規模および遅延時間を評価する．

5.3 命令写像手法

本節では，5.2.6 項で述べた RF11 モデルおよび SEL11 モデルのためのレジスタリネーミングジスタを含め，VLIW 命令列を演算器アレイに写像する命令写像

手法について述べた後，サンプルプログラムを用いて命令写像の例を示す．

5.3.1 アルゴリズム

VLIW 命令列を演算器アレイに写像するにあたり，初段のレジスタファイルから必要な値を伝搬するだけでなく，先行命令が更新したレジスタ番号を後続命令が参照する場合，レジスタファイルからの伝搬ではなく，先行命令の結果を伝搬しなければならない．そこで，各アレイ段が検査するための表を用意し，先行命令が更新したレジスタ番号はレジスタファイルからの伝搬が不要であることを逐次検査し，この表を更新すればよい．この表を伝搬不要表（prop_skp: Propagation Skip Table）と呼び，各アレイ段に設ける．伝搬不要表は論理レジスタ番号をインデックスとして，初期値はすべて 0 とする．伝搬不要表の値が 0 である場合，伝搬必要の可能性があることを示し，VLIW 命令のソースレジスタ番号に使用されていれば伝搬が必要であることがわかる．一方，伝搬不要表の値が 1 である場合，先行命令からフォワーディングできるため伝搬は不要であることがわかる．例えば，i 段目の命令のデスティネーションレジスタを p とし，0 段目から $i+1$ 段目の prop_skp[p] を 1 に更新する．これにより，レジスタファイルの論理レジスタ p は 0 段目から i 段目まで伝搬不要となり，$i+1$ 段目は i 段目の演算器出力レジスタから直接フォワーディングされる．VLIW 命令は自身が必要とするレジスタ番号が伝搬不要でなければ，割り当てられるアレイ段までレジスタ番号を伝搬するよう各アレイ段に設定していく．

伝搬不要表を用いた命令写像アルゴリズムを図 26 と図 27 に示す．図 26 は主な変数の定義と諸関数を表し，図 27 は命令写像のアルゴリズム本体を表している．

以降，命令写像アルゴリズムについて説明する．

(1) 命令写像はデータプリフェッチ命令の次の命令から開始する．

(2) i 番目の VLIW 命令は 0 段目から $i-1$ 段目に以下の設定を行う．

(2–1) 当該 VLIW 命令内の全てのソースレジスタ番号（s）をインデックスとして，伝搬不要表を参照し，prop_skp[s] が 1 ではなかった場合，当該ソースレジスタ番号を伝搬レジスタに割り当てる．（alloc_prop_reg()）内の assign_prop_reg()）

```
N  : the num. of VLIW instructions;
M  : the num. of stages;
L  : the num. of instructions in a VLIW instruction;
SRC: the num. of source registers in an instruction;
VLIW[x]: x-th VLIW instruction in a loop kernel;
op[x]  : x-th instruction in a VLIW instruction;
prop_reg[z]   : propagation registers of z-th stage;
prop_skp[y][z]: an entry indexed by register y in
                a propagation skip table of z-th stage;
exec[z]: execution units of z-th stage;
s: source register number
d: destination register number

alloc_prop_reg(i, j) {
  for (k=0; k<L; k++) {     // for each instruction
    for (l=0; l<SRC; l++) { // for each source register
      s=VLIW[i].op[k].src_reg[l];
      if (prop_skp[j][s]!=1) {
        assign_prop_reg(s, prop_reg[j]);
      }
    }
  }
}
update_prop_skp(i, j) {
  for (k=0; k<L; k++) {      // for each instruction
    d=VLIW[i].op[k].dst_reg; // for destination register
    prop_skp[j][d]=1;
  }
}
update_prop_skp_onlyLD(i, j) {
  for (k=0; k<L; k++) {        // for each instruction
    if (isLD(VLIW[i].op[k])) { // op[k] is a LD instruction
      d=VLIW[i].op[k].dst_reg;
      prop_skp[j][d]=1;
    }
  }
}
```

図 26 命令写像アルゴリズムの諸定義と諸関数

(2–2) 当該 VLIW 命令内の全てのデスティネーションレジスタ番号（d）は伝搬不要となるため，prop_skp[d] に 1 をセットする.（update_prop_skp()）

(2–3) 当該 VLIW 命令のデスティネーションレジスタ番号と既に割り当てられている先行命令のソースレジスタ番号が一致した場合，先行命令に対して自

```
Instruction_mapping_algorithm() {
  for (i=0; i<N; i++) {   // for each VLIW instruction
    for (j=0; j<M; j++) { // for each stage
      if (j<i) {
        alloc_prop_reg(i, j);
        update_prop_skp(i, j);
        conf_self_fwd1(VLIW[i], exec[j]);
      } else if (j==i) {
        map_VLIW_inst(VLIW[i], exec[j]);
        alloc_prop_reg(i, j);
        update_prop_skp(i, j);
        conf_self_fwd1(VLIW[i], exec[j]);
        conf_self_fwd2(VLIW[i], exec[j]);
      } else if (j==i+1) {
        update_prop_skp(i, j);
      } else if (j==i+2) {
        update_prop_skp_onlyLD(i, j);
      }
    }
  }
}
```

図 27 命令写像アルゴリズム

己ループを設定する.（conf_self_fwd1().）

(3) i 番目の VLIW 命令は i 段目に以下の設定を行う.

(3–1) 当該 VLIW 命令を当該アレイ段の演算器に割り当てる.（map_VLIW_inst()）

(3–2) (2–1) と同様に伝搬レジスタの割り当てを行う.

(3–3) (2–2) と同様に伝搬不要表の更新を行う.

(3–4) (2–3) と同様に自己ループの設定を行う.

(3–5) 当該 VLIW 命令のソースレジスタ番号とデスティネーションレジスタ番号が一致している命令に対して，自己ループを設定する.（conf_self_fwd2()）本研究では，自己ループを適用される命令はループカウンタを更新する加算命令または減算命令に限定する．それ以外の命令はソースレジスタ番号とデスティネーションレジスタ番号が一致していても自己ループは設定され

ない．また，ループカウンタの更新を目的としない加算命令および減算命令では自己ループの設定を回避するために，コンパイラまたはプログラマがソースレジスタ番号とデスティネーションレジスタ番号が一致しないように配慮するものとする．

(4) i 番目の VLIW 命令は $i+1$ 段目に以下の設定を行う．

(4–1) 当該 VLIW 命令内のすべての命令のデスティネーションレジスタの値は整数演算器の演算器出力レジスタから $i+1$ 段目の VLIW 命令にフォワーディングできるため，$i+1$ 段目の伝搬不要表に 1 をセットする．（update_prop_skp()）

(5) i 番目の VLIW 命令は $i+2$ 段目に以下の設定を行う．

(5–1) 当該 VLIW 命令内のロード命令のデスティネーションレジスタの値はロード・ストアユニットの演算器出力レジスタから $i+2$ 段目の VLIW 命令にフォワーディングできるため，$i+2$ 段目の伝搬不要表に 1 をセットする．（update_prop_skp_onlyLD()）

(6) VLIW 命令内に後方無条件分岐命令が検出された場合，ループカーネルの終端とみなし命令写像を完了する．それ以外の命令はループカーネル内の命令とみなし，(2) から命令写像を繰り返す．

5.3.2 サンプルプログラムの命令写像例

本項では，5.3.1 項で述べた命令写像アルゴリズムを用いて，サンプルプログラムの命令写像を示す．図 28 に LAPP の演算器周辺のレジスタを示す．VLIW 命令列は初段の命令デコーダより供給される．s1，s2 および s3 は各演算器の演算器入力レジスタを表しており，伝搬レジスタと兼用される．wb は各演算器の演算器出力レジスタを表している．prop_skp は 5.3.1 項で述べた伝搬不要表を表している．

図 29 に LAPP の演算器周辺の接続を示す．i 段目のある演算器の演算器入力レジスタ（s1 および s2）は $i-1$ 段目にある 11 個の演算器入力レジスタと $i-2$ 段目にある 5 個の演算器出力レジスタから値を得ることができる．そのために，図 22

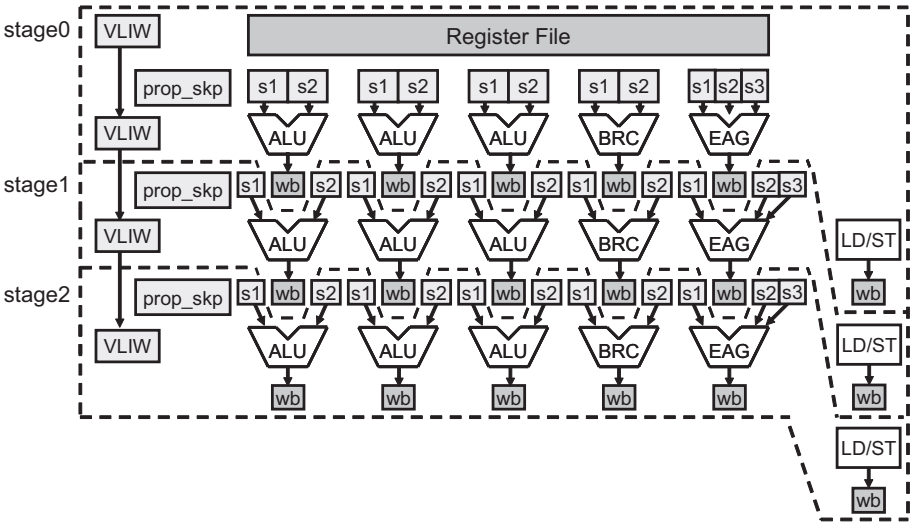


図 28 演算器アレイ周辺のレジスタ

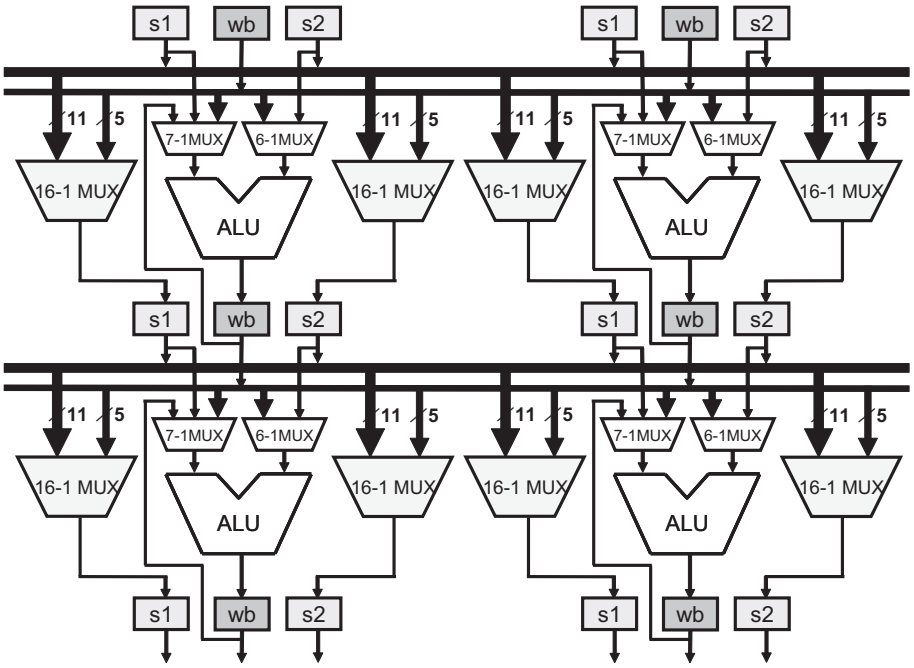


図 29 演算器アレイの内部接続

```
A[N], B[N], C[N];
j=0, k=0, l=0;
for (i=N-1: i>0; i--){
    C[j] = (A[k]<<16) | B[l];
    j++, k++, l++;
}
```

(a) Program code

```
 loop:
VLIW 0 add gr1, 4 ->gr1  sub gr3, 1 -> gr3, z  ld @(gr1, 0) ->gr2
VLIW 1 add gr4, 4 ->gr4  bz end                ld @(gr4, 0) ->gr5
VLIW 2 sll gr2, 16 ->gr2
VLIW 3 or gr2, gr5 ->gr5
VLIW 4 add gr6, 4 ->gr6  bra loop              st, gr5 ->@(gr6, 0)
 end:
```

(b) Assembly code

図 30 命令写像のサンプルプログラム

中の SEL は 11 個の 16 入力 1 出力のマルチプレクサ（16–1 MUX）を持ち，$i-1$ 段目の 11 個の演算器入力レジスタと $i-2$ 段目にある 5 個の演算器出力レジスタから値を選択し，i 段目の演算器入力レジスタに格納する．また，各演算器は自己更新命令に対応するために自身の演算器出力レジスタからフォワーディングした値と，$i-1$ 段の演算器出力レジスタからフォワーディングした値を選択するマルチプレクサ（6–1 MUX と 5–1 MUX）を備える．

図 30（a）および図 30（b）にサンプルプログラムの C コードおよびアセンブリコードをそれぞれ示す．このサンプルコードは A と B を入力配列として用い，A と B の各要素の値をロードし，A の要素を 16 ビットシフトした値と B の要素を論理和により結合した結果を出力配列 C にストアする．ループカーネルは VLIW4 の後方無条件分岐命令（bra）によりループの終端を表しており，VLIW0 から VLIW4 が演算器アレイに写像される．VLIW0 の減算命令（sub）はループカウンタを表し，ループ回数が初期値として gr3 に代入されており，イタレーション毎に減算し，ゼロフラグ（z）を更新する．アセンブリコードでは，VLIW0 と VLIW1 に

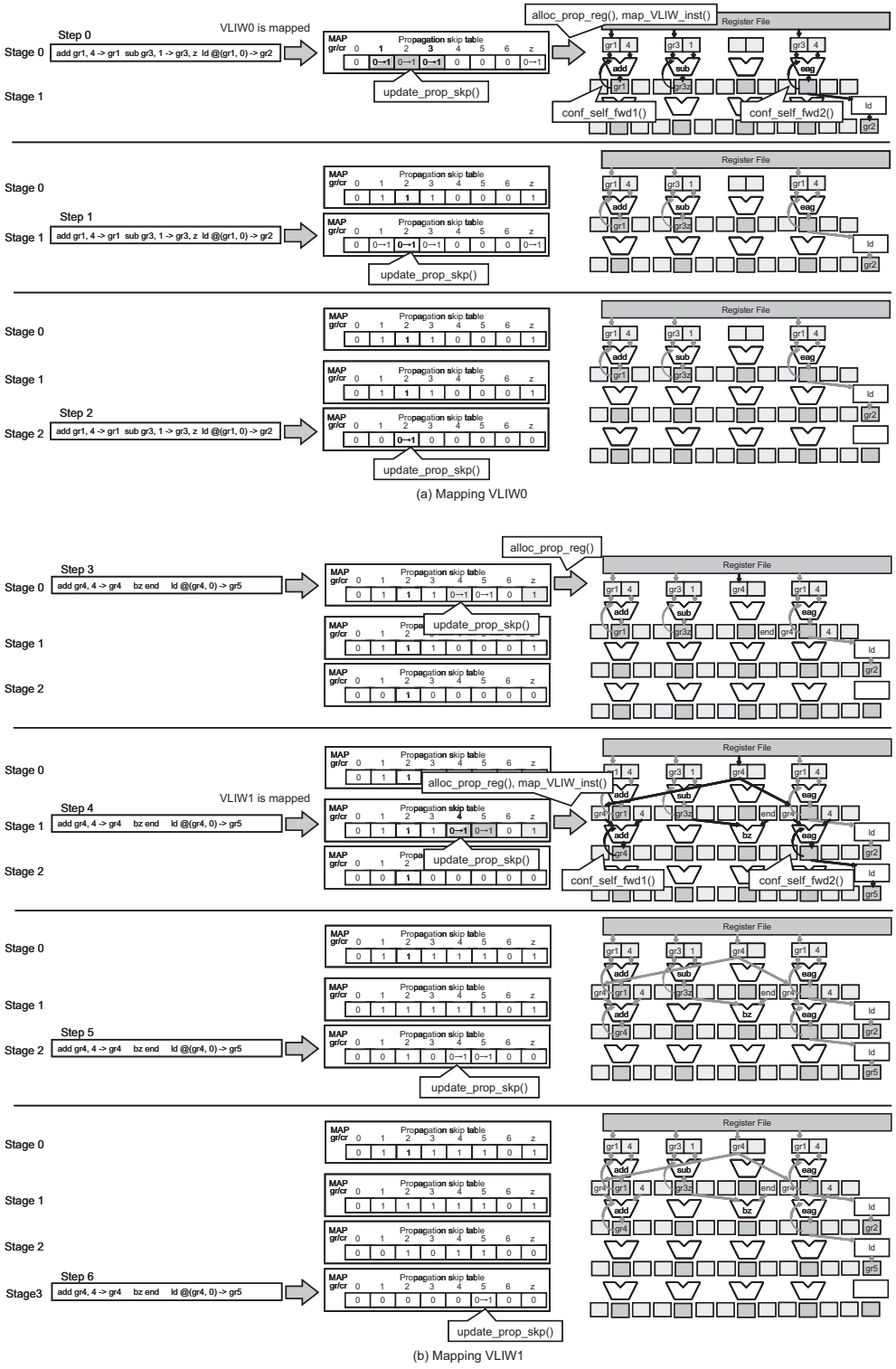
て gr1 と gr4 がそれぞれ A と B の各要素のアドレスを加算命令（add）にて計算し，ロード命令（ld）にて配列 A および配列 B の各値を gr2 と gr5 にロードする．ゼロフラグが成立すると，ループ回数分の実行が完了したことを表し，VLIW1 の条件付き前方分岐命令によりループを脱出する．VLIW2 と VLIW3 にて，16 ビット左シフト命令（sll）と論理和命令（or）により gr2 と gr5 の中間結果を gr5 に代入する．最後に VLIW4 にて，gr6 が C の各要素のアドレスを加算命令（add）にて計算し，ストア命令（st）にて中間結果を配列 C へストアする．

図 31 と図 32 にサンプルプログラムの命令写像の様子を示す．図 31（a）と図 31（b）はそれぞれ VLIW0 と VLIW1 に対する命令写像を 1 ステップずつ実行したものを表し，図 32（a），図 32（b）および図 32（c）はそれぞれ VLIW2，VLIW3 および VLIW4 に対する命令写像の結果を表す．

図 31（a）では，VLIW0 が Step0 にて初段の割り当てられる．このとき，prop_skp[gr1] と prop_skp[gr3] は (3–2) より 1 にセットされていないため，gr1 と gr3 は伝搬レジスタに割り当てられる．(3–3) より，初段の prop_skp[gr1]，prop_skp[gr2]，prop_skp[gr3] および prop_skp[z] に 1 がセットされる．(3–4) より，eag gr1+0 は自己ループが設定され，eag gr1+=4 となる．(3–5) より，add と sub の演算器にも自己ループが設定される．次に，(4–1) より，add と sub の結果は演算器出力レジスタから直接フォワーディングでき伝搬は不要であるため，ステージ 1 の prop_skp[gr1]，prop_skp[gr3] および prop_skp[z] は Step1 にて 1 にセットされる．(5–1) より，ld のロード値はステージ 2 の演算器からフォワーディングできるため，ステージ 1 とステージ 2 の prop_skp[gr2] は Step1 と Step2 にて 1 にセットされる．VLIW0 は (6) にて bra がないため，次の VLIW1 の命令写像を開始する．

同様に，図 31（b）では，VLIW1 の gr4 が (2–1) と (2–2) より，伝搬される必要があるため，Step3 にて初段の空き演算器の演算器入力レジスタに割り当てられる．VLIW1 は Step4 にて，ステージ 1 の演算機に割り当てられる．さらに，VLIW1 の add と ld には自己ループが設定される．続いて Step5 および Step6 にて，add と ld の結果をステージ 2 とステージ 3 にフォワーディングするために，prop_skp[gr4] と prop_skp[gr5] は 1 にセットされる．

さらに，図 32（a）では，prop_skp[gr2] が 1 となっているため，gr2 は Step7 お



(a) Mapping VLIW0

(b) Mapping VLIW1

図 31 VLIW0 と VLIW1 の命令写像

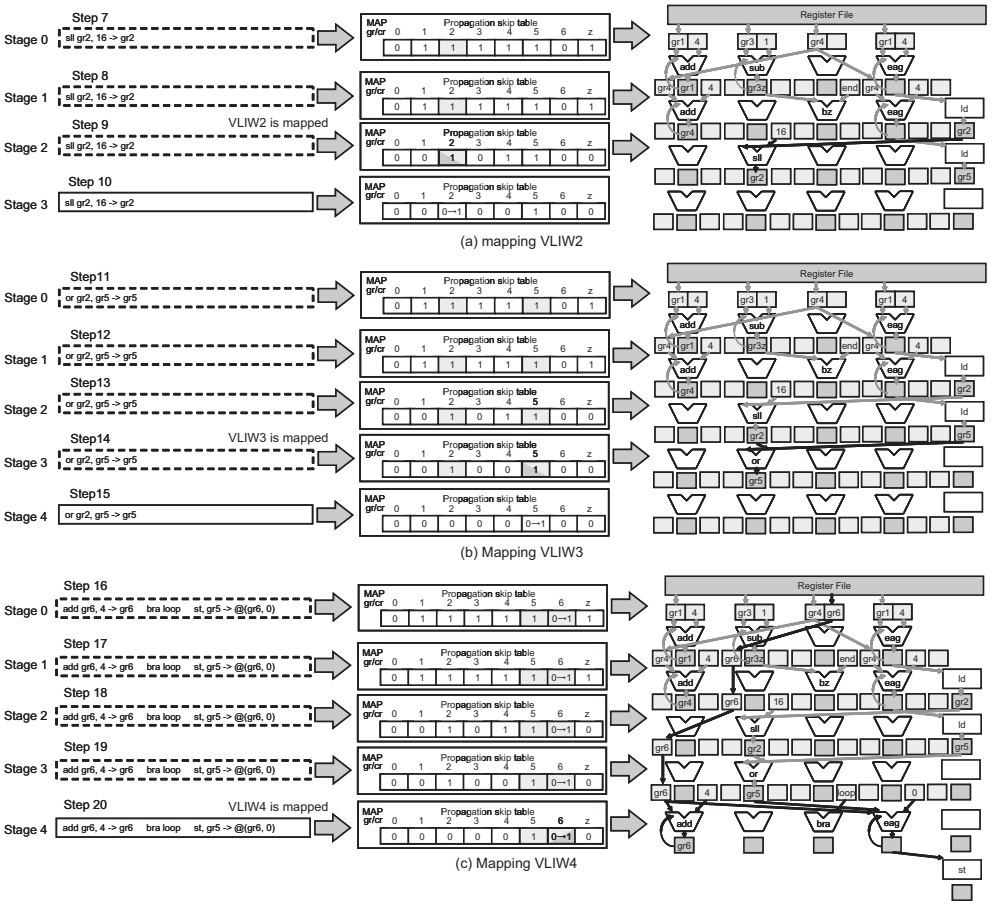


図 32 VLIW2，VLIW3 および VLIW4 の命令写像

よび Step8 にて伝搬不要であることがわかり，gr2 の値はステージ 1 の ld からフォワーディングされる．Step9 にてステージ 2 に割り当てられた VLIW2 は初段の ld の結果を使用することがわかる．ステージ 3 の演算器に sll の結果をフォワードするために，Step10 にて prop_skp[gr2] は 1 にセットされる．

次に，図 32（b）では，prop_skp[gr2] と prop_skp[gr5] が 1 であることから，gr2 と gr5 は Step11 から Step13 まで伝搬不要であることがわかる．gr2 と gr5 の値はステージ 2 の演算器出力レジスタからフォワーディングされる．Step14 にて，ステージ 3 に割り当てられる VLIW3 はステージ 2 とステージ 1 に割り当てられた sll と ld の値を使用する．Step15 にて，ステージ 3 の演算器に or の結果をフォワーディングするために，prop_skp[gr5] は 1 にセットされる．

最後に，図 32（c）では，gr6 が伝搬される必要があり，初段からステージ 3 までの空き演算器の演算器入力レジスタに Step16 から Step19 で割り当てられる．(2–3) より，ステージ 3 の add と st に対する演算器には Step20 にて自己ループが設定される．さらに，VLIW4 に bra が検出されるため，このループカーネルの命令写像は完了する．

以上の命令写像アルゴリズムはソフトウェアおよびハードウェアの両方に実装できる．例えば，ソフトウェア実装では，コンパイラが演算器およびネットワークの設定情報を本アルゴリズムにより生成できる．本研究では LAPP にハードウェア実装することを想定しており，各段に備えられた命令写像機構がパイプライン動作により VLIW 命令から設定情報を生成できる．各段に写像される VLIW 命令は各段に設けられた設定情報レジスタ（VLIW configuration register）に格納され，他段に写像される VLIW 命令は設定伝搬レジスタ（VLIW propagation register）により上段から下段へ伝搬される．図 31 および図 32 の各ステップはハードウェア実装において，各クロックサイクル（CC: Clock Cycle）にて以下のように同時実行される．

CC 0: step 0 (VLIW0 is mapped)

CC 1: step 1, step 3

CC 2: step 2, step 4, step 7 (VLIW1 is mapped)

CC 3: step 5, step 8, step 11

CC 4: step 6, step 9, step 12, step 16 (VLIW2 is mapped)

CC 5: step 10, step 13, step 17

CC 6: step 14, step 18 (VLIW3 is mapped)

CC 7: step 15, step 19

CC 8: step 20 (VLIW4 is mapped)

以上より，i 番目の VLIW 命令は CC($2 \times i$) にて，ステージ i に割り当てられることがわかる．よって，各段の設定に必要となるサイクル数は $2 \times N$（N: VLIW 命令数）となる．図 33 にハードウェアによる命令写像されたときの CC3，CC4 および CC5 における状態を示す．VLIW0，VLIW1，VLIW2，VLIW3 および VLIW4 が上段から下段へ伝搬されながら，各段に割り当てられることがわかる．

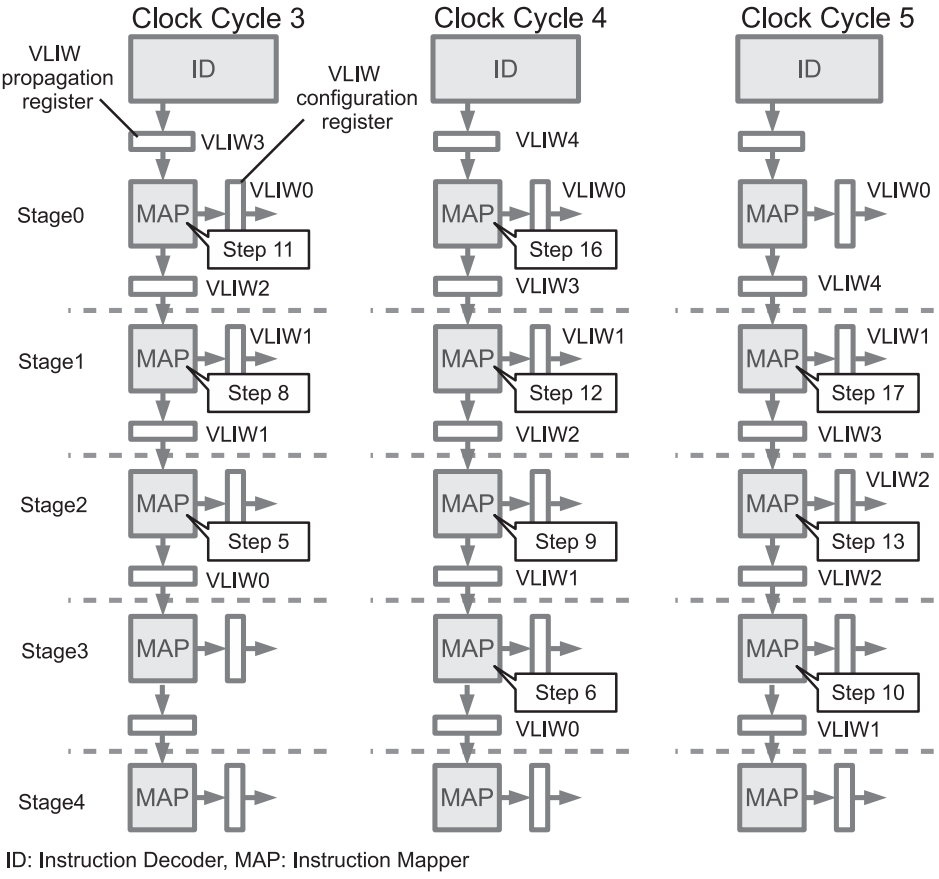


図 33 ハードウェアによる命令写像例

5.4 予備評価

本節では，5.2.6 節で述べた演算器ネットワークおよび 5.3.1 項で述べた命令写像の回路規模および遅延時間を評価する．まず，ハードウェア記述言語を用いて LAPP の各モジュールを設計し，Synopsys 社の Design Compiler と 180nm プロセスのライブラリを用いて論理合成を行う．次に，5.2.6 項で述べた RF32 モデル，RF11 モデルおよび SEL11 モデルの評価を行い，最後に各評価モデルと 5.3.1 項の命令写像を実現する MAP を含め，アレイ 1 段当たりの回路規模を比較する．なお，演算器およびレジスタファイルの構成は表 5 と同じである．

5.4.1 各モジュールの最小遅延時間およびその回路規模

各モジュールの最小遅延時間を求めるために，最も厳しいタイミング制約として 500MHz を与えて論理合成した結果を図 34 と図 35 に示す．図 34 と図 35 はそれぞれ各モジュールの遅延時間および回路規模を表す．ALU と MEDIA は，それぞれ 3 個と 4 個が LAPP に備えられており，それらの合計の回路規模を表す．

図 34 より，SEL11 モデル，RF11 モデルおよび RF32 モデルの遅延時間は 5.6, 6.0 および 7.0 であることがわかる．一方，MAP の遅延時間は 7.0 となり，ALU と MEDIA に続いて 3 番目に遅延時間が大きいことがわかる．ここで，MAP はアレイ設定モードのみで動作し，通常実行モードおよびアレイ実行モードでは動作しないことに注目する．MAP の動作を 1 サイクル動作から 2 サイクル動作に分割することができれば，アレイ設定のサイクル数が 2 倍になるもののデータプリフェッチ時間が十分長ければ隠蔽できる．

図 35 より，RF32 の回路規模は汎用レジスタファイルとメディアレジスタファイルを含み，レジスタ数とマルチプレクサ数が多いため LAPP のモジュールの中で最も大きいことがわかる．一方，物理レジスタと演算器入力レジスタを兼用した伝搬レジスタを持つ SEL11 は評価モデルの中で回路規模と遅延時間の両方で最も小さいことがわかる．また，MEDIA の遅延時間が最も長い 6ns であることから，LAPP は 166MHz で動作することがわかる．

5.4.2 各モジュールの妥当遅延時間およびその回路規模

タイミング制約として 166MHz を与えて論理合成した結果を図 36 と図 37 に示す．図 36 と図 37 はそれぞれ各モジュールの遅延時間および回路規模を表す．

図 36 より，すべてのモジュールがタイミング制約を満たしていることがわかる．プロセッサのパイプラインステージにおける最適な FO4 値には様々な議論がなされている．文献 [68] では最適な FO4 値として 6 から 8 を挙げており，文献 [69] は電力と性能の指標として BIPS/W（(Billions of Instructions Per Second)^3/Watt）を用いて，評価した結果，最適な FO4 値としてパイプラインステージ当たり 18 を挙げている．本研究では，FO4 を遅延時間の大小比較にのみ用い，LAPP にとって最適な FO4 値については議論しないものとする．

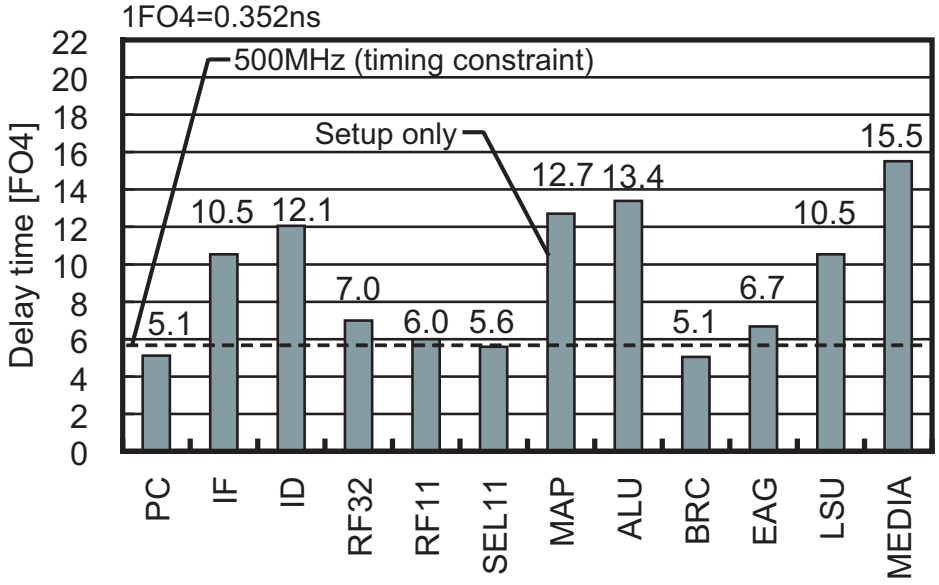


図 34 厳しいタイミング制約における各モジュールの遅延時間

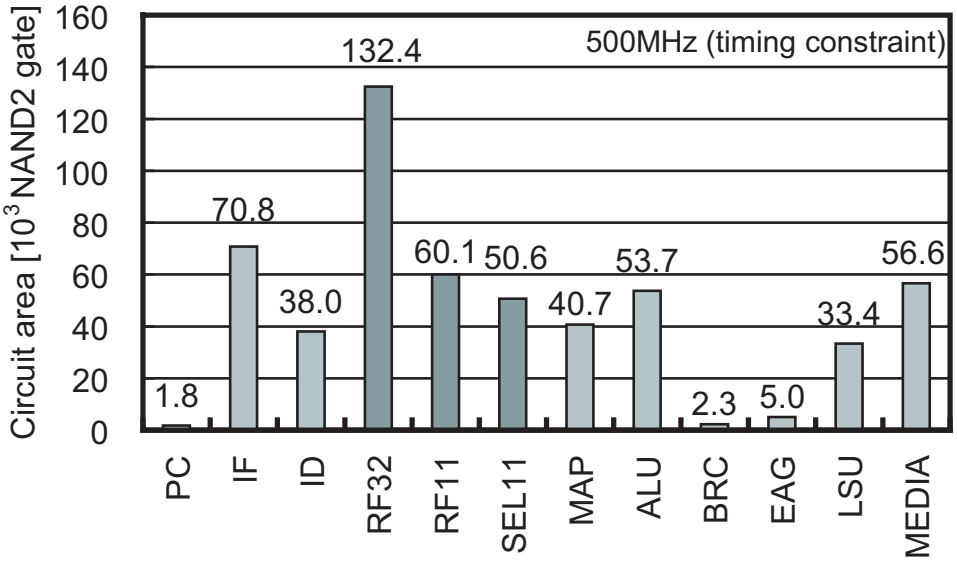


図 35 厳しいタイミング制約における各モジュールの回路規模

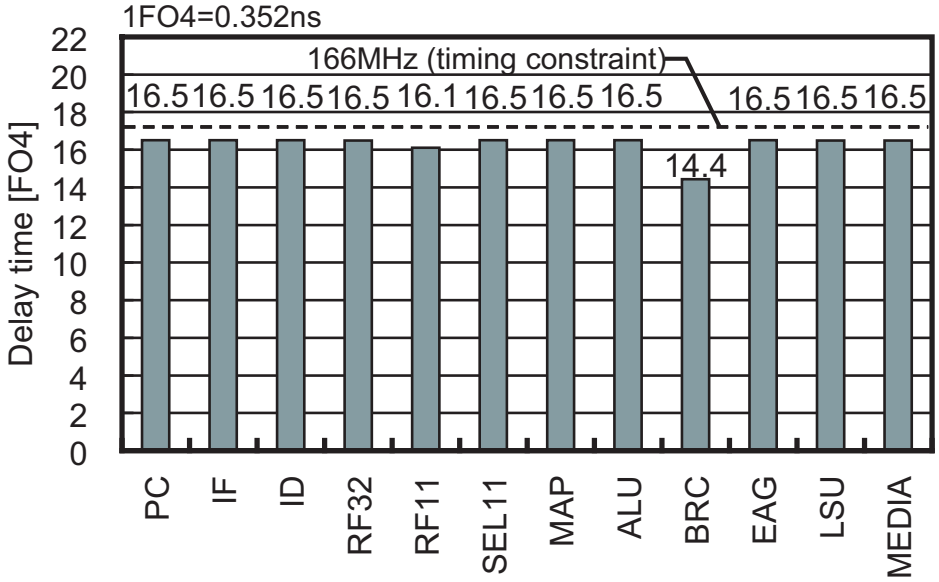


図 36 妥当なタイミング制約における各モジュールの遅延時間

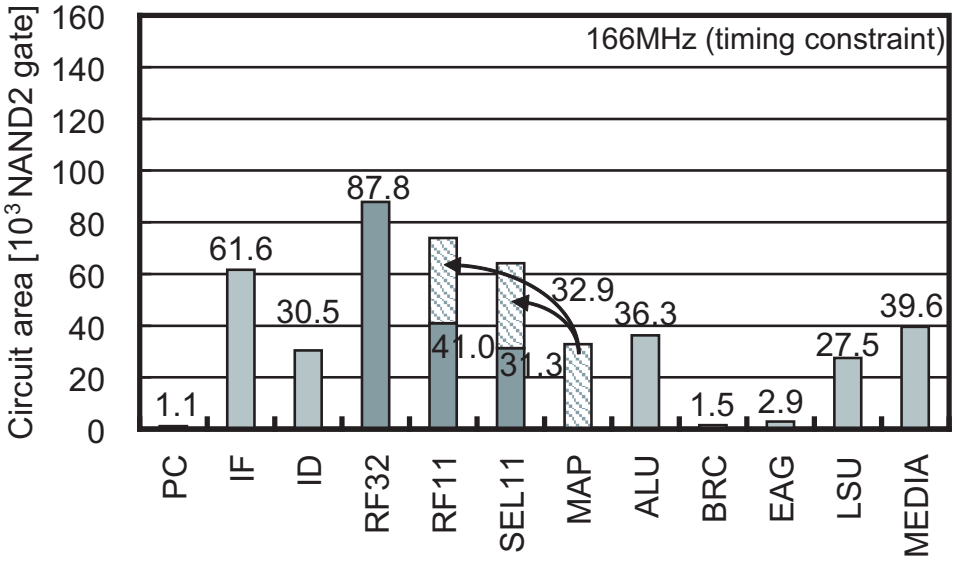


図 37 妥当なタイミング制約における各モジュールの回路規模

図 37 より，妥当なタイミング制約における RF32，SEL11，RF11 および MAP の回路規模は 87.8K，31.3K，41.0K および 32.9K ゲートであることがわかる．SEL11 モデルは SEL11 と MAP の和で求めることができ，RF32 モデル（RF32 のみ）の 84%であることがわかる．各モデルを持つアレイ段の回路規模を求めると，RF32 モデルでは 196K ゲート，SEL11 モデルでは 172K ゲートとなる．よって，SEL11 モデルは RF32 モデルの 88%の回路規模でアレイ段を構成することができ，RF32 よりも配線数および回路規模を削減することができた．

5.5 全体評価

本節では，5.4 節の予備評価をふまえ，SEL11 モデルを持つ LAPP の回路規模および消費電力を評価し，メニコアプロセッサと比較する．

5.5.1 評価モデルと LSI 試作結果

本研究で想定している LAPP の構成は，画像処理ベンチマークを実行するために整数演算器およびメディア演算器を含む 36 段構成である．しかし，本構成は LSI 試作する 180nm プロセスの 5.0×7.5mm 角には大きすぎるため，すべてを実装することができない．そこで，画像処理を代表とする実用的なプログラムではできる限りアレイ段数は多いほうがよいため，アレイ段数をできる限り多く実装する方針に従い，LSI 試作向けに LAPP の構成を検討した．表 6 に本研究が想定している LAPP の基本モデルと LSI 試作モデルの評価パラメータを示す．

まず，性能を犠牲にして回路規模を削減する方針より，初段の単体性能を向上させる分岐予測器およびリターンアドレススタックを取り除くことができる．同様に，性能向上に貢献する I1 キャッシュのウェイ数およびサイズを削減し，回路規模の削減を図った．また，アレイ段数が数段しか実装できない場合，メディア演算器の演算能力を有効活用できないため，メディア演算器，メディアレジスタおよびメディア用伝搬レジスタを実装しないことにした．これにより 1 サイクルあたりの最大実行命令数が 8 から 4 に減少したため，命令デコーダからメディア用デコーダを切り離し，命令デコーダ幅を 8 から 4 に削減した．同様に，実装できる段数に対してアレイ実行で使用するデータストリームおよびデータサイズが

表 6 LAPP の評価パラメータ

	Base model	LSI model
Branch predictor	gshare	-
	PHT: 2 bit×8K entry	
Return address stuck	8 entry	-
Decode width	8 instruction/cycle	4 instruction/cycle
General register file (GR)	32 entry, 11R5W port	32 entry, 11R5W port
Media register file (MR)	32 entry, 7R4W port	-
Functional unit	3 ALU, 1 BRC, 1 EAG	3 ALU, 1 BRC
	1 LSU, 4 MEDIA	1 EAG, 1 LSU
Data transfer speed	8 byte/cycle	8 byte/cycle
Instruction level 1 cache (I1$)	4 way, 16KB	1 way, 4KB
Data level 1 cache (L1$)	4 way, 16KB	1 way, 4KB
I1$ and L1$ cache line size	64 byte	64 byte
L1$ → L0$ Data transfer rate	16 byte/cycle	4 byte/cycle
Store buffer of VLIW stage	4 entry	4 entry
Instruction mapping speed	2 cycle/stage	2 cycle/stage
Propagation register for GR	11	11
Propagation register for MR	9	-
Data level 0 cache (L0$)	4 way, 256B	1 way, 128B
L0$ data propagation rate	16 byte/cycle	4 byte/cycle
L1$ → L0$ data transfer rate	16 byte/cycle	4 byte/cycle
L0$ → LSU data transfer rate	4 byte/cycle	4 byte/cycle
Store buffer of array stage	1 entry	1 entry

表 7 LAPP の LSI 試作結果

Stage	Circuit area [Kagtes]	Area ratio
VLIW stage (logic)	212	0.11
VLIW stage (memory)	757	0.39
Array stage 0	76	0.040
Array stage 1	77	0.040
Array stage 2	81	0.042
Array stage 3	82	0.042
Array stage 4	82	0.042
Array stage 5	82	0.042
Array stage 6	82	0.042
Array stage 7	82	0.042
Array stage 8	82	0.042
Array stage 9	82	0.042
Array stage 10	83	0.043
Array stage 11	67	0.035
Total	1170	1.00

大きすぎるため，L1 キャッシュのウェイおよびサイズはそれぞれ 1 ウェイおよび 4KB に削減した．これにより，L1 キャッシュと L0 キャッシュの間の性能比を調整するために，L0 キャッシュも 1 ウェイに削減し，1 ウェイあたりのサイズを 256 バイトから 128 バイトに削減し，転送速度を 1/4 にした．

3 章で述べた段階的開発手法を用いて 1 年当たり教員と学生を含め 6 名が開発に従事した結果，4 年の研究期間で LAPP の LSI 試作に成功し，テストプログラムが正しく動作したことを確認した．最大動作周波数は 66MHz である．LAPP の試作結果を表 7 に示し，LAPP のフロアプランを図 38 に示す．表 7 より，LSI 試作モデルは 13 段構成を実装できた．また，初段に含まれる I1 キャッシュと L1 キャッシュのメモリマクロが全体の 39%を占めることがわかり，図 38 からも L1 キャッシュが大部分を占めていることがわかる．これは，L1 キャッシュの幅が 512

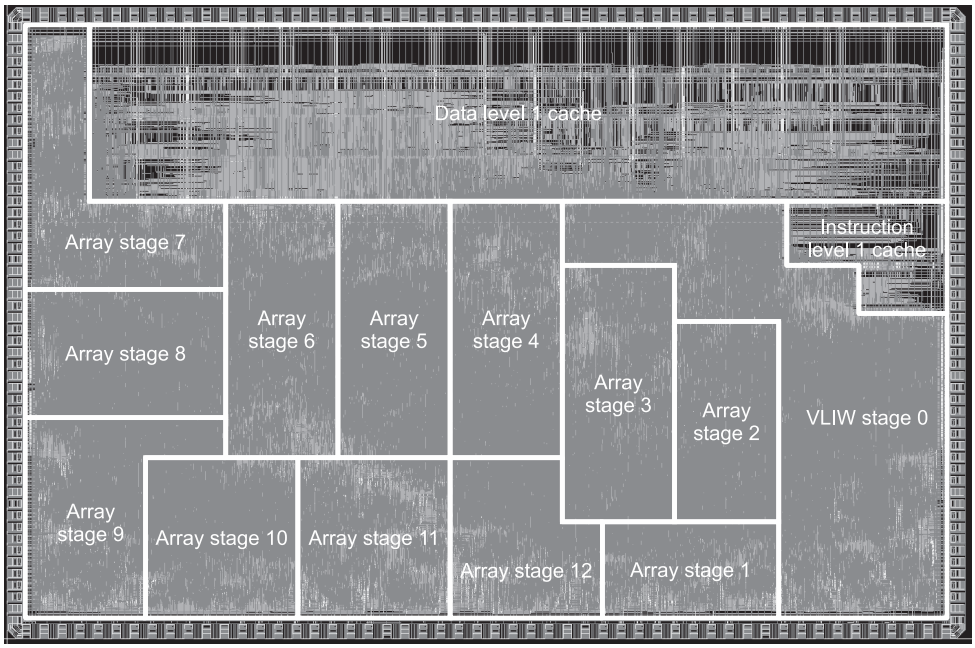


図 38 LAPP のフロアプラン

ビット，深さが 64 エントリの構成であり，さらにバイト書き込みを行うために，8 ビット幅 64 エントリのメモリマクロを 64 個並置したため，面積オーバヘッドが増加したからである．メモリマクロは一般的に幅を小さくし，深さを大きくすることで面積オーバヘッドは減少する．本来は，L1 キャッシュから読み出す値の最大幅は 32 ビットであり，メモリインタフェースの幅が 64 ビットであることから，L1 キャッシュの幅は 64 ビットで十分である．そこで，5.5.2 項以降の基本モデルを使用した評価では 8 ビット幅 512 エントリのメモリマクロを 8 個並置した構成を用いる．

5.5.2 回路規模および消費電力

本項では，LAPP の比較対象となるメニコアプロセッサについて述べ，試作結果を示す．

設計した LAPP の各モジュールを，Synopsys 社の Design Compiler と 180nm プ

ロセスのライブラリを用い，タイミング制約として 100MHz を与えて論理合成を行った．ベンチマークには画像処理プログラムとして，3 段階のフレーム補間（FI-1，FI-2 および FI-3），画像拡大（Z），先鋭化（S），メディアンフィルタ（M），輪郭抽出（E），輪郭ノイズ除去（N）および奥行抽出（SM）を用いた．FI-1 は SAD（Sum of Absolute Difference）演算を用いて画像の差分を計算し，FI-2 は FI-1 で求めた SAD 値から最小値を見つけ，FI-3 は FI-2 の結果を用いて補間画像を生成する．画像のサイズは 320×240 であり，各画素は RGB 各 1 バイトを含む 4 バイトである．各ベンチマークログラムでアレイ実行されるループカーネルには最大 3×3 画素が入力され，演算結果として 1 画素が出力される．ベンチマークプログラムに必要なアレイ段数の最大は 33 段であるため，LAPP の段数を 36 段とした．

論理合成したときの各モジュールの回路規模と消費電力を表 8 に示す．表 8 の消費電力は，Cadence 社の NC-Verilog を用いた RTL シミュレーションにより各モジュールでベンチマークプログラムをキャッシュミスやストールが発生せず毎サイクル動作する理想的な実行の波形を生成し，Synopsys 社の Prime Time を用いて，その波形データとゲートレベルネットリストから見積もった値である．また，キャッシュメモリの回路規模および消費電力は CACTI3.2[70] によって見積もった．

表 8 より従来 VLIW プロセッサ（Normal core）と LAPP の初段（VLIW stage）およびアレイ段（Array stage）を求めた．LAPP の基本モデルと従来 VLIW プロセッサから構成されるメニコアプロセッサの回路規模を表 9 に示す．メニコアプロセッサの回路規模にはコア間の接続などに必要な追加回路は含まれないものとした．表 9 より LAPP の基本モデルは 9.5 コアを搭載するメニコアプロセッサと同回路規模であることがわかる．

5.5.3 性能評価

ソフトウェアシミュレータを用いて LAPP とメニコアプロセッサの性能評価を行った．評価パラメータは表 6 の基本モデルである．

表 10 に LAPP とメニコアプロセッサの性能として IPC を示す．ここで，性能はループカーネルの高速実行を含むベンチマーク全体の実行性能を表す．メニコアプロセッサの性能は 5.5.2 項で得たコア数と従来 VLIW プロセッサの IPC の積で

表 8 LAPP 各モジュールの回路規模と消費電力

Unit	Circuit area [gates]	Power [mW]	Number of Units		
			Normal core	VLIW stage	Array stage
PC	1,075	1.58	1	1	0
IF	51,150	53.30	1	1	0
ID	25,154	22.20	1	1	0
RF	87,278	37.00	1	1	0
I1$	176,837	93.40	1	1	0
L1$	258,419	143.28	1	1	0
EAG	3,313	2.21	1	1	1
ALU	11,109	5.82	3	3	3
MEDIA	7,844	4.41	4	4	4
BRC	1,801	0.77	1	1	1
MAP	24,773	26.30	0	1	1
SEL	34,146	25.56	0	1	1
L0$	24,124	4.04	0	1	1
LSU	9,557	1.18	1	1	1
Normal core	679,287	388.84			
VLIW stage	762,330	-			
Array stage	162,417	-			

表 9 LAPP とメニコアプロセッサの回路規模

		Circuit area [gates]
Many-core	Normal core × 9.5 cores	6,453,227
LAPP (base)	1 VLIW stage + 35 array stages	6,446,925

表 10 LAPP とメニコアプロセッサの性能

Benchmark	No. of VLIW in a loop kernel	LAPP Avg. IPC	Many-core Avg. IPC	Performance ratio
FI-1	20	14.2	13.8	1.03
FI-2	33	28.1	11.9	2.35
FI-3	10	9.0	12.6	0.72
Z	29	35.4	17.7	2.01
S	27	27.3	16.1	1.70
M	23	22.3	19.3	1.16
E	14	11.9	11.3	1.05
N	26	24.0	19.4	1.24
SM	31	25.4	11.8	2.16
Average	24	22.0	14.9	1.49

求め，コア間通信およびメモリアクセスによる性能低下は無いものとして評価している．表 10 より，FI-3 以外のプログラムでは LAPP の性能がメニコアプロセッサよりも高いことがわかる．これは，FI-3 はループカーネルの VLIW 命令数が他のプログラムと比べて少なく，アレイ実行の効果を十分得られていないからである．平均 IPC では，LAPP はメニコアプロセッサの 1.49 倍であることがわかる．

5.5.4 電力評価

5.5.2 項から得られた各モジュールの消費電力と RTL シミュレータから得られた動作サイクル数から求めた各モジュールの動作時間を積算することで LAPP の消費電力量を求め，全実行時間で除算し，平均消費電力を求めた．また，64 ビット比較器の物理設計を行った結果より，使用したライブラリにおけるリーク電流により消費する電力を電源供給時の 0.2%と仮定し，電源 OFF から電源 ON への切り替え時に電源供給時の 8%を消費すると仮定する．また，文献 [71] より DVS により内容を保持するために 33%の電力が消費されると仮定する．

表 11 メニコアに対する LAPP の消費電力比率

Benchmark	Frontend	I$	D$	Reg.	Exe.
FI-1	3.11	3.98	10.2	16.9	25.2
FI-2	7.14	4.09	12.6	43.3	51.1
FI-3	2.75	4.23	10.2	15.7	16.9
Z	4.07	4.68	10.3	58.2	62.4
S	5.02	3.98	10.5	44.8	50.5
M	4.02	4.09	10.5	30.2	34.2
E	3.30	3.86	10.9	18.2	17.5
N	14.4	5.03	11.6	33.4	44.6
SM	6.07	4.09	12.9	37.8	34.3
Average	5.55	4.23	11.1	33.2	37.4

[%]

表 11 にメニコアプロセッサに対する LAPP の消費電力の割合を示す．Frontend は PC，IF および ID の合計を表し，I$は命令 1 次キャッシュを表し，D$は L1 キャッシュと L0 キャッシュの合計を表す．Reg. はレジスタファイルおよび LAPP の伝搬レジスタを表し，Exe. は演算器を表す．ここで，消費電力はベンチマーク全体を実行したときの各モジュールの平均消費電力を表す．表 11 より，LAPP の各モジュールの消費電力は対応するメニコアプロセッサのモジュールよりも小さいことがわかる．

LAPP のフロントエンドと I$の消費電力はそれぞれメニコアプロセッサの 5.55%と 4.23%となっている．これは，メニコアプロセッサでは常に動作しているモジュールであるのに対して，LAPP はアレイ実行中にこれらのモジュールを長期間計画的に停止できるからである．次に，D$は LAPP とメニコアプロセッサのどちらにおいても常に動作するものの，LAPP はメニコアプロセッサの 11.1%となっている．これは，メニコアプロセッサでは各コアが D$を持つのに対して，LAPP では初段のみ D$を持つからである．さらに，L0$はアレイ実行モード以外では停止していることも低消費電力化に効いているからである．最後に，Reg. と Exe. では LAPP とメニコアプロセッサで大きく異なる．LAPP の演算器数はメニコアプロ

表 12 LAPP とメニコアプロセッサの消費電力

Benchmark	No. of VLIW in a loop kernel	LAPP Power [mW]	Many-core Power [mW]	Power Ratio
FI-1	20	378	4028	0.09
FI-2	33	677	4038	0.17
FI-3	10	339	3914	0.09
Z	29	743	4038	0.18
S	27	636	4038	0.16
M	23	496	4047	0.12
E	14	376	4019	0.09
N	26	660	4047	0.16
SM	31	591	4019	0.15
Average	24	544	4021	0.14

セッサの 4 倍以上あり，D$の回路規模を多くの演算器に費している．また，メニコアプロセッサのレジスタファイルには LAPP のレジスタファイルおよび伝搬レジスタが対応している．結果，LAPP の Reg. の消費電力はメニコアプロセッサの 33.2%になり，Exe. の消費電力は 37.4%となる．これは，メニコアプロセッサでは各コアの演算器は実行する命令を予想できないため，常に電源を供給されなければならない．一方，LAPP の初段の演算器はメニコアプロセッサと同じものの，アレイ段の演算器は通常実行モードではすべて停止することができ，アレイ実行モードでも命令が割り当てられていない演算器を長期間停止できるからである．

表 12 に LAPP とメニコアプロセッサで各ベンチマークプログラムを実行したときの消費電力を示す．ここで，消費電力はプロセッサ全体の平均消費電力を表す．メニコアプロセッサの消費電力は従来 VLIW プロセッサでベンチマークプログラムを実行したときの消費電力とコア数の積で表し，コア間データ転送およびメモリアクセスに必要な電力は含まれないものとした．低消費電力技術の適用によって，LAPP の平均消費電力はメニコアプロセッサの 14%であることがわかる．

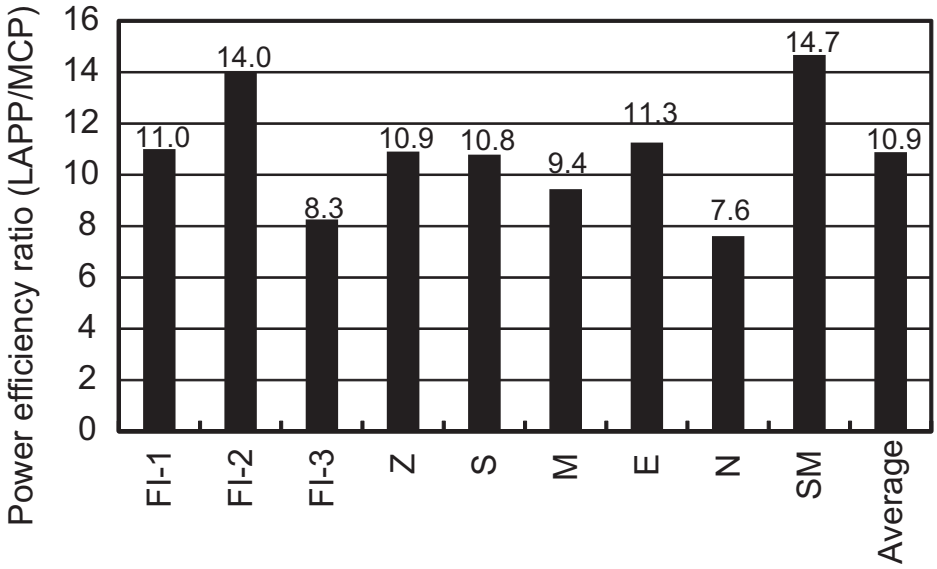


図 39 メニコアプロセッサに対する LAPP の電力効率

5.5.5 電力効率

本節では，5.5.2 項，5.5.3 項および 5.5.4 項の評価結果を踏まえて，総合評価として LAPP とメニコアプロセッサの電力効率を比較する．LAPP とメニコアプロセッサの電力効率を図 39 に示す．縦軸はメニコアプロセッサに対する LAPP の電力効率を表し，横軸はベンチマークプログラムを表す．LAPP の電力効率はメニコアプロセッサの 7.6 倍から 14.7 倍を実現していることがわかり，平均 10.9 倍の電力効率を実現している．以上より，LAPP はメニコアプロセッサよりも高い電力効率を実現できたといえる．

5.6 結言

本章では，メニコアアーキテクチャおよび粗粒度再構成アーキテクチャの長所を持ち，演算器アレイにより画像処理を高速実行する線形アレイ型パイプラインプロセッサ LAPP を提案した．そして，LAPP が多数の演算器に効率良く命令を割り当てるための命令写像手法を提案し，そのアルゴリズムを示した．本命令写

像および値伝搬の評価モデルを示し，ハードウェア設計による予備評価の結果，比較モデルの 84%の回路規模になること，および，比較モデルを持つアレイ段の 88%の回路規模を実現することを示した．さらに，LAPP のソフトウェアシミュレータによる評価から LSI 試作までを行い，6 人/年・研究期間 4 年で LSI 試作に成功した．LAPP の試作結果より得た消費電力パラメータを用いてソフトウェアシミュレータにより評価した結果，36 段構成 LAPP が 9.5 コアを搭載するメニコアプロセッサと同回路規模となることを示し，メニコアプロセッサの 10.9 倍の電力効率を実現することを示した．

6. 結論

本論文では，異種命令混在実行プロセッサ OROCHI と線形アレイ型パイプラインプロセッサ LAPP について述べた．

1 章では，研究背景として組込み機器用途および汎用計算機用途のプロセッサの現状と問題を示し，研究概要と研究成果を示した．

2 章では，これまでのプロセッサ技術を含め，高まるプロセッサへの高性能と低消費電力の要求に対して，提案されている研究について述べ，新しいプロセッサの必要性を示した．

3 章では，近年大規模化するプロセッサの周辺回路の設計を省略し，大学研究室規模でもプロセッサの開発に注力できるプロセッサ開発手法を示した．このプロセッサ開発手法は，ソフトウェアシミュレータによる評価，FPGA による設計・実機検証を経て，LSI 試作・評価を段階的に行うことができ，開発期間や費用に応じて柔軟に対応できる特徴を有することを示した．

4 章では，VLIW 型命令キューを用いて VLIW 方式の余剰演算能力を利用し，他スレッドをスーパスカラ実行することでヘテロジニアスマルチスレッド実行する異種命令混在実行プロセッサ OROCHI を提案した．そして，OROCHI において従来手法である OS スケジューラによるソフトウェア制御と提案手法である VLIW 型命令キューによるハードウェア制御の 2 つのスレッド制御についてソフトウェアシミュレータを用いた予備評価を行った．その結果，従来手法により優先スレッドである FR-V の性能を 95%まで向上できることを示し，スレッド制御を適用しないベースモデルと比較して提案手法により 2%の性能を向上できたことを示した．さらに，OROCHI の開発ではソフトウェアシミュレータによる論理検証，ハードウェア設計・検証および LSI 試作後の評価を段階的に行う開発手法を用いて 8 人/年・研究期間 3 年の開発で LSI 動作に成功した．試作した OROCHI の回路規模，動作周波数，性能および消費電力より電力効率による総合評価を行った結果，OROCHI はマルチコアプロセッサと比較して 79%の回路規模で，汎用プログラムとマルチメディアプログラムを同時実行した場合 1.31 倍の電力効率を実現した．

5 章では，メニコアアーキテクチャおよび粗粒度再構成アーキテクチャの長所を持ち，演算器アレイにより画像処理の高速実行を実現する線形アレイ型パイプ

ラインプロセッサ LAPP を提案した．そして，LAPP が多数の演算器に効率良く命令を割り当てるための命令写像手法とそのアルゴリズムを提案した．次に，本命令写像および値伝搬の評価モデルを示し，ハードウェア設計による予備評価の結果，比較モデルの 84%の回路規模になること，および，比較モデルを持つアレイ段の 88%の回路規模を実現することを示した．さらに，LAPP のソフトウェアシミュレータによる評価から LSI 試作までを行い，6 人/年・研究期間 4 年で LSI 試作に成功した．LAPP の試作結果より得た消費電力パラメータを用いてソフトウェアシミュレータにより評価した結果，36 段構成 LAPP が 9.5 コアを搭載するメニコアプロセッサと同回路規模となることを示し，メニコアプロセッサの 10.9 倍の電力効率を実現することを示した．

以上により OROCHI と LAPP はそれぞれ組込み機器と汎用計算機の用途で高性能と低消費電力が求められるプロセッサにおいて候補となりうる優位性を示せたといえる．

謝辞

本論文は奈良先端科学技術大学院大学情報科学研究科博士前期課程および博士後期課程の研究成果をまとめたものであり，多くの方の御指導と御協力により実現できたものである．

本研究において，他研究分野から本研究分野にやってきた著者を迎え，ソフトウェアからハードウェアまで広い研究分野の知識を得る機会だけでなく，素晴らしい研究環境に加えLSI試作という貴重な機会を与えて頂き，多大なる御指導を頂いた本学コンピューティング・アーキテクチャ講座の中島康彦教授に深く感謝の意を表します．

本研究を行うにあたり，有益な御助言を頂き，多大なる御指導を頂きました本学ディペンダブルシステム学講座の井上美智子教授，本学コンピューティング・アーキテクチャ講座の嶋田創准教授，本学ネットワークシステム学講座の岡田実教授および大阪学院大学の藤原秀雄教授に深く感謝の意を表します．

学生生活および研究活動において，平素から御助言と御指導を頂きました本学コンピューティング・アーキテクチャ講座の中田尚助教，姚駿助教および齊藤光俊特任助教に深く感謝の意を表します．特に，中田尚助教には修士課程在籍時から研究，論文執筆および発表において多大なる御助言と御指導を頂きました．姚駿助教には英語論文の執筆および国際会議の発表において有益な御助言と御指導を頂きました．齊藤光俊特任助教にはLAPPのLSI試作のためのGL回路設計およびGLシミュレーションの検証において多大なる御協力を頂きました．

OROCHIの研究において，発表の機会を与えて頂き御助言と御指導を頂きました広島市立大学の北村俊明教授に深く感謝の意を表します．また，有益な御討論と御協力を頂きました半導体理工学研究センターの須賀敦浩殿，宮本幸昌殿，宮本俊介殿および大西洋一殿，東京工業大学の吉瀬謙二准教授に深く感謝の意を表します．そして，有益な御討論と御協力を頂きました広島市立大学コンピュータシステム研究室OROCHIチームOBの塩田耕太朗氏，元安優氏および山下純一氏，本学コンピューティング・アーキテクチャ講座OROCHIチームOBの小島知也氏，片岡晶人氏，須賀圭一氏，牟田口公洋氏，山原幹雄氏および市來亮人氏に感謝致します．OROCHIのLSI試作における成功は皆様の御協力により実現でき

たものである．

LAPP の研究において，有益な御討論と御協力を頂きました株式会社富士通研究所の井上淳樹殿および中山寛殿に深く感謝の意を表します．また，有益な御討論と御協力を頂きました本学コンピューティング・アーキテクチャ講座 LAPP チーム OB の上利宗久氏，岩上拓矢氏および森浩大氏，本学コンピューティング・アーキテクチャ講座 LAPP チームの下岡俊介氏，大上俊氏および Naveen D.V.R. 氏に感謝致します．LAPP の LSI 試作における成功は皆様の御協力により実現できたものである．

平素から御助言と御指導を頂きました立命館大学の山下茂教授，山形大学の中西正樹准教授，龍谷大学の斉藤光徳教授および滋賀県立大学の山田逸成助教に深く感謝の意を表します．特に，山下茂教授と中西正樹准教授には博士前期課程在籍時より論文の執筆において有益な御助言と御指導を頂きました．斉藤光徳教授と山田逸成助教には龍谷大学理工学部卒業以降も研究活動全般について有益な御助言と御指導を頂きました．

研究活動において，有益な御討論と御協力を頂きました本学コンピューティング・アーキテクチャ講座の Marcos Daniel Villagra 氏，Tanvir Ahmed 氏，笹川幸宏氏，狭間洋平氏，内田行紀氏，村田絵理氏，森高晃大氏，鈴木大陽氏，大谷友哉氏，山中良祐氏，徐浩氏，王昊氏および関賀氏に感謝致します．また，平素から御協力を頂きました本学コンピューティング・アーキテクチャ講座 OB の渡邊良二氏，大賀健司氏，横峯大輔氏，里山宏平氏，本間知教氏，柴田章博氏，鈴木一範氏，平田雄一氏, 堀田敬一氏，深坂紘行氏, 洪勇基氏をはじめ，本学コンピューティング・アーキテクチャ講座 OB の皆様に感謝致します．

本研究の一部は半導体理工学研究センターとの共同研究によるものであり，科学研究費補助金（基盤研究（B) 課題番号 19300012）によるものである．また，本研究の一部は先端的低炭素化技術開発（次世代低電力デバイス安定化計算機構成方式）および科学研究費補助金（若手研究（B）課題番号 22700053）によるものである．本研究は東京大学大規模集積システム設計教育研究センターを通し，ローム (株)，メンター株式会社，シノプシス株式会社および日本ケイデンス株式会社の協力で行われたものである．

日常生活において，スポーツとリフレッシュの機会を与えてくれたフットサルチーム「ボンクラーズ」，龍谷大学理工学部電子情報学科 OB をはじめ，友人の皆様に感謝致します．また，英語能力向上のためにサポートして頂いた ECC（登美が丘校，西大寺校，生駒校および高の原校）の先生，スタッフおよびクラスメートの皆様に深く感謝致します．そして，就職活動において御協力と御指導を頂きましたキャリアサポートセンター内定塾の先生および 13 期塾生の皆様に感謝致します．さらに，社会人基礎力の向上において応援して頂いた富士通株式会社人材採用センターの皆様および内定者の皆様に感謝致します．

最後に，これまで学業を修めるにあたり多大な御支援と御協力を頂いた父 昌久および母 明美に深く感謝致します．

参考文献

[1] David A. Patterson and John L.Henessy. Computer Organization and Design THE HARDWARE / SOFTWARE INTERFACE. Morgan Kaufmann, 2005.

[2] Tobias G. Noll, Thorsten von Sydow, Bernd Neumann, Jochel Schen Schleifer, Thomas Coenen, and Götz Kappen. Reconfigurable Components for Application-Specific Processor Architectures, pp. 25–49. Springer, 2010.

[3] Lan Kuon and Jonathan Rose. Quantifying and Exploring the Gap Between FPGAs and ASICs. Springer, 2010.

[4] Shekhar Borkar and Andrew A. Chien. The Future of Microprocessors. Communications of the ACM, Vol. 54, pp. 67–77, 2011.

[5] Joo M.P. Cardoso and Pedro C. Diniz. Compilation Techniques for Reconfigurable Architectures. Springer, 1 edition, 2008.

[6] Yoonjin Kim and Rabi N. Mahapatra. Design of Low-Power Coarse-Grained Reconfigurable Architectures. CRC Press, 2010.

[7] 中田尚, 片岡晶人, 中島康彦. VLIW 型命令キューを持つスーパスカラプロセッサの命令スケジューリング機構. 情処学論. コンピューティングシステム, Vol. 2, No. 2, pp. 48–62, 2009.

[8] 安藤秀樹. 命令レベル並列処理 プロセッサアーキテクチャとコンパイラ. コロナ社, 2005.

[9] T. Shiota, K. Kawasaki, Y. Kawabe, W. Shibamoto, A. Sato, T. Hashimoto, F. Hayakawa, S. Tago, H. Okano, Y. Nakamura, H. Miyake, A. Suga, and H. Takahashi. A 51.2 GOPS 1.0 GB/s-DMA single-chip multi-processor integrating quadruple 8-way VLIW processors. In Proceeding of the 2005 IEEE International Solid-State Circuits Conference, Vol. 1, pp. 194 –593, 2005.

[10] Intel. Intel SSE4 Programming Reference, 2007.

[11] AMD. AMD 3DNow! Technology Manual, 2000.

[12] ARM. Architecture and Implementation of the ARM Cortex-A8 Microprocessor, 2005.

[13] Shorin Kyo, Takuya Koga, Lieske Hanno, Shouhei Nomoto, and Shin'ichiro Okazaki. A Low-cost Mixed-mode Parallel Processor Architecture for Embedded Systems. In Proceedings of the 21st Annual International Conference on Supercomputing, pp. 253–262, 2007.

[14] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous Multithreading: A Platform for Next-Generation Processors. IEEE Micro, Vol. 17, pp. 12–19, 1997.

[15] Intel. Hyper-Threading Technology Architecture and Microarchitecture, 2005.

[16] Manoj Gupta, Fermin Sanchez, and Josep Llosa. CSMT: Simultaneous Multithreading for Clustered VLIW Processors. IEEE Trans. Comput., Vol. 59, pp. 385–399, 2010.

[17] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. QoS Policies and Architecture for cache/memory in CMP Platforms. ACM SIGMETRICS Performance Evaluation Review, Vol. 35, No. 1, pp. 25–36, 2007.

[18] Jichuan Chang and Gurindar S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In Proceedings of the 21st Annual International Conference on Supercomputing, pp. 242–252, 2007.

[19] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In Proceedings of the 23rd Annual International Symposium on Computer Architecture, pp. 191–202, 1996.

[20] Stijn Eyerman and Lieven Eeckhout. Memory-level parallelism aware fetch policies for simultaneous multithreading processors. ACM Trans. Archit. Code Optim., Vol. 6, pp. 3:1–3:33, 2009.

[21] Dean M. Tullsen and Jeffery A. Brown. Handling Long-latency Loads in a Simultaneous Multithreading Processor. In Proceedings of the 34th International Symposium on Microarchitecture, pp. 318–327, 2001.

[22] Alvin R. Lebeck, Jinson Koppanalil, Tong Li, Jaidev Patwardhan, and Eric Rotenberg. A Large, Fast Instruction Window for Tolerating Cache Misses. In Proceedings of the 29th Annual International Symposium on Computer Architecture, pp. 59–70, 2002.

[23] Texas Instruments. OMAP-L138 C6-Integra DSP+ARM Processor, 2011.

[24] T. Furusawa, I. Katayama, Y. Arai, S. Inoue, M. Matsui, M. Nishikawa, and T. Yoshimoto. A DSP Engine for an Extensible Media Embedded Processor. In Proceedings of the IEEE Asia-Pacific Conference, Advanced System Integrated Circuits 2004, pp. 160–163, 2004.

[25] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. In Proceeding of the 2005 IEEE International Solid-State Circuits Conference, Vol. 1, pp. 184 –592, 2005.

[26] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A Many-core x86 Architecture for Visual Computing. In Proceedings of the ACM SIGGRAPH 2008, pp. 1–15, 2008.

[27] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-Chip Interconnection Architecture of the Tile Processor. IEEE Micro, Vol. 27, No. 5, pp. 15–31, 2007.

[28] NVIDIA. Fermi: NVIDIA ' s Next Generation CUDA Compute Architecture, 2011.

[29] 入江英嗣, 服部直也, 山口健輔, 谷地田瞬, 田中裕治, 坂井修一, 田中英彦. VLDP3：データフローを高速実行する大規模アーキテクチャ. 情処学研報, Vol. 2003, No. 10, pp. 49–54, 2003.

[30] Farhad Mehdipour, Hiroaki Honda, Hiroshi Kataoka, Koji Inoue, and Kazuaki Murakami. An Accelerator Based on Single-Flex Quantum Circuits for a High-Performance Reconfigurable Computer. In Proceedings of Workshop on Accelerators for High-performance Architectures, 2009.

[31] Mark Gebhart, Bertrand A. Maher, Katherine E. Coons, Jeff Diamond, Paul Gratz, Mario Marino, Nitya Ranganathan, Behnam Robatmili, Aaron Smith, James Burrill, Stephen W. Keckler, Doug Burger, and Kathryn S. McKinley. An Evaluation of the TRIPS Computer System. In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 1–12, 2009.

[32] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, William Yoder, and the TRIPS Team. Scaling to the End of Silicon with EDGE Architectures. IEEE Computer, Vol. 37, No. 7, pp. 44–55, 2004.

[33] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Exploiting Loop-Level Parallelism on Coarse-Grained Reconfigurable Architectures Using Modulo Scheduling. In Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1, pp. 296 – 301, 2003.

[34] Bingfeng Mei, Serge Vernalde, Diederik Verkest, and Rudy Lauwereins. Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture: A Case Study. In Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2, pp. 1224 – 1229, 2004.

[35] Bjorn De Sutter, Paul Coene, Tom Vander Aa, and Bingfeng Mei. Placement-and-routing-based Register Allocation for Coarse-grained Reconfigurable Arrays. In Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 151–160, 2008.

[36] Hyunchul Park, Yongjun Park, and Scott Mahlke. Polymorphic Pipeline Array: A Flexible Multicore Accelerator with Virtualized Execution for Mobile Multimedia Applications. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 370–380, 2009.

[37] Basher Shehan, Ralf Jahr, Sascha Uhrig, and Theo Ungerer. Reconfigurable Grid Alu Processor: Optimization and Design Space Exploration. Digital Systems Design, Euromicro Symposium on, pp. 71–79, 2010.

[38] Sascha Uhrig, Basher Shehan, Ralf Jahr, and Theo Ungerer. A Two-Dimensional Superscalar Processor Architecture. Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, Computation World, Vol. 0, pp. 608–611, 2009.

[39] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. SIGARCH Comput. Archit. News, Vol. 25, pp. 13–25, 1997.

[40] Ricardo Santos, Rodolfo Azevedo, and Guido Araujo. 2D-VLIW: An Architecture Based on the Geometry of Computation. In Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors, pp. 87–94, 2006.

[41] Ricardo Santos, Rafael Batistella, and Rodolfo Azevedo. A Pattern Based Instruction Encoding Technique for High Performance Architectures. Int. J. High Perform. Syst. Archit., Vol. 2, pp. 71–80, 2009.

[42] Nobuyuki Ohba and Kohji Takano. An SoC Design Methodology Using FPGAs and Embedded Microprocessors. In Proceedings of the 41st Design Automation Conference, pp. 747–752, 2004.

[43] Naraig Manjikian, Huang Jin, James Reed, and Nathan Cordeiro. Architecture and Implementation of Chip Multiprocessors: Custom Logic Components and Software for Rapid Prototyping. In Proceedings of the 2004 International Conference on Parallel Processing, pp. 483–492, 2004.

[44] Jeffry T Russell and Margarida F Jacome. Architecture-Level Performance Evaluation of Component-Based Embedded Systems. In Proceedings of the 40th Design Automation Conference, pp. 396–401, 2003.

[45] Wander O.Cesário, Damien Lyonnard, Gabriela Nicolescu, Yanick Paviot, Sungjoo Yoo, Ahmed A.Jerraya, Lovic Gauthier, and Mario Diaz-Nava. Multiprocessor SoC Platforms: A Component-Based Design Approach. IEEE Design & Test of Computers, Vol. 19, No. 6, pp. 52–63, 2002.

[46] J.F Nezan, O Deforges, and M Raulet. Rapid Prototyping Methodology For multi-DSP TI C6X Platforms Applied to an Mpeg-2 Coding Application. In Proceedings of the 40th ACM Symposium on Parallel Algorithms and Architectures, pp. 147–148, 2002.

[47] Alberto Sangiovanni-Vincentelli and Grant Martin. Platform-Based Design and Software Design Methodology for Embedded Systems. IEEE Design & Test of Computers, Vol. 18, No. 6, pp. 23–33, 2001.

[48] Sewook Wee, Jared Casper, Njuguna Njoroge, Yuriy Tesylar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun. A Practical FPGA-based Framework for Novel

CMP Research. In Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays, pp. 116–125, 2007.

[49] John D. Davis, Stephen E. Richardson, Charis Charitsis, and Kunle Olukotun. A Chip Prototyping Substrate: The Flexible Architecture for Simulation and Testing (FAST). ACM SIGARCH Computer Architecture News, Vol. 33, No. 4, pp. 34–43, 2005.

[50] Ben Serebrin, John D. Owens, Chen H. Chen, Stephen P. Crago, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, Scott Rixner, and William J. Dally. A Stream Processor Development Platform. In Proceedings of the 2002 IEEE International Conference on Computer Design, pp. 303–308, 2002.

[51] 若杉祐太, 佐藤真平, 植原昂, 藤枝直輝, 渡邉伸平, 高前田伸也, 森洋介, 吉瀬謙二. 極めて低コストで効率的な VDEC チップ試作・検証システムの開発と応用. 情処学研報, Vol. 2009, No. 6, pp. 1–8, 2009.

[52] Yoshiki Saito, Toru Sano, Masaru Kato, Vasutan Tunbunheng, Yoshihiro Yasuda, Masayuki Kimura, and Hideharu Amano. MuCCRA-3: A Low Power Dynamically Reconfigurable Processor Array. In Proceedings of the 15th Asia and South Pacific Conference on Design Automation, pp. 377–378, 2010.

[53] 伊藤伸也, 野本祥平, 冨安洋史, 西川博昭. データ駆動・制御駆動スレッドを同時・多重処理するプロセッサ CUE-v2 の LSI 試作. 信学論 (D-I), Vol. J88-D-I, No. 2, pp. 113–124, 2005.

[54] Nathaniel Pinckney, Thomas Barr, Michael Dayringer, Matthew McKnett, Nan Jiang, Carl Nygaard, David Money Harris, Joel Stanley, and Braden Phillips. A MIPS R2000 IMPLEMENTATION. In Proceedings of the 45th Design Automation Conference, pp. 102–107, 2008.

[55] http://www.uclinux.org/. μClinux.

[56] RTL 設計スタイルガイド Verilog-HDL 編. STARC, 2003.

[57] Fujitsu. FR-V Family FR550 Series Instruction Set Manual Ver. 1.1, 2002.

[58] ARM. ARM Architecture Reference Manual, ARM DDI0100E, 2000.

[59] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In Proceedings of the 4th IEEE International Workshop on Workload Characterization, pp. 3–14, 2001.

[60] 吉村和浩, 中田尚, 中島康彦. 異種命令 SMT プロセッサ OROCHI の実装と分析. 情処学研報, Vol. 2008, No. 75, pp. 1–6, 2008.

[61] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and R. Reed Taylor. PipeRench: A Reconfigurable Architecture and Compiler. IEEE Computer, Vol. 33, pp. 70–77, 2000.

[62] Kazuya Tanigawa, Ryuji Hada, Tetsuo Hironaka, and Akira Kojima. A Reconfigurable Processor PARS and its Compiler. In Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems, pp. 91–100, 2007.

[63] Simon Segars. Low Power Design Techniques for Microprocessors. In Proceeding of the 2001 IEEE International Solid-State Circuits Conference Tutorial, 2001.

[64] Preeeti Ranjan Panda, Aviral Shrivastava, B.V.N. Silpa, and Krishnaiah Gummidipudi. Power-efficient System Design. Springer, 2010.

[65] Hailin Jiang, Malgorzata Marek-Sadowska, and Sani R. Nassif. Benefits and costs of power-gating technique. Vol. 0, pp. 559–566, 2005.

[66] Johan Pouwelse, Koen Langendoen, and Henk Sips. Dynamic voltage scaling on a low-power microprocessor. In Proceedings of the 7th Annual International Conference on Mobile Computing and Networking, pp. 251–259, 2001.

[67] 中田尚, 上利宗久, 中島康彦. 画像処理向け線形アレイ VLIW プロセッサ. 先進的計算基盤システムシンポジウム SACSIS 2009, pp. 293–300, 2009.

[68] M. S. Hrishikesh, Doug Burger, Norman P. Jouppi, Stephen W. Keckler, Keith I. Farkas, and Premkishore Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 fo4 inverter delays. In Proceedings of the 29th Annual International Symposium on Computer Architecture, pp. 14–24, 2002.

[69] Viji Srinivasan, David Brooks, Michael Gschwind, Pradip Bose, Victor Zyuban, Philip N. Strenski, and Philip G. Emma. Optimizing pipelines for power and performance. In Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture, pp. 333–344, 2002.

[70] Steven J. E. Wilton and Norman P. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. IEEE Journal of Solid-State Circuits, Vol. 31, pp. 677–688, 1996.

[71] K. Flautner, Nam Sung Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In Proceedings of the 29th Annual International Symposium on Computer Architecture, pp. 148 –157, 2002.

業績

査読付学術論文

[1] 吉村 和浩, 中田 尚, 中島 康彦, 北村 俊明: “異種命令セットアーキテクチャを持つ高電力効率 SMT プロセッサの開発”, 電子情報通信学会論文誌, Vol. J95-D, No. 6, June, 2012.（採録決定）

[2] 中田 尚, 吉村 和浩, 下岡 俊介, 大上 俊, Naveen D.V.R., 中島 康彦: “画像処理向け線形アレイアクセラレータの性能評価”, 情報処理学会論文誌コンピューティングシステム, Vol. 5, No. 2, (ACS38), 2012.（採録決定）

[3] Kazuhiro Yoshimura, Takuya Iwakami, Takashi Nakada, Jun Yao, Hajime Shimada, Yasuhiko Nakashima: “An Instruction Mapping Scheme for FU Array Accelerator”, IEICE Transactions on Information and Systems, Vol. E94-D, No. 2, pp. 286–297, February, 2011.

査読付国際会議

[1] Jun Yao, Ryoji Watanabe, Kazuhiro Yoshimura, Takashi Nakada, Hajime Shimada, Yasuhiko Nakashima: “An Efficient and Reliable 1.5-way Processor by Fusion of Space and Time”, The 5th Workshop on Dependable and Secure Nanocomputing (WDSN11), pp. 69–74, June, 2011.

[2] Naveen D.V.R., Takuya Iwakami, Kazuhiro Yoshimura, Takashi Nakada, Jun Yao, Yasuhiko Nakashima: “LAPP: A Low Power Array Accelerator with Binary Compatibility”, The 7th Workshop on High-Performance, Power-Aware Computing (HPPAC2011), pp.849–857, May, 2011.

[3] Takuya Iwakami, Munehisa Agari, Kazuhiro Yoshimura, Takashi Nakada, Yasuhiko Nakashima: “Area Optimization of FU Array in Low-Power Accelerators”, IEEE International Symposium on Low-Power and High-Speed Chips, COOL Chips XIII (Poster12), p. 194, April 14–16, 2010.

[4] Kazuhiro Yoshimura, Takashi Nakada, Yasuhiko Nakashima, Toshiaki Kitamura: “An Energy Efficient SMT Processor with Heterogeneous Instruction Set Architectures”, IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN2010), pp. 201–209, February 16–18, 2010.

[5] Kazuhiro Yoshimura, Takashi Nakada, Yasuhiko Nakashima: “An Instruction Decomposition Method for Reconfigurable Decoders”, IEEE International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA2009), March 16–17, 2009.

[6] Kazuhiro Yoshimura, Takashi Nakada, Yasuhiko Nakashima: “A QoS Control Method for a Heterogeneous SMT Processor”, IEEE International Symposium on Low-Power and High-Speed Chips, COOL Chips XI (Poster19), p. 133, April 16–18, 2008.

査読付国内会議

[1] 岩上 拓矢，吉村 和浩，中田 尚，中島 康彦: “仮想化機構による演算器アレイ型アクセラレータの効率化”, 第 9 回先進的計算基盤システムシンポジウム (SACSIS2011), pp. 136–143, 2011 年 5 月 25–27 日.

[2] 森 浩大，大上 俊，下岡 俊介，吉村 和浩，中田 尚，中島 康彦: “演算器アレイ型アクセラレータのための命令変換手法”, 第 9 回先進的計算基盤システムシンポジウム (SACSIS2011) ポスター, pp. 207–208, 2011 年 5 月 25–27 日.

[3] 岩上 拓矢, 吉村 和浩, 上利 宗久, 中田 尚, 中島 康彦: “プログラマビリティを備える低電力アクセラレータの評価”, 第 8 回先進的計算基盤システムシンポジウム (SACSIS2010) ポスター, pp. 103–104, 2010 年 5 月 27–28 日.

国内研究会および大会

[1] 森高 晃大, 下岡 俊介, 吉村 和浩, 姚 駿, 中田 尚, 中島 康彦: “大規模演算器アクセラレータのための複数 FPGA 連結手法”, デザインガイア 2011, 電子情報通信学会技術研究報告 CPSY2011-44, pp. 9–14, 2011 年 11 月 28–30 日.

[2] 齊藤 光俊, 下岡 俊介, 吉村 和浩, 姚 駿, 中田 尚, 中島 康彦: “演算器アレイ型アクセラレータの実装とその分析”, デザインガイア 2011, 電子情報通信学会技術研究報告 ICD2011-91, pp. 53–58, 2011 年 11 月 28–30 日.

[3] 大上 俊, 吉村 和浩, 姚 駿, 中田 尚, 中島 康彦: “演算器アレイにおける高信頼化命令写像手法”, 2011 年並列/分散/協調処理に関する『鹿児島』サマー・ワークショップ (SWoPP 鹿児島 2011), 情報処理学会研究報告 2011-196(19), pp. 1–7, 2011 年 7 月 27–29 日.

[4] 下岡 俊介, 吉村 和浩, 中田 尚, 中島 康彦: “演算器アレイ型アクセラレータにおけるローカルバッファの最適化”, 2011 年並列/分散/協調処理に関する『鹿児島』サマー・ワークショップ (SWoPP 鹿児島 2011), 情報処理学会研究報告 2011-196(18), pp. 1–6, 2011 年 7 月 27–29 日.

[5] 岩上 拓矢, 吉村 和浩, 森 浩大, 中田 尚, 中島 康彦: “演算器アレイを拡張する細粒度時分割機構”, 集積回路研究会 (ICD), 電子情報通信学会技術研究報告 ICD2010-123, pp. 141–146, 2010 年 12 月 16–17 日.

[6] 大上 俊, 岩上 拓矢, 吉村 和浩, 中田 尚, 中島 康彦: “アレイ型アクセラレータにおける演算器間ネットワークの設計”, 集積回路研究会 (ICD) ポスター, 電子情報通信学会技術研究報告 ICD2010-123, pp. 95–96, 2010 年 12 月 16–17 日.

[7] 下岡 俊介, 岩上 拓矢, 吉村 和浩, 中田 尚, 中島 康彦: “演算器アレイ型アクセラレータにおけるメモリアクセス機構の設計”, 集積回路研究会 (ICD) ポスター, 電子情報通信学会技術研究報告 ICD2010-123, pp. 97–99, 2010 年 12 月 16–17 日.

[8] 森 浩大, 岩上 拓矢, 吉村 和浩, 中田 尚, 中島 康彦: “演算器アレイ型アクセラレータのための命令変換手法の検討”, 2010 年並列/分散/協調処理に関する『金沢』サマー・ワークショップ (SWoPP 金沢 2010), 情報処理学会研究報告 2010-196, pp. 1–6, 2010 年 8 月 3–5 日.

[9] 吉村 和浩, 上利 宗久, 中田 尚, 中島 康彦: “演算器アレイ型プロセッサのための命令スケジューラの設計と評価”, 組込み技術とネットワークに関するワークショップ (ETNET2010), 電子情報通信学会技術研究報告 CPSY2009-94, pp.511–516, 2010 年 3 月 26–28 日.

[10] 市来 亮人, 吉村 和浩, 中田 尚, 中島 康彦: “異種命令 SMT プロセッサ OROCHI の ASIC 試作における問題と対策”, 平成 20 年度情報処理学会関西支部大会, 講演論文集, pp. 13–16, 2008 年 10 月 24 日.

[11] 吉村 和浩, 中田 尚, 中島 康彦: “異種命令 SMT プロセッサ OROCHI の実装と分析”, 2008 年並列/分散/協調処理に関する『佐賀』サマー・ワークショップ (SWoPP 佐賀 2008), 情報処理学会研究報告 2008-ARC-179, pp. 1–6, 2008 年 8 月 5–7 日.

[12] 吉村 和浩, 北村 俊明: “異種命令混在実行プロセッサにおける QoS 制御”, STARC フォーラム/シンポジウム 2008 学生ポスター, 2008 年 7 月 16–17 日.

受賞および表彰

[1] 大上 俊, 岩上 拓矢, 吉村 和浩, 中田 尚, 中島 康彦: “アレイ型アクセラレータにおける演算器間ネットワークの設計”, 集積回路研究会 (ICD) ポスター, 電子情報通信学会技術研究報告 ICD2010-123, pp. 95–96, 2010 年 12 月 16–17 日. 電子情報通信学会集積回路研究会優秀若手研究ポスター賞受賞

[2] 吉村 和浩, 市来 亮人, 中田 尚, 中島 康彦: “異種命令混在実行プロセッサ OROCHI の開発”, LSI とシステムのワークショップ 2009 学生ポスター, pp. 191–193, 2009 年 5 月 17–19 日. IEEE Solid-State Circuits Society Japan Chapter Academic Research Award 受賞

その他

[1] 吉村 和浩: “CPU の現状と将来”, 平成 21 年度奈良県立奈良北高等学校「総合的な学習授業」出前講義，2009 年 6 月 16 日.