東京工業大学工学部

学士論文

大規模 FPGA をターゲットとする
実用的なメニーコアプロセッサ

指導教員　吉瀬 謙二　准教授

平成 26 年 2 月

提出者

学科　情報工学科

学籍番号　10_23871

氏名　森 悠

指導教員　印	

学科長　認定印	

平成 25 年度　学士論文内容梗概

大規模 FPGA をターゲットとする
実用的なメニーコアプロセッサ

指導教員　吉瀬 謙二　准教授
情報工学科
10_23871　森 悠

シングルコアプロセッサの性能限界や集積技術の向上により，チップ上に簡素なアーキテクチャのプロセッサコアを多数搭載したメニーコアプロセッサが一般的になりつつある．現在もメニーコアプロセッサに関する研究が多く為されているが，今後プロセスの微細化によってより多くのプロセッサコアが搭載されるようになると，回路規模が爆発的に増大し，従来のソフトウェアシミュレータでは現実的なプロトタイピングを行うことが難しくなる．

ハードウェアを用いたプロトタイピングでは ASIC(Application Specific Integrated Circuit) や FPGA(Field Programmable Gate Array) を用いるのが一般的であるが，FPGA はチップ内の論理回路を再構成できるため，プロセッサアーキテクチャ研究に適している．

本論文ではプロセッサアーキテクチャ研究に適した，シンプルで実用的なメニーコアプロセッサの設計を行う．1 チップの大規模 FPGA をターゲットとして 36 ノード構成のプロセッサを実装する．多数のプロセッサコアを利用するアプリケーションを FPGA 実装したメニーコアプロセッサを動作させ，動作検証を行う．また，メニーコアプロセッサのアーキテクチャ性能の評価や，ソフトウェアシミュレータとのシミュレーション性能の比較・評価を行う．

C 言語で記述された N-Queens 問題を解くアプリケーションを用い，アーキテクチャ性能およびシミュレーション性能を評価した結果，アプリケーションの実行に用いたプロセッサコア数の増加に対し，FPGA 実装したメニーコアプロセッサの性能はほぼ線形のオーダで向上した．FPGA 実装したメニーコアプロセッサにおけるアプリケーション実行は，ソフトウェアシミュレータに対して最大 560 倍高速であった．

目次

第 1 章

緒論

1.1 研究の目的

シングルコアプロセッサの性能限界や集積技術の向上により，時代はメニーコアプロセッサへ遷移しつつある．それを受けて，メニーコアプロセッサに関する多くの研究が為されている．メニーコアプロセッサ研究ではサイクルレベルの評価のためにソフトウェアシミュレータやハードウェアによるプロトタイプが用いられるが，ソフトウェアシミュレータは低速であるため，ハードウェアによるプロトタイプが頻繁に用いられる．

ハードウェアによるプロトタイピングには ASIC(Application Specific Integrated Circuit) や FPGA(Field Programmable Gate Array) が用いられる．FPGA は ASIC に比べて動作周波数が低くなるという欠点がある反面，チップ内の論理回路を再構成できるという利点があるため，プロセッサアーキテクチャ研究におけるプロトタイピングに適している．特に近年の集積回路製造プロセスの微細化により，1 チップの FPGA で実現可能な回路規模が増大している．このため，メニーコアプロセッサのような複雑で大規模な回路でも 1 チップの FPGA に実装することができる．また，FPGA の多くは構造を同じくするという特徴を持つので，多くの労力を費やすことなくボード間の移植を行うことができる．FPGA をターゲットとするプロセッサの回路情報が OpenCores[1] の様に提供されれば，ユーザは手元の FPGA に回路を実装するだけで，評価環境を容易にセットアップすることができる．

本研究の目的は，メニーコアアーキテクチャ研究のための評価環境として活用できる，シンプルで実用的なメニーコアプロセッサを大規模 FPGA をターゲットに設計・実装することである．また，本論文では設計・実装したメニーコアプロセッサのハードウェア量と性能の評価を行う．

1.2　本論文の構成

本論文の構成は以下の通りである．2 章ではメニーコアプロセッサと FPGA の概要について述べる．3 章では設計したメニーコアプロセッサのアーキテクチャに関して，4 章では設計したメニーコアプロセッサの FPGA 実装に関して述べる. 5 章では設計したメニーコアプロセッサのハードウェアリソースと性能の評価を行う．最後に，6 章で本論文の結論をまとめる.

第 2 章

研究の背景

2.1 メニーコアアーキテクチャ

近年ではメニーコアアーキテクチャの研究が進み，メニーコアプロセッサが現実のものとなっている．また，度重なる試作を経て，企業によって民生・業務用として製品化されたプロセッサも存在する．

2.1.1 Intel メニーコアアーキテクチャ

Intel はシングルコアプロセッサの性能限界に直面してから，メニーコアプロセッサ開発に対する取り組みを続けている．その一つが，Tera-scale Computing[2] である．Tera-scale Computing は RMS(Recognition，Mining，Synthesis) 処理を高いレベルで実現することを目的としている．Intel は，今後 RMS 処理の需要が高まるにつれて，Tflops クラスの処理性能，TB/s クラスのメモリ帯域幅，Tbps クラスの I/O 帯域幅が必要とされることを予想しており，これらの共通項である Tera-scale がプロジェクト名の由来となっている．

Tera-scale Computing で試作チップ化された代表的なアーキテクチャとして，Polaris Teraflops Research Chip[3] と SCC(Single-chip Cloud Computer)[4] が挙げられる．Polaris は 80 コアプロセッサのプロトタイプで，2 次元メッシュ型の Network-on-Chip である．各ノードはコア・キャッシュ・ルータを各 1 基ずつ含み，コアは 2 基の浮動小数点演算器によって構成される．このような簡素な構造により，Polaris は 1 つのチップに多くのコアを搭載することができた．実際に試作されたチップでは 1.01〜1.81Tflops の処理性能を達成している．

SCC は Polaris の後に開発された 48 コアプロセッサのプロトタイプで，クラウドデータセンターを 1 チップで模倣するというコンセプトのもとに開発された．ネットワークトポロジは Polaris 同様，2 次元メッシュ型の Network-on-Chip であるが，内部接続の改良により，SCC は 256GB/s という高速なノード間通信を達成している．また，コアはシングルスレッドプロセッサのうち，最も成熟したモデルとされる P54C をベースとしており，このコアを 2 基，共有 L2 キャッシュとルータをそれぞれ 1 基まとめたノードはタイルと呼ばれる．更に 4 タイルをまとめたものは島と呼ばれ，6 島をまとめたものが 1 つのチップになっている．この階層構造を利用し，タイルごとに動作周波数を，島ごとに電圧を制御することにより，SCC は 48 コアを同時に動作させたときの消費電力を 25∼125W まで抑えることができる．

一方，は Tera-scale Computing と同時期に進められていた Intel のもう一つのメニーコア戦略として，HPC を目的とした CPU および GPU のアーキテクチャである Larrabee が挙げられる．当時，GPU の分野では NVIDIA や ATI の製品が主流であり，Intel の Larrabee はそれらと競合する予定であった．Larrabee は 16 コアプロセッサであり，ノードや GPU のための機能ブロックが双方向リングバスにより接続される．コアは P54C をベースとして 16 基のベクタ演算ユニットを追加したものを備える．Intel は Larrabee に続き，プロセスルールを微細化し 32 コアを搭載する Larrabee2 などを計画していたが，当時の他社 GPU に性能面で対抗できないことから，Larrabee2 の開発は一時中止された．しかしその後，メニーコアプロセッサを搭載するアクセラレータの価値が高まったことや，ホスト PC の処理の多くを実行できる Larrabee 系列アーキテクチャの有用性が再評価されたことにより，MIC(Many Integrated Core) アーキテクチャ [5] へと継承された．

MIC アーキテクチャの一つ，Knights Corner は Larrabee と同様の構造を持つ 62 コアプロセッサであり，メニーコアプロセッサを搭載するアクセラレータとして，Xeon Phi の名で製品化されている．Xeon Phi は Linux をスタンドアロンで動作させることが可能であり，ホストコンピュータから PCI-Express を通じて通信を行うことで制御が可能となっている．

図 2.1 に Intel が開発したメニーコアプロセッサのアーキテクチャ遷移を示す．最新の Knights に至るまでの本流は Larrabee に始まるが，その過程である Knights Corner において，Polaris，SCC によって培われたパワーゲーティングや高速なノード間通信技術が生かされていると推測される．

また，今後開発が予定されている Knights Landing は On-package DRAM を採用するほか，P54C ベースのコアではなく Out-of-order 実行可能な Silvermont ベースのコアを採用することや，単体で動作可能なソケット版の開発も行われていることが知られてい

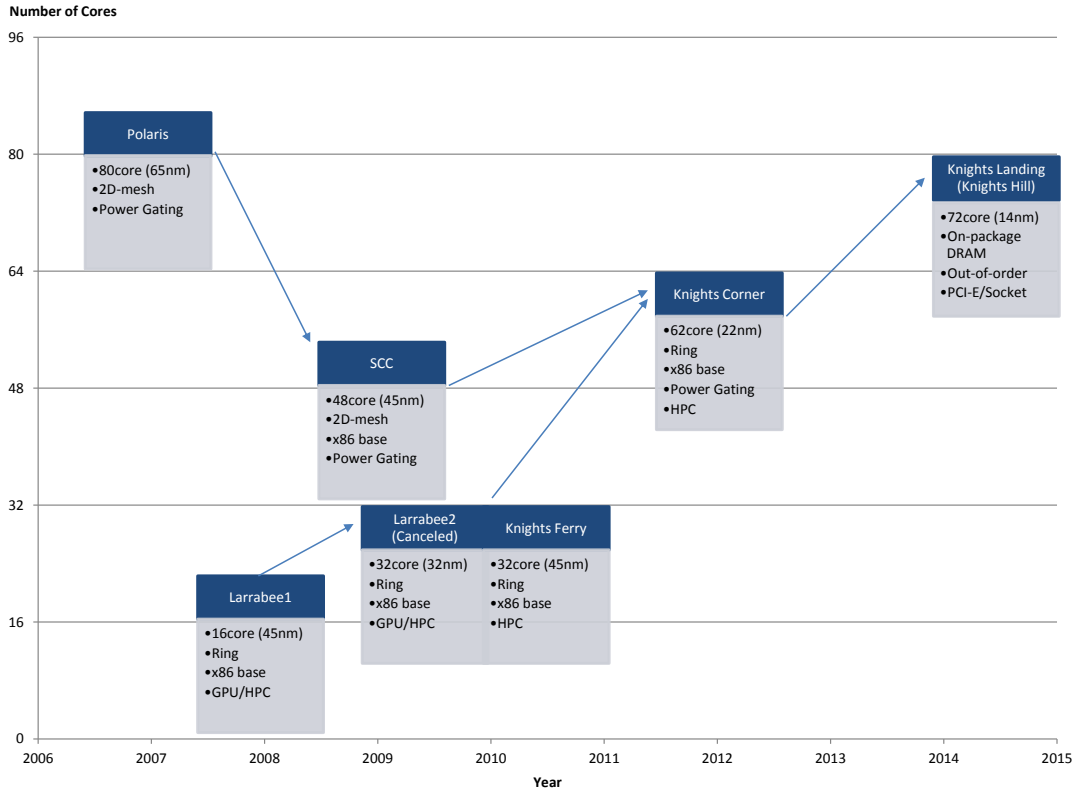


図 2.1　Intel が開発したメニーコアプロセッサ

る．このことから，今後 Intel はメイン CPU においてもメニーコアプロセッサを導入し，同時に個々のコアの性能向上を目指していくことが予想される．

2.1.2　KALRAY メニーコアアーキテクチャ

KALRAY が開発したメニーコアプロセッサ，MPPA MANYCORE[6] は 256 コア，512 コア，1024 コアのプロセッサであり，ASIC が持つ性能と GPU が持つソフトウェア柔軟性の両立をするとされる．MPPA MANYCORE は 2 次元トーラス型の Network-on-Chip であるが，コア数の異なるいずれのモデルにおいても，16 のクラスタと 4 つの I/O サブシステムがネットワークによって接続される．コア数によって異なるのは各クラスタの内部構成であり，各クラスタは 16~64 基のコア，共有メモリ，1 基のシステムコアから成る．Network-on-Chip を採用したことにより，複数のチップをボードレベルで接続することでどのようなサイズのメニーコアプロセッサも構成することが可能である．KALRAY は MPPA MANYCORE のほか，FPGA ボードなどの開発キットなども提供している．

2.2　FPGA 技術

FPGA(Field Programmable Gate Array) はその名の通り，再構成可能な論理ゲートアレイである．FPGA は論理回路ブロック，I/O 部，配線チャネル，BRAM(Block RAM) といった論理回路を構成する基本要素を有している．論理回路ブロックは書き換え可能な LUT(Look up Table) とフリップフロップから構成され，Xilinx では論理セルと呼称されている．I/O 部は外部との信号のやりとりを行うポートで，FPGA の外縁部に位置する．配線チャネルは論理セルや I/O 部を接続するための内部配線アレイであり，配線アレイの交点にはプログラマブルスイッチが配置される．BRAM は FPGA 上に均等に配置され，必要に応じて RAM として利用することができる．

FPGA で利用可能なハードウェアリソースは，一般的に論理回路ブロックの LUT の入力数や論理回路ブロック数，BRAM の容量によって決まる．例えば，一般的な 32bit MIPS アーキテクチャのパイプラインプロセッサを FPGA に実装することを考える．このプロセッサの実装に約 420 個のスライスが必要であるとすると，Xilinx Spartan-6 XC6SLX45 を実装対象とした場合，XC6SLX45 のスライス数は 6,822 個であるので，このプロセッサを約 16 個実装することができる．ここで，スライスとは，Xilinx FPGA における CLB(Configurable Logic Block) を構成する要素である．また，Xilinx Virtex-7 XC7VX485T を実装対象とした場合，XC7VX485T のスライス数は 75,900 個であるので，このプロセッサを約 180 個実装することができる．

2.3　関連研究

FPGA を用いたメニーコアシミュレータに関する研究として，FAST[7] や ATLAS[8], ProtoFlex[9] や RAMP Gold[10] などが挙げられる．FAST は PCB(Printed Circuit Board) に実装されるシミュレーション環境であり，PCB 上に 4 個のプロセッサタイルを持つ．プロセッサタイルは CPU，FPU，L1 キャッシュ，そして複数の FPGA から構成され，複数の CPU を用いた高速なシミュレーションを実現している．ATLAS は PCB 上に複数の FPGA を搭載し，HTM(Hardware Transactional Memory) をサポートすることでより実用的なシミュレーションを可能としている．ProtoFlex は単一の FPGA に実装され，FPGA 上で 16 個のプロセッサを持つ．ハードウェアに比べ，複雑な処理をソフトウェアで処理することでフルシステムのシミュレーション環境を実現している．RAMP Gold は FPGA を用いたメニーコアのシミュレータである．RAMP Gold は 1 基

の Virtex-5 に実装され，共有メモリの 64 コア構成のメニーコアの動作をシミュレーションすることができる．システムは個々の機能ユニットのシミュレーションを行う部分とタイミングを調停する部分に分割されており，これとマルチスレッディングを併用することで高速なシミュレーションを可能としている．

これらのメニーコアシミュレータが大容量な FPGA をターゲットとしているのに対し，ScalableCore System[11] は小容量の FPGA を搭載したボードを多数接続したシステムであり，64 ノードのメニーコアシミュレーションにおいてソフトウェアシミュレータの 14.2 倍の高速化を達成している．

第 3 章

実用的なメニーコアプロセッサの設計

3.1 プログラミングモデル

メニーコアプロセッサは FPGA に実装され，アクセラレータとして機能することを想定して設計を行う．利便性の高い機能を付加するとプロセッサの可用性が高まるが，構造が複雑化してしまう．そのため，本論文ではシンプルな構成のメニーコアプロセッサが FPGA 上で確実に動作することを設計目標とする．この設計目標に従い，設計をシンプルにするため，メニーコアプロセッサ単体での OS 動作はサポートしない．

図 3.1 にプログラミングモデルを示す．ユーザはホストコンピュータと FPGA を搭載したボードを物理的に接続し，TeraTerm などの端末エミュレータを通してボードを制御する．動作開始時，ユーザはメニーコアプロセッサが実行するアプリケーションを USB を介したシリアル通信によって FPGA に転送する．メニーコアプロセッサの実行結果は同様のシリアル通信によってホストコンピュータに転送され，端末エミュレータに表示される．

一般的なメニーコアプロセッサは複数のプロセッサコアを有しており，並列プログラミングにより複数のコアを協調動作させることによって，処理全体でのスループットを向上することができる．設計するプロセッサにおいても並列アプリケーションを動作させることは設計目標の一つであるが，前述した設計目標のために，並列アプリケーションの動作を想定していない．よって，ユーザは複数のアプリケーション間で依存関係が生じることのないプログラム設計を要求される．

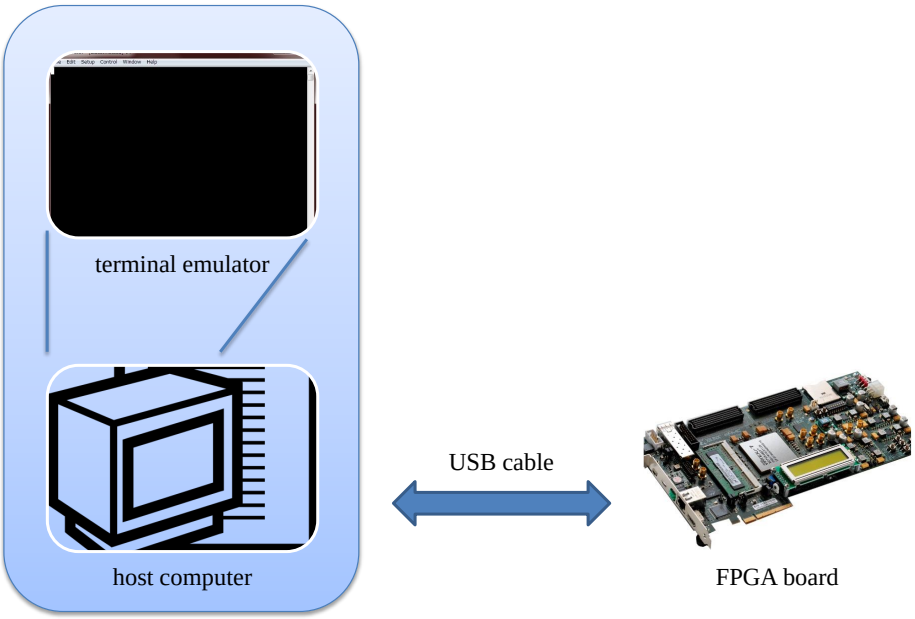


図 3.1　プログラミングモデル

プロセッサコアは拡張性よりも単純性を重視し，MIPS アーキテクチャを採用する．プログラマは C などの高水準言語で記述されたアプリケーションをメニーコアプロセッサで動作させるために，MIPS クロスコンパイラ等を用いてアプリケーションをバイナリ形式の実行ファイルに変換する必要がある．使用可能なメモリ領域は 1 つの実行ファイルにつき 64KB であり，複数の実行ファイルを転送する必要がある場合は一括して転送を行う．

ノード間通信などの複雑な処理はハードウェアが行い，メモリマップド I/O を用いてその制御を行う．**表** 3.1 にプロセッサコアが利用できるのメモリマップド I/O のメモリマップを示す．アドレスは該当するアドレスを，r/w はアドレスに対する読みこみか書き込みかを，bit width はアドレスにおける有効なビット幅を，node type はメモリマップの利用が想定されるノードタイプを，activity は具体的な動作をそれぞれ表している．ノードタイプは P がプロセッサノードを，S がスケジューラノードを示す．ノードタイプに関する詳細は次節にて後述する．

表 3.1　メモリマップド I/O のメモリマップ

address	r/w	bit width	node type	activity
0x00	read	32bit	P/S	get value of cycle counter
0x00	write	8bit	P/S	output text data
0x04	write	0bit	P/S	finalize program
0x08	write	32bit	P/S	set destination for scheduling
0x0c	write	32bit	P/S	set value for scheduling
0x10	write	32bit	P/S	set application ID for scheduling
0x14	read	32bit	P/S	get value of status register
0x18	read	32bit	S	get value of summation register

アドレス 0x00 に対するリードは，動作開始時点からの経過サイクル数を取得する．アドレス 0x00 に対するライトは，下位 8bit をテキスト表示として出力する．アドレス 0x04 に対するライトは，プログラム終了時のファイナライズ処理であり，メニーコアプロセッサがプログラムの終了を判定するために用いる．書き込み内容は破棄される．アドレス 0x08，0x0c，0x10 に対するライトはスケジューリングに必要な情報を書き込む．0x14 に対するリードは，ステータスレジスタの値を取得する．ステータスレジスタの用途はノードによって異なり，スケジューラノードから受け取った値などが格納される．0x18 に対するリードは，加算レジスタの値を取得する．加算レジスタはスケジューラノードのみが持ち，プロセッサノードから受け取った値の和が格納される．

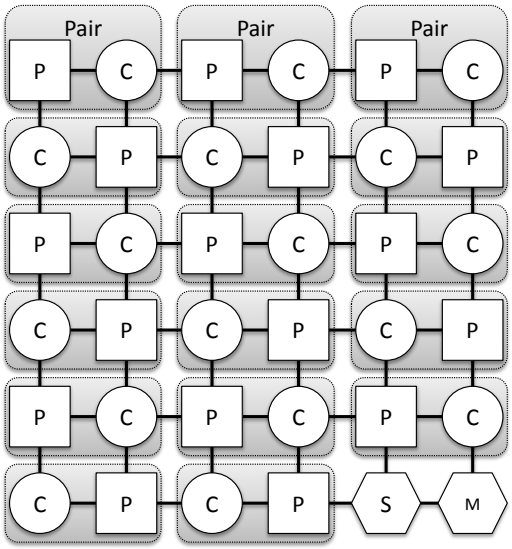


図 3.2　36 ノード構成のメニーコアプロセッサの例

3.2　メニーコアアーキテクチャ

3.2.1　ノードアレイ

図 3.2 に 36 ノード構成のメニーコアプロセッサの例を示す．ノードはプロセッサノード (P)，キャッシュノード (C)，スケジューラノード (S)，メモリノード (M) の 4 種が存在する．各ノードは 2 次元メッシュネットワークで接続され，これを介したパケット交換によりノード間通信が行われる．実線はノード間の物理的配線，色掛けされた領域はプロセッサノードとキャッシュノードのペアを示している．キャッシュノードはプロセッサノードとペアを組むことで，プロセッサノードのプライベートキャッシュとして機能する．プロセッサノードとキャッシュノードは交互に配置されており，ペアは 17 組存在する．ペアは行方向に隣り合うノード同士で組まれ，スケジューラノードとメモリノードは右下端に配置される．また，ノードアレイの配置方法やノード数はユーザが変更することができる．

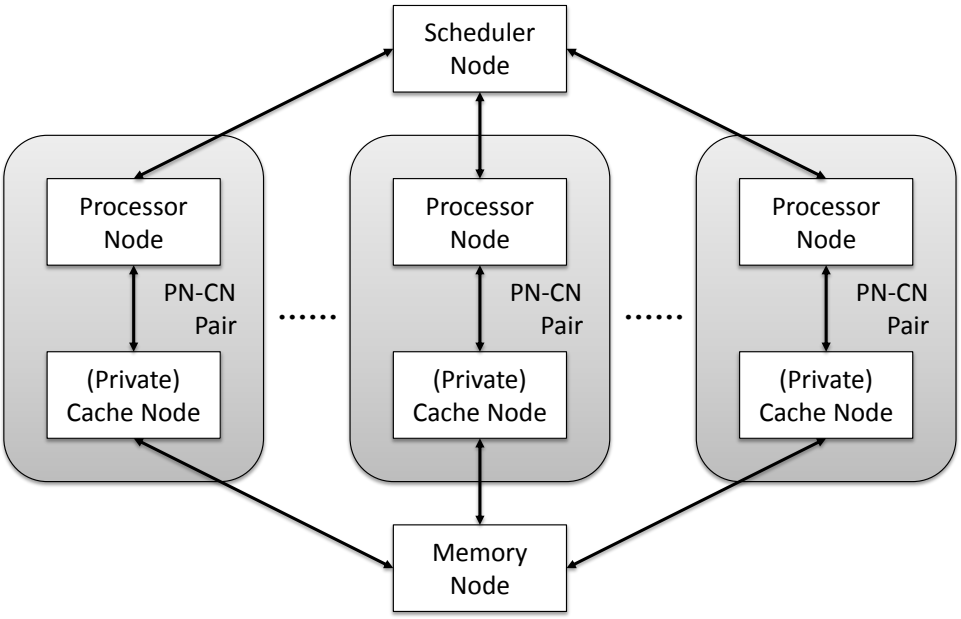


図 3.3　ノードタイプとパケットフロー

3.2.2　ノードタイプ

図 3.3 にメニーコアにおけるノードタイプと，それらの間で行われる通信のパケットフローを示す．ノードタイプはスケジューラノード，プロセッサノード，キャッシュノード，メモリノードの 4 種類が存在する．これらのノードはネットワークを介したパケット交換によってデータ転送や制御を行う．

スケジューラノードとプロセッサノードは内部にプロセッサコアを有しており，プログラムに準じて動作するノードである．一方，キャッシュノードとメモリノードは内部に RAM(Random Access Memory) を有しており，他ノードからのデータ転送要求に応じて読み書きを行うノードである．

スケジューラノードとプロセッサノードの間では，スケジューラノードによる動作指示とプロセッサノードによる動作完了通知がパケット交換によって行われる．各プロセッサノードはスケジューラノードの指示なしに動作を開始することはできない．

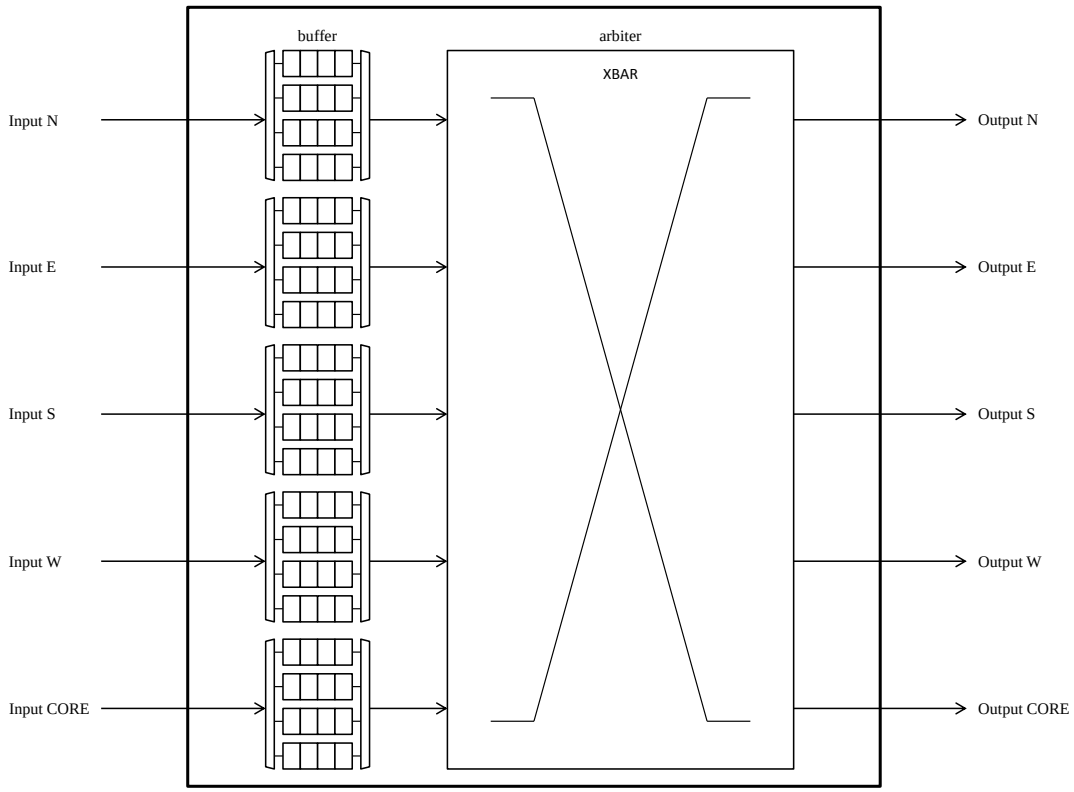


図 3.4 インプットバッファルータのアーキテクチャ

プロセッサノードとキャッシュノードの間では，プロセッサノードからのデータ転送要求とキャッシュノードによる応答がパケット交換によって行われる．各キャッシュノードはプロセッサノードのプライベートキャッシュであり，ノード数はプロセッサノードの数と同じである．

キャッシュノードとメモリノードの間では，キャッシュノードからのデータ転送要求とメモリノードによる応答がパケット交換によって行われる．パケット交換の内容はプロセッサノードとキャッシュノードの間におけるパケット交換の内容に極めて近いが，メモリノードは複数のノードからの要求を処理するという点で異なる．

3.2.3 ネットワークアーキテクチャ

2 次元メッシュネットワークにおいて，ノードに内蔵されるルータは上下左右の 4 ノードの方向に加え，自ノードの方向を加えた，計 5 ノードの方向の接続を仲介する役割を持つ．

メニーコアプロセッサのルータには，一般的なインプットバッファルータを採用する．図 3.4 にインプットバッファルータのアーキテクチャを示す．インプットバッファルータは各方向のパケット入力側に有限容量のバッファを持ち，バッファに格納された情報をもとにルーティングを行う．パケットの出力先のバッファに空きがある時はパケットを転送するが，そうでない場合はパケットを転送することができず，パケットはバッファ内に留まり続ける．

複数のルータによる依存関係が循環する循環依存が生じると，互いの出力先のルータのバッファに空きが出るのを待つような待ち合わせが同時に起こり，デッドロックが発生する．このようなデッドロックの発生を回避する手段として，次元順ルーティングアルゴリズムが知られており，設計するメニーコアプロセッサのルーティングアルゴリズムにはこれを採用する．

設計するメニーコアプロセッサのノード間通信には，要求に対する応答が必ず存在する．メモリノードやスケジューラノードのように，複数のノードからの要求や応答を受け付けるノードは同時に複数の要求を処理することががができない．そのため，非処理中の要求パケットは，処理中の要求に対する応答が完了するまでバッファに留まる．このとき，残留中の要求パケットと処理中の応答パケットの間に循環依存が生じる可能性がある．このようなデッドロックの発生を回避する技術として，物理的に 1 本の回線を仮想的に複数の回線として扱う VC(Virtual Channel) が知られている．

ルータにおいては，バッファを複数本用意し，用途に応じて使い分けることで VC が実現可能である．今回の場合，要求と応答を異なる回線としてバッファを使い分けることでデッドロックが回避される．ノード間通信におけるパケットフローを考慮すると，プロセッサノードとスケジューラノードの間で行われる要求と応答にそれぞれ VC0 と VC1 を，キャッシュノードとメモリノードの間で行われる要求と応答にそれぞれ VC2 と VC3 を利用する．プロセッサノードとキャッシュノードの間で行われる通信はペア間のプライベート通信であり，処理中に互いのパケットがネットワーク上のバッファに留まることはない．前述の手法によってデッドロックの発生は防がれているため，既に他のノード間通信に割り当てられた VC を共用したとしてもデッドロックは発生しない．よって，プロセッサノードとキャッシュノードの間で行われる要求と応答にそれぞれ VC0 と VC1 を利用する．

このように，設計するメニーコアプロセッサでは，ノード間通信において次元順ルーティングアルゴリズムを適用し，4 本の VC をパケットの内容によって使い分けることでデッドロックを回避している．

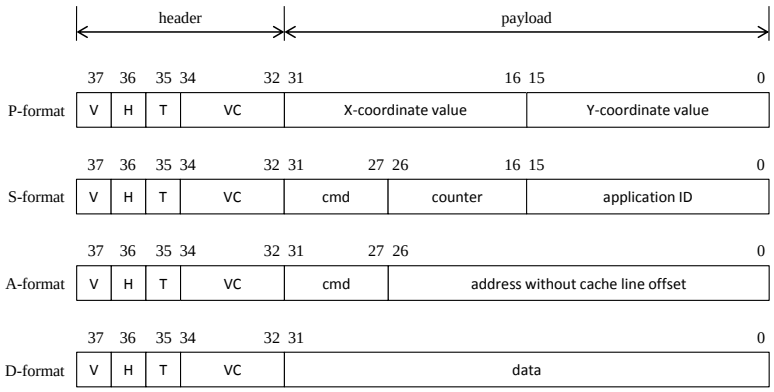


図 3.5 フリット形式

3.2.4 パケット構成

ノード間通信で用いられるパケットは複数のフリットから構成されている．メニーコアの各ノードは物理的には 38bit のパラレル信号線で配線されており，クロックサイクル毎に 38bit のフリットを伝送することができる．

図 3.5 にノード間通信に用いられるフリット形式を示す．フリット形式は大別して 4 種類に分類され，P/S/A/D-format が存在する．各フリットは上位 6bit のヘッダと下位 32bit のペイロードから構成される．ヘッダはパケット単位のルーティングに必要な情報であり，ルータはヘッダを用いて一連のフリットが同一のパケットであることを認識する．このため，ヘッダは全てのフリット形式において共有の仕様を持つ．

ヘッダの内容は上位から順番に有効ビット (V)，先頭ビット (H)，後尾ビット (T)，仮想チャネル (VC) である．V, H, T はそれぞれ，自身のフリットが有効であるか，パケットの先頭であるか，パケットの後尾であるかどうかを示している．VC はネットワーク上に残

留するパケットによって生じるデッドロックを回避するためのものであり，チャネル番号を示す．

ノード間通信ではパケットの宛先や送信元を区別しなければならないので，左上端のノードを基点とし，行方向を X 軸，列方向を Y 軸としたときの XY 座標で 2 次元メッシュ上のノードの位置を表すことにする．X 座標と Y 座標をそれぞれ 16bit で表現すると，これらの連接はノード固有の 32bit の値となる．これを physical ID とする．P-format のペイロードは physical ID そのものである．

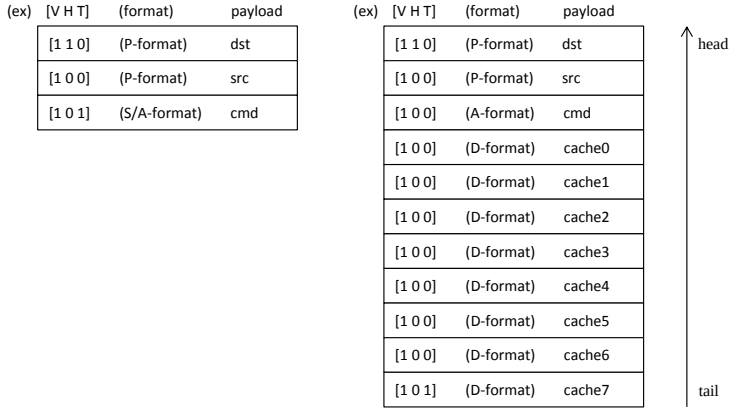
S-format および A-format のペイロードにはパケットの目的を示すコマンド (cmd) が含まれる．cmd はスケジューリングの動作指示と応答，キャッシュラインのフェッチ要求と応答，キャッシュラインのライトバック要求と応答のいずれかを示しており，ノードは cmd を参照することでパケット全体の要求が把握できる．

S-format はスケジューラノードとプロセッサノードの間でやりとりされる，特にスケジューリングのためのフォーマットである．ペイロードには動作指示のための application ID やデータが含まれる．ペイロードの内容に関しては次節にて後述する．

A-format および D-format はプロセッサノードやキャッシュノード，メモリノードの間で行われるキャッシュライン転送のためのフォーマットである．メニーコアにおけるキャッシュの転送は 8word のキャッシュライン毎に行われるので，キャッシュライン転送要求の際に完全なメモリアドレスを用いる必要はない．メモリアドレスの下位 2bit は 4byte メモリによるアライメントオフセットであり，更に下位 3bit は 8word キャッシュラインによるキャッシュラインオフセットであるので，キャッシュラインの指定のためにはメモリアドレスの上位 27bit さえあればよく，これがペイロードに含まれる．

一方，D-format はペイロード全体が 32bit のデータとなっており，キャッシュラインのフェッチ要求に対する応答やキャッシュラインのライトバックなど，キャッシュラインの実体がペイロードとなる．キャッシュラインオフセットはフリット順によって決定する．

図 3.6 にノード間通信に用いられるパケット形式を示す．**図** 3.6(a) の制御形式のパケットはパケットの送信先と送信元を示す 2flit の P-format と，パケットの要求を示す S-format または A-format の合計 3flit から構成される．この形式はパケットの目的が単純なキャッシュラインの転送要求やスケジューリングの場合に用いられる．一方，**図** 3.6(b) のデータ形式のパケットは**図** 3.6(a) の制御形式のフリットに 8flit の d-format を加えた合計 11flit から構成される．この形式はパケットの要求がキャッシュライン本体の転送の場合に用いられる．



(a) ctrl packet (b) data packet

図 3.6 パケット形式

3.2.5 プロセッサコア

プロセッサノード及びスケジューラノードには同一の仕様を持つプロセッサコアが内蔵される．

図 3.7 にプロセッサコアのデータパスを示す．プロセッサコアは 32bit のデータパスを持つ MIPS アーキテクチャのパイプラインプロセッサであり，パイプラインステージは命令フェッチ (IF)，命令デコード (ID)，実行 (EX)，メモリアクセス (MA)，ライトバック (WB) の 5 段である．

プロセッサコアの命令セットは MIPS アーキテクチャの主要な命令のサブセットであり，

- Load Double Word
- Store Double Word

を除く整数命令と

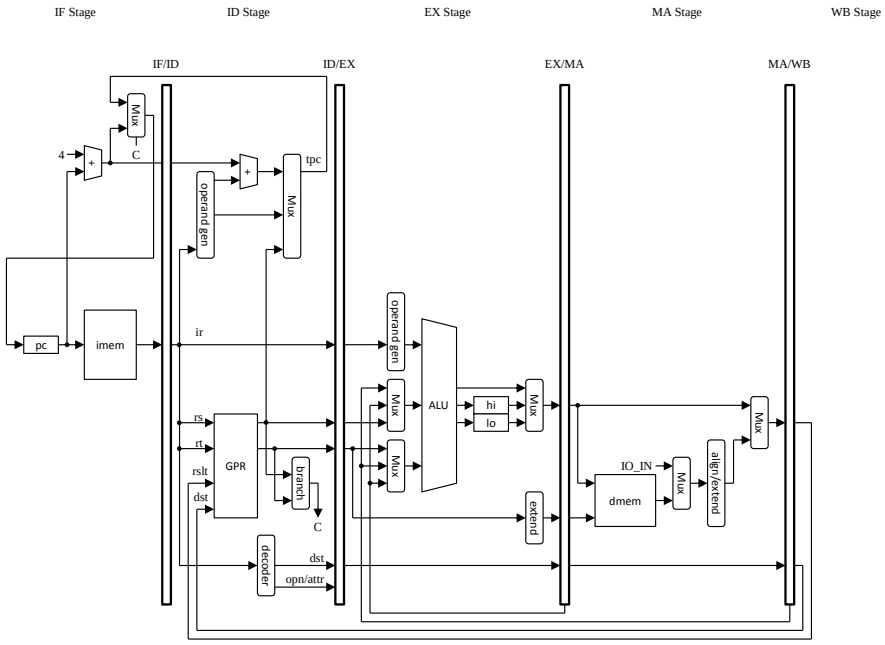


図 3.7　プロセッサコアのデータパス

- Branch if Less than or Equal Zero
- Branch if Grater Than Zero
- Branch if Less Than Zero
- Branch if Grater than or Equel Zero

等のゼロとの比較に関する疑似命令をサポートしている．

3.2.6　I/O インターフェース

図 3.8 に I/O インターフェースを示す．設計したメニーコアプロセッサはホストコンピュータと通信を行うための I/O インターフェースを備える．I/O インターフェースは 1 組のシリアル入出力であり，これはハードウェアに依存した設計を極力控えることでボード間移植性を高めることを設計目標としている．

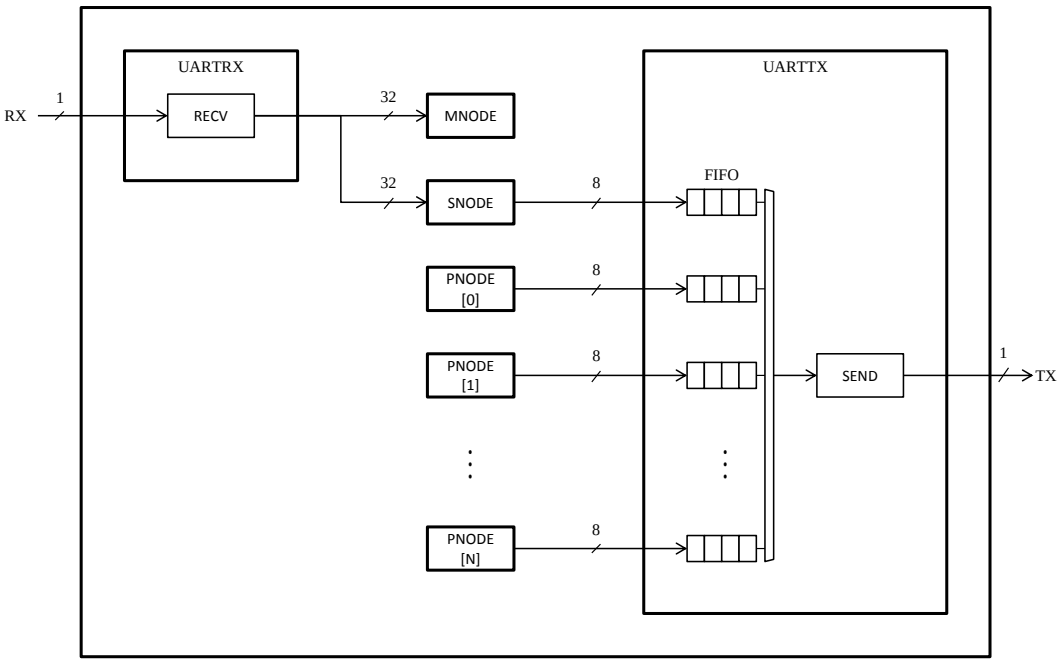


図 3.8 I/O インターフェース

受信 (RX) ポートはホストコンピュータから転送されるバイナリを RAM に格納するのに用いられる．この格納処理は専用のモジュールであるプログラムローダが行う．プログラムローダはメモリノードに直結されており，転送されたバイナリはメモリノードとスケジューラノードの RAM に順次格納される．

送信 (TX) ポートは各プロセッサノードやスケジューラノードのテキスト出力をホストコンピュータに転送するのに用いられる．各プロセッサノードやスケジューラノードは，メモリマップド IO のアドレス 0x0 によってテキストを出力するので，メニーコアプロセッサはこれらの出力を一括管理するモジュールを持つ．このモジュールはノードが結果を出力すると，各ノードのための FIFO に一時格納し，ホストコンピュータへの転送が可能になり次第，格納したデータを順次転送する．複数のノードが結果を出力を行うプログラムを実行した場合，各ノードの出力結果のみを連続して転送すると，端末エミュレータに表示される出力は各ノードの出力が混在したものになってしまう．このような場合，出力がどのノードによるものなのかを判別するため，出力を管理するモジュールは出力を行ったノードを区別するための識別子を転送したあと，そのノードの出力を転送する．

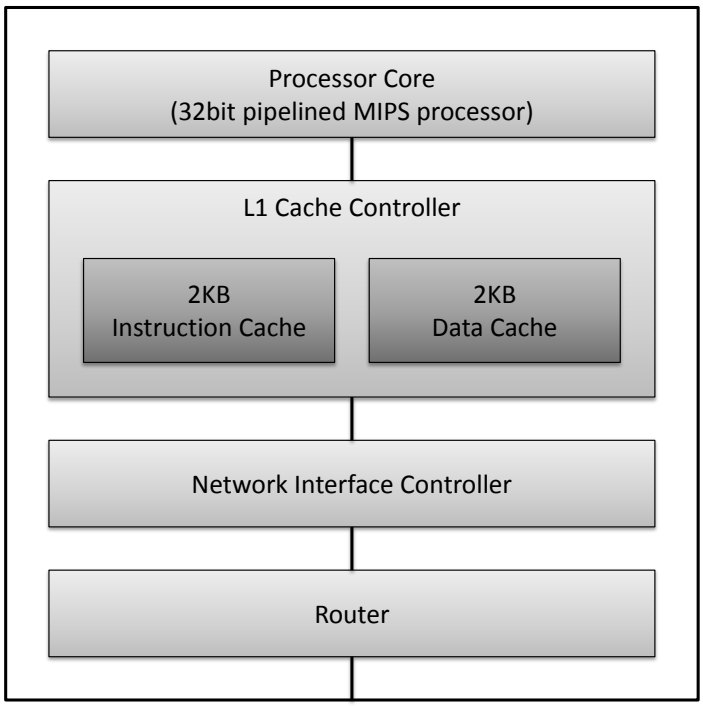


図 3.9　プロセッサノードの内部構成

3.3　プロセッサノードアーキテクチャ

プロセッサノードはプログラムの実行を行うノードである．**図** 3.9 にプロセッサノードの内部構成を表すブロック図を示す．プロセッサノードはプロセッサコア，1 次キャッシュコントローラ，NIC(Network Interface Controller)，ルータで構成される．プロセッサノードは前節にて説明したプロセッサコアを 1 基内蔵している．

キャッシュコントローラはプロセッサに接続されるモジュールであり，内部に独立した 2KB の命令キャッシュとデータキャッシュを持つ．キャッシュはダイレクトマップ方式であり，8word のキャッシュラインを 64 エントリまで記憶できる．キャッシュの制御は有効ビットとダーティビット，タグによって行われる．

プロセッサコアが命令メモリやデータメモリを参照するとき，キャッシュコントローラはキャッシュのヒット/ミスを判定する．いずれの参照要求もヒットしていれば，プロセッサは参照対象を意識することなくキャッシュの内容を読み書きできる．しかし，いず

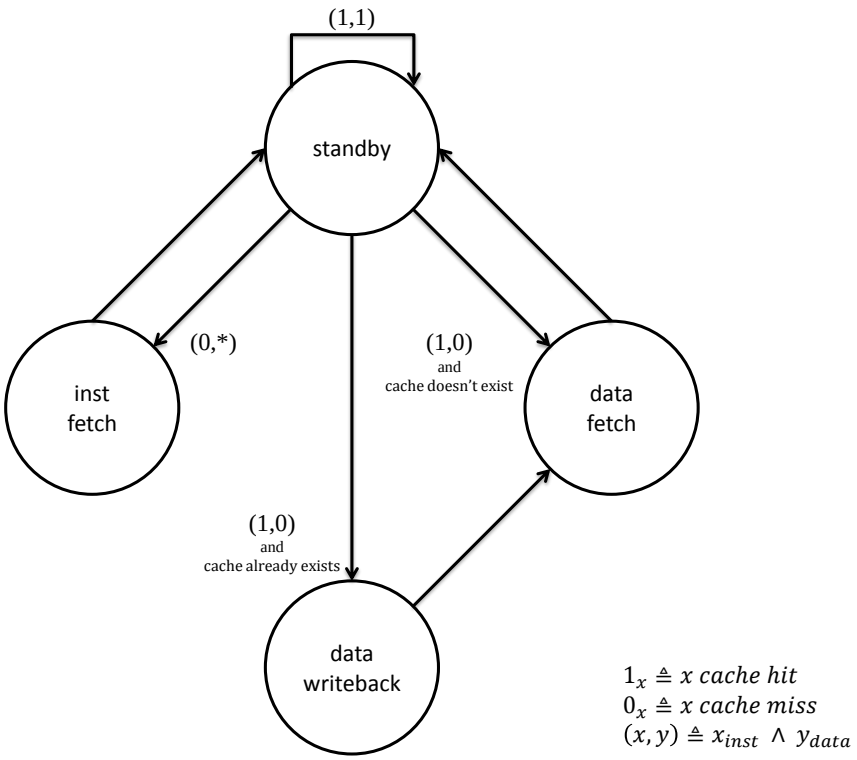
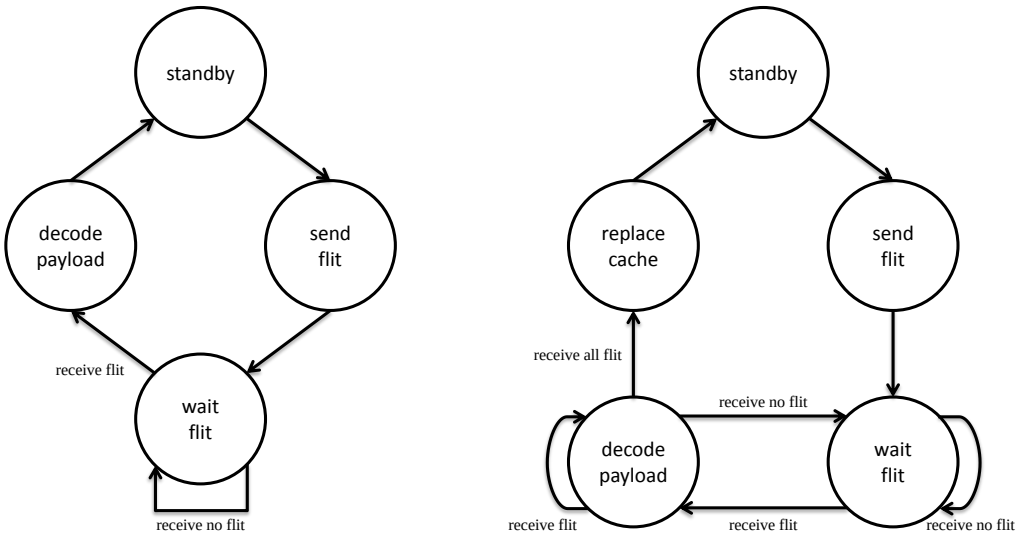


図 3.10　1 次キャッシュコントローラの状態遷移図

れかの参照要求がミスを起こした場合はプロセッサに対して即座にストール信号を入力し，動作を一時的に停止させる．またプロセッサが動作を停止している間，キャッシュコントローラは NIC にキャッシュラインの転送要求を指示し，順次キャッシュラインの入れ替えを行う．全ての参照要求がヒットする状態になるとプロセッサに対するストール信号を解除し，動作を継続する．

図 3.10 に 1 次キャッシュコントローラの状態遷移図を示す．キャッシュラインの入れ替えは命令キャッシュを優先して順次行われる．これが状態遷移図における inst fetch 状態に該当する．通常，プロセッサは命令メモリに対する書き込みは行わないので，命令キャッシュの内容はフェッチの際に上書きされるが，データキャッシュの内容はプロセッサによって更新されるので，上書き前にライトバックが必要なことがある．特に所定のエントリに有効かつダーティで，かつタグの異なるキャッシュラインが既に存在していた場合，一度そのキャッシュラインをメインメモリへライトバックし，その後改めてフェッチによる上書きを行う．これが状態遷移図における data writeback 状態，data fetch 状態に該当する．



(a) writeback request　　(b) fetch request

図 3.11　プロセッサノードにおける NIC の状態遷移図

NIC はキャッシュライン転送要求とパケットの相互変換を行うモジュールであり，全てのノードタイプの構成に含まれる．NIC は S/A-format の cmd ごとにキャッシュラインの一時保存やパケットの生成といった一連の動作を定義されており，他のモジュールから引数のように cmd を受け取ることで所定の動作を行う．プロセッサノードでは複数存在する cmd のうち，スケジューリングの動作指示に対する応答，キャッシュラインのフェッチ要求，キャッシュラインのライトバック要求の 3 コマンドをキャッシュコントローラから指示される．

図 3.11 にプロセッサノードにおける NIC の状態遷移図を示す．**図** 3.11(a) はキャッシュラインのライトバック要求コマンドにおける状態遷移図である．キャッシュコントローラからライトバックしたいキャッシュラインを NIC のバッファに格納し，**図** 3.6(b) で示したデータ形式パケットの各フリットを生成する．このパケットの送信先はペアのキャッシュノードであり，生成されたフリットは fifo を通じてルータに入力される．これが状態遷移図における send flit 状態に該当する．キャッシュノードはライトバックが完了するとライトバック完了通知を示す制御形式のパケットを返送するので，NIC は A-format のフリッ

トを受信するまで待機し，受信し次第スタンバイ状態に移行する．これが状態遷移図における wait flit 状態，decode payload 状態に該当する．

図 3.11(b) はキャッシュラインのフェッチ要求コマンドにおける状態遷移図である．ペアのキャッシュノードに対して，フェッチしたいキャッシュラインのアドレスとともに**図** 3.6(a) で示した制御形式パケットの各フリットを生成し，送信する．これが状態遷移図における send flit 状態に該当する．その後キャッシュノードからキャッシュラインを含むデータ形式のパケットが返送されるので，NIC は D-format のフリットを受信し次第，バッファにペイロードであるキャッシュラインを格納していく．これが状態遷移図における wait flit 状態，decode payload 状態に該当する．全てのフリット（キャッシュライン）を受信すると，バッファの内容をまとめてキャッシュコントローラに送信する．この時にキャッシュコントローラ内でキャッシュラインの上書きが行われ，完了し次第スタンバイ状態に移行する．これが状態遷移図における replace cache 状態に該当する．

スケジューラノードはプロセッサノードに対して application ID を発行し，制御形式のパケットを送出する．プロセッサノードはこのパケットに含まれる S-format のフリットから抽出した application ID を自身に割り当てると，プロセッサコアが実行するべきアプリケーションや利用するメモリ領域が設定され，動作が開始される．また，スケジューラノードは任意の値をプロセッサノードに与えることができ，プロセッサノードはフリット受信時，この値を自身のステータスレジスタに格納する．ステータスレジスタの値は**表** 3.1 で示したアドレスによって参照が可能である．

プロセッサコアがアプリケーションの実行を終了すると，キャッシュコントローラがプロセッサやキャッシュの動作を停止させ，制御形式のパケットを用いて動作完了通知を行うよう NIC に指示する．この際，プロセッサノードは任意の値をスケジューラに与えることができる．制御形式のパケットには，この値とプロセッサが設定動作開始時にスケジューラから発行された application ID を返送する．その後スケジューラノードからの動作指示を待ち，application ID の割り当てから始まる一連の動作を繰り返す．

このスケジューリングに関する動作において，NIC の状態遷移図は**図** 3.11(a) のライトバック要求の状態遷移図とほぼ変わらない．ただし初期状態ではプロセッサコアはスケジューラノードからの動作指示を受けるまで動作を開始することはできないことから，キャッシュコントローラおよび NIC は wait flit 状態で開始されるという点で異なる．

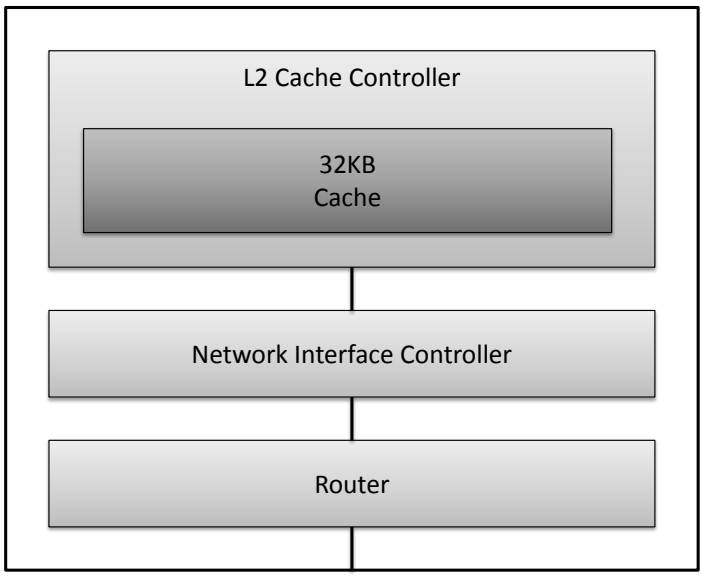


図 3.12　キャッシュノードの内部構成

3.4　キャッシュノードアーキテクチャ

キャッシュノードはプロセッサノードとペアを組み，プロセッサノードとメモリノードの間で 2 次キャッシュとしての役割を果たす．プロセッサノードはペアとなるキャッシュノードの physical ID を自ノードの physical ID から算出するが，キャッシュノードはキャッシュの転送要求を行ったプロセッサノードを自身のペアとして認識する．よって，ペアのプロセッサノードに割り当てられた application ID をキャッシュノードが直接保持する必要はなく，キャッシュノードはメニーコアプロセッサの動作開始時点から，スケジューリングに関係なく動作し続ける．また，キャッシュノード内でキャッシュミスが起こった場合，キャッシュノードはメモリノードに対して二次的にキャッシュの転送要求を行う．

図 3.12 にキャッシュノードの内部構成を表すブロック図を示す．キャッシュノードは 2 次キャッシュコントローラ，NIC，ルータで構成される．

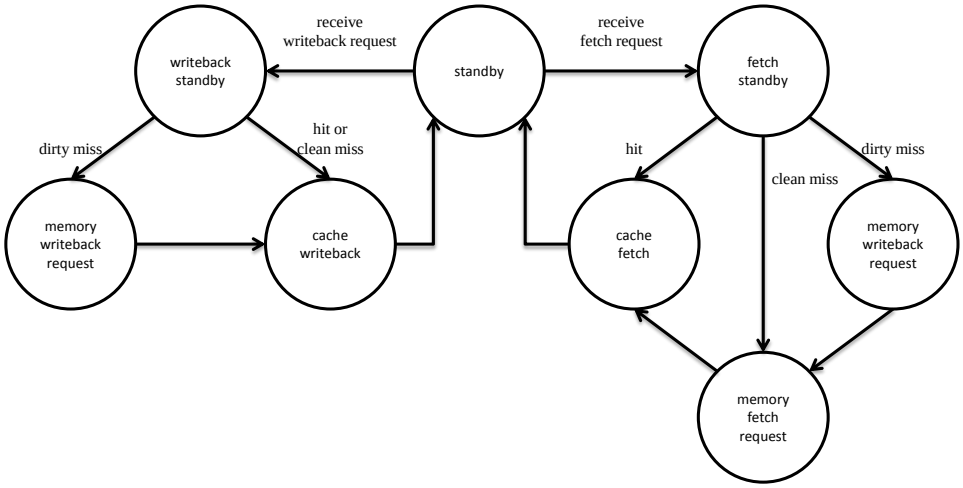


図 3.13 2 次キャッシュコントローラの状態遷移図

キャッシュコントローラは内部に 32KB のキャッシュを持つ．キャッシュの構造はプロセッサノードの 1 次キャッシュと同様であり，こちらは 8word のキャッシュラインを 1024 エントリまで記憶できる．

図 3.13 に 2 次キャッシュコントローラの状態遷移図を示す．dirty miss，clean miss はミスしたとき，当該エントリに既に別のキャッシュラインが格納されているか否かを示す．2 次キャッシュコントローラは 1 次キャッシュコントローラと異なり，命令とデータの区別は行わない．プロセッサノードからライトバック要求またはフェッチ要求を受けたとき，ヒットした場合はキャッシュ内で処理を行うことができるが，ミスした場合はメモリへの二次的なライトバックまたはフェッチを行ったのち，キャッシュ内で本来行うべき処理を行う．メニーコアプロセッサにおいて，1 次キャッシュは 2 次キャッシュの，2 次キャッシュは主記憶のサブセットであることが保証される．特にライトバック時，同一のエントリに存在する異なるキャッシュラインをライトバックするケースは実際に存在しないが，今後，複数のプロセッサノードが同一のキャッシュを参照するような拡張を行うことを考慮し，状態を定義している．

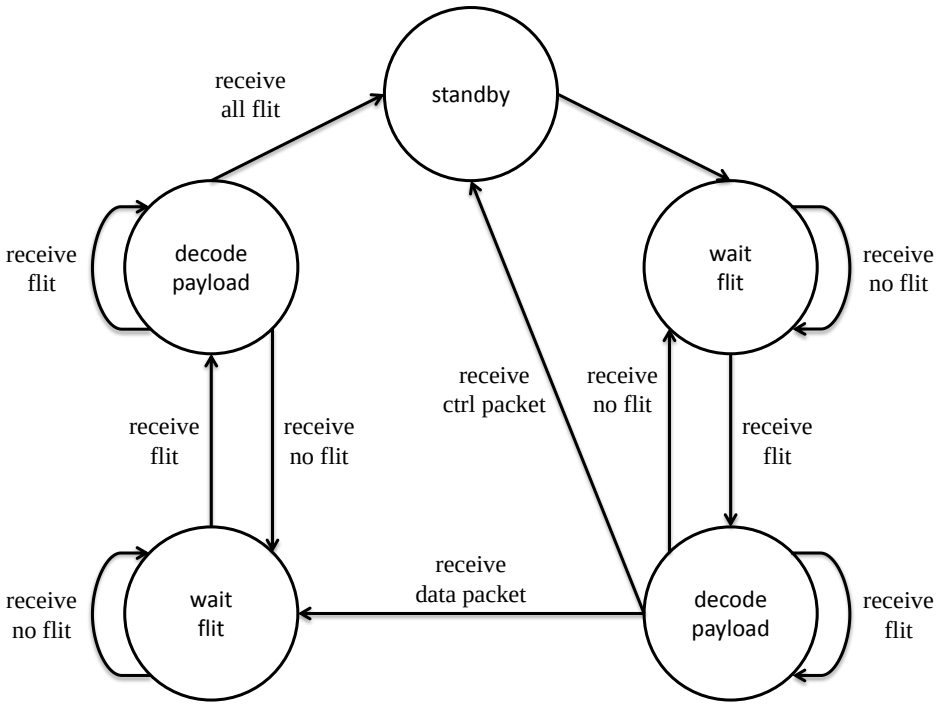


図 3.14　キャッシュノードにおける NIC の受信コマンドの状態遷移図

NIC はプロセッサノードに含まれるものと同等である．キャッシュノードでは複数存在する cmd のうち，

- プロセッサノードからの要求をデコードする受信コマンド
- プロセッサノードからのフェッチ要求に対する応答コマンド
- プロセッサノードからのライトバック要求に対する応答コマンド
- メモリノードへのフェッチ要求コマンド
- メモリノードへのライトバック要求コマンド

を用いる．メモリノードへのフェッチ要求コマンド，ライトバック要求コマンドの状態遷移図は**図** 3.11 で示したものに準ずる．

図 3.14 にキャッシュノードにおける NIC の受信コマンドの状態遷移図を示す．受信コマンドは受信待機し，フリットを受信し次第ペイロードのデコードを行うが，A-format のフリットを受信するまではパケット全体のフリット数が予測できない．そのため，A-format のフリットを受信した時点で，改めて受信処理を開始するか，受信を継続するか

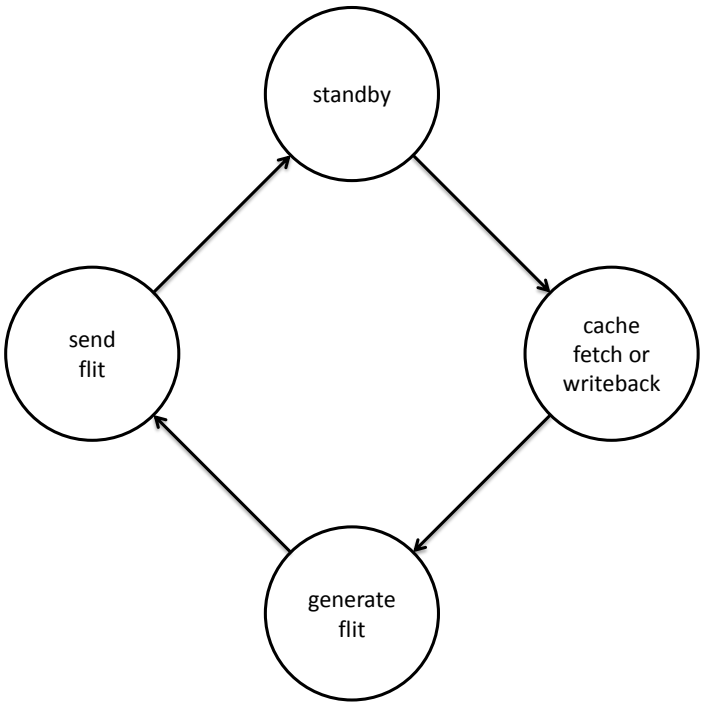


図 3.15 キャッシュノードにおける NIC の応答コマンドの状態遷移図

を判断する．これが状態遷移図における右下の decode payload 状態に該当する．一連の受信処理が終了すると，キャッシュ処理に必要なアドレスやキャッシュラインはローカルバッファに格納され，キャッシュコントローラは**図** 3.13 における writeback standby 状態または fetch standby 状態に移行する．この後初めてキャッシュのヒット判定とミス判定が行われ，必要に応じて二次的なキャッシュ処理が行われる．

図 3.15 にキャッシュノードにおける NIC の応答コマンドの状態遷移図を示す．これは**図** 3.13 における cache writeback 状態または cache fetch 状態に該当する．受信コマンドによって応答に必要な情報は予めバッファに格納されているので，状態遷移は単純で，キャッシュの処理を行ったのち，処理完了を示すパケットを生成してプロセッサノードに返送する．

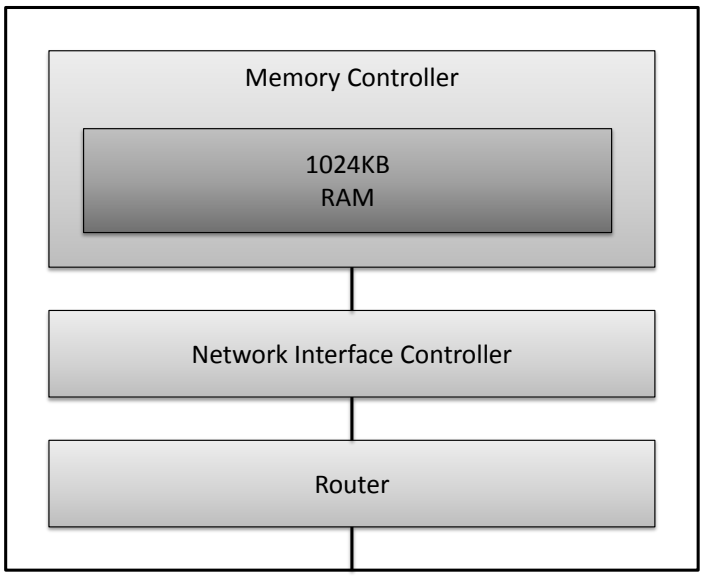


図 3.16　メモリノードの内部構成

3.5　メモリノードアーキテクチャ

図 3.16 にメモリノードの内部構成を表すブロック図を示す．メモリノードはメモリキャッシュコントローラ，NIC，ルータで構成される．

メモリコントローラは内部に 1024KB の RAM を持つ．RAM は 64KB の 16 領域に分割され，各領域にプログラムが格納される．一般的な FPGA ボードは RAM リソースとして，On-Chip RAM に相当する BRAM と Off-Chip DRAM を有する．BRAM のリソースは FPGA のモデルに依存し，容量は Off-Chip DRAM に比べると小さい．Off-Chip DRAM のリソースは外部 RAM を利用するために大きいが，DRAM のインターフェースや仕様はボードによって異なる．BRAM だけでなく Off-Chip DRAM も併用した設計とすることで，BRAM のリソースに縛られることなく利用可能なメモリ領域は増加するが，DRAM 制御のためのボードに依存した実装が要求される．本論文における実装ではメンテナンス性を重視し，ボード間移植性の低下を防ぐために BRAM のみを用いた．

NIC はプロセッサノードに含まれるものと同等である．メモリノードでは複数存在する cmd のうち，キャッシュノードからの要求をデコードする受信コマンド，キャッシュノードの要求に応答するフェッチ応答，ライトバック応答コマンドを用いる．受信コマンドの状態遷移図は**図** 3.14 で，フェッチ応答，ライトバック応答コマンドの状態遷移図は**図** 3.15 で示したものに準ずる．

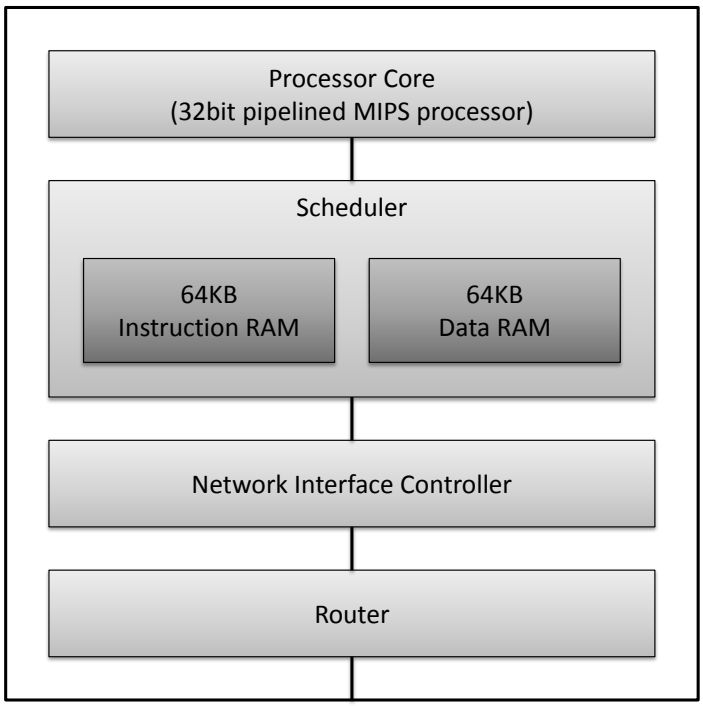


図 3.17　スケジューラノードの内部構成

3.6　スケジューラノードアーキテクチャ

図 3.17 にスケジューラノードの内部構成を表すブロック図を示す．プロセッサノードはプロセッサコア，スケジューラ，NIC，ルータで構成される．プロセッサコアはプロセッサノードに内蔵されるものと同等のものを 1 基内蔵している．スケジューラは内部に 64KB の命令メモリとデータメモリを持つ．スケジューリングのための，スケジューラノードのプロセッサコアが実行するプログラムは，動作開始時にメモリノードではなくスケジューラノードの RAM に直接転送される．

スケジューラは内部にステータスレジスタを持つ．ステータスレジスタは各プロセッサノードの稼働状況を保持・表現するレジスタであり，各プロセッサノードにつき 1bit のプロセッサノード長のレジスタである．即ち，実行するプログラムが格納されたメモリ領域が割り当てられ，実行中のプロセッサノードがあるとき，そのプロセッサノードに対応するステータスレジスタのビットが立つ．

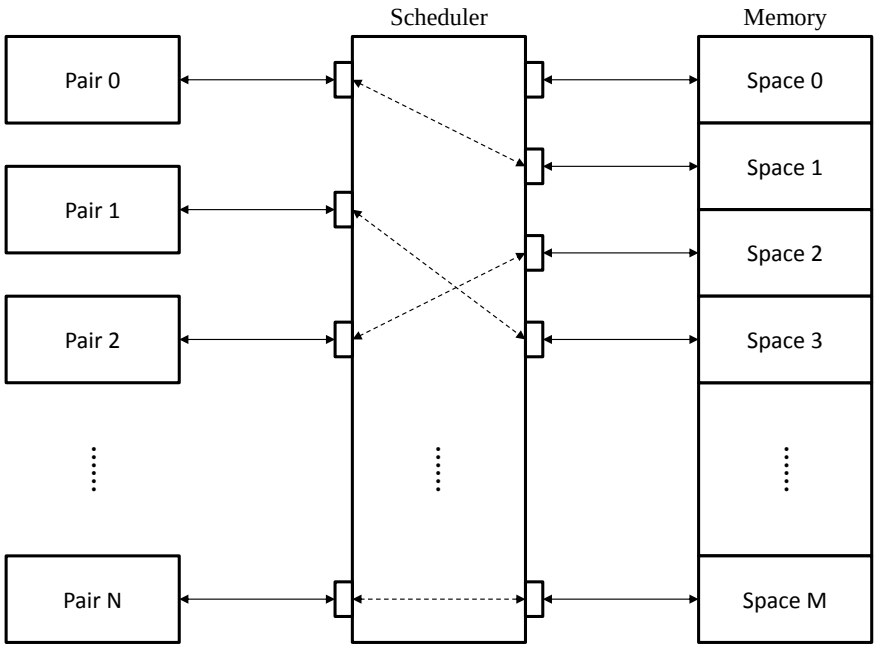


図 3.18 スケジューラノードによる割り当ての例

ステータスレジスタの更新はプロセッサノードからの実行完了通知を受け取ったときと，スケジューラがプロセッサノードに対して実行を指示した時に行われる．スケジューラはアプリケーションによるスケジューリングを行うが，動作指示対象のプロセッサノードを指すステータスレジスタのビットが伏せられていることを確認してから動作指示を行い，かつ動作中のプロセッサノードに対して処理を行わないことを前提としているので，これらの更新は独立に行うことができる．よって，スケジューラはプロセッサノードから動作完了を示すパケットを受信と，必要に応じたプロセッサノード（とキャッシュノードのペア）へのメモリ領域の割り当てを並列的に行う．

図 3.18 にスケジューラノードによるメモリ領域割り当ての例を示す．この例では，ペアが N 組，プログラムが格納されたメモリ領域が M 領域存在している．スケジューラノードはペア 0 に領域 1 を，ペア 1 に領域 3 を，ペア 2 に領域 2 を，ペア N に領域 M を割り当てている．この状態からペア 0 がプログラムの実行を終了したと仮定すると，動作完了通知によりステータスレジスタが更新される．ステータスレジスタの更新をスケジューラが検知すると，スケジューラはペア 0 に対して領域 0 を新たに割り当てることができる．

```
1  int main(void){
2    int pair, space=0;
3
4    for(pair=0; pair < N+1; pair++){
5      activate(pair, space);
6      space += 1;
7    }
8
9    while(space < M+1){
10     if(get_stat()){
11       pair = find_deactive();
12       activate(pair, space);
13       space += 1;
14     }
15   }
16
17   return 0;
18 }
```

図 3.19　割り当てを行う C プログラムの例

図 3.19 にスケジューリングのための C プログラムの例を示す．ただし，ペア数 N とプログラムが格納された領域数 $M(\leq 16)$ は $N < M$ の関係にあるとする．4 行目から 7 行目は初期化シーケンスであり，各ペアに対してメモリ領域を順番に割り当てている．9 行目から 15 行目は追加シーケンスであり，動作を終了したペアに対してメモリ領域を順番に割り当てている．

メモリ領域は連続しているので，1 領域内を表現可能な論理アドレスを物理アドレスに変換するためには，領域を区別するためのタグを論理アドレスの上位に付加すればよい．プログラムを 1 回だけ実行するのであれば，スケジューラノードによるメモリ領域の割り当てはタグの送信だけで済む．しかしプログラムを複数回にわたって実行する必要があるとき，一つの領域を複数回割り当てることになる．キャッシュノードはアドレスによってのみメモリの内容を区別しているので，アドレスから領域を区別することはできても，プログラム実行のイテレーション回数の区別ができない．なので，ある領域を複数回にわたって特定のペアに割り当てると，直前のイテレーションによるキャッシュが実行中のイテレーションによるキャッシュとして誤認される可能性がある．これを防ぐため，タグにはメモリ領域だけでなく，イテレーションを区別するための情報も含まれる必要がある．

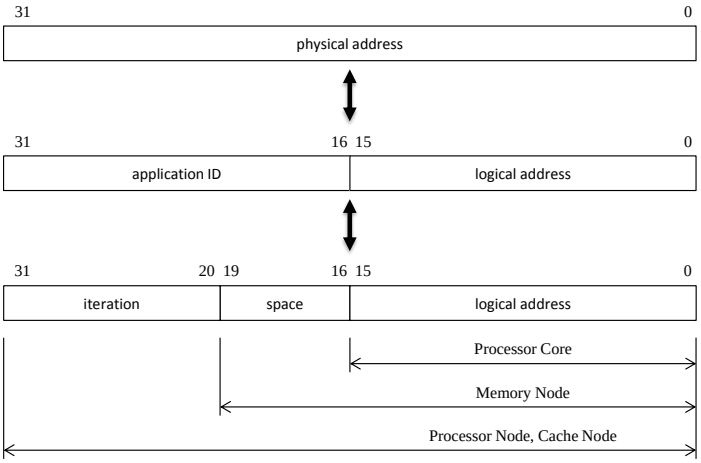


図 3.20 アドレスの構成

図 3.20 にアドレス構成を示す．1 領域は 64KB なので，プロセッサコアがこれを表現するための論理アドレスは 16bit である．メモリ領域は全体で 16 領域存在するので，領域を区別するための 4bit のタグを加えると 20bit となる．メモリノードにおいて，これ以外の部分である上位 12bit は破棄される．一方，キャッシュノードはメモリアドレス全体をキャッシュの制御に用いている．このような特性から，アドレスの上位 12bit をイテレーション回数を示すタグとすると，プロセッサコアやメモリノードの動作に影響を及ぼさずに，キャッシュノードがイテレーションを区別することが可能となる．スケジューラはペアがメモリ領域とイテレーションを区別するための情報として，この 16bit タグを application ID としてプロセッサに送信すればよい．

スケジューラノードはステータスレジスタの他，1 本の加算レジスタを持つ．加算レジスタは，設計したメニーコアプロセッサで自由なノード間のデータの送受信を可能にする拡張に向けて，実験的に導入したレジスタである．加算レジスタはプロセッサノードから受け取ったパケットのうち，制御形式のフリットに含まれる値を加算し続ける．加算レジスタの内容は**表** 3.1 で示したアドレスによって参照が可能である．

第 4 章

実用的なメニーコアプロセッサの実装

4.1 実装環境

設計したメニーコアプロセッサの論理回路は Verilog HDL で記述し，Xilinx ISE 14.5 を用いて論理合成を行う．また，実装には Virtex-7 XC7VX485T を搭載する VC707 Evaluation Board を用いる．設計したメニーコアプロセッサは Xilinx 製の FPGA チップを搭載し，USB による 1 組の双方向シリアル通信が可能な FPGA ボード上で動作するが，本論文では実際に多くのコアを動作させるために，単一の大規模 FPGA をターゲットとして実装を行う．Xilinx Virtex-7 XC7VX485T は 75,900 個のスライスを持つ大規模な FPGA である．

4.2 モジュールレベルアーキテクチャ

図 4.1 に実装するメニーコアプロセッサのノード構成を示す．ノードアレイのサイズは 6×6 であり，ノード構成はプロセッサノードとキャッシュノードのペアが 16 組，スケジューラノードとメモリノードが各 1 ノードで構成される．**図** 3.2 で示した構成ではノードアレイに 17 組のペアが含まれているが，本設計におけるメモリ領域は 16 領域に分割されており，同時に実行可能なプロセッサ数が 16 に制限される．よって，**図** 3.2 で示した構成の場合，プログラムを実行しないプロセッサノードが少なくとも 1 ノード存在する．このような余分なノードによるハードウェア量の増加を防ぐため，ペアの数を 17 から 16 に削減し，代わりにルータのみで構成されるフェイクノードへの置き換えを行う．

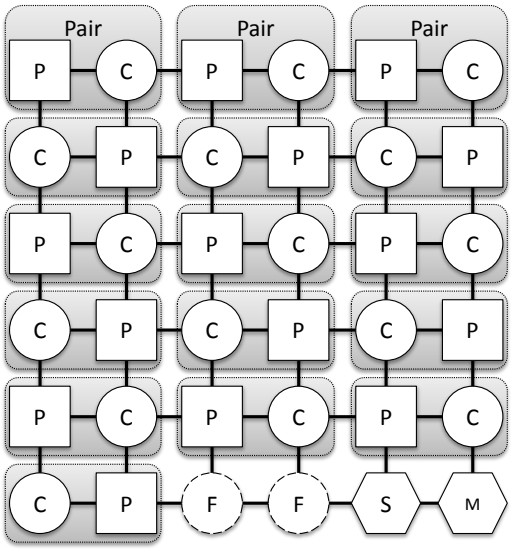


図 4.1 実装するメニーコアプロセッサのノード構成

図 4.2 に記述したコードのモジュール関係とコード行数を示す．矢印は矢印元のモジュールが矢印先のモジュールをインスタンスとして含むことを示す．また，基本的に下位のモジュールはのインスタンスは上位のモジュールにつき 1 個生成されるが，モジュール付近に数の表示があるモジュールについては，複数個のインスタンスが生成されることを示す．例えば，MCORE は PNODE と CNODE のインスタンスを 16 個と，FNODE のインスタンスを 2 個持つ．

GEN はメニーコアプロセッサ本体に入力するクロック信号とリセット信号を生成するモジュールである．MCORE はメニーコアプロセッサ本体とシリアル入出力を行う部分から成る．MCORE はノードのインスタンスを複数生成し，ノードのネットワーク信号ピンを 2 次元メッシュ状に接続してノードアレイを構成する．また，テキスト出力やプログラム転送のための信号線をノードに直接配線する．各ノードタイプは共通して持つモジュールとユニークなモジュールを有している．CORE，NIC，ROUTER では複数のノードタイプによってそれらのインスタンスが生成されるが，MEM1，SCHD，MEM2，RAM のインスタンスはそれぞれ一つのノードタイプによってのみ生成される．

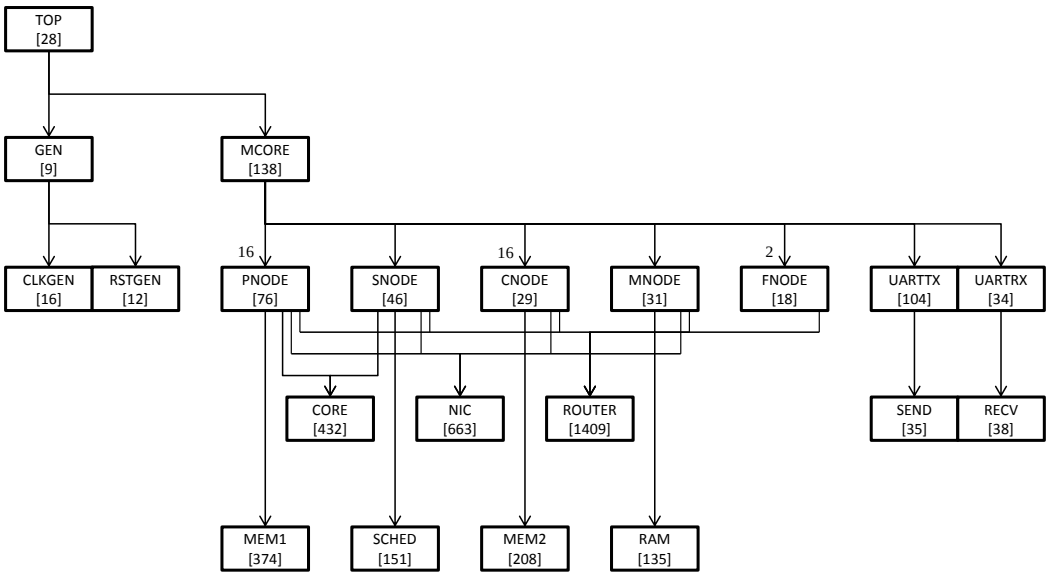


図 4.2　記述したコードのモジュール関係とコード行数

4.3　動作検証

アプリケーションのコンパイルやバイナリの生成は Ubuntu 13.04 上で行う．アプリケーションは C 言語を用いて数種類記述し，これを GCC ベースの LINUX 用 MIPS クロスコンパイラを用いて実行可能ファイルを生成したのち，バイナリ形式に変換する．このバイナリをメモリイメージとして設計したメニーコアプロセッサのメモリに格納して実行する．設計したメニーコアプロセッサはの動作検証は，同一のバイナリを用いてソフトウェアシミュレーションの実行結果と FPGA 実装したメニーコアプロセッサの実行結果を照合して行う．ソフトウェアシミュレータは Synopsys VCS MX H-2013.06 を用いる．プロセッサ用には演算やメモリ上のスタックの利用，文字の出力などを行うアプリケーションを，スケジューラ用には逐次割り当てを行うスケジューリングアルゴリズムを用いたアプリケーションを用意し，これらを用いて動作検証を行った結果，ソフトウェアシミュレーションと FPGA の動作結果が同一であることが確認できた．

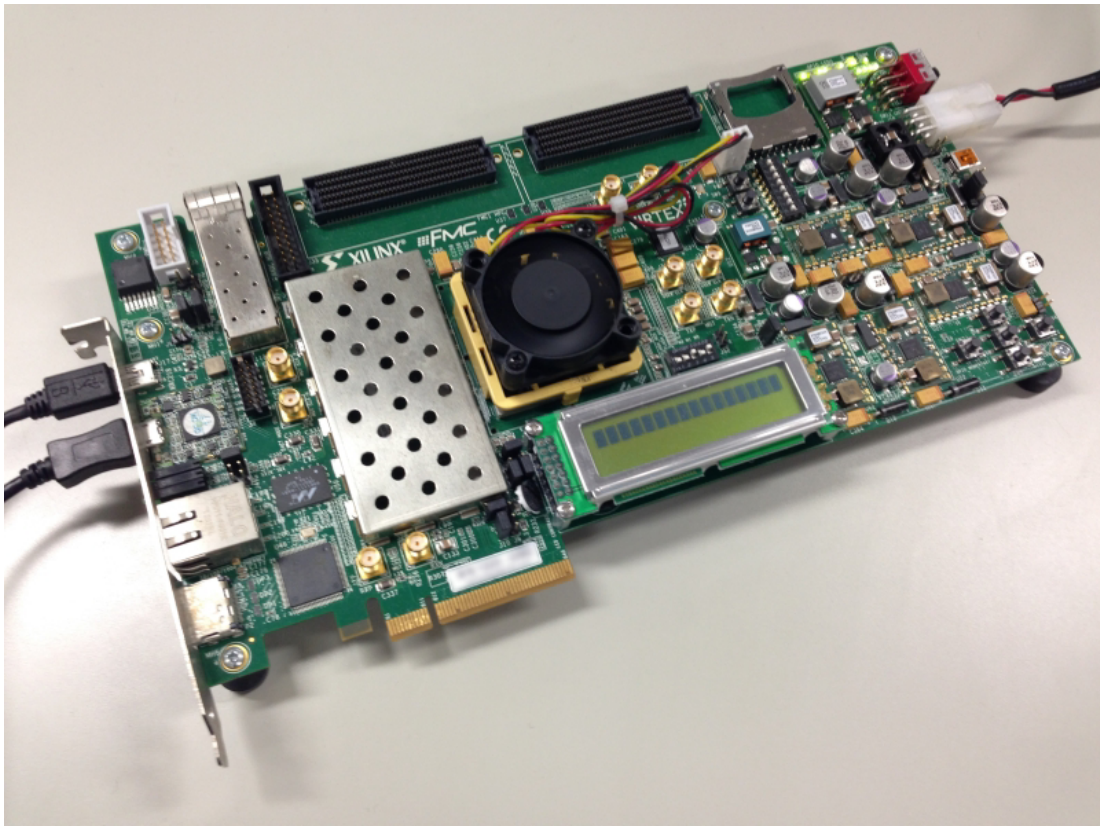


図 4.3　動作中の VC707 の様子

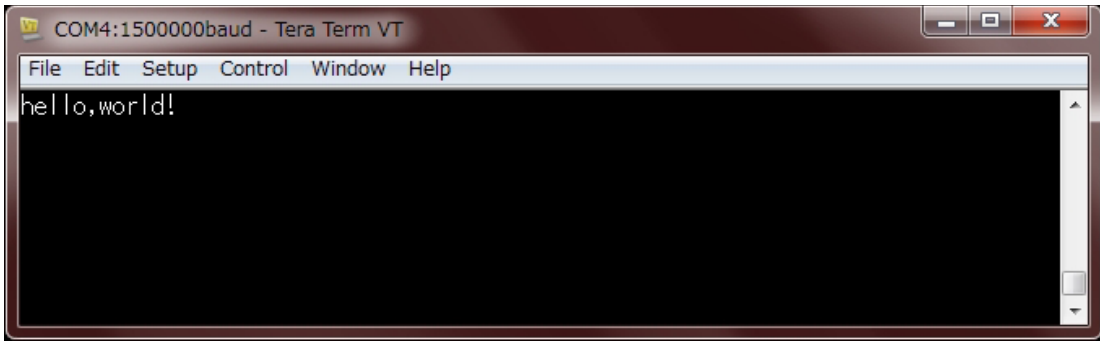


図 4.4　動作中の端末エミュレータの様子

図 4.3 に動作中の VC707 の様子を，**図** 4.4 に動作中の端末エミュレータの様子を示す．端末エミュレータには VC707 に実装されたメニーコアプロセッサによる文字出力のアプリケーションの実行結果が表示されている．

第 5 章

評価

5.1 ハードウェアリソース

表 5.1 にハードウェアリソース使用量を示す．used は実装において消費したハードウェア量，available は Virtex-7 XC7VX485T において利用可能なハードウェア量，utilization は available に対する used の割合を示している．また，論理合成に要した時間は 6,163 秒であった．

ISE による論理合成は最適化により冗長な回路の削減が行われるため，実際に消費するハードウェア量は記述したコードから推定されるハードウェア量よりも少ないものとなる．例えば，ノードアレイにおける外縁部のノードのルータにはノードアレイ外の方向にも通信ポートが用意されているが，このポートは実際に利用しないため，この方向の通信のための回路は最適化により削減される．このように，同じコードをもとにノードのインスタンスを複数生成したとしても，ノードアレイ上の位置やパケットフローによってノード毎のハードウェア量にばらつきが生じてしまうため，構成を変更した場合のメニーコアプロセッサ全体のハードウェア量は単純に推測することができない．

表 5.1　ハードウェアリソース使用量

	used	available	utilization
Slice	52,970	75,900	69%
Slice Regs	74,748	607,200	12%
Slice LUTs	167,760	303,600	55%
BRAM	483	3,090	15%

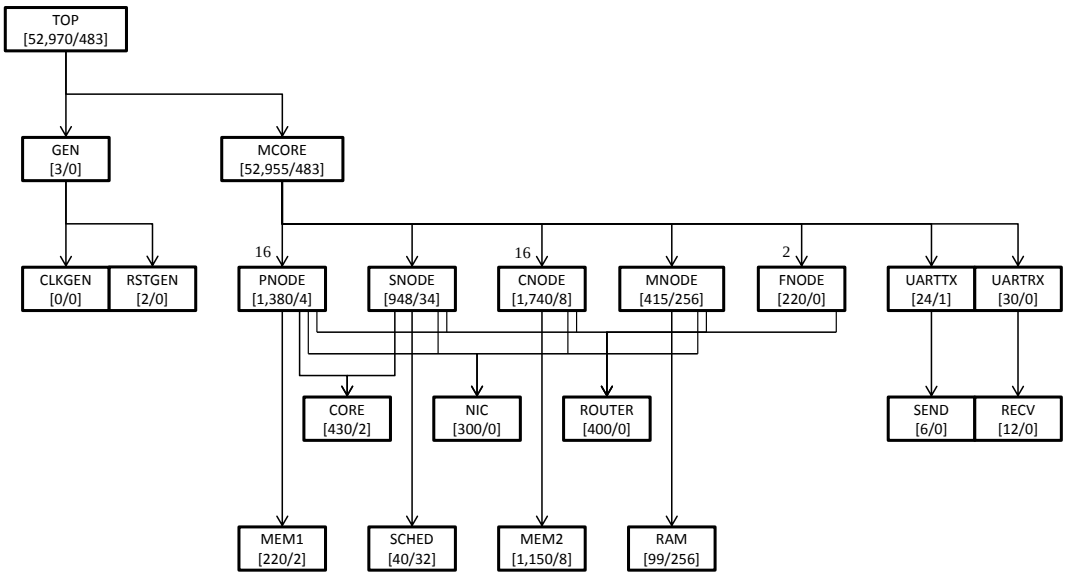


図 5.1　モジュール毎のスライス数と BRAM の使用量 [スライス数/BRAM]

このことに留意し，**図** 5.1 にモジュール毎のスライス数と BRAM の使用量を示す．インスタンスが複数生成されるモジュールのスライス数に関しては，平均から端数を除いた数を示している．一方，BRAM はキャッシュや RAM の実体として用いられるので，最適化による回路の削減は行われず，インスタンス毎のハードウェア量のばらつきは生じない．

表 5.1 を用いて残りのハードウェア量を算出し，モジュール毎の平均スライス数で除算を行うと，現在のノード構成に加えてプロセッサノードとキャッシュノードのペアを 7 組追加でき，全体で 34 コアの構成で実装が可能である．また，プライベートキャッシュを廃止し，キャッシュノードが複数のプロセッサノードのキャッシュを処理できるよう拡張した場合，全体で 52 コアの構成で実装が可能である．

5.2 アーキテクチャ性能

実装したメニーコアプロセッサがアクセラレータとして利用されることを想定したアーキテクチャ性能を評価する．アーキテクチャ性能の評価は N-Queens 問題を解くアプリケーションを用いて行う．

図 5.2 にスケジューラ用のプログラムの一部を示す．3 行目は n の値から部分問題の数を算出している．この部分問題を示す値をプロセッサに与えることで，プロセッサコアの並列処理を可能としている．7 行目から 9 行目はメモリ領域ごとのアプリケーション ID を初期化している．12 行目から 20 行目は逐次割り当てシーケンスであり，ステータスレジスタの値を取得し，いずれかのコアが実行状態になければそのコアを特定し，部分問題の値とアプリケーション ID を割り当てている．割り当て後は割り当て済みの部分問題とメモリ領域のアプリケーション ID を更新している．22 行目から 29 行目は出力シーケンスであり，ステータスレジスタの値を取得し，全ての部分問題をプロセッサコアが処理し終えたことが確認できた場合，加算レジスタの値を出力する．

同様に，**図** 5.3 にプロセッサ用のプログラムの一部を示す．3 行目はスケジューラ同様，部分問題の数を算出している．6 行目はステータスレジスタの値を取得し，スケジューラから割り当てられた部分問題の値を取得している．7 行目は部分問題を解き，結果をレジスタに格納している．8 行目は結果をスケジューラに返送している．スケジューラノードはこのパケットを受信すると，結果を加算レジスタに加算する．

このプログラムを用いて，実装したメニーコアプロセッサが N-Queens 問題を解くのに要したサイクル数を計測した．サイクル数はプログラムの終了時，**表** 3.1 のアドレス 0x00 のリードによって得られる値を用いた．**図** 5.4 にサイクル数に基づく対 1 コア比性能を示す．プログラムを実行するコアは 1 コアから 16 コアまで 1 コア刻みで変動させ，各計測サイクル数の逆数を 1 コアの場合のサイクル数の逆数で正規化した値を性能としている．データの系列は N-Queens のサイズ N が 7,8,9,10 の場合，および線形である．線形の系列は，コア数が増加したとき，性能もコア数に比例して増加するという前提でプロットされている．

実装したメニーコアプロセッサにはノード間通信によるオーバーヘッドが存在するため，対 1 コア比性能は線形よりも少ないものとなる．特に N=7 の系列，コア数 11 以上のときにこの影響は顕著である．しかし，N-Queens のサイズ N が大きくなるにつれて，全体の実行時間に対するプロセッサ演算時間が占める割合が増加し，通信のオーバーヘッドの影響が少なくなるため，徐々に線形に近づいていく様子が観測できる．

```
1  int main(void){
2    int n    = 10;
3    int jobs = get_sub_prob_num(n);
4    int space[16];
5    int i, stat, core, sum;
6
7    for(i=0; i<16; i++){
8      space[i] = i;
9    }
10
11   i = 0;
12   while(i < jobs+1){
13     stat = get_stat();
14     if(stat!=0xffff){
15       core = find_deactive(stat);
16       activate(core, i, space[core]);
17       i++;
18       space[core] += 16;
19     }
20   }
21
22   while(1){
23     stat = get_stat();
24     if(stat==0) {
25       sum = get_sum();
26       display_dec(sum);
27       break;
28     }
29   }
30
31   return 0;
32 }
```

図 5.2　性能評価に用いるスケジューラ用の C プログラム

```c
int main(void){
  int n    = 10;
  int jobs = get_sub_prob_num(n);
  int stat, ret;

  stat = get_stat();
  ret = solver(n, stat, jobs);
  acknowledge(ret);

  return 0;
}
```

図 5.3 性能評価に用いるプロセッサ用の C プログラム

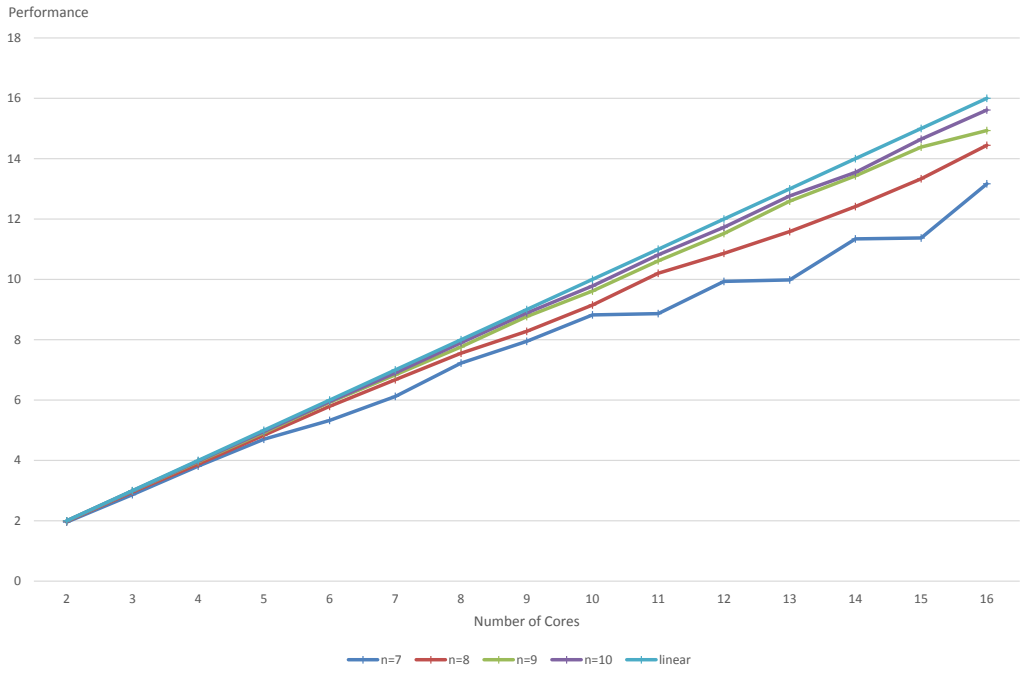


図 5.4 サイクル数に基づく対 1 コア比性能

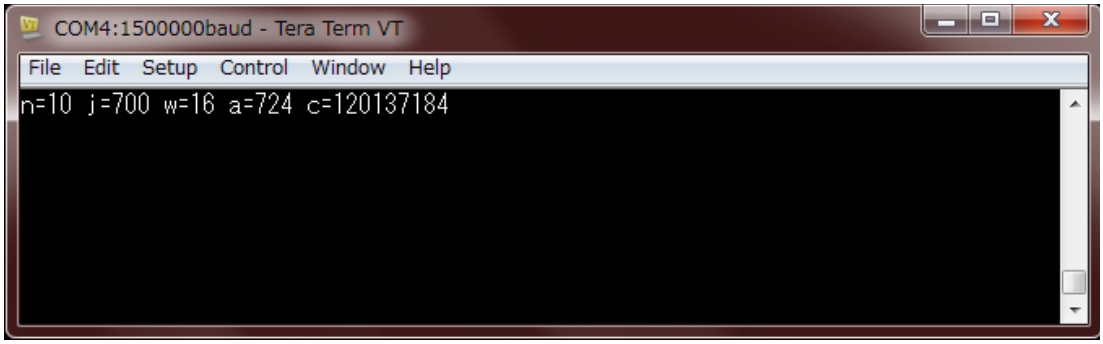


図 5.5　性能評価プログラムを実行したときの端末エミュレータの様子

5.3　シミュレーション性能

実装したメニーコアプロセッサがシミュレータとして利用されることを想定したシミュレーション性能を評価する．シミュレーション性能の評価はアーキテクチャ性能評価と同様に，N-Queens 問題を解くアプリケーションを用い，ソフトウェアシミュレータの実行時間と FPGA 実装したメニーコアプロセッサの実行時間の比較を行う．FPGA に実装されたメニーコアプロセッサの動作周波数は 24MHz であり，実行サイクル数を動作周波数の値で除算することにより，実行に要した実時間が算出される．

図 5.5 に性能評価プログラムを実行したときの端末エミュレータの様子を示す．端末エミュレータには VC707 に実装されたメニーコアプロセッサによる N-Queens のプログラムの実行結果が出力されており，N=10 のとき部分問題が 700 個存在し，16 の実行コアに部分問題の割り当てを行った結果，N-Queens の解は 724 であり，実行に要したサイクル数は 120,137,184 サイクルであったことが読み取れる．このことから，実行時間は 5.005716 秒であったことがわかる．

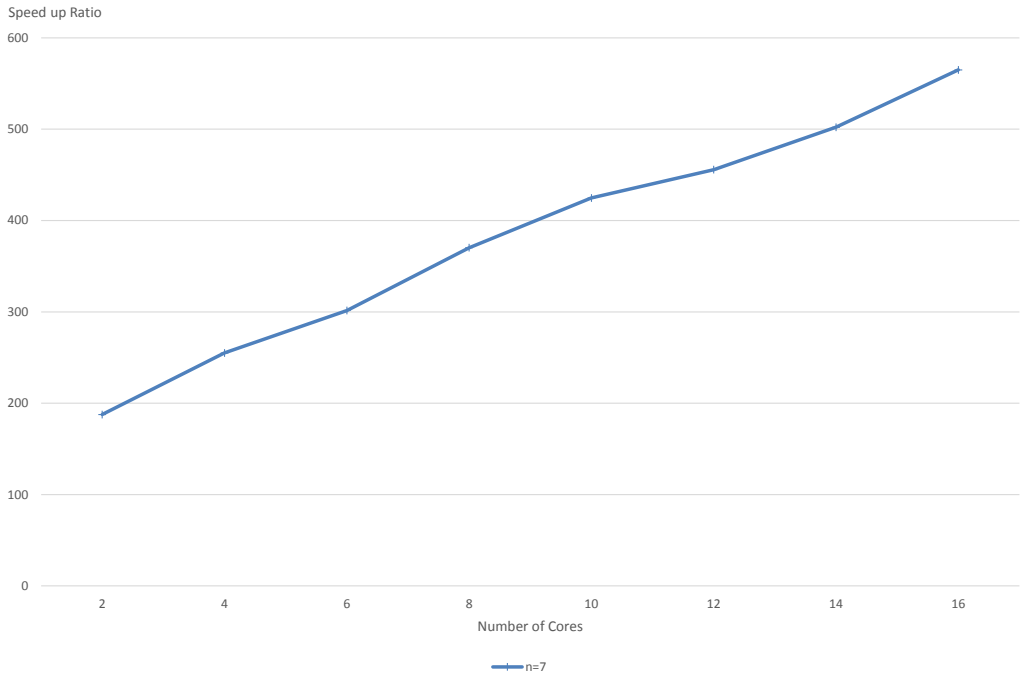


図 5.6　ソフトウェアシミュレータに対する FPGA 実装の高速化比

図 5.6 にソフトウェアシミュレータに対する FPGA 実装の高速化比を示す．プログラムを実行するコアは 2 コアから 16 コアまで 2 コア刻みで変動させ，実行時間の測定は N=7 のときのみ行った．1 コアの場合のソフトウェアシミュレーションの実行時間は約 300 秒であった．コア数が増加するといずれのシミュレーションにおいても実行に要するサイクル数は減少する．しかしソフトウェアシミュレータの場合，シミュレーション対象の回路規模の増大に起因するオーバーヘッドの増加がシミュレーション実行時間に影響を与える．実際，16 コアの場合のソフトウェアシミュレーションの実行時間は約 100 秒であり，実行時間がサイクル数に比例しない結果となった．一方，FPGA 実装はサイクル数に比例した実行時間となり，コア数を増加させればさせるほどソフトウェアシミュレータに対する高速化比が高まる結果となった．

第 6 章

結論

シンプルで実用的なメニーコアプロセッサの設計を行い，FPGA への実装を行った．ソフトウェアシミュレータと FPGA を用いて並列アプリケーションによる評価を行い，コア数の増加に対し FPGA に実装したメニーコアプロセッサの性能がほぼ線形のオーダで向上することを示した．また，実装したメニーコアプロセッサにおけるアプリケーション実行はソフトウェアシミュレータに対して最大 560 倍高速であることを示した．

今後，設計したメニーコアプロセッサをより実用的なものにするために，様々な拡張を行っていくことが要求される．例えば, 本論文における設計指針からハードウェアの RAM 資源を BRAM に限定していたが，大容量のデータを扱うアプリケーションを実行するためには Off-chip DRAM は必要不可欠な存在である．また，同様の設計指針からキャッシュノードはプロセッサノードのプライベートキャッシュとしていたが，一つの FPGA に多数のコアを搭載するためにはキャッシュノードを共有キャッシュとする必要があり，また，高度なキャッシュ制御技術を実現するためにも同期処理の導入は避けられない．こういった拡張を施しながら，より洗練された設計を目指していくことが課題である．

謝辞

本研究を進めるにあたり．適切かつ熱心な指導をしていただいた指導教員の吉瀬謙二准教授に深く感謝いたします．

ネットワークに関する知識をご教授頂くばかりでなく，設計に深く携わっていただいた佐藤真平先輩，最適化にご助力頂いた Thiem Van Chu 先輩に心よりお礼申し上げます．

また数々のご助言を頂いた吉瀬研究室の皆様，特に設計に関して相談に乗っていただいた高前田伸也先輩，笹河良介先輩，池田貴一先輩，デバッグや校閲に協力していただいた小林諒平先輩には大変お世話になりました．

参考文献

[1] Opencores. http://opencores.org/.

[2] Intel. Tera-scale computing research overview. http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-tera-scale-research-paper.pdf.

[3] Intel. Teraflops Research Chip Overview. http://download.intel.com/pressroom/kits/Teraflops/Teraflops_Research_Chip_Overview.pdf.

[4] Intel. Single-Chip Cloud Computer: Project. http://www.intel.com/content/www/us/en/research/intel-labs-single-chip-cloud-computer.html.

[5] Intel. Many Integrated Core Architecture – Advanced. http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html.

[6] KALRAY. MPPA MANYCORE MPPA 256. http://www.kalray.eu/products/mppa-manycore/mppa-256/.

[7] John D. Davis, Stephen E. Richardson, Charis Charitsis, and Kunle Olukotun. A chip prototyping substrate: the flexible architecture for simulation and testing (FAST). *ACM SIGARCH Computer Architecture News*, Vol. 33, No. 4, pp. 34–43, nov 2005.

[8] Sewook Wee, Jared Casper, Njuguna Njoroge, Yuriy Tesylar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun. A practical FPGA-based framework for novel CMP research. In *FPGA '07 Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pp. 116–125, New York, NY, USA, 2007. ACM.

[9] Eric S. Chung, Michael K. Papamichael, Eriko Nurvitadhi, James C. Hoe, Ken Mai, and Babak Falsafi. ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, Vol. 2, No. 15, pp. 1–32, jun 2009.

[10] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, KrsteAsanović. RAMP gold: an FPGA-based architecture simulator for multiprocessors. In *DAC '10 Proceedings of the 47th Design Automation Conference*, pp. 463–468, New York, NY, USA, 2010. ACM.

[11] 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 (ACS), Vol. 4, No. 1, pp. 24–42, feb 2011.