# 論文 / 著書情報
# Article / Book Information

| | |
|---|---|
| 題目(和文) | 新しいRTLモデリングによるメニーコアプロセッサの設計 |
| Title(English) | Design of a Manycore Processor on a Novel RTL Modeling Method |
| 著者(和文) | 佐藤真平 |
| Author(English) | Shimpei Sato |
| 出典(和文) | 学位:博士（工学）,<br>学位授与機関:東京工業大学,<br>報告番号:甲第9662号,<br>授与年月日:2014年9月25日,<br>学位の種別:課程博士,<br>審査員:吉瀬　謙二,横田　治夫,宮崎　純,金子　晴彦,渡部　卓雄 |
| Citation(English) | Degree:,<br>Conferring organization: Tokyo Institute of Technology,<br>Report number:甲第9662号,<br>Conferred date:2014/9/25,<br>Degree Type:Course doctor,<br>Examiner:,,,, |
| 学位種別(和文) | 博士論文 |
| Type(English) | Doctoral Thesis |

TOKYO TECH Pursuing Excellence
TOKYO INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

平成 26 年度 学位論文

# 新しい RTL モデリングによる メニーコアプロセッサの設計

東京工業大学
大学院情報理工学研究科 計算工学専攻

佐 藤　　真 平

平成 26 年 9 月

# 概要

本論文では，新しい RTL モデリング手法を用いてメニーコアプロセッサを設計，評価する．プロセッサや GPU，FPGA などの LSI に集積されるトランジスタ数は増加している．トランジスタの増加に伴い，より大規模な回路が設計可能となったが，回路規模に対するシングルコアのプロセッサの性能向上は鈍化している．このため，プロセッサの性能向上は，これまでのシングルコアのプロセッサによる性能向上から，複数のコアを搭載するマルチコアプロセッサによる性能向上へとシフトしている．プロセッサのコア数は増加し，マルチコアからメニーコアの時代へと向かっており，様々なメニーコアアーキテクチャを効率的に設計，評価する必要がある．

プロセッサなどの LSI の設計はアーキテクチャ設計，論理設計，回路設計，物理設計というフローで行われる．主に，アーキテクチャ設計と論理設計においてアーキテクチャの評価が必要となる．一般に，アーキテクチャ設計では，C 言語や C++ 言語などの汎用プログラミング言語で評価対象をモデリングし評価する．このときのアーキテクチャは，機能レベル，動作レベル，レジスタトランスファレベル（RTL）など様々な抽象度で記述される．論理設計では，Verilog HDL や VHDL などのハードウェア記述言語を用いて評価対象をモデリング，評価する．このとき，ハードウェアは RTL で記述されることが一般的である．

アーキテクチャ設計において汎用プログラミング言語を用いてハードウェアを記述し評価する理由には，シミュレーションが高速である点や柔軟なパラメータサーベイが可能である点が挙げられる．論理設計においてハードウェア記述言語が用いられる理由は，論理合成ができ，論理設計以降のフローである回路設計，物理設計に必要となるためである．このように，アーキテクチャ設計と論理設計では，同じハードウェアのために異なる言語による記述がなされており，複雑なハードウェアになるほどそれぞれの記述の等価性を検証するコストが高くなる．効率的なアーキテクチャの設計，評価には，このアーキテクチャ設計と論理設計における検証のコストを改善する必要がある．

一方，複数のコアを搭載するマルチコアプロセッサ，メニーコアプロセッサでは，並列

処理によりアプリケーションの性能向上を図る．複数のコアを用いた並列処理にはコア間の通信が不可欠で，チップ上に数コアを搭載するマルチコアプロセッサではバスを用いた通信路によりコア間の通信を行っている．10 コア以上を搭載するメニーコアプロセッサでは，バスによる通信ではレイテンシ，スループットの性能不足より十分にアプリケーションの並列性を抽出できないため，Network-on-Chip（NoC）と呼ばれるルータを介したネットワークが用いられる．メニーコアプロセッサの設計において，NoC のルータアーキテクチャが重要な要素となる．

本論文では，メニーコアプロセッサを設計するにあたり，まずハードウェアを RTL モデリングするための新しい環境を提案する．この環境は，C++ でハードウェアを記述する新しいハードウェア記述言語 ArchHDL と，ArchHDL のソースコードを Verilog HDL に自動変換するツールから成る．これにより，Verilog HDL と同じ抽象度による RTL 記述にもかかわらず C++ によるハードウェアモデリングの利点である柔軟で高速なシミュレーションを達成する．したがって，アーキテクチャ設計と論理設計を ArchHDL のみで行うことが可能となる．また，Verilog HDL に変換できることから，容易に回路設計，物理設計といったフローへ移行することができる．

次に，メニーコアプロセッサの重要な要素である高性能な NoC ルータアーキテクチャを提案する．高スループットな NoC ルータとして Distributed Shared-Buffer Router（DSB ルータ）が知られている．しかし，このルータはレイテンシの点では典型的な NoC ルータに劣る．本研究では，ルータのレイテンシ削減手法を提案し，DSB ルータに適用する．提案手法により，DSB ルータの特徴であるスループットの性能を維持しつつ低レイテンシ化を達成する．

最後に，メニーコアアープロセッサを設計し評価する．研究・教育を目的として設計されたメニーコアアーキテクチャ M-Core をベースに，提案する高性能な NoC ルータを搭載し，ハードウェアに改良を加えたメニーコアプロセッサを設計する．新しいハードウェア設計環境を用いた様々なパラメータによる評価から，NoC のアーキテクチャを策定する．そして，並列アプリケーションを用いた評価により，設計するメニーコアプロセッサにおける高性能な NoC ルータによる性能向上を示す．

# 目次

# 図目次

# 表目次

# 第 1 章

# はじめに

## 1.1 研究の目的

プロセッサや GPU，FPGA などの LSI に集積されるトランジスタ数は増加傾向にある [1]．図 1.1 に，2010 年までのトランジスタ数，プロセッサの性能，コア数などの推移をまとめたグラフを示す．このグラフは文献 [1] より抜粋した．チップ上に集積可能なトランジスタは設計プロセスの微細化技術により増加してきたが，これによるにトランジスタ数の増加は鈍化している [2]．しかし，3 次元積層技術などにより今後も LSI に集積されるトランジスタ数は増加していくことが予測される [3]．トランジスタの増加に伴い，より大規模な回路が設計可能となったが，回路規模に対するシングルコアのプロセッサの性能向上は鈍化している [4]．このため，プロセッサの性能向上は，これまでのシングルコアから，複数のコアを搭載するマルチコアプロセッサによる性能向上へとシフトしている．プロセッサのコア数は増加傾向にあり，マルチコアからメニーコアの時代へと向かっている．今後，様々なメニーコアアーキテクチャを効率的に設計，評価する必要がある．

プロセッサなどの LSI の設計はアーキテクチャ設計，論理設計，回路設計，物理設計というフローで行われる [5, 6]．アーキテクチャ設計と論理設計においては，アーキテクチャの評価が必要となる．一般に，アーキテクチャ設計では，C 言語や C++ 言語などの汎用プログラミング言語で評価対象をモデリングする．このときのアーキテクチャは，機能レベル，動作レベル，レジスタトランスファレベル（RTL）などの様々な抽象度で記述される．論理設計では，Verilog HDL や VHDL などのハードウェア記述言語を用いてモデリングする．このとき，一般に評価対象は RTL で記述される．

アーキテクチャ設計において汎用プログラミング言語を用いてハードウェアを記述する理由には，シミュレーションが高速である点や柔軟なパラメータサーベイが可能である点

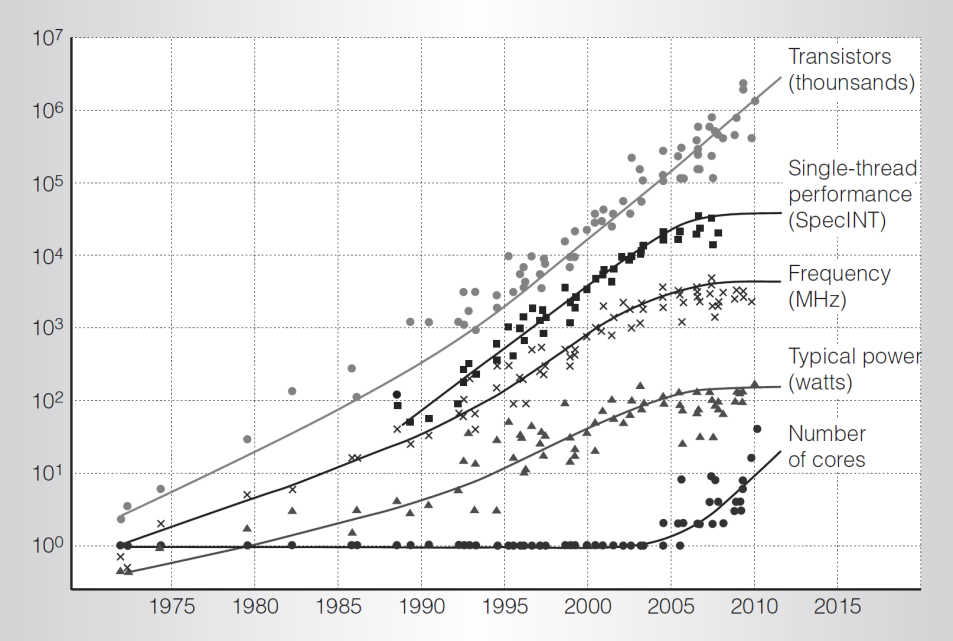


図 1.1　2010 年までのトランジスタ数，プロセッサの性能，コア数などの推移．（文献 [1] より抜粋）

が挙げられる．論理設計においてハードウェア記述言語が用いられる理由は，論理合成ができ，論理設計以降のフローである回路設計，物理設計における検証に必要となるためである．このように，アーキテクチャ設計と論理設計の段階では，同じハードウェアのために異なる言語による記述がなされており，複雑なハードウェアになるほどそれぞれの記述の等価性を検証するコストが高くなる．効率的なアーキテクチャの設計，評価には，このアーキテクチャ設計と論理設計における検証のコストを改善する必要がある．

複数のコアを搭載するマルチコアプロセッサ，メニーコアプロセッサでは，並列処理によりアプリケーションの性能向上を図る．複数のコアを用いた並列処理にはコア間の通信が不可欠で，チップ上に数コアを搭載するマルチコアプロセッサではバスを用いた通信路 [7, 8] によりコア間の通信を行っている．一方，10 コア以上を搭載するメニーコアプロセッサでは，バスによる通信ではレイテンシ，スループットの性能不足より十分にアプリケーションの並列性を抽出できないため，Network-on-Chip（NoC）と呼ばれる通信路が用いられる [9, 10, 11, 12, 13].

NoC は，コア同士をルータを介して接続するチップ上に構成されるネットワークである．ルータを介したパケット通信によりコア間の通信を行うため，コア数に応じたスケーラビリティの点でバスよりも優れている．図 1.2 に，典型的なメニーコアプロセッサとそ

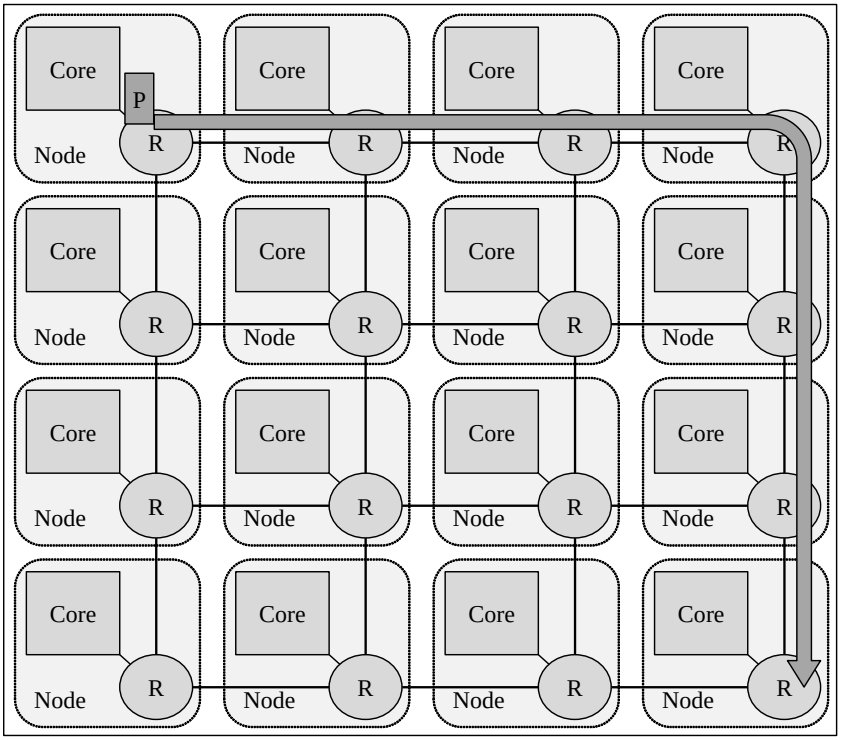


図 1.2 Network-on-Chip で接続されたメニーコアプロセッサにおけるパケット転送

のプロセッサにおいてパケット通信が行われる様子を示す．このメニーコアプロセッサは複数の**ノード**がタイル状に並べられた構成のプロセッサである．ノードは，**コア**と**ルータ**で構成される．コアは，Processing Element（PE），キャッシュやメモリなどの要素で構成される．各コアは，ルータ（図中の R）により形成される 2 次元メッシュ状のネットワークで接続されている．図は，左上のコアから右下のコアへ通信を行っている様子を示している．この例では，左上のコアで生成されたパケット（図中の P）は，まず右方向に転送され，次に下方向に転送され宛先のコアに到達する．

NoC におけるレイテンシとスループットは，アプリケーションの並列処理性能に影響を与える．レイテンシとスループットに影響を与える NoC の要素にトポロジ，ルーティング，ルータアーキテクチャがある [11]．トポロジは，ルータ同士の接続によって形成されるネットワークの形で，図 1.2 では各ルータは上下左右の 4 方向のルータと接続し 2 次元メッシュのネットワークを形成している．ルーティングは，パケットが通過する経路の決定方法で，図 1.2 では 2 次元メッシュで典型的な次元順ルーティングを例としている．レイテンシ関しては，ホップ数とパケットを次のホップに転送するための遅延を考慮する必要がある．トポロジとルーティングによってパケットが送信先に到達するまで通過する

ルータの数（ホップ数）が決まる．ルータアーキテクチャによって，パケットを次のホップに転送するための遅延が決まる．スループットに関しては，バイセクションバンド幅という指標がある．これは，ネットワークを最もバンド幅が狭くなるように 2 等分したときのバンド幅である．トポロジによってバイセクションバンド幅が決まり，ルーティングとルータアーキテクチャによってそのバンド幅に対する実効性能が決まる．メニーコアプロセッサの設計では，トポロジは 2 次元メッシュが，ルーティングは次元順ルーティングが採用されることが多い．したがって，メニーコアプロセッサにおける並列処理性能の向上には高性能な NoC ルータが必要となる．

本研究は，新しいハードウェアの RTL モデリング環境を提案し，その上でメニーコアプロセッサを設計する．まず，新しいハードウェアの RTL モデリング環境として，C++ 言語でハードウェアを RTL で記述することができ，かつ高速にシミュレーションを行うことができる設計環境を提案する．そして，メニーコアプロセッサの性能を決定する要素として NoC のルータアーキテクチャに注目し，高スループットかつ低レイテンシを実現するルータアーキテクチャを提案する．最後に，新しい設計環境を利用して，提案する NoC ルータを搭載するメニーコアプロセッサを設計する．並列アプリケーションを用いた評価から，提案する NoC ルータによる性能向上を示す．

## 1.2 本研究の意義

本研究による貢献を以下に挙げる．

**効率的にハードウェアを設計，評価するための新しい設計環境の提案**

効率的にハードウェアを設計, 評価するための新しい設計環境では, C++ 言語でハードウェアを RTL で設計するための新しい言語 ArchHDL を提案する．ArchHDL が提供するライブラリを利用して記述したハードウェアは，RTL で記述されているにもかかわらず，C++ 言語による柔軟なパラメータサーベイが可能である．また，ライブラリによりシミュレーションが並列化でき，商用の Verilog HDL シミュレータよりも高速に RTL で記述されたハードウェアのシミュレーションが可能となる．同時に，ArchHDL で記述したソースコードを Verilog HDL のソースコードに自動変換するツールを開発する．これにより，ハードウェア記述に制約が生じるが，ArchHDL でハードウェアを記述するだけで Verilog HDL のソースコードを生成でき，論理合成ツールによる FPGA 実装などが可能になる．

**メニーコアプロセッサのための高性能な Network-on-Chip ルータの提案**

メニーコアプロセッサのための高性能な Network-on-Chip（NoC）ルータの提案では，高スループットな NoC ルータとして知られる Distributed Shared-Buffer NoC ルータ [14]（DSB ルータ）に，アーキテクチャの改良によるレイテンシを削減する提案手法を適用し，DSB ルータの高性能化を達成する．

**新しいハードウェア設計環境を利用した，メニーコアプロセッサの設計と評価**　提案する NoC ルータを搭載するメニーコアプロセッサを設計し評価する．評価には，提案する新しいハードウェア設計環境を用いる．研究・教育を目的として設計されたメニーコアアーキテクチャ M-Core[15] をベースに，提案する高性能な NoC ルータを搭載し，ハードウェアの改良を加えたメニーコアプロセッサを設計する．並列アプリケーションを用いた評価から，提案する NoC ルータによる性能向上を示す．

## 1.3 本論文の構成

本論文の構成を以下に示す．本論文は全 6 章から成る．

**「第 2 章 メニーコアアーキテクチャ」**では，本研究の背景であるメニーコアアーキテクチャおよび Network-on-Chip（NoC）ルータアーキテクチャについて述べる．まず，メニーコアプロセッサに関わる用語を整理し，NoC について概説する．次に，これまでに報告されているメニーコアアーキテクチャをいくつか紹介する．さらに，NoC におけるレイテンシを削減することを目的としたルータアーキテクチャを紹介する．

**「第 3 章 ハードウェアの RTL モデリングのための新しい設計環境の提案」**では，C++ でハードウェアを RTL で記述する新しいハードウェア記述言語 ArchHDL を提案する．また，ArchHDL で記述したソースコードを Verilog HDL に自動変換するためのトランスレーションツールを開発する．評価において，ArchHDL で記述したハードウェアのシミュレーション速度と商用の Verilog HDL シミュレータによるシミュレーション速度を比較し，同じ抽象度で記述したハードウェアのシミュレーションが商用のツールよりも高速に行えることを示す．

**「第 4 章 高性能 Network-on-Chip ルータアーキテクチャ」**では，NoC ルータのレイテンシを削減する手法を提案し，DSB ルータに適用する．合成トラフィックを用いたネットワークの評価から，提案手法が DSB ルータのレイテンシを削減することを示す．

**「第 5 章 メニーコアプロセッサの設計」**では，メニーコアアーキテクチャ M-Core をベースに，前章で提案した NoC ルータを搭載し，ハードウェアの改良を加えたメニーコアプロセッサを設計し評価する．ArchHDL を用いた評価により，メニーコアプロセッサの NoC

のパラメータを策定し，並列アプリケーションを用いて設計するメニーコアプロセッサの性能を評価する．

最後に，**「第 6 章 おわりに」**で本論文をまとめる．本研究の貢献を整理し，今後の展望について述べる．

# 第 2 章

# メニーコアアーキテクチャ

本章では，いくつかのメニーコアアーキテクチャと Network-on-Chip ルータアーキテクチャを紹介する．まず，メニーコアアーキテクチャを説明するために本論文で用いる用語を定義し，Network-on-Chip について概説する．そして，メニーコアアーキテクチャとレイテンシ削減を目的とするルータアーキテクチャを紹介する．

## 2.1 メニーコアアーキテクチャに関する用語の定義

本論文で用いるメニーコアアーキテクチャに関する用語を定義する．本研究は，タイル型のメニーコアプロセッサの設計を目的としている．そのため，図 2.1 に示すタイル型のメニーコアアーキテクチャを例に用語を整理する．

**プロセッサ**　計算機の構成要素である CPU を指す．

**シングルコアプロセッサ**　1 コアのプロセッサを指す．

**マルチコアプロセッサ**　2〜4 コアのプロセッサを指す．

**メニーコアプロセッサ**　マルチコアプロセッサより多くのコアを搭載するプロセッサを指す．図 2.1 に示すような，複数の同じ要素（図中の Node），メモリコントローラ，I/O コントローラなどを搭載する．

**ノード**　メニーコアプロセッサを構成する要素．タイル型のアーキテクチャでは，同じノードが複数搭載される．ノードは，通信のためのルータとコアで構成される．

**コア**　演算処理を行うための論理回路（Processing Element，PE），キャッシュやメモリなどで構成されるユニット．メニーコアプロセッサでは，通信のためのネットワークインタフェースなどもコアに含む．

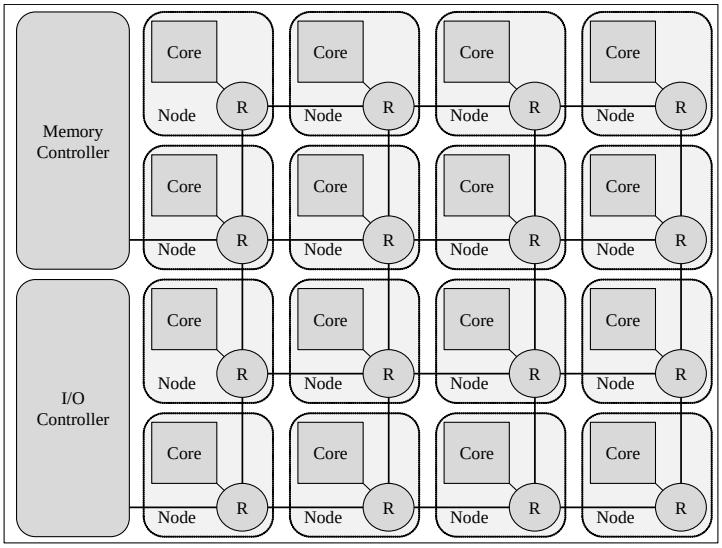


図 2.1 タイル型のメニーコアアーキテクチャの例

**ルータ** メニーコアプロセッサの構成要素間で通信を行うためのパケットを転送するユニット.

**Network-on-Chip（NoC）** メニーコアプロセッサの構成要素を接続するネットワーク. ルータとルータ間を接続する物理チャネルで構成される.

## 2.2 Network-on-Chip の概説

Network-on-Chip（NoC）はメニーコアプロセッサなどのチップに搭載される要素をルータを介して接続するネットワークである [9]. 2 コアや 4 コアのマルチコアプロセッサなどチップに搭載される要素が少ない時は，通信路としてシンプルなバスが用いられている [7, 8] が，数十から数百のコアを搭載するメニーコアプロセッサでは，通信性能の問題からバスではなくルータを介したネットワークが用いられる [10]. Network-on-Chip（NoC）の構成を決定する要素として大きく次が 4 つが挙げられる [11].

1. トポロジ
2. ルーティング
3. フロー制御
4. ルータアーキテクチャ

トポロジは，ネットワークを構成する通信路の形である．代表的なトポロジとして，リング，2 次元メッシュ，2 次元トーラス，ツリーなどがある．チップ上に同じ要素を複数搭載するタイルアーキテクチャでは，設計のスケーラビリティの点から 2 次元メッシュのトポロジが採用されることが多い．

ルーティングはネットワークを流れるパケットの通信経路の選択方法である．ルーティング方式は，固定型ルーティングと適用型ルーティングの 2 つに分類できる．固定型ルーティングは，送信先が同じであれば必ず同じ経路をパケットが通過するルーティング方式である．代表的な方式は 2 次元メッシュにおける XY 次元順ルーティングで，コアの位置を XY 座標で表したときに，まず X 方向にパケットを転送し，次に Y 方向に転送し，送信先に到達するというルーティング方式である．適用型ルーティングは，隣接ルータの状況などにより転送方向を変える，送信先が同じでもパケットが同じ経路を通過するとは限らない方式である．このルーティングはネットワークのバンド幅を有効に使えるが，デッドロックを起こさないように対策する必要がある．メニーコアプロセッサでは，設計の簡単さから固定型ルーティングが採用されることが多い．

フロー制御は，ルータにおける資源（バッファ）が不足し，後続のパケットを受け入れることが出来ない場合に，パケットを送信しないように隣接ルータに状態を通知する仕組みである．これにより，ネットワークにおいてパケットを落とすことなく適切に送信先に転送することが出来る．NoC における代表的なフロー制御は，Xon/Xoff 方式とクレジット方式の 2 つである．Xon/Xoff 方式は，ルータの資源が枯渇する前にパケットの送信を停止する信号を隣接ルータに送り，パケットが受け入れ可能になるとパケットの送信を再開する信号を隣接ルータに送る方式である．クレジット方式は，毎サイクル利用可能な資源量を隣接ルータに通知する方式である．Xon/Xoff 方式はクレジット方式よりも少ないハードウェア量で制御することができる．また，クレジット方式は Xon/Xoff 方式よりも細かい粒度で制御することができる．メニーコアプロセッサでは，設計の容易さや，ネットワークの性能に与える影響からクレジット方式が採用されることが多い．

本論文は，タイル型のメニーコアアーキテクチャの設計を目的としているため，トポロジは 2 次元メッシュ，ルーティングは XY 次元順ルーティング，フロー制御はクレジット方式という典型的なものをベースに議論を進める．したがって，NoC においてはルータアーキテクチャに注目していく．

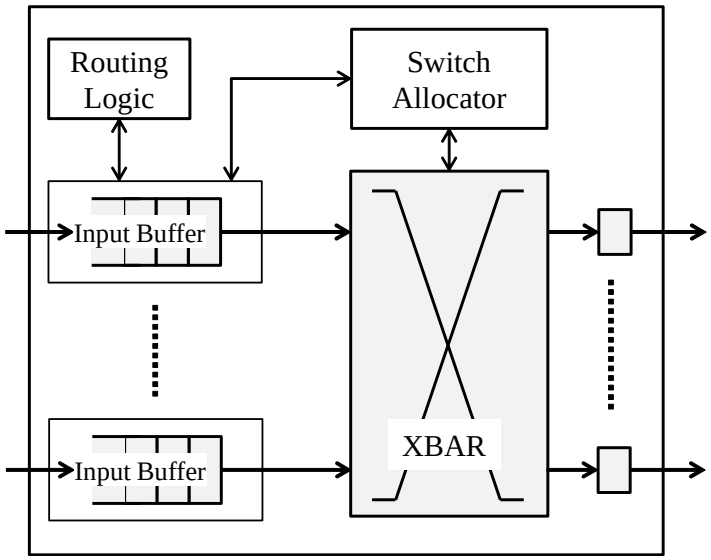


図 2.2 Input-buffered ルータのアーキテクチャ

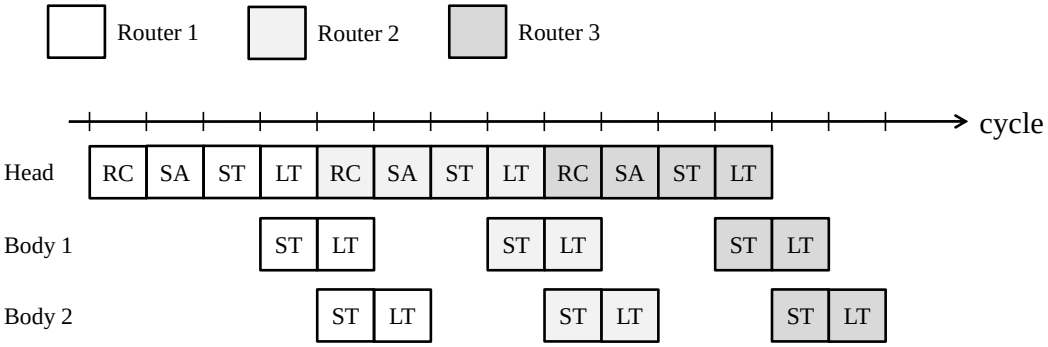


図 2.3 Input-buffered ルータにおけるフリットの転送．ポートの競合など転送のストールが発生しない場合．

### 2.2.1 典型的な Network-on-Chip ルータアーキテクチャ

図 2.2 に典型的な NoC ルータである Input-buffered ルータ [13] のアーキテクチャを示す．入力側にパケットを格納するバッファを持つルータである．2 次元メッシュ接続されたルータにおける入出力ポート数は，隣接する 4 ルータとコアへの合計 5 ポートとなる．入力されたパケットは，入力バッファに格納され，調停の後，クロスバーを通過し，出力側に転送される．

NoC において，パケットは複数のフリットと呼ばれる単位に分割されて送信される．ルータでは，このフリットの単位で転送を制御する．先頭のフリットはヘッドフリットと

呼ばれ，最後尾のフリットはテールフリット，それ以外のフリットはボディフリットと呼ばれる．一般にヘッドフリットには送信先の情報が格納され，ヘッドフリットを含むいくつかのフリットがパケットのヘッダ情報を持つ．そしてテールフリットまでの残りのフリットがペイロードとなる．

図 2.3 に，Input-buffered ルータにおけるフリット転送の様子を示す．ポートの競合などの転送を妨げる要因が一切無い状況における 3 つのルータを通過するフリット転送で，ヘッドフリットを含む連続した 3 フリットを転送する様子を表している．ルータでは，それぞれのフリットはパイプライン的に処理され次のホップに転送される．それぞれのステージにおける処理は以下のようになる．

**Route Computation（RC）**

ルータに入力されたヘッドフリットは，Routing Logic により出力方向が計算される．ヘッドフリットの入力バッファへの格納と同時にこの処理が行われる．Routing Logic は，入力ポートごとに搭載される．

**Switch Allocation（SA）**

Switch Allocation（SA）ステージでは，入力ポートに対する出力ポートの割り当てが行われる．ここで，出力ポートを獲得した入力ポートは，すべてのフリット（1 パケット）を送信するまで出力ポートを確保し続ける．したがって，その入力ポートと出力ポートを接続するクロスバーも同時に確保される．割り当てにはアロケータ [16] が用いられる．割り当てのアルゴリズムは，公平性やスループットに影響を与える．実装のしやすさや，公平性，スループットの性能からラウンドロビンを改良した iSLIP[17] が広く用いられている．

**Switch Traversal（ST）**

Switch Traversal（ST）ステージでは，出力ポートにフリットが転送可能であれば，入力バッファから出力ポートに 1 フリット転送する．フリットはクロスバーを通過し，出力ポートにバッファされる．

**Link Traversal（LT）**

Link Traversal（LT）ステージはフリットを 1 サイクルバッファリングする．メニーコアプロセッサのノードの設計によってルータ間の配線長が長くなる場合に設けられる．

図 2.3 に示す通り，ヘッドフリットは 4 ステージ，その他のフリットは 2 ステージの処理により隣接ルータに送信される．ヘッドフリットにのみ 2 ステージ多く処理が必要になるため，後続のフリットは入力バッファに格納され，先行するフリットの転送を待つ遅延が

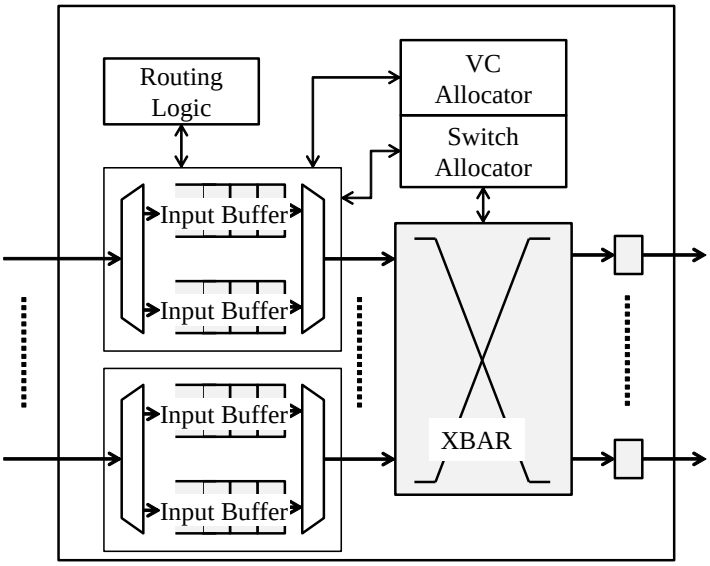


図 2.4 Input-buffered 仮想チャネルルータのアーキテクチャ

発生する．

Input-buffered ルータにおいて，異なる入力バッファに出力ポートが同じパケットがある時，調停により片方のパケットはもう片方のパケットがすべて転送されるまで入力バッファに留まり，転送がストールされる．これは，Head of Line Blocking（HOL Blocking）と呼ばれ，ネットワークの性能を低下させる要因となる．

この問題を解決するために仮想チャネル [18] が用いられる．物理チャネルはいくつかの仮想チャネルを持ち，仮想チャネルが割り当てられている転送であれば，異なるパケットを，一方のチャネルを止めることなく転送することが可能になる．つまり，異なる入力バッファに同じ出力方向のパケットがあるとき，それぞれのパケットが出力ポートの仮想チャネルを獲得していれば，物理チャネルを時分割利用し，それらのフリットを交互に出力側に転送することができるようになる．これにより，HOL Blocking を解消し，ネットワークのスループットを向上させることができる．

図 2.4 に，仮想チャネルを実装した Input-buffered ルータのアーキテクチャを示す．入力バッファは，仮想チャネルごとに分けられている．この入力バッファの実装および各入力ポートからのクロスバへの入力の実装はいくつか選択肢がある [13] が，ここでは，仮想チャネルごとに入力バッファは分けられ，クロスバへの入力は各入力ポートから 1 つという実装を仮定する．入力されたパケットは，バッファに格納され，仮想チャネル割り当て，スイッチ割り当てを経て出力側に転送される．

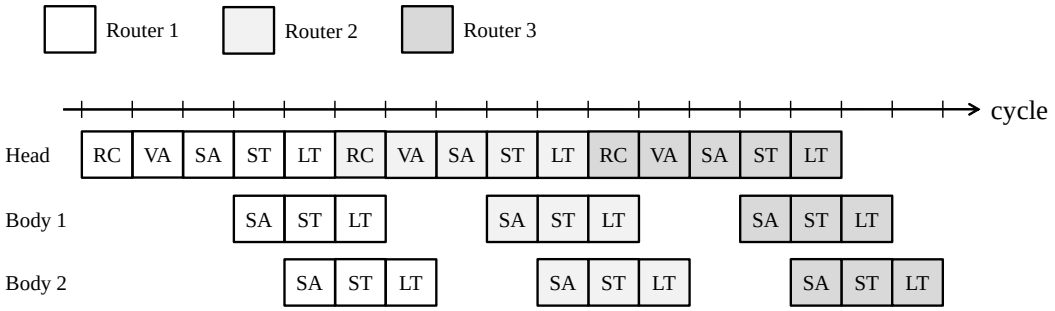


図 2.5 Input-buffered 仮想チャネルルータにおけるフリットの転送．ポートの競合など転送のストールが発生しない場合．

図 2.5 に Input-buffered 仮想チャネルルータにおけるフリット転送の様子を示す．ポートの競合などの転送を妨げる要因が一切無い状況における 3 つのルータを通過するフリット転送で，ヘッドフリットを含む連続した 3 フリットを転送する様子を表している．ルータにおける処理は以下のようなる．仮想チャネルを用いることで，先に述べた仮想チャネル無しのルータにおける処理と異なる部分がある．

**Route Computation（RC）**

仮想チャネル無しのルータと同様である．

**Virtual Channel Allocation（VA）**

Virtual Channel Allocation（VA）ステージで，入力ポートの仮想チャネルに対して出力ポートの仮想チャネルが割り当てられる．仮想チャネル無しのルータの場合は入力ポートに対して出力ポート（物理チャネル）を割り当てていたが，仮想チャネルルータでは入力ポートの仮想チャネルに対して出力ポートの仮想チャネルを割り当てる．

**Switch Allocation（SA）**

Switch Allocation（SA）ステージでクロスバーの調停が行われる．VA が済んでいる入力ポートの仮想チャネルは，同じ出力ポートの仮想チャネルが割り当てられている他の仮想チャネルと時分割でその出力ポート（物理チャネル）を利用することができる．したがって，どの入力ポートの仮想チャネルが出力側の物理チャネルを利用するかをこのステージで調停している．仮想チャネルなしのルータではパケット単位で出力ポートの割り当てを行うが，仮想チャネルルータの場合はフリット単位で割り当てを行う．そのため，この調停は毎サイクル行われる．

**Switch Travers（ST）**

Switch Traversal（ST）ステージで，SA が成功したチャネルは入力バッファのフリッ

トをクロスバーを経由して出力側に送信する．仮想チャネルルータでは，SA ステージにおいてフリットが送信可能かをチェックする．したがって，SA が成功した仮想チャネルの入力バッファから必ずフリットが送信される．クロスバーを通過したフリットは，仮想チャネル無しのルータと同様に出力ポートでバッファされる．

**Link Travers（LT）**

仮想チャネル無しのルータと同様である．

図 2.5 に示す通り，ヘッドフリットは 5 ステージ，その他のフリットは 3 ステージの処理により隣接ルータに送信される．

この Input-buffered 仮想チャネルルータは，NoC において最も一般的な入力バッファ型ルータである．以降では，このルータを **IBR** と呼び，NoC ルータのベースとして議論を進める．

### 2.2.2 Network-on-Chip ルータにおける典型的なレイテンシ削減技術

NoC の性能の尺度には大きくレイテンシとスループットの 2 つがある．メニーコアプロセッサのコア数の増加に伴い，そのネットワークの通信レイテンシがアプリケーションの実行に与える影響が大きくなる．そのため，NoC におけるレイテンシの削減が重要となる．ここでは，先に述べた典型的な NoC ルータに適応される代表的なレイテンシ削減技術について述べる．

ネットワークのレイテンシには，Zero-load Latency という指標がある．これは，そのネットワークにおいて，パケットが衝突せずに転送されたときの平均レイテンシである．

$$T_0 = H_{avg} t_r + T_s \tag{2.1}$$

あるネットワークの Zero-load Latency $T_0$ は，式 (2.1) で表される．$H_{avg}$ はそのネットワークの平均ホップ数，$t_r$ はルータにおける遅延，$T_s$ はパケットの先頭から末尾までの遅延である．サイクル数で考えると，$t_r$ はルータのパイプライン段数，$T_s$ はパケット長となる．特に，ネットワークのトポロジを $N \times N$ の 2 次元メッシュ，ルーティングを XY 次元順ルーティングとし，それぞれのコアが一様に通信（ランダム通信）をすると仮定すると，$H_{avg}$ は，$2N/3$ とすることができる．

例えば，8×8 の 2 次元メッシュにおいてパケット長を 8 フリットとしたとき，パイプラインが 5 段のルータの場合の Zero-load Latency は，

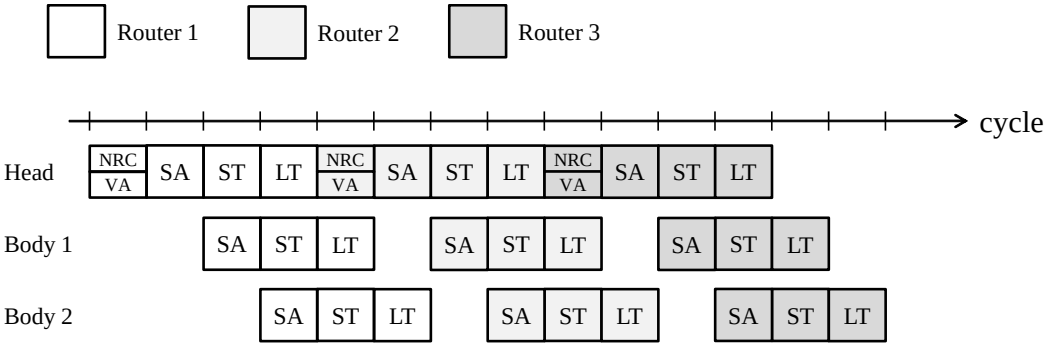


図 2.6 NRC を適用した Input-buffered 仮想チャネルルータにおけるフリット転送

$$T_0 = H_{avg} t_r + T_s = \left(\frac{16}{3}\right) 5 + 8 \approx 34.67$$

より，約 34.67 サイクルとなる．

式 (2.1) より，通信レイテンシを削減する 1 つのアプローチとして NoC ルータのパイプライン削減が有効である．Input-buffered 仮想チャネルルータ（IBR）に適用される典型的なレイテンシ削減技術として，Next Hop Routing Computation（NRC）[11] と Speculative Switch Allocation[19] がある．

NRC は，Look-ahead Routing Computation とも呼ばれ，RC ステージで次ホップのルータにおける出力方向を計算する．パケットにはそのルータにおける出力方向の結果が含まれており，その情報を用いて仮想チャネル割り当てを行う．この処理をあらかじめ行うことで，RC ステージと VA ステージを同時に処理することができ，パイプラインステージを削減できる．しかし，出力方向の計算結果をヘッドフリットに含める必要があり，フリットのビット幅とルータ間の配線量が増加する．

Speculative Switch Allocation は，仮想チャネル割り当て（VA）と同時に投機的にスイッチ割り当て（SA）を行う．投機的な SA に成功すれば VA ステージと SA ステージが同時に処理（VSA）され，パイプラインステージが削減できる．投機的な SA に失敗した場合は，VA ステージと SA ステージは逐次に処理されるためパイプラインステージは削減されず，通常の VA と SA を異なるステージで行う IBR と同じレイテンシとなる．SA ステージでは，すでに仮想チャネルが割り当てられているチャネルからの要求と投機的な SA をおこなうチャネルからの要求の 2 種類の要求があり，すでに仮想チャネルが割り当てられているチャネルからの要求が優先される．

図 2.6 に NRC を適用した IBR におけるフリット転送，図 2.7 に NRC と投機的な Switch Allocation の両方を適用した IBR におけるフリット転送の様子を示す．3 つのルー

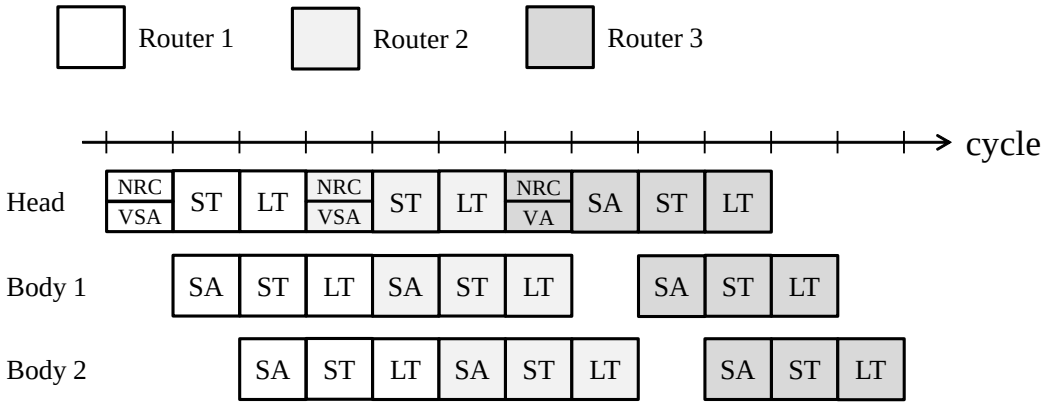


図 2.7　NRC と投機的な Switch Allocation を適用した Input-buffered 仮想チャネルルータにおけるフリット転送．3 つ目のルータにおいて投機的な SA に失敗している．

タを通過するフリット転送で，ヘッドフリットを含む連続した 3 フリットを転送する様子を表している．投機的な SA を行うルータの例では，3 つ目のルータにおいて投機的な SA が失敗した場合のフリット転送の様子を示している．

NRC を採用することで，RC と VA を同時に行うことができ，パイプライン段数を 4 段に削減することができる．さらに，投機的な Switch Allocation を採用することで VA と SA を同時に行うことができ，NRC と併せることでパイプライン段数を 3 段に削減することができる．投機的な SA に失敗した場合は，NRC を採用する 4 段パイプラインのルータと同じ挙動となる．3 段パイプラインでのフリット転送は，ヘッドフリットを含むすべてのフリットについて入力バッファでの遅延を無くすことができ，効率よく通信を行うことができる．

式 (2.1) より，8×8 の 2 次元メッシュ，パケット長 8 フリットのネットワークでは，パイプライン段数 4 の場合の Zero-load Latency は約 29.33 サイクル，パイプライン段数 3 の場合は 24 サイクルとなる．したがって，パイプライン段数 5 の場合に対して，4 段の場合は約 15,4%，3 段の場合は約 30.8% のレイテンシ削減を達成する．

## 2.3　タイル型のメニーコアアーキテクチャ

### 2.3.1　Cell Broadband Engine

Cell Broadband Engine（Cell/B.E.）[20, 21] は，SONY，東芝，IBM によって開発されたメニーコアアーキテクチャである．開発の目的は，ゲーム・マルチメディアアプリケー

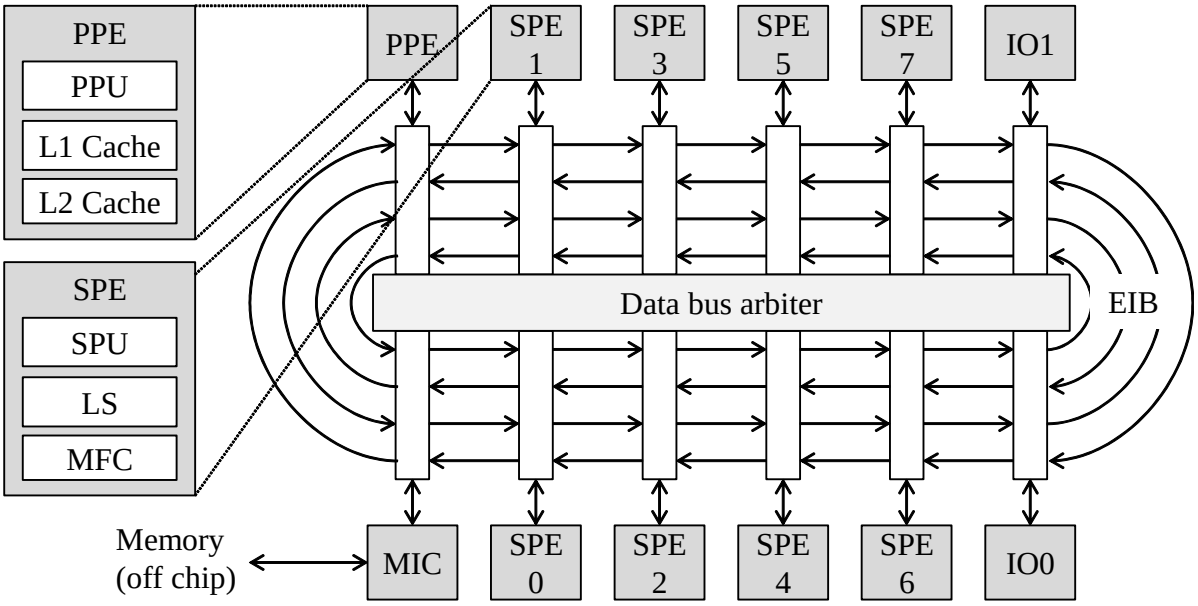


図 2.8　Cell Broadband Engine のアーキテクチャ

ションに対して高い性能を発揮するプロセッサの実現である．さらに，リアルタイム性能や様々なプラットホームで利用されることを目標としている．2000 年頃から開発が始まり，2006 年にゲーム機である PlayStation 3 に搭載され市販された．2006 年当時で，商用では最もコア数の多いプロセッサであった．Cell/B.E. は，IBM のブレードサーバ QS22[22] や，東芝の SpursEngine$^{TM}$[23] と呼ばれるアクセラレータにも応用された．

図 2.8 に Cell/B.E. のアーキテクチャを示す．PowerPC Processor Element（PPE）と呼ばれる 64 ビット PowerPC アーキテクチャに準拠したコアと，Synergistic Processor Element（SPE）と呼ばれる独自のアーキテクチャのコアを複数持ち，それらのコアとメモリコントローラ（MIC）などが Element Interconnect Bus（EIB）と呼ばれるネットワークで接続されている．

設計のコンセプトとして以下の点が挙げられる．

- 低消費電力かつ高性能を達成する回路設計
- PowerPC アーキテクチャによる従来のプログラミング環境の提供
- SIMD 命令によるゲーム，マルチメディア，科学技術計算アプリケーション処理の高性能化
- 高いバンド幅を維持するネットワーク

PPE は，プロセッサに 1 個搭載されている．PowerPC Processor Unit（PPU）と呼ばれ

る PowerPC アーキテクチャで，Simultaneous Multithreading（SMT）技術を用いて 2 スレッドを同時に処理することができるインオーダー実行パイプラインの PE が搭載されている．また，32KB の L1 命令キャッシュおよび L1 データキャッシュ，512KB の L2 データキャッシュを搭載する．OS はこのコアで動作する．

SPE は，プロセッサに 8 個搭載されており，Synergistic Processor Unit（SPU）と呼ばれる SIMD 命令を実行することができるインオーダー実行パイプラインの PE が搭載されている．SPE はそれぞれ独立したメモリ空間を持ち，256KB のローカルメモリ（Local Storage, LS）を利用してアプリケーションを実行する．SPE 間のデータ通信は，Memory Flow Controller（MFC）と呼ばれるハードウェア制御によるローカルメモリ間の DMA 転送により行われる．

EIB は，リング型のトポロジを採用するネットワークである．時計回り，反時計回りそれぞれに 2 本ずつの合計で 4 本の物理チャネルがあり，各チャネルのデータ幅は 16 バイトである．各コアへの接続にスイッチが設けられている．このネットワークはパケットスイッチング方式ではなく，あらかじめ経路を確保してからデータを送信するサーキットスイッチング方式を採用している．そのために，スイッチ確保の要求を処理するアービタ（Data bus arbiter）があり，全コアからの要求を一元処理している．

### 2.3.2 Raw Microprocessor

Raw Microprocessor[24, 25] は，マサチューセッツ工科大学で 1995 年頃から開発されたアーキテクチャである．このプロセッサは，並列アプリケーションに対してスケーリングするように設計されている．また，このアーキテクチャは Tilera 社の TILE64[26] や TILE-GX[27] に応用されている．

図 2.9 に Raw Microprocessor のアーキテクチャを示す．Computational Resource と呼ばれる MIPS ライクなコアが 16 個，4×4 のタイル状に搭載されている．これらのコアが，Static Network および Dynamic Network という 2 種類のネットワークで接続されている．ルータ間の配線遅延が 1 サイクルになるように 1 つのコアのハードウェア規模が設計されており，100 コア，1000 コアのオーダーまでコア数を増加させることを想定している．

Computational Resource は，MIPS ライクな命令セットを実行するインオーダーの 8 段パイプラインの PE を搭載するコアである．32KB のデータキャッシュと 96KB の命令キャッシュを搭載している．

このプロセッサの特徴はネットワークにある．Static Network と呼ばれるネットワークと Dynamic Network と呼ばれるネットワークが 2 つずつあるルータ間およびルータと

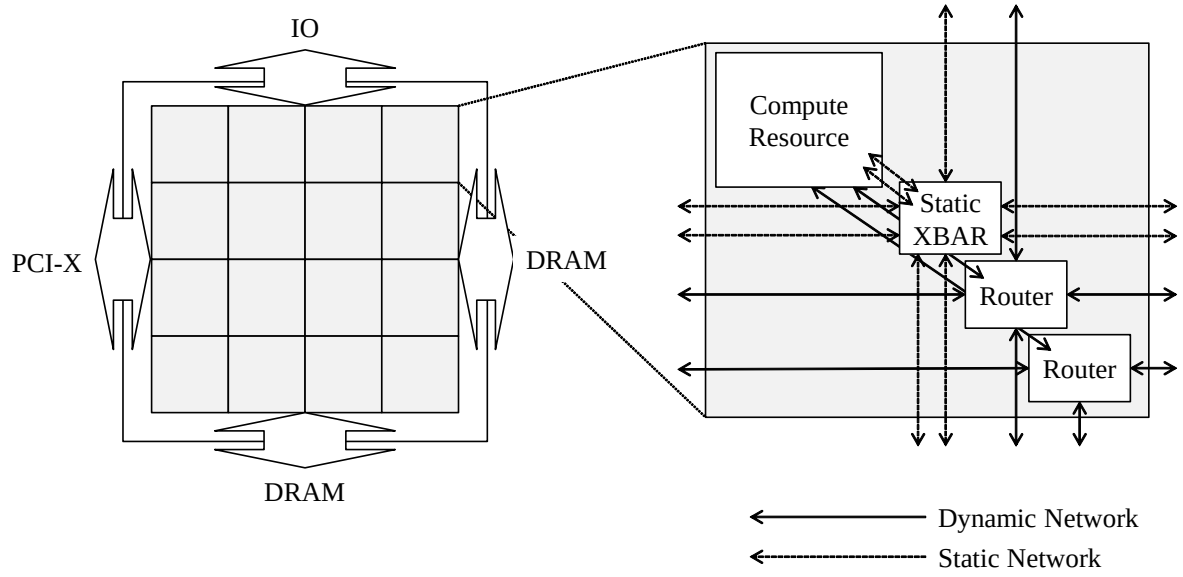


図 2.9　Raw Microprocessor のアーキテクチャ

Computational Resource はそれぞれ合計 4 本の物理チャネルで接続されている．それぞれの物理チャネルは双方向の 32bit 幅のチャネルとなっている．

Static Network は，通信路をあらかじめ固定することができるサーキットスイッチング方式のネットワークで，アプリケーションのコンパイル時に通信を静的に推測することができる通信に用いられる．各方向に接続されている 2 本の物理チャネルは 1 つのクロスバースイッチ（図 2.9 中の Static XBAR）で制御されている．通信路が固定されているために，通信のレイテンシが静的に決まる．

Dynamic Network は，アプリケーションのコンパイル時に通信を静的に推測することができない通信に用いられるパケットスイッチング方式のネットワークである．ルーティング方式は次元順で，パケットは最長で 31 フリットまでとなっている．各方向の 2 本の物理チャネルそれぞれに 1 つのルータが設けられている．したがって，メッシュ接続の物理ネットワークが 2 つあると見なすことができる．ルータはフリットが直進する場合は 1 サイクルで隣接ルータに転送し，フリットが方向を変える場合は 2 サイクルで隣接ルータに転送することができる．

Raw Microprocessor は，1 つのノードに Static Network のためにクロスバースイッチが 1 個，Dynamic Network のためにルータが 2 個搭載されており，ネットワークのためのハードウェア規模はコアである Computational Resource に対して大きくなっている．

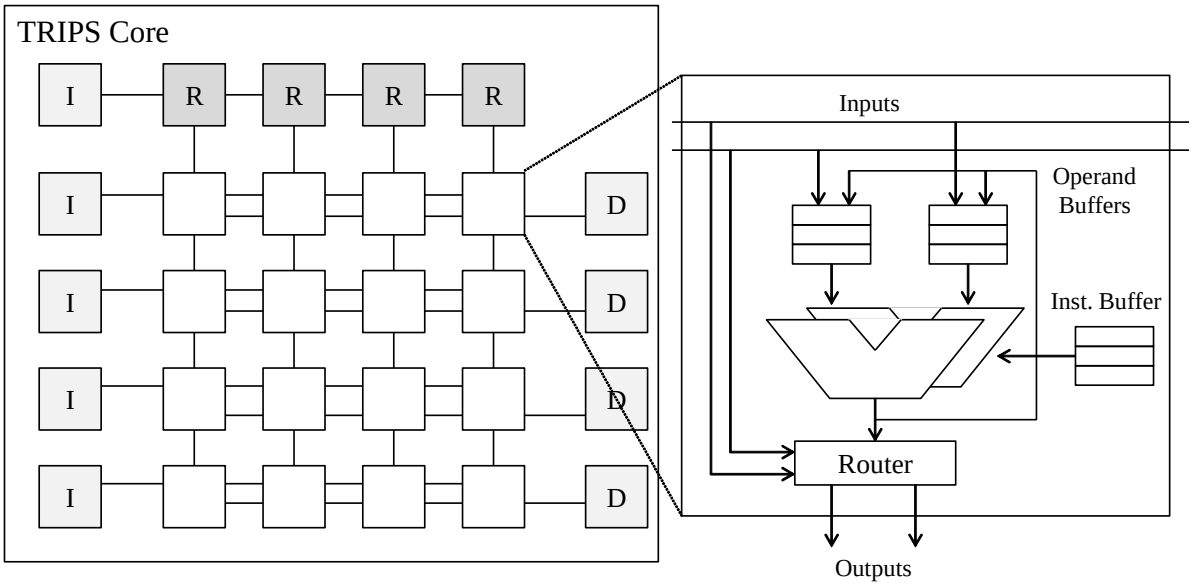


図 2.10　TRIPS Core のアーキテクチャ

### 2.3.3　TRIPS

TRIPS[28, 29] は，2003 年にテキサス大学から発表されたタイルアーキテクチャのプロセッサである．命令レベル並列性，スレッドレベル並列性，データレベル並列性の抽出を達成するために，演算能力の高いコアを用いている．

TRIPS は，TRIPS Core と呼ばれる演算能力の高いコアと，メモリを複数搭載している．これらは，Point-to-Point に互いに接続されている．TRIPS Core は，複数の演算器が 2 次元メッシュのネットワークで接続されたコアになっている．

図 2.10 に，TRIPS Core のアーキテクチャを示す．TRIPS Core は，16 個の演算器を 4×4 のタイル状に並べたコアである．演算器を並べたタイルの上辺にレジスタ・ファイル，左辺に命令キャッシュ，右辺にデータキャッシュを並べた構造となっている．命令キャッシュとデータキャッシュは複数のバンクに分けられ，それぞれ 16KB のキャッシュとなっている．このコアは，命令発行幅 16Way のアウトオブオーダー実行の PE に相当する．

### 2.3.4　Single-chip Cloud Computer

Single-chip Cloud Computer[30, 31]（SCC）は，2010 年に Intel 社が発表したメニーコアプロセッサおよび並列プログラミングの研究を目的としたプロセッサである．同社が 2007

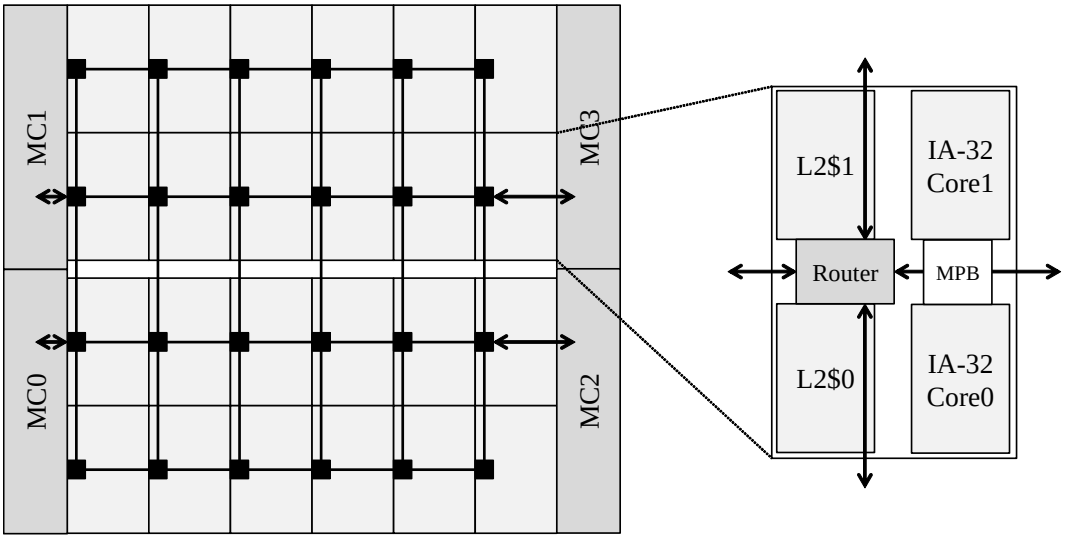


図 2.11　Single-chip Cloud Computer のアーキテクチャ

年に発表した TeraFLOPS Processor[32, 33] をベースに，浮動小数点演算器をタイル状に並べていたものを，OS の動作するコアをタイル状に並べるように変更し，より高性能な NoC を用いている．

図 2.11 に SCC のアーキテクチャを示す．タイルと呼ばれるノードが 6×4 の 2 次元メッシュ状に接続されている．1 つのノードにはコアが 2 個搭載されており，プロセッサに搭載されるコア数は合計で 48 となる．プロセッサの両端にメモリコントローラが搭載されている．

コアは，IA-32 命令セットを実行する Pentium[34] 相当の処理能力を持っている．また，コアには 16KB の命令キャッシュおよび L1 データキャッシュ，256KB の L2 データキャッシュがあり，これらはプライベートキャッシュとなっている．ノードにはコア間通信のための Message Passing Buffer（MPB）と呼ばれる 16KB メモリが搭載されている．このメモリはノード内の 2 つのコアで共有して使う．

NoC のトポロジは 2 次元メッシュで，各ノードを 6×4 の形に接続している．ルーティングは，固定型ルーティングを採用している．ルータ間の物理チャネルの幅は，16 バイトである．仮想チャネルは 8 本あり，2 本はデッドロックが発生しないように特定のプロトコルが使用し，残りの 6 本はスループット向上のために自由に使えるチャネルになっている．ルータの入力バッファは 24 フリット分あり，ポートの競合などの原因でパケットがルータ内に留まった時に 1 パケット分のフリットがバッファ内に収まるようになっている．

ルータは，Next Hop Routing Computation を採用する 4 段パイプライン構成である．入

力バッファに 1 パケットが収まるため，物理チャネルを複数の仮想チャネルで時分割に利用せずにパケット単位で占有するアーキテクチャになっている．

### 2.3.5　その他のメニーコアアーキテクチャ

その他にもいくつかのメニーコアアーキテクチャがある．Intel より 2008 年に発表された Larrabee[35, 36] は，画像処理を高速に行うことを目的としたメニーコアアーキテクチャで，複数の x86 命令セットを実行するインオーダーパイプラインのコアがリング型のトポロジのネットワークで接続されている．プライベートな L1 キャッシュと共有の L2 キャッシュを搭載するアーキテクチャである．同社は，このアーキテクチャを応用し，2010 年に Many Integrated Core Architecture[37]（MIC Architecture）を発表している．

AsAP[38, 39] は，2006 年にカリフォルニア大学デービス校より発表されたメニーコアアーキテクチャで，マルチメディア処理のための DSP である．36 個の演算器を 6×6 の 2 次元メッシュ状に接続している．1 つの演算器は隣接する 2 つの演算器よりデータを受け取り，隣接する 1 つの演算器に計算結果を送るアーキテクチャとなっている．

Heracles[40, 41] は，2011 年にマサチューセッツ工科大学より発表された研究基盤として利用することを目的とした，FPGA 実装できるオープンソースなメニーコアアーキテクチャである．MIPS 命令セットを実行するインオーダーパイプラインのコアが 2 次元メッシュの NoC で接続されている．各コアには，ローカルメモリとプライベートな L1 命令キャッシュ，データキャッシュが搭載されている．ルータは Input-buffered 仮想チャネルルータで，Link Traversal（LT）を設けない 4 段パイプラインのルータとなっている．

## 2.4　レイテンシ削減を目的とする先端 Network-on-Chip ルータアーキテクチャ

### 2.4.1　Prediction Router

Prediction Router[42] は，RC の処理を投機的に行うことでパイプラインステージの削減を達成する NoC ルータである．投機的に行われた RC の結果を利用して VA と SA も投機的に行いヘッドフリットに処理にかかるレイテンシを削減する．RC の処理は，パケットが入力される前に予測に基づき行う．予測した RC が成功した場合はパイプラインが削減され，失敗した場合は従来のルータと同じパイプライン処理を行う．したがって，従来のルータと比較して，パケットの出力方向の予測失敗によるレイテンシへのペナルティーは

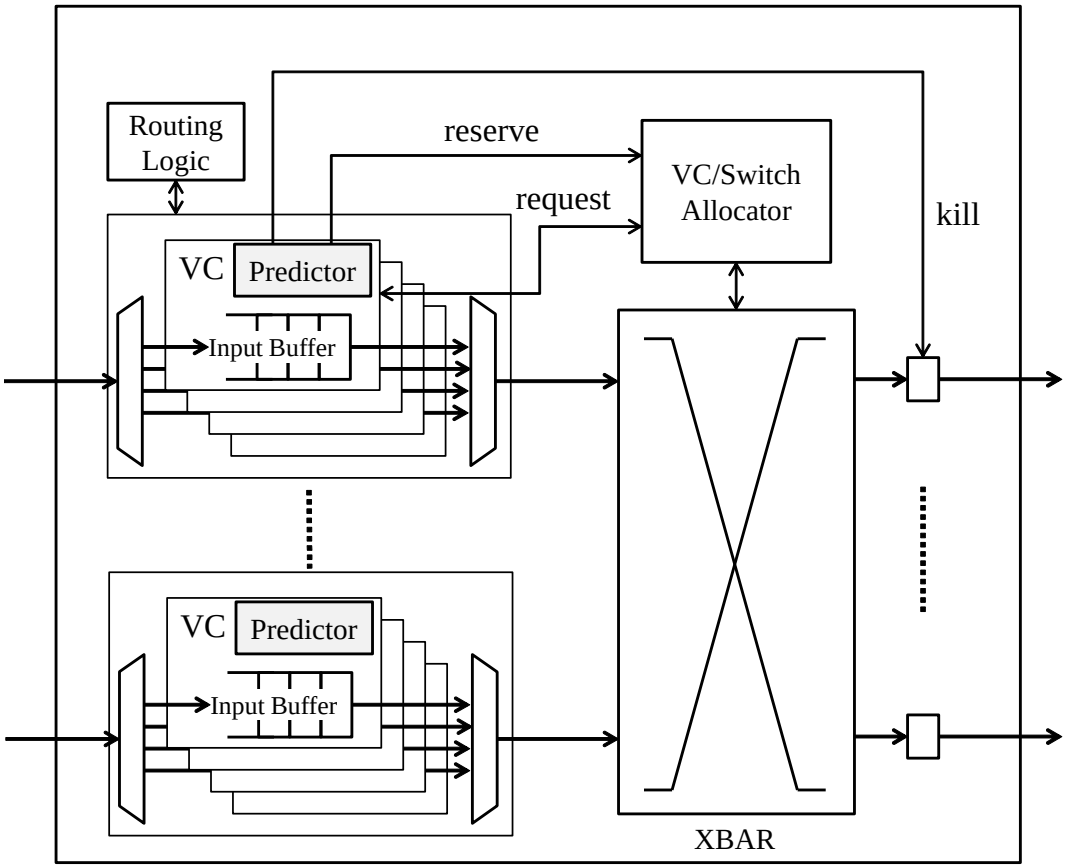


図 2.12　Prediction Router のアーキテクチャ

ない．

図 2.12 に Prediction Router のアーキテクチャ，図 2.13 に Prediction Router がベースとする IBR のフリット転送，図 2.14 に Prediction Router のフリット転送を示す．3 つのルータを通過するフリット転送で，ヘッドフリットを含む連続した 3 フリットを転送する様子を表している．ベースとする IBR は，NRC を用いず，投機的な SA を行うルータである．Prediction Router のフリット転送では，3 つ目のルータで予測が失敗した場合の挙動を示している．

Prediction Router の入力側の仮想チャネルには Predictor が設けられ，予測に基づきパケットが入力される前に出力側の仮想チャネルと出力ポートを確保する．この予測に基づく出力側の仮想チャネルと出力ポートの確保は**仮予約**と呼ぶ．仮予約されたチャネルやポートが他のチャネルにより実際に確保（予測による確保ではない）される時は，仮予約状態が解消される．予測成功によるフリット転送は，RC と VSA の処理が削減され，1 サイクルで処理される．

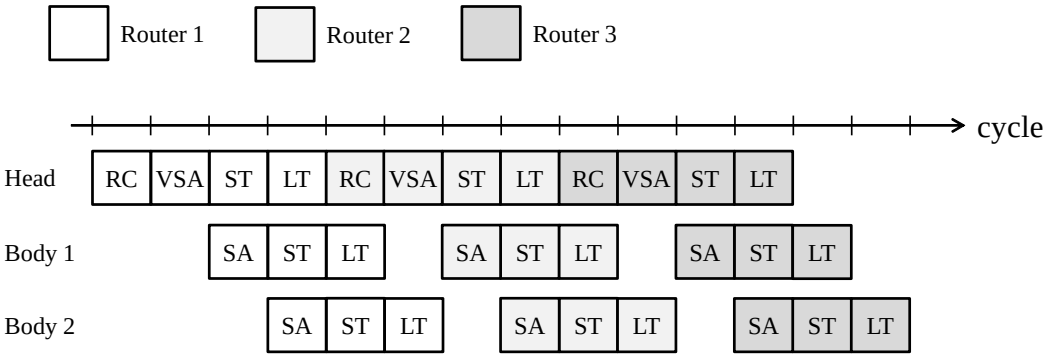


図 2.13　Prediction Router のベースとなる IBR におけるフリット転送

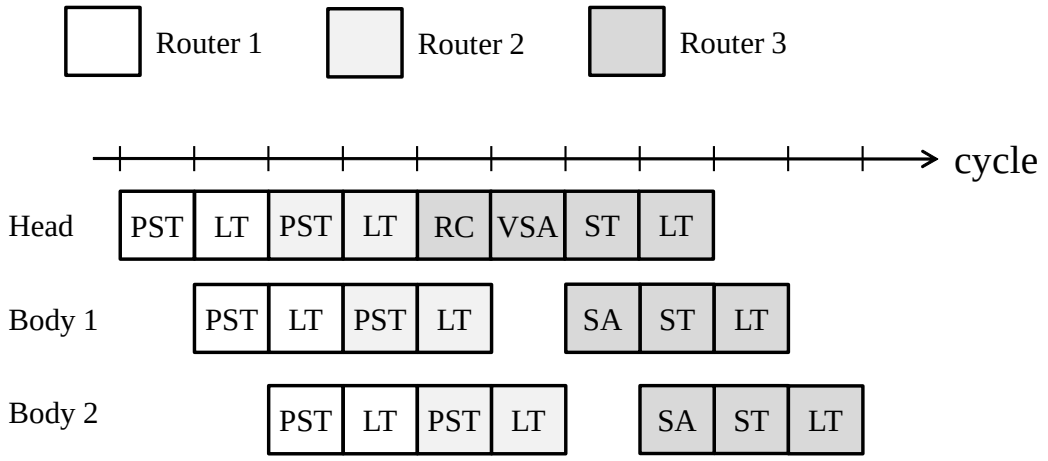


図 2.14　Prediction Router におけるフリット転送

出力側の仮想チャネルと出力ポートが仮予約により確保されている時にそのチャネルにフリットの入力があった場合，仮予約による VA と SA が済んでいるため ST の処理が行われ，入力されたフリットは予測に基づく出力ポートに送信される．この予測に基づくフリット送信を Predictive Switch Traversal（PST）と呼ぶ．PST と同時に RC が行われ，予測の結果がチェックされる．

予測の結果が正しかった場合，仮予約状態のチャネルとポートを実際に確保する状態に変更し，後続のフリットを送信する．これにより，各フリットは 2 サイクルで次のホップに送信される．

予測の結果が間違っていた場合，次のサイクルで予測した出力ポートにフリットを無効化する信号を送信する．予測した出力ポートでは PST で送信されたフリットがバッファされており，無効化の信号によりそのフリットの次ホップへの出力をキャンセルする．これにより，予測による間違ったパケット送信が他のルータに伝搬することはない．また，入

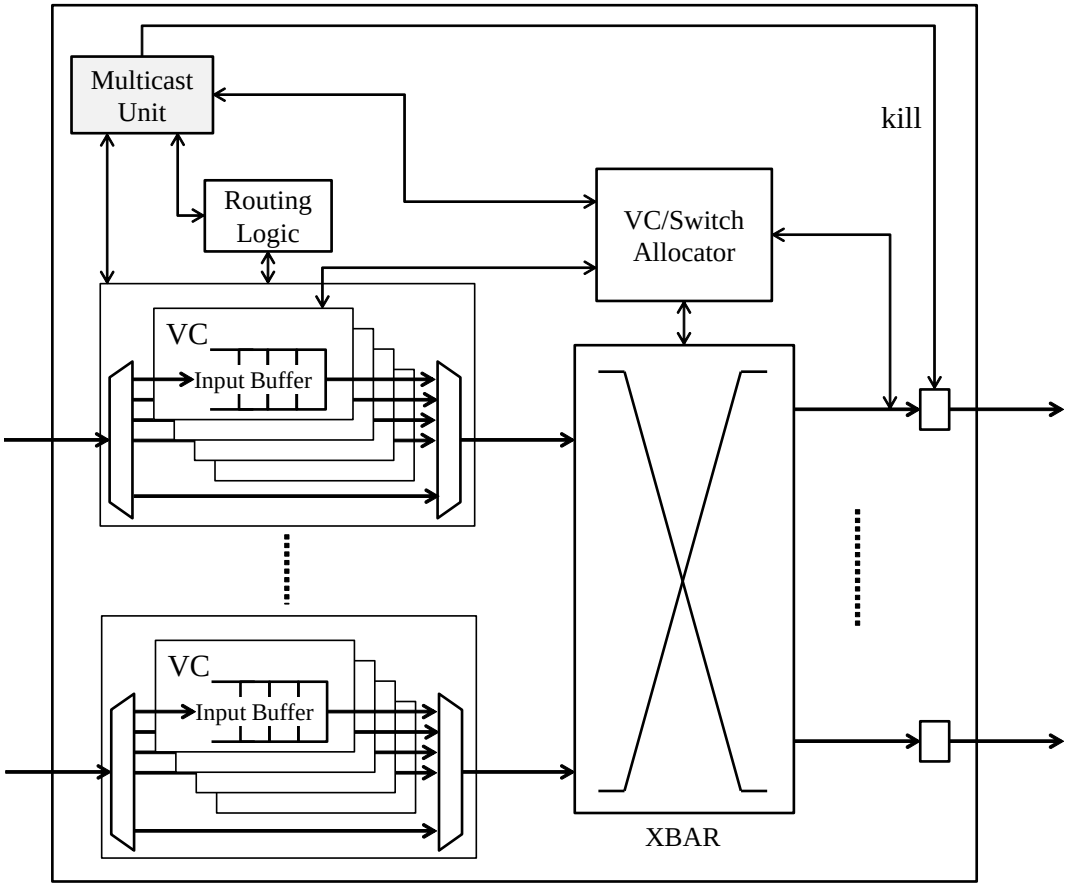


図 2.15　McRouter のアーキテクチャ

力側の仮想チャネルは従来の IBR と同じ処理によるパケット転送が行われ，ベースとしている 4 サイクルでの転送となる．

Prediction Router は，実装する予測アルゴリズムによるが，ベースとする 4 段パイプラインの IBR と比較して高々 15% 程度のハードウェア量の増加で実装することができる．ネットワークの評価に用いられるランダム通信などの合成トラフィックによる評価では，どの予測アルゴリズムを用いても 80% 前後の予測成功率を達成する．しかし，実アプリケーションによる評価では，アルゴリズムと相性のよいアプリケーションでは 90% を超える予測成功率を達成するが，多くの場合の予測成功率は 60% 前後となる．

### 2.4.2　McRouter

McRouter[43] は，Prediction Router よりも積極的に投機的な処理を行い，Prediction Router と同じレイテンシでのフリット転送を達成する NoC ルータである．

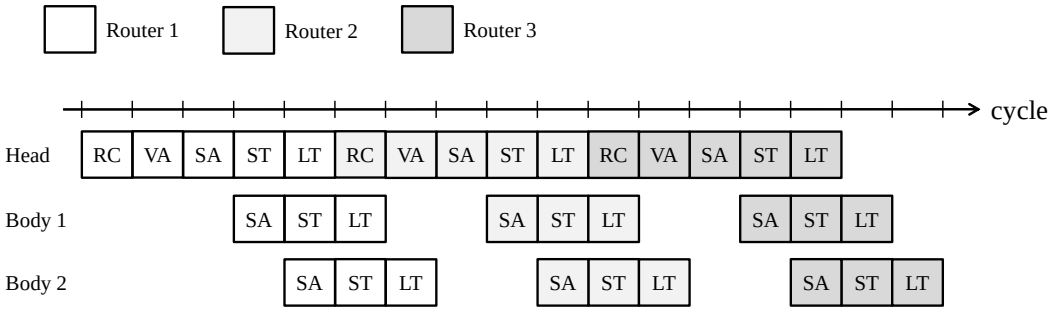


図 2.16　McRouter のベースとなる IBR のパイプライン

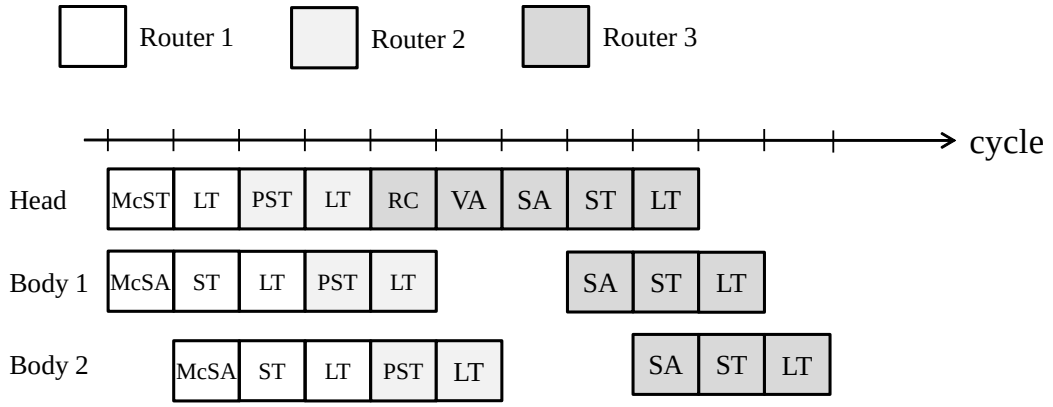


図 2.17　McRouter のパイプライン

Prediction Router では，あらかじめパケットの出力方向を予測して出力ポートの資源を確保していた．McRouter では，フリットが入力された時に可能であれば，その入力ポートから出力可能な方向すべてにフリットをマルチキャストする．マルチキャストは従来のフリット転送を妨げることなく行われ，従来のルータと比較してレイテンシへのペナルティーはない．

図 2.15 に McRouter のアーキテクチャ，図 2.17 に McRouter のベースとなる IBR のフリット転送，図 2.17 に McRouter のフリット転送を示す．ベースとする IBR は，NRC と投機的な SA を用いない 5 段パイプラインのルータである．3 つのルータを通過するフリット転送で，ヘッドフリットを含む連続した 3 フリットを転送する様子を表している．McRouter のフリット転送では，3 つ目のルータでマルチキャストが失敗した場合の挙動を示している．

McRouter の入力ポートには，Multicast Unit が設けられ，入力されたフリットのマルチキャストを制御する．マルチキャストと同時に処理される RC の結果により正しくない出力方向のフリットを無効化する．無効化の方法は Prediction Router と同様に，次のサイク

ルにおいて出力ポートでバッファされているフリットの出力を無効化する．マルチキャストによる転送が行える場合は 2 サイクルでのフリット転送となり，マルチキャストが行えない場合はベースとする IBR と同じ 5 サイクルでのフリット転送となる．実アプリケーション実行時におけるルータのポート使用率が低いことに着目し，予測ではなくマルチキャストを用いることで，低負荷な状態のネットワークで Prediction Router よりも多くレイテンシの削減を実現するルータである．

Multicast Unit は，クロスバーの確保状況を監視し，マルチキャスト可能かどうかを判断する．そのサイクルにおいて通常の SA によりクロスバーが確保されていない場合，マルチキャストが可能である．その状況でヘッドフリットの入力があったとき，フリットをバッファに格納せずに直接クロスバーに送信する（McST）．このときに同時に RC と VA の処理を行う．RC の結果により，正しくない方向にマルチキャストされたフリットは次のサイクルにおいて出力ポートのバッファから出力される際に無効化される．これにより，間違ったパケットが次ホップのルータに伝搬することはない．また，同時に処理される VA はマルチキャストするすべての方向の仮想チャネルを要求する．VA の結果は，マルチキャストされたフリットがクロスバーから出力されるタイミングでフリットに追加される．正しい出力方向の仮想チャネルが獲得できなかった場合は，マルチキャストによる転送がキャンセルされる．マルチキャストが可能で，正しい出力方向の仮想チャネルが獲得できた場合は 2 サイクルでフリット転送がされ，そうでない場合は従来の 5 サイクルでのフリット転送となる．

マルチキャストによりヘッドフリットが出力ポートに転送された場合，2 サイクルでのフリット転送を継続するには後続のボディフリットがルータに入力される前に Switch Allocation を行う必要がある．この場合の SA は，通常の SA と異なりクロスバーの確保は行わず，ST のタイミングでマルチキャストと同様にクロスバーを制御し，パケットの出力方向にのみフリットを送信する（McSA）．したがって，ヘッドフリットがマルチキャストにより送信され，すべての後続フリットも 1 サイクルで送信される時は，Multicast Unit による制御でフリットを転送する．Multicast Unit によるフリットが送信できない場合は，以降は従来のルータと同じ処理でフリットの転送が行われる．

合成トラフィックを用いた評価では，McRouter は Prediction Router に対して，ネットワーク負荷が低い範囲ではより多くのフリット転送でレイテンシの削減を達成するが，ネットワーク負荷が高い範囲ではマルチキャストが行われず IBR と同程度の性能となる．実アプリケーションを用いた評価では，ほとんどのアプリケーションにおいて性能向上を達成し，IBR に対して平均で 28%，Prediction Router に対して平均で 5% の性能向上となる．

# 第 3 章

# ハードウェアの RTL モデリングのための新しい設計環境の提案

本章では，ハードウェアを効率よく設計，評価するための環境として，新しいハードウェア記述言語である ArchHDL[44, 45, 46, 47, 48] と，ArchHDL で記述したソースコードを Verilog HDL のコードに変換するツール [49] を提案する．

## 3.1 開発の動機

C++ などのオブジェクト指向型の汎用プログラミング言語を用いたハードウェアのモデリングでは，一般にハードウェアのモジュールはクラスとして記述される [50, 51, 52, 53, 15]．しかし，そのような汎用プログラミング言語によるハードウェア記述では，モジュール間の接続の記述が困難であるという問題がある．

図 3.1 に，汎用プログラミング言語では記述しにくいハードウェアの例を示す．モジュール A，モジュール B ともに入力されたワイヤの値をインクリメントしレジスタに格納する，出力としてレジスタの値を出力するというハードウェアである．モジュール A の出力がモジュール B の入力，モジュール B の出力がモジュール A の入力となっている．

図 3.2 に，汎用プログラミング言語で図 3.1 のハードウェアをモジュールごとに処理を分けて記述した例を示す．この記述は，モジュールごとにまとめられ直感的にハードウェアを理解しやすい記述ではあるが，ハードウェアのふるまいとしては間違った記述になっている．1 行目で，ワイヤ *A_in* の値が *B_out* により更新されている．このときの *A_in* は，現サイクルにおけるレジスタ *B_reg* の値である必要があり，その値に関する代入は，8 行目において行われている．

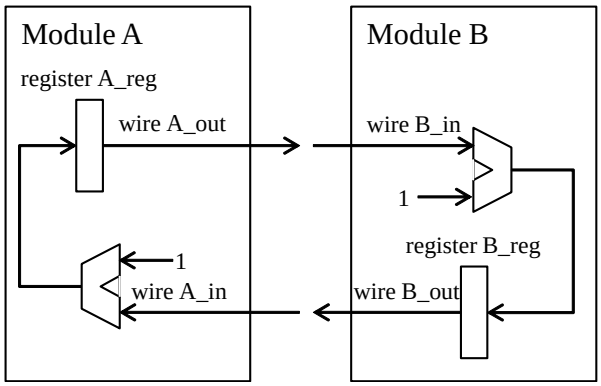


図 3.1 汎用プログラミング言語では記述しにくいハードウェアモジュールの例．それぞれモジュールが，互いに出力を参照し合っている．

```
1 // Module A
2 A_in = B_out;
3 A_out = A_reg;
4 A_reg = A_in + 1;
5
6 // Module B
7 B_in = A_out;
8 B_out = B_reg;
9 B_reg = B_in + 1;
```

図 3.2 モジュールごとに処理をまとめた理解しやすい記述ではあるが，ハードウェアのふるまいとしては間違っている記述例

```
 1 A_out = A_reg;
 2 B_out = B_reg;
 3
 4 // Module A
 5 A_in = B_out;
 6 A_reg = A_in + 1;
 7
 8 // Module B
 9 B_in = A_out;
10 B_reg = B_in + 1;
```

図 3.3 ハードウェアのふるまいを正しくするために代入の順序を考慮した記述例

このハードウェアのふるまいを正しく記述した例を図 3.3 に挙げる．この記述では，1 行目，2 行目においてモジュール A，モジュール B の現サイクルのレジスタの値をあらかじめワイヤの変数に代入している．これにより，それぞれのモジュールからの出力ワイヤへの参照において，現サイクルの想定する値を取得することができ，正しいハードウェアのふるまいの記述となる．

メニーコアプロセッサをオブジェクト指向型の汎用プログラミング言語を用いてモデリングをするとき，メニーコアのノードをクラスとして定義してノード間を互いに接続すると図 3.1 に示したような問題が発生する．

図 3.4 に，Cell/B.E. をモデルとするシミュレータ SimCell[50, 51, 52] のメイン関数を示す．4 行目で Cell/B.E. のコアである SPE のインスタンス，7 行目でネットワークである

```
 1 int main(int argc, char **argv)
 2 {
 3   Chip *chip = new Chip();
 4   Spe *spe[MAX_SPE_NUM];
 5   new MainMemory(chip);
 6   new Mmu(chip);
 7   Eib *eib(chip);
 8
 9   for (int i = 0; i < MAX_SPE_NUM; i++)
10     spe[i] = new Spe(chip, i + 1);
11   eib = new Eib(chip);
12
13   initialize(chip, argv);
14
15   while (loopcond(chip) == LOOP_RUN) {
16     chip->cycle++;
17     for (int i = 0; i < chip->spe_num; i++) {
18       spe[i]->spu->step();
19       spe[i]->mfc->step();
20     }
21     eib->step();
22   }
23   finalize(chip);
24   return 0;
25 }
```

図 3.4 Cell/B.E. シミュレータ SimCell のメイン関数

EIB のインスタンスを生成している．シミュレーションを行うメインループは，15 行目からの while ループである．ループ内では，17 行目から 20 行目ですべての SPE の処理を行い，21 行目でネットワークにおける処理を行っている．SimCell におけるネットワークのシミュレーションは，厳密な Cell/B.E. のネットワークのモデルではなく簡素化したモデルとなっている．このモデルにおける DMA 転送は，EIB のインスタンスによる処理でネットワークのバンド幅から計算されたサイクル数の間に送信元 SPE のローカルメモリから送信先のローカルメモリに値をコピーするという設計になっている．図 3.3 の記述例で示した，あらかじめレジスタからの出力をワイヤ代入する記述と似た構造になっている．

このように，汎用プログラミング言語によるメニーコアプロセッサの記述は，簡素化したネットワークでは図 3.4 のような簡潔な記述が可能であるが，Network-on-Chip の挙動を厳密に記述しようとすると値の代入順序に依存関係が生じ，保守性や可読性が損なわ

れる．

Verilog HDL や VHDL などのハードウェア記述言語では，継続的代入[1] やノンブロッキング代入[2] がサポートされており，代入の順序によらないハードウェアの記述が可能となっている．これらの代入文の記述が汎用プログラミング言語において可能になれば，汎用プログラミング言語による高速なシミュレーションや柔軟なシミュレーションを維持しつつ，直感的なハードウェアの記述が可能となる．

## 3.2 ハードウェア記述言語 ArchHDL の提案

### 3.2.1 コンセプト

ArchHDL は，C++ ベースのハードウェア記述言語である．ユーザは，ArchHDL ライブラリを用いて C++ でハードウェアを RTL で記述する．ライブラリは，ハードウェアを記述するための *Module* クラス，*reg* クラス，および *wire* クラスを提供し，またステップ実行によるシミュレーションのための関数を提供する．

このハードウェア記述言語の特徴は，以下の 2 点である．

- 組み合わせ回路を関数を用いて記述する
- レジスタへのノンブロッキング代入がサポートされている

組み合わせ回路を関数で記述するために，C++11 と呼ばれる C++ の ISO 標準で追加されたラムダ式を利用する．また，ノンブロッキング代入は Verilog HDL や VHDL などの

---

[1] Verilog HDL では，ワイヤを定義するために用いられる代入文．*wire a*，*reg b*，*reg c* があり，*wire a* を

```
assign a = b + c;
```

と継続的代入文を用いて記述したとき，*a* は，*b*，*c* の値の変化に追従して常に最新の値を保持するワイヤとなる．

[2] レジスタへの代入文が評価されたときに，プログラマによって指定されたタイミングで左辺の値の更新がなされる代入．Verilog HDL で *reg a*，*reg b* があり，

```
always @(posedge clock) begin
    a <= b;
    b <= a;
end
```

というノンブロッキング代入の記述があったとき，値の代入のタイミングは clock の立ち上がりエッジ（0 から 1 に変わるタイミング）で *b* の値を *a* に，*a* の値を *b* に代入する．代入される値は，指定されたタイミング直前の値で，このコードは *a* と *b* の値のスワップになる．

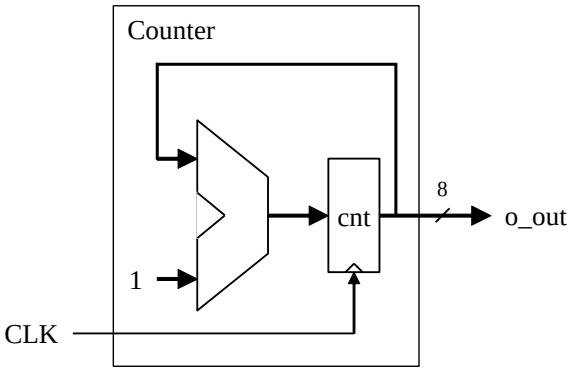


図 3.5　8 ビットカウンタ回路

```
 1  module Counter
 2    (input wire CLK,
 3     output wire [7:0] o_out);
 4
 5      reg [7:0] cnt;
 6      assign o_out = cnt;
 7      always @(posedge CLK) begin
 8          cnt <= cnt + 1;
 9      end
10  endmodule
```

図 3.6　Verilog HDL による 8 ビットカウンタ回路の記述例

```
 1  #include "arch_hdl.h"
 2  #include "arch_hdl_types.h"
 3
 4  class Counter : public Module {
 5   public:
 6    wire<uint_8> o_out;
 7
 8    reg<uint_8> cnt;
 9    void Assign() {
10      o_out = cnt;
11    }
12    void Always() {
13      cnt <<= (cnt() + 1) & 0xff;
14    }
15  };
```

図 3.7　ArchHDL による 8 ビットカウンタ回路の記述例

ハードウェア記述言語ではサポートされているが，一般に汎用プログラミング言語ではサポートされていない．ArchHDL ライブラリにより，C++ でノンブロッキング代入を実現している．

これらの特徴とライブラリにより提供されるクラスを利用することにより，ユーザは ArchHDL で Verilog HDL に似たスタイルでハードウェアを記述することができる．また，ArchHDL で記述したハードウェアは C++ のプログラムとしてシミュレーションを行うことができる．ArchHDL によるシミュレーションは，同じ抽象度で記述した Verilog HDL によるシミュレーションよりも高速である．

### 3.2.2　ArchHDL によるハードウェア記述

図 3.5 に示す 8 ビットカウンタ回路を用いて ArchHDL によるハードウェア記述を説明する．この回路は，*cnt* という 8 ビットのレジスタを持ち，*o_out* からそのレジスタの値を

出力している．*cnt* の値は，入力されるクロック *CLK* に従い毎サイクルインクリメントされる．

図 3.6 に，図 3.5 の 8 ビットカウンタ回路を Verilog HDL で記述した例を示す．*module Counter* は，入力としてクロック *CLK*，出力として 8 ビットのワイヤ *o_out* のポートをもつモジュールとして宣言されている（1 行目から 3 行目）．モジュール内には，8 ビットのレジスタ *cnt* が宣言されている（5 行目）．ワイヤ *o_out* はレジスタ *cnt* の値を出力するワイヤとして assign 文を用いて定義されている（6 行目）．レジスタ *cnt* の値は，always ブロックにおいて記述されているように（7 行目から 9 行目），クロック *CLK* の立ち上がりエッジにおいてインクリメントされる．レジスタ *cnt* は 8 ビットのレジスタとして宣言されているため，インクリメントされた値が 8 ビットを超える場合（0x100 となる場合）は，代入時に上位ビットがカットされ 0 が代入される．

図 3.7 に, 図 3.5 の 8 ビットカウンタ回路を ArchHDL で記述した例を示す．ArchHDL でハードウェアを記述するには，**arch_hdl.h** および **arch_hdl_types.h** というヘッダファイルをインクルードする（1, 2 行目）．この arch_hdl.h が ArchHDL でハードウェアを記述するためのライブラリとなっており，arch_hdl_types.h は後述する ArchHDL から Verilog HDL への変換ツールのためのデータ型を提供するファイルとなっている．ArchHDL では，ライブラリにより提供される *Module* クラス，*reg* クラス，*wire* クラス，*Assign* 関数，*Always* 関数を用いてハードウェアの挙動を RTL で記述する．

ハードウェアモジュールは，ライブラリ内で定義されている *Module* クラスを継承したクラスとして宣言する．図 3.7 では，*Counter* というモジュールを宣言している（4 行目）．Module クラスを継承して定義するクラスは Verilog HDL における Module に相当する．以降の説明では，ユーザが *Module* クラスを継承して定義したハードウェアモジュールであるクラスを *Module* 子クラスと呼ぶ.

ArchHDL には Verilog HDL のようなポートを記述する文法はない．そのため，ハードウェアのポートとして利用するワイヤなどはモジュール内で利用されるワイヤなどと同様に宣言する．また，ArchHDL は明示的にクロックを記述しない．したがって，8 ビットカウンタの記述でポートとして宣言される変数は 8 ビットのワイヤ *o_out* のみとなる（6 行目）．*wire* クラスは，テンプレートクラスとしてライブラリで定義されており，テンプレート引数として”<>” 内にデータ型を記述する．このクラスのインスタンスは，Verilog HDL における *wire* のインスタンスに相当する．

図 3.7 の 6 行目において，ワイヤ *o_out* で *uint_8* というデータ型を用いている．これは，Verilog HDL への変換ツールによる解析を容易にするために定義したデータ型である．*“uint”* か *“int”* で符号の有無を表し，“_” に続く数字がビット幅を表す．したがって，

*uint_8* は符号無しの 8 ビットのデータを表している．これらの型は，arch_hdl_types.h において定義されている．

このワイヤに *o_out* という *“o_”* から始まる変数名を用いている．これは，データ型と同様に変換ツールによる解析を容易にするための命名規則である．ArchHDL では明示的なポートの宣言方法がないため，*“o_”* で始まる変数名のインスタンスを出力ポート，*“i_”* で始まる変数名を入力ポートとしている．この命名規則により，ArchHDL のコードを Verilog HDL に変換する際に変数名の解析のみで適切にポートの宣言とすることができる．

レジスタは *reg* クラスのインスタンスとして宣言される．図 3.7 では，8 行目において 8 ビットのレジスタ *cnt* を宣言している．*reg* クラスはライブラリにおいて *wire* クラスと同様にテンプレートクラスとして宣言されており，テンプレート引数としてデータ型をとるクラスである．このクラスのインスタンスは，Verilog HDL における *reg* のインスタンスに相当する．

組み合わせ回路の定義は *Assign* 関数内において，*wire* クラスのインスタンスに対して関数オブジェクトを代入することで記述する．ArchHDL では，原則としてこの関数オブジェクトを C++11 という C++ の ISO 標準において追加されたラムダ式を用いて記述する．しかし，記述を簡潔にするために *wire* もしくは *reg* クラスのインスタンス 1 つを代入する場合はラムダ式を用いずに記述できるようにライブラリでサポートしている．図 3.7 ではサポートされた簡潔な記述方法を用いてワイヤ *o_out* にレジスタ *cnt* を代入している（10 行目）．この式は，Verilog HDL の assign 文による代入に相当し，図 3.6 における 6 行目の記述に対応する．ラムダ式を用いた組み合わせ回路の記述の詳細は後述する．

レジスタへの値の代入は *Always* 関数内に記述する．ArchHDL では，ライブラリによって *reg* クラスのインスタンスへのノンブロッキング代入がサポートされている．ノンブロッキング代入は”<<=” 演算子を用いて記述する．図 3.7 では，12 行目から 14 行目の *Always* 関数において，*cnt* の値をインクリメントする式ノンブロッキング代入を用いて記述されている．*Always* 関数は Verilog HDL の *always @(posedge CLK)* ブロックに相当する．つまり，デフォルトの ArchHDL ライブラリでは暗黙に 1 つのクロックが定義され，レジスタへの代入タイミングが 1 つ（例えば立ち上がりエッジ）サポートされているということである．

*wire* や *reg* クラスは関数オブジェクトとして実装されており，インスタンスを関数呼び出しすることで値を取得できる．したがって，図 3.7 の 13 行目の右辺では，現サイクルの値を *cnt()* により取得している．

提供されている *uint_8* といったビット幅を明示するデータ型は，単に *unsigned int* 型もしくは *int* 型の別名として実装されており，ビット幅に応じたマスクなどは行われない．そ

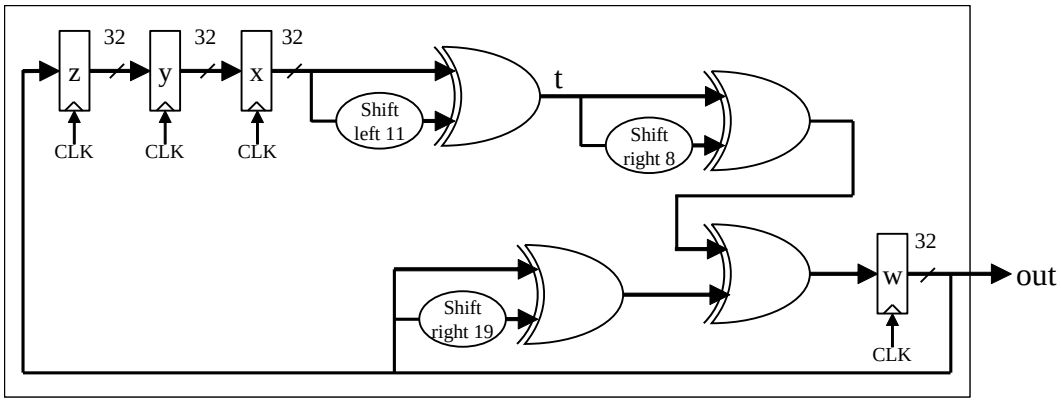


図 3.8　Xorshift アルゴリズムによる疑似乱数生成回路

のため，8 ビットカウンタの実装では，インクリメントしたレジスタの値を 0xff でマスクしている（13 行目）．

### 3.2.3　組み合わせ回路の記述

ArchHDL では組み合わせ回路を関数として記述する．具体的には，ハードウェアモジュールであるクラス内の *Assign* 関数内で *wire* クラスのインスタンスに関数オブジェクトを代入することでその組み合わせ回路を定義する．*wire* クラスのインスタンスへの代入は，次の 3 つがサポートされている．

- *reg* クラスのインスタンス
- *wire* クラスのインスタンス
- ラムダ式により記述された関数

また，ラムダ式による組み合わせ回路の記述には，return 文のみで記述する方法と，いくつかの式を用いて記述する方法の 2 つがある．図 3.7 で示した代入は，*wire* クラスのインスタンスに *reg* クラスのインスタンスを代入する簡潔な記述であった．ここでは，いくつかの記述例を用いて ArchHDL における組み合わせ回路の記述を説明する．

図 3.8 に記述のサンプルとして用いる Xorshift アルゴリズムによる疑似乱数生成回路を示す．この回路は 32 ビットの疑似乱数を生成する回路で，4 つのレジスタを持ち，XOR 演算とシフト演算により乱数を生成する．次に示すハードウェア記述にはレジスタへの初期値の設定やシードの入力などが含まれているが，この図ではそれらを省略している．この回路の記述例を用いて，return 文のみのラムダ式で組み合わせ回路を記述する方法を説明する．

```
 1 module Xorshift
 2   (input wire CLK,
 3    input wire i_rst_x,
 4    input wire i_enable,
 5    input wire [31:0] i_seed,
 6    output wire [31:0] o_out);
 7
 8     reg [31:0] x;
 9     reg [31:0] y;
10     reg [31:0] z;
11     reg [31:0] w;
12     wire [31:0] t;
13
14     assign o_out = w;
15     assign t = x ^ (x << 11);
16
17     always @(posedge CLK) begin
18         if (!i_rst_x) begin
19             x <= 123456789;
20             y <= 362436069;
21             z <= 521288629;
22             w <= 88675123 ^ i_seed;
23         end
24         else begin
25             if (i_enable) begin
26                 x <= y;
27                 y <= z;
28                 z <= w;
29                 w <= (w ^ (w >> 19)) ^ (t ^ (t >> 8));
30             end
31         end
32     end
33 endmodule
```

図 3.9 Verilog HDL による Xorshift アルゴリズムの疑似乱数生成回路の記述例

図 3.9 に Xorshift アルゴリズムによる疑似乱数生成回路を Verilog HDL で記述した例を示す．Xorshift は，入力としてクロック信号 *CLK*，リセット信号 *i_rst_x*，乱数生成回路を駆動させるための信号 *i_enable*，および初期値設定のためのシード *i_seed*，出力として生成された乱数 *o_out* のポートをもつモジュールである．図 3.8 の回路図には示していない，初期値設定のためのリセット信号，シードの入力，回路を駆動させるためのイネーブル信号が追加されている．

図 3.10 に，図 3.9 の Verilog HDL のコードを ArchHDL で記述した例を示す．この記述の 14 行目から 17 行目の *Assign* 関数において，出力用のワイヤ *o_out* と乱数生成の演算で用いられるワイヤ *t* のふるまいが記述されている．15 行目の *o_out* への代入は，ラムダ式を用いない簡潔な記述方法を用いて *reg* クラスのインスタンスを代入している．16 行目の *t* への代入は，return 文のみで記述されるラムダ式[3] が用いられている．

図 3.10，16 行目のラムダ式

[3] C++11 におけるラムダ式の記述は，lambda-introducer と呼ばれる角括弧”[]” の記述から始まる．それ以降は，通常の関数の記述と同様に記述する．lambda-introducer の角括弧内にラムダ式外の変数への参照方法を記述する（例では=となっている）が，ここでは詳細な説明を省く．ArchHDL のハードウェア記述では，[=] のみを利用している．

```
 1 class Xorshift : public Module {
 2  public:
 3   wire<uint_1> i_rst_x;
 4   wire<uint_1> i_enable;
 5   wire<uint_32> i_seed;
 6   wire<uint_32> o_out;
 7
 8   reg<uint_32> x;
 9   reg<uint_32> y;
10   reg<uint_32> z;
11   reg<uint_32> w;
12   wire<uint_32> t;
13
14   void Assign() {
15     o_out = w;
16     t = [=]() { return x() ^ (x() << 11); };
17   }
18   void Always() {
19     if (!i_rst_x()) {
20       x <<= 123456789;
21       y <<= 362436069;
22       z <<= 521288629;
23       w <<= 88675123 ^ i_seed();
24     } else {
25       if (i_enable()) {
26         x <<= y();
27         y <<= z();
28         z <<= w();
29         w <<= (w() ^ (w() >> 19)) ^ (t() ^ (t() >> 8));
30       }
31     }
32   }
33 };
```

図 3.10　ArchHDL による Xorshift アルゴリズムの疑似乱数生成回路の記述例

```
[=]() { return x() ^ (x() << 11); }
```

は，*x* の値と *x* を 11 ビット左シフトした値を XOR した結果を返す関数となっている．ラムダ式内における *x* の値の取得は，レジスタへの値の代入時と同様に関数呼び出しを用いる．この式は，図 3.9 の 15 行目に相当し，Verilog HDL における assign 文で記述された式と同じふるまいとなる．

これまでの例で示したように，Verilog HDL の assign 文を用いた組み合わせ回路の記述は，ArchHDL における組み合わせ回路の記述方法のうち *reg* クラスのインスタンスの代入，*wire* クラスのインスタンスの代入，および return 文のみで記述されるラムダ式の代入を用いることで同様の記述が可能である．一方で，Verilog HDL における組み合わせ回路の記述には function 文もしくは *always @(*)* ブロックを用いた記述方法がある．これを ArchHDL で記述するためには，いくつかの式を用いて記述されるラムダ式を用いる．

図 3.11 に 2 to 4 デコーダを Verilog HDL で記述した例を示す．この回路は，2 ビットの入力 *i_in* の 0 から 3 の値をそれぞれ 4 ビットの 0b0001，0b0010，0b0100，0b1000 の値に変換し *o_out* から出力する回路である．デコーダ部分は，*always @(*)* ブロックを用いて記述している．

```
 1 module DEC
 2   (input wire [1:0] i_in,
 3    output reg [3:0] o_out);
 4
 5     always @(*) begin
 6         out = 0;
 7         case (i_in)
 8             0: o_out = 'b0001;
 9             1: o_out = 'b0010;
10             2: o_out = 'b0100;
11             3: o_out = 'b1000;
12         endcase
13     end
14 endmodule
```

図 3.11 Verilog HDL による 2 to 4 デコーダの記述例

```
 1 class DEC : public Module {
 2  public:
 3   wire<uint_2> i_in;
 4   wire<uint_4> o_out;
 5
 6   void Assign() {
 7     o_out = [=]() {
 8       uint_4 val;
 9       val = 0;
10       switch (i_in()) {
11         case 0: val = 0b0001; break;
12         case 1: val = 0b0010; break;
13         case 2: val = 0b0100; break;
14         case 3: val = 0b1000; break;
15       }
16       return val;
17     };
18   }
19 };
```

図 3.12 ArchHDL による 2 to 4 デコーダの記述例

Verilog HDL において *always @(*)* ブロックを用いて組み合わせ回路を記述する場合，図 3.11 の 3 行目のように変数を *reg* で宣言し，5 行目から 13 行目のように *always @(*)* ブロック内でその変数に対してブロッキング代入により値を代入する記法をとる．

図 3.12 に 2 to 4 デコーダを ArchHDL で記述した例[4] を示す．ArchHDL においてこのような回路を記述する場合，変数は図 3.12 の 4 行目のように *wire* で宣言する．そして，7 行目から 17 行目のように代入するラムダ式内において一時変数を用いて回路のふるまいを記述する．このように，ラムダ式を適切に使うことで，ArchHDL において Verilog HDL と比較して遜色ない組み合わせ回路の記述が可能となる．

### 3.2.4 ArchHDL によるテスト記述

図 3.13 に，ArchHDL で記述した疑似乱数生成器のためのテストコードの例を示す．ライブラリのインクルードなどコードの一部は省略している．このテストコードは，疑似乱数生成器のシードとして 1 を与え，30 サイクルにわたって生成された乱数を表示するものである．

40 行目からのメイン関数は，テストのためのモジュール *TestTop* を生成し，while ループ内でライブラリが提供する *Step* 関数を呼ぶ構造になっている．*TestTop* のインスタンスが生成される時に，すべてのクラスやレジスタのインスタンスがライブラリに登録される．

[4] 図中の 0b から始まる数値は，2 進数リテラルでコンパイラ拡張によりサポートされている．GCC ではバージョン 4.3 以降，Clang ではバージョン 3.2 以降でサポートしている．また，C++ の次期 ISO 標準では言語に組み込まれる予定である．

```
 1 #include "common/xorshift.h"
 2
 3 class TestTop : public Module {
 4  public:
 5   static const unsigned int HALT_CYCLE = 30;
 6
 7   reg<uint_1> HALT;
 8   reg<uint_32> cycle;
 9
10   wire<uint_1> rst_x;
11   wire<uint_32> seed;
12   wire<uint_32> rand;
13   Xorshift xorshift;
14
15   void PortConnect() {
16     xorshift.i_rst_x = rst_x;
17     xorshift.i_enable = rst_x;
18     xorshift.i_seed = seed;
19     rand = xorshift.o_out;
20   }
21   void Assign() {
22     rst_x = [=]() { return (cycle() < 1) ? 0 : 1; };
23     seed = [=]() { return 1; };
24   }
25   void Initial() {
26     HALT = 0;
27     cycle = 0;
28   }
29   void Always() {
30     cycle <<= cycle() + 1;
31     HALT  <<= (cycle() >= HALT_CYCLE);
32
33     if (!rst_x()) {
34     } else {
35       printf("%08x\n", rand());
36     }
37   }
38 };
39
40 int main() {
41   TestTop testtop;
42   do {
43     ArchHDL::Step();
44   } while (!testtop.HALT());
45   return 0;
46 }
```

図 3.13　ArchHDL による Xorshift 疑似乱数生成回路のテスト記述

*Step* 関数は，ライブラリに登録されたすべてのクラスの *Always* 関数を一括で呼び出す関数で，この関数の 1 回の呼び出しが 1 サイクルのシミュレーションとなる．

*TestTop* クラス内で，宣言した *Xorshift* クラスのインスタンスと接続するためのワイヤとして，*rst_x*，*seed*，*rand* を宣言している（10 行目から 12 行目）．*Xorshift* の入出力とそれらワイヤの接続は *PortConnect* 関数内に記述する（15 行目から 20 行目）．モジュールのインスタンスの入出力とモジュール外部のワイヤなどに接続するために *PortConnect* 関数を使用する．その記述は，ArchHDL にはポートの概念がないため，モジュール内に宣言されたワイヤとモジュール外のワイヤを接続する記述となっている．したがって，*PortConnect* 関数は，機能としては *Assign* 関数と同じ関数である．しかし，Verilog HDL への変換ツールによる解析を簡単にするために *Assign* 関数と *PortConnect* 関数を分けて記述している．

シミュレーションのために，*HALT* と *cycle* というレジスタを宣言している（7，8 行目）．

*HALT* はシミュレーション終了のフラグのために使用するレジスタで，*cycle* は実行サイクル数をカウントしている．これらのレジスタは，25 行目から 28 行目の *Initial* 関数内で値を初期化している．この関数は Verilog HDL の *initial* ブロックに相当する．0 で初期化された *cycle* の値は，メイン関数の *Step* 関数が呼ばれる，すなわち *TestTop* のインスタンスの *Always* 関数が呼ばれるたびにインクリメントされる．同様に 0 で初期化された *HLAT* は，定数である *HALT_CYCLE*（5 行目）と *cycle* の値の比較により 1 がセットされる．メイン関数の while ループでは，ループ終了の条件として *HALT* の値を用いており，*HALT* の値が 1 の時にループを終了する．

### 3.2.5　ArchHDL ライブラリの実装

#### ソフトウェアアーキテクチャ

ArchHDL のライブラリには，*Module* クラス，*wire* クラス，*reg* クラス，これらの 3 のクラスのインタフェースクラス，*Singleton* クラスの 7 個のクラスが定義されている．また，ステップ実行によるシミュレーションを行うための *Step* 関数が定義されている．ここでは，標準ライブラリのインクルードをのぞくすべてのライブラリのコードを示しながら ArchHDL の実装について述べる．

図 3.14 に *RegisterInterface* クラス，*ModuleInterface* クラス，*WireInterface* クラス，*Singleton* クラスおよび *Step* 関数の定義を示す．*ModuleInterface* クラス，*WireInterface* クラス，*RegisterInterface* クラスはそれぞれ *Module* クラス，*wire* クラス，*reg* クラスのインタフェースクラスである．*Singleton* クラスは，*Module* 子クラス，*wire* クラス，*reg* クラスのインスタンスをシングルトン・パターンにより一元管理する．このクラスは，ArchHDL のライブラリにおいて核となるクラスである．

*Singleton* クラスは，メンバ変数として *Module* クラス，*wire* クラスのインスタンスへのポインタを格納する可変長配列を持つ（24，25 行目）．*Module* 子クラス，*wire* クラスのインスタンスが生成される際に，そのインスタンスへのポインタが *Singleton* クラスに渡され，配列に追加される．また，値の代入操作がなされた *reg* クラスのインスタンスへのポインタを格納する可変長配列を持つ（27 行目）．これらのポインタは *Singleton* クラスに渡される際にそれぞれのインタフェースクラスに自動でアップキャストされる（34 行目から 42 行目）．

*Step* 関数（72 行目から 80 行目）は，1 サイクルのシミュレーションを行う関数である．*Step* 関数を呼び出すと，*Singleton* クラスの *Exec* 関数が呼ばれる．ただし，初回 *Step* 関数の呼び出しのみ *Singleton* クラスの *Assing* 関数と *Initial* 関数が呼ばれる．*Step* 関数を繰り

```
 1 class ModuleInterface {
 2  public:
 3   virtual void PortConnect() = 0;
 4   virtual void Assign() = 0;
 5   virtual void Initial() = 0;
 6   virtual void Always() = 0;
 7 };
 8
 9 class RegisterInterface {
10  public:
11   virtual void Update() = 0;
12 };
13
14 class WireInterface {
15  public:
16   virtual int Assign() = 0;
17   virtual void Clear() = 0;
18 };
19
20 namespace ArchHDL {
21
22 class Singleton {
23  private:
24   std::vector<ModuleInterface*> modules_;
25   std::vector<WireInterface*> wires_;
26
27   std::vector<RegisterInterface*> update_registers_;
28
29  public:
30   static Singleton& GetInstance(void) {
31     static Singleton singleton;
32     return singleton;
33   }
34   void AddModule(ModuleInterface* mi) {
35     modules_.push_back(mi);
36   }
37   void AddWire(WireInterface* wi) {
38     wires_.push_back(wi);
39   }
40   void AddRegister(RegisterInterface* ri) {
41     update_registers_.push_back(ri);
42   }
43   void Assign() {
44     for (auto module : modules_) {
45       module->PortConnect();
46       module->Assign();
47     }
48     int cond;
49     do {
50       cond = 0;
51       for (auto wire : wires_) {
52         cond += wire->Assign();
53       }
54     } while (cond);
55   }
56   void Initial() {
57     for (auto module : modules_) {
58       module->Initial();
59     }
60   }
61   void Exec() {
62     update_registers_.clear();
63     for (auto module : modules_) {
64       module->Always();
65     }
66     for (auto reg : update_registers_) {
67       reg->Update();
68     }
69   }
70 };
71
72 void Step() {
73   static bool first_step = true;
74   if (first_step) {
75     first_step = false;
76     ArchHDL::Singleton::GetInstance().Assign();
77     ArchHDL::Singleton::GetInstance().Initial();
78   }
79   ArchHDL::Singleton::GetInstance().Exec();
80 }
81
82 }  // namespace ArchHDL
```

図 3.14 ArchHDL ライブラリの各インタフェースクラス，Singleton クラス Step 関数の定義

返し呼び出すことにより，複数サイクルにわたるシミュレーションが行なわれる．

*Assign* 関数（43 行目から 55 行目）は，保持しているすべての *Module* 子クラスのインスタンスの *PortConnect* 関数と *Assign* 関数を呼ぶ（44 行目から 47 行目）．これにより，ユーザが *Module* 子クラス内に宣言したワイヤへの代入操作が行われる．

*Exec* 関数（61 行目から 68 行目）は，保持しているすべての *Module* 子クラスのインスタンスの *Always* 関数を呼び（63 行目から 65 行目），値の代入がなされた *reg* クラスのインスタンスの *Update* 関数を呼ぶ（66 行目から 68 行目）．*Always* 関数においてノンブロッキング代入により値が代入された *reg* クラスのインスタンスは，「<<=」演算子による操作の際に，*Singleton* クラスにそのポインタが渡され，可変長配列に追加される．*reg* クラスのインスタンスへのポインタが格納される可変長配列は，*Exec* 関数のはじめに初期化される（62 行目）．値が変更されたレジスタを管理するこの実装は，*Update* 関数の処理が必要の無いインスタンスの関数呼び出し削減し高速化を実現するための実装で，例えば，メモリなど多数の *reg* クラスのインスタンスを生成した場合にシミュレーションが高速化される．

*Always* 関数によりすべての *reg* クラスのインスタンスについて次のサイクルにおける値が計算される．そして *Update* 関数により *reg* クラスのインスタンスの値が更新される．この *Always* と *Update* の処理によりレジスタのノンブロッキング代入の挙動を実現している．

#### reg クラスの実装

図 3.15 に，*reg* クラスの定義を示す．*reg* クラスは，扱うデータ型をテンプレート引数にとるテンプレートクラスである．また，インタフェースクラスである *RegiterInterface* クラスを継承する．

ArchHDL においてノンブロッキング代入を実現するために，*reg* クラスはメンバ変数にテンプレート引数で与えられたデータ型の変数 *curr_*と *next_*の 2 つの変数を持つ（4，5 行目）．*curr_*は，あるサイクルにおけるレジスタの値で，*next_*はその次のサイクルのレジスタの値である．レジスタの値を取得するには *reg* クラスのインスタンスを関数として呼び出す（25 行目から 27 行目）．

ノンブロッキング代入の実装のために，演算子オーバーロードにより”<<=” 演算子を再定義している（19 行目から 24 行目）．*Module* 子クラスの *Always* 関数において，*reg* クラスのインスタンスに対して”<<=” 演算子による代入を行うと *next_*に値が代入される．このとき，*Singleton* クラスに自身のポインタを渡す（23 行目）．*Singleton* クラスにおいて，ポインタを保持している *reg* クラスのインスタンスの *Update* 関数を呼ぶことで，*next_*の値が *curr_*にコピーされる（12 行目から 14 行目）．

```
 1 template <typename T>
 2 class reg : RegisterInterface {
 3  private:
 4   T curr_;
 5   T next_;
 6
 7   // disallow copy and assign
 8   reg<T>(const reg<T>& rhs);
 9   reg<T>& operator=(const reg<T>& rhs);
10  public:
11   reg() : curr_(0), next_(0) {}
12   void Update() {
13     curr_ = next_;
14   }
15   void operator=(T val) {
16     curr_ =  val;
17     next_ = val;
18   }
19   void operator<<=(T val) {
20     if (val == next_) return;
21
22     next_ = val;
23     ArchHDL::Singleton::GetInstance().AddRegister(this);
24   }
25   T operator()() {
26     return curr_;
27   }
28   std::function<T ()> GetLambda() const {
29     return [=]() { return curr_; };
30   }
31 };
```

図 3.15　ArchHDL ライブラリにおける reg クラスの定義

ライブラリの *Exec* 関数では，すべての *Module* 子クラスのインスタンスの *Always* 関数を呼び出した後に，すべての *reg* クラスのインスタンスの *Update* 関数を呼び出す．したがって，*Always* 関数が呼び出されている間に取得できるレジスタの値 *curr_*は *Update* 関数が呼ばれるまで更新されず保持されている．これによりレジスタへのノンブロッキング代入の挙動を実現する．

テスト記述や初期値設定のために”=” 演算子による値の代入も定義している（15 行目から 18 行目）．“=” 演算子による値の代入は，式が評価された時点で *curr_*と *next_*の両方の値を変更する．

### wire クラスの実装

図 3.16 に，*wire* クラスの定義を示す．*wire* クラスは，テンプレート引数として扱うデータ型をとるテンプレートクラスである．また，インタフェースクラスの *WireInterface* クラスを継承している．

ArchHDL では，組み合わせ回路を関数として記述するため，*wire* クラスはメンバ変数に関数オブジェクト *lambda_*を持つ（4 行目）．この関数オブジェクトは，評価するとテンプレート引数として与えられたデータ型を返す関数オブジェクトである．

組み合わせ回路は，*Module* 子クラスの *Assign* 関数において，*wire* クラスのインスタン

```
 1 template <typename T>
 2 class wire : WireInterface {
 3  private:
 4   std::function<T ()> lambda_;
 5   const wire<T>* prev_;
 6
 7   bool called_;
 8   T val_;
 9
10   // disallow copy and assign
11   wire<T>(const wire<T>& other);
12   //wire<T>& operator=(const wire<T>& rhs);
13  public:
14   wire(): lambda_(nullptr), prev_(nullptr), called_(false) {
15     ArchHDL::Singleton::GetInstance().AddWire(this);
16   }
17   void operator=(std::function<T ()> lambda) {
18     lambda_ = lambda;
19   }
20   void operator=(const wire<T>& rhs) {
21     std::function<T ()> lambda = rhs.GetLambda();
22     if (lambda == nullptr) {
23       prev_ = &rhs;
24     } else {
25       lambda_ = lambda;
26     }
27   }
28   void operator=(const reg<T>& rhs) {
29     lambda_ = rhs.GetLambda();
30   }
31   int Assign() {
32     if (lambda_ == nullptr) {
33       assert(prev_ != nullptr);
34       std::function<T ()> lambda = prev_->GetLambda();
35       if (lambda == nullptr) {
36         return 1;
37       } else {
38         lambda_ = lambda;
39       }
40     }
41     return 0;
42   }
43   T operator()() {
44     return lambda_();
45   }
46   void Clear() {
47     called_ = false;
48   }
49   std::function<T ()> GetLambda() const {
50     return lambda_;
51   }
52 };
```

図 3.16　ArchHDL ライブラリにおける wire クラスの定義

スに *reg* クラスのインスタンス，*wire* クラスのインスタンス，ラムダ式により記述された関数を代入することにより定義することができる．それぞれの代入操作が”=” 演算子のオーバーロードにより 17 行目から 30 行目に定義されている．

*wire* クラスのコンストラクタ（14 行目から 16 行目）では，メンバ変数を初期化し，自身のポインタを *Singleton* クラスに渡す処理が行われる．*wire* クラスのオブジェクトを関数として呼び出すと，自身のメンバである *lambda_*を評価した結果を返す（43 行目から 45 行目）．これにより，*wire* クラスのインスタンスを関数として評価することで，そのサイクルでのワイヤの値が取得できる．

```
 1 class Module : public ModuleInterface {
 2  private:
 3   // copy constructor
 4   Module(const Module& other);
 5   Module& operator=(const Module& rhs);
 6  public:
 7   Module() {
 8     ArchHDL::Singleton::GetInstance().AddModule(this);
 9   }
10   virtual void PortConnect(){}
11   virtual void Assign(){}
12   virtual void Initial(){}
13   virtual void Always(){}
14 };
```

図 3.17 ArchHDL ライブラリにおける Module クラスの定義

### Module クラスの実装

図 3.17 に，*Module* クラスの定義を示す．*Module* クラスは，インタフェースクラスの *ModuleInterface* クラスを継承するクラスである．ArchHDL では，ハードウェアを記述する際に，このクラスを継承してモジュールを記述する．

コンストラクタ（7 行目から 9 行目）では，自身のポインタを *Singleton* クラスに渡す．*ModuleInterface* クラスにおいて *PortConnect* 関数，*Assign* 関数，*Initial* 関数，および *Always* 関数が純粋仮想関数として宣言されているため，これらの空の関数をクラス内に定義している．

### 複数サイクルにわたるシミュレーションの実行

ArchHDL による複数サイクルにわたるシミュレーションの様子を図 3.18 を用いて説明する．シミュレーションに用いる回路は図 3.5 に示した 8 ビットカウンタ回路とする．

複数サイクルにわたるシミュレーションは，ライブラリの提供する *Step* 関数を繰り返し呼び出すことで実現する．

*Step* 関数の初回の呼び出しでは，*Exec* 関数の呼び出しの前にモジュールの *Assign* 関数，*PortConnect* 関数，*Initial* 関数が呼ばれ，ワイヤへのラムダ式などの代入，レジスタへの初期値の設定が行われる．図 3.18 の例では，モジュール *Counter* の *Assign* 関数が呼ばれ，*wire o_out* に *reg cnt* が代入される．

初回以降の *Step* 関数の呼び出しは，繰り返し *Exec* 関数を呼び出す処理となる．*Exec* 関数では，まずすべてのモジュールの *Always* 関数が呼び出される．図の例では，*Counter* クラスの *Always* 関数が呼び出され，*cnt* の現在の値をインクリメントし，0xff でマスクした値が *cnt* にノンブロッキング代入により代入される．次に，ノンブロッキング代入による

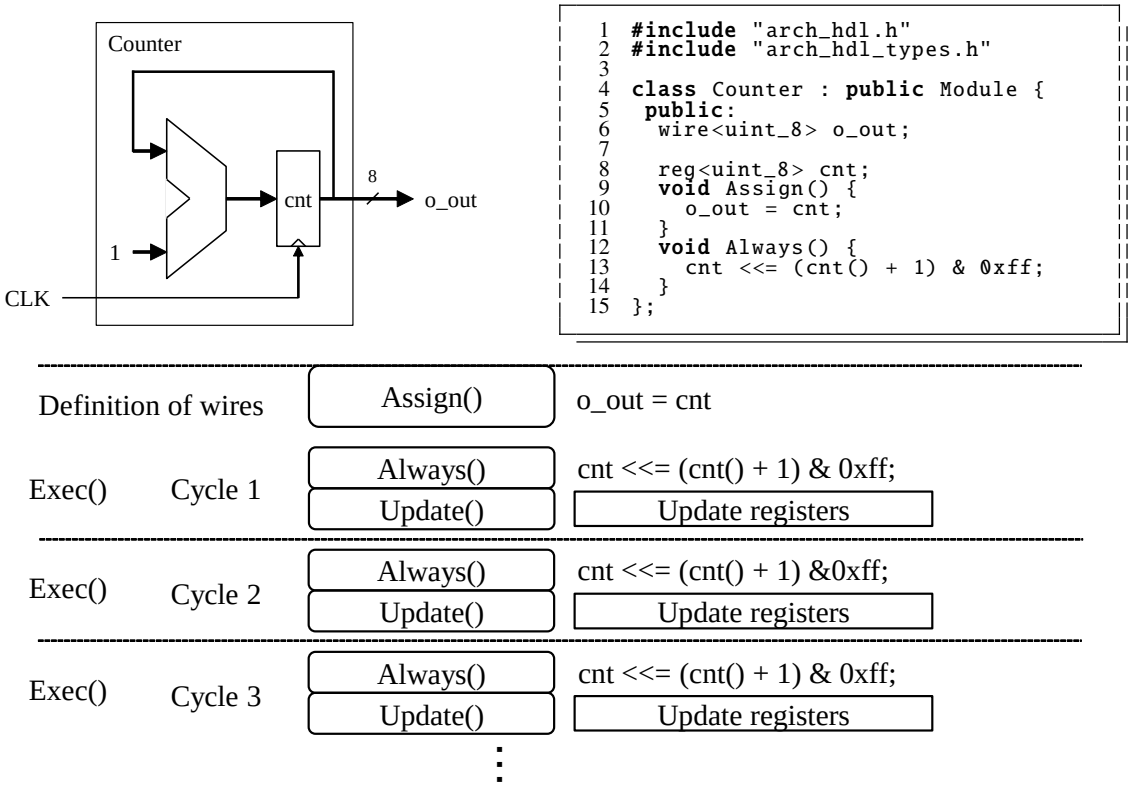


図 3.18　ArchHDL による複数サイクルにわたる 8 ビットカウンタ回路のシミュレーションの様子

値の代入があったすべての *reg* クラスの *Update* 関数が呼ばる．図の例では，*cnt* の *Update* 関数が呼ばれ，値の更新が行われる．以降，*Exec* 関数が呼び出されるたびに，*Always* 関数と *Update* 関数が呼び出され，*cnt* へのノンブロッキング代入と値の更新が繰り返され，複数サイクルにわたるシミュレーションが実現する．

### 3.2.6　Verilog HDL に対する ArchHDL の利点と欠点

ArchHDL によるハードウェア記述の利点として次の 2 点が挙げられる．

1. オブジェクト指向プログラミングによる直感的なモジュール記述
2. C++ 言語の標準ライブラリなどを利用した柔軟なテストベンチの記述

LSI や FPGA のハードウェアリソースは増大傾向にあり，メニーコアプロセッサのように同一のモジュールを複数搭載するハードウェアを記述する場面が増している．ArchHDL

```
 1 #pragma once
 2
 3 #include "arch_hdl.h"
 4 #include "arch_hdl_types.h"
 5
 6 #include "config.h"
 7 #include "flit.h"
 8 #include "nic.h"
 9 #include "router.h"
10
11 class Node : public Module {
12  public:
13   wire<uint_1> i_rst_x;
14   wire<uint_32> i_id;
15   wire<Flit> i_in_flit[NUM_DIRECTION - 1];
16   wire<Flit> o_out_flit[NUM_DIRECTION - 1];
17   wire<uint_1> i_in_flow_control[(NUM_DIRECTION - 1) * NUM_VIRTUAL_CHANNEL];
18   wire<uint_1> o_out_flow_control[(NUM_DIRECTION - 1) * NUM_VIRTUAL_CHANNEL];
19
20   NIC nic;
21   wire<Flit> nic_in_flit;
22   wire<Flit> nic_out_flit;
23   wire<uint_1> nic_in_flow_control[NUM_VIRTUAL_CHANNEL];
24   wire<uint_1> nic_out_flow_control[NUM_VIRTUAL_CHANNEL];
25
26   Router router;
27   wire<Flit> router_in_flit[NUM_DIRECTION];
28   wire<Flit> router_out_flit[NUM_DIRECTION];
29   wire<uint_1> router_in_flow_control[NUM_DIRECTION * NUM_VIRTUAL_CHANNEL];
30   wire<uint_1> router_out_flow_control[NUM_DIRECTION * NUM_VIRTUAL_CHANNEL];
31 };
```

図 3.19　Network-on-Chip ルータのノードの定義の一部

は，モジュールやレジスタ，ワイヤを配列を用いて宣言することができる．これにより，モジュール，ワイヤのインスタンスを配列で宣言し for 文を用いて回路を記述するといったように，複数の同一モジュールを直感的に記述することができる．

図 3.19 に Network-on-Chip ルータ（*Router*）とネットワークインタフェース（*NIC*）を含むモジュール *Node* の定義の一部を示す．*Node* の定義に必要な *PortConnect* 関数，*Assign* 関数, *Always* 関数などは省略している．このノードは, 入力としてリセット信号（*i_rst_x*），ノード ID（*i_id*）, フリットの入力と出力（*i_in_flit*，*o_out_flit*），フロー制御信号の入力と出力（*i_in_flow_control*，*o_out_flow_control*）が宣言されている．

フリットの入力と出力およびフロー制御の入力と出力は配列を用いて宣言している（15 行目から 18 行目）．例えば，このノードが 2 次元メッシュ状に接続される場合，フリットの入出力は 4 方向，フロー制御の入出力信号は 4 方向ごとに仮想チャネル本数分が宣言される．接続方向の数は定数である *NUM_DIRECTION* によりパラメータとして設定されている．このように，ArchHDL では配列と定数を用いることで入出力方向の数の変更などにも容易に対応できる簡潔な記述が可能である．

Verilog HDL では，ポートとして宣言するワイヤなどに配列を用いることができない．そのため，図 3.19 のようなモジュールを記述する場合，例えば各方向からの入力については，入力方向ごとにワイヤをポートとして 1 つずつ宣言するか，入力方向すべてを 1 つの

ワイヤにまとめてビット幅の大きなポートとして宣言する必要がある．前者による記述では，モジュールの汎用性が損なわれ，入力方向の数の変更などに容易に対応できなくなってしまう．また，モジュール内の記述においても同一のコードを複数記述する可能性が生じ，保守性が損なわれる．後者による記述では，ビット幅にパラメータを用いることでそのモジュールの汎用性が損なわれることはない．しかし，例に挙げたノードの場合，それぞれの入力は別のノードからの出力であり，それらをビット連結してノードの入力とする記述は連結の順番などを考慮する必要があり，直感的なハードウェア記述の妨げとなる．

一方，アーキテクチャ設計における検証では様々なパラメータを用いた評価が必要となり，柔軟なテスト記述が求められる．ArchHDL でのテスト記述は，乱数や可変長配列など C++ の標準ライブラリを利用することができるため，典型的なソフトウェアシミュレータと同等の評価が可能である．さらに，このときに同じ抽象度で記述した Verilog HDL によるシミュレーションより，ArchHDL によるシミュレーションが高速である点も有用性がある．

ArchHDL によるハードウェア記述の欠点として次の点が挙げられる．

1. 1 つのクロック信号のみでハードウェアを記述する
2. レジスタへの代入タイミングはその 1 つのクロック信号の立ち上がりエッジのみ
3. データ型として C++ の組み込み型である 32 ビットや 64 ビットの整数を用いる
4. C++ に組み込まれた演算を用いる

ArchHDL ライブラリの実装と後述する Verilog HDL への変換ツールによる解析を容易にするために，ArchHDL ライブラリを利用したハードウェア記述には，Verilog HDL によるハードウェア記述の可用性と比較して，いくつか制限がある．

ArchHDL ライブラリでは，1 つのクロック信号のみでハードウェアを記述し，レジスタへの代入のタイミングも 1 つに制限されているため，複数のクロック信号を用いるハードウェアやクロックの立ち上がりエッジと立ち下がりエッジの両方をレジスタへの代入のタイミングに利用するハードウェアの記述はできない．これは，レジスタへの代入タイミングである *Always* 関数が 1 つしかライブラリで宣言されていないためである．しかし，ライブラリがシンプルなため，ユーザの改変で任意にレジスタへの代入のタイミングを導入することが可能である．

ArchHDL によるハードウェア記述では，データ型として C++ の組み込み型である 32 ビットや 64 ビットの整数を用いる．変換ツールによる解析を容易にするために提供されている型名にビット幅が含まれているデータ型も 32 ビットや 64 ビットの整数型の別名として実装されている．したがって，任意のビット幅の整数はサポートしていない．このた

```
1 void Exec() {
2   update_registers_.clear();
3   for (auto module : modules_) {
4     module->Always();
5   }
6   for (auto reg : update_registers_) {
7     reg->Update();
8   }
9 }
```

図 3.20 並列化前の ArchHDL ライブラリの Exec 関数

```
1  void Exec() {
2    update_registers_[omp_get_thread_num()].clear();
3  #pragma omp for
4    for (size_t i = 0; i < modules_.size(); ++i) {
5      modules_[i]->Always();
6    }
7    for (auto reg : update_registers_[omp_get_thread_num()]) {
8      reg->Update();
9    }
10 }
```

図 3.21 OpenMP による並列化後の ArchHDL ライブラリの Exec 関数

め，特定のビット幅により演算を行いたい場合は，適切に値をマスクしたり，ユーザが独自にデータ型を定義する必要がある．また，演算も C++ に組み込まれた演算を用いるため，ハードウェア記述言語におけるビット切り出し，連結演算などはサポートしていない．これについても，シフト演算やマスクを利用したり，構造体を用いたデータ型を定義することで対応することができる．

このように，ArchHDL によるハードウェア記述には Verilog HDL による記述と比較していくつか制限がある．これはライブラリの実装をシンプルにするためや，シミュレーションを高速にするためであり，必要に応じてユーザがライブラリを拡張することで目的のハードウェアを記述する上で Verilog HDL による記述と遜色ない記述が可能となる．そのため，これらの制限は ArchHDL によるハードウェア記述において大きな妨げにはならないと考えている．

### 3.2.7 ArchHDL ライブラリの並列化による高速化

ArchHDL ライブラリの *Exec* 関数は OpenMP を利用して並列化することができる．*Exec* 関数はシミュレーションにおいて繰り返し呼ばれ，実行の大部分を占める関数である．この関数を並列化することによりシミュレーションの高速化を実現する．

図 3.20 に並列化前の ArchHDL ライブラリの *Exec* 関数，図 3.21 に OpenMP により並列化した *Exec* 関数を示す．並列化した箇所は次の 2 点である．

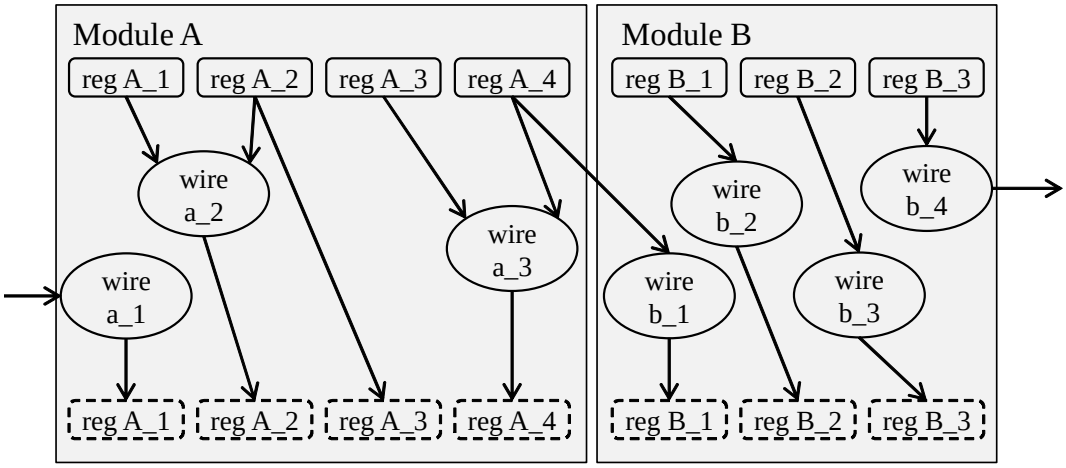


図 3.22　Always 関数の実行の様子

1. *Module* 子クラスの *Always* 関数の呼び出しの並列化
2. *reg* クラスの更新操作の並列化

*Module* 子クラスの *Always* 関数を呼び出しの並列化は，OpenMP の for ループを並列化する指示文により実装している（図 3.21，3 行目から 6 行目）．*reg* クラスの更新操作の並列化は，ノンブロッキング代入により値の更新が必要なレジスタを保持する可変長配列をスレッド数分用意し，更新のための関数呼び出しをスレッドごとに分散している（図 3.21，2 行目および 7 行目から 9 行目）．実際に並列化したコードは，例えば 2 行目の *omp_get_thread_num()* や 4 行目の *modules_.size()* など，細かな関数呼び出しを削減するためにこれらの値を変数に代入するなどの最適化を施している．

図 3.22 を用いて，各 *Module* 子クラスの *Always* 関数が並列に実行される仕組みを説明する．*Module A* にはレジスタが *A_1* から *A_4* までの 4 個とワイヤが *a_1* から *a_3* までの 3 個が定義されている．*Module B* にはレジスタが *B_1* から *B_3* までの 3 個とワイヤが *b_1* から *b_4* までの 4 個が定義されている．図の上部，直線で囲われたレジスタは *reg* クラスの現在のサイクルの値を保持するメンバ変数である *curr_*を参照していることを意味し，図の下部，破線で囲われたレジスタは次のサイクルの値を保持するメンバ変数である *next_*を参照していることを意味している．例えば，*reg A_2* は，ノンブロッキング代入により *wire a_2* の値が代入される．*wire a_2* は *reg A_1* と *reg A_2* の値を参照するワイヤである．したがって，*reg A_1* の *curr_*の値と *reg A_2* の *curr_*の値で何らかの演算を行った結果（*wire a_2*）が *reg A_2* の *next_*に代入されるという操作になる．

*Module A* の *Always* 関数が実行されるとき，*reg A_1* には *wire a_1* を評価した値，*reg*

*A_2* には *wire a_2* を評価した値，*reg A_3* には *reg A_2* の値，*reg A_4* には *wire a_3* を評価した値がそれぞれノンブロッキング代入により代入される．*Module B* の *Always* 関数が実行されるとき，*reg B_1* には *wire b_1* を評価した値，*reg B_2* には *wire b_2* を評価した値，*reg B_3* には *wire b_3* を評価した値がそれぞれノンブロッキング代入により代入される．このとき，すべてのレジスタの値は現在のサイクルにおける値（*curr_*）を参照しており，*Always* 関数の実行によってこれらの値が変更されることはない．したがって，これらの代入操作は，評価順序に関わらず並列に実行可能である．

また，一般にハードウェアモジュールを記述するときに，あるモジュール内に宣言されているレジスタが他のモジュールによって更新されるような記述をすることはない．例えば，*Module B* の *Always* 関数内に *Module A* の *reg A_1* に対する代入文が記述されることはない．したがって，そのように記述された *Module* 子クラスの *Always* 関数の呼び出しは，他の *Module* 子クラスの *Always* 関数の呼び出しに依存関係がなく，並列に呼び出すことが可能である．

レジスタの更新にブロッキング代入が用いられている場合，その値の参照がモジュール内に限定されていれば（例えば *Module A* の *reg A_2* や *reg A_3*），他のモジュールの *Always* 関数の実行に影響がないため並列に実行可能である．そのレジスタの値がモジュールの外から参照されるとき（例えば *Module A* の *reg A_4*），*Always* 関数が呼び出される順番によって実行結果が変わってしまう．したがって，並列化により実行ごとにことなる結果となる可能性があるが，このような記述は実際のハードウェアにおける動作においても値を定めることができないためシミュレーションとして問題は無いと考えられる．

## 3.3 ArchHDL のコードから Verilog HDL のコードへの変換ツール

ArchHDL で記述したハードウェアを Verilog HDL のコードに変換するトランスレータを開発している．記述において，場合により冗長な記述をする必要があるが，ArchHDL によるハードウェア記述の利点を損なうことなく自動で Verilog HDL のコードを生成することが出来る．

これまでのハードウェアの記述例で示したように，ArchHDL によるハードウェアの記述は，*reg* クラスや *wire* クラスなどライブラリによりサポートされているクラスを利用して記述することにより Verilog HDL に似た記述になる．特に，回路の挙動を記述する代入や演算の式は，設計者が Verilog HDL で記述する場合に同じ式を記述するはずである．したがって，ArchHDL から Verilog HDL へのコードの変換は，最適化などを必要とせずに

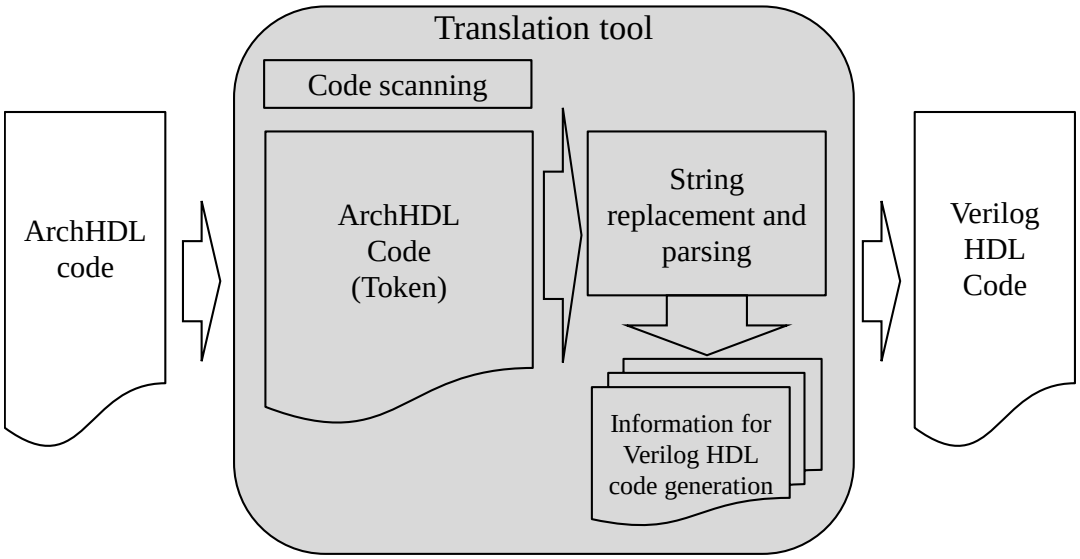


図 3.23 ArchHDL のコードから Verilog HDL のコードへの変換フロー

設計者の意図する Verilog HDL の記述に変換することが出来る．

図 3.23 に，ArchHDL のコードから Verilog HDL のコードに変換する手順を示す．トランスレータは，入力として ArchHDL のコードを受け取り，Verilog HDL のコードを出力する．変換は以下の手順で行われる．

1. ArchHDL のコードのトークンに分解
2. 文字列置換，構文解析による Verilog HDL のコードのための情報生成
3. Verilog HDL のコードの出力

手順 2 における構文解析では，データ型に構造体を利用した場合やインスタンスで配列を使用した場合など Verilog HDL ではそのまま記述できない構文のみを解析している．ArchHDL で記述した式は基本的にはそのまま Verilog HDL でも記述することができる式であるため，詳細な構文解析は行っていない．

### 3.3.1 変換のための ArchHDL における記述の制約

変換ツールで Verilog HDL に変換するための ArchHDL におけるハードウェア記述の制約をまとめる．Verilog HDL によるハードウェアの記述と ArchHDL によるハードウェアの記述の大きな違いは配列の扱いである．ArchHDL では，ポートとして利用するワイヤ，レジスタに配列を用いることができるが，Verilog HDL ではそれらに配列を用いることが

できない．配列で宣言されたワイヤやレジスタをポートとして利用するために，その記述に制約を設ける．

**クラス**

- モジュールは，ライブラリが提供する *Module* クラスを public で継承したクラスにより定義する．*Module* クラスを public で継承したクラス以外のクラスは定義できない．
- テスト記述のためのモジュールは，*TestTop* というクラス名で定義する．*TestTop* クラスのメンバ変数に，テスト終了フラグをセットする *HALT* という名前のレジスタ型変数を宣言する．

**定数・変数・クラスのインスタンス**

- 定数は，“static const int” を使用する．
- ハードウェアのワイヤはライブラリが提供する *wire* クラス，レジスタはライブラリが提供する *reg* クラスを用いて宣言する．これらの変数を配列で宣言する場合は，2 次元配列までとする．
- wire や reg クラスを使用せずに変数を宣言した場合は，その変数をレジスタとして変換する．
- for 文のイテレータとして，初期化式において”int” か”uint” を使用することができる．これらの変数は”integer” 型の変数に変換される．
- 入力ポートを宣言する場合はポート名を「i_」で始まる名前，出力ポートを宣言する場合はポート名を「o_」で始まる名前にする．
- ポートとなる変数が *wire* クラスの変数であれば配列を用いて宣言することができる．その場合の配列は 1 次元までである．
- クラスのインスタンスは 1 次元配列を用いて宣言することができる．

**構造体**

- wire や reg の変数に使用するデータ型として，構造体を定義し使用することができる．構造体のメンバ変数に配列を使用することはできない．
- 構造体のコンストラクタ，代入演算子のオーバーロードを定義することができる．

**関数**

- クラス内の関数は，*PortConnect*，*Assign*，*Initial*，*Always* の 4 つの関数のみ定義できる．また，メイン関数以外の他の関数を定義することはできない．
- *PortConnect* 関数には，モジュール内に宣言した他のモジュールのポートの接続を記述する．
- *Assign* 関数には，*wire* で宣言された変数への代入を記述する．
- *Initial* 関数には，レジスタ型変数への値の代入を記述する．代入は，演算子”=” によるブロッキング代入を用いる．
- *Always* 関数には，レジスタ型変数へ値の代入を記述する．代入はノンブロッキング代入とブロッキング代入を用いる．
- メイン関数には，*TestTop* クラスのインスタンスの宣言および，*TestTop* クラスのメンバ変数 *HALT* を条件文とした while 文のみを記述する．while 文では，ライブラリが提供する *Step* 関数を呼ぶ．

## 3.4 関連研究

### 3.4.1 Verilog HDL，VHDL 以外の言語を用いるハードウェア記述言語

Verilog HDL や VHDL 以外の言語を用いるハードウェア記述言語として SystemC[54], MyHDL[55]，Chisel[56] などがある．これらは，それぞれのベースとする言語のプログラムとしてコンパイル，シミュレーションが可能である．また，ハードウェアを RTL で記述することができる．しかし，記述のスタイルは Verilog HDL とは異なる．

SystemC は C++ でハードウェアを記述する．プログラマはライブラリによって提供されるクラスやマクロを用いてハードウェアを記述する．

MyHDL は Python でハードウェアを記述する．ソースコードは Verilog HDL，VHDL に変換することができる．ネイティブコードにコンパイルしたときのシミュレーション速度はフリーの Verilog HDL シミュレータである Icarus Verilog[57] の約 3 倍である．

Chisel は Scala でハードウェアを記述する．ソースコードは Verilog HDL に変換することができる．高速なシミュレーションのために，Scala の記述から C++ シミュレータ用のクラスを生成する．これによる C++ でのシミュレーションは，商用の Verilog HDL シミュレータである Synopsys VCS に対して約 8 倍高速である．

### 3.4.2 Reactive Programming

Reactive Programming は，時間による変化や外的要因による変化などの振る舞いを記述するためのプログラミングパラダイムである．シミュレーションやアニメーションなどのイベント駆動型のアプリケーションやユーザインタフェースなどのインタラクティブなアプリケーションを記述するために開発された [58]．このパラダイムは，文献 [59] において初めて提唱されている．

Reactive Programming における重要な概念として，*Behavior* と *Event* がある．*Behavior* は，時間と共に変化する値を表す．*Event* は，外的要因によって変化する値を表す．Verilog HDL において，assign 文で定義されるワイヤは *Behavior* に相当し，always ブロックで決められたタイミングで値が代入されるレジスタは *Event* に相当する．

Reactive Programming を可能とするライブラリが多くの言語で開発されている．特に，純関数型言語における Reactive Programming は，Functional Reactive Programming（FRP）と呼ばれ，Haskell をベースとしたライブリとして Fran[59]，Yampa[60]，Reactive[61]，Grapefruit[62] などがある．また Haskell 以外にも，Java ベースの Frappé[63]，Scala ベースの Scala.React[64] などがある．Sodium[65] は，複数の言語に対応しており，Java，Haskell，C++ で Reactive Programming が可能なライブラリである．

Reactive Programming を用いてハードウェアを記述する言語として Estrel[66] がある．Esterel は，イベントベースの記述によりハードウェアを記述する．なお，Scala をベースとする Chisel は，Reactive Programming は用いていない．

## 3.5 評価

ハードウェアの ArchHDL によるシミュレーションと Verilog HDL によるシミュレーションの速度を比較し，ArchHDL の有用性を示す．まず評価に用いるハードウェアについて述べ，次にシミュレーション時間の評価結果を示す．

### 3.5.1 評価に用いるハードウェア

評価には，Network-on-Chip（NoC）ルータおよびプロセッサを用いる．それぞれのハードウェアは ArchHDL で実装する．Verilog HDL のコードは変換ツールにより生成する．

#### Network-on-Chip ルータ

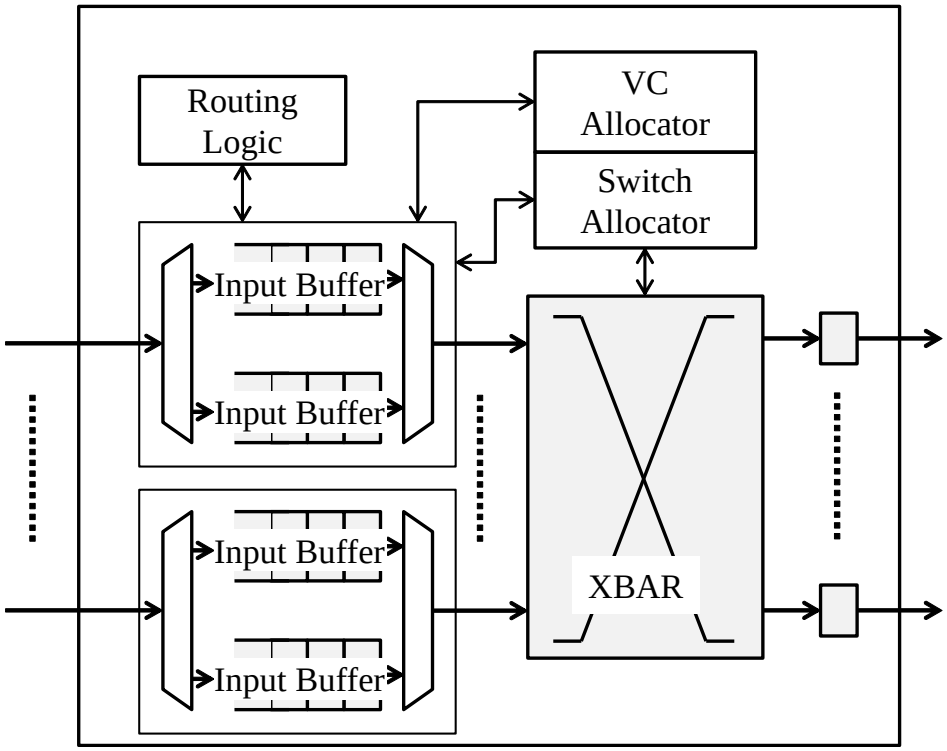


図 3.24 評価に用いる Network-on-Chip ルータのアーキテクチャ

図 3.24 に評価に用いる NoC ルータのアーキテクチャを示す．典型的な仮想チャネル付きの Input-buffered ルータである．ルータのパイプラインは Route Computation（RC），Virtual Channel Allocation（VA），Switch Allocation（SA），Switch Traversal（ST）の 4 段とする．ルータのみを用いた評価のため Link Traversal（LT）のない構成としている．スイッチング方式はワームホール，ルーティングは XY 次元順，フロー制御はクレジット方式とする．仮想チャネルおよびスイッチの調停アルゴリズムは iSLIP を用いる．仮想チャネルの割り当ては固定とする．したがって，パケットは生成時に設定された仮想チャネル番号は宛先に到達するまで変わらない．

ArchHDL のソースコードの実装ではチャネル数をパラメータ化し，プログラマが指定することができる記述となっている．同様に，バッファサイズもパラメータ化し，プログラマが指定することができる記述となっている．また，入力バッファの実装は，図ではチャネルごとに分けて示しているが，入力ポートごとに 1 つのメモリとして実装している．したがって，各ポートに搭載されているバッファは，サイズがバッファサイズ × 仮想チャネル数の 1 read/write のメモリである．仮想チャネルアロケータ，スイッチアロケータ，クロスバーの記述は，固定の仮想チャネル割り当ておよび XY 次元順ルーティング向けに最適

```
 51 crossbar.h                  71 crossbar.v
333 ib_vc_router.h             342 ib_vc_router.v
133 input_port_control.h       162 input_port_control.v
 25 memory.h                    25 memory.v
 15 mux.h                       23 mux.v
 67 output_port_control.h       94 output_port_control.v
 57 round_robin_allocator.h     73 round_robin_allocator.v
 50 round_robin_arbiter.h       66 round_robin_arbiter.v
 29 route_computation_unit.h    22 route_computation_unit.v
-------------------------------------------------------------
760 total                      878 total
```

図 3.25 ArchHDL で記述したルータのコード量

図 3.26 Verilog HDL に変換したルータのコード量

化を施している．

このルータの ArchHDL で記述したコード量を図 3.25 に示す．また，ArchHDL のコードを Verilog HDL に変換したコード量を図 3.26 に示す．主なソースコードの増加分は，ポートとして用いるワイヤを ArchHDL で *wire* は配列で宣言しているが，変換した Verilog HDL のコードでは，それらをビット連結により 1 つのワイヤにまとめてポートとして宣言し，モジュール内でビット切り出しによりワイヤの配列に分けるという記述に変更している分である．

**プロセッサ**

図 3.27 に実装するプロセッサのデータパスを示す．このプロセッサは 5 段パイプラインのプロセッサで, 命令セットは 32 ビットの MIPS である．文献 [67] において紹介されているプロセッサである．ArchHDL でプロセッサを記述したコード量は 914 行，Verilog HDL に変換後のコード量は，848 行である．

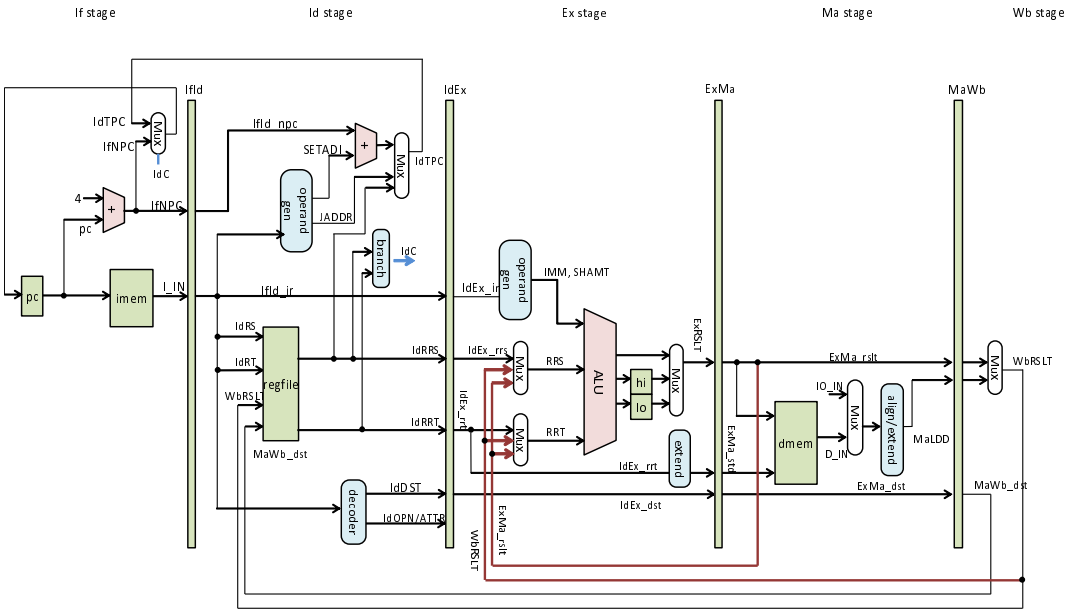


図 3.27　評価に用いるプロセッサのデータパス

### 3.5.2　シミュレーション速度の評価

ArchHDL で記述したハードウェアのシミュレーション速度について，Verilog HDL に変換したコードを Synopsys VCS を用いてシミュレーションした場合と比較する．シミュレーション時間は time コマンドにより 10 回計測を行い，その平均値をとる．評価環境は次の通りである．

- CPU：Intel Xeon E5-2687W（物理 8 コア，論理 16 コア）
- メモリ：32 GB
- OS：Ubuntu 13.04 x86_64
- GCC：バージョン 4.7.3，最適化オプション -Ofast
- インテルコンパイラ：バージョン 14.0.1，最適化オプション -Ofast
- Clang：バージョン 3.3，最適化オプション -Ofast
- VCS：バージョン H-2013.06，64 ビット版

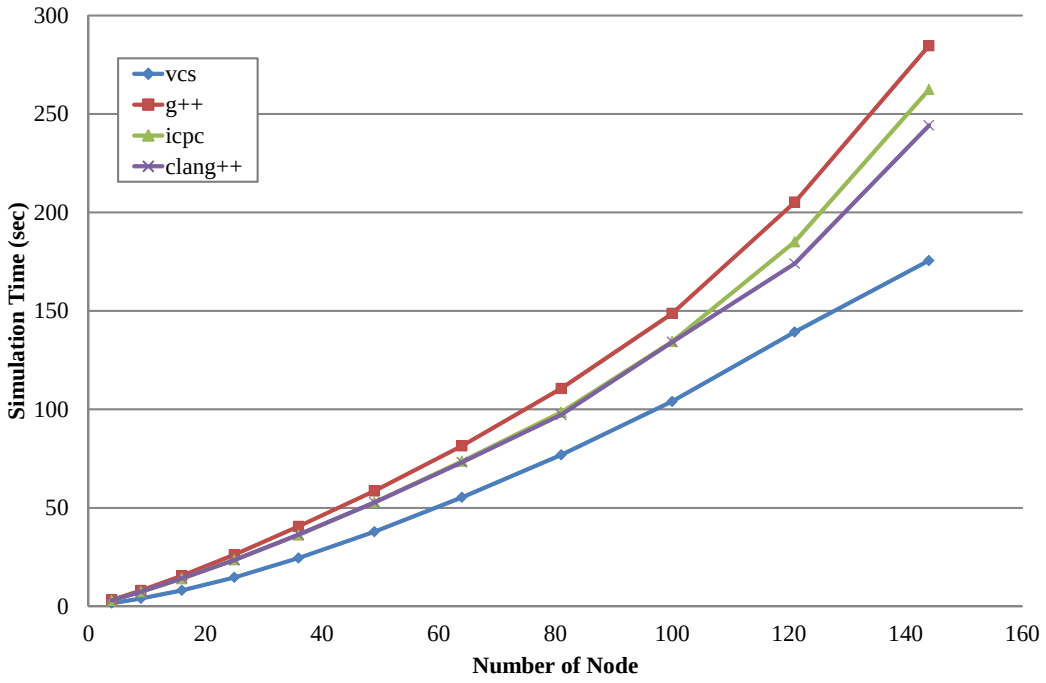


図 3.28　Network-on-Chip のシミュレーション時間の比較

**Network-on-Chip ルータを用いた評価**

ArchHDL で実装した NoC ルータを N×N の 2 次元メッシュ状に接続し，ネットワーク性能のシミュレーションを行い，そのシミュレーション時間を計測する．

シミュレーション環境は次の通りである．

- ネットワークサイズ：N = 2 〜 12
- トラフィックパターン：ランダム
- レート：0.2 flit/node/cycle
- シミュレーションサイクル数：50k サイクル
- フリット幅：38 ビット
- バッファサイズ：4
- 仮想チャネル数：4

図 3.28 に，ネットワークサイズを変化させたときのコンパイラごとのシミュレーション時間を示す．シミュレーション時間は，どのコンパイラについてもネットワークサイズが増加するにつれ同様の時間増加となった．VCS によるシミュレーション時間がすべての

ネットワークサイズについて最も早く，次いでインテルコンパイラ，Clang，GCC という結果となった．ネットワーク性能のシミュレーションにおいて，GCC に対する VCS によるシミュレーションは 1.5 倍から 2 倍高速である．シミュレーション速度の比は，ネットワークサイズにしたがって大きくなる．

図 3.29 に，OpenMP を用いて並列化した場合のシミュレーション時間を示す．コンパイラは GCC を利用する．スレッド数が計算機の物理コア数と等しい 8 スレッドの場合と論理コア数と等しい 16 スレッドの場合のシミュレーション速度を示す．*vcs* および *g++ 1 thread* で示されているシミュレーション時間は図 3.28 と同じである．

並列化により 16 スレッドを利用したシミュレーション時間が最も早い結果となった．16 スレッドを使用したシミュレーションは，VCS に対して 1.8 倍から 5 倍高速である．シミュレーション速度の比は，ネットワークサイズにしたがって大きくなる．

図 3.30 に ArchHDL によるシミュレーションにおいて，1 スレッドの場合のシミュレーション時間を基準としたときのスレッド数ごとの速度比をネットワークサイズごとに示す．コンパイラは GCC を利用する．物理コア数と等しい 8 スレッドまでは，どのネットワークサイズにおいても並列化による高速化がみられる．8 スレッドより多いスレッド数の範囲では，ネットワークサイズが大きくなるにつれ並列化による高速化がみられる．ネットワークサイズが最大である 12×12 の場合で最も並列化の効果があり，16 スレッドの時に 8.3 倍の高速化を達成している．

#### プロセッサを用いた評価

ArchHDL で実装したプロセッサ上でアプリケーションを実行し，そのシミュレーション時間を計測する．実行するアプリケーションはクイックソートで，逆順に並んだ 1,024 要素をソートする．このアプリケーションの実行サイクル数は約 5M サイクルである．

図 3.31 にコンパイラごとのシミュレーション時間を示す．シミュレーション時間は，インテルコンパイラが最も早く，次いで Clang，GCC，VCS という結果となった．ArchHDL によるシミュレーション時間に大きな差は無く，VCS のシミュレーション時間の約 3 倍高速である．

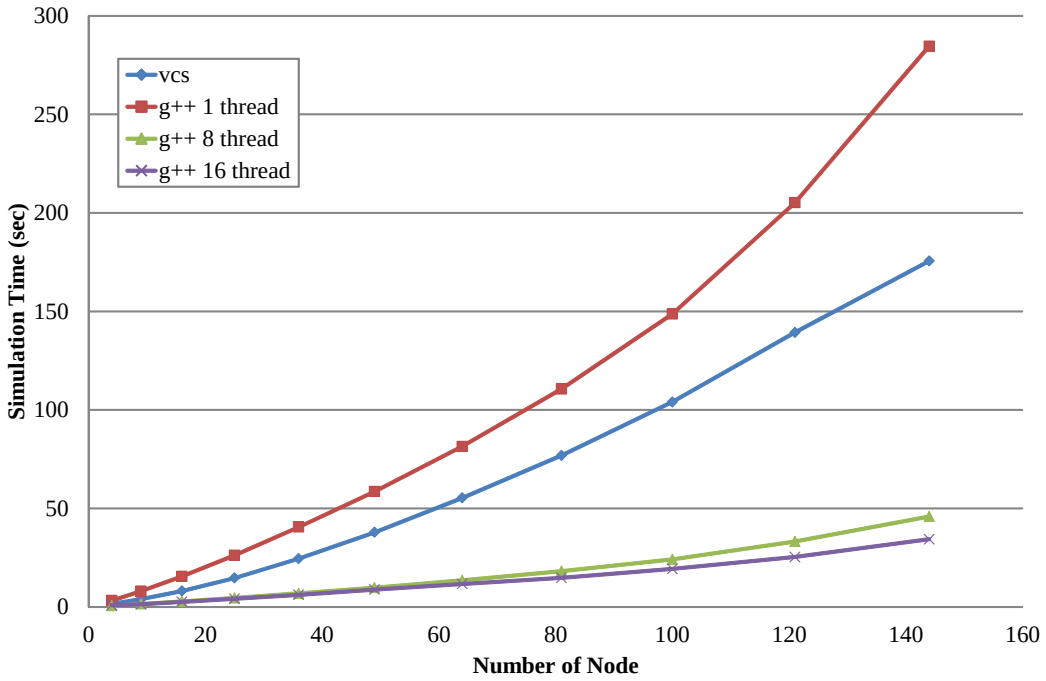


図 3.29　Network-on-Chip のシミュレーションについて, ArchHDL を OpenMP を用いて並列化した場合のシミュレーション時間

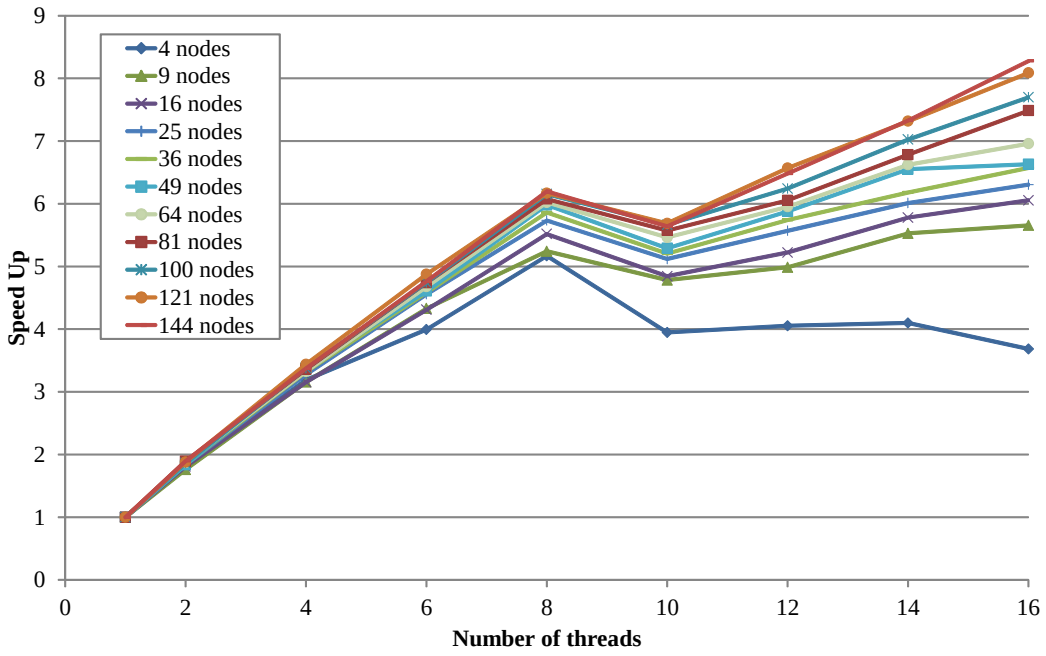


図 3.30　ネットワークサイズごとの並列化効率

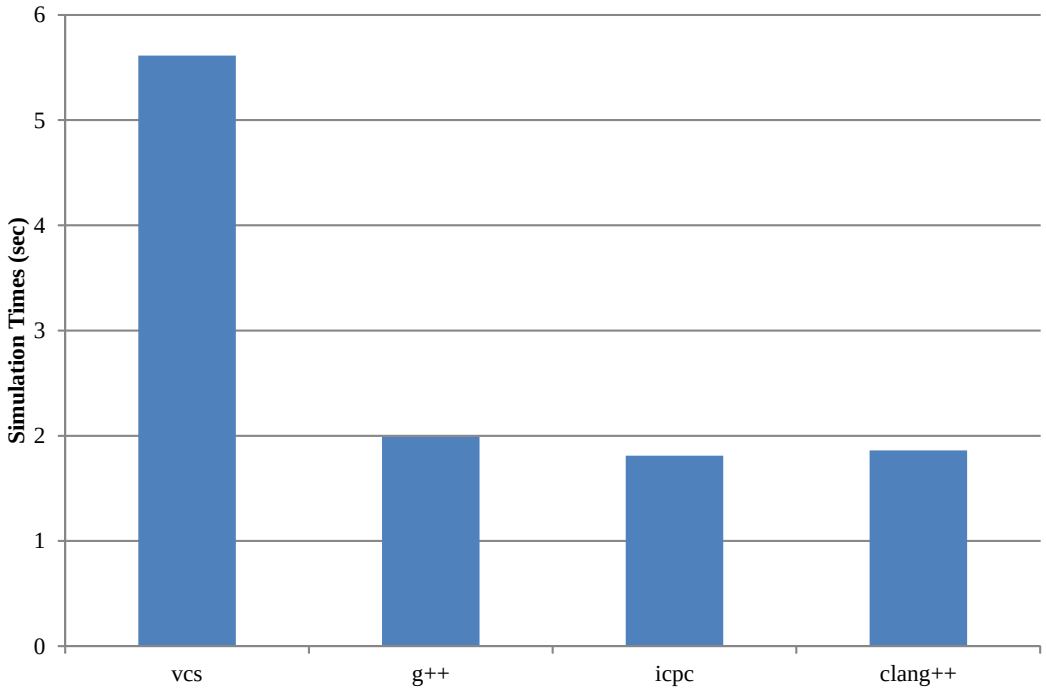


図 3.31　プロセッサのシミュレーション時間の比較

# 第 4 章

# 高性能 Network-on-Chip ルータアーキテクチャの提案

本章では，典型的な Input-Buffered 仮想チャネルルータ（IBR）よりも高性能な Network-on-Chip（NoC）ルータアーキテクチャを提案する．IBR よりもスループットにおいて高性能な Distributed Shared-Buffer NoC ルータ [68, 69, 14]（DSB ルータ）は，レイテンシにおいては IBR より劣る．本章では，まず DSB ルータのアーキテクチャを概説する．そして，そのパイプラインを削減する手法 [70, 71, 72] により，レイテンシにおいても IBR と同等の性能を達成する NoC ルータアーキテクチャを提案する．

## 4.1 Distributed Shared-Buffer NoC ルータ

高スループットな NoC ルータとして，Distributed Shared-buffer（DSB）ルータが提案されている．図 4.1 に DSB ルータのマイクロアーキテクチャを示す．DSB ルータは，共有メモリ型のアーキテクチャで，仮想チャネルごとの入力バッファの他に，ミドルメモリと呼ばれる共有メモリを搭載している．共有メモリを利用するために，ルータ内にクロスバーが 2 つ搭載されている．

入力チャネル側には Input-buffered Router（IBR）と同様のバッファが設けられている．入力バッファに格納されたフリットは，使用するミドルメモリが割り当てられると，1 つ目のクロスバーを通過し，ミドルメモリに格納される．ミドルメモリに格納されたフリットは，あらかじめ出力ポートを使用する順序が決められており，その順序に従って 2 つ目のクロスバーを通過し出力チャネルに転送される．ミドルメモリを利用することで，出力ポートにおける競合を解消し入力バッファの解放が早まるため，高いスループットを達成

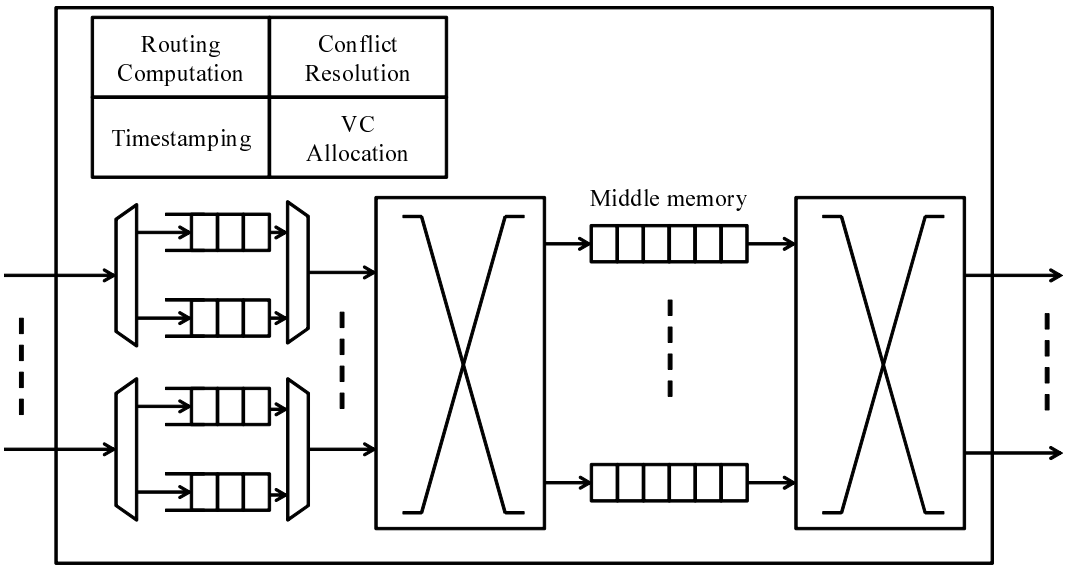


図 4.1 Distributed Shared-buffer NoC ルータのマイクロアーキテクチャ

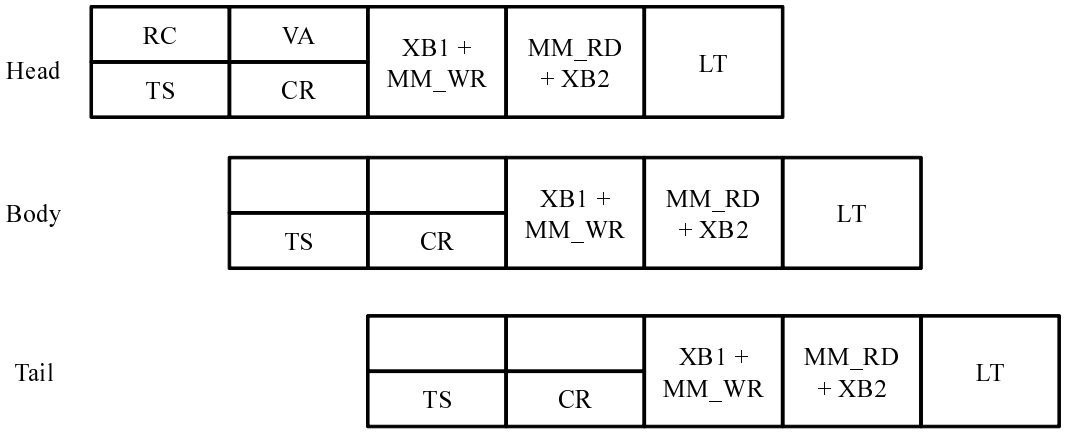


図 4.2 Distributed Shared-buffer NoC ルータのパイプライン

する．

DSB ルータでは仮想チャネルを採用しており，各入力チャネルのバッファは仮想チャネルごとに分けられている．フロー制御はクレジット方式を採用し，仮想チャネルごとにフロー制御が行われる．あるパケットの先頭フリットがルータに入力されてからすべてのフリットが隣接ルータに転送されるまで，1 つの入力バッファを 1 つのパケットが占有する．

図 4.2 に DSB ルータのパイプラインを示す．ミドルメモリとクロスバーの制御のためにパイプラインは 5 ステージ構成になっている．ルータに入力されたフリットは入力バッファに格納され，以下の処理を経て隣接するルータに転送される．

**Route Computation (RC)**

DSB ルータの RC ステージは Next Hop Routing Computation を採用しているため，次のルータにおける出力方向を計算する．典型的な Look-ahead Routing Computation と同様のハードウェア構成である．

**Timestamping (TS)**

DSB ルータの特徴的なステージである．このステージでは，各入力ポートからチャネルを 1 つ選択し，そのフリットがミドルメモリから読み出される予定時刻（タイムスタンプ）を計算する．フリットはこのタイムスタンプに従ってミドルメモリから読み出され，隣接するルータに転送される．同じ出力方向のフリットに対して同時にタイムスタンプを与える必要がある場合は，予定時刻をインクリメントしながら 1 サイクルずれたタイムスタンプを発行していく．この処理により，同じ時刻に同じ出力ポートを使用するフリットが存在しないことを保証する．このステージにおける処理の詳細は後に説明する．

**Virtual Channel Allocation (VA)**

パケットに出力先の仮想チャネルを割り当てる．前のステージで，タイムスタンプが発行されたパケットについて仮想チャネルが未割り当てであれば，出力先の仮想チャネルを割り当てる．IBR における VA は，典型的には 1 つの出力ポートが同時に複数の仮想チャネルを割り当てられる構成となっているが，DSB ルータでは 1 つの出力ポートが同時に 1 つしか仮想チャネルを割り当てない軽量化された構成を採用している．

**Conflict Resolution (CR)**

ミドルメモリは 1 ポートのメモリであるため，タイムスタンプを発行された複数のフリットを異なるミドルメモリに格納する必要がある．また，同じタイムスタンプが発行されたフリットは異なるミドルメモリに格納する必要がある．このステージでは，以上の条件を満たすように前のステージで選択されたチャネルに対して適切にミドルメモリを割り当てる．

**First crossbar traversal (XB1) + Middle memory write (MM_WR)**

このステージで，フリットは 1 つめのクロスバー 1 を経由し，ミドルメモリに格納される．TS と CR の処理により，ミドルメモリは同時に 1 つのフリットの書き込みと読み出しが行われるように割り当てられる．したがって，ミドルメモリは 1 read/write のメモリとなる．

**Middle memory read (MM_RD) + Second crossbar traversal (XB2)**

このステージで，フリットはミドルメモリから読み出され，2 つめのクロスバーを

通過し，各出力ポートに送られる．

**Link Traversal (LT)**

隣接するルータにフリットを転送する．

パイプラインの 1 つめのステージで RC ステージと TS ステージ，2 つめのステージで VA ステージと CR ステージが同時に処理される．したがって，パイプラインは 5 段の構成になる．RC ステージおよび VA ステージはヘッドフリットに対してのみ処理される．

### 4.1.1 Timestamping（TS）ステージ

TS ステージでは，フリットがミドルメモリから読み出され，2 つめのクロスバーを通過する時刻を計算し，フリットに発行する．この処理により，それぞれの出力ポートをフリットが通過する時刻を一意に決定し，フリットの出力ポートにおける競合を回避する．このステージは以下のような流れで処理される．

- 各入力ポートにおいてタイムスタンプを発行するチャネルを選択する．チャネルは LRU（Least Recently Used）により決定する．ただし，仮想チャネルが未割り当てのチャネルは他のチャネルより優先度が低くなっている．そのため，仮想チャネルがすでに割り当てられており，フリットを転送可能なチャネルがある場合は，仮想チャネルが未割り当てのチャネルが選択されないように制御される．
- 選択されたチャネルに対してタイムスタンプを発行する．そのチャネルが前のサイクルにおいてタイムスタンプが発行されたチャネルであれば，タイムスタンプはバッファの 2 番目のフリットに対して発行される．
- 同じ出力ポートに転送される複数のフリット（入力ポートは異なる）にタイムスタンプを発行する場合，タイムスタンプをインクリメントし，それぞれのフリットに異なるタイムスタンプを発行する．このときの優先度は固定で，ポート番号順にタイムスタンプがインクリメントされる．
- タイムスタンプを発行したフリットが次のステージにおいて，仮想チャネルの割り当て，もしくはミドルメモリの割り当てに失敗した場合，発行されたタイムスタンプは無効となる．

タイムスタンプの計算は次のように行われる．ある仮想チャネルにタイムスタンプを発行する際に，その仮想チャネルが使用する出力ポート $p$ に転送される先行するフリットがミドルメモリに 1 つも格納にされていない場合を考える．このとき，出力ポート $p$ に転送するフリットに発行するタイムスタンプ（*Timestamp*）は次の式で計算される．

$$Timestamp = Current\_Time + 3 \tag{4.1}$$

出力ポート $p$ に転送される先行するフリットがミドルメモリに 1 つも格納されていない場合，TS ステージを処理中の出力ポート $p$ へ転送されるフリットがミドルメモリから読み出される時刻は 3 サイクル後なので，現在時刻（*Current_Time*）に 3 を加えた値がタイムスタンプとなる．

次に，ミドルメモリに出力ポート $p$ を使用するフリットが格納されている場合を考える．DSB ルータでは，出力ポートごとに最後に発行されたタイムスタンプの値が保持されている．出力ポート $p$ を使用するフリットに対して最後に発行されたタイムスタンプを $LAT[p]$ とあらわす．あるサイクルで TS ステージを処理中のフリットには，出力ポートにおける衝突を防ぐために，その出力ポートの最後に発行されたタイムスタンプより遅い時刻を発行しなければならない．したがって，出力ポート $p$ の最も早いタイムスタンプ（*Timestamp*）は，式 4.1 とあわせて次の式で計算される．

$$Timestamp = max(LAT[p] + 1, Current\_Time + 3) \tag{4.2}$$

同じ出力ポートを使用する複数のフリットに同時にタイムスタンプを発行する場合，入力ポート順の固定優先度に従ってタイムスタンプを発行する．出力ポート $p$ を使用する複数のフリットに対して同時にタイムスタンプを発行する場合を考える．簡単のために，まずミドルメモリには出力ポート $p$ を使用するフリットが格納されていない場合を考える．同じ出力ポートに転送される複数のフリットには異なるタイムスタンプを発行する必要がある．DSB ルータでは，入力ポート $i$ に対して発行するタイムスタンプについてオフセット値（*offset*[$i$]）を設定する．オフセット値は，同じ出力ポートを要求している入力ポートに対して固定優先度に従った連続した値である．一番優先度の高いフリットのオフセット値は 0 で，優先度順にインクリメントしていく．したがって，入力ポート $i$ に発行されるタイムスタンプ $TS[i]$ は，次の式で表される．

$$TS[i] = Current\_Time + 3 + offset[i] \tag{4.3}$$

*offset*[$i$] は，入力ポート $i$ に設定されたオフセット値を表す．たとえば，入力ポート 4 のフリットが出力ポート $p$ を要求し，他に出力ポート $p$ を要求するフリットが存在しない場合，*offset*[4] は 0 となる．また，入力ポート 3 のフリットが出力ポート $p$ を要求し，他に入力ポート 0 と 1 のフリットが出力ポート $p$ を要求している場合，*offset*[3] は 2 となる．

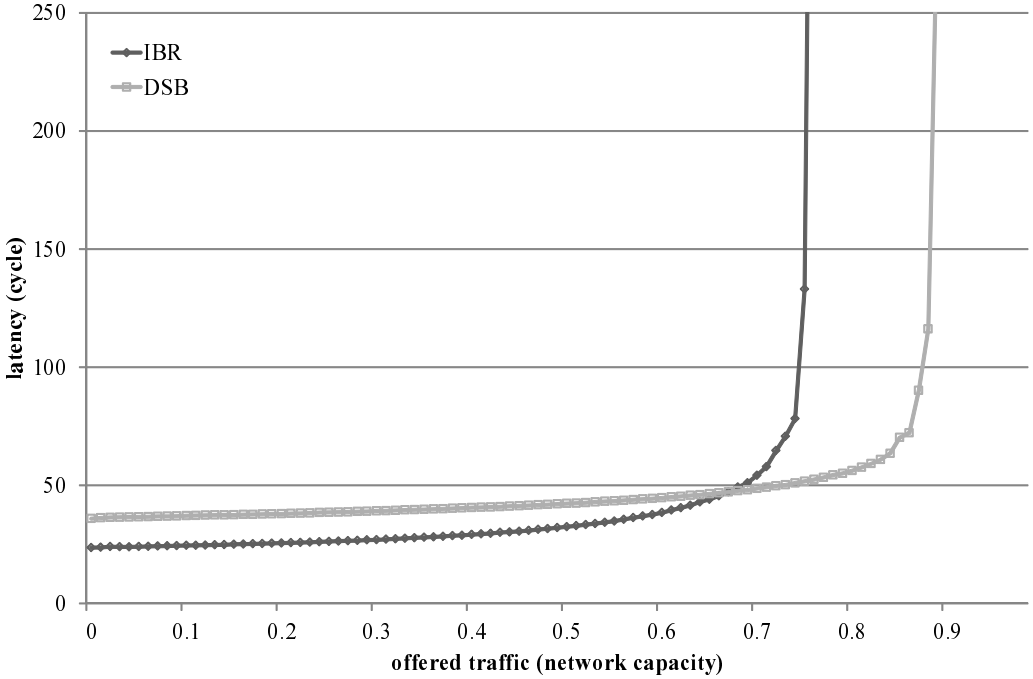


図 4.3 IBR と DSB ルータの性能比較

これらをまとめて，出力ポート $p$ を要求している入力ポート $i$ のフリットに発行するタイムスタンプ $TS[i]$ は次の式で表すことができる．

$$TS[i] = max(LAT[p] + 1, Current\_Time + 3) + offset[i] \tag{4.4}$$

### 4.1.2 Distributed Shared-Buffer NoC ルータの性能

3 段パイプラインの IBR と DSB ルータの性能を比較する．図 4.3 は，IBR と DSB ルータにおいて，パケットの注入レートを変化させたときの平均レイテンシの変化を示している．シミュレーションは，8×8 の 2 次元メッシュネットワークで，XY 次元順ルーティング，通信パターンはランダム，パケット長は 4 フリットの環境で行った．2 次元メッシュのネットワークのためルータの入出力ポート数は 5，IBR の仮想チャネル数は 8，各入力バッファのサイズは 5 フリット，DSB ルータの仮想チャネル数は 5, 各入力バッファのサイズは 4 フリット，ミドルメモリは 5 個で，各メモリのサイズは 20 フリットとし，IBR と DSB ルータのトータルのバッファサイズは共に 200 フリットである．これらのパラメータは，文献 [14] との比較のために同じ設定を用いている．

グラフの横軸はパケットを注入するレートを表しており，Network Capacity [1] で正規化している．注入レートが低い場合は，パイプライン段数が少ないため DSB ルータに比べて IBR のレイテンシが小さい．しかし，DSB ルータは IBR と比べて注入レートが高い領域において通信が可能であり，スループットが高いことがわかる．

## 4.2 Distributed Shared-buffer NoC ルータのパイプラインバイパス方式の提案

### 4.2.1 提案手法

Distributed Shared-buffer（DSB）NoC ルータは 3 段パイプラインの Input-buffered ルータ（IBR）と比較して，高スループットであるがレイテンシの点で劣っている．レイテンシは，ルータのパイプライン段数に大きく影響されるため，パイプライン段数を削減することでレイテンシの改善が見込まれる．そこで，DSB ルータのパイプラインをバイパスすることでパイプライン段数を 1 段削減する手法を提案する．

提案手法は，DSB ルータの第 3 ステージである第 1 クロスバーの通過とミドルメモリへの書き込みを行わず，フリットを第 4 ステージである第 2 クロスバーの通過へ直に出力する．図 4.4 に本手法を適用した DSB ルータのマイクロアーキテクチャを示す．入力ポートごとに第 1 クロスバーの手前から第 2 クロスバーの手前（ミドルメモリからの出力）にフリットを転送するバイパス回路が設けられている．バイパス回路は各入力ポートに 1 本実装する．例えば，入力ポート 0 のバイパス回路は，入力バッファの出力から，*Middle memory*[0] の出力に接続される．

この手法は，DSB ルータのように共有メモリ型のアーキテクチャを採用するルータで，隣接ルータにフリットを転送する際に 2 つのクロスバーを通過するような場合において有効な手法である．フリットは 1 段目のクロスバーのみをスキップし，2 段目のクロスバーは通常通りに通過するため，バイパスのための回路が固定の経路であっても，フリットを適切な方向に出力することができる．もし，クロスバーを 1 つしか持たないルータにおいて本手法のようにクロスバーをスキップするバイパス経路を設けた場合，バイパスできるフリットは特定の方向に出力されるフリットに限定されるため，バイパスの有効性が低下すると考えられる．

ここで，ミドルメモリの数について制約を設ける．提案手法は，入力ポートごとに異な

[1] ランダムの通信パターンを用いた場合のそのネットワークのスループットの理論値．8×8 の 2 次元メッシュの場合の Network Capacity は 0.5 となる．

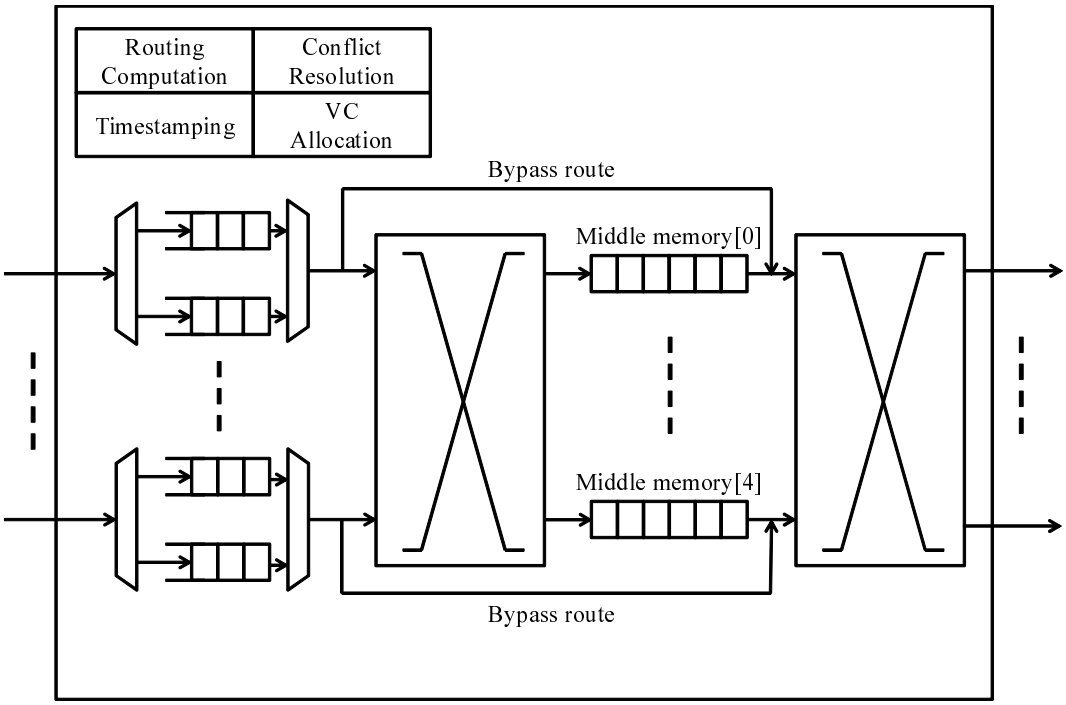


図 4.4 パイプラインをバイパスする Distributed Shared-buffer NoC ルータのマイクロアーキテクチャ

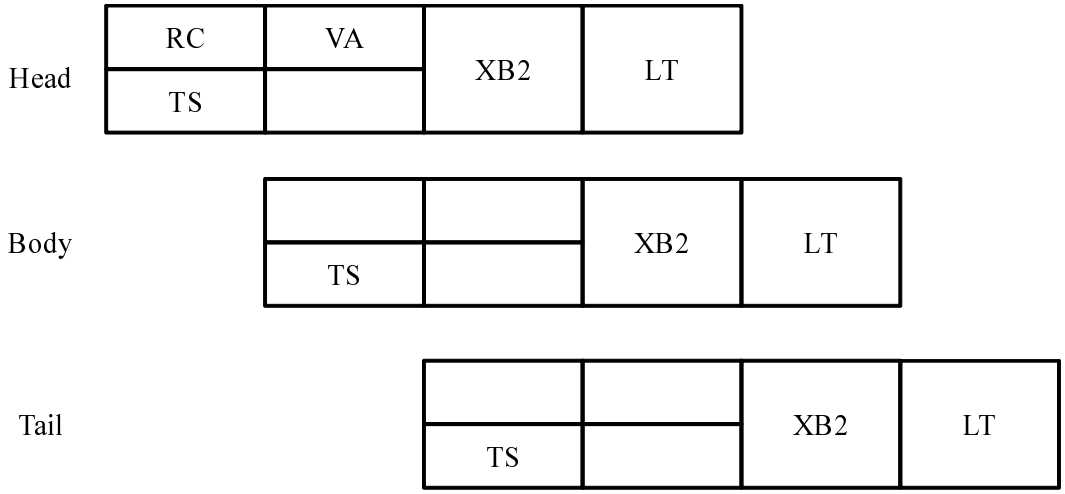


図 4.5 パイプラインのバイパスが成功した場合の Distributed Shared-buffer NoC ルータのパイプラインステージ

るミドルメモリからの出力へのバイパス回路が追加されるため，ミドルメモリは少なくとも入力ポート数分が必要である．例えば，メッシュネットワークの場合の入出力ポート数は 5 であるためミドルメモリは最低でも 5 個必要となる．文献 [14] の評価においてミドル

メモリの個数が 5 の場合と 10 の場合が比較されている．この評価から，ミドルメモリが 5 個の場合と 10 個の場合の性能差がほとんどないことが報告されている．以降は，簡単のためメッシュネットワークに限定し，ミドルメモリの個数を 5 として議論を進める．

図 4.5 に本手法によるパイプラインのバイパスが成功した場合のパイプラインの構成を示す．第 1 ステージの段階でフリットはバイパス可能かどうか判定され，バイパス回路を経由することでミドルメモリを利用せずに直に第 2 クロスバーに送られる．フリットがパイプラインをバイパスする場合，TS ステージでは *Current_Time* + *2* の値がタイムスタンプとして与えられる．このタイムスタンプの値から第 2 ステージにおいて第 1 クロスバーに出力するかバイパス回路に出力するかを決定する．これにより，従来のパイプラインから 1 段削減することができ，隣接するルータにフリットを 4 サイクルで転送することができる．

バイパス不可能な場合は，従来の 5 段パイプラインの DSB ルータと同様の挙動となる．したがって，バイパスができない場合のペナルティは無く，従来の DSB ルータと比較して性能の低下はないと考えられる．

### 4.2.2 TS ステージにおけるバイパス可否の判定方法

提案手法において，バイパス可能かどうかの判定はパイプラインの第 1 ステージで行う．そのときに，バイパスするフリットがミドルメモリから出力されるフリットと衝突しないことを保証する必要がある．フリットが格納されるミドルメモリが決定するのは CR ステージである．したがって，第 1 ステージの段階であるフリットがバイパス可能かどうかの判定には，先行するフリットがどのミドルメモリに格納されるかが不明なため，*Current_Time* + *2* のタイムスタンプを発行されたフリットが存在するかどうかという条件を用いる．

*Current_Time* + *2* のタイムスタンプを発行されたフリットが存在するかを判定するために，出力ポートごとに保持されている最後に発行されたタイムスタンプ（LAT）の値を利用する．この値より小さいタイムスタンプは発行された可能性があり，フリットがミドルメモリに格納されている可能性がある．逆に，この値より大きなタイムスタンプは発行されておらず，*Current_Time* + *2* の方が大きい値である場合，バイパス可能と判断できる．

図 4.6, 図 4.7 を用いて，具体的に説明する．図 4.6, 図 4.7 はある時刻におけるタイムスタンプの発行状況を示しており，発行されたタイムスタンプを出力ポートごとに示している．Conventional は従来の DSB ルータにおけるタイムスタンプの発行，Proposal は提案手法におけるタイムスタンプの発行である．図の黒丸は出力ポート $p$ における $LAT[p]$ を

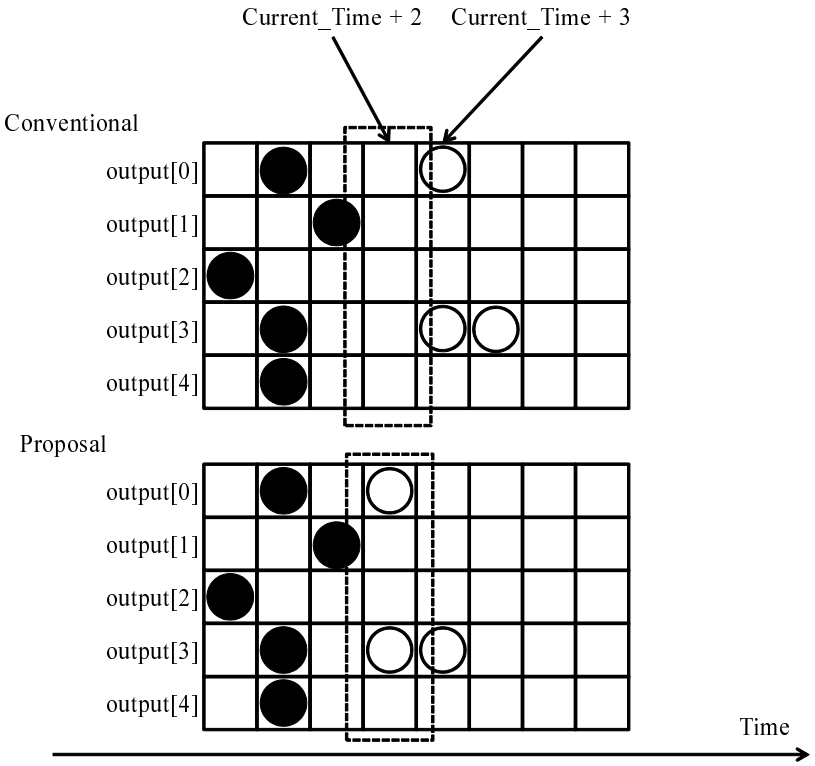


図 4.6 バイパス可能かどうかの判定例 (1)

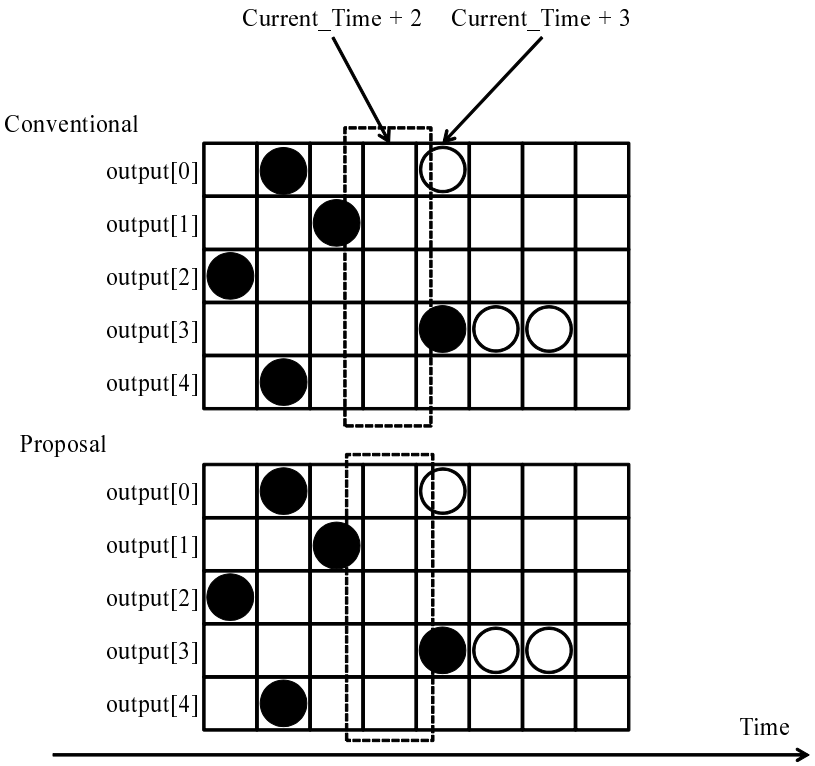


図 4.7 バイパス可能かどうかの判定例 (2)

示しており，白丸が新たに発行されたタイムスタンプを示している．どちらの図も，ある時刻に *output*[0] を要求するフリットが 1 つ，*output*[3] を要求するフリットが 2 つ存在し，それらのフリットにタイムスタンプを発行する様子を示している．

図 4.6 において従来の DSB ルータではいずれの *LAT*[*p*] よりも *Current_Time* + *3* の方が大きいため，新たに発行されるタイムスタンプは *Current_Time* + *3* 以上の値となっている．これに対して，バイパスする DSB ルータではすべての *LAT*[*p*] が *Current_Time* + *2* より小さいためバイパス可能と判定し，2 つのフリットに対して *Current_Time* + *2* のタイムスタンプを発行している．DSB ルータではタイムスタンプを発行されるフリットは異なる入力バッファのフリットであるため，バイパス回路の競合は発生しない．また，バイパス回路から直に第 2 クロスバーにフリットが転送されても，その時刻にミドルメモリから読み出されるフリットは存在せず，出力ポートの競合も発生しない．

図 4.7 は，バイパス不可能な場合の例である．*LAT*[3] の値が *Current_Time* + *2* よりも大きな値となっている．この場合，*Current_Time* + *2* を発行されたフリットが存在する可能性があり，フリットをバイパスしてしまうと同じタイムスタンプのフリットがミドルメモリからの出力で衝突するする可能性がある．したがって，このような状況ではバイパス不可能と判断し，従来の DSB ルータと同様にタイムスタンプを発行する．

パケット内のフリットの連続性は，先行するフリットが第 2 ステージにおける処理に失敗した時に，前のステージで与えられたタイムスタンプを無効化することで維持される．バイパス可能な場合は，第 2 ステージにおける仮想チャネル割り当てが失敗した場合のみこの処理が発生するが，これは従来の DSB ルータと同様の処理である．また，LAT を基にバイパス可能かどうかの判定をおこなっているため，バイパスするフリットが先行するフリットを追い越すことはない．したがって，提案手法においてもパケット内のフリットの連続性は維持される．

### 4.2.3 パイプラインバイパスのために追加されるハードウェア

パイプラインをバイパスするためのハードウェアとしては，バイパスのためのデータパスとその制御ロジックである．

図 4.8 に，バイパスのために追加するデータパスを示す．図は，ある入力ポート *p* に追加されるバイパス経路示しており，黒線で示した信号線とマルチプレクサが追加されるハードウェアである．このバイパス経路は，入力ポートごとに 1 つ追加される．したがって，2 次元メッシュにおける 5 入力のルータでは，このバイパス経路はルータ中に 5 個追加される．

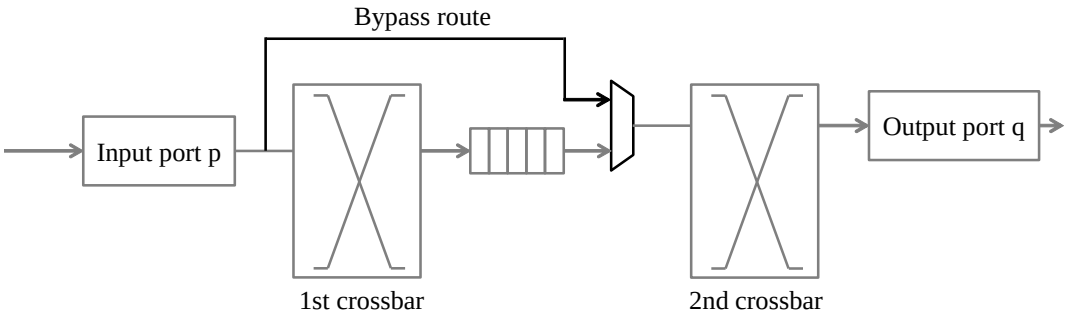


図 4.8 バイパスのために追加するデータパス

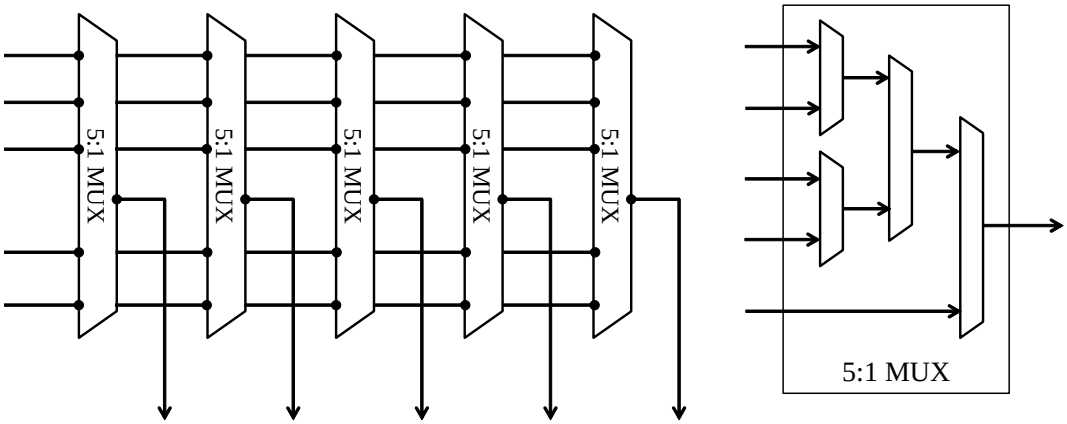


図 4.9 5×5 のクロスバーの構成と 5:1 マルチプレクサの構成

一般に，NoC ルータのハードウェア量の約 90% がバッファとクロスバーであることが知られている [13]．また，バッファサイズにもよるが，バッファとクロスバーのハードウェア量はほぼ同量である．図 4.9 に 5:1 マルチプレクサで構成した 5×5 のクロスバーと 2:1 マルチプレクサで構成した 5:1 マルチプレクサを示す．図から，5×5 クロスバーは 20 個の 2:1 マルチプレクサで構成されることがわかる．

バイパスのためのデータパスに必要なハードウェア量は，5 入力のルータの場合で，5 個の 2:1 マルチプレクサと信号線である．したがって，データパスのためのハードウェア増加量は，従来の DSB ルータの 1 つのクロスバーの 1/4 程度と見積もることができ，全体の数 % であると推測できる．

バイパスのための制御ロジックにおいてハードウェアの追加を必要とするのは TS ステージである．パイプラインバイパスの実現には以下の 2 点が条件となる．

- バイパス可能かどうかは，すべての $LAT[p]$ が $Current\_Time + 2$ より小さいかどう

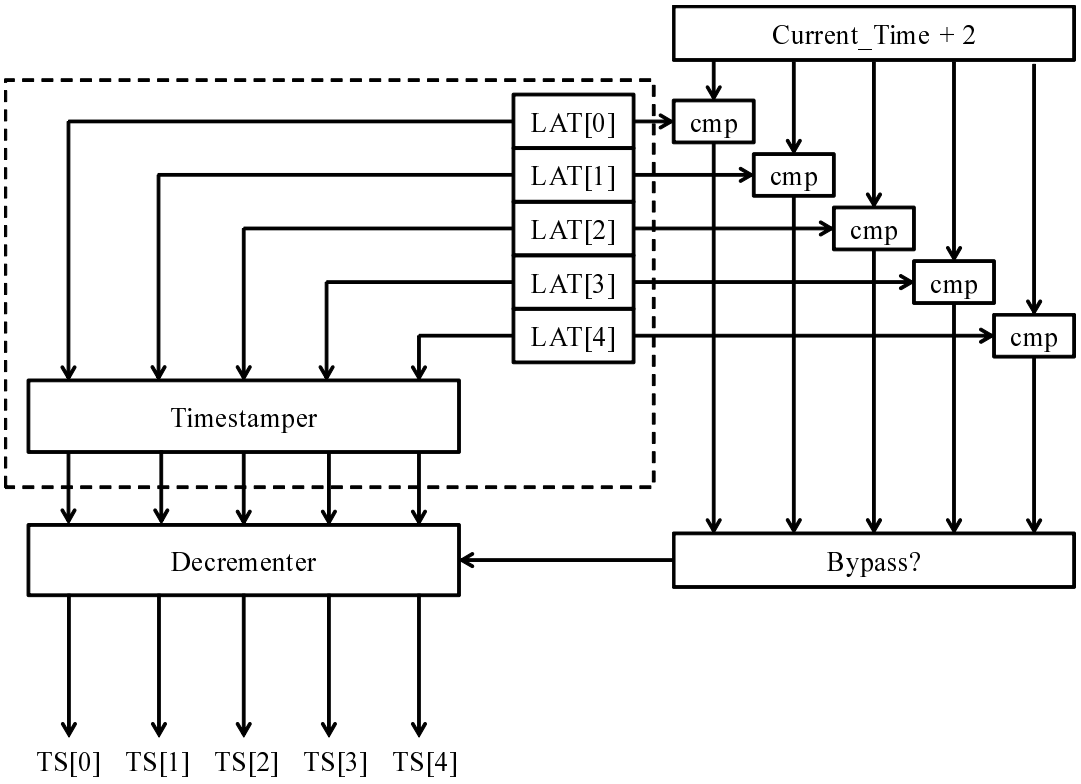


図 4.10 バイパスのために変更を加えた TS ステージ

かで判定する．

- バイパス可能な場合は従来の DSB において発行されたタイムスタンプすべてを −1 した値をタイムスタンプとして発行する．

図 4.10 に，上に挙げた条件を実現する TS ステージのブロック図を示す．図中の破線で囲まれた範囲は従来の DSB ルータにおける TS ステージに必要なハードウェアである．全ての $LAT[p]$（出力ポート $p$ における最後に発行されたタイムスタンプ）と $Current\_Time + 2$ の大小を比較器により比較し，その結果からバイパスするかどうかを決定する．バイパスする場合はデクリメンタにより計算された $TS[p]$（入力ポート $p$ に発行されるタイムスタンプ）がすべて −1 される．これらを実現するハードウェアの規模は DSB ルータ全体と比べれば十分に少ない．

表 4.1 チャネル数とバッファサイズ

| Config. | Input buffer (per port) | | Middle memory | | Total buffer |
|---|---|---|---|---|---|
| | #VCs | Flits/VC | MM | Flits/MM | (flits) |
| Input buffered router | 8 | 5 | - | - | 200 |
| DSB router | 5 | 4 | 5 | 20 | 200 |

### 4.2.4 評価

#### 評価方法

評価には，独自に開発したフリットレベルのサイクルアキュレートなソフトウェアシミュレータを用いる．このシミュレータは，文献 [15] において開発されたシミュレータのネットワーク部分をベースに，パケットジェネレータを追加し，任意の通信パターンによる評価を可能にしたものである．シミュレータに 3 段パイプラインの IBR，DSB ルータ，パイプラインをバイパスする DSB ルータを実装し，スループットとレイテンシの比較，バイパス成功率の評価から提案手法の有効性を示す．

評価環境は，8×8 の 2 次元メッシュトポロジ，XY 次元順ルーティングのネットワークを用いる．表 4.1 に，IBR および DSB ルータの仮想チャネル数，バッファ量のパラメータをまとめる．バッファ量は IBR と DSB ルータで等しくなるように設定し，合計で 200 フリット分とする．このため，IBR は仮想チャネル 8 本，各入力バッファ 5 フリットとする．DSB ルータと提案手法を適用した DSB ルータは同じパラメータを用いる．通信パターンは，代表的な Random，Complement，Tornado の 3 種を用いる．シミュレーションは，ウォームアップとして 10,000 サイクルを実行した後に計測を開始し，100,000 サイクルまで実行する．これらのパラメータや評価方法は，文献 [14] と比較し，提案手法の有効性を明確に示すために同じ手法をとっている．

本評価においては，すべての仮想チャネルを等しく扱う．したがって，パケットに新たに仮想チャネルを割り当てる場合は，割り当て可能な仮想チャネルの中から公平性のある優先度に基づきチャネルを選択する．

図 4.11, 図 4.12 に Random 通信における仮想チャネルの使用率を示す．図 4.11 は仮想チャネル 8，入力バッファ 5 の IBR，図 4.12 は仮想チャネル 5，入力バッファ 4 の DSB ルータの仮想チャネル使用率をパケットの注入レートごとに積み上げ面グラフで表している．横軸はパケットの注入レートを表している．ラベルは，割り当てられた仮想チャネル数を表している．使用率は，ルータの各ポートにおいて同時にパケットに割り当てられて

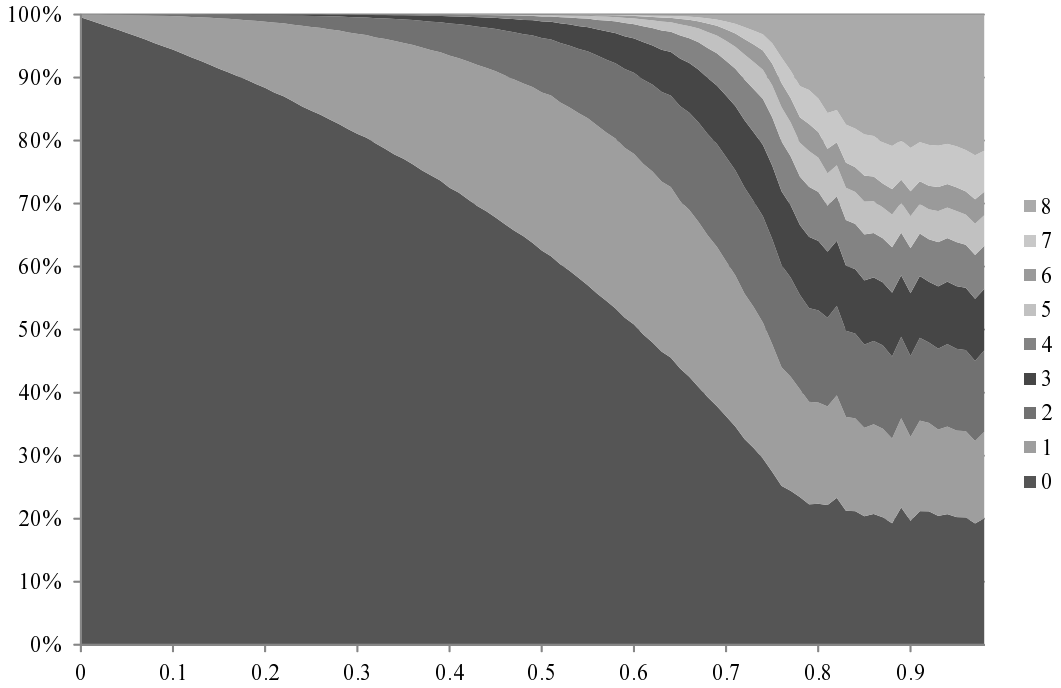


図 4.11 Random 通信における IBR の仮想チャネル使用率

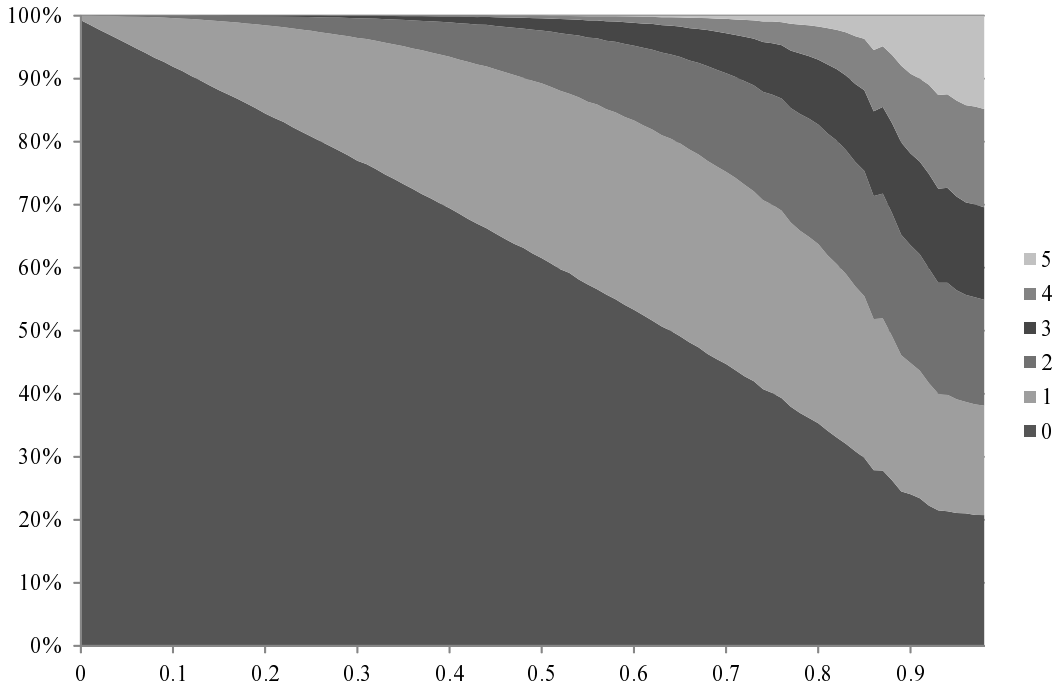


図 4.12 Random 通信における DSB ルータの仮想チャネル使用率

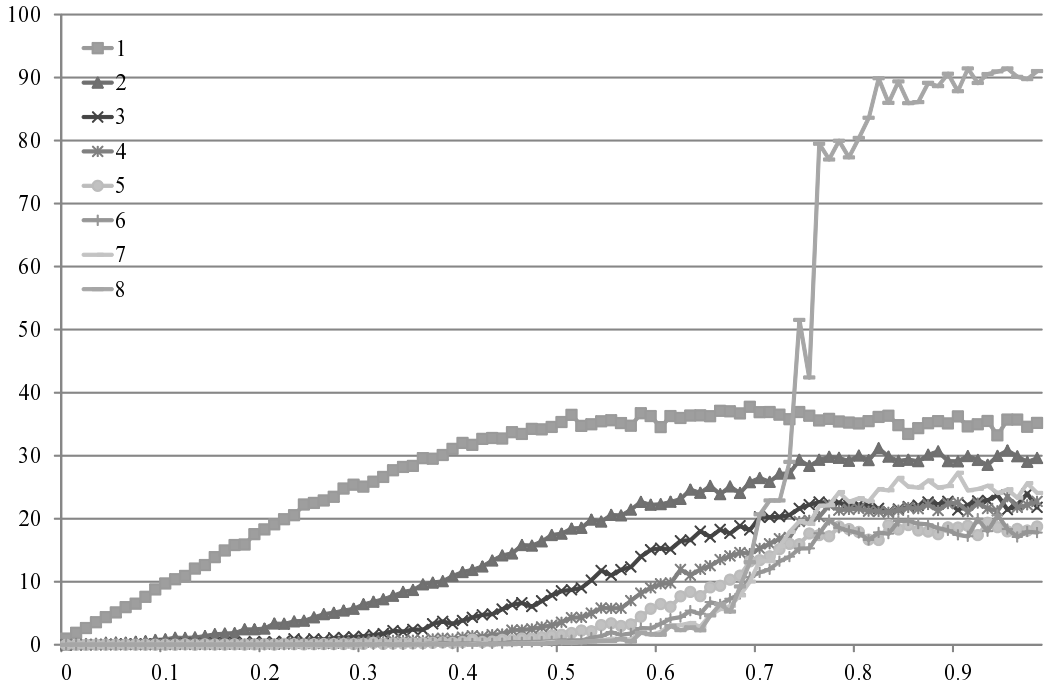


図 4.13 Random 通信における IBR の仮想チャネル使用率の最大値

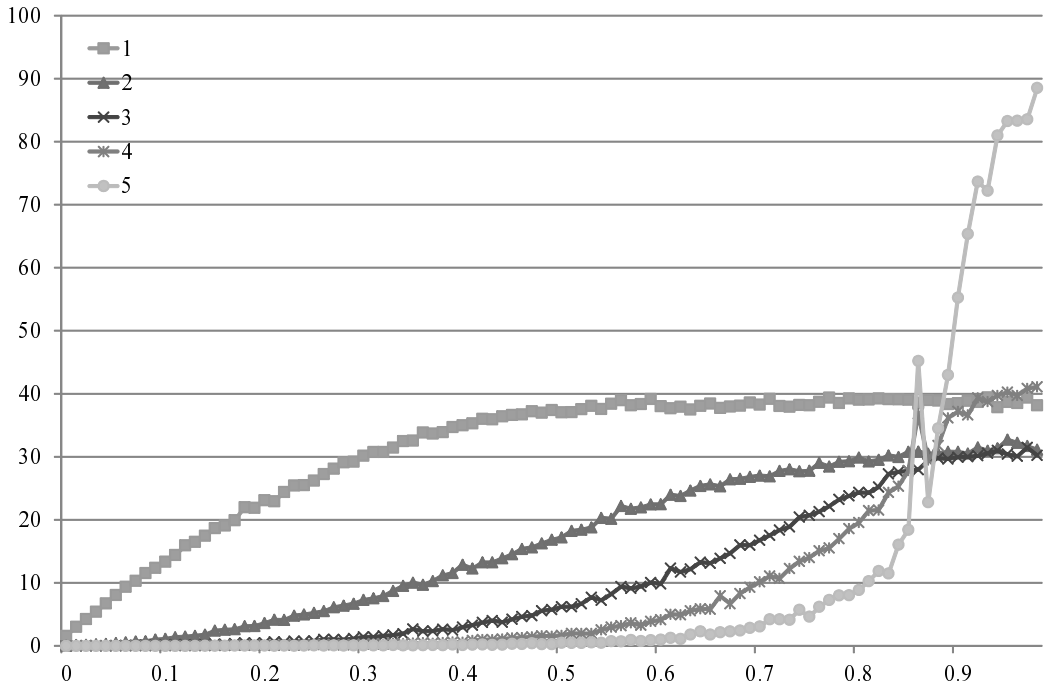


図 4.14 Random 通信における DSB ルータの仮想チャネル使用率の最大値

いる仮想チャネルの数ごとに，その期間（サイクル数）を合計し平均したものである．したがって，100% はシミュレーションサイクル数となる．たとえば，図 4.11 において注入レートが 0.7 のときは，シミュレーションサイクル数の 25% のサイクルにおいて使用された仮想チャネルは 1 であったことを示している．

図 4.13, 図 4.14 に Random 通信における仮想チャネルの使用率の最大値を示す．図 4.13 は仮想チャネル 8，入力バッファ 5 の IBR，図 4.14 は仮想チャネル 5，入力バッファ 4 の DSB ルータの仮想チャネル使用率をパケットの注入レートごとに仮想チャネルの使用率の最大値を示している．横軸はパケットの注入レートを表している．使用率の最大値は，ルータの各ポートにおいてパケットに割り当てられている仮想チャネルの数ごとに，その期間（サイクル数）を合計したものの最大値である．縦軸は，シミュレーションサイクル数に対するそれぞれのチャネル数が割り当てられた時間の割合である．たとえば，図 4.13 において注入レートが 0.7 のときに，あるポートにおいて使用されている仮想チャネルが 1 であった期間が他のポートと比べて最も長く，シミュレーションサイクル数の約 38% のサイクルであることを示している．

グラフより，IBR, DSB ルータにおいてパケットの注入レートが高い場合にすべての仮想チャネルが割り当てられている期間が確認できる．また，平均値と最大値の比較から，XY 次元順ルーティングによる局所性が確認できる．以上より，評価のパラメータとして用いた仮想チャネル数は，スループットに影響をあたえる有効な値である．

**スループット，レイテンシ，バイパス成功率の評価**

図 4.15，図 4.17，図 4.19 にパケット長を 4 フリットとし，パケットの注入レートを変化させたときの IBR，DSB ルータ，パイプラインをバイパスする DSB ルータの平均レイテンシを示す．それぞれ，図 4.15 は Random 通信の場合の平均レイテンシ，図 4.17 は Complement 通信の場合の平均レイテンシ，図 4.19 は Tornado 通信の場合の平均レイテンシ，を表している．グラフの横軸は，パケットの注入レートを表しており，Network Capacity で正規化している．また，レートの上限をそれぞれの通信パターンのスループットの理論値までとしている．縦軸は，レイテンシをサイクル数で表している．グラフのラベル *IBR* は 3 段パイプラインの IBR の平均レイテンシ，*DSB* は従来の DSB ルータの平均レイテンシ，*DSB_Bypass* は提案手法を適用した DSB ルータの平均レイテンシを表している．

図 4.16，図 4.18，図 4.20 にパケット長を 4 フリットとし，パケットの注入レートを変化させたときのバイパスの成功率を示す．それぞれ，図 4.16 は Random 通信の場合のバイパス成功率，図 4.18 は Complement 通信の場合のバイパス成功率，図 4.20 は Tornado 通

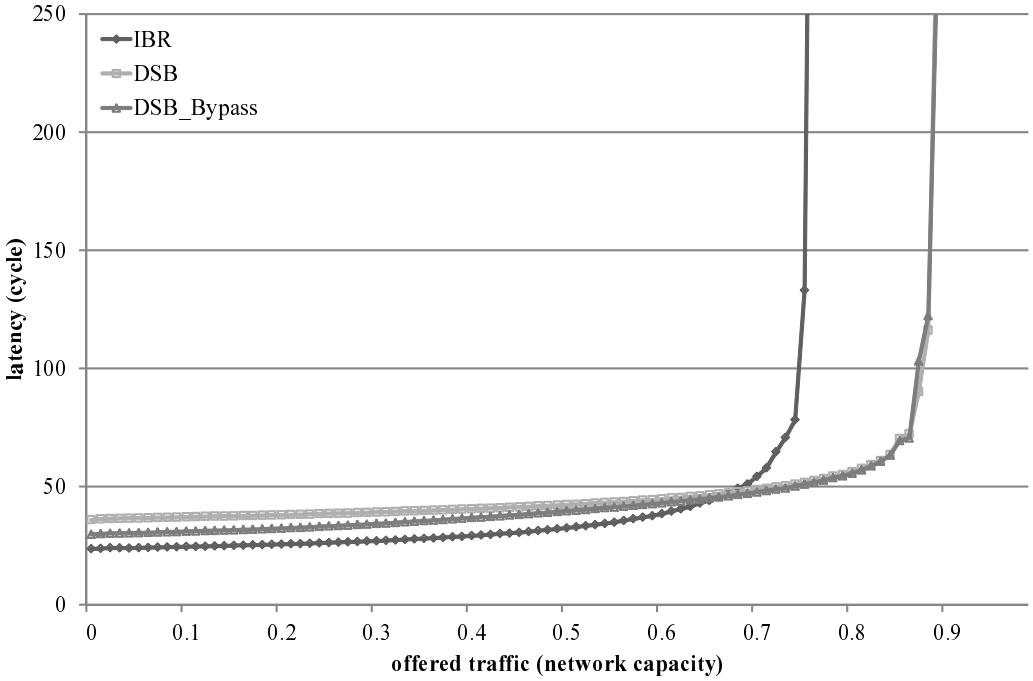


図 4.15 Random 通信の場合のレイテンシ（パケット長 4 フリット）

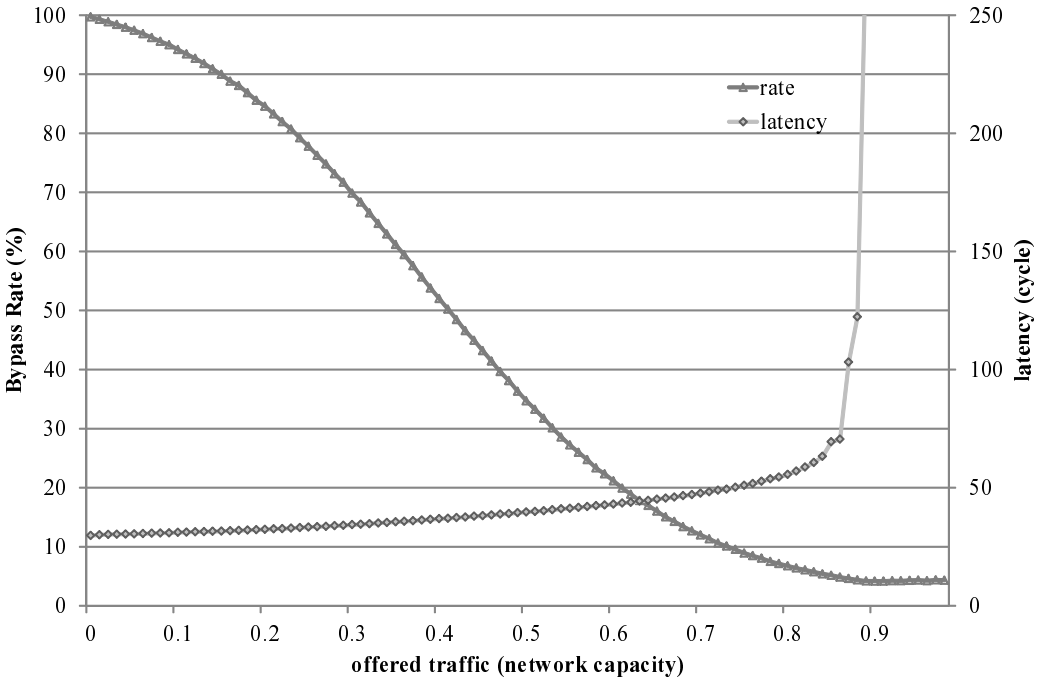


図 4.16 Random 通信の場合のバイパス成功率（パケット長 4 フリット）

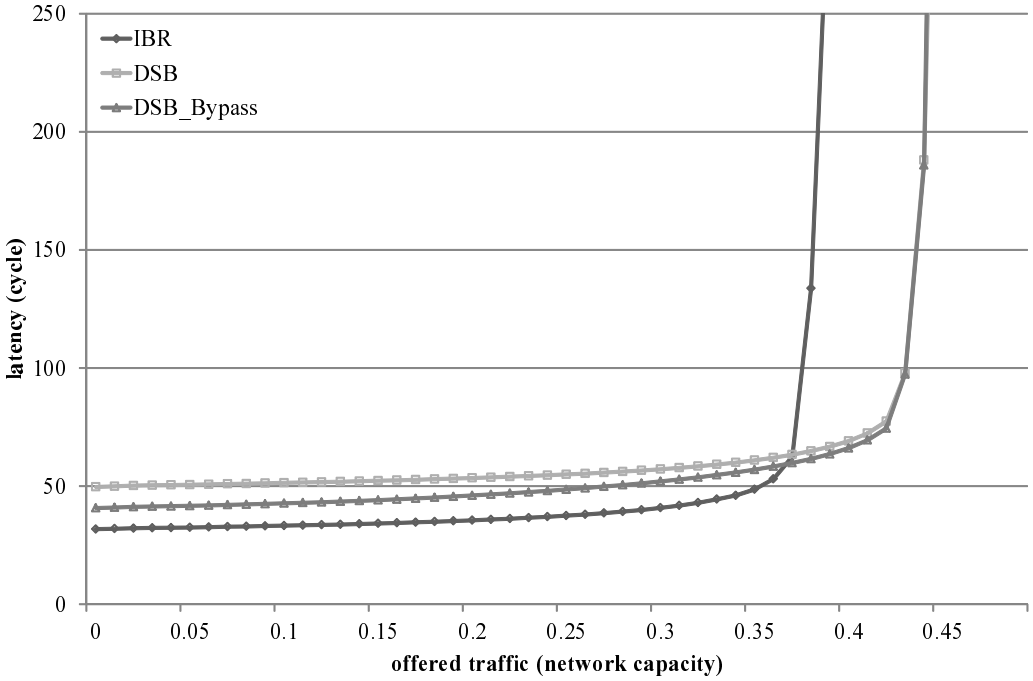


図 4.17 Complement 通信の場合のレイテンシ（パケット長 4 フリット）

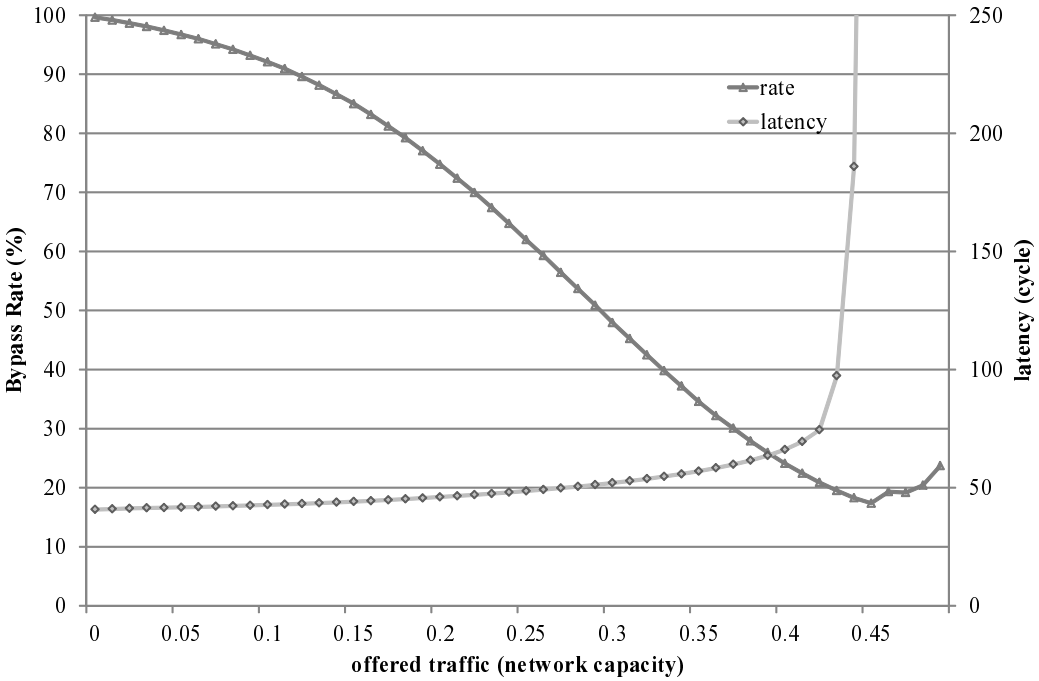


図 4.18 Complement 通信の場合のバイパス成功率（パケット長 4 フリット）

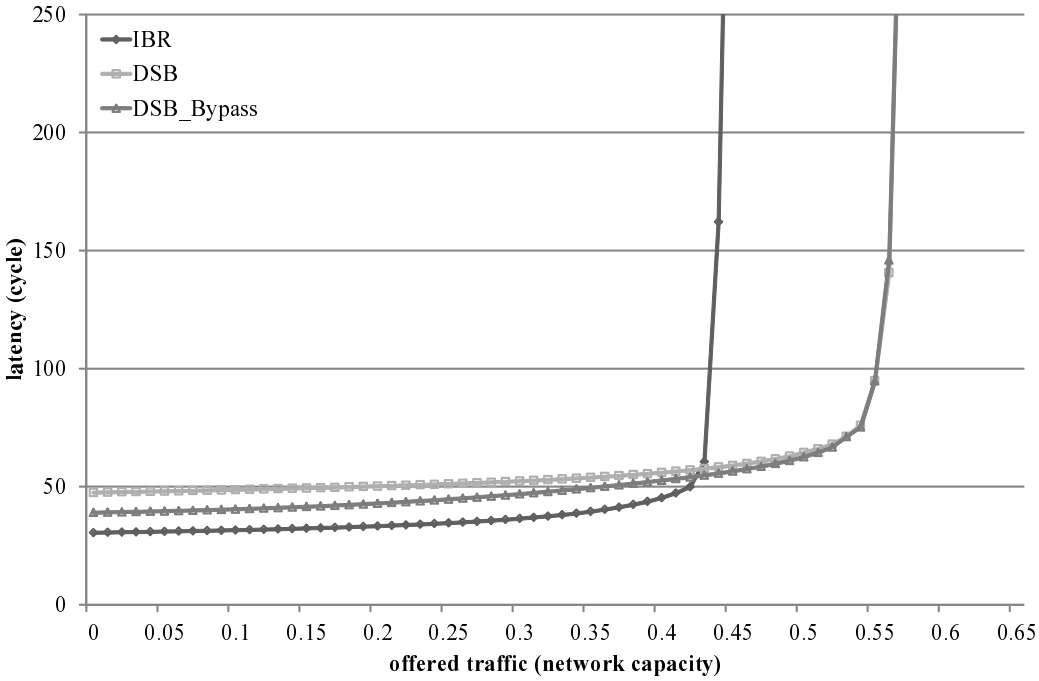


図 4.19 Tornado 通信の場合のレイテンシ（パケット長 4 フリット）

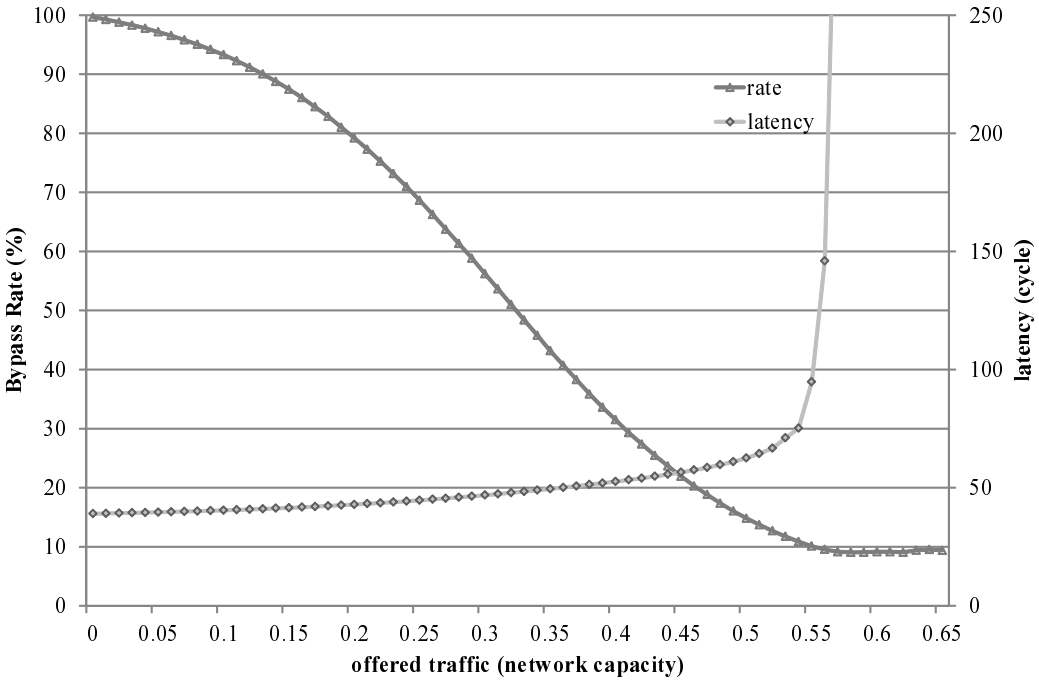


図 4.20 Tornado 通信の場合のバイパス成功率（パケット長 4 フリット）

表 4.2 最大のレイテンシの削減率とバイパス成功率（パケット長 4 フリット）

| traffic pattern | reduction rate (%) | bypass rate (%)) |
|---|---|---|
| uniform | 17.2 | 99.7 |
| complement | 17.9 | 99.6 |
| tornado | 17.9 | 99.6 |

信の場合のバイパス成功率，を表している．これらのグラフは，提案手法を適用した DSB ルータのレイテンシをあわせて示している．グラフの横軸は，パケット注入レートを表しており，Network Capacity で正規化している．また，レートの上限をそれぞれの通信パターンのスループットの理論値までとしている．グラフの左縦軸は，バイパスの成功率をパーセントで表しており，右縦軸はレイテンシをサイクル数で表している．

図 4.15，図 4.17，図 4.19 から，Random，Complement，Tornado のすべての通信パターンにおいて，従来の DSB ルータと比較して提案手法はパケットの注入レート低い領域においてレイテンシの改善が確認できる．また，注入レートが高い領域では，パイプラインをバイパスする DSB ルータと従来の DSB ルータのスループットは同じになり，提案手法を適用したことによるスループットに対してのペナルティがないことが確認できる．

図 4.16, 図 4.18, 図 4.20 から，Random, Complement, Tornado のすべての通信パターンにおいて注入レートが極めて低い場合にほぼ 100% のバイパス成功率を達成し，注入レートが高くなるに従ってバイパス成功率が低下していることが確認できる．Complement と Tornado 通信で，レイテンシが飽和した領域の注入レートにおいてバイパス成功率が上昇しているのは，2 つのパターンの特徴で，フリットを送信するチャネル選択の公平性が低い場合にスループットの低下が発生するためであると考えられる．Complement 通信の方が Tornado 通信と比較してスループットの制約が厳しいため，チャネル選択の公平性が低い場合のスループット低下が顕著になる．

表 4.2 にパケット長を 4 フリットとした場合の提案手法を適用した DSB ルータの従来の DSB ルータに対する最大の平均レイテンシの削減率およびバイパス成功率を示す．平均レイテンシの削減率およびバイパス成功率は注入レートが最も低い場合（今回取得したデータでは 0.01）に最大となり，Random 通信で削減率 17.2%，バイパス成功率 99.7%，Complement 通信と Tornado パターンでともに削減率 17.9%，バイパス成功率 99.6% を達成している．

最も注入レートが低い場合でも成功率が 100% を達成しない理由は，異なる入力バッファから同じ出力方向に同時にフリットを送信する場合がまれにあるからであると考えら

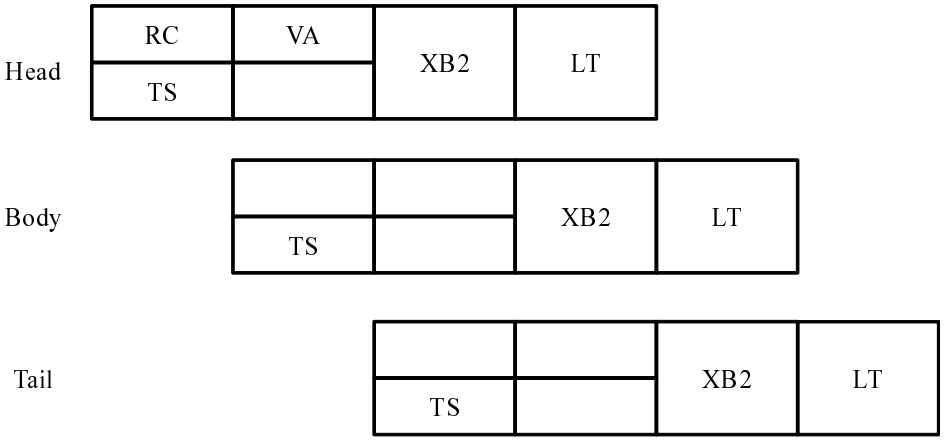


図 4.21 従来手法のパイプラインのバイパスが成功した場合の Distributed Shared-buffer NoC ルータのパイプライン

れる．異なる入力バッファから同じ出力方向にフリットを送信する状況は，宛先が同じフリットがルータからネットワークインタフェース方向に出力される場合や，次元順ルーティングにおいてパケットの進む方向が変わる場合に発生する．

## 4.3 Distributed Shared-buffer NoC ルータのパイプラインバイパス方式の改良

### 4.3.1 提案手法

ルータのパイプライン段数を削減することで NoC におけるレイテンシの改善が見込まれる．ここでは，DSB ルータのパイプラインをバイパスすることで 1 段削減する従来手法を改良し，パイプライン段数を 2 段削減する手法を提案する．

従来手法において，フリットをバイパスする場合，そのフリットは選択されたチャネルのバッファの先頭に格納されている．あるサイクルにおいて，バッファの先頭にあるフリットをバイパス回路に送るか，クロスバーに送るかは，フリットに発行されているタイムスタンプの値で判断する．

図 4.21 に，従来手法のパイプラインバイパスが成功した場合の Distributed Shared-buffer NoC ルータ（DSB ルータ）のパイプラインを示す．この図は，図 4.5 の再掲である．図から，パイプラインが可能な場合はボディフリットとテールフリットに関しては第 2 ステージの処理が無い．したがって，これらのフリットに関してはパイプラインバイパスによって第 2 ステージもスキップすることができる．ヘッドフリットに関しては，仮想チャ

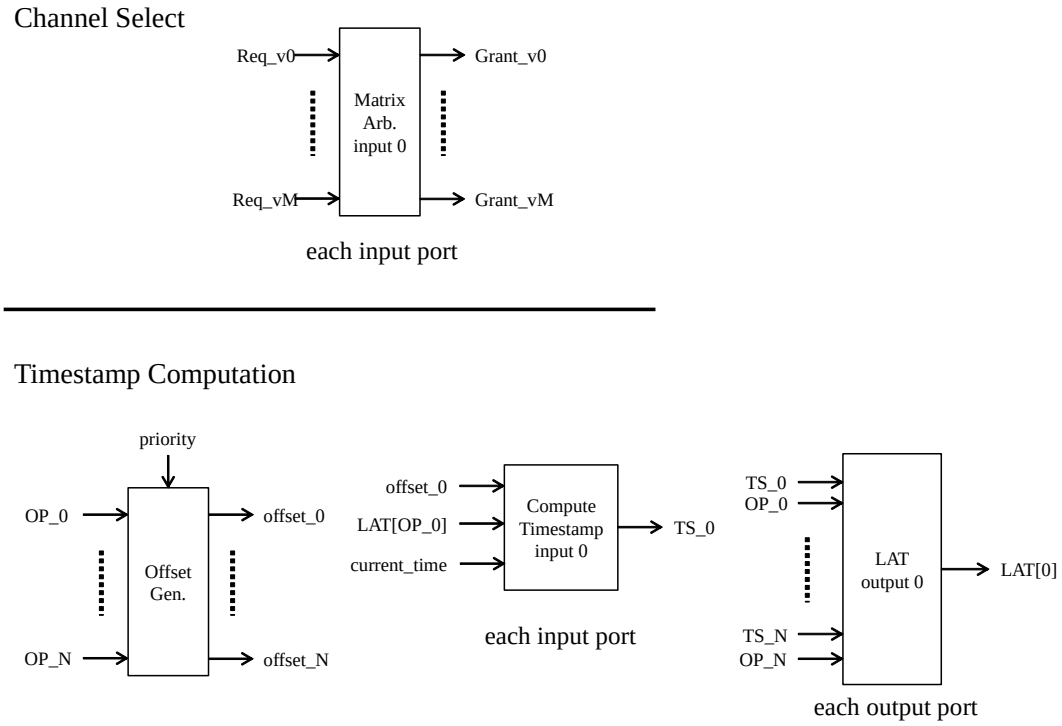


図 4.22 Distributed Shared-buffer NoC ルータのタイムスタンプ発行処理の詳細

ネルの割り当ての処理が第 2 ステージで必要である．ヘッドフリットの第 2 ステージで処理されている仮想チャネルの割り当て（VA）を第 1 ステージで行うことができれば，すべてのフリットについてパイプラインを 2 段スキップさせることができ，従来手法からさらにレイテンシの削減を達成する．

### 4.3.2 仮想チャネル割り当ての変更

DSB ルータは NRC を採用し，あるルータにおけるパケットの出力方向は，前のホップで計算済みである．仮想チャネルの割り当ては，出力方向の情報があればよいため，仮想チャネル割り当てを第 1 ステージで行うことは可能である．ここでは，TS ステージの処理と同時に VA を行うことによるルータの動作周波数への影響を議論する．

図 4.22 に，DSB ルータのタイムスタンプ発行処理の詳細を示す．TS ステージの処理は各入力ポートにおけるチャネル選択（Channel Select）と選択されたチャネルに対するタイムスタンプの計算（Timestamp Computation）に分けられる．

各ポートのチャネル選択では，フリットが転送可能状態にある仮想チャネルを選択し，リクエストをアービタに出す（Req_v0 から Req_vM）．図は，入力ポート 0 のアービタを

示し，また仮想チャネル本数を M とした場合を示している．アービタは Matrix アービタ [11] を採用しており，アービタ内のレジスタに保存されている情報から LRU によるリクエストの選択を行う．アービタよりグラント（Grant_v0 から Grant_vM）を受け取った仮想チャネルは，次のタイムスタンプ計算の処理を行う．

DSB ルータでは，選択したチャネルが第 2 ステージの処理に失敗すると発行したタイムスタンプは無効化される．そのため，高いスループットを維持するには第 2 ステージにおける処理の失敗を少なくし，有効なタイムスタンプを継続的に発行する必要がある．したがって，アービタにリクエストする際に，すでに仮想チャネルが割り当て済みかどうかや，転送可能なフリットがバッファに存在するかや，隣接ルータのバッファに空きがあるかなどを考慮する．これらの処理を経て，各仮想チャネルはリクエストを出している．

タイムスタンプの計算では，まず，各入力ポートで選択されたチャネル（以降では，簡単に入力ポートと呼ぶ）について，そのチャネル出力方向からタイムスタンプのオフセットを計算する（Offset Gen.）．オフセットジェネレータは，各入力ポートからの出力方向の情報（OP_0 から OP_N）からオフセットを計算し，それぞれの入力ポートに対してオフセット値（Offset_0 から offset_N）を出力する．図は，入力ポート数を N とした場合を示している．オフセット値は，同じ出力方向のポートがある場合に優先度（priority）にしたがってインクリメントした値となる．

オフセット値が計算されると，各入力ポートはタイムスタンプの計算を行う（Compute Timestamp）．計算には，オフセット値（offset_0），出力方向における最後に発行されたタイムスタンプの値（LAT[OP_0]），現在時刻の値（current_time）を用いる．図は，入力ポート 0 におけるタイムスタンプ計算の処理を示している．タイムスタンプ（TS_0）は，前節において述べた式 (4.4) による計算で求められる．

最後に，それぞれの入力ポートで計算されたタイムスタンプの値を比較し，各出力ポートにおける最後に発行されたタイムスタンプの値（LAT[0]）の計算を行う（LAT）．図は，出力ポート 0 における LAT の値の計算を示している．各入力ポートのタイムスタンプの値（TS_0 から TS_N）と出力方向（OP_0 から OP_N）から最も大きなタイムスタンプ値を求め，LAT とする．

第 1 ステージにおける仮想チャネル割り当ては，Channel Select で選ばれたチャネルのうち，まだ仮想チャネルが割り当てられていないチャネルに対して行われる．したがって，Timestamp Computation と同時に仮想チャネル割り当てが処理される．DSB ルータの仮想チャネル割り当ては，典型的な IBR よりもシンプルな方法を用いており，各出力ポートにアービタが 1 つ設けられているだけである．したがって，Channel Select における処理とほぼ同程度の処理で仮想チャネル割り当ては完了し，Timestamp Computation の処理と比

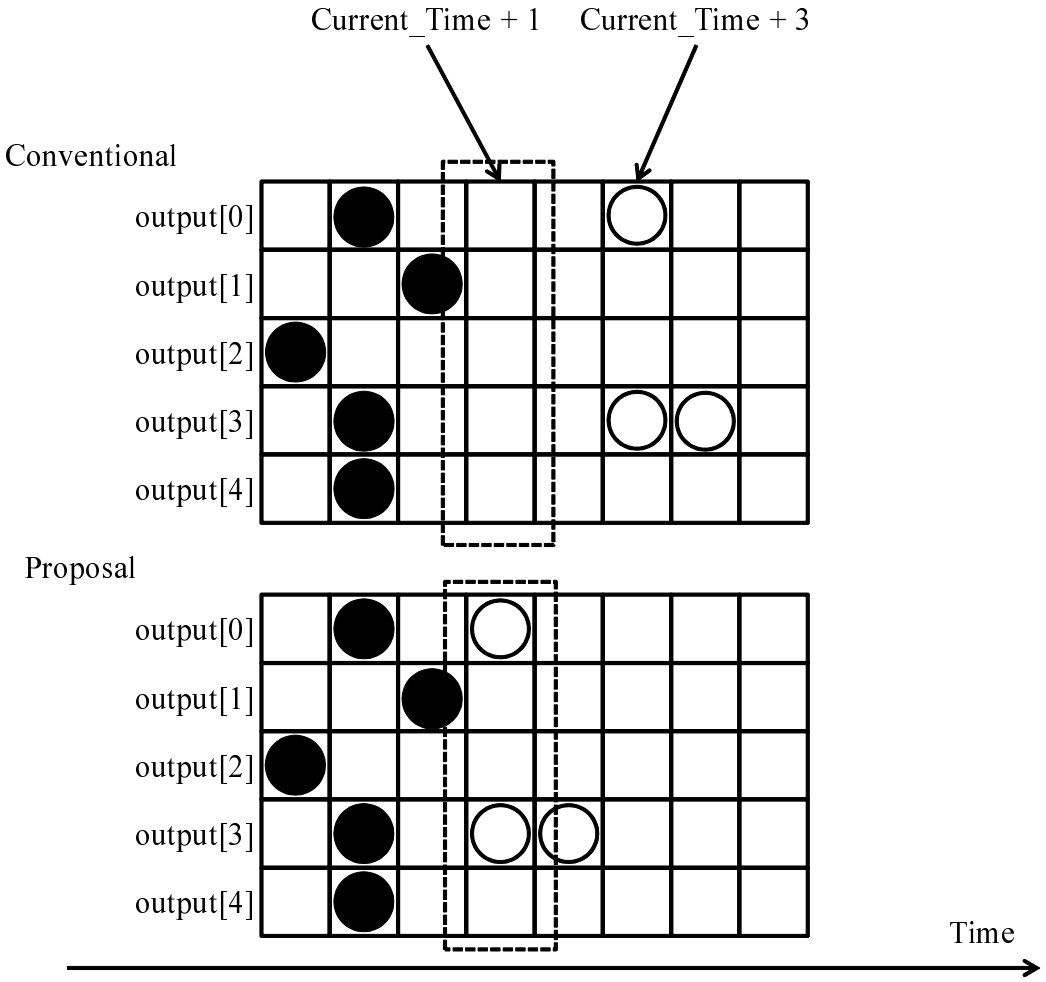


図 4.23 バイパス可能かどうかの判定例

較しても少ない遅延で行うことができると推測される．

以上の考察から，仮想チャネル割り当てを第 1 ステージで行うことによる遅延の増加はなく，この変更による動作周波数への影響はないと考えられる．

### 4.3.3 バイパス可否の判定

従来手法においてパイプラインのスキップが可能かどうかを判定するために，*Current_Time + 2* のタイムスタンプが発行されているかをチェックしていた．パイプラインを 2 段スキップするためには，判定のために *Current_Time + 1* のタイムスタンプを発行されたフリットが存在するかどうかという条件を用いる．TS ステージは図 4.10 に示した従来手法におけると同様の構成で，バイパスが可能な場合は，発行されるタイムスタンプ

（*Current_Time* + *3*）を −2 する点が異なる．

図 4.23 を用いて，具体的に説明する．図 4.23 は発行されたタイムスタンプを出力ポートごとに示している．図の黒丸はある出力ポート（*output*[*p*]）における *LAT*[*p*] を示しており，白丸が新たに発行されるタイムスタンプを示している．図の右側にある丸ほどタイムスタンプの値は大きくなる．図の例では，*output*[0] を要求するフリットが 1 つ，*output*[3] を要求するフリットが 2 つ存在している．Conventional は従来の DSB ルータにおけるタイムスタンプ，Proposal は改良したバイパス手法におけるタイムスタンプの様子である．

図 4.23 において従来の DSB ルータではいずれの *LAT*[*p*] よりも *Current_Time* + *3* の方が大きいため，新たに発行されるタイムスタンプは *Current_Time* + *3* 以上の値となっている．これに対して，バイパスする DSB ルータではすべての *LAT*[*p*] が *Current_Time* + *1* より小さいためバイパス可能と判定され，2 つのフリットに対して *Current_Time* + *1* のタイムスタンプが発行されている．

ここで，*output*[3] を要求する 2 つめのフリットは *Current_Time* + *2* のタイムスタンプが発行されている．これは，先行するフリットがパイプラインを 2 段スキップし，このフリットはパイプラインを 1 段スキップするという転送になる．どちらのフリットもミドルメモリには格納されず，バイパス回路を経由した転送が行われる．

### 4.3.4 バイパス方式の従来手法との比較

改良手法と従来手法の異なる点は次の 2 点である．

- 仮想チャネル割り当てを第 1 ステージでおこなう
- TS ステージで *Current_Time* + *1* のタイムスタンプを発行する

バイパスは従来手法と同様に 1 つめのクロスバーの手前から，2 つめのクロスバーの手前までの固定された回路を利用する．したがって，改良手法におけるバイパスのためのデータパスは，図 4.4 で示した従来手法のアーキテクチャと同じになる．

図 4.24 に，改良手法によりパイプラインを 2 段スキップする DSB ルータのパイプラインを示す．2 段スキップのために仮想チャネル割り当てを第 1 ステージに変更している．第 1 ステージでフリットはバイパス可能かどうか判定され，バイパス回路を経由することでミドルメモリを利用せずに直に第 2 クロスバーに送られる．フリットがパイプラインをバイパスする場合，TS ステージでは *Current_Time* + *1* の値がタイムスタンプとして与えられる．このタイムスタンプにより第 2 ステージでバイパス回路を使用するかを決定する．これにより，DSB ルータのパイプラインを 2 段削減することができ，隣接するルータ

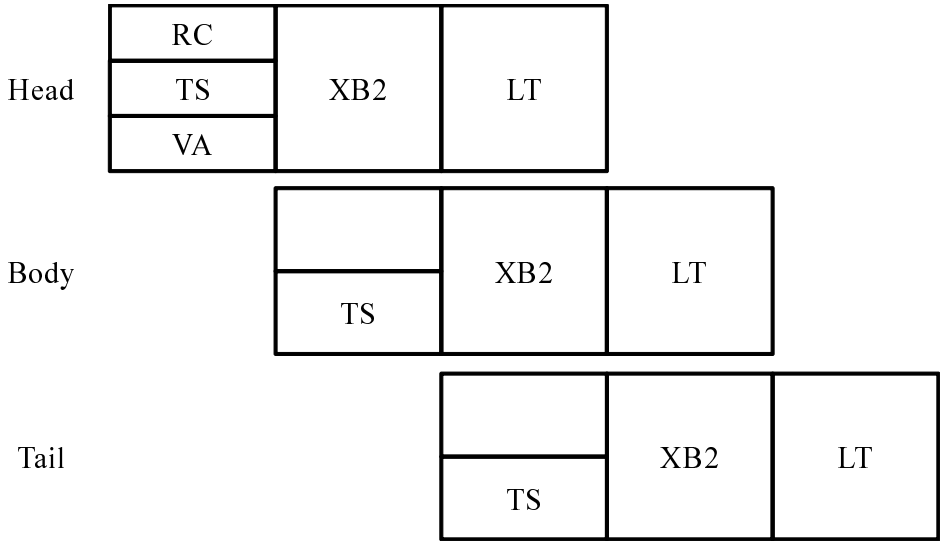


図 4.24 改良したパイプラインをバイパスする DSB ルータのパイプラインステージ

にフリットを 3 サイクルで転送することができる．

同じ出力ポートを要求する 2 つのフリットに同時にタイムスタンプを発行するとき，パイプラインが 2 段スキップ可能であれば *Current_Time + 1* と *Current_Time + 2* のタイムスタンプが発行される．このとき *Current_Time + 2* が発行されたフリットは，パイプラインの第 2 ステージでは何も処理されずに第 3 ステージでバイパスの回路に転送される．この転送は，従来手法と同じになり，パイプラインを 1 段スキップする 4 サイクルでのフリット転送になる．バイパス不可能な場合はオリジナルの DSB ルータと同じ動作となるため，高いスループットは維持される．

### 4.3.5 評価

評価は，従来手法の評価と同様に，独自に開発したフリットレベルのサイクルアキュレートなソフトウェアシミュレータを用いて行う．シミュレータに 3 段パイプラインの IBR，オリジナルの DSB ルータ，パイプラインをバイパスする DSB ルータ（従来手法），改良したパイプラインをバイパスする DSB ルータを実装し，スループットとレイテンシの比較から提案手法の有効性を示す．

評価環境は，8×8 の 2 次元メッシュトポロジ，XY 次元順ルーティングのネットワークを用い，パケット長は 4 フリットとする．IBR および DSB ルータの仮想チャネル数，バッファ量のパラメータは，従来手法の評価時に用いた設定（表 4.1）と同じ値を用いる．提案

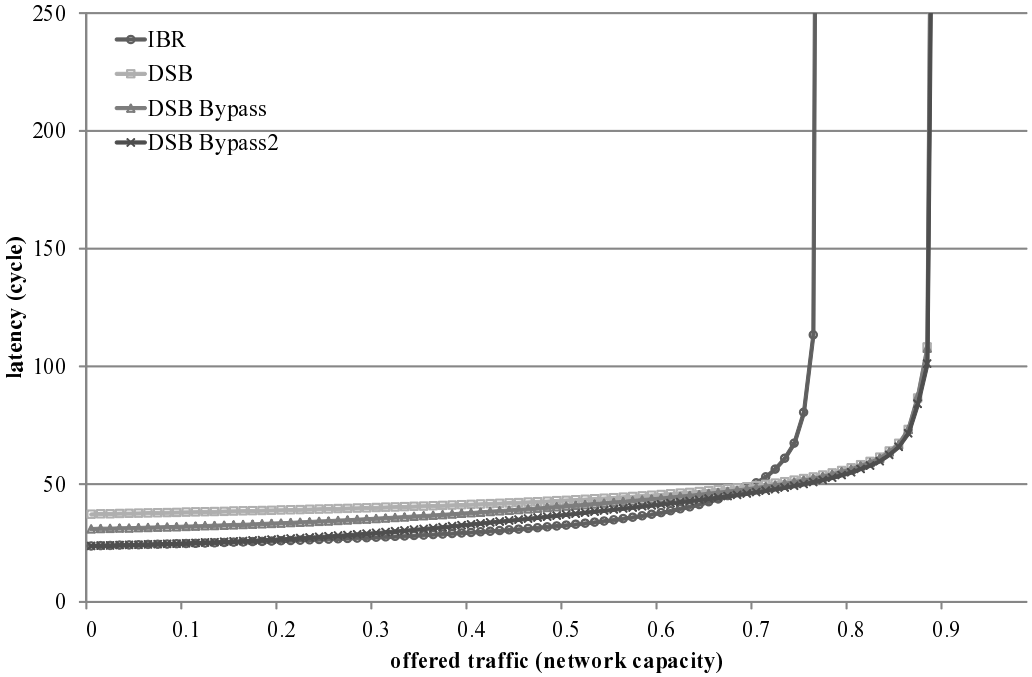


図 4.25 Random パターンの場合のレイテンシ

表 4.3 Zero Load Latency におけるレイテンシの削減率

| traffic pattern | rate(%) |
|---|---|
| uniform | 36.1 |
| complement | 37.1 |
| tornado | 37.0 |

手法を適用した DSB ルータはベースの DSB ルータ同じパラメータを用いる．通信パターンは，Random，Complement，Tornado の 3 種を用いる．シミュレーションは，ウォームアップとして 10,000 サイクルを実行した後に計測を開始し，100,000 サイクルまで実行する．

図 4.25，図 4.26，図 4.27 にパケットの注入レートを変化させたときの，IBR, DSB ルータ，従来のバイパスする DSB ルータ，バイパスを改良した DSB ルータの平均レイテンシを示す．バイパスを改良した DSB ルータのレイテンシは，注入レートが低い領域では，3 段パイプラインの IBR と同等のレイテンシを達成しており，バイパスの効果が確認できる．注入レートが高い領域では，DSB ルータとスループットは同じになり，バイパス回路を追加したことによるペナルティがないことが確認できる．

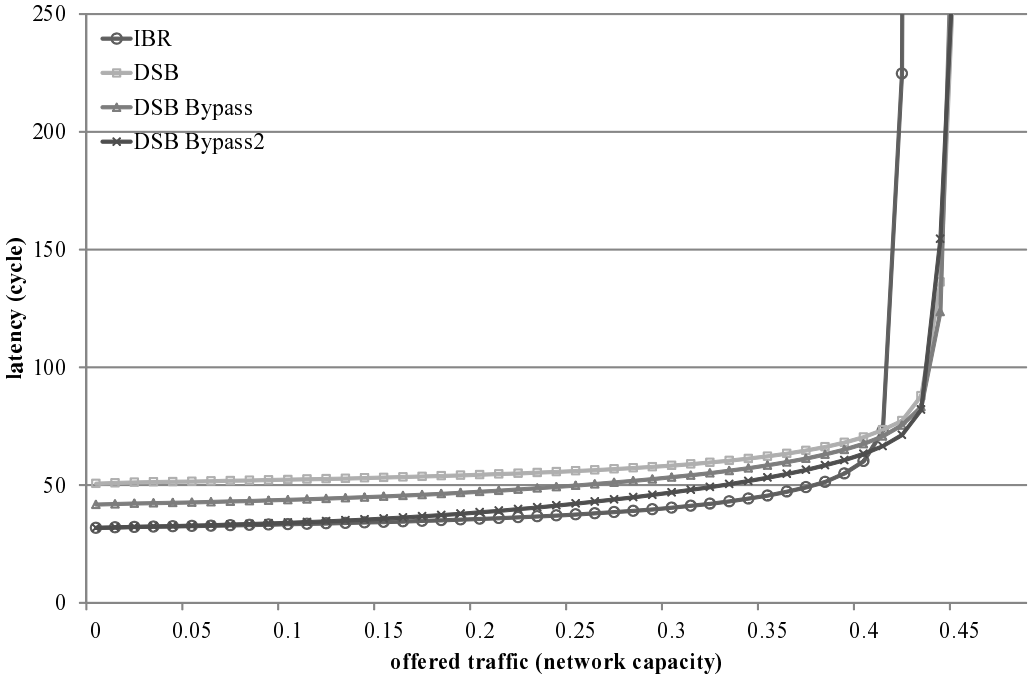


図 4.26 Complement パターンの場合のレイテンシ

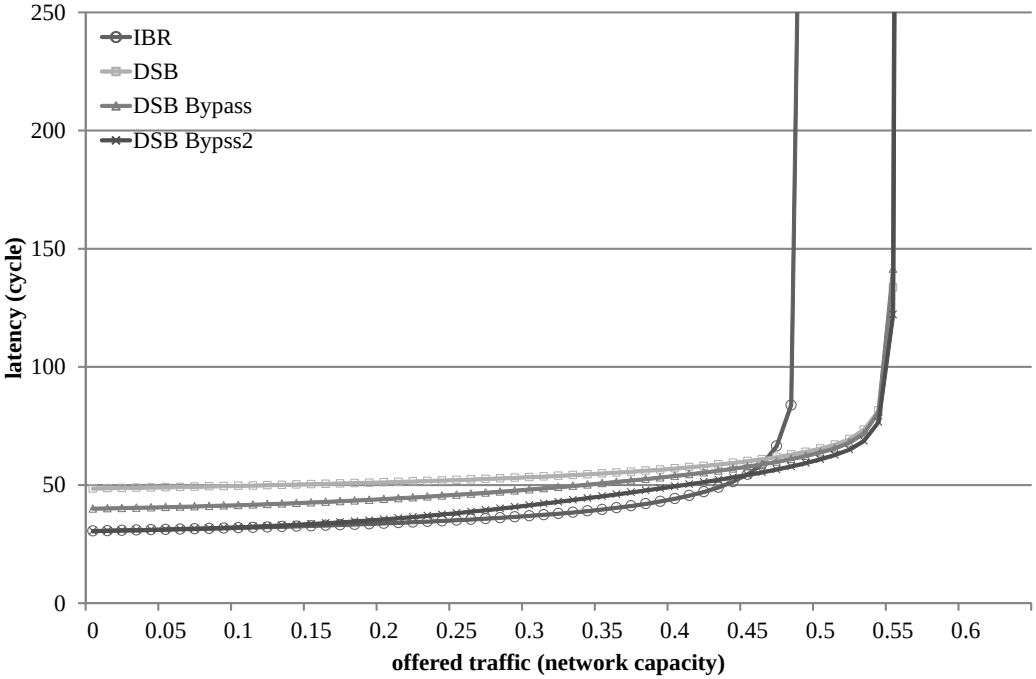


図 4.27 Tornado パターンの場合のレイテンシ

表 4.3 にバイパスを改良した DSB ルータのベースとなる DSB ルータに対するレイテンシの削減率を示す．Zero load latency における削減率は，最大で 37.1%，平均で 36.7% を達成している．

## 4.4 関連研究

### 4.4.1 RoShaQ

RoShaQ[73] は，DSB ルータと同様に入力バッファと共有バッファを持ち，クロスバーを 2 つ用いて構成される NoC ルータである．共有バッファを用いることでスループットの向上を図っている．また，クロスバー 2 つを経由することによるレイテンシ増加を解決するために，提案手法と似たバイパス手法が用いられている．

図 4.28 に，RoShaQ ルータのアーキテクチャを示す．このルータは入出力ポートでは仮想チャネルを用いない．入力バッファに格納されたフリットは，1 つ目のクロスバーを経由して割り当てられた Shared Queue に格納される．Shared Queue に格納されたフリットは，2 つ目のクロスバーを経由して割り当てられた出力ポートに転送される．DSB ルータでは共有バッファを割り当てる前に出力ポートを使用するタイミングをすべて決定するが，RoShaQ は入力バッファから Shared Queue，Shared Queue から出力ポートのそれぞれの転送のタイミングで割り当てが行われる．Shared Queue から出力ポートへの転送がない時は，バイパス経路を用い，入力バッファから Output XBAR に直接フリットを転送する．

図 4.29 に RoShaQ ルータのパイプラインを示す．まずは，バイパスを行わない場合の処理の流れを述べる．Routing Computation は，Next Hop Routing Computation を採用し，次のルータにおける経路計算を行う（NRC）．Output Port Allocation（OPA）で出力ポートの割り当てを行う．ただし，バイパスを行わない場合を示しているため，この割り当ては失敗となる．Shared Queue Allocation（SQA）で Shared Queue の割り当てを行う．Shared Queue が割り当てられたパケットは，Shared Queue Switch Traversal（SQST）ステージでクロスバーを通過し Shared Queue に格納される．Shared Queue に格納されたパケットは，Output Port Allocation（OPA）ステージで出力ポートの割り当てを行う．この割り当て処理では，第 1 ステージからのリクエストも同時に調停する．そして，第 3 ステージからのリクエストがある場合はそちらを優先する．出力ポートが割り当てられた Shared Queue のパケットは，Output Switch Traversal（OST）ステージでクロスバーを通過し出力ポートに転送される．

図 4.30 にバイパスを行う場合のパイプラインを示す．第 1 ステージにおける OPA に成

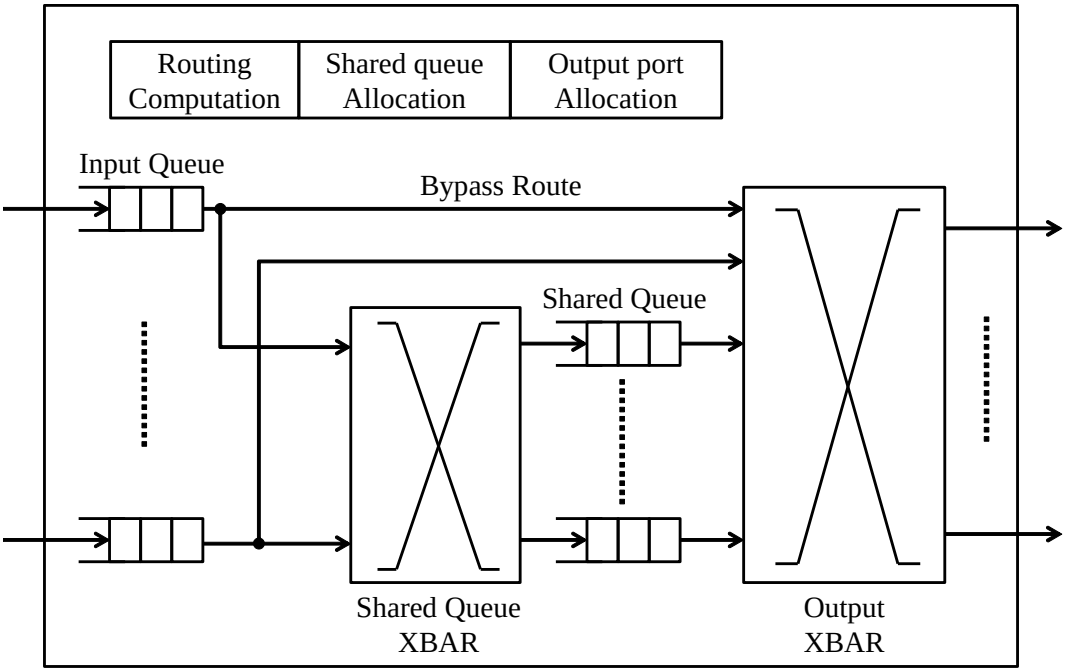


図 4.28 RoShaQ ルータのアーキテクチャ

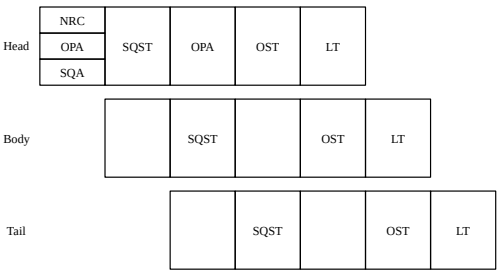


図 4.29 RoShaQ ルータのパイプライン

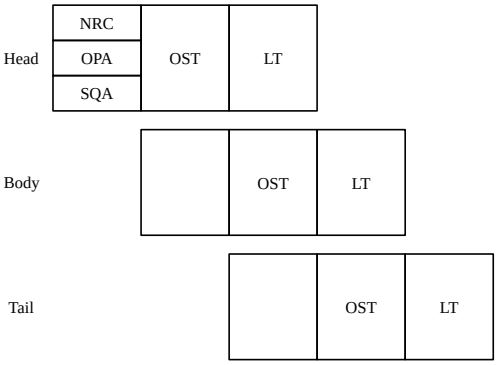


図 4.30 バイパスが成功した場合の RoShaQ ルータのパイプライン

功するとそのパケットはバイパス回路を経由して出力ポートに転送される．第 1 ステージで同時に処理される SQA に成功しても，OPA に成功しバイパス回路を用いる場合は Shared Queue の割り当てをキャンセルする．次に，OST ステージで入力バッファにあるフリットは直接 Output XBAR に送られ，出力ポートに転送される．バイパスを行うため第 2 ステージの SQST，第 3 ステージの OPA を行わずにフリットを送信することができ，レイテンシの削減を達成する．

入出力ポート数を $P$，共有バッファの数を $N$ としたとき，RoShaQ ルータの

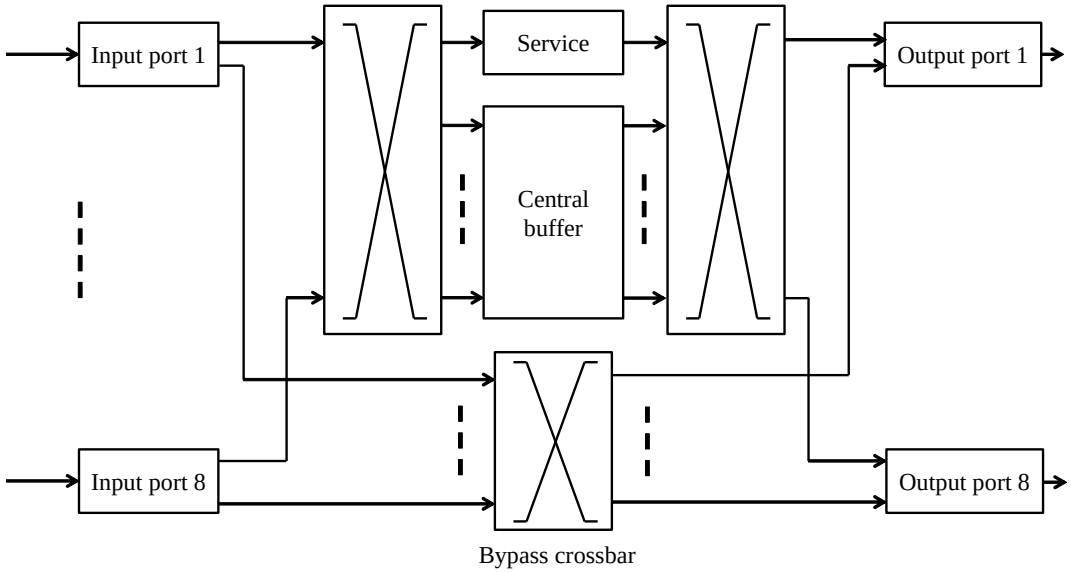


図 4.31 Colony ルータのアーキテクチャ

Shared Queue XBAR は DSB ルータの 1 つめのクロスバーと同じ *P:N* の構成となる．しかし，Output XBAR は *(P+N):P* の構成となっており，提案手法を適用した DSB ルータとは異なる構成になる．これは，同時に Shared Queue からと Input Queue からの転送を処理する必要があるためである．提案手法を適用した DSB ルータは，クロスバーの入力の直前にセレクタが必要となるが，2 つめのクロスバーを *N:P* の構成としている．これは，フリットのバイパスを共有バッファにフリットが格納されていない場合のみ行っており，共有バッファとバイパス経路からフリットの転送が同時に発生しないことが保証されるためである．

RoShaQ ルータにおけるバイパスと提案手法におけるバイパスは，パケット単位でバイパスを行うかフリット単位でバイパスを行うかの違いがある．RoShaQ ルータでは，パケット単位でバイパスの制御を行っており，第 1 ステージで OPA が成功した場合そのパケットのすべてのフリットはバイパス回路を経由して転送される．DSB ルータは仮想チャネルを用いるため，同時に同じ出力ポートの仮想チャネルが割り当てられるパケットが存在する．この時に，第 1 ステージで発行されるタイムスタンプが，あるパケットの一連のフリットに対して連続して与えられない場合がある．このため，提案手法を適用した DSB ルータでは，フリット単位でバイパスの制御を行っている．

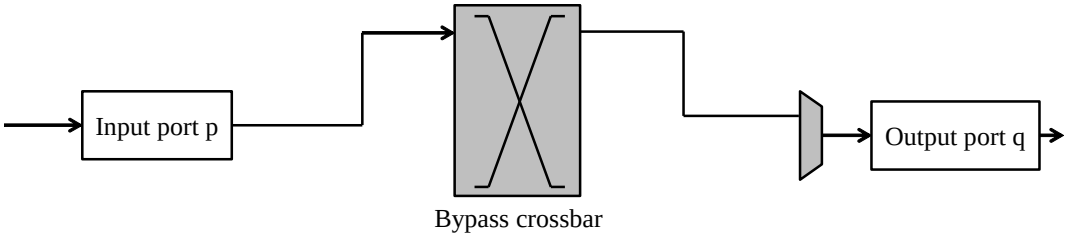


図 4.32 Colony ルータのバイパス経路

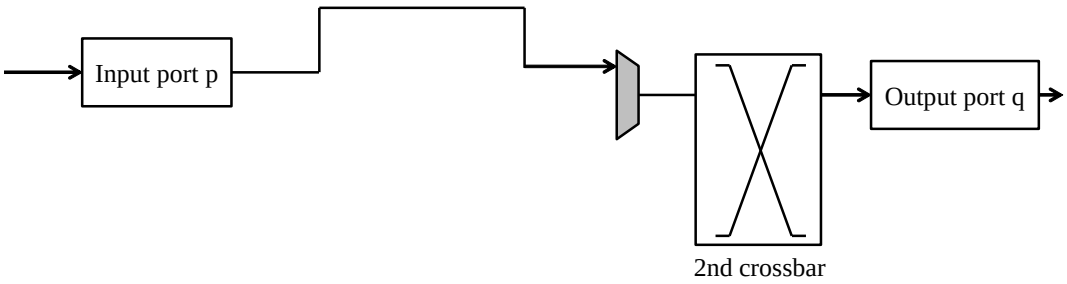


図 4.33 DSB ルータのバイパス経路

### 4.4.2 Colony Router

図 4.31 に Colony Router のアーキテクチャを示す．Colony router[74, 11] は，IBM RS/6000 SP システムというマルチコンピュータシステムのためのルータで，NoC ルータではないが DSB ルータと似たアーキテクチャを持つ．Central buffer と呼ばれる共有のバッファとバイパス用のクロスバーを用いてパケットを転送する．

入力ポートに格納されたパケットは，まずバイパス用のクロスバーに出力ポートのリクエストを出す．バイパス用のクロスバーが使用可能な場合，パケットはバイパス用のクロスバーを使用して隣接ルータに転送される．バイパス用のクロスバーが使用不可能な場合，パケットは Central buffer へのアクセス要求を出す．

このルータはバイパス用のクロスバーが使用不可の場合，入力されたパケットが隣接ルータに出力されるまでに 2 つのクロスバーを通過する．バイパス回路は，共有メモリへの読み書きを回避するように設けられている．パケットをバイパスする場合は，このバイパス回路上にあるクロスバーを経由して直接出力ポートに転送される．バイパスが可能な場合は，入力されたパケットが隣接ルータに出力されるまでに 1 つのクロスバーを通過する．

我々の提案手法と異なる点は，我々の方式ではバイパスするための新たなクロスバーを必要としない点である．図 4.32 に Colony ルータのバイパスのための回路，図 4.33 に提案手法である DSB ルータのバイパスのための回路を示す．それぞれ，入力ポート $p$ から出力ポート $q$ へバイパスする場合のデータパスを示している．灰色で示したクロスバーおよび，マルチプレクサがバイパスのために追加されたハードウェアである．

Colony ルータと提案手法の大きな違いは，新たにクロスバーを設けている点である．提案手法では，出力ポートにおけるフリットの衝突が解決されていることに着目し，バイパス経路を利用したフリットを適切な出力ポートに転送するために 2 つめのクロスバーを利用している．これにより，提案手法はより少ないハードウェア量でバイパス回路を構成することができ，優位性がある．

### 4.4.3 経路バイパスに関する研究

Express Virtual Channel[75] は，2 ホップ先までの方向が決定できる場合に，隣接しないルータ間で仮想的にバイパス経路を構成することで中継するルータにおける VA と SA ステージを削減する方式である．この手法は，局所性を持つ通信パターンにおいてはレイテンシ削減の効果がない．提案手法は経由する全てのルータにおいてレイテンシ削減の可能性がある．

Token flow Control[76] は，近隣のルータ間でリソース情報を共有し，混雑がない場合にバイパス経路を形成し隣接ルータへの転送レイテンシを削減する．この方式は常に近隣のルータ間でリソース情報を共有する必要がある．提案手法は，近隣のルータの情報を必要とせずにバイパスによりレイテンシを削減することができる．

Mad-postman[77] は，次元順ルーティングの規則性に着目し，ビットシリアル転送を行う場合にパケットヘッダが全て到着する前に隣接するルータに転送を開始することでレイテンシを削減する．文献 [78] は，通信経路の使用頻度に着目し，頻繁に使われる経路にバイパスを構成する．文献 [79] は，あらかじめ経路を固定した上でデータを転送する．つまり，NoC ルータを利用してサーキットスイッチング方式の転送を仮想的に構成することでレイテンシを削減する．これらのバイパス方式は転送ルートやルーティング方式などに依存するが，提案手法のバイパス方式は転送ルートやルーティング方式に依存せずに適用することができる．

# 第 5 章

# メニーコアプロセッサの設計

研究・教育を目的としたメニーコアアーキテクチャとして，M-Core[80, 15] アーキテクチャが提案されている．本章では，ArchHDL を用いて M-Core をベースとしたメニーコアプロセッサ M-Core Advanced を設計する．特に，メニーコアの重要な要素である Network-on-Chip（NoC) ルータを ArchHDL を利用した評価から策定する．

## 5.1 M-Core アーキテクチャ

M-Core アーキテクチャ [80, 15] は，メニーコアプロセッサの研究・教育を支援するための実用的な基盤環境として設計されたスクラッチパッドメモリ型のタイルアーキテクチャである．理解しやすいアーキテクチャを目指しており，シンプルな構成になっている．

図 5.1 に，M-Core アーキテクチャを示す．M-Core のノードには，MIPS プロセッサを搭載する計算ノード（Comp. Node），オフチップのメモリコントローラであるメモリノード（Mem. Node），ルータのみを搭載するパスノードがある．これらのノードが NoC ルータで 2 次元メッシュ状に接続されている．M-Core アーキテクチャは，Cell/B.E. の PPE, SPE などの Processing Element が MIPS アーキテクチャに変更され，リング型の集中管理のネットワークが 2 次元メッシュの NoC に変更されたものと考えると理解しやすい．

計算ノードは，32 ビットの MIPS32 命令セットを実行するコアを搭載する．初期の MIPS プロセッサ [81, 67] は，インオーダーの 5 段パイプラインのプロセッサであるが，1 サイクルにほぼ 1 命令を実行する効率のよいアーキテクチャである．このため，ハードウェアの複雑化を避けるために M-Core アーキテクチャの計算ノードはシングルサイクルの PE を搭載している．

各計算ノードは独立したメモリ空間を持つ．命令，データはノードメモリにロードして

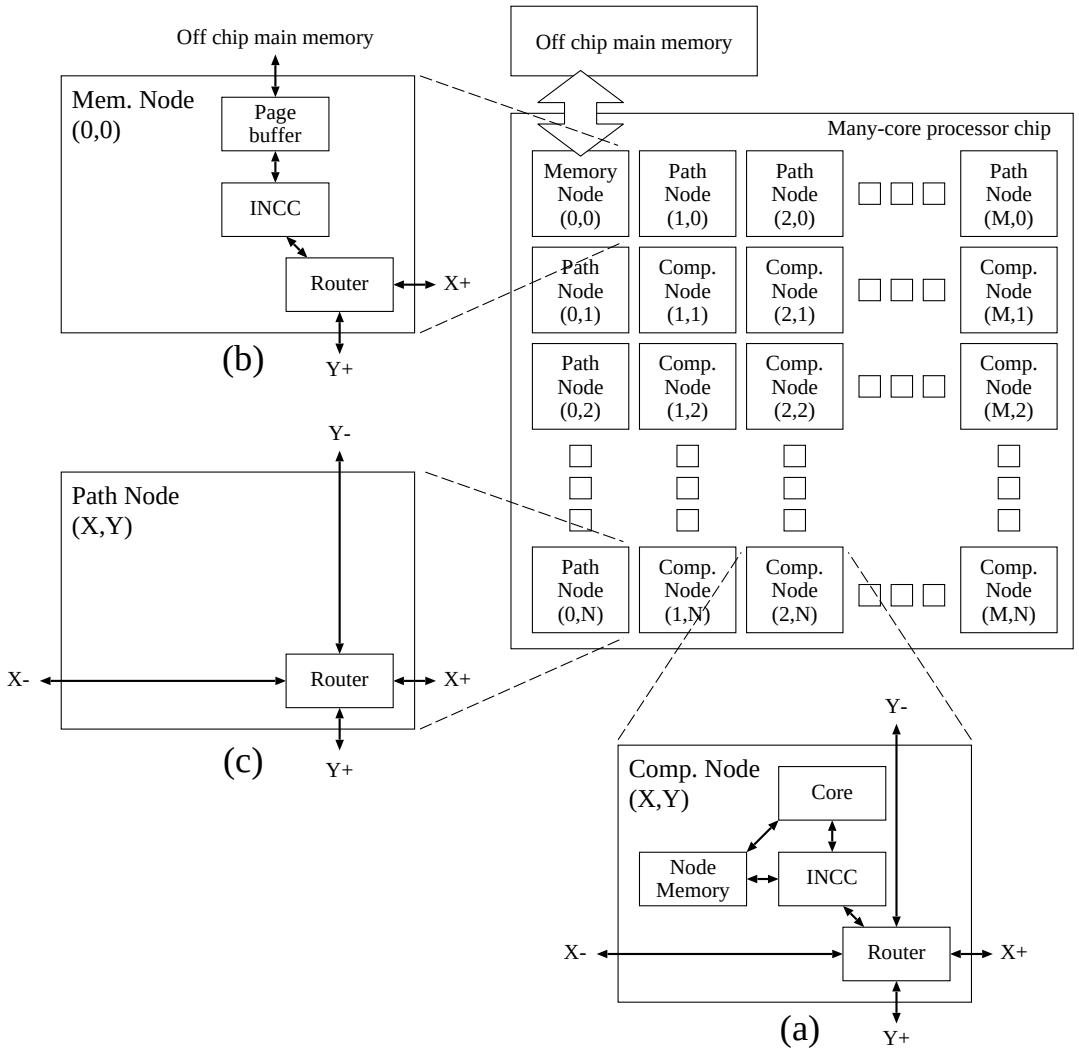


図 5.1 M-Core アーキテクチャ

実行，参照する．ノードメモリの容量は 512KB としている．ノードメモリへのアクセスは，PE からの命令フェッチ，PE からのロード・ストア命令，ネットワークからリード・ライトの 3 種類がある．これらのアクセスが 1 サイクルで同時に発生することを想定し，3 リード・2 ライトのメモリとなっている．

ノード間のデータ通信はノードメモリ間の DMA 転送により行われる．DMA 転送を制御するのが計算ノード，メモリノードに搭載されている Inter Node Communication Controller（INCC）である．INCC は，メモリマップド IO により PE から DMA 転送の情報を受け取るとローカルメモリを参照し，パケットを生成し，NoC 経由でデータを送信する．また，受信したパケットにしたがってローカルメモリにデータを書き込む．

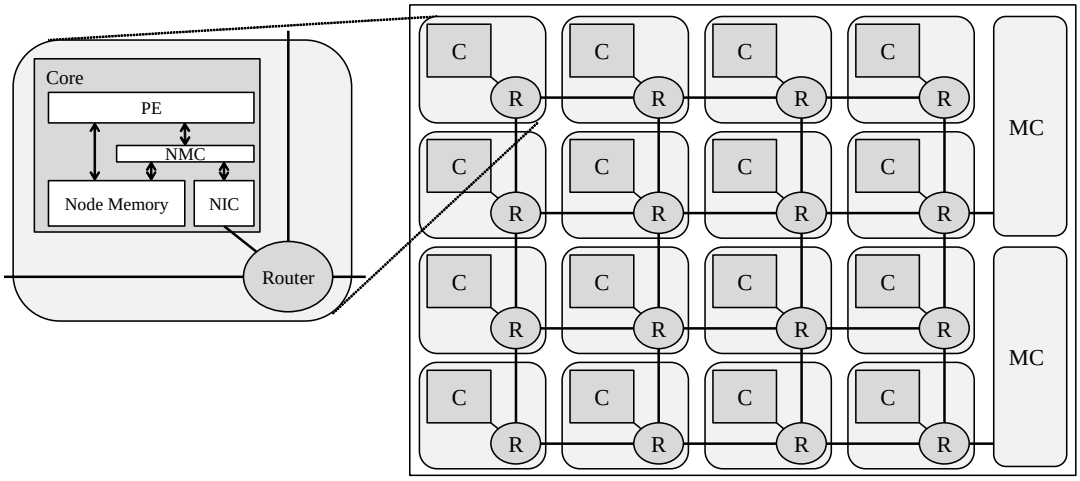


図 5.2　16 コア構成の M-Core Advanced

NoC ルータは，仮想チャネルを持たない Input-Buffered ルータである．典型的な仮想チャネルなし Input-Buffered ルータは，4 段程度のパイプライン処理によりフリットを転送するが，M-Core アーキテクチャのルータは，ハードウェアの複雑化を避けるためにフリット転送に関わる処理すべてを 1 サイクルで行うアーキテクチャを採用している．入出力チャネルのデータ幅は，32 ビットとなっている．ルーティングは XY 次元順ルーティング，フロー制御は Xon/Xoff を採用している．

## 5.2　メニーコアプロセッサ M-Core Advanced の設計

M-Core アーキテクチャは，メニーコアアーキテクチャの研究のみならず，教育にも利用することを目的として設計されており，アーキテクチャの理解を妨げる可能性のあるハードウェアの複雑化を避けている．そのため，FPGA やチップとして実装しようとすると実現困難なハードウェアを持つアーキテクチャとなっている．ここでは，M-Core アーキテクチャを FPGA やチップに実装できるように改良したメニーコアプロセッサ M-Core Advanced を設計する．

### 5.2.1　M-Core Advanced

図 5.2 に 16 コアの M-Core Advanced のアーキテクチャを示す．ここでは，M-Core アーキテクチャからの変更点を中心に M-Core Advanced の各要素について述べる．

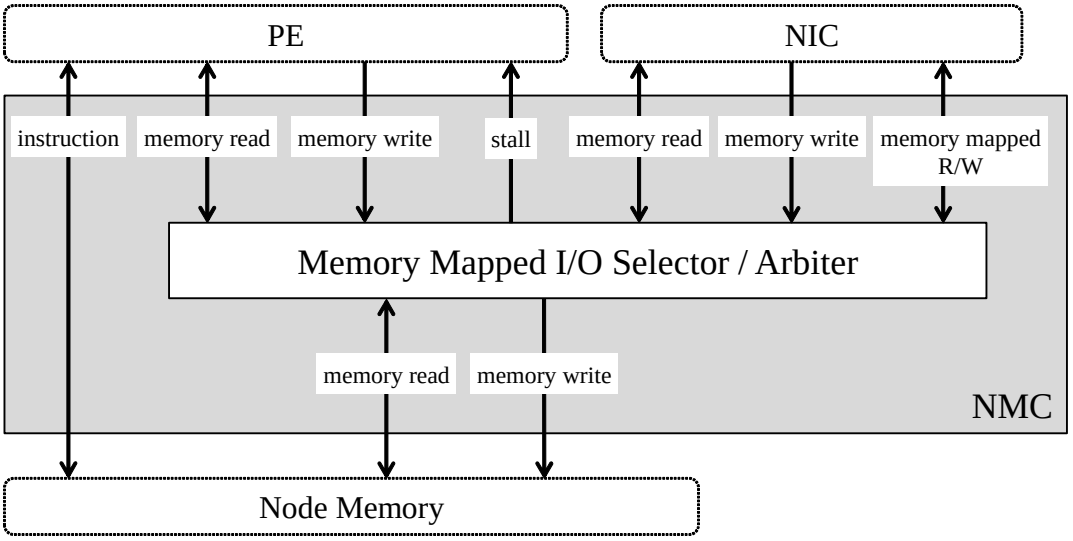


図 5.3　ノードメモリコントローラ

**パスノードの廃止**

M-Core では，メモリノードと計算ノードをメッシュ状に接続するために，ルータのみが搭載されているパスノードが導入されている．これは，メモリアクセスのための通信において XY 次元順ルーティングを維持するために必要なノードである．M-Core Advanced では，図 5.2 に示すように，メモリコントローラをチップの端にある特定のコアに接続する Intel SCC のような配置とする．このような配置にすることで，XY 次元順ルーティングではメモリコントローラに通信が届かなくなるコアができてしまうが，メモリコントローラが接続されているルータにおいてメモリアクセスのパケットを適切に制御することで解決することができる．このような設計により，パスノードを廃止することができる．

**ノードメモリのポート数の削減とノードメモリコントローラの導入**

M-Core では，PE と DMA によるアクセスを同時に処理するために 3 リード・2 ライトという現実的ではないポート数のメモリが必要となる．M-Core Advanced では，ノードメモリコントローラ（NMC）においてノードメモリへのアクセスを調停し，ノードメモリのポートを 2 リード・1 ライトに削減する．NMC の導入に伴い，M-Core では INCC と呼んでいるパケット生成のためのハードウェアをネットワークインタフェースコントローラ（NIC）と呼ぶ．

図 5.3 に M-Core Advanced のノードメモリコントローラ（NMC）のブロック図を示す．

ノードメモリは，命令のためにリードポートを 1 つ，データのためにリードとライトを 1 ポートずつ利用できる．命令フェッチのためのメモリアクセス（Instruction）は，直に PE とノードメモリを接続している．

ノードメモリへのデータの参照は，PE からのロード・ストア，DMA による読み出し書き込みのアクセスとなる．DMA によるメモリアクセスが発生している間に同時に PE からのアクセスが発生した場合に，NMC により調停を行い，PE をストールさせる．これにより，データのアクセスに必要なポートを 1 リード・1 ライトとすることができる．PE および NIC のためのデータ読み出し（memory read），データ書き込み（memory write）は，アドレスによりメモリマップド I/O へのアクセスかノードメモリへのアクセスかを判定するセレクタおよび，PE と NIC のノードメモリ参照を調停するアービタを介して PE，NIC，ノードメモリにそれぞれ接続されている．NIC によるノードメモリのアクセスは，調停により優先され，PE によるアクセスに妨げられることなく行われる．メモリアクセスの調停の結果，PE をストールさせるための信号（stall）は，NMC から PE に接続されている．PE からのアクセスがメモリマップド I/O の場合は，それらのアドレスやデータはメモリマップド I/O のための信号線（memory mapped R/W）により，NIC に伝達される．

#### 仮想チャネルルータの導入

NoC はメニーコアアーキテクチャにおいて性能を決定する重要な要素である．M-Core では複雑さを回避するために，仮想チャネルのないフリットを 1 サイクルで転送できるルータを搭載している．M-Core Advanced では，より現実的な実装のため多くのメニーコアアーキテクチャが採用するように仮想チャネルルータを導入する．ルータは，前章にて提案したパイプラインをバイパスする Distributed Shared-buffer ルータを採用する．NoC に関する具体的なアーキテクチャは，パラメータサーベイの結果から決定し詳細は後述する．

#### ネットワークインタフェースコントローラ

M-Core において INCC と呼ばれているパケット生成，受信の処理を行うハードウェアを仮想チャネルに対応するために変更する．このハードウェアは，M-Core Advanced ではネットワークインタフェースコントローラ（NIC）と呼ぶ．

図 5.4 に，M-Core Advanced のネットワークインタフェースコントローラのブロック図を示す．NIC の機能は，パケットの送信と受信に大別できる．

パケットの送信は，PE から送られた DMA 情報からパケットを生成する部分と，生成されたパケットをフリット単位で NoC に送信する部分からなる．PE は，メモリマップド I/O

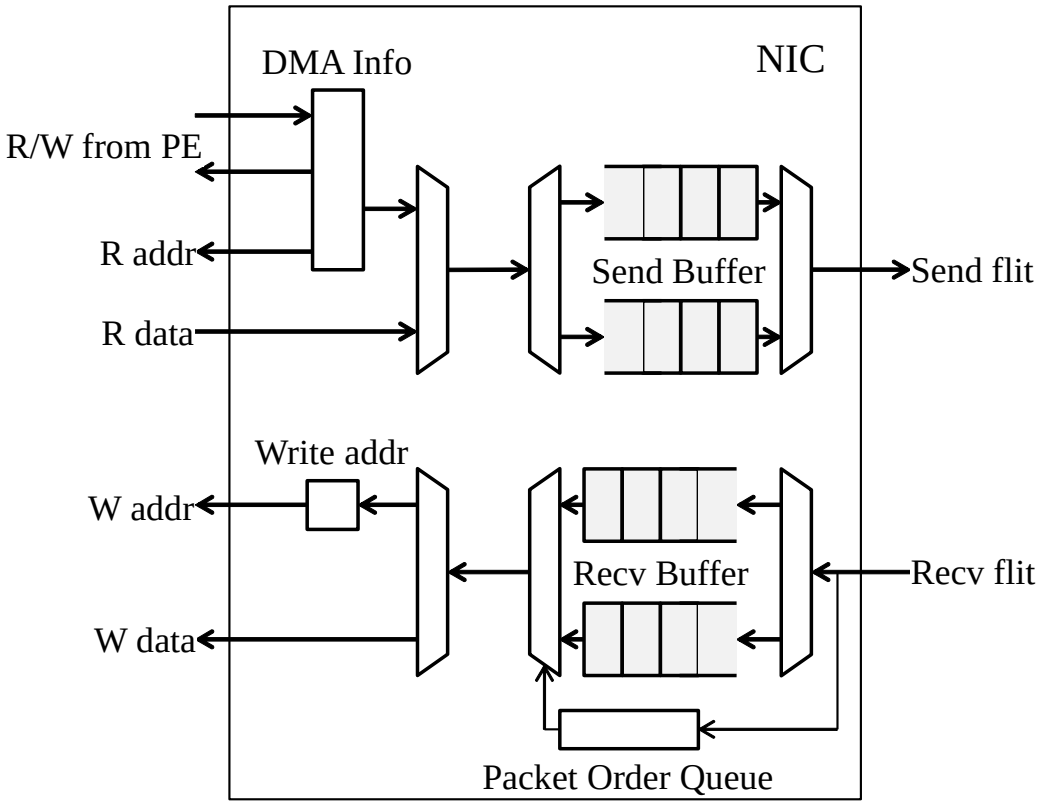


図 5.4　ネットワークインタフェースコントローラ

により DMA 転送に必要な送信先 ID，送信先アドレス，送信元アドレス，転送サイズの情報を NIC に転送する（R/W from PE）．送られた DMA 情報は一旦レジスタ（DMA Info）に保存され，すべての情報がそろったところでパケットの生成を開始する．パケットの生成はフリット単位で毎サイクル行われ，ヘッダ情報やメモリから読み出されたデータ（R addr, R data）からフリットを生成しバッファ（Send Buffer）に格納する．フリットのデータ幅は M-Core と同じく 32 ビットである．バッファは，仮想チャネルごとに分けられており，アーキテクチャによって決められた 1 パケット分のフリットを保持できるサイズとなっている．転送サイズが最大のパケット長より大きい場合は，複数のパケットに分割される．仮想チャネルは，ラウンドロビンで 0 番から順に使用する．

パケットを送信する部分では，フリットが生成され次第，NoC にそのフリットを送信する．送信は，パケット単位で行われる．したがって，次のパケット生成が開始されフリットがバッファに格納されても，先に生成されたパケットがすべて送信されるまでは後から生成されたパケットが送信されることはない．

パケットの受信は，NoC から送られてきたフリットをバッファに格納する部分と，パ

ケットの受信順に従いノードメモリにデータを書き込む部分からなる．NoC から送られてきたフリットは，その仮想チャネル番号にしたがって適切にバッファ（Recv Buffer）に格納される．Recv Buffer は，仮想チャネルごとに分けられ，1 パケットを保持できるサイズとなっている．パケットの到着順は Packet Order Queue により管理され，ノードメモリへのデータの書き込みはパケットの到着順に処理される．データを書き込む処理はフリット単位で行われる．毎サイクル Recv Buffer よりフリットを読み出し，ヘッダ情報からの書き込みアドレスを取得し，ノードメモリへデータを書き込む.

### 5.2.2 API

M-Core Advanced において並列アプリケーション開発のためのソフトウェアライブラリを開発する．ライブラリには，動作しているコア数を取得する関数や自身のコア ID を取得する関数などの基本的な関数，DMA 転送を指示する関数などが実装されている．DMA 転送は，関数に宛先のコア ID，データの宛先アドレス，データの送信元アドレス，データサイズを渡すことで行うことができる．

DMA 転送において，データの受け手は自身のノードメモリにデータが書き込まれるタイミングを知ることができないため，データが書き込まれる予定のアドレスを参照し，ポーリングによりデータの受信を確認する必要がある．これは，DMA 転送を指示する関数を用いたデータ転送では，ブロッキング転送のみを用いて並列アプリケーションの開発を行わなければならないことを意味する．

小規模な並列アプリケーションはブロッキング転送のみで記述することができるが，大規模なアプリケーション開発ではノンブロッキング転送によりデータ転送を記述する場合がある．そのために，MPI[82] のメッセージパッシングモデルによるデータ通信に相当するライブラリを開発する．

MPI では，送信側で send，受信側で receive と明示的にデータの送受信を記述する．また，ノンブロッキング転送が可能で，いくつかの send，receive を記述した後，すべての通信の完了を待つ（wait）といった処理ができる．これにより柔軟にデータ通信を記述でき，様々な並列アプリケーションの開発が容易になる．

文献 [83] において，DMA のみでデータの送信がなされる環境における MPI の実装が報告されている．この実装を参考に，M-Core Advanced において，すべてソフトウェアで制御する MPI のデータ転送に相当する関数を実装している．メッセージパッシングモデルによるデータ転送実現のために，1 つのデータ転送に 3 ウェイ・ハンドシェイクを使用している．そのため，DMA 転送を指示する関数によるデータ転送と比較して，1 回の転送にか

かる遅延が大きくなるが，並列アプリケーション開発においては有用なライブラリとなっている．

## 5.3 M-Core Advanced の評価

M-Core Advanced を ArchHDL で記述し，並列アプリケーションによる性能評価を行う．まず，基本的な通信アプリケーションを用いたパラメータサーベイにより，M-Core Advanced の Network-on-Chip（NoC）の詳細を決定する．そして，並列アプリケーションを用いた評価を行う．

### 5.3.1 Network-on-Chip のパラメータの策定

並列アプリケーションにおいてよく用いられる 1 対 1 通信，1 対全通信，全対 1 通信，全対全通信 [84] のレイテンシをシミュレーションにより測定し，NoC のパラメータを策定する．すべての通信は，開発した MPI 相当の通信を行うライブラリを用いて実装する．また，1 対全通信，全対 1 通信，全対全通信は 2 次元メッシュに最適化した通信パターンで実装している．1 つのコアが送信するデータはすべての通信パターンで 1KB とする．

シミュレーションには, 5 段パイプラインの Input-Buffered 仮想チャネルルータ（IBR_5），3 段パイプラインの IBR（IBR_3），Distributed Shared-Buffer Router（DSB），2 段バイパスする DSB ルータ（DSB_2）を用いる．これらのルータの総バッファ量を等しく 160 フリット分（1 ポートあたり 32 フリット）とし，仮想チャネルの本数（それに伴うバッファ量），パケット長を変更してシミュレーションを行う．

表 5.1 にシミュレーションを行うルータの仮想チャネル数，入力バッファのエントリ数，DSB ルータの場合はさらにミドルメモリの本数，ミドルメモリのエントリ数をまとめる．これらの設定で，パケット長を 4，8、16 としたときの各通信パターンのレイテンシを計測する．ただし，入力バッファのエントリ数がパケット長より長い場合，例えばバッファのエントリ数が 8 でパケット長 4 の場合，実装によりレイテンシに影響がないためエントリ数と同じ長さのパケット長でのみシミュレーションを行う．また，ノード数は 4×4 の 16 ノード，8×8 の 64 ノード，16×16 の 256 ノードの 3 つのサイズでシミュレーションを行う．

M-Core Advanced で，パケットはヘッダ情報に 3 フリットを必要とする．したがって，ペイロードはパケット長が 4 の場合は 1 フリット分の 4 バイト，パケット長が 8 の場合は 20 バイト，パケット長が 16 の場合は 52 バイトとなる．つまり，1KB のデータを送信す

表 5.1 シミュレーションを行うルータの設定

| | VCs | Flits/VC | MM | Flits/MM |
|---|---|---|---|---|
| IBR 5stage | 2 | 16 | - | - |
| | 4 | 8 | - | - |
| | 8 | 4 | - | - |
| IBR 3stage | 2 | 16 | - | - |
| | 4 | 8 | - | - |
| | 8 | 4 | - | - |
| DSB | 4 | 4 | 5 | 16 |
| DSB Bypass | 4 | 4 | 5 | 16 |

るために，パケット長 4 の場合は 256 パケット，パケット長が 8 の場合は 52 パケット，パケット長が 16 の場合は 20 パケットが送信される．

図 5.5 に 1 対 1 通信を行ったときのノード数，パケット長ごとのレイテンシを示す．横軸は，ノード数およびパケット長でまとめて表示している．縦軸はレイテンシで，実行サイクル数を表している．

グラフにおいて，パケット長が 4 のときのルータの設定でバッファサイズが 8 と 16 の場合など，値がない箇所がある．これは，バッファに空きがあっても複数のパケットが 1 つのバッファ入らないような制御を行っているため，先に述べたとおりパケット長よりも長いバッファの設定は，パケット長と同じ長さのバッファの設定と同じ性能となり，ルータの総バッファ量を同じにして比較する上で公平な評価とならないためである．以降に示すグラフもこの理由から，グラフに値がない箇所がある．

図 5.6 に，それぞれの項目について，パイプラインバイパスを行う DSB ルータのレイテンシを 1 としたときの相対性能を示す．すなわち，それぞれのノード数，パケット長ごとのまとまりのうち，DSB_2 で示される値を基準とした性能を表している．1 対 1 通信において，ノード数の増加にともない，レイテンシが増加している様子が分かる．それぞれの設定のレイテンシにおいて，3 段パイプラインの IBR（IBR_3）が最もレイテンシが低いことが分かる．パイプラインバイパスを行う DSB ルータは，仮想チャネル数が 8 の IBR_3 と同等の性能を示している．仮想チャネル数が 8 の IBR と DSB ルータの比較は，文献 [14] で行われている評価と同じ条件であり，想定通りの結果が得られている．

図 5.7 に 1 対全通信を行ったときのノード数，パケット長ごとのレイテンシを示す．また，図 5.8 に，パイプラインバイパスを行う DSB ルータのレイテンシを 1 としたときの相

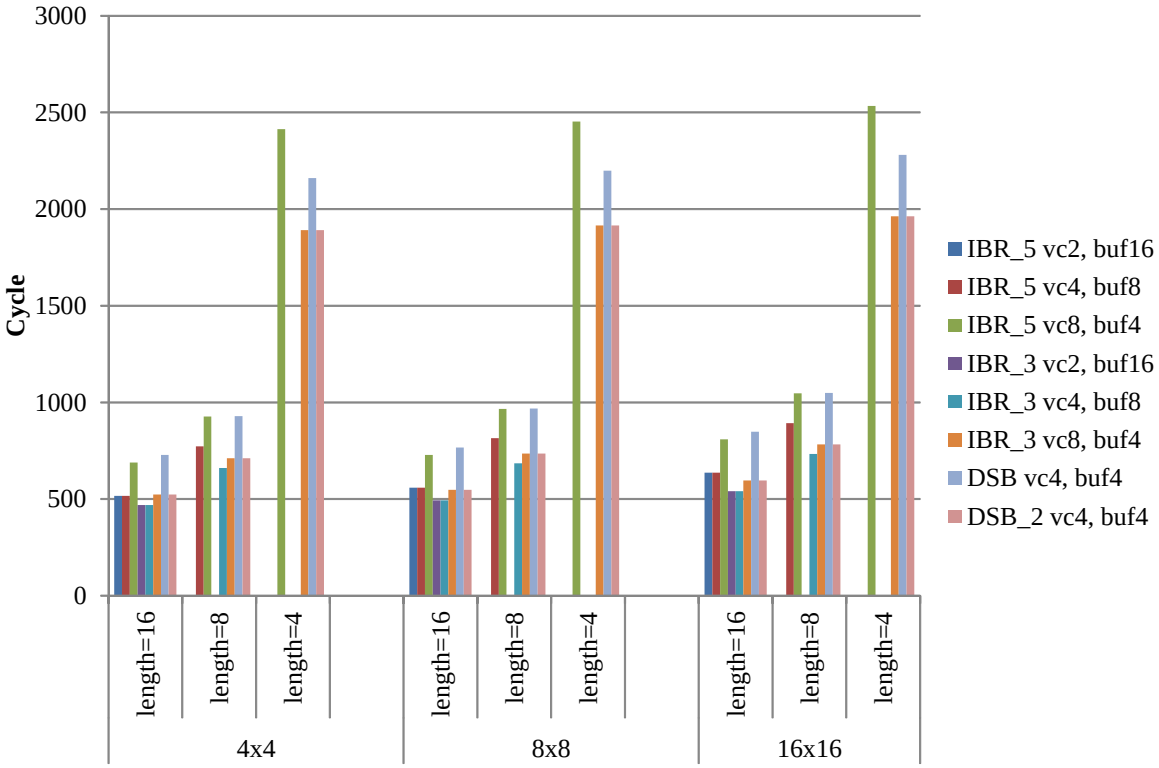


図 5.5 1 対 1 通信のノード数，パケット長ごとのレイテンシ

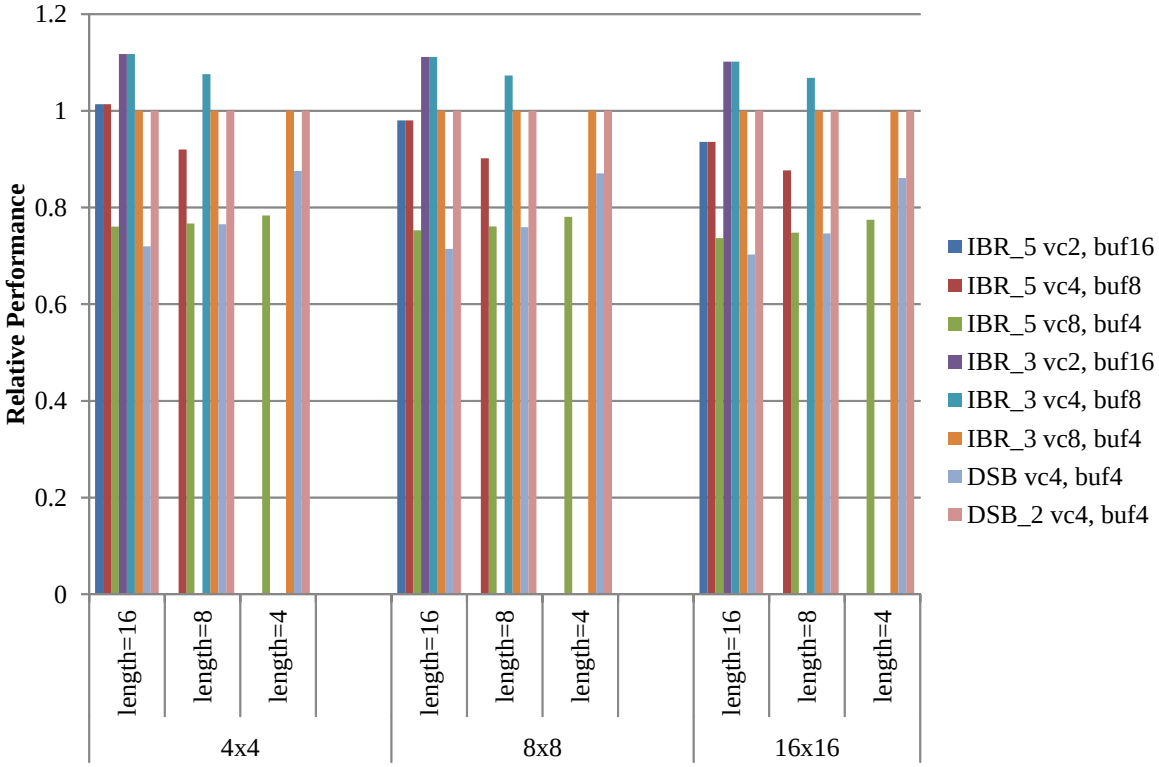


図 5.6 1 対 1 通信のノード数，パケット長ごとのレイテンシの相対性能

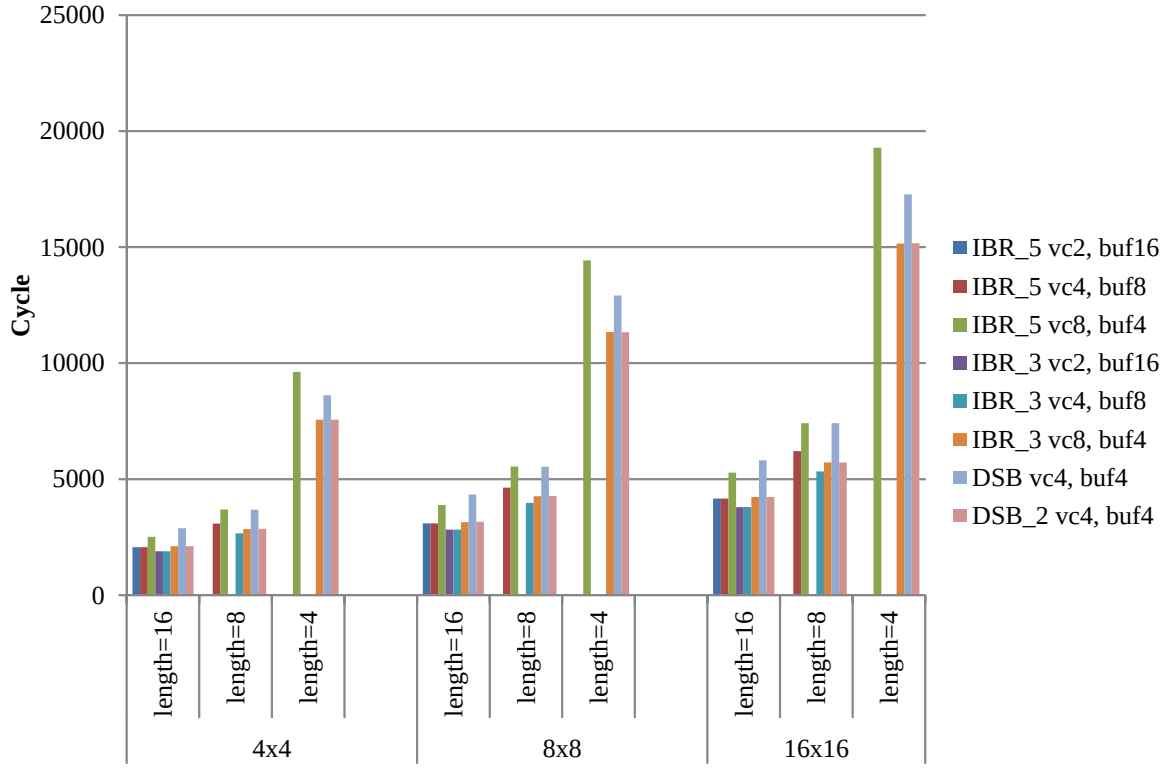


図 5.7 1 対全通信のノード数，パケット長ごとのレイテンシ

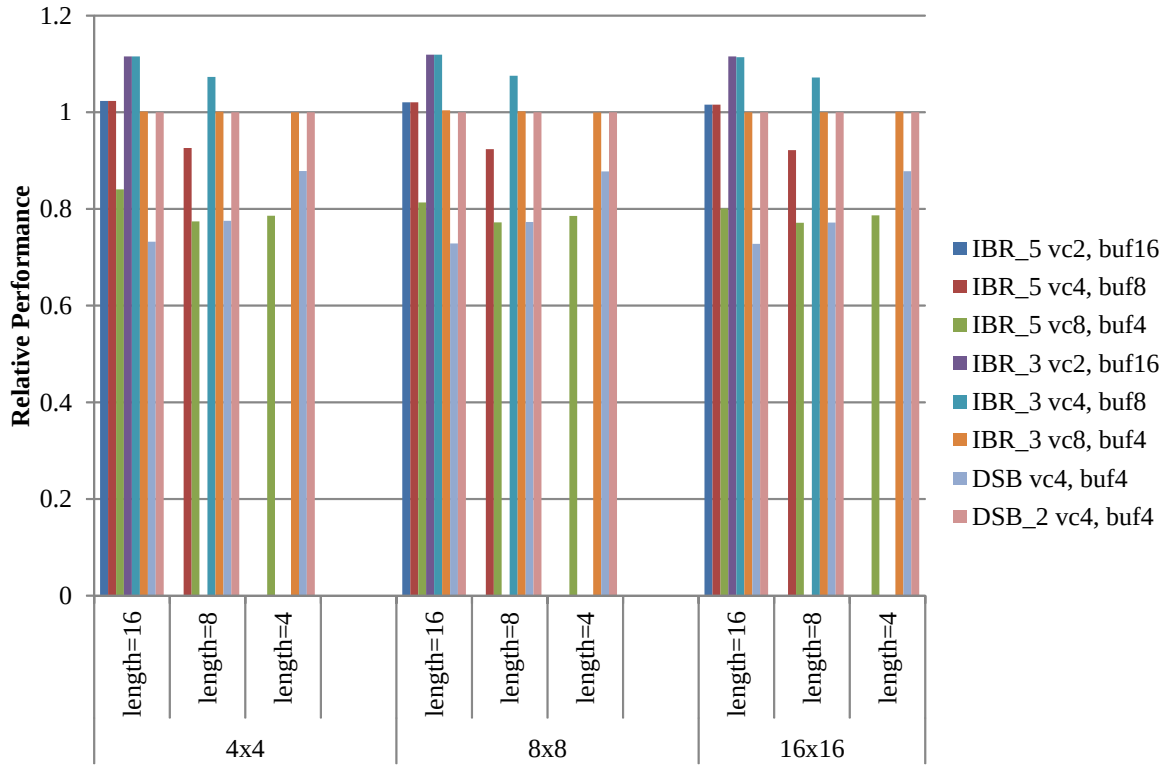


図 5.8 1 対全通信のノード数，パケット長ごとのレイテンシの相対性能

対性能を示す．縦軸，横軸は 1 対 1 通信で示したグラフと同様である．

2 次元メッシュに最適化された通信パターンの 1 対全通信では，パケットの衝突が発生しないため，レイテンシの性能が表れる評価結果となる．それぞれのグラフの傾向は，1 対 1 通信と同様になり，想定通りの結果が得られている．

図 5.9 に全対 1 通信を行ったときのノード数，パケット長ごとのレイテンシを示す．また，図 5.10 に，パイプラインバイパスを行う DSB ルータのレイテンシを 1 としたときの相対性能を示す．縦軸，横軸は 1 対 1 通信で示したグラフと同様である．

2 次元メッシュに最適化された通信パターンの全対 1 通信では，パケットの衝突が発生しないため，レイテンシの性能が表れる評価結果となる．それぞれのグラフの傾向は，1 対 1 通信と同様になり，想定通りの結果が得られている．

図 5.11 に全対全通信を行ったときのノード数，パケット長ごとのレイテンシを示す．また，図 5.12 に，パイプラインバイパスを行う DSB ルータのレイテンシを 1 としたときの相対性能を示す．縦軸，横軸は 1 対 1 通信で示したグラフと同様である．

全対全通信は，2 次元メッシュに最適化されたパターンにおいて，すべてのノードが同時に通信を行うためスループットの性能が表れる評価結果となる．パケット長を 8 とした評価において，これまで示したレイテンシの性能が表れていた評価では，パイプラインバイパスを行う DSB ルータの性能は 3 段パイプラインの IBR の最もよい性能を下回っていた．しかし，全対全通信の評価においては，パケット長 8 においてパイプラインバイパスを行う DSB ルータが 3 段パイプラインの IBR と同等の性能を示す結果となった．

1KB のデータ送信に必要なパケット数を考慮して，すべてのノード数の場合において，パケット長が 4 の場合と 8 の場合の比較ではパケット長が 8 の方が少ないレイテンシで通信を行うことができている，しかしパケット長が 8 の場合と 16 の場合の比較では，データ送信に必要なパケット数が約半分にもかかわらず，大きなレイテンシの差は認められない．したがって，M-Core Advanced の NoC におけるパケット長は 8 とする．

### 5.3.2 並列アプリケーションによる性能評価

並列アプリケーションを用いて M-Core Advanced の性能を評価する．アプリケーションは，バイトニックソート，行列積，ステンシル計算，N クイーンの 4 種とする．この評価では，コアと NoC の性能のバランスを考慮する．そのために，シミュレーションに用いるアプリケーションは長いレイテンシを要するオフチップのメモリへのアクセスは行わず，ノードメモリのみを使用する．このため，ある程度大規模なアプリケーションを実行するためにノードメモリサイズを 2 MB として評価している．

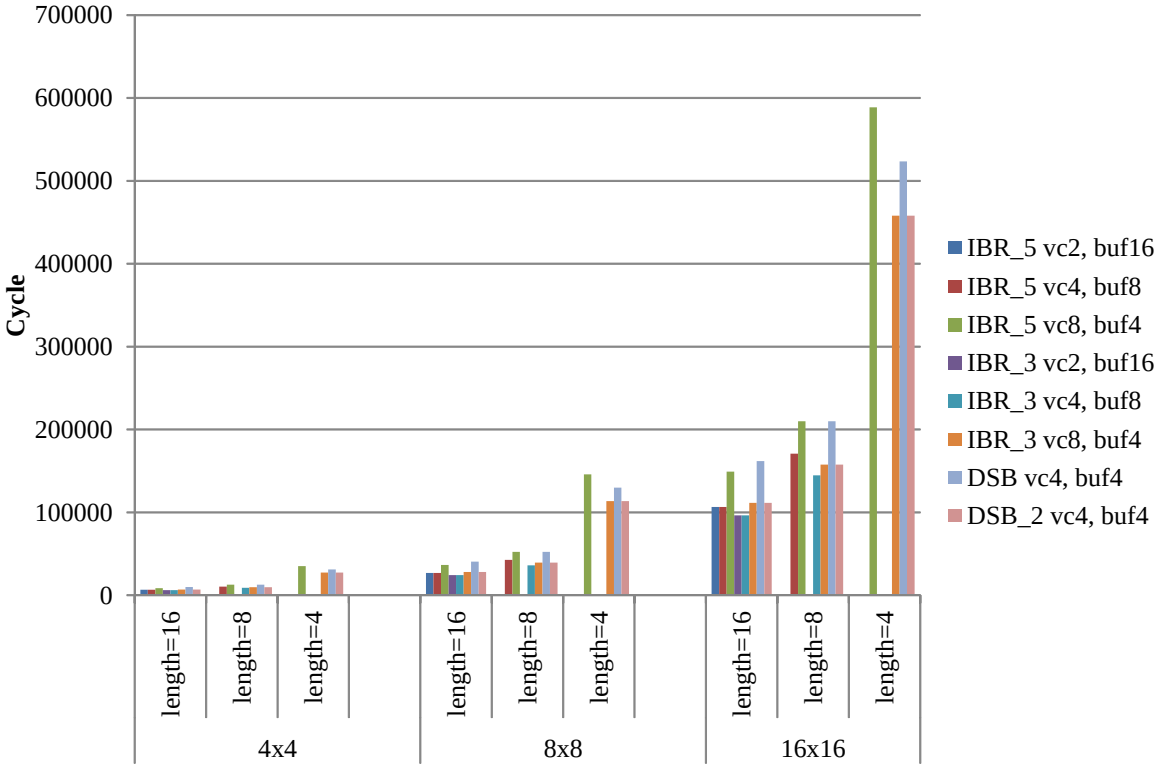


図 5.9 全対 1 通信のノード数，パケット長ごとのレイテンシ

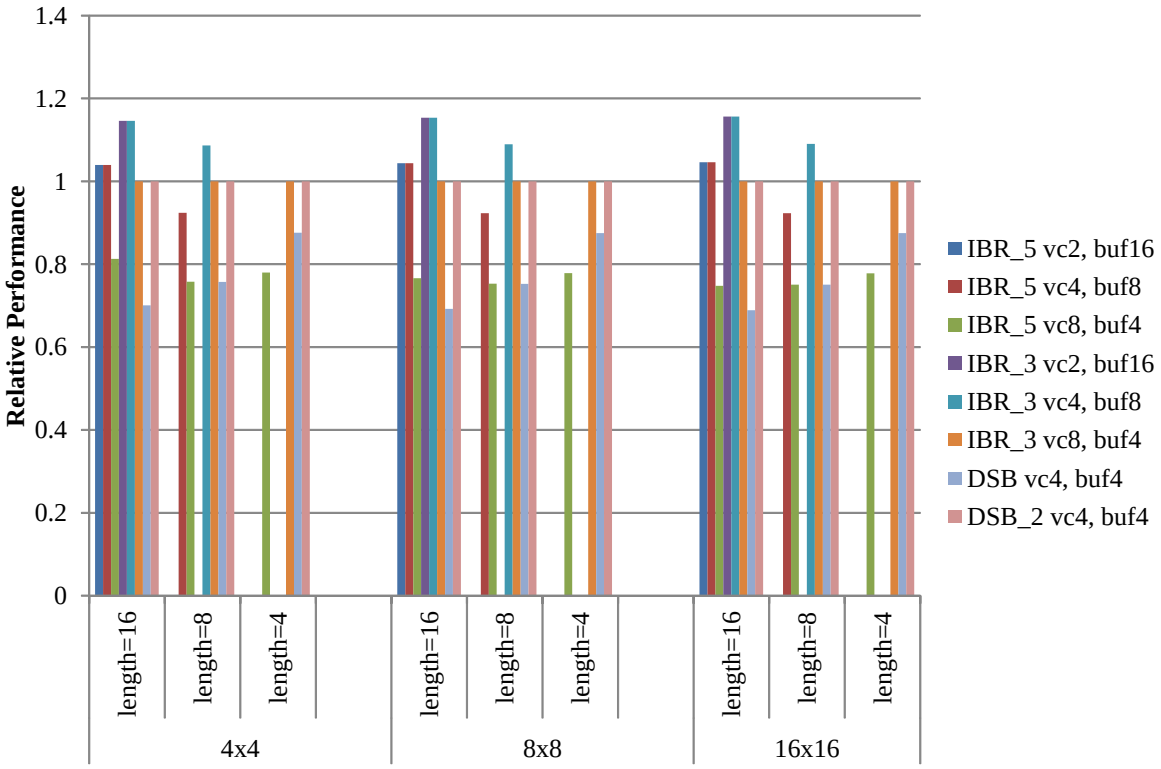


図 5.10 全対 1 通信のノード数，パケット長ごとのレイテンシの相対性能

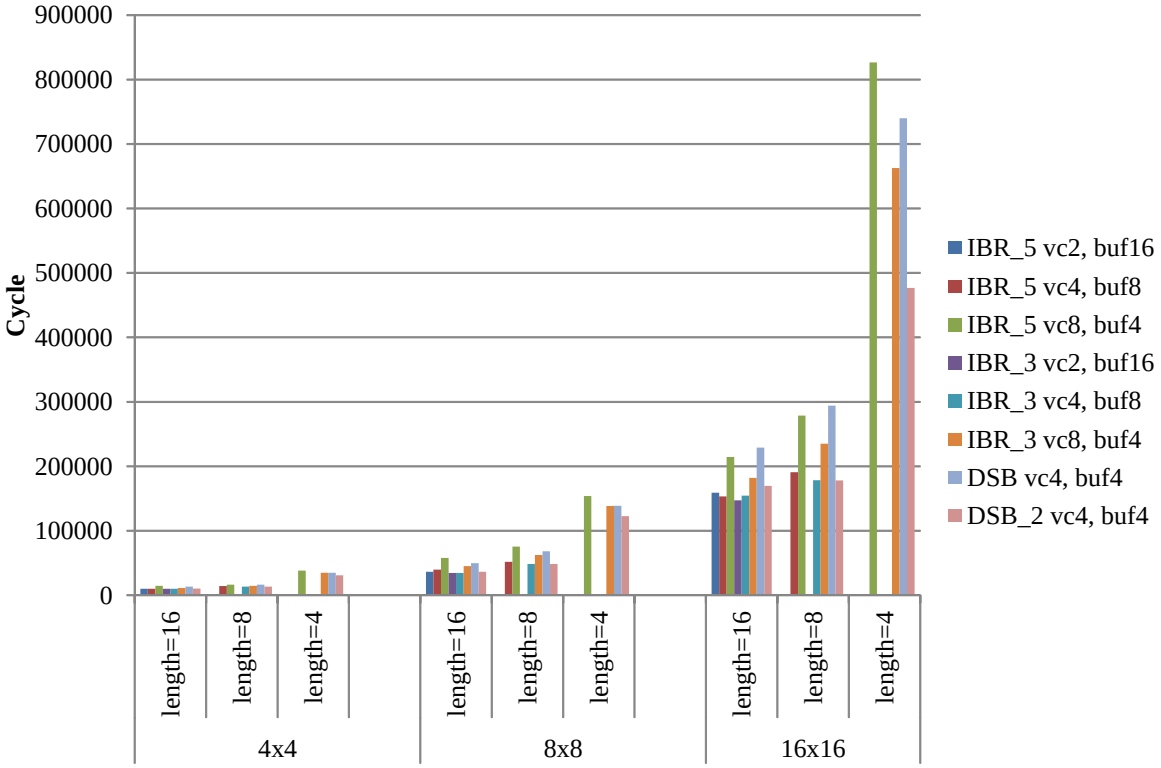


図 5.11　全対全通信のノード数，パケット長ごとのレイテンシ

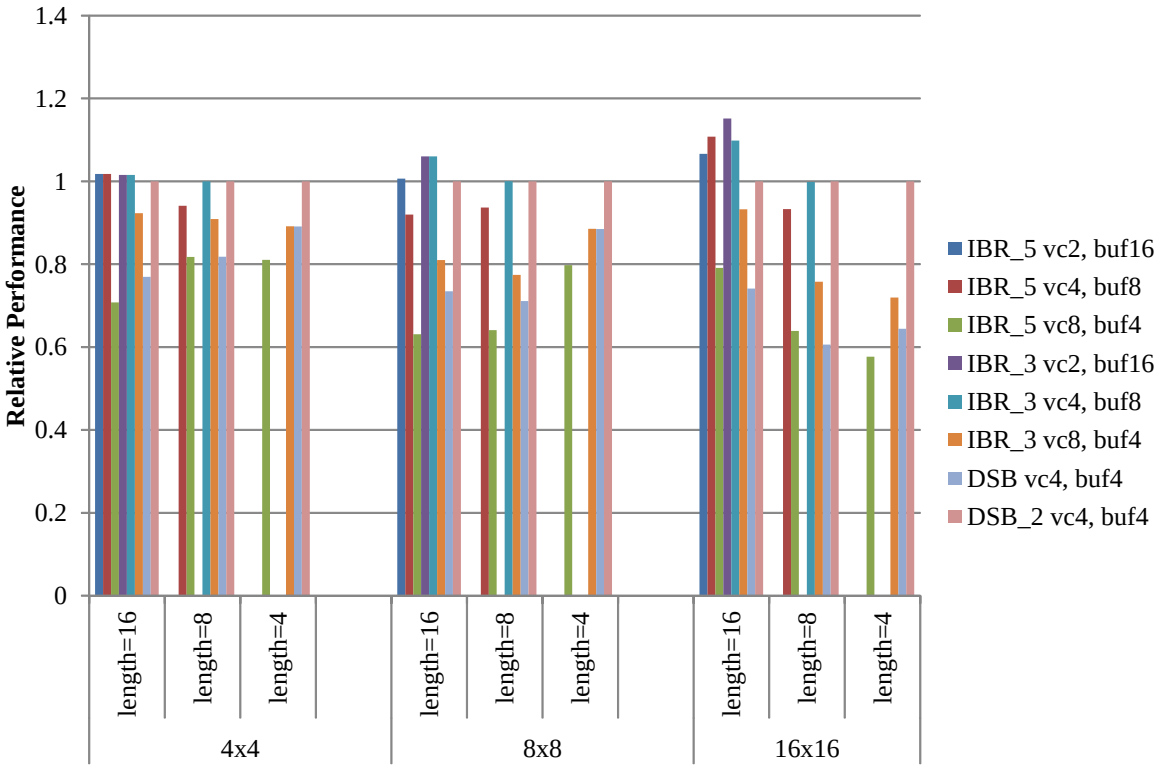


図 5.12　全対全通信のノード数，パケット長ごとのレイテンシの相対性能

アプリケーションの仕様を以下にまとめる．

**バイトニックソート（bitonic）**

バイトニックソートは，文献 [85] による並列ソートで，128K 個の要素を並列にソートする．要素の初期値は，Xorshift アルゴリズムによる乱数で初期化している．

**行列積（mm_cannon）**

Cannon のアルゴリズム [86] により並列化された行列積．行列のサイズ $256 \times 256$，要素は 32 ビットの整数とする．

**ステンシル計算（stencil）**

ステンシル計算 [87] は，2 次元配列で宣言されたデータのある要素について，4 近傍および自身の値の平均を求める処理を繰り返す．配列のサイズは $256 \times 256$，繰り返しの回数は 32 回とする．また，要素は 32 ビットの整数とする．

**N クイーン（nq）**

MPI を用いて並列化された N クイーン [88]．MPI によるデータ転送の記述をライブラリが提供するデータ転送に置き換えている．問題サイズは 13 とする．

シミュレーションを行なうノード数は $4 \times 4$ の 16 ノード，$8 \times 8$ の 64 ノード，$16 \times 16$ の 256 ノードの 3 つのサイズとする．比較のために NoC のパラメータ設定に用いたすべてのルータを用いて評価を行う．IBR の設定は，通信アプリケーションの評価において最もよい性能を示した仮想チャネル 4，バッファのエントリ数 8 とする．

図 5.13 にノード数，アプリケーションごとの実行サイクル数を示す．縦軸は実行サイクル数を表している．横軸は，ルータごとの比較のためにノード数，アプリケーションごとにまとめて表示している．また，図 5.14 にノード数，アプリケーションごとの実行サイクル数をパイプラインをバイパスする DSB ルータを基準とした相対性能で示す．アプリケーションの実行サイクル数において，パイプラインをバイパスする DSB を搭載する M-Core Advanced でのアプリケーションの性能は，3 段パイプラインの IBR に次ぐ性能となった．また，ノード数の増加に伴って異なるルータによる性能差が大きくなっている．

図 5.15 にパイプラインをバイパス DSB ルータを搭載する M-Core Advanced のアプリケーションごとの相対性能を示す．縦軸は相対性能で，ノード数が $4 \times 4$ の時の実行サイクル数を基準としている．すべてのアプリケーションについてノード数の増加に伴う性能向上が得られている．

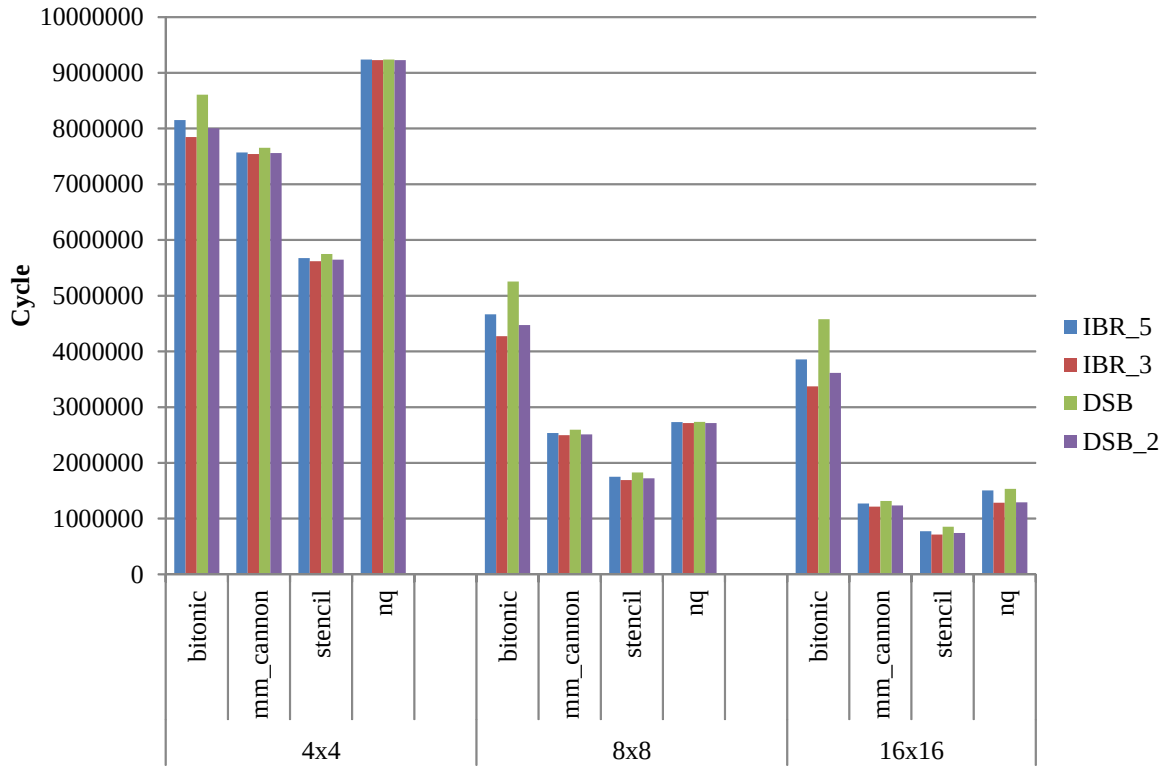


図 5.13 ノード数，アプリケーションごとの実行サイクル数

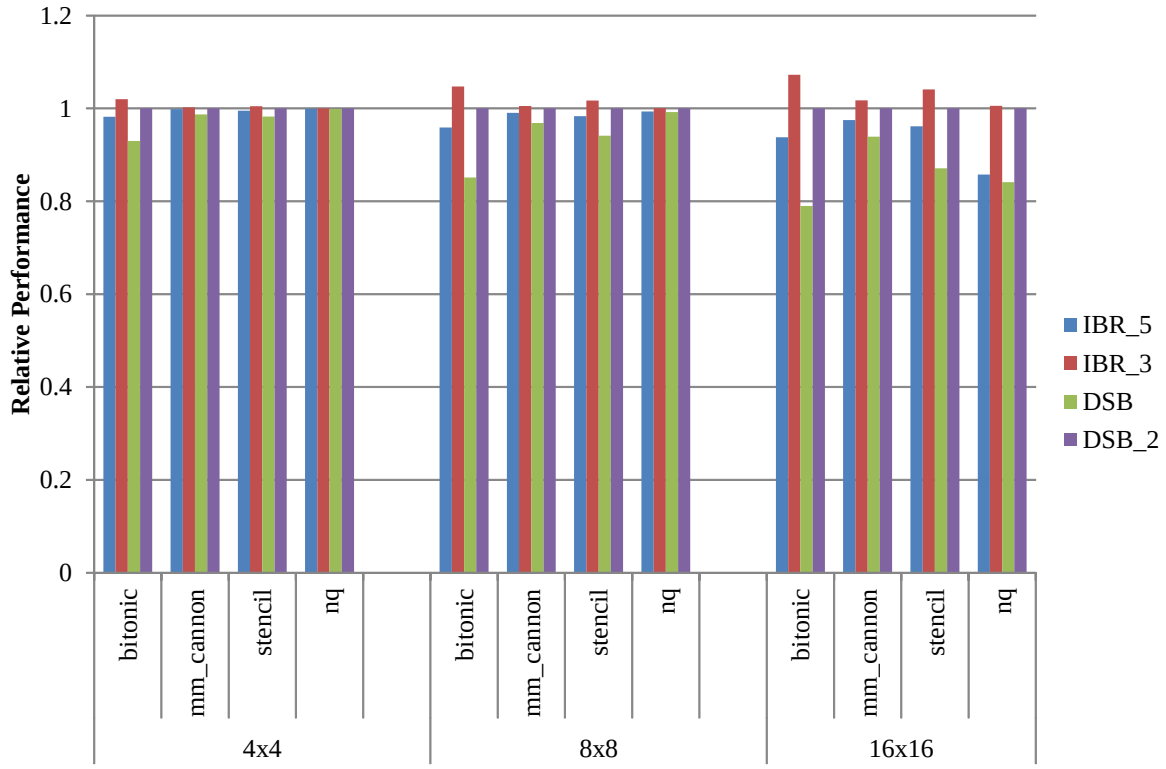


図 5.14 ノード数，アプリケーションごとの実行サイクル数の相対性能

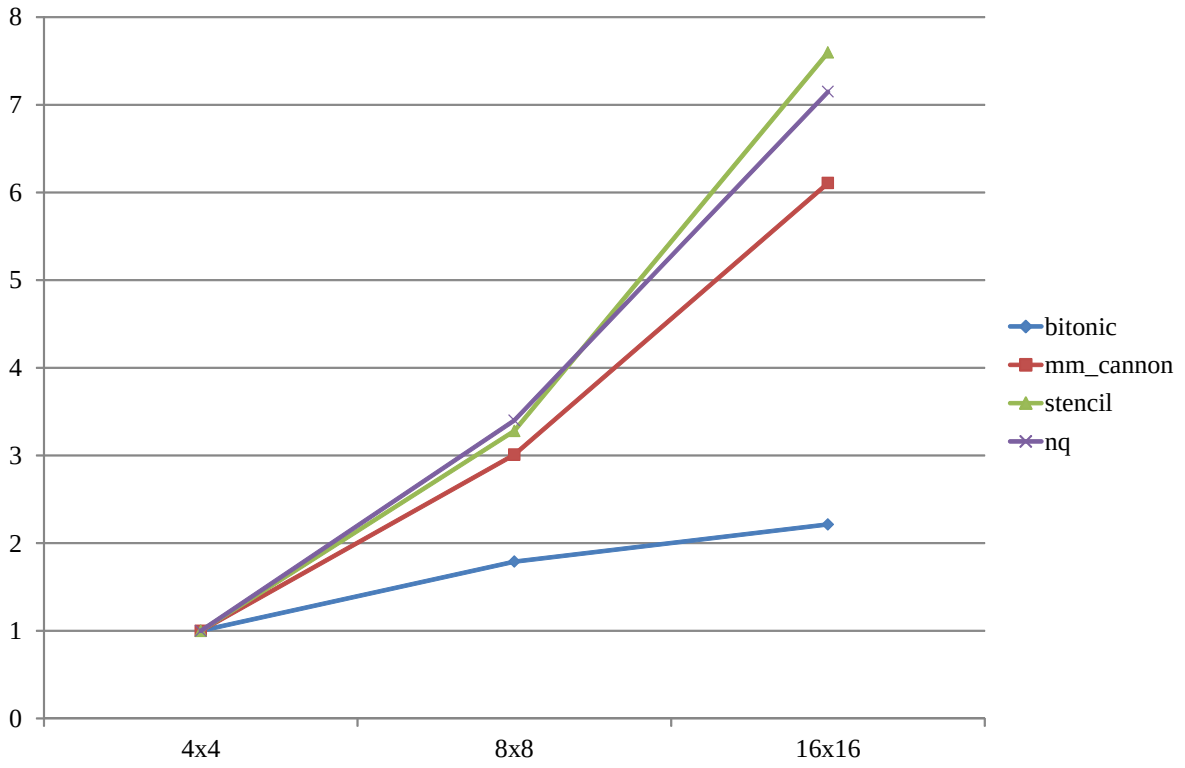


図 5.15 パイプラインをバイパス DSB ルータを搭載する M-Core Advanced のアプリケーションごとの相対性能

### 5.3.3 議論

一般に入力バッファのエントリ数とパケット長を等しくするとネットワークの性能を生かすことができる [11]．評価で使用した DSB ルータの設定は，文献 [14] と同じ値を用いたため，バッファのエントリ数は 4 に固定していた．そのため，パケット長が 8 や 16 の場合の評価は，IBR のバッファのエントリ数を 8 としたときの性能と比較するとやや不利な評価であるといえる．バッファのエントリ数が 4 の場合であれば，各ルータとの比較においてパイプラインをバイパスする DSB ルータは最もよい性能を示している．そのなかで，パケット長が 8 の場合の全対全通信の評価において，DSB ルータが 3 段パイプラインの IBR と同等の性能を達成し，スループットが求められるアプリケーションでは DSB ルータの性能が発揮されることが確認できる．

並列アプリケーションによる評価では，ノード数が大きくなるにつれてルータごとの性能差が大きくなる結果となった．これは，評価に用いたアプリケーションがスループット

よりもレイテンシの性能に影響されるためであると考えられる．スループットを必要とする並列アプリケーションとして高速フーリエ変換（FFT）が知られているが，本論文において実装した MIPS アーキテクチャは，浮動小数点演算をサポートしていないため，評価に用いることができなかった．

# 第 6 章

# おわりに

## 6.1 結論

トランジスタ数の増加に伴い，チップ上に集積される要素が増加している．本研究は，そのようなメニーコアの時代における効率のよいアーキテクチャの設計，評価を行うために新しい RTL モデリング手法を提案し，その手法を用いて現実的なメニーコアプロセッサを設計した．本研究による貢献を以下に挙げる．

**効率的にハードウェアを設計，評価するための新しい環境の提案**

効率的にハードウェアを設計, 評価するための新しい設計環境では, C++ 言語でハードウェアを RTL で設計するための新しい言語 ArchHDL を提案する．ArchHDL が提供するライブラリを利用して記述したハードウェアは，RTL で記述されているにもかかわらず，C++ 言語による柔軟なパラメータサーベイが可能である．また，ライブラリによりシミュレーションが並列化でき，商用の Verilog HDL シミュレータよりも高速に RTL で記述されたハードウェアのシミュレーションが可能となる．同時に，ArchHDL で記述したソースコードを Verilog HDL のソースコードに自動変換するツールを開発する．これにより，ハードウェア記述に制約が生じるが，ArchHDL でハードウェアを記述だけで Verilog HDL のソースコードを生成でき，論理合成ツールによる FPGA 実装などが可能になる．

**メニーコアプロセッサのための高性能な Network-on-Chip ルータの提案**

メニーコアプロセッサのための高性能な NoC ルータの提案では，高スループットな NoC ルータとして知られる Distributed Shared-Buffer NoC ルータ [14]（DSB ルータ）に，アーキテクチャの改良によるレイテンシを削減する提案手法を適用し，DSB

ルータの高性能化を達成する．また，提案手法の適用範囲を拡げ，さらなるレイテンシの削減により，より高性能な NoC ルータを実現する．

**新しいハードウェア設計環境を利用した，現実的なメニーコアプロセッサの設計と評価**

提案する新しいハードウェア設計環境を用いてメニーコアプロセッサを設計し評価する．具体的には，研究・教育を目的として設計されたメニーコアアーキテクチャ M-Core[15] をベースに，提案する高性能な NoC を搭載するメニーコアプロセッサを設計する．様々なパラメータと並列アプリケーションを用いた評価から，提案する NoC ルータによる性能向上を示す．

## 6.2 今後の展望

本研究で提案している効率的にハードウェア設計，評価するための新しい環境についての今後の課題として以下が挙げられる．

- デバッガビリティの向上
- 自動変換ツールの高機能化

ArchHDL のライブラリサポートによるハードウェア記述能力やシミュレーション速度についての有用性は，メニーコアプロセッサの実装や評価によって示すことができた．しかし，記述したハードウェアのデバッグに関しては，ワイヤの接続ミスのような初歩的な問題であっても一切サポートされておらず，デバッグが困難である．ArchHDL の有用性を高めるためにデバッガビリティを向上させることが必要である．

現状の自動変換ツールは，ツールが変換できる記述にするために ArchHDL によるハードウェア記述を制限してしまう．本研究における変換ツールの実装により，ArchHDL は Verilog HDL に変換可能であることが確認できた．より詳細にソースコードを解析することにより，ArchHDL によるハードウェア記述を制限することなく Verilog HDL への変換を達成することが望まれる．

現実的なメニーコアプロセッサの提案についての今後の課題として以下が挙げられる．

- より多くのアプリケーションを用いた評価
- オフチップのメモリも含めた性能の評価

設計したメニーコアプロセッサの評価では，通信アプリケーションから提案する NoC ルータによる性能向上が確認できたが，並列アプリケーションにおいては，提案するルー

タによる性能向上を定量的に示すことができなかった．これは，評価に用いたアプリケーションが，提案する NoC ルータの性能を十分に生かせなかったためであり，より多くのアプリケーションによる評価が必要であると考えている．

本研究では，NoC とコアの性能のバランスを考慮するためにオフチップのメモリを含めない評価を行った．オフチップのメモリアクセスに関しては，メモリアクセスレイテンシが大きいため，その性能を生かすためにはバンド幅を有効に使うことが必要となる．この点を考慮した評価から，メニーコアアーキテクチャを検討していくことが重要である．

# 謝辞

本研究の一部は，科学技術振興機構・戦略的創造研究推進事業（CREST）の「アーキテクチャと形式的検証の協調による超ディペンダブルプラットフォームの構築」の支援によります．また，本研究の一部は，日本学術振興会の特別研究員制度の支援によります．

はじめに，学部 4 年の 1 年間，修士課程の 2 年間，そして博士課程の 5 年半の 8 年以上の間，始終熱心なご指導を賜りました，指導教員の吉瀬謙二准教授に深く感謝の意を表します．私のような，気まぐれで，言うことを聞かない学生を時には厳しく，辛抱強く導いて下さり誠にありがとうございます．先生には，研究の方針から論文執筆に至るまで，あらゆる面において貴重な御助言を頂きました．また多くの発表の機会を与えて頂き，たくさんの研究者と知り合い，見聞を広めることができました，改めて御礼申し上げます．

吉瀬研究室の皆様には，活発な議論と多くの貴重な意見を頂きありがとうございました．特に，本研究を遂行する上で，藤枝直輝氏，植原昂氏，高前田（山崎）伸也氏，姜軒氏，佐野伸太郎氏には多大なるご協力をいただきました．ここに感謝致します．また，私の気晴らしにつきあって頂いた研究室の皆様，ありがとうございます．

吉瀬研究室の 1 期生のため，先輩のいない私は研究や研究室での生活について他の研究室の先輩方に幾度となく助けて頂きました．東京工業大学 林晋平助教には，学部 4 年の頃から研究に対する姿勢や後輩の指導についてなど多くの御助言を頂きました．研究室に研究員として 1 年間在籍した（株）イーツリーズ・ジャパンの三好健文博士には，研究プロジェクトを通じて研究者としての姿勢を勉強させていただきました．東京工業大学 荒堀喜貴助教には，研究が思うように進まない私を気にかけ，御助言を頂きました．また，学会など学外で会う年代の近い皆様にも多くの助言を頂きました．ここに深く感謝致します．

本論文の審査員として，横田治夫教授，宮崎純教授，渡部卓雄准教授，金子晴彦准教授には的確な指摘と提案を頂きました．また，中間審査において，早稲田大学の森欣司教授，国立情報学研究所の米田友洋教授，IBM 東京基礎研究所の鈴村豊太郎博士には貴重なご意見を頂きました．助言教員として小林隆志准教授には，私が研究に悩んでいた時期に貴重な御助言を頂きました．お世話になった多くの先生方に深く感謝致します．

学外における研究の場として，情報処理学会の計算機アーキテクチャ研究会，電子情報通信学会コンピュータシステム研究会で活発な議論をした皆様に感謝致します．

最後に，なかなか博士論文をまとめられなかった私を信じ，支えてくれた家族に深く感謝いたします．

# 参考文献


[1] Christos Kozyrakis, Aman Kansal, Sriram Sankar, and Kushagra Vaid. Server Engineering Insights for Large-Scale Online Services. *IEEE Micro Magazine*, Vol. 30, No. 4, pp. 8–19, July 2010.

[2] International Technology Roadmap for Semiconductors, http://www.itrs.net.

[3] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel Loh, Don McCaule, Pat Morrow, Donald Nelson, Daniel Pantuso, Paul Reed, Jeff Rupley, Sadasivan Shankar, John Shen, and Clair Webb. Die Stacking (3D) Microarchitecture. In *Proceedings of the 39th International Symposium on Microarchitecture (MICRO '06)*, pp. 469–479, December 2006.

[4] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA '00)*, Vol. 28, pp. 248–259, June 2000.

[5] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Pearson Education, fourth edition, 2011.

[6] 伊藤則之, 安永守利. プロセッサ設計手法の現状と今後：高性能化を実現する設計フローと CAD システム. 電子情報通信学会論文誌. D, Vol. J94-D, No. 12, pp. 2004–2030, 2011 年 12 月.

[7] David Flynn. AMBA: Enabling Reusable On-Chip Designs. *IEEE Micro Magazine*, Vol. 17, No. 4, pp. 20–27, 1997.

[8] IBM Corporation, The CoreConnect Bus Architecture, http://www-01.ibm.com/chips/techlib/techlib.nsf/products/CoreConnect_Bus_Architecure, 1999.

[9] William J. Dally and Brian Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *Proceedings of the 38th Conference on Design Automation (DAC '01)*, pp. 684–689, June 2001.

[10] Luca Benini and Giovanni De Micheli. Networks on Chips: a New SoC Paradigm. *IEEE Computer Magazine*, Vol. 35, No. 1, pp. 70–78, January 2002.

[11] William J. Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 2004.

[12] Luca Benini and Giovanni De Micheli. *Networks on Chips: Technology And Tools*. The Morgan Kaufmann Series in Systems on Silicon. Morgan Kaufmann, 2006.

[13] Jose Flich and Davide Bertozzi. *Designing Network On-Chip Architectures in the Nanoscale Era*. Computational Science. Taylor & Francis, 2011.

[14] Rohit Sunkam Ramanujam, Vassos Soteriou, Bill Lin, and Li-Shiuan Peh. Extending the Effective Throughput of NoCs With Distributed Shared-Buffer Routers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 30, No. 4, pp. 548–561, April 2011.

[15] 植原昂, 佐藤真平, 吉瀬謙二. メニーコアプロセッサの研究・教育を支援する実用的な基盤環境. 電子情報通信学会論文誌 D, Vol. J93-D, No. 10, pp. 2042–2057, 2010 年 10 月.

[16] Daniel U. Becker and William J. Dally. Allocator Implementations for Network-on-Chip Routers. In *Proceedings of the 2009 ACM/IEEE conference on Supercomputing (SC '09)*, pp. 1–12, November 2009.

[17] Nick McKeown. The iSLIP Scheduling Algorithm for Input-Queued Switches. *IEEE/ACM Transactions on Networking*, Vol. 7, No. 2, pp. 188–201, April 1999.

[18] William J. Dally. Virtual-Channel Flow Control. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA '90)*, pp. 60–68, 1990.

[19] Li-Shiuan Peh and William J. Dally. A Delay Model and Speculative Architecture for Pipelined Routers. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA '01)*, pp. 255–266, January 2001.

[20] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, Vol. 49, No. 4/5, pp. 589–604, July 2005.

[21] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell Multiprocessor Communication Network: Built for Speed. *IEEE Micro Magazine*, Vol. 26, No. 3, pp. 10–23, May 2006.

[22] J.-S. Vogt, R. Land, H. Boettiger, Z. Krnjajic, and H. Baier. IBM BladeCenter QS22:

Design, performance, and utilization in hybrid computing systems. *IBM Journal of Research and Development*, Vol. 53, No. 5, pp. 3:1–3:14, September 2009.

[23] 檜田和浩, 近藤伸宏. メディアストリーミングプロセッサ SpursEngine とその応用例. 東芝レビュー, Vol. 63, No. 7, pp. 17–21, 2008 年 7 月.

[24] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro Magazine*, Vol. 22, No. 2, pp. 25–35, March 2002.

[25] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA '04)*, pp. 2–13. IEEE, June 2004.

[26] Shane Bell, Bruce Edwards, John Amann, Rich Conlin, Kevin Joyce, Vince Leung, John MacKay, Mike Reif, Liewei Bao, John Brown, Matthew Mattina, Chyi-Chang Miao, Carl Ramey, David Wentzlaff, Walker Anderson, Ethan Berger, Nat Fairbanks, Durlov Khan, Froilan Montenegro, Jay Stickney, and John Zook. TILE64 Processor: A 64-Core SoC with Mesh Interconnect. In *Proceedings of the 2008 International Solid-State Circuits Conference (ISSCC '08)*, pp. 88–89. IEEE, February 2008.

[27] Carl Ramey. TILE-Gx100 ManyCore Processor: Acceleration Interfaces and Architecture. *Hot Chips 23*, August 2011.

[28] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA '03)*, pp. 422–433. IEEE Comput. Soc, June 2003.

[29] 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 Magazine*, Vol. 37, No. 7, pp. 44–55, July 2004.

[30] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Gregory Ruhl, and Saurabh Dighe. The 48-core SCC Processor: the Programmer's View. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, November 2010.

[31] Praveen Salihundam, Shailendra Jain, Tiju Jacob, Shasi Kumar, Vasantha Erraguntla, Yatin Hoskote, Sriram Vangal, Gregory Ruhl, and Nitin Borkar. A 2 Tb/s 6 x 4 Mesh Network for a Single-Chip Cloud Computer With DVFS in 45 nm CMOS. *IEEE Journal of Solid-State Circuits*, Vol. 46, No. 4, pp. 757–766, April 2011.

[32] Sriram Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Priya Iyer, Arvind Singh, Tiju Jacob, Shailendra Jain, Sriram Venkataraman, Yatin Hoskote, and Nitin Borkar. An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS. In *Proceedings of the 2007 International Solid-State Circuits Conference (ISSCC '07)*, pp. 98–589. IEEE, February 2007.

[33] Sriram R. Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Arvind Singh, Tiju Jacob, Shailendra Jain, Vasantha Erraguntla, Clark Roberts, Yatin Hoskote, Nitin Borkar, and Shekhar Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, Vol. 43, No. 1, pp. 29–41, January 2008.

[34] J. Schutz. A 3.3V 0.6 $\mu$m BiCMOS superscalar microprocessor. In *Proceedings of the 1994 International Solid-State Circuits Conference (ISSCC '94)*, pp. 202–203. IEEE, 1994.

[35] 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. *ACM Transactions on Graphics*, Vol. 27, No. 3, pp. 1–15, August 2008.

[36] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Pradeep Dubey, Stephen Junkins, Adam Lake, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, Michael Abrash, Jeremy Sugerman, and Pat Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *IEEE Micro Magazine*, Vol. 29, No. 1, pp. 10–21, August 2009.

[37] Alejandro Duran and Michael Klemm. The IntelÂő Many Integrated Core Architecture. In *Proceedings of the 2012 International Conference on High Performance Computing*

*& Simulation (HPCS '12)*, pp. 365–366. IEEE, July 2012.

[38] Zhiyi Yu, Michael Meeuwsen, Ryan Apperson, Omar Sattari, Michael Lai, Jeremy Webb, Eric Work, Tinoosh Mohsenin, Mandeep Singh, and Bevan Baas. An Asynchronous Array of Simple Processors for DSP Applications. In *Proceedings of the 2006 International Solid-State Circuits Conference (ISSCC '06)*, pp. 1696–1705. IEEE, 2006.

[39] Bevan Baas, Zhiyi Yu, Michael Meeuwsen, Omar Sattari, Ryan Apperson, Eric Work, Jeremy Webb, Michael Lai, Tinoosh Mohsenin, Dean Truong, and Jason Cheung. AsAP: A Fine-Grained Many-Core Platform for DSP Applications. *IEEE Micro Magazine*, Vol. 27, No. 2, pp. 34–45, March 2007.

[40] Michel A. Kinsy, Michael Pellauer, and Srinivas Devadas. Heracles: Fully Synthesizable Parameterized MIPS-Based Multicore System. In *Proceedings of the 21st International Conference on Field Programmable Logic and Applications (FPL '11)*, pp. 356–362. IEEE, September 2011.

[41] Michel A. Kinsy, Michael Pellauer, and Srinivas Devadas. Heracles: A Tool for Fast RTL-Based Design Space Exploration of Multicore Processors. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '13)*, p. 125, New York, New York, USA, February 2013. ACM Press.

[42] Hiroki Matsutani, Michihiro Koibuchi, Hideharu Amano, and Tsutomu Yoshinaga. Prediction Router: A Low-Latency On-Chip Router Architecture with Multiple Predictors. *IEEE Transactions on Computers*, Vol. 60, No. 6, pp. 783–799, June 2011.

[43] Yuan He, Hiroshi Sasaki, Shinobu Miwa, and Hiroshi Nakamura. McRouter: Multicast within a Router for High Performance Network-on-Chips. *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*, pp. 319–330, October 2013.

[44] 佐藤真平, 吉瀬謙二. C++ をベースとする新しいハードウェア記述の検討. 情報処理学会研究報告 2013-ARC-205, No. 75, pp. 1–7, 2013 年 4 月.

[45] 金子達哉, 佐藤真平, 吉瀬謙二. ArchHDL で記述したハードウェアの論理シミュレーションの高速化. 情報処理学会研究報告 2013-ARC-206, No. 25, pp. 1–8, 2013 年 7 月.

[46] Shimpei Sato and Kenji Kise. ArchHDL: A New Hardware Description Language for High-Speed Architectural Evaluation. In *Proceedings of the 7th IEEE International Symposium on Embedded Multicore/Many-core System-on-Chip (MCSoC '13)*, pp. 107–112, September 2013.

[47] 佐藤真平, 吉瀬謙二. ArchHDL によるハードウェア記述の実践. 情報処理学会研究報

告 2014-ARC-208, No. 21, pp. 1–8, 2014.

[48] Shimpei Sato and Kenji Kise. Ultra-High Speed Architectural Simulation Methodology. *The 16th International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA '14)*, March 2014.

[49] 佐藤真平, 吉瀬謙二. ハードウェアモデリング言語 ArchHDL から Verilog HDL へのトランスレータの設計. 電子情報通信学会技術研究報告 CPSY2013-10-03, Vol. 113, No. 234, pp. 1–6, 2013 年 10 月.

[50] 佐藤真平, 藤枝直輝, 田原慎也, 吉瀬謙二. 実用的かつコードのシンプルさを追求した Cell BE の機能レベルシミュレータ SimCell の設計と実装. 第 19 回 コンピュータシステム・シンポジウム論文集（ComSys 2007）, pp. 39–47, 2007 年 11 月.

[51] Shimpei Sato, Naoki Fujieda, Akira Moriya, and Kenji Kise. Processor Simulator SimCell to Accelerate Research on Many-core Processor Architectures. *Proceedings of the Workshop on Cell Systems and Applications (WCSA)*, pp. 119–127, June 2008.

[52] Shimpei Sato, Naoki Fujieda, Akira Moriya, and Kenji Kise. SimCell: A Processor Simulator for Multi-Core Architecture Research. *IPSJ Transactions on Advanced Computing Systems*, Vol. 2, No. 1, pp. 146–157, March 2009.

[53] 藤枝直輝, 渡邉伸平, 吉瀬謙二. 教育・研究に有用な MIPS システムシミュレータ SimMips. 情報処理学会論文誌, Vol. 50, No. 11, pp. 2665–2676, 2009 年 11 月.

[54] IEEE. IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Standard for Standard SystemC Language Reference Manual*, 2011.

[55] Jan Decaluwe. MyHDL: a Python-Based Hardware Description Language. *Linux Journal*, Vol. 2004, No. 127, p. 5, November 2004.

[56] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing Hardware in a Scala embedded Language. In *Proceedings of the 49th Design Automation Conference (DAC '12)*, pp. 1216–1225, June 2012.

[57] Icarus Verilog, http://iverilog/icarus.com.

[58] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A Survey on Reactive Programming. *ACM Computing Surveys*, Vol. 45, No. 4, pp. 1–34, August 2013.

[59] Conal Elliott and Paul Hudak. Functional Reactive Animation. *ACM SIGPLAN Notices*, Vol. 32, No. 8, pp. 263–273, August 1997.

[60] Yampa, http://www.haskell.org/haskellwiki/Yampa.

[61] Reactive, http://www.haskell.org/haskellwiki/Reactive.

[62] Grapefruit, http://www.haskell.org/haskellwiki/Grapefruit.

[63] Antony Courtney. Frappé: Functional Reactive Programming in Java. In *Proceedings of the 3rd International Symposium on Practical Aspects of Declarative Languages (PADL '01)*, pp. 29–44. Springer-Verlag, March 2001.

[64] Ingo Maier and Martin Odersky. Deprecating the Observer Pattern with Scala.React. *EPFL-REPORT-176887*, 2012.

[65] Sodium, https://github.com/kentuckyfriedtakahe/sodium.

[66] Gérard Berry and Laurent Cosserat. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In *Proceedings of Sminar on Concurrency*, pp. 389–448. Springer-Verlag, July 1984.

[67] David A. Patterson and John L. Hennessy. *Computer Organization and Design*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, fourth rev edition, 2012.

[68] Vassos Soteriou, Rohit Sunkam Ramanujam, Bill Lin, and Li-Shiuan Peh. A High-Throughput Distributed Shared-Buffer NoC Router. *IEEE Computer Architecture Letters*, Vol. 8, No. 1, pp. 21–24, June 2009.

[69] Rohit Sunkam Ramanujam, Vassos Soteriou, Bill Lin, and Li-Shiuan Peh. Design of a High-Throughput Distributed Shared-Buffer NoC Router. In *Proceedings of the 4th ACM/IEEE International Symposium on Networks-on-Chip (NOCS '10)*, pp. 69–78. IEEE, May 2010.

[70] 姜軒, 佐藤真平, 吉瀬謙二. Distributed Shared-buffer ルータの遅延を削減するパイプラインバイパス方式. 情報処理学会研究報告 2011-ARC-194, No. 13, pp. 1–10, 2011 年 3 月.

[71] 佐藤真平, 笹河良介, 吉瀬謙二. Distributed Shared-Buffer NoC ルータのためのパイプラインバイパス手法の改良. 情報処理学会研究報告 2011-ARC-196, No. 4, pp. 1–9, 2011 年 7 月.

[72] 佐藤真平, 吉瀬謙二. Distributed Shared-Buffer NoC ルータのためのパイプラインバイパス手法. 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. 1, pp. 88–102, 2012 年 1 月.

[73] Anh T. Tran and Bevan M. Baas. RoShaQ: High-Performance On-Chip Router with Shared Queues. In *Proceedings of the 29th International Conference on Computer Design (ICCD '11)*, pp. 232–238, October 2011.

[74] Craig B. Stunkel, Jay Herring, Bulent Abali, and Rajeev Sivaram. A New Switch Chip for IBM RS/6000 SP Systems. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (SC '99)*, pp. 1–17, January 1999.

[75] Amit Kumar, Li-Shiuan Peh, Partha Kundu, and Niraj K. Jha. Express Virtual Channels: Towards the Ideal Interconnection Fabric. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA '07)*, Vol. 35, pp. 150–161. ACM, June 2007.

[76] Amit Kumar, Li-Shiuan Peh, and Niraj K. Jha. Token Flow Control. In *Proceedings of the 41st International Symposium on Microarchitecture (MICRO '08)*, pp. 342–353, November 2008.

[77] C. Izu, R. Beivide, and C. Jesshope. Mad-Postman: a Look-ahead Message Propagation Method for Static Bidimensional Meshes. In *Proceedings of the 2nd Euromicro Workshop on Parallel and Distributed Processing*, pp. 117–124, 1994.

[78] Dongkook Park, Reetuparna Das, Chrysostomos Nicopoulos, Jongman Kim, N. Vijaykrishnan, Ravishankar Iyer, and Chita R. Das. Design of a Dynamic Priority-Based Fast Path Architecture for On-Chip Interconnects. In *Proceedings of the 15th IEEE Symposium on High-Performance Interconnects (HOTI '07)*, pp. 15–20, August 2007.

[79] George Michelogiannakis, Dionisios Pnevmatikatos, and Manolis Katevenis. Approaching Ideal NoC Latency with Pre-Configured Routes. In *Proceedings of the 1st International Symposium on Networks-on-Chip (NOCS '07)*, pp. 153–162, May 2007.

[80] Koh Uehara, Shimpei Sato, Takefumi Miyoshi, and Kenji Kise. A Study of an Infrastructure for Research and Development of Many-Core Processors. In *Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT '09)*, pp. 414–419, December 2009.

[81] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, fifth edition, 2012.

[82] Message Passing Interface Forum, http://www.mpi-forum.org/.

[83] 森本健司, 松本尚, 平木敬. メモリベース通信を用いた高速 MPI の実装と評価. 情報処理学会論文誌, Vol. 40, No. 5, pp. 2256–2268, 1999 年 5 月.

[84] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Guputa. *Introduction to Parallel Computing*. Peason, second edition, 2003.

[85] D. Nassimi and S. Sahni. Bitonic Sort on a Mesh-Connected Parallel Computer. *IEEE Transactions on Computers*, Vol. C-28, No. 1, pp. 2–7, January 1979.

[86] Lynn Elliot Cannon. A Cellular Computer to Implement the Kalman Filter Algorithm. *Doctoral Dissertation, Montana State University Bozeman, MT, USA*, 1969.

[87] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David A. Patterson, John Shalf, and Katherine Yelick. Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures. In *Proceedings of the 2008 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12. IEEE, November 2008.

[88] Kenji Kise, Takahiro Katagiri, Hiroki Honda, and Toshitsugu Yuba. Solving the 24-queens Problem using MPI on a PC Cluster. *Technical Report UEC-IS-2004-6*, pp. 1–6, June 2004.

# 著者発表文献

## 論文誌

1. 佐藤真平, 吉瀬謙二. Distributed Shared-Buffer NoC ルータのためのパイプラインバイパス手法. 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. 1, pp. 88–102, 2012 年 1 月.
2. 高前田伸也, 佐藤真平, 藤枝直輝, 三好健文, 吉瀬謙二. メニーコアアーキテクチャの HW 評価環境 ScalableCore システム. 情報処理学会論文誌コンピューティングシステム, Vol. 4, No. 1, pp. 24–42, 2011 年 2 月.
3. 植原昂, 佐藤真平, 吉瀬謙二. メニーコアプロセッサの研究・教育を支援する実用的な基盤環境. 電子情報通信学会論文誌 D, Vol. J93-D, No. 10, pp. 2042–2057, 2010 年 10 月.
4. Shimpei Sato, Naoki Fujieda, Akira Moriya, and Kenji Kise. SimCell: A Processor Simulator for Multi-Core Architecture Research. IPSJ Transactions on Advanced Computing Systems, Vol. 2, No. 1, pp. 146–157, March 2009.

## 国際会議

1. Shimpei Sato and Kenji Kise. ArchHDL: A New Hardware Description Language for High-Speed Architectural Evaluation. In Proceedings of the 7th IEEE International Symposium on Embedded Multicore/Many-core System-on-Chip (MCSoC '13), pp. 107–112, September 2013.
2. Yuichiro Tanaka, Shimpei Sato, and Kenji Kise. The Ultrasmall Soft Processor. In Proceedings of the 4th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART '13), pp. 63–68, June 2013.

3. Takakazu Ikeda, Shinya Takamaeda-Yamazaki, Naoki Fujieda, Shimpei Sato, and Kenji Kise. Read Density Aware Fair Memory Scheduling. The 3rd JILP Workshop on Computer Architecture Competitions (JWAC-3): Memory Scheduling Championship (MSC), June 2012.
4. Shinya Takamaeda, Shimpei Sato, Takefumi Miyoshi, and Kenji Kise. SmartCore System for Dependable Many-core Processor with Multifunction Routers. In Proceedings of the 1st International Conference on Networking and Computing (ICNC '10), pp. 133–139, November 2010.
5. Shintaro Sano, Masahiro Sano, Shimpei Sato, Takefumi Miyoshi, and Kenji Kise. Pattern-based Systematic Task Mapping for Many-core Processors. In Proceedings of the 2nd Workshop on Ultra Performance and Dependable Acceleration Systems (UPDAS '10), pp. 173–178, November 2010.
6. Koh Uehara, Shimpei Sato, Takefumi Miyoshi, and Kenji Kise. A Study of an Infrastructure for Research and Development of Many-Core Processors. In Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT '09), pp. 414–419, December 2009.
7. Shimpei Sato, Naoki Fujieda, Akira Moriya, and Kenji Kise. Processor Simulator SimCell to Accelerate Research on Many-core Processor Architectures. Proceedings of the Workshop on Cell Systems and Applications (WCSA), pp. 119–127, June 2008.

## 国内会議（査読付き）

1. 佐野伸太郎, 佐野正浩, 佐藤真平, 三好健文, 吉瀬謙二. メニーコアプロセッサの性能向上を達成するパターンに基づいたタスク配置手法. 第 8 回先進的計算基盤システムシンポジウム論文集（SACSIS 2010）, pp. 167–174, 2010 年 5 月.
2. 佐藤真平, 植原昂, 吉瀬謙二. メニーコアプロセッサのオンチップネットワーク性能を向上させる SmartCore システム. 第 7 回先進的計算基盤システムシンポジウム論文集（SACSIS 2009）, pp. 27–35, 2009 年 5 月.
3. 植原昂, 佐藤真平, 高前田伸也, 渡邉伸平, 吉瀬謙二. メニーコアプロセッサの HW/SW 研究開発を加速する実用的な基盤環境. 第 7 回先進的計算基盤システムシンポジウム論文集（SACSIS 2009）, pp. 199–207, 2009 年 5 月.
4. 森谷章, 藤枝直輝, 佐藤真平, 吉瀬謙二. メニーコアプロセッサに向けたデータ供給を

支援する多機能キャッシュコア. 第 6 回先進的計算基盤システムシンポジウム論文集（SACSIS 2008）, pp. 421–430, 2008 年 6 月.
5. 佐藤真平, 藤枝直輝, 田原慎也, 吉瀬謙二. 実用的かつコードのシンプルさを追求した Cell BE の機能レベルシミュレータ SimCell の設計と実装. 第 19 回コンピュータシステム・シンポジウム論文集（ComSys 2007）, pp. 39–47, 2007 年 11 月.

## 研究会

1. 佐藤真平, 吉瀬謙二. ArchHDL によるハードウェア記述の実践. 情報処理学会研究報告 2014-ARC-208, No. 21, pp. 1–8, 2014.
2. 佐藤真平, 吉瀬謙二. ハードウェアモデリング言語 ArchHDL から Verilog HDL へのトランスレータの設計. 電子情報通信学会技術研究報告 CPSY2013-10-03, Vol. 113, No. 234, pp. 1–6, 2013 年 10 月.
3. 笹河良介, 佐藤真平, 吉瀬謙二. 高信頼メニーコアシステム SmartCore における TMR 実行の提案と評価. 電子情報通信学会技術研究報告 CPSY2013-10-03, Vol. 113, No. 234, pp. 7–12, 2013 年 10 月.
4. 金子達哉, 佐藤真平, 吉瀬謙二. ArchHDL で記述したハードウェアの論理シミュレーションの高速化. 情報処理学会研究報告 2013-ARC-206, No. 25, pp. 1–8, 2013 年 7 月.
5. 佐藤真平, 吉瀬謙二. C++ をベースとする新しいハードウェア記述の検討. 情報処理学会研究報告 2013-ARC-205, No. 75, pp. 1–7, 2013 年 4 月.
6. 田中雄一郎, 笹河良介, 佐藤真平, 吉瀬謙二. 世界最小ソフトプロセッサの設計と応用. 情報処理学会研究報告 2013-EMB-28, No. 26, pp. 1–6, 2013.
7. 笹河良介, 佐藤真平, 吉瀬謙二. 2 次元メッシュ上のマルチキャスト通信における使用仮想チャネル数の削減. 情報処理学会研究報告 2012-ARC-199, No. 2, pp. 1–4, 2012 年 3 月.
8. 高前田（山崎）伸也, 佐藤真平, 吉瀬謙二. 高機能ルータを利用した DMR 実行メニーコアにおける効率的なタスク配置手法の検討. 情報処理学会研究報告 2012-ARC-199, No. 3, pp. 1–8, 2012 年 3 月.
9. 池田貴一, 佐藤真平, 吉瀬謙二. 冗長実行時の SmartCore システムの性能評価. 情報処理学会研究報告 2011-ARC-197, No. 32, pp. 1–8, 2011 年 11 月.
10. 佐藤真平, 笹河良介, 吉瀬謙二. Distributed Shared-Buffer NoC ルータのためのパイプ

ラインバイパス手法の改良. 情報処理学会研究報告 2011-ARC-196, No. 4, pp. 1–9, 2011 年 7 月.

11. 姜軒, 佐藤真平, 吉瀬謙二. Distributed Shared-buffer ルータの遅延を削減するパイプラインバイパス方式. 情報処理学会研究報告 2011-ARC-194, No. 13, pp. 1–10, 2011 年 3 月.
12. 植原昂, 佐藤真平, 佐野伸太郎, 吉瀬謙二. メニーコアプロセッサの研究・教育を支援する実用的な基盤環境 M-Core. 情報処理学会研究報告 2010-ARC-188, No. 8, pp. 1–10, 2010 年 2 月.
13. 佐藤真平, 植原昂, 三好健文, 吉瀬謙二. SmartCore システムによるメニーコアプロセッサの信頼性向上手法. 情報処理学会研究報告 2010-ARC-187, No. 13, pp. 1–6, 2010 年 1 月.
14. 佐野伸太郎, 佐野正浩, 佐藤真平, 三好健文, 吉瀬謙二. メニーコアプロセッサのためのネットワークトラフィックに着目したタスク配置問題の解析と考察. 電子情報通信学会技術研究報告 CPSY2009-11-20, Vol. 109, No. 296, pp. 31–36, 2009 年 11 月.
15. 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. 情報処理学会研究報告 2009-ARC-183, No. 6, pp. 1–8, 2009 年 4 月.
16. 吉瀬謙二, 植原昂, 佐藤真平. メニーコアプロセッサのディペンダビリティ向上と高性能化を目指す SmartCore システム. 情報処理学会研究報告 2008-ARC-180, No. 101, pp. 49–52, 2008 年 10 月.
17. 植原昂, 佐藤真平, 森谷章, 藤枝直輝, 高前田伸也, 渡邉伸平, 三好健文, 小林良太郎, 吉瀬謙二. シンプルで効率的なメニーコアアーキテクチャの開発. 情報処理学会研究報告 2008-ARC-180, No. 101, pp. 39–44, 2008 年 10 月.
18. 藤枝直輝, 佐藤真平, 吉瀬謙二. 二重分岐ヒントを考慮したソフトウェア分岐予測の可能性検討. 情報処理学会研究報告 2008-ARC-177, No. 19, pp. 121–126, 2008 年 3 月.
19. 森谷章, 藤枝直輝, 佐藤真平, 吉瀬謙二. 多機能メニーコアにおけるデータ供給を支援するキャッシュコアの提案. 情報処理学会研究報告 2008-ARC-176, No. 1, pp. 53–58, 2008 年 1 月.
20. 佐藤真平, 藤枝直輝, 田原慎也, 吉瀬謙二. Cell BE 機能レベルシミュレータの設計と実装. 情報処理学会研究報告 2007-ARC-174, No. 79, pp. 187–192, 2007 年 8 月.

# ポスター・全国大会など

1. Shimpei Sato and Kenji Kise. Ultra-High Speed Architectural Simulation Methodology. The 16th International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA '14), March 2014.
2. Haruka Mori, Shimpei Sato, Chu Van Thiem, and Kenji Kise. Design and Implementation of Manycore Processor for a Large FPGA. The 76th National Convention of IPSJ, March 2014.
3. 笹河良介, 佐藤真平, 吉瀬謙二. NoC におけるロングエッジファースト（LEF）ルーティングの提案. 第 75 回情報処理学会全国大会, 2013 年 3 月.
4. 笹河良介, 佐藤真平, 吉瀬謙二. SmartCore システムのデッドロック回避. 第 74 回情報処理学会全国大会, 2012 年 3 月.
5. 佐藤真平, 吉瀬謙二. メニーコアプロセッサの空間冗長性を利用する TMR の提案. 第 73 回情報処理学会全国大会, 2011 年 3 月.
6. Shimpei Sato, Shinya Takamaeda, and Kenji Kise. DMR mode of SmartCore system. Poster session at the 16th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '10), December 2010.
7. 高前田伸也, 佐藤真平, 三好健文, 吉瀬謙二. メニーコアアーキテクチャの HW 評価環境 ScalableCore システムの活用〜ディペンダブルプロセッサの実装〜. 第 8 回先進的計算基盤システムシンポジウムポスターセッション（SACSIS 2010）, 2010 年 5 月.
8. 佐藤真平, 植原昂, 三好健文, 吉瀬謙二. SmartCore システムによるメニーコアプロセッサの信頼性向上手法. 情報処理学会研究報告 2010-ARC-187, No. 13, pp. 1—6, 2010 年 1 月.
9. 植原昂, 佐藤真平, 三好健文, 吉瀬謙二. 洗練されたメニーコアアーキテクチャの開発. 第 72 回情報処理学会全国大会, 2010 年 3 月.
10. 佐野伸太郎, 佐野正浩, 佐藤真平, 三好健文, 吉瀬謙二. メニーコアプロセッサの性能向上を目指すタスク配置手法. 第 72 回情報処理学会全国大会, 2010 年 3 月.
11. Shinya Takamaeda, Shimpei Watanabe, Shimpei Sato, Koh Uehara, Yuhta Wakasugi, Naoki Fujieda, Yosuke Mori, and Kenji Kise. ScalableCore : High-Speed Prototyping System for Many-Core Processors. Poster session at the 12th IEEE International Symposium on Low-Power and High-Speed Chips (COOL Chips XII), p. 161, April

2009.

12. 吉瀬謙二, 佐藤真平, 森谷章, 藤枝直輝, 若杉祐太, 渡邉伸平, 植原昂, 森洋介, 高前田伸也, 高橋朝英, 棟岡朋也, 山田裕介, 権藤克彦, 小林良太郎, 三好健文, 中條拓伯. MieruPC プロジェクト：中身が見える計算機システムを構築する研究・教育プロジェクト. 第 20 回コンピュータシステム・シンポジウムポスター・デモセッション（ComSys 2008）, 2008 年 11 月.
13. 佐藤真平, 森谷章, 吉瀬謙二. 計算機アーキテクチャ研究を加速するプロセッサシミュレータ SimCell の開発とその応用. 第 13 回電子情報通信学会東京支部学生会研究発表会, 2008 年 3 月.
14. 佐藤真平, 藤枝直輝, 吉瀬謙二. 計算機アーキテクチャ研究を加速する Cell/B.E. のプロセッサシミュレータ SimCell のススメ. The 4th IEEE Tokyo Young Researchers Workshop ポスターセッション, 2007 年 12 月.
15. 森谷章, 藤枝直輝, 佐藤真平, 吉瀬謙二. マルチコア・プロセッサにおける SimCell を用いたキャッシュコアの可能性の検討. 第 19 回コンピュータシステム・シンポジウムポスター・デモセッション（ComSys 2007）, 2007 年 11 月.
16. 佐藤真平, 吉瀬謙二. 多数コアを集積する CMP における階層型ネットワークの検討. 第 69 回情報処理学会全国大会, 2007 年 3 月.