修士論文　　2004 (平成 16) 年度

論文題目

ISIS-SimpleScalarの実装

慶應義塾大学大学院理工学研究科
開放環境科学専攻

学籍番号 ******** 藥袋 俊也

指導教員 教　授 天野 英晴

慶應義塾大学大学院理工学研究科

2005 年 3 月

Summary of M. S. Thesis

School	Student No.	SURNAME, Firstname
Open and Environmental System	********	MINAI, Toshiya

Title

Implementation of ISIS-SimpleScalar

Recently, multiprocessors have been employed in a lot of systems and the structure have become increasingly complex. For developing recent complex computing systems, it is essential to examine many factors of the system using software simulator in the early step of design.

Several software simulators have been provided and used to estimate the performance of computing systems. For example, SimOS by the University of Stanford, SimpleScalar by the University of Wisconsin, and ISIS developed by Amano Laboratory in Keio University.

SimOS provides multiprocessor system simulators with high-performance processor models. Since SimOS was developed aiming at the evaluation of a specific parallel computer, users need to build a new simulator model from scratch if they want to evaluate a new multiprocessor systems.

SimpleScalar tool set provides high-performance processor simulators. It is widely used for the researches concerning a single-processor or the architectures of processors. But it cannot treat parallel computing systems with more than one processor, shared memory and networks.

ISIS is the parallel computer simulator library written in C++ language. It provides components of parallel systems including cache and some network topologies. Simulators can be built by connecting these components. Users can implement their specific simulators easily using the components provided by ISIS library, and can simulate the behavior of the target system exactly. However, ISIS doesn't provide recent high-end processors which support multiple-instruction-issue, out-of-order execution, and branch prediction.

As stated above, there is no simulator that can easily implement a new system with high-end processors.

ISIS-SimpleScalar combines the benefits of both ISIS and SimpleScalar by incorporating SimpleScalar into ISIS library. ISIS-SimpleScalar enables detailed simulation of parallel systems with high-end processors.

In this report, the implementation of ISIS-SimpleScalar will be presented. Simulation results of both a single-processor and a multiprocessor system built by ISIS-SimpleScalar show that the simulation speed is practical and ISIS-SimpleScalar provides detailed statistical data for the target system.

Graduate School of Science and Technology, KEIO UNIVERSITY

(The summary of thesis should be written within 35 lines)

修 士 論 文 要 旨

開放環境科学専攻	学 籍 番 号 ********	フリガナ 氏 名	ミナイ トシヤ 薬袋 俊也

(論文題名)

ISIS-SimpleScalar の実装

(内容の要旨)

近年、オンチップマルチプロセッサがさまざまな用途で用いられるようになり、さまざまな構成の並列計算機が提案されている。それに伴い、開発過程におけるシミュレーションによる事前検証や性能評価がとても重要になってきている。

これまでに数多くのシミュレータが実装され、性能評価システムとして使用されてきた。代表的なシミュレータとして、Stanford 大学の SimOS、Wisconsin 大学の SimpleScalar、本研究室の ISIS が挙げられる。

SimOS は、高性能なプロセッサのモデルを提供し、マルチプロセッサシステムのシミュレーションに対応している。しかし、特定のアーキテクチャを持つ並列計算機の性能評価を目的として開発されたため、新たな並列計算機を評価する必要性が生じた場合には新しいシミュレータを実装する必要がある。また、シミュレータの実行に特定の動作環境を必要とするという欠点も持つ。

SimpleScalar は、高性能なプロセッサの詳細なモデルを提供しており、シングルプロセッサシステムやプロセッサアーキテクチャの検証に広く利用されているが、マルチプロセッサシステムのシミュレーションに対応していない。

ISIS は、細部まで考慮した記述を行うため動作が正確であり、汎用的な部品をライブラリとして提供しているためシミュレータの実装が比較的容易である。しかし、近年主流となっている複数命令同時発行、out-of-order 実行、分岐予測などをサポートしたプロセッサを提供していない。

このように、近年の高性能なプロセッサを詳細にモデルし、且つ、あらゆる構成のマルチプロセッサシステムを簡単に構築できるシミュレータが存在しないという問題点があった。そこで本研究では、SimpleScalar が提供する高性能なプロセッサをマルチプロセッサシステムシミュレーションへ対応させ、ISIS によって記述されたシステムと接続できるようにした。

本研究で実装した ISIS-SimpleScalar を用いてシングルプロセッサ、およびマルチプロセッサシステムのシミュレータを構築し、評価を行った。その結果、現実的な時間内でシミュレーションを終えられることやマルチプロセッサシステムに関する詳細なシミュレーションを行えることなどが確かめられた。

慶 應 義 塾 大 学 大 学 院 理 工 学 研 究 科 前 期 博 士 課 程

(内容の要旨は約 25 行程度で記入のこと)

目 次

図目次

表 目 次

第 1 章

緒論

計算機システムは、現在に至るまで急速な性能向上を続けてきた。計算機の高性能化が新たな利用機会を生み出し、新たな利用機会が計算機の更なる性能向上を促すという循環は、今も衰える様子を見せない。多くの場合、利用者が計算機を選択する際には、性能が重要な関心事となる。特に、性能向上のために複数のプロセッサを搭載する構成を持つ並列計算機に対しては、この傾向が顕著である。最近では 1 つのチップ内に複数のプロセッサを搭載したオンチップマルチプロセッサがさまざまな用途で用いられるようになってきている。並列計算機開発者は、利用者の要求に応えるために、高い性能を持つ計算機をなるべく短期間で開発しなければならない。

今日の計算機は、ハードウェア設計者が投入したさまざまな性能改善技法の組み合わせで構成されている。この複雑な構成を持つ計算機上で、同じく複雑な演算を行うアプリケーションプログラムが実行され、莫大な量の計算をこなす。計算機の総合性能は、これらすべての要素の組み合わせによって決定される。したがって、計算機開発の設計段階で新規の性能改善技法を考案したとしても、その技法がどの程度の効果を持つのかは、計算機を完成させてすべての要素を組み合わせた後で評価を行わなければ正確には判断できない。評価の結果、その技法には効果がないということが明らかになれば、開発をやり直すことになる。しかし、計算機開発は期間や資金などのコストを要する活動であるため、可能ならば手戻りを伴わずに開発を行いたい。

この要求を満たす手段として、計算機の事前性能予測が有効である。計算機を完成させることなく完成後の計算機の性能を予測することができれば、どのような性能改善技法が有効なのかを、計算機開発過程の早期に判断することができる。

計算機の事前性能予測手段として広く用いられているのが、ソフトウェアシミュレーションである。ソフトウェアシミュレーションとは、ターゲットマシン (予測対象の計算機) の動作を、シミュレータと呼ばれるソフトウェアプログラムを使って模倣する予測手法である。シミュレータをホストマシン (シミュレータを実行する計算機) 上で実行することで、性能評価結果を得る。

これまでに数多くのソフトウェアシミュレータが実装され、性能評価システムとして使用されてきた。代表的なシミュレータとしては、Stanford 大学の SimOS[1][2]、Wisconsin 大学の SimpleScalar[3][4][5][6]、本研究室の ISIS[7][8] が挙げられる。

SimOS は、近年のプロセッサやキャッシュ等のモデルを提供し、マルチプロセッサシステムのシミュレーションに対応している。しかし、特定のアーキテクチャを持つ並列計算機の性能評価を目的として開発されたため、新たな並列計算機を評価する必要性が生じた場合には新しいシミュレータを実装する必要がある。このオーバヘッドは研究者にとって大きな負担となる。また、SimOS は並列計算機上のアプリケーションの振る舞いを評価するだけでなく、オペレーティングシステムの動作まで含めた完全な並列計算機の性能評価を目的として開発されたため、実行時にシミュレーション方式を動的に切り替えながらシミュレーションを行うことができる。しかし、この方

式を採用するシミュレータは、切り替え可能な方式の数が増加するにつれて複雑度が上昇し、柔軟性は減少するという傾向がある。また、シミュレータの実行に特定の動作環境を必要とするという欠点も持つ。

SimpleScalar は、アプリケーションプログラムおよびプロセッサアーキテクチャの性能解析を目的とするシングルプロセッサシミュレータである。Alpha、ARM、x86 アーキテクチャの投機的実行や分岐予測などを扱うことができる。また、高速なものから精度の高いものまで、精度と速度に応じたいくつかの種類のシミュレーションを行うことができる。SimpleScalar はシングルプロセッサシステムやプロセッサアーキテクチャの検証には幅広く利用されているが、マルチプロセッサシステムのシミュレーションに対応していない。

ISIS は、細部まで考慮した記述を行うため動作が正確であり、汎用的な部品をライブラリとして提供しているためシミュレータの実装が比較的容易である。しかし、複数命令同時発行、out-of-order 実行、分岐予測などをサポートしたプロセッサを提供していないため、近年主流となっているような高性能なプロセッサを搭載したシステムのシミュレーションを行うことができない。

このように、近年の高性能なプロセッサを詳細にモデルし、且つ、あらゆる構成のマルチプロセッサシステムを簡単に構築できるシミュレータが存在しないという問題点があった。そこで本研究では、SimpleScalar が提供する詳細で高性能なプロセッサをマルチプロセッサシステムシミュレーションへ対応させ、ISIS によって記述されたシステムと接続できるようにした。

本論文の構成を次に示す。

2 章では、並列計算機の性能評価方法について概観する。特に、ソフトウェアシミュレーションについて、シミュレーション方式と実装形態に着目し詳説する。また、既存のシミュレータである SimOS、SimpleScalar、ISIS について詳しく紹介し、最後に本研究の目的を述べる。

3 章では、高性能なプロセッサを搭載したさまざまな構成のマルチプロセッサシステムを比較的容易に構築できるシミュレータとして開発した ISIS-SimpleScalar の実装を示す。

4 章では、ISIS-SimpleScalar を用いて実装したシングルプロセッサ、およびマルチプロセッサシステムのシミュレータの評価を行うことによって、今回実装した ISIS-SimpleScalar の性能やシミュレーション結果の妥当性を検証する。

最後に 5 章にて総括する。

第 2 章

背景

近年、計算機の処理能力向上の解決法として、並列処理によるアプローチが研究されている。最近ではオンチップマルチプロセッサがさまざまな用途で用いられるようになり、開発過程においてシミュレーションによる事前検証や性能評価が重要になっている。

ISIS-SimpleScalar を実装するにあたって、本章では既存の性能評価手法や実現形態について概観する。

まず、現在までに行われてきた研究事例に基づき、並列計算機の性能評価手法について整理する。続いて、本研究で注目する性能評価手法であるソフトウェアシミュレーションについて、シミュレーション方式および実装形態別に解説する。続いて、既存のシミュレータである SimOS、SimpleScalar、ISIS について詳しく紹介し、最後に本研究の目的を述べる。

2.1 計算機の性能評価方法

計算機の性能評価方法は、解析モデルによる方法、実機計測による方法、シミュレーションによる方法の 3 つの方法に大別される。

2.1.1 解析モデルによる方法

解析モデルを用いる方法は、評価対象の計算機の構成要素: 例えばプロセッサなどの動作が確率的に行われるものと仮定する方法である。比較的簡単なモデルを使用するため取り扱いが簡単だが、メモリアクセスの局所性などが反映されないなどの欠点があるため、高い精度は望めない。代表的なモデルに、待ち行列モデルと確率モデルがある。

待ち行列モデルは、プロセッサのバスに対するアクセスなどを待ち行列を用いたモデルによって記述する方法である。各プロセッサは一定の確率でアクセスを発行し、そのアクセスが待ち行列に入れられる。単純なモデルであれば待ち行列理論を用いて解くことが可能であるが、実際の計算機に相当するモデルを理論的に解くことは困難である。そのため、待ち行列によってモデル化された系を後述するシミュレーションを用いて解くのが一般的である。

確率モデルは、プロセッサ、バス、キャッシュなどが一定の時間間隔ごとに確率的に状態遷移するものと見なし、このモデルをマルコフ解析の手法を用いて解く方法である。ただし、実際には状態遷移行列が膨大になるため、系の定常状態を仮定するなどの手法が用いられる。また、後述するようにシミュレーションと併用する手法もある。

2.1.2 実機計測による方法

実機計測による方法は、評価対象の計算機に計測ハードウェアを付加してデータを収集する方法である。

この方法による計測システムとして、慶應義塾大学で開発されたバス結合型並列計算機 ATTEMPT-0 の交信評価システム [9]、スイッチ結合型並列計算機 SNAIL の交信評価システム MAID がある。前者は、計算機のプロセッサボードにタイマとカウンタを搭載した測定ボードを接続することで、バス利用率やバス使用権獲得のための待ち時間などのバスに関する情報と、キャッシュヒット率などのキャッシュメモリに関する情報を収集するシステムである。後者は、計算機のネットワークボードに接続することで、共有メモリへのアクセス数、再送されるパケット数などの情報を収集するシステムである。

この方法による評価では、評価対象そのものを実際に動作させて評価を行うため、評価対象の挙動に影響を及ぼすことなく情報を収集できるので、正確な評価を行うことが可能である。しかし、測定を行うためには、稼働状態にある評価対象の計算機と、ハードウェアで実現した測定機器が必要である。ハードウェアで実現した測定機器は、高コストで柔軟性に欠けるという欠点がある。また、例えば商用プロセッサ内部の挙動など、測定できない部分は評価できない。

2.1.3 シミュレーションによる方法

一般に、シミュレーションとは、評価対象を何らかのモデルで表現し、モデルに基づいて評価対象を模擬する装置を構築して動作させ、その挙動を調べる手法である。状況にもよるが、ほとんどの場合モデルは数式やプログラムとして記述され、模擬装置には計算機が使用される。

模擬装置の挙動が評価対象の挙動とどの程度一致するのかは、モデルの精度によって決定される。モデルが正確であるほど、模擬装置の挙動は評価対象の挙動に近くなり、得られる評価結果が正確になる。しかし、それに伴ってモデルを表す数式やプログラムが複雑化あるいは大規模化し、模擬装置が処理しなければならない演算量が増加するため、シミュレーションに要する時間は長くなる。このように、評価の精度とシミュレーションに要する時間の間にはトレードオフの関係がある。

計算機による模擬装置としては、ソフトウェア、専用ハードウェア、ソフトウェアと専用ハードウェアの組み合わせの 3 通りが考えられる。多くの場合、速度に難点はあるものの、取扱いの容易さからソフトウェアのみを用いる方法が選択される。模擬装置であるソフトウェアを何らかの計算機上で稼働させることで、シミュレーションを行う。

本論文では、評価対象の計算機をソフトウェアシミュレーションによって性能評価する方法に焦点を当てる。以降、模擬装置であるソフトウェアをシミュレータ、評価対象の計算機をターゲットマシン、シミュレータを稼働させる計算機をホストマシンと呼ぶ。

2.2 シミュレーション方式

この節では、本研究で注目する性能評価手法であるソフトウェアシミュレーションについて、シミュレーション方式および実装形態別に解説する。

2.2.1 確率モデルシミュレーション

確率モデルシミュレーションは、計算機がプロセッサとメモリおよびその間の接続のみで構成されるものと見なし、プロセッサが発行するメモリアクセスの種類や比率などを確率で与えて評価を行う方法である。シミュレーションによる評価方法の中ではもっとも簡単なモデルであり、シミュレータの実装が容易で、他のシミュレーション方式よりもシミュレーションの実行時間が短いという利点がある。しかし、ターゲットマシン上のアプリケーションの動作を結果に反映しないため、評価の精度が低いという欠点がある。そのため、現実に即さない結果を導いてしまう可能性がある。

超並列計算機の結合網などの巨大なシステムをシミュレーションで評価する場合には、シミュレーションの実行時間を現実的な範囲内に留めるためにこの手法を用いることがある。しかし、得られた評価結果の信頼性は低いので、システム全体の動作の概略を把握する程度の用途に限定するべきであろう。

2.2.2 トレース駆動型シミュレーション

トレース駆動型シミュレーションは、確率モデルシミュレーションと同様に計算機の構成要素がプロセッサとメモリとその接続のみであるものと見なし、何らかの手段を用いて取得したアドレストレースデータと呼ばれるメモリアクセス履歴に基づいてプロセッサのメモリアクセスを発行させて評価を行う方法である。命令セットやメモリ階層、分岐予測機構、命令パイプラインなどの解析によく使用される。

アドレストレースの取得方法には、他のシミュレーション方式のシミュレータから取得する方法と実機から取得する方法の 2 通りがある。シミュレータから取得する方法では、後述する実行駆動型シミュレーションや命令レベルシミュレーションが用いられる。実機から取得する方法には、主として次の 2 つの方法がある。

- プロセッサのマイクロコードに対してアドレストレース出力用の変更を加える方法
- アドレストレースを取得する計算機用に生成された実行可能ファイルを解析し、アドレストレース出力用の命令列を埋め込む方法

トレース駆動型シミュレーションは、莫大な量のトレースデータを取り扱う必要があるため、プロセッサ数や実行時間に対する制約が大きい。さらに、同期のためのメモリアクセスはトレースデータを収集した並列計算機のアーキテクチャに依存するため、解析できるアーキテクチャの範囲が限定されるという大きな欠点を持つ。

また、トレースデータ中のメモリアクセスの順序は基本的に固定されているため、シミュレータ上のメモリシステムの構成の変更によりアクセスの遅延時間が変化した場合にも、本来生じるべきメモリ参照順序の変更が生じないので、正確なシミュレーションを行うことができない。

2.2.3 実行駆動型シミュレーション

実行駆動型シミュレーションは、ターゲットマシン上で実行されるアプリケーションプログラムの内部にシミュレータを呼び出すコードを埋め込み、これをホストマシン上で実行してプログ

ラム実行中に解析を行う方法である。シミュレータの実装形態と実行方法を規定したシミュレーション方式であり、ターゲットマシンに対するモデル化の手段は規定されていない。

並列計算機を対象とする場合、プロセッサ間の相互作用に重点を置いてターゲットマシンをモデル化する場合が多いようである。この場合は、アプリケーションプログラムのソースコードを解析し、同期操作や共有メモリアクセスなどのプロセッサ間の相互作用を伴う操作をメモリシステムのシミュレータを呼び出すコードに置き換える。その他の通常の計算などにはアプリケーションコードをそのまま使用する。このプログラムをコンパイルしシミュレータコードとリンクすることで、シミュレータが埋め込まれたアプリケーションプログラムを作成する。これをホストマシン上で実行することで評価を行う。ターゲットマシン上のプロセッサ間の相互作用は、ホストマシン上のプロセス間の相互作用にマッピングされる。

実行駆動型シミュレーションは、アプリケーションプログラムが実際に動く様子をシミュレートするため、トレース駆動型シミュレーションよりもイベント順序が正確である。しかし、通常の計算はアプリケーションコードがそのまま高速に実行するのに対し、同期操作などターゲットマシンをモデル化した箇所はシミュレータが低速に実行するため、両者の実行速度に大きな差異が生じる。そのため、アプリケーションプログラムの時間軸方向の振る舞いが本来の振る舞いと異なってしまう可能性がある。また、本来はターゲットマシン上で実行されるはずのアプリケーションプログラムをホストマシン上で実行するため、評価結果はホストマシンの性能や挙動の影響を受ける。これらの特徴があるため、イベント順序がより正確であっても、必ずしもトレース駆動型シミュレーションより正確であるとは限らない。

2.2.4 バイナリ変換型シミュレーション

バイナリ変換型シミュレーションは、コンパイル後のアプリケーションのオブジェクトコードに対して時刻情報などの各種情報収集を行うコードを埋め込むバイナリ変換を施し、アプリケーションの実行中に解析データを収集する方法である。実現手段がやや異なるが、アプリケーションプログラム中にシミュレータを埋め込む実行駆動型シミュレーションの特殊な形態と見なすことができる。実行駆動型シミュレーションと同様に、ターゲットマシンに対するモデル化の手段は規定せず、シミュレータの実装形態と実行方法を規定したシミュレーション方式である。

アプリケーション実行中の大半の命令を直接実行するため、実行駆動型シミュレーションと同様に実行速度が高速であるという利点がある。さらに、実行駆動型シミュレーションの欠点であった時刻計測も、実行駆動型シミュレーションと比較してかなり正確に行うことができる。ただし、ターゲットマシン用の機械語列をホストマシン上で直接実行するため、ホストマシンのプロセッサの命令セットがターゲットマシンのプロセッサの命令セットと一致している必要がある。

2.2.5 命令レベルシミュレーション

命令レベルシミュレーションは、プロセッサの機械語命令のレベルまでモデル化して評価する方法である。ターゲットマシン用のアプリケーションプログラムの実行可能ファイルを 1 機械語命令ずつ読み込み、プロセッサ内のレジスタやキャッシュの状態遷移、バスやメモリ、その他のデバイスをシミュレートする。どのようなアーキテクチャにも対応可能で柔軟性が高く、精度の高いシミュレーションが可能であるが、シミュレーションに要する時間が長い。

命令レベルシミュレーション方式は、かつては実行時間の面から見て並列計算機を評価するに

は非現実的な方法であった。しかし、実行環境としての計算機の速度が飛躍的に向上し、シミュレータの実行速度が改善されたため、現在では正確な評価を行う場合にはこの方法が使用されることが多い。

2.2.6 回路シミュレーション

命令レベルシミュレーションよりもさらに精緻なシミュレーション方式として、RTL (Register Transfer Level) シミュレーションとゲートレベルシミュレーションがある。RTL シミュレーションは半導体チップ内のレジスタ間のデータ転送までを、ゲートレベルシミュレーションはチップ内の論理ゲートの動作までをモデル化しシミュレートする。

これらのシミュレーション方式は、デジタル回路設計の動作検証に用いられる。回路内の構成要素の挙動をソフトウェアシミュレーションで追跡することで、回路を物理的なチップとして実現する前に動作検証を行うことができる。チップ製造は非常に高価であり、また一度チップ化した回路は後から修正を行うことが不可能であるため、事前に可能な限りシミュレーションによる動作検証を行う。

これらのシミュレーション方式は非常に精度が高く、計算機全体の性能評価を行うには速度が遅すぎるという問題がある。また、現在の計算機の構成要素のほとんどは同期式制御を用いているため、クロック単位まで正確な命令レベルシミュレーション以上にシミュレーションの精度を上げても、評価結果は変化しない。

本研究では性能評価手段としてのシミュレーションを扱うため、動作検証を目的としたシミュレーション方式である RTL シミュレーションとゲートレベルシミュレーションは扱わない。

2.2.7 複合型シミュレーション

複合型シミュレーションは、前述した複数のシミュレーション方式を単体のシミュレータ内に実装し、実行時にシミュレーション方式を動的に切り替えながら実行を行う方式である。複合型シミュレーションは、他の複数のシミュレーション方式を動的に切り替えるメカニズムを用いることを規定している。使用される個々のシミュレーション方式については規定がなされていないが、切替えの前後でターゲットマシンの時刻や状態を受け渡す必要があるため、各シミュレーション方式がこれらの情報を共有するメカニズムを備えている必要がある。また、シミュレーション方式を動的に切り替えるメカニズムが必要である。

切替え可能なシミュレーション方式の数が増加すると、全シミュレーション方式に共通する要素が減少する。相違がある部分同士で情報を共有するためには、個別に相違を打ち消す情報共有メカニズムを使用しなければならない。したがって、複合型シミュレーションは切替え可能なシミュレーション方式の数が増加するにつれて複雑度が上昇し、柔軟性は減少する。また、ある複合型シミュレーション方式のシミュレータがあったとき、そのシミュレータは切替え可能なシミュレーション方式の中で最も複雑なシミュレーション方式以上の複雑度を持つ。

2.3 関連研究

2.3.1 SimOS

SimOS は、2.2.7 項で説明した複合型シミュレーション方式の代表的なシミュレータであり、命令レベルシミュレーションとバイナリ変換型シミュレーションを単体のシミュレータ上に実装し、実行時にシミュレーション方式を動的に切り替えながら実行を行う。

SimOS は並列計算機上のアプリケーションの振る舞いを評価するだけでなく、オペレーティングシステムの動作まで含めた完全な並列計算機の性能評価を目的として開発された。オペレーティングシステムのブート時の処理までシミュレートする必要性から、評価にあまり関係しない部分はあまり正確ではないが高速に動作するシミュレーション方式を、正確な評価を必要とする部分には実行時間を惜しまずに正確なシミュレーション方式をと言うように、動的なシミュレーション方式の切り替えをサポートしている。

SimOS は SGI の Challenge シリーズ等の UNIX マルチプロセッサ上にシミュレーション層と呼ばれる階層を形成し、ターゲットマシンのハードウェアをシミュレートする。SimOS で用いられている階層構造を図 2.1 に示す。シミュレートされる並列計算機上のオペレーティングシステムやアプリケーションはシミュレーション層のさらに上位に配置される。シミュレーション層は大別して CPU シミュレータとメモリやデバイス等のハードウェアシミュレータから構成されている。

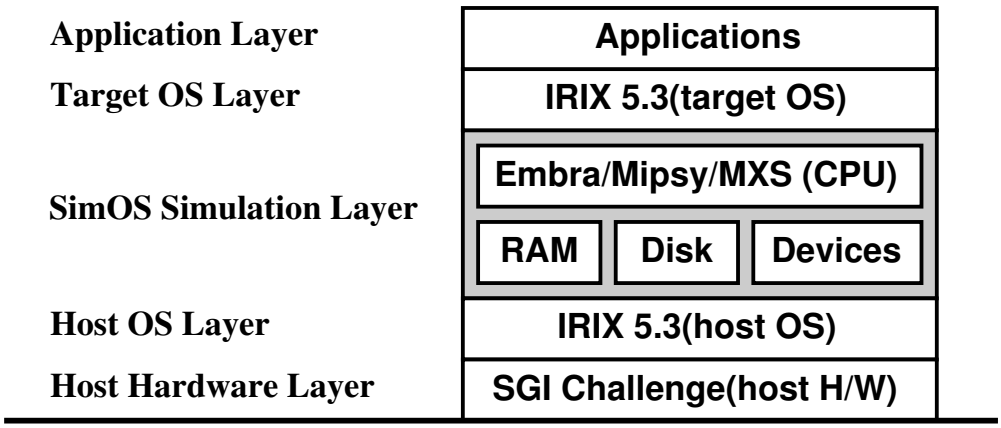


図 2.1: SimOS の階層構造

ユーザに速度と正確さの選択権を与えるために、シミュレーション層は高速なものから正確なものまで多数の実装がなされている。

- **直接実行モード**
  - ターゲットマシンのプロセッサをホストマシン上のプロセスに写像
  - 命令の大半はホストマシン上で直接実行
  - ホストとターゲットが同一である場合にのみ使用可能
  - ホストのハードウェアの機能をそのまま上位階層に提供
  - 実行時間は直接実行した場合の 2 倍程度
  - オペレーティングシステムのデバッグやテストに有効

  - 性能評価には不向き

- **バイナリ変換モード**
  - 実行時にオブジェクトコードのバイナリを動的に変換し、ターゲットマシンのオブジェクトをホストマシン上で実行する
  - ターゲットマシン上での実行時間を計測できる
  - 実行時間は 12 倍程度
  - キャッシュのヒット/ミス判定を行う場合は 35 倍程度
  - 単純なアーキテクチャやオペレーティングシステムの研究に有効
- **詳細モード**
  - プロセッサ、キャッシュ、メモリ等を正確にシミュレートする
  - 数千倍の実行時間を要する
  - メモリシステムについてはさらにいくつかの選択肢がある

CPU シミュレータは Embra/Mispy/MXS の 3 種類が用意されており、実行時に動的に変更を行うことができる。Embra は前述したバイナリ変換型シミュレータで、実行時間が実機の数十倍程度である高速な実行を行うことができる。Mipsy は命令レベルシミュレータで、プロセッサ内の命令実行をフェッチ - デコード - 実行の簡単な 3 段命令パイプラインで処理するプロセッサモデルを使用している。実行時間は実機の数千倍程度である。MXS はさらに正確な命令レベルシミュレータで、プロセッサ内部で行われる動的スケジューリングである複数命令同時発行や out-of-order 実行、 分岐予測までシミュレートする。極めて正確な動作をするが、実機の数万倍という膨大な実行時間がかかる。

SimOS を用いた場合、必要な部分だけ正確な方式で実行できるため、正確さを保ったまま実行を高速化することができる。例えばオペレーティングシステムのブート部分を Embra で高速に実行し、キャッシュ使用開始時に Mipsy に変更、さらにキャッシュの状態が安定したところから MXS で詳細な評価を行うといったことが可能である。

2.3.2 SimpleScalar

SimpleScalar は、2.2.5 項で説明した命令レベルシミュレーション方式の代表的なシミュレータであり、1992 年に Wisconsin 大学の Multiscalar プロジェクトの一環として開発が開始された。SimpleScalar ツールセットは、コンパイラ、アセンブラ、リンカ、ライブラリ及びシミュレータからなる。シミュレータは、図 2.2 のようにホストマシンの C コンパイラでコンパイルでき、シミュレータ上で実行するバイナリは、SimpleScalar が提供する GNU Gcc、GNU As、GNU Ld によって生成することができる。

SimpleScalar は、MIPS、Alpha、ARM など、複数のアーキテクチャをモデルにしたプロセッサシミュレータをアーキテクチャ毎に提供している。また、MIPS 互換の PISA、Alpha、Power PC、x86、ARM などの複数の命令セットアーキテクチャをサポートしており、それぞれの命令セットに対応する GNU Gcc、GNU As、GNU Ld のパッチが提供されている。

本研究では、ISIS で提供している R3081 と同じ MIPS 系であるという理由から、MIPS アーキテクチャをモデルにした PISA 依存のプロセッサシミュレータをマルチプロセッサシステムに対応

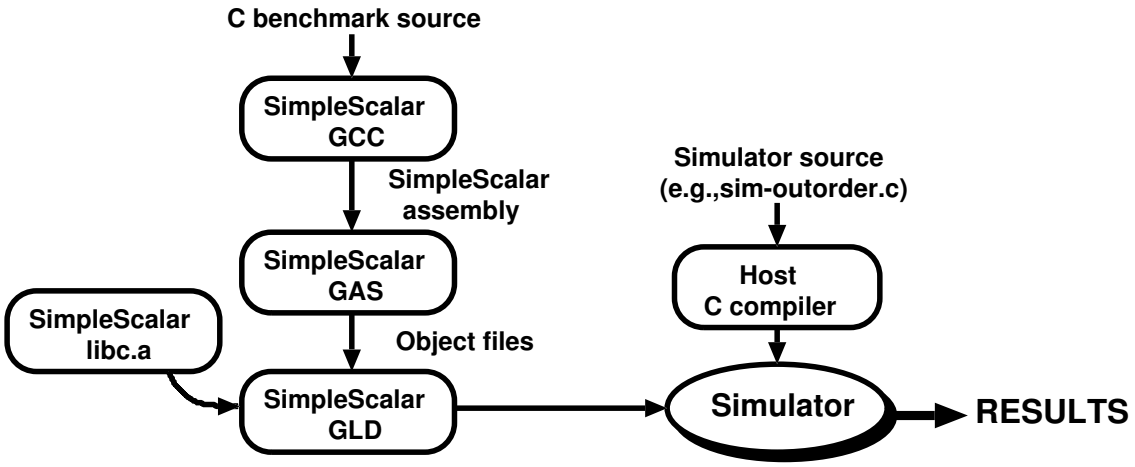


図 2.2: SimpleScalar tool set overview

させた。以後の説明は、MIPS アーキテクチャをモデルにした PISA 依存のプロセッサシミュレータについて行う。

2.3.2.1 命令定義

SimpleScalar のシミュレータが実行する命令について説明する。

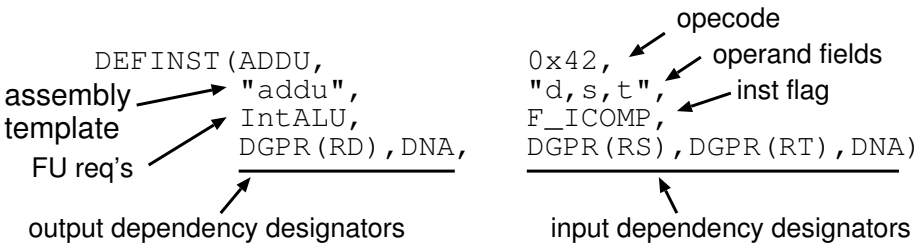
命令定義フォーマットは図 2.3 のような形式になっており、すべての命令が 1 つのファイル (`machine.def`) 中で、定義されている。

1 番目の要素はシミュレータプログラム内で使用する識別子、2 番目の要素は命令のオペコードである。3、4 番目の要素は逆アセンブラによって使用され、それぞれ文字列とオペランドフィールドを表す。5 番目の要素は実行に必要な Functional Unit 名、6 番目の要素は命令の種類である。7、8 番目の要素にはその命令が書き込みを行うレジスタ番号、9〜11 番目の要素には読み出しを行うレジスタ番号が定義されている。

プロセッサシミュレータは、命令をデコードすることにより識別子 (`ADDU`) を得た後、machine.def を読み出すことによって、対応するレジスタ番号や Functional Unit 名などの情報を取得する。その後、識別子_`IMPL (ADDU_IMPL)` を呼び出し、命令を実行する。この例では、レジスタ番号 RS、RT のレジスタの内容を加算し、レジスタ番号 RD のレジスタに書き込む。

2.3.2.2 レジスタ

整数レジスタは `r0` から `r31` まで、浮動小数点単精度レジスタは `f0` から `f31` まである。浮動小数点レジスタは 2 つ繋ぎ合わせることにより倍精度での使用が可能である。また、プログラムカウンタ `PC`、計算結果が 2 レジスタに渡る乗算命令や除算命令に使用される `HI`、`LO`、浮動小数点レジスタの状態を示す `FCC` を持つ。図 2.4 に示す。




```
#define ADDU_IMPL    \
{ \
   SET_GPR(RD, GPR(RS) + GPR(RT));    \
}
```

GPR = General Purpose Register

図 2.3: 命令定義フォーマット

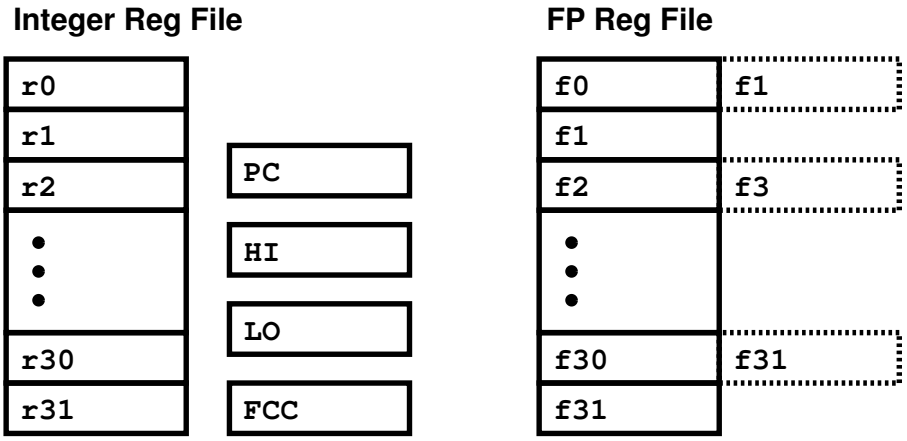


図 2.4: レジスタ

2.3.2.3 アドレス空間

アドレス空間の割当は図2.5のようになっている。`0x00000000`〜`0x003fffff`のアドレスは使用しない。`0x00400000`〜`0x0fffffff`のアドレスはプログラムを格納する際に使用される。`0x10000000`〜`0x7fffbfff`のアドレスはプログラム実行中に必要になったデータに割り当てられる。スタックデータもこの領域を使用する。スタックは`0x7fffbfff`側からアドレス番地が小さくなる方向に割り当てられていく。引数、環境変数などのデータは`0x7fffc000`〜`0x7fffffff`に格納される。

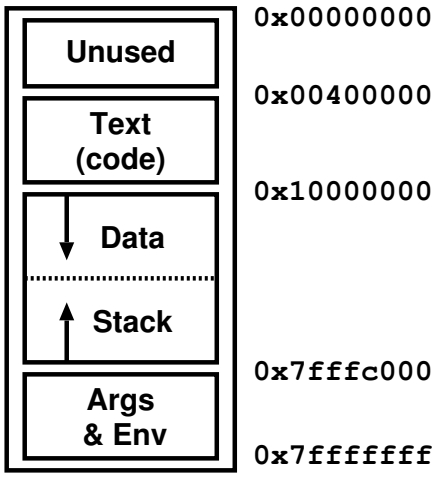


図 2.5: アドレス空間

2.3.2.4 シミュレータ

SimpleScalar は、表 2.1 に示すように、性能、柔軟性、精度の異なる複数のプロセッサシミュレータを幅広く提供している。

表 2.1: SimpleScalar baseline simulator models.

Simulator	Description	Lines of code	Simulation speed
sim-safe	Simple functional simulator	320	6MIPS
sim-fast	Speed-optimized functional simulator	780	7MIPS
sim-profile	Dynamic program analyzer	1300	4MIPS
sim-bpred	Branch predictor simulator	1200	5MIPS
sim-cache	Multilevel cache memory simulator	1400	4MIPS
sim-outorder	Detailed microarchitectuaral timing model	3900	0.3MIPS

- sim-fast
  - functional シミュレーションを行う
  - 非常に高速であるが、精密さに欠ける

  - キャッシュを保持しない
  - すべての命令が連続的に処理され、並列処理されない

- sim-safe
  - sim-fast に、正確なメモリの割り当てとメモリアクセス制御が加えられたシミュレータ
- sim-profile
  - 命令、アドレス、メモリアクセス、分岐などの詳細な統計を取るためのシミュレータ
- sim-bpred
  - branch predictor を考慮したシミュレータ
  - branch predictor として、次のものを設定できる
    * nottaken – 常に分岐しないと予測
    * taken – 常に分岐すると予測
    * perfect – 分岐予測が 100%当たると仮定
    * bimod – 2 ビットカウンタによる予測
    * 2lev – 2 レベル予測機構による予測
    * comb – bimodal predictor と 2-level predictor を結合した branch predictor
- sim-cache
  - キャッシュを考慮したシミュレーション
  - キャッシュアクセスのレイテンシを考慮しない
    * 高速
    * キャッシュのヒット率、キャッシュラインの replace などを測定可能
    * プログラムの実行時間におけるキャッシュの効果を測ることはできない
    * L1 キャッシュ、L2 キャッシュ、instruction TLB、data TLB の設定が可能
- sim-outorder
  - 最も詳細で高性能なプロセッサのシミュレータ
  - 複数命令同時発行、out-of-order 実行、分岐予測などをサポート
  - キャッシュ、branch predictor、Fuctional Unit など、プロセッサに関する様々なリソースの設定変更が可能

sim-outorder

本研究では、SimpleScalar が提供する最も高性能なプロセッサモデルである sim-outorder を ISIS に接続した。そのため、sim-outorder について詳説する。

sim-outorder の命令の実行は、図 2.6 に示すように 6 つのステージでパイプライン処理される。各ステージでの処理は以下を想定している。

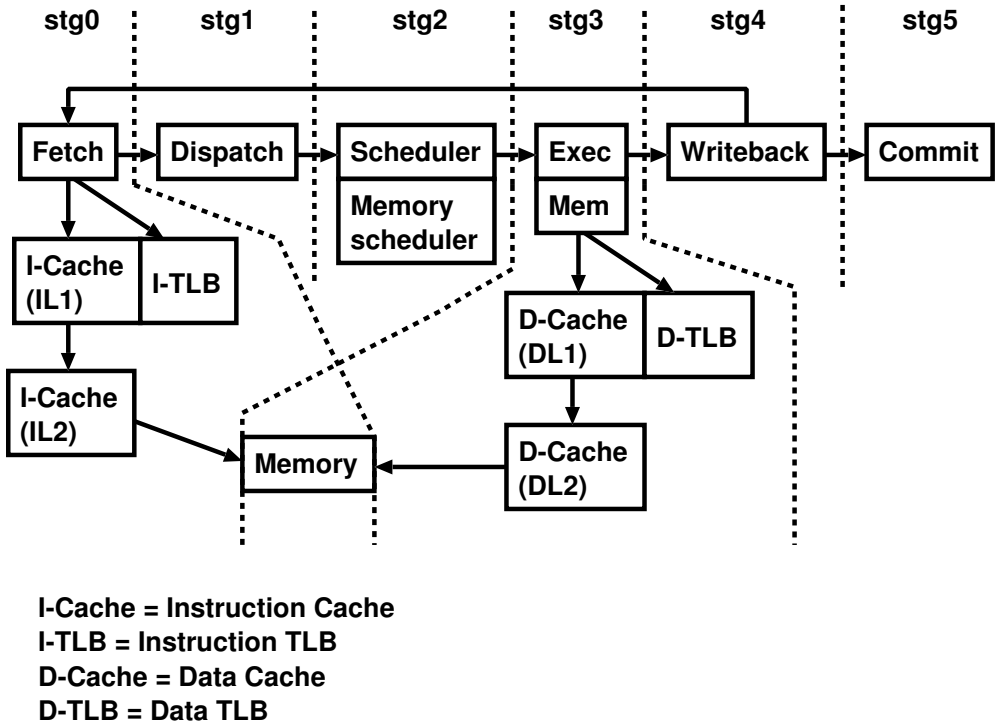


図 2.6: sim-outorder のパイプライン

- Fetch Stage
  - 命令幅分の命令を I-Cache (Instruction Cache) または Memory から取り出し、IFQ に送る
- Dispatch Stage
  - IFQ から取り出した命令のデコード行い、RUU (Register Update Unit) に配置する
  - メモリアクセスアドレス、レジスタに依存関係がなければ、readyq に投入する
- Scheduler Stage
  - メモリアクセスアドレス、レジスタの依存関係などによって、readyq の順番を入れ替える
  - readyq の先頭から命令を取り出し、その命令が使用する Functional Unit やメモリアクセスポートが確保できた場合には、命令を issue する
- Execute Stage
  - 命令の実行、メモリアクセスを行う
  - 実行結果およびメモリから読み出されたデータを eventq に投入する
- Writeback Stage
  - eventq から実行結果を取り出し、分岐予測が失敗しているか否かを判断し、分岐予測が失敗していた場合には、元の正当な状態に戻す

  - 命令の実行が終了したことによって、issue 可能になった命令を readyq に投入する

- Commit Stage
  - RUU の先頭から命令を取り出し、その命令の実行が終わっていた場合には、終了処理を行う
    * 実行結果でレジスタを更新する
    * Store 命令の場合には、実行結果をメモリに書き込む

以上のパイプライン処理は次の for 文によって行われる。

```
for (;;) {
    ruu_commit();
    ruu_writeback();
    ruu_issue();
    ruu_dispatch();
    ruu_fetch();
}
```

- `void ruu_fetch(void)`
  - 命令幅分の命令を I-Cache または Memory から読み出し、IFQ に送る
- `void ruu_dispatch(void)`

  IFQ に送られた命令を in-order で取り出し、その命令のデコートおよび実行、メモリアクセスやレジスタの更新を、すべて同時に行う
  - その命令を RUU に送る
  - メモリアクセスアドレス、レジスタに依存関係がなければ、readyq に投入し、実行可能な順序に並び替える
- `void ruu_issue(void)`
  - レジスタやメモリアクセスアドレスの依存関係によって out-of-order に並べられた readyq 内の命令を先頭から取り出し、利用する Functional Unit やメモリアクセスポートの確保を試みる
  - Functional Unit やメモリアクセスポートが確保できた場合には、Functional Unit を占有する時間またはメモリアクセス時間を求め、その時間により Writeback イベントの順序を決定し、eventq に送る
- `void ruu_writeback(void)`
  - Writeback イベント時間に達する順に並べられた eventq 内の命令を先頭から取り出し、その命令によって利用されていた Functional Unit を解放する
  - 分岐予測が失敗しているか否かを判断し、失敗していた場合には元の正当な状態に戻す
  - 命令の実行が終了したことにより実行可能になった命令を readyq に投入する

- `void ruu_commit(void)`
  - in-order に並べられている RUU の命令を先頭から取り出し、もしその命令が Writeback イベントを終了していた場合には、その命令の終了処理を行う
  - Store 命令の場合には、メモリアクセスポートを確保する

このように、実際の処理では、命令の実行やメモリアクセス、レジスタの更新は、`ruu_dispatch()` において in-order かつ同時に行われる。その後のステージにおいて、レジスタやメモリアクセスアドレスの依存関係、実行時間やメモリアクセス時間が考慮され、Functional Unit やメモリアクセスポートが out-of-order で占有される。そうすることによって、in-order で実行された命令は、out-of-order で実行されたように処理される。

また、for 文を見ると、パイプラインの各ステージが逆順に記述されていることが分かる。こうすることで、命令は同一クロックで複数の Stage をまたがない。つまり、例えば 1 回目のループで `ruu_fetch()` において IFQ に投入された命令は、2 回目以降のループの `ruu_dispatch()` で処理される。2 回目のループで `ruu_dispatch()` において readyq に投入された命令は、3 回目以降のループの `ruu_issue()` で処理される。

2.3.3 ISIS

並列計算機シミュレータライブラリ ISIS は、マルチプロセッサチップに代表される小規模並列計算機の性能評価を目的として開発された。

2.3.3.1 概要

ISIS は主として並列計算機をターゲットとした、計算機シミュレータのための C++言語 [10] によるクラスライブラリである。C++を採用したことによりシステム全体をクラス単位で効率良く管理でき、しかも C 言語と同様に高速に動くプログラムが容易に作成可能である。

2.3.3.2 ユニット

ISIS では、プロセッサやメモリといったハードウェアの機能ブロックをユニットと呼ばれるクラス単位で実装している。ユニットを結合させることにより評価対象の並列計算機を構成する。各ユニットはクロック単位での状態遷移機能を持つ。図 2.7 に並列計算機の構成例を示す。この例ではプロセッサやキャッシュ、メモリ、バスがユニットに相当する。それぞれのユニットが接続されたユニットと通信を行いながら状態遷移を繰り返し、系全体のシミュレーションが行われる。

クラスとして記述されたユニットはそれぞれ独立して取り扱えるように構成されている。また、並列計算機を構成するために必要な、個々のオブジェクト同士を結合するためのインタフェースを各クラスに持たせている。

2.3.3.3 ユニット間の通信方式

ユニット間の結合と通信のインタフェースを規定するために、情報を仲介する存在としてパケット、結合を仲介する存在としてポートを導入する。

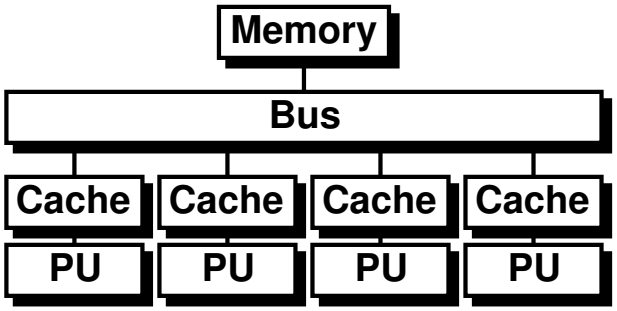


図 2.7: 並列計算機構成例

パケットは、相互接続されたユニット間でやり取りされるすべての情報を表現する。例えばルータ間で送受信されるフリット、プロセッサがバスに出力するメモリアクセス要求等はこのパケットとして扱われる。

ポートは、ユニット同士を結合している通信路への入出力端子を表現する。接続したいユニット同士のポートを接続することで、ユニット間の通信路が構築される。

図 2.8 に結合と通信のモデルを示す。この図では、2 つのユニットが 1 つのユニットの仲介によって通信を行っている。3 ユニットはパケットの送受信によって情報をやり取りする。パケットの転送には、接続されたユニットへのポートを用いる。このように、ユニット間の通信はすべてパケットとポートによって行われる。

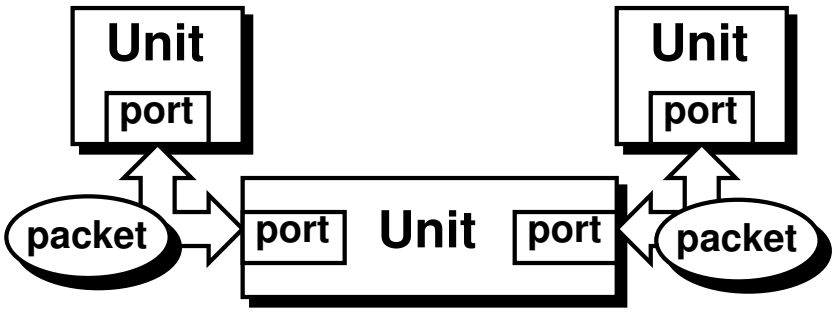


図 2.8: 結合と通信のモデル

2.3.3.4 シミュレータの構築手順

シミュレーション環境はクラスライブラリの形でユーザに提供される。ユーザはこのライブラリを用いてトップモジュールを記述し、任意のシミュレータを構築する。この様子を図 2.9 に示す。

ユーザはまず、評価対象となる並列計算機を機能ブロックに分解し、それぞれの機能ブロックに対応するユニットをライブラリ内から選択する。これらのユニットをトップモジュール内で結合させる。対応するユニットが用意されていない場合には、ライブラリ内のユニットの派生クラスという形で所望の機能を持つユニットを新たに実装する。パケットやポートについても同様に、必要があれば派生クラスを実装する。

トップモジュールおよび派生クラスの記述が終了したら、それらのソースファイルをコンパイ

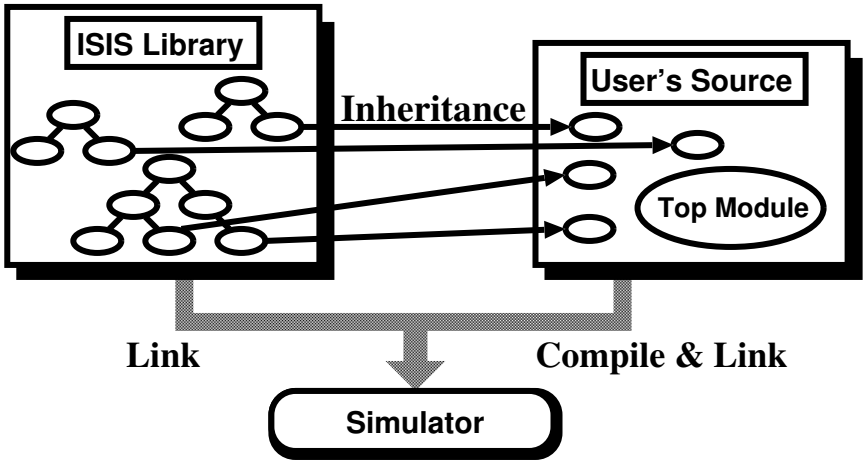


図 2.9: シミュレータの構築

ルし、オブジェクトファイルを生成する。生成したオブジェクトファイルとクラスライブラリをリンクすることで、シミュレータの実行バイナリを生成する。

2.3.3.5 ハードウェアシミュレータ

この節では、並列計算機内の機能ブロックに対応するハードウェアシミュレータの主なものについて説明する。

プロセッサ

プロセッサとしては、MIPS 社の R3000 プロセッサ [11][12] および R3010 浮動小数点演算コプロセッサ [13] が実装されている。

R3000 は、RISC アーキテクチャを採用した 32 ビットマイクロプロセッサである。5 段の命令パイプライン処理により、ほとんどの命令を 1 プロセッササイクルで実行する。用意されている命令は基本的なアドレッシングモードと演算命令だけであり、複雑な命令はいくつかの命令を組み合わせることで実現する。仮想記憶機構、例外処理機構は、プロセッサ内に内蔵された CP0 と呼ばれるコプロセッサでサポートする。また、命令用とデータ用に分割されたダイレクトマップ方式の 1 次キャッシュをサポートする。

R3000 などの RISC アーキテクチャのプロセッサは、命令のパイプライン処理の効率を高め、処理性能の向上をはかっている。ISIS は正確な性能評価を行うことを目的としているので、R3000 プロセッサを正確にシミュレートするためには命令パイプライン処理は無視できない。そこで、R3000 プロセッサシミュレータでは R3000 の 5 段命令パイプラインやライトバッファ、1 次キャッシュ等の機能をクロックレベルで正確にシミュレートするよう実装がなされている。プロセッサシミュレータの内部構造を図 2.10 に示す。

R3010 は、R3000 プロセッサ用の浮動小数点演算コプロセッサである。R3000 プロセッサに接続することで、単精度および倍精度の浮動小数点演算命令を実行する。メモリアクセス、キャッシュアクセス、整数レジスタと浮動小数点レジスタ間の転送命令など、コプロセッサ外部と関係する処理の多くは R3000 プロセッサが行う。コプロセッサに入力された命令は内部の 6 段命令パ

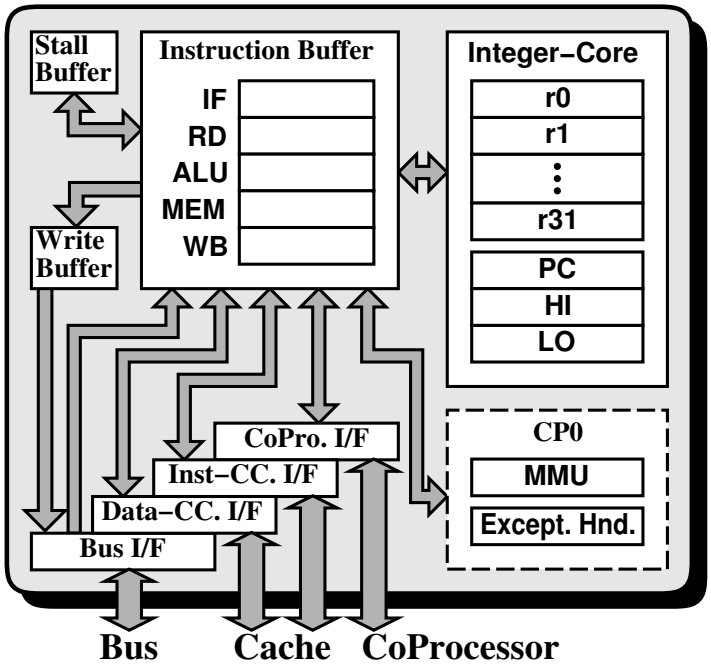


図 2.10: R3000 プロセッサシミュレータの内部構造

イプラインによって処理される。また、浮動小数点演算命令はその演算の種類によって演算時間が大きく異なるため、コプロセッサ内の演算資源が競合しない限り命令はオーバラップして実行される。

R3000 プロセッサ用のコプロセッサである R3010 浮動小数点演算コプロセッサは、R3000 プロセッサシミュレータと同様に、クロック単位で R3010 コプロセッサの 6 段命令パイプラインをシミュレートする。コプロセッサ内での演算資源の競合、レジスタ値のフォワーディング等も正確にシミュレートされる。メモリアクセス等は R3000 プロセッサシミュレータに処理を委ねている。

バス

バスはユニット間の通信を仲介する機関であり、プロセッサやメモリ等、多くのユニットはバスを通じて通信を行う。

バスシミュレータには、ポート間に構成される通信路をそのままバスとして使用する。バスがユニットとして実装されると、バスを経由して通信を行う際に必ず 1 クロックの遅延を余儀なくされるため、バスはユニットとして実装しない。図 2.11 にバスの構成方法を示す。バス上に存在するデータを表すパケットクラスとして、`packet_base` の派生クラスである `bus_packet_base` クラスおよび `bus_packet` を使用する。また、ユニット間の結合には、`port_base` クラスの派生クラスである `bus_port_base` クラスおよび `bus_port` を使用する。

`bus_packet_base` は `packet_base` クラスの派生クラスで、計算機内のバスのアドレス線、データ線、コントロール線をカプセル化したパケットの抽象基底クラスである。アドレスの問い合わせ、データの問い合わせ機能を持つ。

`bus_packet` は `bus_packet_base` クラスの派生関数で、プロセッシングエレメント内のローカ

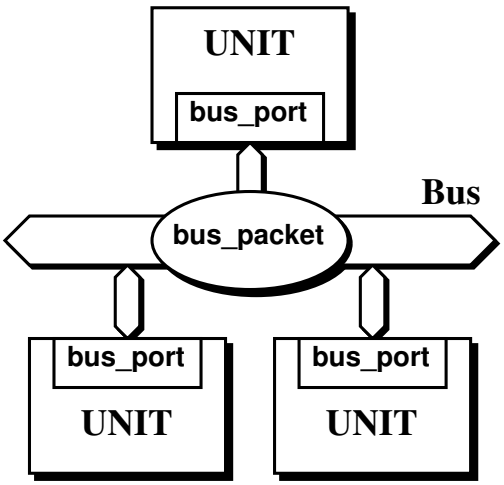


図 2.11: バスの構成方法

ルバス等、特に特殊な機能を持たないバス上の情報を表す。

メモリ

メモリは、実際の記憶装置としてのメモリを表す memory クラスと、そのメモリを制御するメモリコントローラを表す memory_control_unit クラスの 2 つのクラスで実装されている。

memory クラスは並列計算機の主記憶装置を表現するクラスである。このクラスは内部で動的にページ単位の記憶領域管理を行っており、要求されたサイズのメモリをすぐには確保しない。値が書き込まれた時点でページ毎に実体が確保される。従って莫大な量の記憶領域を要求しても、実際に巨大な領域に書き込みを行わない限りホストマシンの資源が不足することはない。この機能により、ホストマシンよりも巨大な資源を持つ並列計算機を容易にシミュレートすることができる。

memory_control_unit クラスは記憶装置を制御するためのコントローラを表すクラスで、synchronous_unit クラスの派生クラスである。バスとメモリを接続することで動作する。単一ワード転送、バースト転送をサポートしている。また、リードアクセス時間、ライトアクセス時間を個別に設定できる。

ファイル入出力機構

ファイル入出力の方が演算よりもはるかに実行時間を要するため、並列計算機の演算性能を評価する際にはファイル入出力は行わないようにする場合が多い。演算性能を評価するようなプログラムでファイル入出力を行うのは、実行開始時にデータを入力する部分と実行終了後に結果を出力する部分である。そこで、 本ライブラリではファイル入出力については正確なシミュレーションを行わないことにし、なるべく簡単な方法でファイル入出力をサポートする。ユーザが正確なファイル入出力をシミュレートしたい場合は、ハードディスクユニットやオペレーティングシステムを実装する必要がある。

file_io_unit クラスは、シミュレータ上で実行されるソフトウェアに対してファイル入出力

機能を提供するユニットである。このユニット 1 つで完全なファイル入出力機能をサポートする。シミュレータ上のソフトウェアからのファイルに対するアクセスは、ホストマシン上のファイルに対する操作に変換される。

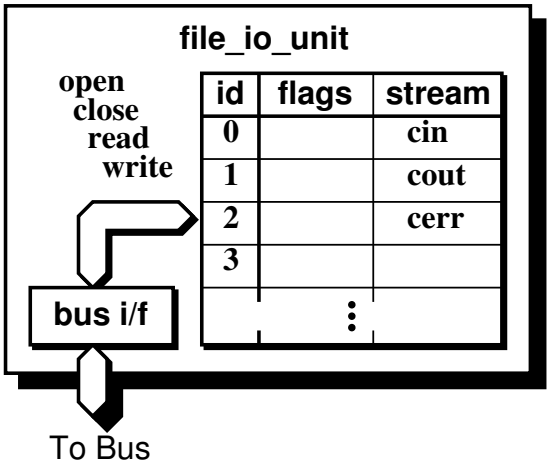


図 2.12: file_io_unit クラスの内部構造

図 2.12 に file_io_unit クラスの内部構造を示す。このクラスはシミュレートされるソフトウェア上でオープンされているファイルのファイル記述子番号に対応するストリームを保持している。ソフトウェア側からファイルに対する操作が発行されると、ユニットは指定されたファイル記述子に対応するストリームに対して操作を行い、結果をソフトウェアに返す。例えばファイルオープン要求が発行されると、新たにファイルストリームをオープンし、新しいファイル記述子を割り当て、得られたファイル記述子番号を返す。

また、ソフトウェアからのアクセスはメモリと同様の方法でアクセス可能なハードウェアレジスタを通じて行われる。シミュレータ上のソフトウェアはこのハードウェアレジスタをアドレス空間内に配置し、そのアドレスにアクセスすることでファイル入出力を行う。ただし、ハードウェアレジスタを操作するプログラムを直接記述するのはユーザにとって大きな負担となるため、シミュレータ上のソフトウェアに対してハードウェアレジスタを操作する関数群が提供されている。

タイマ

timer_unit クラスは、シミュレータ上で実行されるソフトウェアからシミュレータ上の時刻を読み取るために用意されている。クラス内には現在の時刻を示す変数があり、クロック入力毎に自動的に増加していく。このクラスも file_io_unit と同様にハードウェアレジスタを通じてソフトウェアからアクセスされる。また、タイマの値はシミュレータ内からも直接参照できるため、シミュレータとシミュレータ上のソフトウェアで同じ時刻の値を共有することができる。

2.3.3.6 ISIS の欠点

近年マルチプロセッサシステムにおいて主流となっているプロセッサは、out-of-order 実行、複数命令同時発行、分岐予測などをサポートした高性能なプロセッサである。しかし、ISIS では、

プロセッサとしては、MIPS 社の R3000 プロセッサから派生した IDT 社の R3081 をモデルにした単命令発行、in-order 実行を行う比較的簡素なプロセッサのみを機能ユニットとして提供している。そのため、ISIS において高性能なプロセッサを用いたシステムを想定してシミュレータを構成するには、新たにプロセッサシミュレータを実装する必要がある。

2.4 本研究の目的

近年、オンチップマルチプロセッサが様々な用途で用いられるようになってきており、様々な構成の並列計算機が提案されている。それに伴い、開発過程において、シミュレーションによる事前検証や性能評価がとても重要になってきている。

そして現在、シミュレータへ求められることとして、以下のことが挙げられる。

- マルチプロセッサシステムのシミュレーションが可能
  - 複数のプロセッサからのメモリアクセス競合を正確に再現可能
  - ネットワークレイテンシがシステム性能に与える影響を測定可能
- 近年主流となっているような高性能なプロセッサを搭載したシステムのシミュレーションを行うことができる
- あらゆる構成のシステムを構築できる高い汎用性を持っている

また、学術目的で使用するユーザにとっては、無償で提供されていることが望ましい。

Stanford 大学の SimOS は、特定のアーキテクチャを持つ並列計算機の性能評価を目的として開発されたため、あらゆる構成のシステムを容易に構築できるような高い汎用性は持ち合わせていない。また、シミュレータの実行に特定の動作環境を必要とするという欠点も持つ。Wisconsin 大学の SimpleScalar は、シングルプロセッサシステムやプロセッサアーキテクチャの検証に幅広く利用されているが、マルチプロセッサシステムのシミュレーションに対応していない。本研究室の ISIS は、汎用的な部品をライブラリとして提供しているためシミュレータの実装が比較的容易であり、細部まで考慮した記述を行うため動作が正確であるという特徴を持つが、近年主流となっている高性能なプロセッサを提供していない。

このように、近年の高性能なプロセッサを詳細にモデルし、且つ、あらゆる構成のマルチプロセッサシステムを簡単に構築できるようなシミュレータが存在しないという問題点があった。そこで本研究では、SimpleScalar が提供する詳細で高性能なプロセッサをマルチプロセッサシステムシミュレーションへ対応させ、ISIS によって記述されたシステムと接続できるようにする。こうすることにより、高性能なプロセッサを搭載できるという SimpleScalar の利点と、あらゆる構成のシステムを構築できるという ISIS の利点を持ち合わせたシミュレータを提供できることになる。

第 3 章

設計と実装

本研究では、SimpleScalar が提供する詳細で高性能なプロセッサのシミュレータをマルチプロセッサシステムシミュレーションへ対応させ、ISIS によって記述されたシステムと接続できるようにした。

3.1 sim-outorder と ISIS で実装されたシステムとの接続

sim-outorder が発行するメモリアクセス要求を ISIS で実装されたシステムで処理できるようにするため、図 3.1 のように sim-outorder に ISIS で用意されているポートを付加した。このポートを介して ISIS で実装されたシステムと互換性のあるパケットを送受信することによって、情報のやり取りができるようにした。

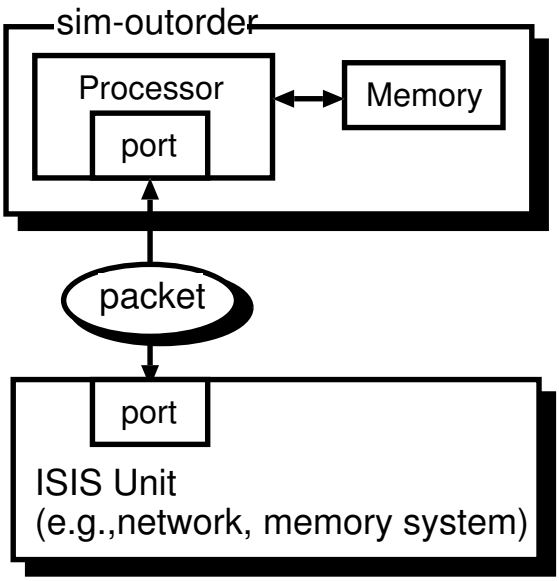


図 3.1: sim-outorder と ISIS で実装されたシステムとの接続

3.2 SimpleScalar をマルチプロセッサシステムシミュレーションへ対応

3.2.1 クラス化

C 言語で実装されている SimpleScalar のソースコードを ANSI 準拠の C++互換に書き直した。また、マルチプロセッサシステムではプロセッサを構成するモジュールがプロセッサ数分必要になる。そのため、sim-outorder の実行に関わるすべてのモジュールをクラス化した。各モジュールをクラス化することにより、インスタンスを生成するだけで、プロセッサ数分のモジュールを用意できるようになった。

クラス化では、各モジュールが独自に使用する関数は private で宣言し、上位モジュールによって使用される関数は public で宣言した。次の例では、cache_access() は、上位モジュールであるプロセッサクラスがキャッシュアクセスするときに呼び出す関数であるため public で宣言し、cache_bcopy() は、同じ cache_simple クラス内の cache_access() によってのみ呼び出される関数であるため private で宣言した。

<例> cache.h

```
class cache_simple
{
    private:
        /* cache_access() によって呼び出される関数 */
        void
        cache_bcopy(enum mem_cmd, struct cache_blk_t *,
                    md_addr_t, byte_t *, int);
        …
        …

    public:
        /* プロセッサに呼び出される関数 */
        unsigned int
        cache_access(struct cache_t *cp, enum mem_cmd cmd,
                     md_addr_t addr, void *vp, int nbytes, tick_t now,
                     byte_t **udata, md_addr_t *repl_addr);
        …
        …
};
```

3.2.2 アドレス空間の変更

マルチプロセッサシステムにおいてはプロセッサ外部にある共有メモリ領域へのアクセスが考えられるため、アドレス空間の割当を図 3.2 のように変更し、共有メモリ領域のアドレスに関しては address.h に定義した。

オリジナルの SimpleScalar から変更したメモリ領域を次にまとめる。

- 0x10000000〜0x7fffffff

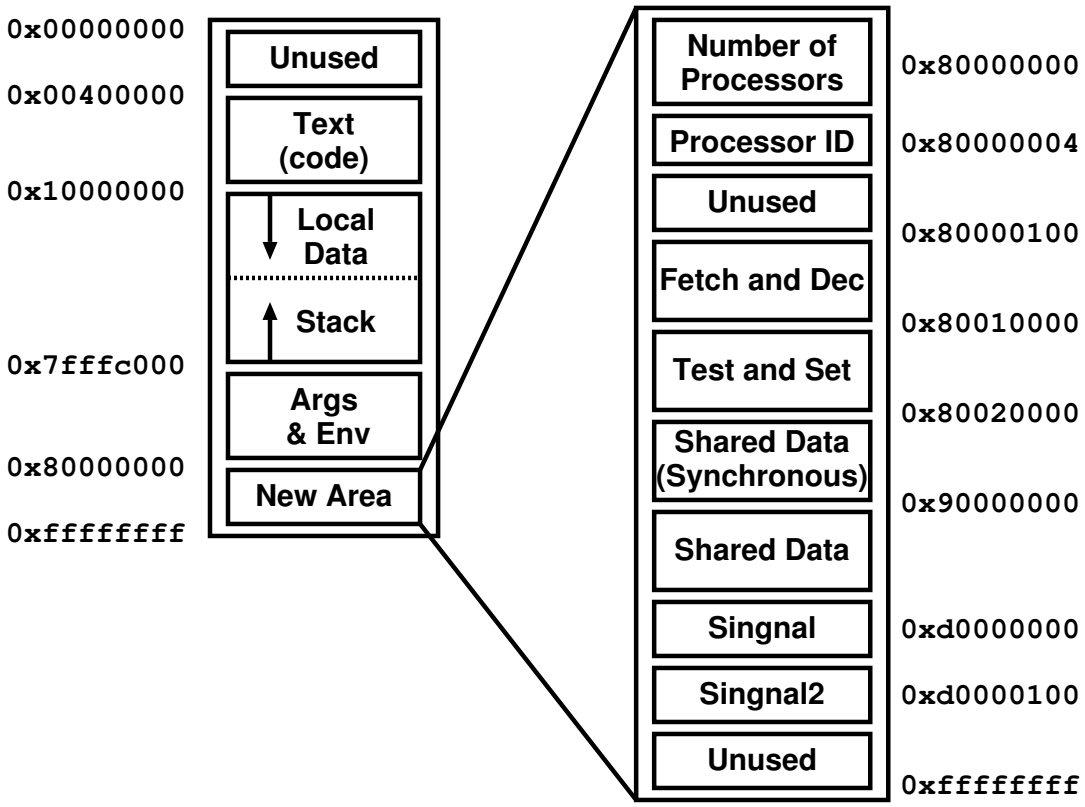


図 3.2: アドレス空間

  - 各プロセッサが独自に利用するローカルデータ

- `0x80000000`
  - 各プロセッサが独自に利用し、シミュレータ上のプロセッサ数を格納する
- `0x80000004`
  - 各プロセッサが独自に利用し、プロセッサ ID を格納する
- `0x80000100`〜`0x8000fffc`
  - Fetch and Dec によって使用される領域
- `0x80010000`〜`0x8001fffc`
  - Test and Set によって使用される領域
- `0x80020000`〜`0x8ffffffc`
  - 同期のために複数のプロセッサ間で共有される共有データ領域
- `0x90000000`〜`0xcffffffc`
  - 複数のプロセッサ間で共有される共有データ領域

- 0xd0000000、0xd0000100
  - シミュレータ上で実行するプログラム中に、0xd0000000 番地と 0xd00000100 番地へアクセスするコードを入れると、その間の統計データが取られる (3.4.1 項参照)

Fetch and Dec と Test and Set を行う 0x80000100〜0x8001fffc へのアクセスは、通常のメモリアクセスと異なるため投機的な発行を許していない。

3.2.3 sim-outorder の動作修正

2.3.2.4 項で示したように、オリジナルの sim-outorder は out-of-order 実行をサポートしているが、実際には命令は ruu_dispatch() において in-order で実行される。しかし、その後のステージにおいて、レジスタやメモリアクセスアドレスの依存関係、Functional Unit 数やメモリアクセスポート数などを考慮した処理が行われることによって、命令はあたかも out-of-order で実行されたかのように振舞われる。ruu_dispatch() において命令が実行される際、メモリからのデータの読み出しやメモリへのデータの書き込みもレイテンシなしで in-order に行われる。

オリジナルの sim-outorder が対象としているシングルプロセッサシステムのシミュレーションを行う場合には、このような処理方法で問題が起こることはない。しかし、マルチプロセッサシステムにおいては以下のことを考慮する必要があるため、問題が生じる。

1. バスなどの共有ネットワークを介して共有メモリがアクセスされるマルチプロセッサシステムにおいては、プロセッサがメモリを直接アクセスすることやレイテンシなしでアクセスすることはない

2. マルチプロセッサシステムにおいては、共有データが各プロセッシングノードのキャッシュに格納されることが考えられ、キャッシュ間のコンシステンシ制御やキャッシュラインの一貫性の保持を管理する必要がある

3. メモリがインタリーブされている場合など、システムによってはプロセッサから発行されたアドレスとは別のアドレスにデータが割り当てられる場合がある

もし、ruu_dispatch() における瞬時のメモリアクセスを維持するならば、共有ネットワークでのメモリアクセスの競合やキャッシュ間のコンシステンシ制御に関して正確なシミュレーションを行うことは非常に困難になる。また、3 に関するアドレス変換は sim-outorder 内で行わなければならない。メモリアクセス方式がシステムに依存することは大いに考えられ、その度に sim-outorder 内のアドレス変換に関わるコードを修正しなければならないのは、シミュレータとしての可搬性に欠ける。

このような問題に対応するため、プロセッサからのメモリアクセス要求が正規のタイミングで共有ネットワークに発行されるように動作修正する。メモリアクセス要求が正規のタイミングで発行されることによって、共有ネットワークでのメモリアクセスの競合や共有データキャッシュ間のコンシステンシ制御を正確に且つ容易に再現することができる。また、要求がプロセッサ外部に発行されることにより、アドレス変換がプロセッサ外部で行えるため、sim-outorder のコードを修正せずに様々なメモリアクセス方式に対応できるようになる。

実行が終了するまで繰り返す for 文を、以下のように変更した。

```
for (;;) {
    ruu_commit();
    ruu_writeback();
    ruu_issue();
    ruu_dispatch();
    ruu_fetch();
    get_data();
}
```

各 Stage での動作は以下の通りである。

- `void get_data(void)`
  - 入力ポート (外部システムから送られてくるパケットを受信するためのポート) に共有メモリからの読み出し要求に対する返信が転送されてきていたら、そのデータを受信する
- `void ruu_fetch(void)`
  - 2.3.2.4 項で紹介した sim-outorder の Fetch Stage
  - 命令幅分の命令を I-Cache または Memory から読み出し、IFQ に送る
- `void ruu_dispatch(void)`
  - 2.3.2.4 項で紹介した sim-outorder の Dispatch Stage
  - IFQ に送られた命令を in-order で取り出し、その命令のデコードを行う
  - すべての命令を RUU (Register Update Unit) と reservation station に送る
- `void ruu_issue(void)`
  - 2.3.2.4 項で紹介した sim-outorder の Scheduler Stage
  - reservation station に入っている命令のうち、
    1. Functional Unit を利用する命令は、以下の条件が揃った時点で Functional Unit を占有し、execution buffer に投入する
       - レジスタまたは reorder buffer から、値の読み出しが完了
       - Functional Unit が利用可能
    2. ローカルメモリ (`0x10000000`〜`0x7fffffff`) への Load 命令は、以下の条件が揃った時点で、ローカルメモリアクセスポートを占有し、execution buffer に投入する
       - メモリアクセスするアドレスが決定
       - 他のメモリアクセス命令との依存関係なし
       - ローカルメモリアクセスポートが利用可能
    3. 共有メモリ (`0x80000100`〜`0xffffffff`) への Load 命令は、以下の条件が揃った時点で、外部出力ポート (外部システムに要求パケットを発行するためのポート) に共有メモリへの読み出し要求を発行し、execution buffer に投入する

    - メモリアクセスするアドレスが決定
    - 他のメモリアクセス命令との依存関係なし
    - 外部出力ポートが利用可能

- `void ruu_writeback(void)`
  - 2.3.2.4 項で紹介した sim-outorder の Execute Stage と Writeback Stage
  - execution buffer に入った命令のうち、
    1. Functional Unit を利用する命令は、execution buffer に投入されてからの経過時間が Functional Unit を占有する時間に達したら、Functional Unit の解放、命令の実行を行い、実行結果を reorder buffer に送る
    2. ローカルメモリへの Load 命令は、execution buffer に投入されてからの経過時間がメモリアクセスレイテンシに達したら、ローカルメモリからデータを読み出し、その値を reorder buffer に送る
    3. 共有メモリへの Load 命令は、データが受信済みだったら、その値を reorder buffer に送る
  - 分岐予測が失敗しているか否かを判断し、分岐予測が失敗していた場合には、元の正当な状態に戻す

- `void ruu_commit(void)`
  - 2.3.2.4 項で紹介した sim-outorder の Commit Stage
  - RUU の先頭から命令を取り出し、その命令の実行結果が reorder buffer に投入されていたら、その値を用いてレジスタの更新を行い、命令の終了処理を行う。その命令が Store 命令の場合、
    1. ローカルメモリへの Store 命令ならば、ローカルメモリに値を書き込む
    2. 共有メモリへの Store 命令ならば、外部出力ポートに共有メモリへの書き込み要求とデータを発行する

3.2.4 命令定義の変更

各 Stage において上記のような動作を実現するために、命令が定義されている `machine.def` を修正した。オリジナルの sim-outorder では、レジスタの読み出し、メモリアクセス、計算、レジスタへの書き込みなどを 1 つの`#define` 文を呼び出すことにより同時に行っている。しかし、修正したプログラムでは、値の読み出し、計算、レジスタの更新などは、別々のタイミング、または別々の Stage で実行される。

次に、修正した命令定義として 2 つの例を挙げる。

3.2.4.1 `ADD`

2 つの値を加算する命令である `ADD` は次のように定義され、値の読み出し、計算、レジスタの更新が同時に行われていた。

```
#define ADD_IMPL                                \
  {                                             \
      if (OVER(GPR(RS), GPR(RT)))               \
          DECLARE_FAULT(md_fault_overflow);     \
                                                \
      SET_GPR(RD, GPR(RS) + GPR(RT));           \
  }
```

しかし、修正したプログラムでは、値の読み出し、計算、レジスタの更新は別々のタイミングで行われる。しかも、値の読み出し元はレジスタに限らず、reorder buffer の場合もある。そのため、ADD は、レジスタまたは reorder buffer から値を読み出し計算する部分、レジスタを更新する部分の 2 つの部分に分割し、新たな関数として定義した。

ruu_writeback() での動作

```
/* レジスタまたは reorder buffer から値を読み出し計算する部分 */
void ADD_impl()
  {
      sword_t temp, temp2;

      /* reorder buffer または GPR(RS)、GPR(RT) から値を読み出す */
      choose_operand(in1_flag, &temp, RS, 1);
      choose_operand(in2_flag, &temp2, RT, 2);

      if (OVER(temp, temp2)) assert(0);

      /* 2 つの値を加算 */
      result1 = temp + temp2;
      result1_flag = true;
  }
```

ruu_commit() での動作

```
/* レジスタを更新する部分 */
void ADD_reg_impl()
  {
      /* 加算結果でレジスタを更新 */
      set_gpr(RD, result1);
  }
```

3.2.4.2 LW

メモリから 4byte のデータを読み出す命令である LW は、次のように定義され、値の読み出し、メモリアクセスアドレスの計算、メモリアクセス、レジスタの更新が同時に行われていた。

```
#define LW_IMPL                                          \
  {                                                      \
      word_t _result;                                    \
      enum md_fault_type _fault;                         \
                                                         \
      _result = READ_WORD(GPR(BS) + OFS, _fault);        \
      SET_GPR(RT, _result);                              \
  }
```

しかし、修正したプログラムにおいて、これらは別々のタイミング、または別々の Stage で実行される。そのため、レジスタまたは reorder buffer から値を読み出し、メモリアクセスアドレスを計算する部分、共有メモリに値の読み出し要求を発行する部分、メモリから読み出された値を確保する部分、レジスタを更新する部分の 4 つの部分に分割し、新たな関数として定義した。

ruu_issue() での動作

```
/* レジスタまたは reorder buffer から値を読み出し、アドレスを計算する部分 */
void LW_addr_impl(void)
 {
     sword_t temp;

     /* reorder buffer または GPR(RS)、GPR(RT) から値を読み出す */
     choose_operand(in2_flag, &temp, BS, 2);

     /* メモリアクセスアドレスの計算 */
     addr = (temp + OFS);
     /*メモリアクセスする際の byte 幅の指定 */
     addr_area = 4;

     /* 共有メモリ領域のアドレスならば */
     if ( addr > MACHINE_ADDRESS_BOTTOM )
         load_flag = true;
     use_lsq_flag = true;
  }
```

ruu_issue() での動作

```
/* メモリアクセス要求を発行する部分 */
void LW_set_req_impl(void)
  {
      bool flag = false;

      /* 使用されていない外部出力ポートに要求を発行 */
      for ( int i = 0; i < outside_port_num; i++ ) {
          if ( !outgoing_port[i].have_packet()
               || outgoing_port[i].have_packet()
                  && outgoing_port[i].inst_id() == -1 ){
              outgoing_port[i].set_read_detail(access_adr, req_label,
                                               addr_area, puid);
              flag = true;
              break;
          }
      }
      if ( !flag )
          exec_stop_flag = true;
  }
```

ruu_writeback() での動作

```
/* メモリから読み出された値を確保する部分 */
void LW_impl(void)
  {
      /* 共有メモリへのアクセスならば */
      if ( load_flag ) {
          /* 共有メモリから読み出された値を確保 */
          result1 = load_data;
      /* ローカルメモリへのアクセスならば */
      } else {
          enum md_fault_type _fault;
          /* 前にある Store 命令から値が流されてきたら */
          if ( lsq_data_flag )
              /* 前にある Store 命令から流されてきた値を確保 */
              result1 = lsq_data;
          else
              /* ローカルメモリから値を読み出し */
              result1 = read_word(access_adr, _fault);
      }
      result1_flag = true;
  }
```

ruu_commit() での動作

```
/* レジスタを更新する部分 */
void LW_reg_impl(void)
  {
      /* 確保した値でレジスタを更新 */
      set_gpr(RT, result1);
  }
```

このように、各命令の動作は次のように分割してそれぞれ定義し、関数名を統一した。

- ＊_impl – 値の読み出しや計算結果の生成、読み出した値を確保する部分
- ＊_addr_impl – メモリアクセスアドレスを計算する部分
- ＊_set_req_impl – 共有メモリに読み出し要求を発行する部分
- ＊_reg_impl – レジスタを更新する部分

のように関数名を統一した。

3.3 実装したクラスの解説

以下に ISIS-SimpleScalar を実装するために作成したクラスについての説明を行う。クラス間の関係は図 3.3 のようになっている。

3.3.1 isim_system クラス (isim_system{.h,.cc})

シミュレータ全体のシステムを 1 つにまとめたクラスである。図 3.3 のように、arg クラス、option_simple クラス、stats_simple クラス、memory_simple クラス、isim_processor クラスを内包する。また、プロセッサ外部のネットワークや共有メモリも内包し、これらの機能ユニットの接続やクロックの分配を行う。

3.3.2 arg クラス (arg{.h,.cc})

コマンドラインオプションとして指定されたプロセッサ数、プロセッサが外部システムと通信するためのポート数を読み取るクラスである。指定しない場合のプロセッサ数は 1、ポート数は 2(入力ポート 2、出力ポート 2) である。

-p [number]
　プロセッサ数を指定する

-o [number]
　ポート数を指定する

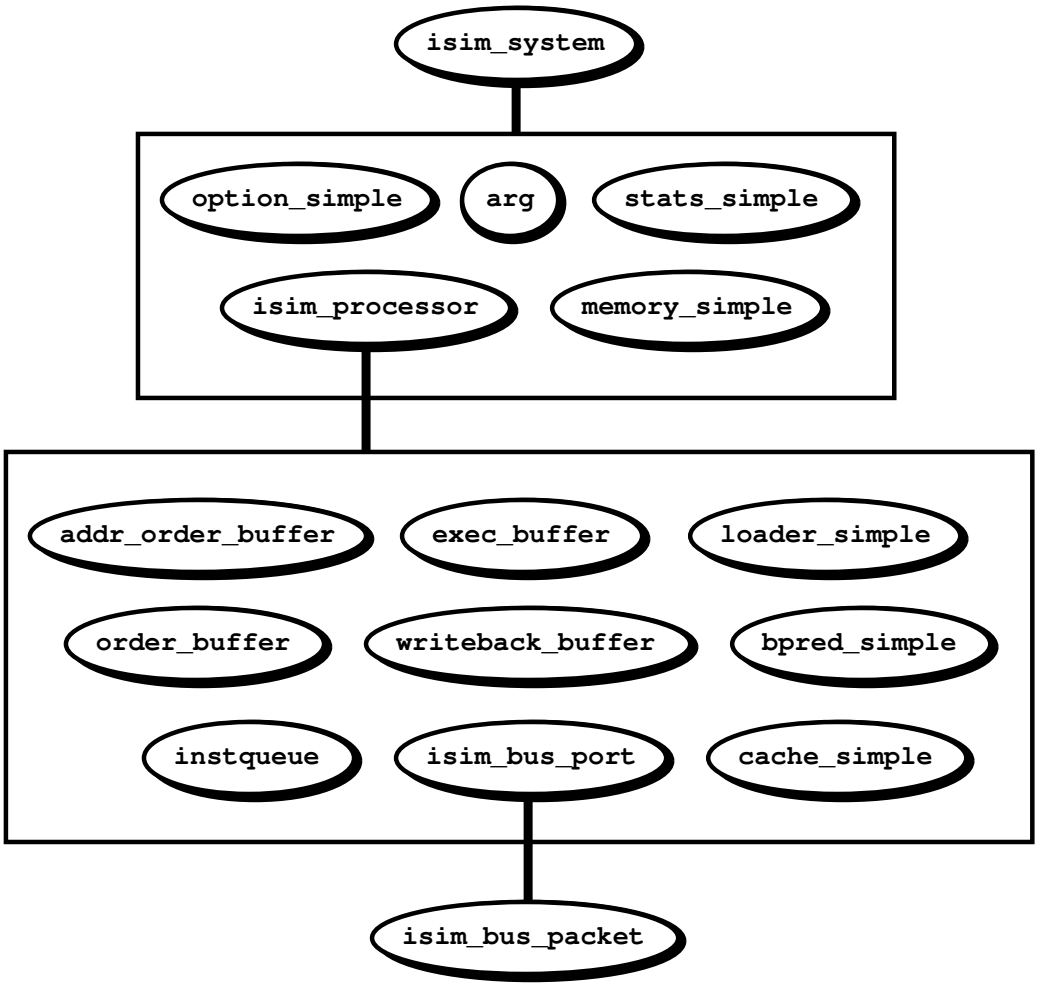


図 3.3: ISIS-SimpleScalar のクラス

3.3.3 option_simple クラス (options{.h,.cc})

SimpleScalar が提供する options.c をクラス化した。キャッシュ、branch predictor など、システムに関するさまざまなコマンドラインオプションを解釈する。

3.3.4 stats_simple クラス (stats{.h,.cc}, stats_eval.cc)

SimpleScalar が提供する stats.c をクラス化した。シミュレーション中にプロセッサ毎に取られる、実行クロック数やローカルキャッシュのヒット率などの統計データの指定、初期化、出力などを行う。

3.3.5 memory_simple クラス (memory_s{.h,.cc})

SimpleScalar が提供する memory.c をクラス化した。ISIS-SimpleScalar では、命令やローカルデータを格納するローカルメモリを表す。メモリ領域は `0x00000000`〜`0x7fffffff` である。

3.3.6 isim_processor クラス (isim_processor{.h,.cc}, instruction.cc, inst_simple.cc)

3.2.3 項で説明した、動作修正した sim-outorder を表すクラスである。addr_order_buffer クラス、order_buffer クラス、instqueue クラス、exec_buffer クラス、writeback_buffer クラス、loader_simple クラス、bpred_simple クラス、cache_simple クラス、isim_bus_port クラスを内包する。

パイプライン内の各 Stage の動作は isim_processor.cc に記述した。3.2.4 項で説明した通り、sim-outorder の動作修正に伴い命令定義を変更した。各命令は isim_processor クラスのメンバ関数として、instruction.cc 内に記述した。また、レジスタの更新やプロセッサ外部への要求発行など、複数の命令で行われる処理をメンバ関数として inst_simple.cc にまとめて定義し、各命令が呼び出せるようにした。

また、共有メモリ領域へのメモリアクセスが発生した場合には、isim_bus_port クラス型の外部出力ポートに isim_bus_packet クラス型のメモリアクセス要求を発行する。その要求を受信した外部システムは、要求に応じた処理を行う必要がある。外部システムがすべき具体的な処理内容は、付録 B に記した。

3.3.7 addr_order_buffer クラス (addr_order_buffer{.h,.cc})

Load 命令および Store 命令のメモリアクセスアドレスを保持し、メモリアクセスの順序管理を行うクラスである。メモリアクセスの順序管理の方法は次の通りである。

- 解決していない Store 命令が前にある場合には、それに続く Load 命令は、準備ができていたとしても stall する
- Store 命令がメモリに書き込む値が決定したら、その値を保持する
  - stall している Load 命令のメモリアクセスアドレスが、前にある Store 命令と同じである場合には、Store 命令がメモリに書き込むべき値を Load 命令に流す

Load、Store 命令の順序は、次の構造体のリストで管理している。

```
struct ADList {
    INST_SEQ_TYPE seq;              //命令 ID (命令を特定する連続番号)
    md_addr_t addr, addr2;          //メモリアクセスアドレス
    int addr_area;                  //メモリアクセスする際の byte 幅
    bool l_or_s;                    //Load 命令 or Store 命令
    sword_t lsq_data, lsq_data2;    //Store 命令がメモリに書き込む値
    struct ADList *next, *prev;
};
```

また、メンバ関数として、

- void updatelist(INST_SEQ_TYPE seq, md_addr_t addr, md_addr_t addr2, int addr_area)
  - メモリアクセスアドレスが決定したところで、該当する命令の addr、addr2、addr_area を更新

- void
  insert_store_data(INST_SEQ_TYPE seq, sword_t lsq_data, sword_t lsq_data2)
  - Store 命令がメモリに書き込む値が決まったところで、該当する命令の lsq_data、lsq_data2 を更新
- bool
  checklist(INST_SEQ_TYPE, md_addr_t addr, md_addr_t addr2, int addr_area,
  bool l_or_s, sword_t* lsq_data, sword_t* lsq_data2, bool* lsq_data_flag,
  bool use_lsq_flag)
  - Load 命令に関するメモリアクセスアドレスの依存関係を調べ、依存関係がある場合には false、ない場合には true を返す

  <例外> メモリアクセスアドレスに依存関係がある場合でも、前にある Store 命令とアクセスするアドレスが同じで、且つその Store 命令のメモリに書き込む値が決まっている場合には、その値 (lsq_data、lsq_data2) を Load 命令に流し、true を返す

等を定義した。

3.3.8 order_buffer クラス (order_buffer{.h,.cc})

各命令の命令 ID と使用するレジスタ番号を保持し、レジスタ間の依存関係によって実行の順序を管理するクラスである。

命令の実行順序は、次の構造体のリストで管理している。

```
struct ORDERList {
    INST_SEQ_TYPE seq;   //命令 ID (命令を特定する連続番号)
    int reg;             //使用するレジスタ番号
    bool rw;  //レジスタから値を読み出すのか、それともレジスタに値を書き込むのか
    struct ORDERList *next, *prev;
};
```

また、メンバ関数として、

- int
  check_order(INST_SEQ_TYPE seq, bool addr_flag, int in1, int in2, int in3,
  int in4, int out1, int out2, int out3, int *seq, int *opr,
  int *d_rw, int *my_rw);
  - 引数として、命令が使用するレジスタ番号を受け取り、レジスタ間の依存関係を調べる
  - 依存関係がない場合には負の数を返す
  - 依存関係がある場合には依存関係の数を返し、依存関係のある命令の命令 ID とレジスタ番号などを渡す
    - 依存関係の数を受け取った各命令は reorder buffer を調べる

* reorder buffer から依存関係の数分だけ値を受け取ることができれば、reorder buffer の値を用いて命令を実行する

等を定義した。

3.3.9 instqueue クラス (instqueue{.h,.cc})

3.2.3 項で説明した、RUU および reservation station を表すクラスである。各命令に対するあらゆる情報を保持し、in-order で並べられる。

各命令は、次の構造体のリストで管理している。

```
struct INSTList {
    md_inst_t inst;                        //命令の種類
    INST_SEQ_TYPE seq;                     //命令 ID (命令を特定する連続番号)
    int in1, in2, in3, in4, out1, out2, out3;  //使用するレジスタ番号
    struct res_template *fu;               //使用する Functional Unit
    INST_SEQ_TYPE branch_seq;              //従うべき分岐命令の命令 ID
    int branch_result;                     //従うべき分岐命令の分岐結果
    md_addr_t adr, adr2;                   //メモリアクセスアドレス
    word_t load_data, load_data2;          //共有メモリから読み出された値
    sword_t lsq_data, lsq_data2;           //Store 命令から流されてきた値
    int exec_time;                         //実行する時刻
    md_addr_t now_PC, next_PC, pred_PC; //その命令のプログラムカウンタなど
      …
      …
    struct INSTList *next;
};
```

また、メンバ関数として、

- `instqueue::INSTList*`
  `return_reservation_station_inst(int time)`
  - issue が済んでいない命令構造体へのポインタを返す
- `instqueue::INSTList*`
  `return_inst(INST_SEQ_TYPE seq)`
  - 引数として与えられた命令 ID を持つ命令構造体へのポインタを返す
- `int`
  `get_ruu_inst(void)`
  - リスト内にある命令の数を返す
- `int`
  `get_lsq_inst(void)`

  - リスト内にあるメモリアクセス命令の数を返す

- void
  set_good_branch(INST_SEQ_TYPE branch_seq)
  - 引数として与えられた命令 ID を持つ分岐命令に従う命令の branch_result を 1(分岐予測成功) にする

- void
  set_bad_branch(INST_SEQ_TYPE branch_seq)
  - 引数として与えられた命令 ID を持つ分岐命令に従う命令の branch_result を 2(分岐予測失敗) にする

等を定義した。

3.3.10 exec_buffer クラス (exec_buffer{.h,.cc})

3.2.3 項で説明した execution buffer を表すクラスである。メンバ変数として instqueue クラスで定義された構造体 INSTList 型の変数へのポインタを保持し、リスト構造で管理した。実行が終了する時刻 (Functional Unit での計算が終了する時刻、ローカルメモリへのアクセスが終了する時刻) 順にリストから削除される (out-of-order)。

3.3.11 writeback_buffer クラス (writeback_buffer{.h,.cc})

Writeback イベントを終了した命令を保持するクラスである。各命令がレジスタを更新する際の値 (計算結果やメモリから読み出した値) を保持し、reorder buffer の役割を担う。命令は命令 ID で昇順に並べられる (in-order)。

各命令は次の構造体のリスト構造で管理している。

```
struct WBList {
    /* instqueue クラスで定義された構造体 INSTList 型の変数へのポインタ */
    instqueue::INSTList *inst;
    int reg1, reg2;          //書き込むを行うレジスタ番号
    /* レジスタを更新する値 */
    int r_i;                 //int 型
    sfloat_t r_f;            //float 型
    dfloat_t r_d;            //double 型
    sword_t r_1, r_2;        //signed int 型
      …
      …
    struct WBList *next;
};
```

メンバ関数として、

- void
  get_element(INST_SEQ_TYPE seq, sword_t* r_1, sword_t* r_2, int* r_i,
  sfloat_t* r_f, dfloat_t* r_d)
  - 引数として与えられた命令 ID を持つ命令の実行結果を渡す
  - 各命令がレジスタを更新する際に呼び出される
- void
  get_data(int reg_num, INST_SEQ_TYPE seq, bool* r_1_flag, sword_t* r_1,
  bool* r_2_flag, sword_t* r_2, bool* r_i_flag, int* r_i,
  bool* r_f_flag, sfloat_t* r_f, bool* r_d_flag, dfloat_t* r_d, int time);
  - 引数として与えられた命令 ID を持つ命令が、引数として与えられたレジスタ番号のレジスタに書き込む値 (その命令の実行結果) を渡す
  - reorder buffer の値を取得したい命令によって呼び出される
- void
  is(INST_SEQ_TYPE seq)
  - 引数として与えられた命令 ID を持つ命令の実行結果が reorder buffer に存在しているかどうかによって、true、false を返す

等を定義した。

3.3.12 loader_simple クラス (loader_s{.h,.cc})

SimpleScalar が提供する loader.c をクラス化した。シミュレータが実行するバイナリを読み込み、0x00400000〜0x0fffffff に格納する。

3.3.13 bpred_simple クラス (bpred{.h,.cc})

SimpleScalar が提供する bpred.c をクラス化した。branch predictor を表すクラスで、branch predictor の初期化、参照、更新を行う。

3.3.14 cache_simple クラス (cache{.h,.cc})

SimpleScalar が提供する cache.c をクラス化した。ISIS-SimpleScalar では 0x00000000〜0x7fffffff の領域の命令やローカルデータを格納するローカルキャッシュを表す。命令やデータは実際には格納しておらず、キャッシュアクセスが発生したときには、キャッシュヒット/ミスが判定され、キャッシュアクセス時間が返される。命令やデータはメモリから読み出される。

3.3.15 isim_bus_port クラス (isim_bus_port{.h,.cc})

isim_processor と ISIS で実装されたプロセッサ外部にあるユニットとの接続に使用されるポートを表すクラスである。図 3.1 の port を表す。複数ポート同士を接続すると、接続されたすべて

のポートで共有される 1 つの経路が確保される。あるポートからパケットを入力すると、接続されたポートすべてから入力したパケットの参照および取得をすることができる。ただし、入力できるパケットは内包する isim_bus_packet クラスで定義されているもののみである。

3.3.16 isim_bus_packet クラス (isim_bus_packet{.h,.cc})

ISIS で実装されたシステムとやり取りされるパケットを表すクラスである。
メンバ変数として、

```
md_addr_t addr;          //メモリアクセスアドレス
int data_size;           //メモリアクセスする際の byte 幅
int inst_id;             //命令 ID
int rw;                  //Read or Write or Reply
int puid;                //送信元プロセッサの ID
/* データ */
word_t data, data2;      //unsigned int 型
half_t data_half;        //unsigned short 型
byte_t data_byte;        //unsigned char 型
```

を定義した。rw は、値によって

- 0 – 共有メモリへの読み出し要求
- 1 – 共有メモリへの書き込み要求
- 2 – 共有メモリからの返信

を表す。また、データは rw の値によって

- 共有メモリへ書き込むデータ
- 共有メモリから読み出されたデータ

のどちらかを表す。

3.4 統計データ

シミュレーション終了後に出力される統計データをプロセッサ数分出力するように変更した。
また、統計データとして、新たに、

- local_read_access
  - ローカルメモリ領域 (0x10000000〜0x7fffffff) への Load 命令の総計
- local_write_access
  - ローカルメモリ領域への Store 命令の総計

- shared_read_access
  - 共有メモリ領域 (0x90000000〜0xffffffff) への Load 命令の総計
- shared_write_access
  - 共有メモリ領域への Store 命令の総計
- sync_read_access
  - 同期のための共有メモリ領域 (0x80000100〜0x8fffffff) への Load 命令の総計
- sync_write_access
  - 同期のための共有メモリ領域への Store 命令の総計
- other_insts
  - 上記以外の命令の総計

を追加した。

3.4.1 特定部分統計データ

0xd0000000 番地へのメモリアクセスがあってから 0xd0000100 番地へのメモリアクセスがあるまでカウントされる特定部分統計データの取得を可能にした。0xd0000000 番地へアクセスするコードと 0xd0000100 番地へアクセスするコードをシミュレータが実行するプログラムに埋め込むことによって、ユーザは任意の部分の統計データを取ることができる。

3.5 コマンドラインオプション

オリジナルの SimpleScalar はシングルプロセッサ対応であるため、コマンドラインオプションが指定されたときには、1 つのプロセッサのパラメータを更新するだけでよかった。しかし、マルチプロセッサシステムでは複数のプロセッサが存在するため、パラメータが指定された場合には、すべてのプロセッサのパラメータを更新するように修正した。

また、新たなオプションとして次を追加した。

- -cache:flush
  - 0xd0000000 番地へのメモリアクセスがあったら、特定部分統計データ (3.4.1 項参照) 取得用のキャッシュに切り替える
  - 0xd0000100 番地へのメモリアクセスがあったら、切り替える前のキャッシュに戻す
  - 特定部分統計データ (3.4.1 項参照) の取得時に限定したキャッシュに関する統計データを取得できる
- -output [number]
  - 統計データの出力方法を [number] によって変える

- 0 – 統計データを出力しない
- 1 – 特定部分統計データのみ出力する
- 2 – 全時間の統計データのみ出力する
- 3 – 全時間の統計データと特定部分統計データの両方を出力する

デフォルト (-output 3) では、統計データの出力は次のようになる。

```
sim: ** simulation statistics **


      ------------ statistics of processor 0 ------------


                  = full-time statistics =

load_local_data             7063560 # total of loads executed(local data)
store_local_data            1987535 # total of stores executed(local data)

                        ..............
                        ..............

mem.ptab_accesses         182758548 # total page table accesses
mem.ptab_miss_rate           0.0000 # first level page table miss rate

                 = particular time statistics =

pts_load_local_data         4536836 # total of loads executed(local data)
pts_store_local_data        1169988 # total of stores executed(local data)

                        ..............
                        ..............

pts_lsq_latency              6.0241 # avg LSQ occupant latency (cycle's)
pts_lsq_full                 0.8094 # fraction of time (cycle's) LSQ was full


      ------------ statistics of processor 1 ------------


                  = full-time statistics =

load_local_data             6394516 # total of loads executed(local data)
```

```
store_local_data            1201412 # total of stores executed(local data)

                          ..............
                          ..............

mem.ptab_accesses         144373176 # total page table accesses
mem.ptab_miss_rate           0.0000 # first level page table miss rate

                  = particular time statistics =

pts_load_local_data         4537379 # total of loads executed(local data)
pts_store_local_data        1169911 # total of stores executed(local data)

                          ..............
                          ..............

pts_lsq_latency              6.0241 # avg LSQ occupant latency (cycle's)
pts_lsq_full                 0.8094 # fraction of time (cycle's) LSQ was full
```

3.6 ライブラリ化

今回実装した ISIS-SimpleScalar を ISIS の一部としてライブラリ化した。今回実装したシステム全体を内包するクラスである isim_system は、内包すべきクラスがユーザ自身の設計に委ねられる。そのため、コードを修正しやすいようにライブラリ化していない。

図 3.4 のように、シミュレータの構築には ISIS-SimpleScalar のライブラリが利用でき、ホストマシンの C コンパイラでコンパイルできる。シミュレータ上で実行するバイナリは、SimpleScalar が提供する GNU Gcc、GNU As、GNU Ld によって生成することができる。

利用方法などについての詳細は、付録 B に記述した。

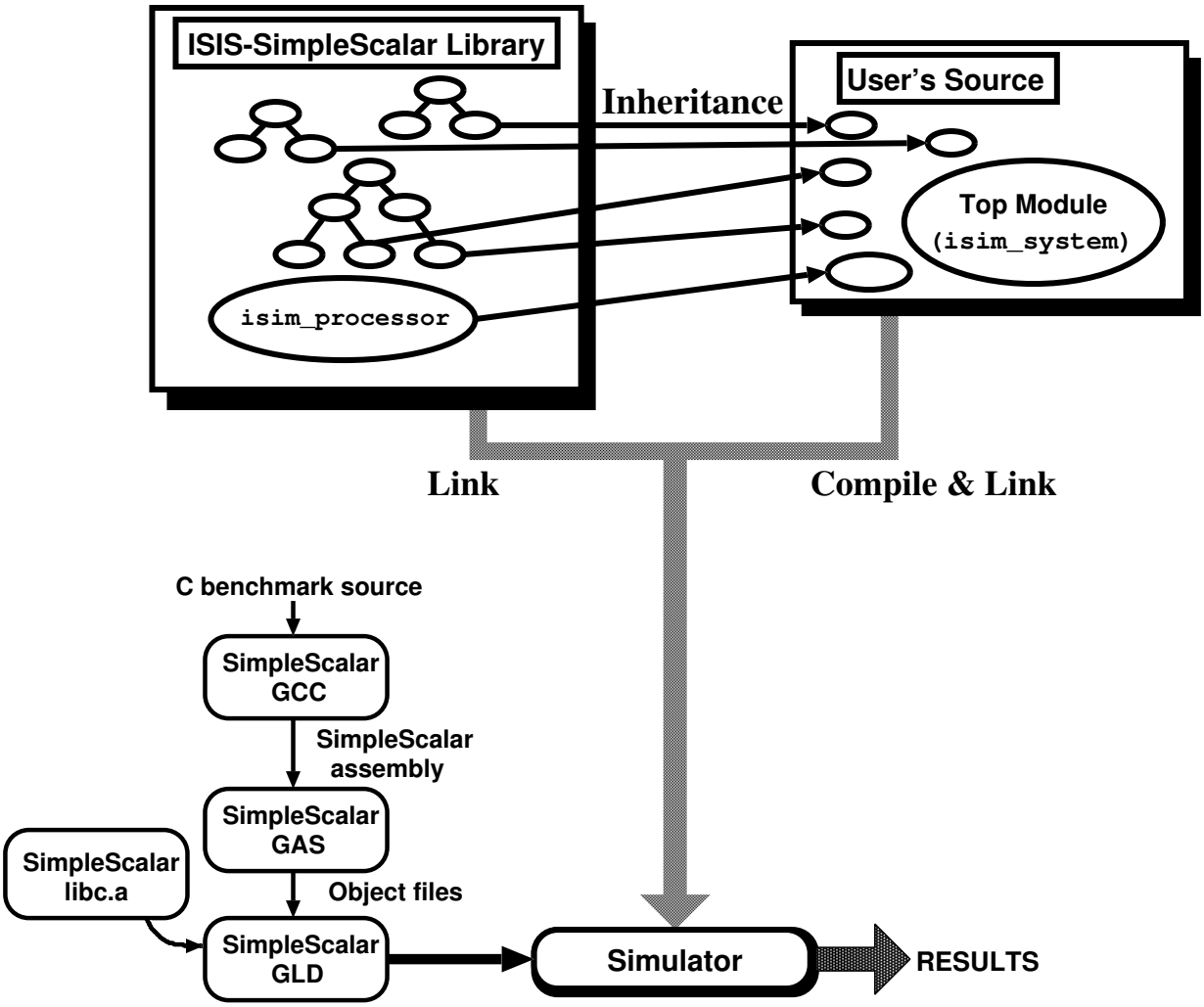


図 3.4: ISIS-SimpleScalar overview

第 4 章

評価

本章では、作成した ISIS-SimpleScalar の評価を行う。

評価には、SimpleScalar ツールセットに添えられていた test-math、test-fmath、test-printf、test-llong、test-lswlr の 5 つのテストプログラムと、SPLASH2 ベンチマーク集から以下の 3 つのアプリケーションを使用した。

表 4.1: 評価に使用したアプリケーション

FFT	高速フーリエ変換	2^{14}
LU	行列の LU 分解	128×128
RADIX	整数の基数ソート	131072 keys

4.1 シングルプロセッサシステムシミュレーション

シングルプロセッサシステムをシミュレーションしたときの評価を行うために、図 4.1 のようなシングルプロセッサシステムのシミュレータを実装し、評価した。

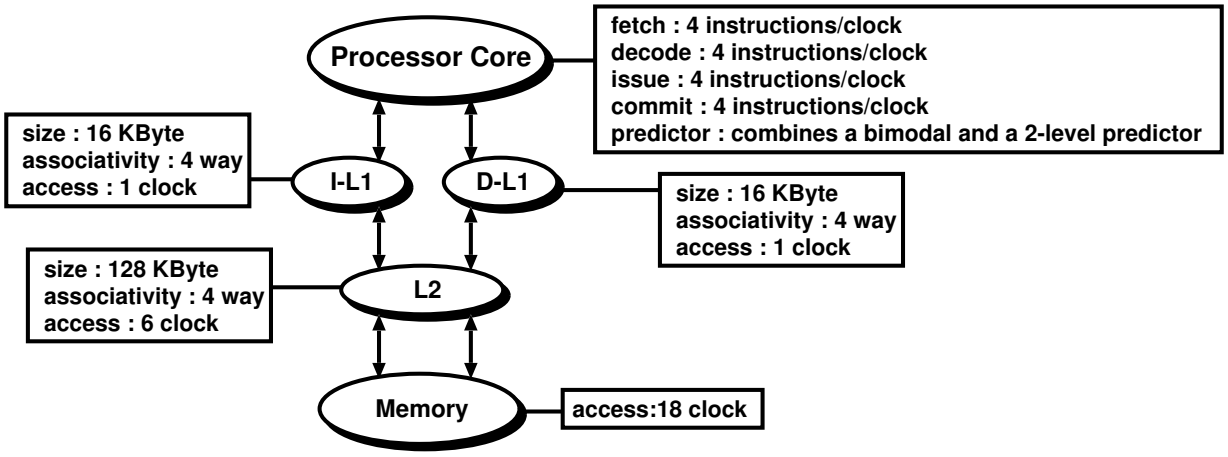


図 4.1: シングルプロセッサシステムシミュレータ

4.1.1 オリジナルの sim-outorder との誤差

図 4.1 の Processor Core をオリジナルの sim-outorder に設定したときと、今回実装した ISIS-SimpleScalar に設定したときのそれぞれの場合において、上記の 8 つのプログラムの実行した。表 4.2 にその実行クロック数とその誤差率を示す。

表 4.2: 実行速度の比較

	original sim-outorder (clock)	ISIS-SimpleScalar (clock)	誤差率 (%)
test-math	205241	202211	1.48
test-fmath	56616	56072	0.96
test-printf	1084485	1102972	1.70
test-llong	25288	25273	0.06
test-lswlr	12798	12370	3.34
FFT	36022015	35732826	0.80
LU	14717829	14627253	0.62
RADIX	26582546	28024111	5.42

オリジナルの sim-outorder では `ruu_dispatch()` において命令が実行されるため、本来は Write-back Stage で得られる分岐結果もその段階で得られてしまう。その分岐結果は次のクロックから使用され、本来より早いタイミングで命令がフェッチされてしまう。これにより、キャッシュへのアクセスの仕方も変化するため、キャッシュのヒット率やアクセス時間も変化してしまう。

実行クロック数に数%程度の誤差があるアプリケーションもあるが、このようなことが誤差の原因となる。

4.1.2 シミュレーション時間

オリジナルの sim-outorder と ISIS-SimpleScalar のそれぞれにおいて、前述した 8 つのプログラムを実行した。表 4.3 にシミュレーション時間と速度低下を示す。

どのアプリケーションを実行した場合にも 2 倍〜4 倍程度の速度低下はあるものの、シミュレーションに非現実的な時間は要さない。命令の実行を in-order から out-of-order に修正したこと、reorder buffer からのデータの取得を可能にしたこと、プロセッサ外部へのアクセスを可能にしたことなどを考慮すれば、この速度低下は十分妥当な数字であると言える。

4.2 マルチプロセッサシステムシミュレーション

ISIS-SimpleScalar を用いたマルチプロセッサシステムのシミュレーション例として、図 4.2 のようなシミュレータを実装した。各 PU は図 4.1 のような構成となっているが、図 4.1 に記載されているメモリやキャッシュは、各プロセッサによって独自に利用されるローカルデータを格納するローカルキャッシュ、ローカルメモリとして利用される。

一般的に知られていることとして、この例のように 1 つの共有メモリを複数のプロセッサで共

表 4.3: シミュレーション時間の比較

	original sim-outorder (sec)	ISIS-SimpleScalar (sec)	速度低下 (倍)
test-math	0.35	0.91	2.60
test-fmath	0.09	0.22	2.44
test-printf	2.20	6.50	2.95
test-llong	0.05	0.12	2.40
test-lswlr	0.02	0.04	2.00
FFT	118.28	363.51	3.07
LU	49.48	154.21	3.12
RADIX	79.17	317.08	4.00

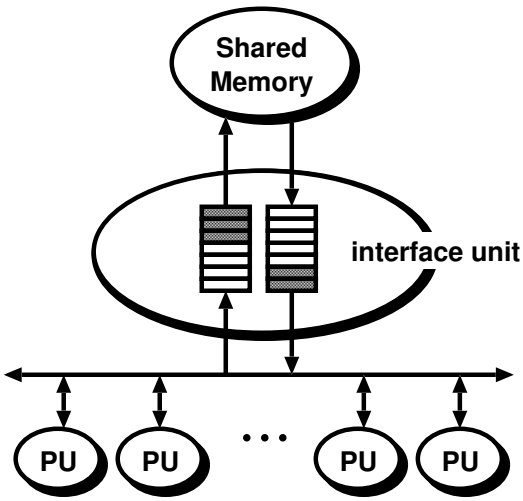


図 4.2: マルチプロセッサシステムシミュレータ

有しているマルチプロセッサシステムでは、複数のプロセッサからのメモリアクセス要求が一極集中するため、メモリアクセスレイテンシがある一定以上に大きくなると、メモリアクセス要求の待ち行列が長くなり、性能向上が望めなくなる。一方、メモリアクセスレイテンシが小さい場合には、メモリアクセスは比較的スムーズになるが、ある一定以上に PU 数を増やすとバスが混雑するため、性能向上が望めなくなる。

また、シミュレーション時間は、シミュレーションする PU 数を増やしたときや、実行クロック数が増えた場合に増加する傾向にある。

これを今回 ISIS-SimpleScalar を用いて実装したシミュレータで評価を行うことによって確かめ、そのことによってシミュレータの妥当性やシミュレータの性能などを検証する。

4.2.1 レイテンシが大きい場合の台数効果と平均メモリアクセス待ち時間の変化

共有メモリへのアクセスレイテンシを 6 とした場合に、PU 数を 1、2、4、8、16 と変化させ、そのそれぞれにおいて表 4.1 の 3 つのアプリケーションを実行した。PU を増やすことによって、実行クロック数での性能向上がどの程度得られたかを図 4.3 に、interface unit に到達してからメモリアクセスが開始されるまでの平均待ち時間がどのように変化したかを図 4.4 に示す。

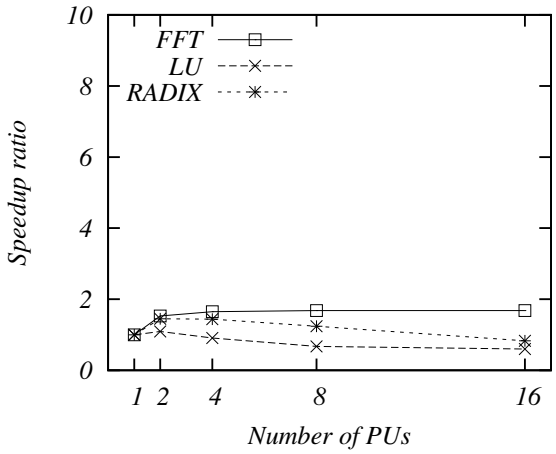


図 4.3: 台数効果 (アクセスレイテンシ 6)

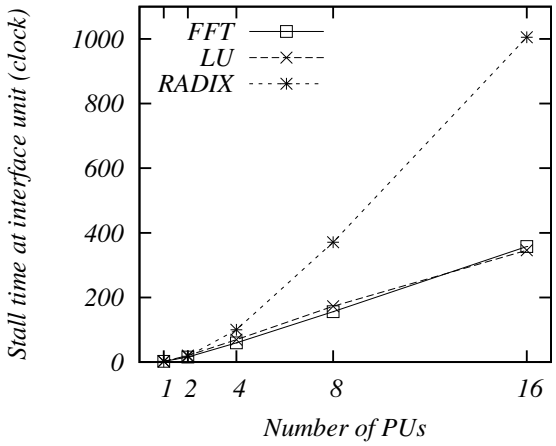


図 4.4: 平均メモリアクセス待ち時間の変化 (アクセスレイテンシ 6)

図 4.3 より、どのアプリケーションを実行した場合にも台数効果が得られていないことが分かる。一般的に図 4.2 のようなシステムでは、アクセスレイテンシが一定以上に大きくなると、メモリアクセス要求の待ち行列が長くなり、性能向上が望めなくなる。図 4.4 を見ると、PU 数を増やす毎に平均メモリアクセス待ち時間が長くなっており、それが台数効果を抑制した原因である

と考えられるため、今回のシミュレーションにおいても一般的な傾向があったことが認められる。

4.2.2 レイテンシが小さい場合の台数効果とバス利用率の変化

共有メモリへのアクセスレイテンシを 1 とした場合に、PU 数を 1、2、4、8、16 と変化させ、そのそれぞれにおいて表 4.1 の 3 つのアプリケーションを実行した。PU を増やすことによって、実行クロック数での性能向上がどの程度得られたかを図 4.5 に、共有バスがどの程度混雑したかを図 4.6 に示す。図 4.6 の縦軸のバス利用率は、単位時間当たりに共有バスが処理した平均アクセス数を示す値であり、バスの場合には複数のアクセスを同時に処理することができないため、その値が 1 を越えることはない。

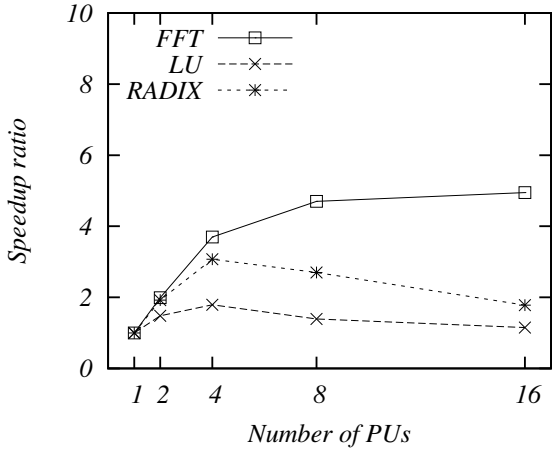


図 4.5: 台数効果 (アクセスレイテンシ 1)

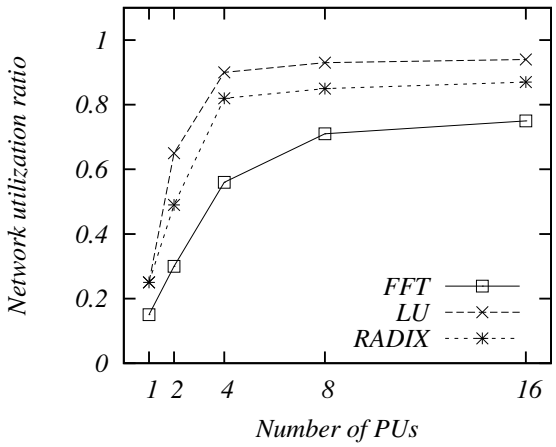


図 4.6: バス利用率の変化 (アクセスレイテンシ 1)

図 4.5 より、どのアプリケーションを実行した場合にも PU 数を増やす毎に台数効果が抑制され、特に LU や RADIX を実行した場合にその傾向が強いことが分かる。一般的に図 4.2 のようなシステムにおいて、アクセスレイテンシが小さい場合には、PU 数を増やす毎にバスの混雑度が増すため、性能向上が望めなくなる。図 4.6 より、PU 数を増やすことによって単位時間に共有バスが処理したアクセス数が増えており、特にそれは LU や RADIX を実行した場合に顕著である。よって、バスの混雑が台数効果を抑制したと考えられ、今回のシミュレーションにおいても一般的な傾向があったことが認められる。

4.2.3 メモリアクセスの割合

共有メモリへのアクセスレイテンシを 1 とした場合には、PU 数を増やしてもバスの混雑により台数効果が得られなかった。特に LU を実行した場合にその傾向が顕著であった。

ここで、全命令に占める共有メモリアクセス命令の割合を図 4.7 に示した。

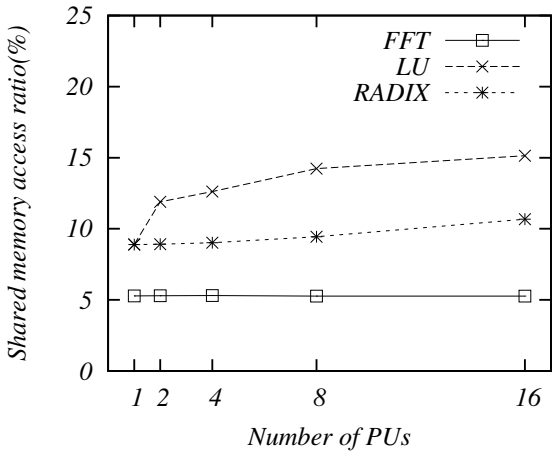


図 4.7: 全命令に占める共有メモリアクセス命令の割合

図 4.7 より、LU は他のアプリケーションに比べ共有メモリにアクセスする命令の割合が高いことが読み取れ、このことが LU のバス利用率を高めたと考えることができる。また、図 4.7 より、LU を実行した際の共有メモリアクセスの割合は、PU 数を 1 から 2 にしたときに急増していることが分かる。

図 4.8 に共有メモリアクセスに占める同期のためのメモリアクセスの割合を示した。図 4.8 を見ると、LU を実行したときに発行される同期のためのメモリアクセスが PU 数を 2 にしたときに急激に増加していることが分かり、これが PU 数を 1 から 2 にしたときに共有メモリアクセスの割合を急増させた要因であったと考えることができる。

このように、ISIS-SimpleScalar で実装されたシミュレータを用いることにより、LU を実行した場合に台数効果が得られなかった根本的な要因をつかむことができた。

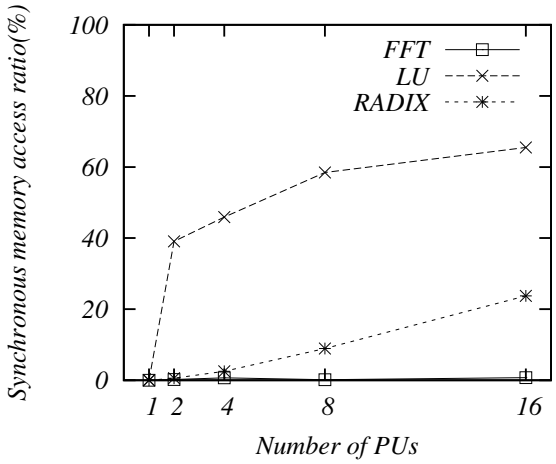


図 4.8: 共有メモリアクセスに占める同期のための共有メモリアクセスの割合

4.2.4 レイテンシが大きい場合と小さい場合の比較

共有メモリへのアクセスレイテンシを 1 とした場合に、interface unit に到達してからメモリアクセスが開始されるまでの平均待ち時間がどのように変化したかを図 4.9 に、共有メモリへのアクセスレイテンシを 6 とした場合に、PU を増やすことによって共有バスがどの程度混雑したかを図 4.10 に示す。

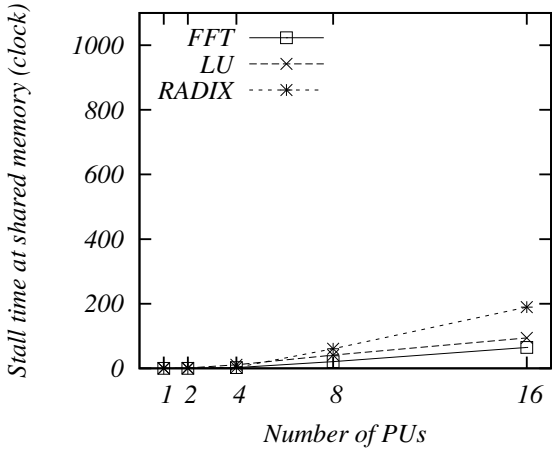


図 4.9: 平均メモリアクセス待ち時間の変化 (アクセスレイテンシ 1)

図 4.4 と図 4.9、図 4.6 と図 4.10 を比較することにより、アクセスレイテンシを 6 としたときの方が、どの PU 数においても平均メモリアクセス待ち時間が長く、アクセスレイテンシを 1 としたときの方が、どの PU 数においてもバスの利用率が高いことが分かる。この結果からも、アクセスレイテンシを 6 とした場合には平均メモリアクセス待ち時間の増大が、アクセスレイテンシ

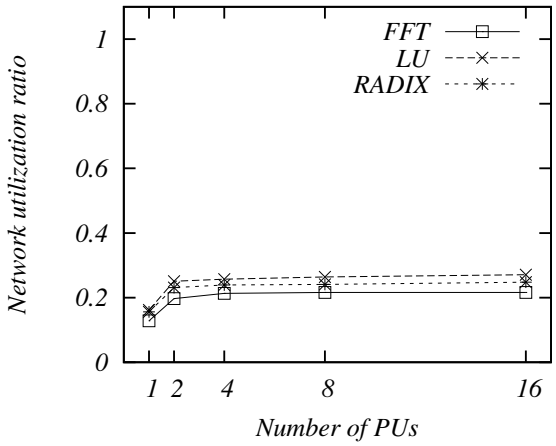


図 4.10: バス利用率の変化 (アクセスレイテンシ 6)

を 1 とした場合にはバスの混雑が、台数効果を抑制した最も大きな原因であることが認められる。

4.2.5 シミュレーション時間

アクセスレイテンシを 1 とした場合のシミュレーション時間の変化を図 4.11 に、アクセスレイテンシを 6 とした場合のシミュレーション時間の変化を図 4.12 に示す。

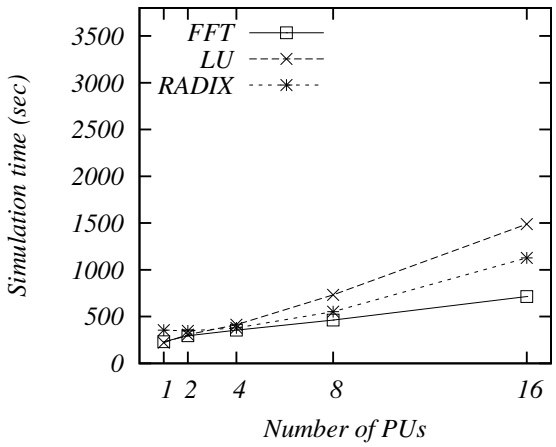


図 4.11: シミュレーション時間の変化 (アクセスレイテンシ 1)

図 4.11、図 4.12 を見ると、アクセスレイテンシに関係なく、どのアプリケーションを実行した場合にも、PU 数を増やすことによってシミュレーション時間が増したことが分かる。アクセスレイテンシを 6 としたときの方が、アプリケーションの実行にかかるクロック数が大きくなるため、シミュレーションに時間を要しているが、最もシミュレーション時間を要したで LU でも、16PU

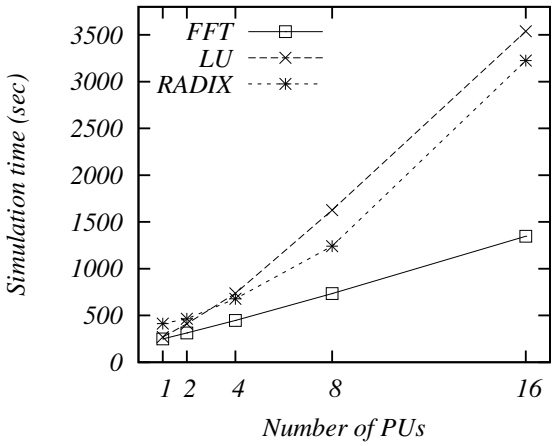


図 4.12: シミュレーション時間の変化 (アクセスレイテンシ 6)

で 60 分程度でシミュレーションを終えることができた。これより、ISIS-SimpleScalar を用いて実装されたシミュレータは、現実的な時間内でシミュレーションを行うことができると言える。

第 5 章

結論

本研究では、近年の高性能なプロセッサを詳細にモデルし、且つ、あらゆる構成のマルチプロセッサシステムを比較的容易に構築できるシミュレータの開発を目的に、ISIS-SimpleScalar の実装を行った。

実装では、SimpleScalar をマルチプロセッサシステムシミュレーションに対応させ、ISIS で実装されたシステムとの接続を可能にした。命令定義の変更や reservation station、reorder buffer の実装により、命令の実行やメモリアクセスを in-order から out-of-order に行えるように修正した。

実装した ISIS-SimpleScalar を用いてシングルプロセッサ、およびマルチプロセッサシステムのシミュレータを構築し、評価を行った。その結果、現実的な時間内でシミュレーションを終えられることが確認でき、特にマルチプロセッサシステムのシミュレーションにおいて、共有ネットワークの競合などを正確に再現できることが確かめられた。

また、ISIS-SimpleScalar を用いたシミュレータの構築には ISIS が提供するライブラリの利用が可能であるため、さまざまなシステムを比較的容易に構築できるという ISIS の利点が、そのまま享受できる。

謝辞

本研究の機会を与えて下さり、終始御指導下さった天野 英晴教授に深く感謝致します。

お忙しい中査読の労を執って頂きました田辺 靖貴氏、住吉 正人氏に心より感謝致します。

本研究に御協力頂き、さまざまな面で支えて下さった塙 敏博氏、緑川 隆氏、田辺 靖貴氏、住吉 正人氏、高橋瑞季氏に心より感謝致します。

また、本研究に御協力頂いた、天野研究室の皆様に心より感謝致します。

ふんがさん、一緒にやったソフトやサッカー、めっちゃ楽しかったです。研究への意欲が俄然高まりました。人生への意欲がなくなったとき、ふんが研に遊びに来ようと思います。そのときは、今と変わらぬ愛情を下さい！

ちゃっき〜さん、今こうやって幸せでいられるのはちゃっき〜さんのおかげです。どうしたらいいか分からないとき、ちゃっき〜さんはいつも優しく俺を包んで下さいました。めっちゃあったかかったです！俺の修論はちゃっきーさんと共著です！ありがとうございました！

どぅみ〜、どぅみ〜が SNAIL に入ってくれて、どぅみ〜が運動に誘ってくれて、どぅみ〜が Baetles の音楽くれて、本当幸せでした。どぅみ〜がいるから学校に来るのがいつも楽しみでした。修論のシールとカバー貼ってくれてありがとう！修論のレベルが急上昇しました！確実に。

大介さん、茂野さん、SNAIL に入ったのはお二人が SNAIL にいらっしゃったからです。いつも先輩達を頭の中に描きながら研究していました。いつまでも俺の憧れの存在でいて下さい！

堤さん、堤さんには新人教育のとき C 言語を教えていただきました。何にも分かってない俺に毎日優しく遅くまで教えて下さいました。あの数週間はそれまでの 3 年間よりも濃い内容でした。堤さんに教えてもらったリスト構造の書き方、修論でいっぱい使いました。堤さんと一緒にやったソフトも楽しかったです！堤さんがセンターにいるから思い切ってプレーすることができました。また一緒にソフトしたいです！

伊豆さん、俺、伊豆さんみたいな誰からも好かれる人間になりたいです。伊豆さんと一緒にサッカーしたり、ソフトしたり、すべての時間が本当楽しかったです。伊豆さん本当カッコいいッス！これからも伊豆さん目指して生きて行きます！

長名さん、茂野さん、伊豆さん、安福さん、花見のとき家まで送ってくれてありがとうございました。健康に研究できたのは皆さんのおかげです。

のすけさん、伊豆さん、一緒に TEU で仕事ができて、バイトの時間がバイトじゃないくらい楽しくて、研究の十分な気分転換になりました。いつも生徒以上に質問してすいませんでした。

吉見、吉見のこと、いつまでも忘れないよ！

相良、アパートの契約が切れてから、快く居候させてくれて本当にありがとう！居心地が良すぎて、研究が捗りまくりました！いつも元気をくれてありがとう！

金子くん、宮本くん、横田くん、また、みんなで鍋を食べたいです。2004 年度の TA は楽しかったです。ありがとう！鍋マジうまかったです！

体育会準硬式野球部のみんな、みんなと一緒に頑張った思い出があったから、どんな辛いときもその先の幸せを思い描きながら研究することができました。みんなとの思い出は、これからもずっと俺を助けてくれると思います！どうもありがとう！

四谷大塚で一緒にバイトしたみんな、みんなといると自分が面白い人間かと勘違いするくらいでした。いつもいっぱい笑ってくれてありがとう！バイトの時間はすごく幸せな時間でした。みんながいたから、くじけずに研究できました。もしまたみんなで遊ぶことがあれば、そのときはまたたくさんの元気を下さい！

地元のみんな、俺はみんなの影響をいっぱい受けて成長しました。みんないい奴ばっかりだから、どんなに落ち込んだときも、みんなのことを考えると、十分過ぎるくらいの自信が湧いてきました。それはこれからもきっと、ずっとそうです。幼少時代、少年時代、青年時代を、みんなと一緒に過ごせて本当よかったです！ありがとう！

家族のみんな、いつも迷惑ばっかりかけてごめんなさい。大学院進学のことや就職のこと、いつもわがままを突き通してきたこと、本当ごめんなさい。それなのに、困ったときはいつも助けてくれて、感謝しています。おじいちゃんとおばあちゃんの孫として、お父さんとお母さんの子供として、美由希の兄として生まれてこられたこと、とても幸せに思います。みんなと家族であることを思うと、無敵な気分になれます！これからも俺が頑張れるように、これから少しでもみんなに恩返しできるように、いつまでも元気でいて下さい！

さまざまな理由で名前を挙げられなかった方々、皆さんがいたから最後まであきらめずに研究することができました。本当に素晴らしい学生生活になりました。心から感謝しています。大学と大学院にいた約 6 年間、本当にたくさんの人からたくさんの元気をもらいました。みんなとの思い出をパワーに、これからも頑張っていきたいと思います。

2005 年春

参考文献

[1] M. Rosenblum, S.A. Herrod, E. Witchel, and A. Gupta : Complete Computer Simulation: The SimOS Approach, IEEE Parallel and Distributed Technology, vol. 3, no. 4, Winter 1995.

[2] The SimOS Home Page, http://simos.stanford.edu/

[3] D. Burger, T. Austin : The SimpleScalar Tool Set, Version 2.0, 1997.

[4] T. Austin, E. Larson, and D. Ernst : SimpleScalar: An Infrastructure for Computer System Modeling, IEEE Computer, 35(2), Feb 2002.

[5] Todd Austin : SimpleScalar Hacker's Guide, SimpleScalar LLC

[6] SimpleScalar LLC Home Page, http://www.simplescalar.com/

[7] 若林正樹, 天野英晴 : 並列計算機シミュレータの構築支援環境, 電子情報通信学会論文誌 D-I Vol.J84D-I No.3, pp.1-10, March 2001.

[8] ISIS Home Page, http://www.am.ics.keio.ac.jp/isis/index-ja.html

[9] Takuya Terasawa, Oh Yamamoto, Tomohiro Kudoh, and Hideharu Amano : A performance evaluation of the multiprocessor testbed ATTEMPT-0, Parallel Computing, Vol.21, pp.701-730, 1995.

[10] Bjarne Stroustrup : プログラミング言語 C++ 第 3 版, トッパン, 1998.

[11] IDT R30xx Family Software Reference Manual, Integrated DeviceTechnology, Inc., 1994.

[12] NEC : ユーザーズマニュアル V_R3000A^{TM} 32 ビットマイクロプロセッサアーキテクチャ編, 1993.

[13] IDT79R3081 RISController Hardware User's Manual, Integrated Device Technology, Inc., December 11, 1992.

[14] 藥袋俊也, 緑川隆, 田辺靖貴, 茂野真義, 天野英晴 : オンチップマルチプロセッサ向け内部接続網の検討, 情報処理学会研究報告 2003-ARC-153, pp.1-6, May 2003.

[15] 住吉正人, 緑川隆, 茂野真義, 田辺靖貴, 藥袋俊也, 天野英晴 : 一時的にディレクトリを保持する MINDIC スイッチの設計と評価, 情報処理学会研究報告 2004-EVA-8, August 2004.

[16] 藥袋 俊也, 塙 敏博, 田辺靖貴, 天野英晴 : ISIS-SimpleScalar の実装, 情報処理学会研究報告 2004-ARC-160, pp.29-34, December 2004.

付 録

発表論文

A.1 研究会

[14] 藥袋俊也, 緑川隆, 田辺靖貴, 茂野真義, 天野英晴 : オンチップマルチプロセッサ向け内部接続網の検討, 情報処理学会研究報告 2003-ARC-153, pp.1-6, May 2003.

[15] 住吉正人, 緑川隆, 茂野真義, 田辺靖貴, 藥袋俊也, 天野英晴 : 一時的にディレクトリを保持する MINDIC スイッチの設計と評価, 情報処理学会研究報告 2004-EVA-8, August 2004.

[16] 藥袋俊也, 塙敏博, 田辺靖貴, 天野英晴 : ISIS-SimpleScalar の実装, 情報処理学会研究報告 2004-ARC-160, pp.29-34, December 2004.

付 録

ISIS-SimpleScalar User's Guide

B.1 プラットフォーム

現在、以下のプラットフォームでの動作が確認済みである。

- isis-1.1.1-2003101601-ss (テスト版 ISIS 対応 ISIS-SimpleScalar)

コンパイラ / OS	gcc-2.95.3	gcc-3.3.5
Fedora Core 3 (Linux)	○	○
Plamo 4 (Linux)	○	○

- isis-1.1.1-ss (開発版 ISIS 対応 ISIS-SimpleScalar)

コンパイラ / OS	gcc-2.95.3	gcc-3.3.5
Fedora Core 3 (Linux)	○	×
Plamo 4 (Linux)	○	×

以後、OS を Fedora Core 3 として説明を行う。尚、他のプラットフォームでの利用方法などは ISIS-SimpleScalar のホームページ (http://www.am.ics.keio.ac.jp/proj/snail/isis-ss/index.html) に記述してある。今後、ISIS-SimpleScalar に更新があった場合にも、このページ上で報告する予定である。

B.2 ISIS-SimpleScalar

現在、次の ISIS-SimpleScalar が利用可能である。

- テスト版 ISIS 対応の ISIS-SimpleScalar – isis-1.1.1-2003101601-ss
- 開発版 ISIS 対応の ISIS-SimpleScalar – isis-1.1.1-ss

isis-1.1.1-2003101601-ss での説明を行う。

B.2.1 ダウンロード

ISIS のホームページ (http://www.am.ics.keio.ac.jp/isis/index-ja.html) と、ISIS-SimpleScalar のホームページ (http://www.am.ics.keio.ac.jp/proj/snail/isis-ss/index.html) から、次をダウンロードする。

- isis-1.1.1-2003101601.tar.gz
- isis-1.1.1-2003101601-ss.tar.gz

B.2.2 解凍

以下の順序で解凍することによって、ISIS に ISIS-SimpleScalar のライブラリが投入され、Makefile なども ISIS-SimpleScalar 対応のものに更新される。

```
% tar zvxf isis-1.1.1-2003101601.tar.gz
% tar zvxf isis-1.1.1-2003101601-ss.tar.gz
```

解凍すると、isis-1.1.1/lib/simoutorder/に ISIS-SimpleScalar に関連する次のファイルが入っている。

addr_order_buffer{.h,.cc}

addr_order_buffer クラスが定義してある。Load 命令および Store 命令のメモリアクセスアドレスを保持し、メモリアクセスの順序管理を行う。

address.h

ISIS-SimpleScalar が扱う共有メモリ領域のアドレス空間が定義してある。

arg{.h,.cc}

arg クラスが定義してある。コマンドラインオプションとして指定されたプロセッサ数、プロセッサが外部システムと通信するためのポート数を読み取る。

bpred{.h,.cc}

SimpleScalar が提供する bpred.c をクラス化した bpred_simple クラスが定義してある。branch predictor を表し、branch predictor の初期化、参照、更新を行う。

cache{.h,.cc}

SimpleScalar が提供する cache.c をクラス化した cache_simple クラスが定義してある。ISIS-SimpleScalar においては 0x00000000〜0x7fffffff の領域の命令やローカルデータを格納するローカルキャッシュを表す。命令やデータは実際には格納しておらず、キャッシュアクセスが発生

したときには、キャッシュヒット/ミスが判定され、キャッシュアクセス時間が返される。命令やデータはメモリから読み出される。

ecoff.h

SimpleScalar ECOFF 形式のバイナリを生成するための定義がなされている。

endian_s{.h,.cc}

ホストマシンとターゲットマシン上のエンディアンを決めるための関数が定義してある。

exec_buffer{.h,.cc}

execution_buffer クラスが定義されている。実行中の命令を管理するクラスである。命令は実行される時刻 (Functional Unit での計算が終了する時刻、またはローカルメモリへのアクセスが終了する時刻) で昇順に並べられ、同じ時刻に実行される命令は命令 ID で昇順に並べられる。

host.h

ホストマシンに依存する関数の定義変更などがなされている。

instqueue{.h,.cc}

instqueue クラスが定義してある。各命令に対するあらゆる情報を保持し、命令は in-order で並べられる。Register Update Unit および reservation station の役割を担う。

isim_bus_packet{.h,.cc}

isim_bus_packet クラスが定義してある。ISIS-SimpleScalar が提供するプロセッサ (isim_processor) とプロセッサ外部のシステムとの間でやり取りされるパケットを表す。

isim_bus_port{.h,.cc}

isim_bus_port クラスが定義してある。isim_processor とプロセッサ外部にあるユニットとの接続に使用されるポートを表すクラスである。入力できるパケットは isim_bus_packet クラスで定義してあるもののみである。

isim_processor{.h,.cc}、instruction.cc、inst_simple.cc

isim_processor クラスが定義してある。
isim_processor.cc には、パイプライン内の各 Stage の動作が記述してある。

- `void ruu_fetch(void)` – Fetch Stage
- `void ruu_dispatch(void)` – Dispatch Stage
- `void ruu_issue(void)` – Schedule Stage
- `void ruu_writeback(void)` – Exec Stage と Writeback Stage
- `void ruu_commit(void)` – Commit Stage

instruction.cc には、`isim_processor` が実行する各命令の処理が定義されている。

- `void ＊_impl(void)` – 値の読み出しや計算結果の生成、読み出した値を確保する部分
- `void ＊_addr_impl(void)` – メモリアクセスアドレスを計算する部分
- `void ＊_set_req_impl(void)` – 共有メモリに読み出し要求を発行する部分
- `void ＊_reg_impl(void)` – レジスタを更新する部分

inst_simple.cc には、レジスタの更新やプロセッサ外部への要求発行など、複数の命令で行われる処理をメンバ関数としてまとめて定義し、各命令が呼び出せるようにした。

loader_s{.h,.cc}

SimpleScalar が提供する loader.c をクラス化した `loader_simple` クラスが定義してある。シミュレータが実行するバイナリを読み込み、`0x00400000`〜`0x0fffffff` に格納する。

machine{.h,.cc}

Functional Unit、immediate field、メモリ領域など、プロセッサが必要とするさまざまな定義がなされている。

machine.def

`isim_processor` で実行される各命令の識別子、オペコード、使用する Functional Unit 名、使用するレジスタ番号などの情報が定義してある。

memory_s{.h,.cc}

SimpleScalar が提供する memory.c をクラス化した `memory_simple` クラスが定義してある。ISIS-SimpleScalar においては、命令やローカルデータを格納するローカルメモリを表す。メモリ領域は `0x00000000`〜`0x7fffffff` である。

misc{.h,.cc}

メッセージ出力など、多数の便利なサポート関数などを定義している。

only_syscall.h

システムコールに関する定義がなされている。

options{.h,.cc}

SimpleScalar が提供する options.c をクラス化した `option_simple` クラスが定義してある。キャッシュ、branch predictor など、システムに関するさまざまなコマンドラインオプションを解釈する。

order_buffer{.h,.cc}

`addr_order_buffer` クラスが定義してある。各命令の命令 ID と使用するレジスタ番号を保持し、レジスタ間の依存関係によって実行の順序を管理する。

regs_head.h

レジスタが構造体として定義してある。

resource{.h,.cc}

`resource_simple` クラスが定義してある。Functional Unit を生成する。

stats{.h,.cc}

SimpleScalar が提供する stats.c をクラス化した `stats_simple` クラスが定義してある。シミュレーション中にプロセッサ毎に取られる、実行クロック数やローカルキャッシュのヒット率などの統計データの指定、初期化、出力などを行う。

version.h

SimpleScalar のバージョンに関する定義がなされている。

writeback_buffer{.h,.cc}

`writeback_buffer` クラスが定義してある。Writeback イベントを終了した命令と、各命令がレジスタを更新する際の値 (計算結果やメモリから読み出した値) を保持し、reorder buffer の役割を担う。命令は命令 ID で昇順に並べられる。

B.2.3 インストール

インストールには ANSI/ISO C++コンパイラ (gcc-2.95.3、gcc-3.3.5 で動作確認済) が必要である。/usr/local/sim/isis-ss-inst にインストールするとする。

```
% cd isis-1.1.1
% ./configure --prefix=/usr/local/sim/isis-ss-inst \
 --disable-sample --disable-shared
% make
% make install
```

B.3 SimpleScalar のインストール

SimpleScalarが提供するプロセッサシミュレータ上で実行できる実行バイナリは、ISIS-SimpleScalarを用いて実装したシミュレータ上でも実行することができる。その実行バイナリを生成するために必要なツールのインストール方法を以下に示す。

B.3.1 ダウンロード

SimpleScalar のホームページ (http://www.simplescalar.com/) から、以下をダウンロードする。

- simpleutils-990811.tar.gz
- gcc-2.7.2.3.tar.gz
- simpletools-2v0.tar.gz

B.3.2 binutils のインストール

インストールには ANSI/ISO C++コンパイラ (gcc-3.3.5 でインストール確認済) が必要である。/usr/local/sim/simplescalar-inst にインストールするとする。

```
% tar zvxf simpleutils-990811.tar.gz
% cd simpleutils-990811
% ./configure --host=i386-intel-linux --build=i386-intel-linux \
 --target=sslittle-na-sstrix --with-gnu-as --with-gnu-ld \
 --prefix=/usr/local/sim/simplescalar-inst
% make
% make install
```

B.3.3 gcc のインストール

インストールには ANSI/ISO C++コンパイラ (gcc-3.3.5 でインストール確認済) が必要である。/usr/local/sim/simplescalar-inst にインストールするとする。

```
% tar zvxf gcc-2.7.2.3.tar.gz
% cd gcc-2.7.2.3
```

libgcc2.c の 98 行目に以下を追加

- `#define BITS_PER_UNIT 8`

```
% ./configure --host=i386-intel-linux --build=i386-intel-linux \
 --target=sslittle-na-sstrix --with-gnu-as --with-gnu-ld \
 --prefix=/usr/local/sim/simplescalar-inst
% make LANGUAGES="c c++" CFLAGS="-O3" CC="gcc" \
 LIBGCC2_INCLUDES="-I/usr/include"
```

エラーが出たら、insn-output.c の 675、750、823 行目の最後に"\"を追加

- `return "FIXME\n\`

```
% make LANGUAGES="c c++" CFLAGS="-O3" CC="gcc" \
 LIBGCC2_INCLUDES="-I/usr/include"
% make install LANGUAGES="c c++" CFLAGS="-O3" CC="gcc" \
 LIBGCC2_INCLUDES="-I/usr/include"
```

B.3.4 crt0.o と libc.a の投入

```
% tar vxfz simpletools-2v0.tar.gz
% cp sslittle-na-sstrix/lib/crt0.o \
 /usr/local/sim/simplescalar-inst/sslittle-na-sstrix/lib/
% cp sslittle-na-sstrix/lib/libc.a \
 /usr/local/sim/simplescalar-inst/sslittle-na-sstrix/lib/
```

B.4 サンプルシミュレータ

ISIS-SimpleScalar で実装されたサンプルシミュレータを実行する方法を示す。ここで紹介するサンプルシミュレータは、図 B.1 のような構成のシステムのシミュレータである。複数のプロセッシングユニット (PU) と 1 つの共有メモリ (shared memory) が単純な共有バス (shared bus) で接続されている。

また、各 PU は図 B.2 のような構成となっているが、各パラメータはコマンドラインオプションを用いて簡単に変更できる。

B.4.1 ダウンロード

ISIS-SimpleScalar のホームページ (http://www.am.ics.keio.ac.jp/proj/snail/isis-ss/index.html) から、以下をダウンロードする。

- simple_bus_system.tar.gz

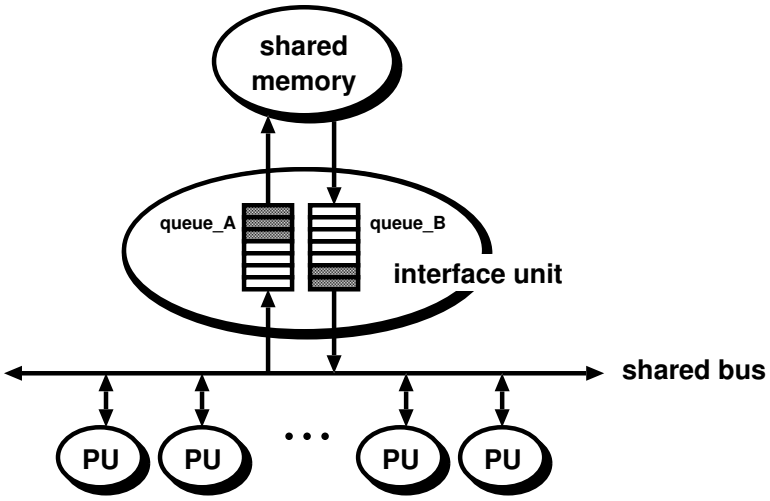


図 B.1: サンプルシミュレータ

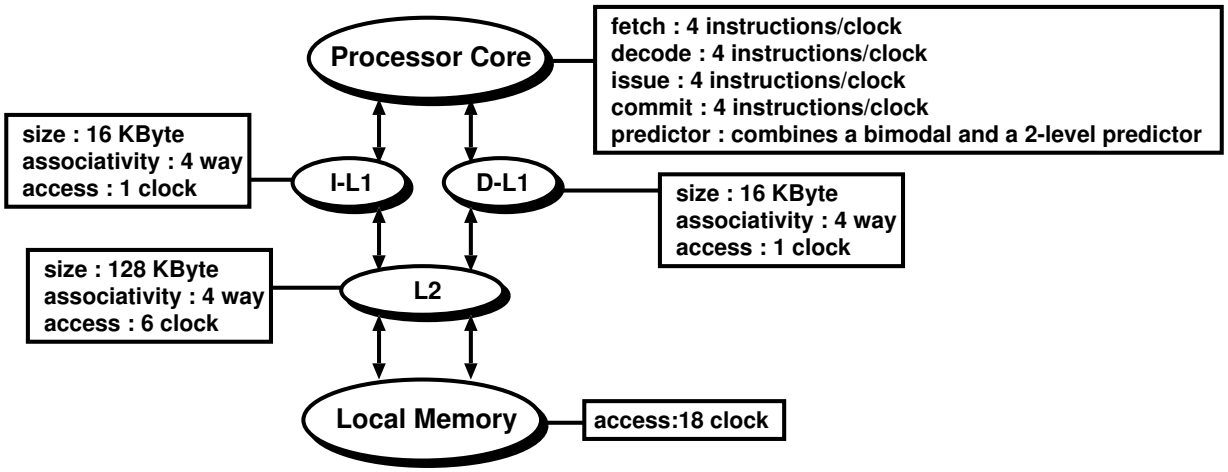


図 B.2: 各 PU の構成

B.4.2 解凍

```
% tar zvxf simple_bus_system.tar.gz
% cd simple_bus_system
```

解凍すると、以下のファイルが入っている。

- Makefile
- main.cc
- shared_memory_access_queue{.h,.cc}
- shared_bus{.h,.cc}
- shared_memory{.h,.cc}
- isim_system{.h,.cc}

shared_memory_access_queue{.h,.cc}

アドレス、データ、データサイズ、命令 ID、プロセッサ ID などの要素をリスト管理する。
メンバ関数として、

- `void insertlist(int inst_id, md_addr_t addr, word_t dw1, word_t dw2, half_t dh, byte_t db, int data_size, int rw, int puid, int time, bool flag)`
  - キューの末尾に要素を追加する
- `void deletelist(int inst_id, md_addr_t addr, int puid)`
  - 該当するキュー内の要素を削除する
- `void get_data(int* id, md_addr_t* addr, word_t* dw1, word_t* dw2, half_t* dh, byte_t* db, int* data_size, int* rw, int* puid, int now, bool* flag)`
  - キューの先頭要素を渡す

が定義してある。

shared_bus{.h,.cc}

図 B.1 の shared bus を表す。interface unit と PU に接続され、要求やデータを送信したい interface unit や PU に対してアービトレーションを行い、それらに順次使用権を与える。要求やデータの管理には shared_memory_access_queue{.h,.cc} で定義されたキューを用いる。
メンバ関数として、

- `void get_packet(void)`
  - PU および interface unit からの要求を受け取り、キューの末尾に順次追加する
- `void send_packet(void)`
  - キューの先頭から要素を取り出し、その要求に応じて PU または interface unit に転送する

が定義してある。

shared_memory{.h,.cc}

図 B.1 の interface unit、shared memory を表す。shared bus を介して転送されてきた要求に対して、shared memory からの読み出しや shared memory への書き込みを行う。また、読み出し要求に対しては、PU への読み出しデータ送信要求を shared bus に発行する。要求やデータの管理には shared_memory_access_queue{.h,.cc} で定義されたキューを用いる。

メンバ関数として、

- `void clock(void)`
  - shared bus から転送されてきた要求を受け取り、キュー A の末尾に順次追加する
  - キュー A の先頭から要求を取り出し、その要求に応じたメモリアクセスを行う
  - 読み出しデータ送信要求をキュー B の末尾に順次追加する
  - キュー B の先頭から要求を取り出し、shared bus に発行する

が定義してある。

isim_system{.h,.cc}

シミュレータ全体のシステムを 1 つにまとめ、ユニットの接続やクロックの分配を行う。

メンバ関数として、

- `void initialize(unsigned int, int)`
  - 引数として与えられた数分の PU を用意する
  - 引数として与えられた数分のポートを用意する
  - PU と shared bus、interface unit と shared bus を接続する
  - shared memory の領域を指定する
  - 各 PU に ID を割り当てる
- `void sim_main(void)`
  - シミュレータにクロックを入れる

等が定義してある。

B.4.3 シミュレータの make

Makefile を修正する。CC には、ISIS-SimpleScalar をインストールした際と同じバージョンの ANSI/ISO C++コンパイラを指定し、isisdir には、ISIS-SimpleScalar がインストールしてあるディレクトリを指定する。
<例>

- `CC = /usr/local/gcc/bin/g++`
- `isisdir = /usr/local/sim/isis-ss-inst`

```
% make
```

"simulator"という実行ファイルができれば、成功である。

B.5 シミュレータ上でアプリケーションの実行

シミュレータ上でのアプリケーションの実行について説明する。/usr/lib に libstdc++.so.5 がない場合には、ホストマシンの gcc/lib/libstdc++.so.5 のリンクを貼る必要がある。
<例>

```
% ln -s /usr/local/gcc-3.3.5/lib/libstdc++.so.5 /usr/lib
```

B.5.1 用意されている実行バイナリの実行

simple_bus_system/tests 内に、以下の実行バイナリが入っている。

- SimpleScalar が提供するテストプログラム
  - tests/bin.little/test-fmath
  - tests/bin.little/test-llong
  - tests/bin.little/test-lswlr.out
  - tests/bin.little/test-math
  - tests/bin.little/test-print
- マルチプロセッサシステムシンプルテストプログラム
  - tests/test/kuku.out (九九の並列計算)

実行は次のようにする。

```
% ./simulator (simulator オプション) 実行バイナリ (実行バイナリオプション)
```

実行例は次の通りである。

- SimpleScalar が提供するテストプログラム (シングルプロセッサ用)

```
% ./simulator tests/bin.little/test-lswlr.out
% ./simulator tests/bin.little/test-fmath
```

- マルチプロセッサシステムシミュレーション用
  - コマンドの末尾の-p2 や-p16 というオプションは、プロセッサ数を指定する実行バイナリへのオプションである

```
% ./simulator -p 2 tests/test/test.out -p2
% ./simulator -p 16 tests/test/test.out -p16
```

B.5.2 実行バイナリの作成

独自に作成したプログラムの実行バイナリを生成し、それをシミュレータ上で実行する方法を説明する。

プログラムの作成

<例> hello_world.c

```
#include <stdio.h>

int main (void) {
    printf("Hello World!\n");
    return 0;
}
```

コンパイル

SimpleScalar が/usr/local/sim/simplescalar-inst にインストールされているとする。

```
% /usr/local/sim/simplescalar-inst/bin/sslittle-na-sstrix-gcc \
 hello_world.c -I/usr/include -o hello_world.out
```

実行

```
% ./simlator hello_world.out
```

実行結果

```
-----------------以上省略 -----------------
sim: ** starting performance simulation **
Hello World!

sim: ** simulation statistics **
-----------------以下省略 -----------------
```

B.5.3 マルチプロセッサシステムシミュレーション用並列計算プログラムの作成

マルチプロセッサシステム上で実行する並列計算プログラムの作成方法について説明する。simple_bus_system/tests/test 内に、並列計算プログラムの作成を助ける次のプログラムが入っている。これらのプログラムのコンパイルには、同じディレクトリ内に ISIS-SimpleScalar の address.h へのリンクが貼ってある必要がある。

```
ln -s /usr/local/sim/isis-ss/include/isis/address.h address.h
```

get_puid.c

プロセッサ ID を読み出すためのアドレスへアクセスするプログラムである。main プログラムに、

```
unsigned int puid = get_puid();
if ( puid == 0 ) {
  処理 A;
}
```

と記述すれば、プロセッサ ID が 0 のプロセッサのみが処理 A を行う。

get_punum.c

システム上のプロセッサ数を読み出すためのアドレスへアクセスするプログラムである。main プログラムに、

```
unsigned int punum = get_punum();
```

と記述すれば、punum にプロセッサ数が代入される。

barrier.c

すべてのプロセッサで同期をとるためのプログラムである。main プログラムに、

```
barrier();
```

と記述すれば、すべてのプロセッサがこの barrier() に達するまで、他のプロセッサは barrier() から抜けない。つまり、

```
処理 A;
barrier();
処理 B;
```

と記述すれば、すべてのプロセッサが処理 A を終了してから、各プロセッサは処理 B を始める。

shared_addr_allocate.c

address.h で定義された SHARED_MEM_TOP〜SHARED_MEM_BOTTOM の範囲からデータ領域を確保するプログラムである。main プログラムに、

```
unsigned int *mem;
mem = (unsigned int *)shared_addr_allocate(sizeof(unsigned int));
```

と記述すれば、すべてのプロセッサで共有される unsigned int 型の大きさの領域を SHARED_MEM_TOP〜SHARED_MEM_BOTTOM の範囲から確保する。

shared_addr_for_sync_allocate.c

address.h で定義された SHARED_MEM_FOR_SYNC_TOP〜SHARED_MEM_FOR_SYNC_BOTTOM の範囲からデータ領域を確保するプログラムである。main プログラムに、

```
unsigned int *mem;
mem = (unsigned int *)shared_addr_for_sync_allocate(sizeof(unsigned int));
```

と記述すれば、すべてのプロセッサで共有される unsigned int 型の大きさの領域を SHARED_MEM_FOR_SYNC_TOP〜SHARED_MEM_FOR_SYNC_BOTTOM の範囲から確保する。

fad_addr_allocate.c

同期機構を実現するための Fetch and Dec を行うためのプログラムである。main プログラムに、

```
unsigned *mem;
mem = (unsigned int*)fad_addr_allocate(sizeof(unsigned int));
```

と記述すれば、すべてのプロセッサで共有される unsigned int 型の大きさの領域を address.h で定義された FAD_MEM_TOP〜FAD_MEM_BOTTOM の範囲から確保する。その後、

```
if ( get_puid() == 0 )
    *mem = get_punum();

barrier();

unsigned int *a;
a = (unsigned int *)shared_addr_allocate(sizeof(unsigned int));
*a = 0;

if ( *mem > 1 )
    while(!(*a)) {}
else
    *a = 1;
```

と記述し、シミュレータの FAD_MEM_TOP〜FAD_MEM_BOTTOM へのメモリアクセスに対する処理を

- メモリの値を読み出し、その値から 1 引いた数をメモリに書き込む
- プロセッサには、1 引く前の値を返す

とすれば、すべてのプロセッサ間で同期を取ることができる。これを応用したものが上記した barrier.c である。

tas_addr_allocate.c

同期機構を実現するための Test and Set を行うためのプログラムである。main プログラムに、

```
unsigned *idlock;
idlock = (unsigned*)tas_addr_allocate(sizeof(unsigned));
```

と記述すれば、すべてのプロセッサで共有される unsigned 型の大きさの領域を address.h で定義された TAS_MEM_TOP〜TAS_MEM_BOTTOM の範囲から確保する。その後、

```
unsigned int *id;
id = (unsigned*)shared_addr_allocate(sizeof(unsigned));
*id = 0;
unsigned int Mynum;
*idlock = 1;

barrier();

while (*idlock) {}

Mynum = id;
id++;

*(Global->idlock) = 1;
```

と記述し、シミュレータの TAS_MEM_TOP〜TAS_MEM_BOTTOM へのメモリアクセスに対する処理を

- メモリの値が 1 ならばメモリに 0 を書き込みプロセッサに 0 を返す
- メモリの値が 0 ならプロセッサに 1 を返す

とすれば、各プロセッサの Mynum がすべて異なる値となる。

system_clock.c

このプログラムは、統計データを取得するためのものであり並列計算プログラムの作成を助けるためのものではないが、説明する。ISIS-SimpleScalar では、0xd0000000 番地へのメモリアクセスがあってから 0xd0000100 番地へのメモリアクセスがあるまでカウントされる特定部分統計データの取得が可能である。0xd0000000 番地へアクセスするコードと 0xd0000100 番地へアクセスするコードをシミュレータが実行するプログラムに埋め込むことによって、ユーザは任意の部分の統計データを取ることができる。

main プログラムに、

```
unsigned int start = calc_start();
```

と記述すれば、0xd0000000 番地へのアクセスが発行される。つまり、シミュレータ側で特定部分統計データの取得が始まる。

```
unsigned int end = calc_end();
```

と記述すれば、0xd0000100 番地へのアクセスが発行される。つまり、特定部分統計データの取得が終了する。取得された統計データは、実行終了時に全時間統計データとは別に出力される。

Makefile

SimpleScalar が/usr/local/sim/simplescalar-inst にインストールされているとする。

```
CC = /usr/local/sim/simplescalar-inst/bin/sslittle-na-sstrix-gcc
```

PROGRAM でコンパイルソースを指定する。

```
PROGRAM = main.c \
          get_puid.c \
          get_punum.c \
          barrier.c \
          shared_addr_allocate.c \
          shared_addr_for_sync_allocate.c \
          fad_addr_allocate.c \
          tas_addr_allocate.c \
          system_clock.c
```

B.6 新しいシミュレータの実装

ISIS-SimpleScalar を用いたシミュレータの構築には ISIS が提供するライブラリの利用が可能であるため、さまざまなシステムを比較的容易に構築できるという ISIS の利点がそのまま享受できる。ISIS については、isis-1.1.1 online manual (http://www.am.ics.keio.ac.jp/isis/manual/index-ja.html) を参考にして頂きたい。

新しいシミュレータの実装には、サンプルシミュレータのコードが参考になる。shared_bus{.h,.cc} や shared_memory{.h,.cc} を参考にプロセッサ外部のモジュールを記述し、`isim_system` クラス (isim_system{.h,.cc}) のメンバ関数 `void initialize(unsigned int, int)` でモジュールのポート同士を接続する記述を行い、`void sim_main(void)` で各モジュールにクロックを入れる記述を行えば良い。

サンプルシミュレータのコードを示し、具体的に説明する。

`void initialize(unsigned int, int)`

システムを構成するモジュールを表すクラスのインスタンス生成や、それらの接続を行う。

```
1:   void
2:   isim_system::initialize(unsigned int punum, int portnum)
3:   {
4:       soo.resize(punum);
5:       shared_mem_isis.set_top(0x80000000);
6:       shared_mem_isis.resize(0x7fffffff);
7:       shmem.connect_mem(shared_mem_isis);
8:       shbus.set_each_pu_port_num(portnum);
```

```
 9:       shbus.set_num_of_pu_port(portnum*punum);
10:       shmem.ref_shbus_port().connect(shbus.ref_mm_port());
11:       for ( unsigned int i = 0; i < punum; i++ ) {
12:           soo[i].set_puid(i);
13:           soo[i].set_punum(punum);
14:           soo[i].set_punum(portnum);
15:           for ( int p = 0; p < portnum; p++ ) {
16:               shbus.ref_to_pu_port(i*portnum+p).connect(soo[i].ref_to_pu_
      port(p));
17:               shbus.ref_from_pu_port(i*portnum+p).connect(soo[i].ref_from
      _pu_port(p));
18:       }
19:   }
```

以上のコードにより、図 B.3 のようなシステムが構築される。

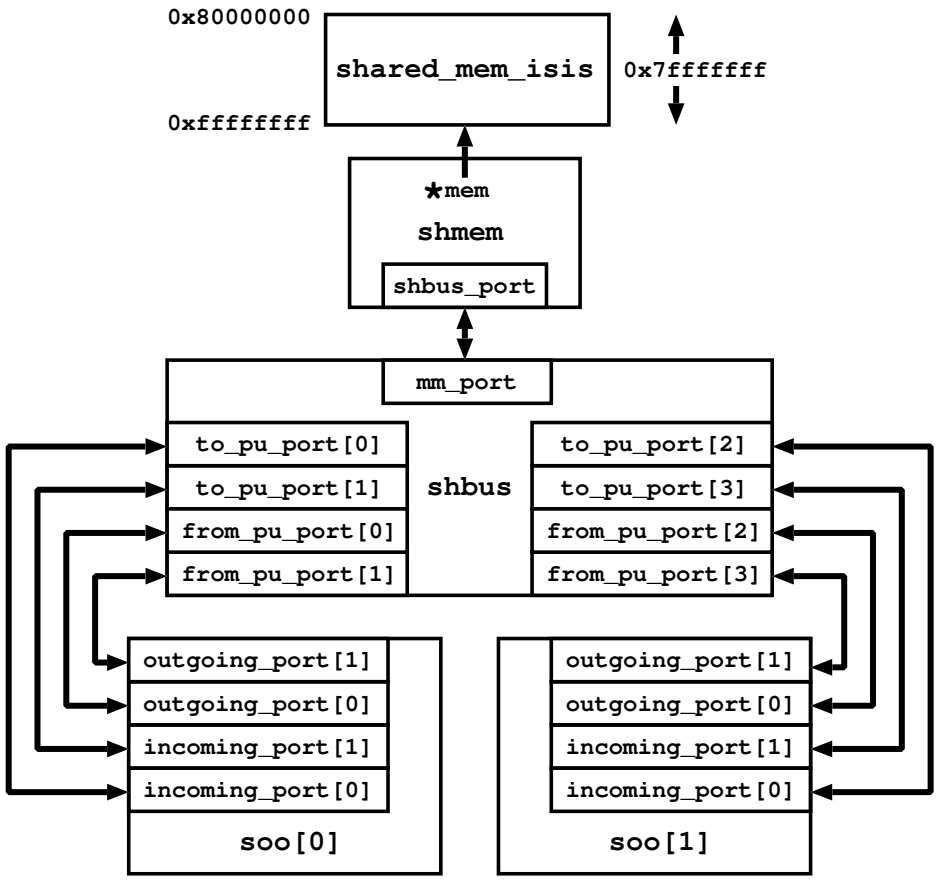


図 B.3: 上記コードによって構築されるシステム

サンプルコードに沿って、システムが構成される流れを示す。

2: 引数としてプロセッサ数とポート数を受け取る

4: isim_processor クラスのインスタンス soo の数を指定されたプロセッサ数分にする
   - 図 B.3 はプロセッサ数が 2 の場合

5: mapped_memory クラス (ISIS が提供するメモリ) のインスタンス shared_mem_isis の先頭アドレスを 0x80000000 に設定する

6: shared_mem_isis のサイズを 0x7fffffff に設定する (終端アドレス = 0xffffffff)

7: shmem がアクセスするメモリを shared_mem_isis に設定する

8: shbus に soo のポート数を通知する

9: shbus の soo 側のポートを生成
   - to_pu_port*soo 毎のポート数*soo 数
   - from_pu_port*soo 毎のポート数*soo 数

10: shmem の shbus_port と shbus の mm_port を接続する

12: soo にプロセッサ ID を割り振る

13: soo にシステム上の soo 数を通知する

14: soo が外部と通信するためのポート数を指定された数にする

16: shbus の to_pu_port と soo の incoming_port を接続する

17: shbus の from_pu_port と soo の outgoing_port を接続する

void sim_main(void)

各モジュールにクロックを入れる。パケットの送信や受信を行う記述をしている場合には、クロックが入力されることで、void initialize(unsigned int, int) において接続したモジュール間でパケットがやり取りされる。

```
     void
     isim_system::sim_main(void)
     {

                 ..............
                 ..............


1:       for (;;) {
2:           for ( unsigned int i = 0; i < soo.size(); i++ )
3:               soo[i].clock_out();
4:
5:           shbus.get_packet();
6:           shbus.send_packet();
7:           shmem.clock();
8:
9:           for ( unsigned int i = 0; i < soo.size(); i++ ) {
```

```
10:                soo[i].clock_in();

                  ..............
                  ..............

              }
          }
      }
```

サンプルコードに沿って、システムが動作する流れを示す。

1: シミュレータはプログラムの実行が終了するまで、1 ループ 1 クロックとして処理を進める

3: soo が clock_out() の処理 (ruu_commit()、ruu_writeback()、ruu_issue()) を行う

- 共有メモリへの要求がある場合には、soo の出力ポート (outgoing_port) にパケットが発行される

5: soo および shmem からパケットが発行されている場合には、shbus がそれらを受信する

6: shbus が get_packet() で受信したパケットを送信先に転送する

7: shmem が clock() の処理を行う

- shbus からパケットが送信されてきている場合には、そのパケットを受信する
- メモリアクセスを行う
- メモリアクセスが終了したら、読み出し要求に対するデータが shmem の shbus_port に発行される

10: soo が clock_in() の処理 (ruu_dispatch()、ruu_fetch()、get_data()) を行う

- shmem からの読み出しデータが転送されてきている場合には、それを受信する

以上のように、作成したプロセッサ外部のモジュールを void initialize(unsigned int, int) で繋げ、void sim_main(void) で各モジュールにクロックを入力すれば、シミュレータが動作する。

B.6.1 プロセッサから発行される要求に対する処理

ISIS-SimpleScalar が提供するプロセッサ isim_processor は、実行するプログラム中に共有メモリ領域へのアクセスがあると、共有メモリへの要求を外部出力ポートに発行する。isim_processor 外部のシステムは、isim_processor からの要求を受け取り、要求に応じた処理を行わなければならない。

B.6.1.1 isim_processor が扱うパケット

isim_processor から発行されるパケットには、以下の情報が付加されている。

```
md_addr_t addr;          //メモリアクセスアドレス
int data_size;           //メモリアクセスする際の byte 幅
int inst_id;             //命令 ID
int rw;                  //Read or Write or Reply
int puid;                //送信元プロセッサの ID
/* データ */
word_t data, data2;      //unsigned int 型
half_t data_half;        //unsigned short 型
byte_t data_byte;        //unsigned char 型
```

B.6.1.2 isim_processor から発行されるパケットの取得

isim_processor 外部のシステムは、isim_processor の外部出力ポート (outgoing_port) に発行されたパケットに記載されている情報を元に、それぞれに応じた処理を行う必要がある。パケットに記載された情報は、次のようにすれば取得できる。isim_processor の外部出力ポート (outgoing_port) に port_A という名の外部システムのポートが接続されているとする。

```
if ( port_A.have_packet() && port_A.inst_id() != -1 ) {
    int inst_id      = port_A.inst_id();
    md_addr_t addr   = port_A.addr();
    word_t data      = port_A.data();
    word_t data2     = port_A.data2();
    half_t data_half = port_A.data_half();
    byte_t data_byte = port_A.data_byte();
    int data_size    = port_A.data_size();
    int rw           = port_A.rw();
    int puid         = port_A.puid();
    port_A.reset_packet();

    /* パケットに応じた処理を以下に記述 */

            ..............
            ..............

}
```

パケットの初期状態における inst_id の値は-1 であり、isim_processor から発行されるパケットの inst_id には正の値が入るため、パケットの情報取得の条件に port_A.inst_id() != -1 を入れることによって、無駄な処理を省くことができる。また、情報を取得した後、

```
port_A.reset_packet();
```

とすることによって、パケットは初期化され、isim_processor は新しいパケットを外部出力ポート (outgoing_port) に発行できるようになる。

B.6.1.3 isim_processor へのパケットの送信

メモリからの読み出しデータを isim_processor に返したい場合には、次のようにすればプロセッサに送信できる。isim_processor の入力ポート (incoming_port) に port_B という名の外部システムのポートが接続されているとする。

```
if ( !port_B.have_packet()
     || ( port_B.have_packet()
          && port_B.inst_id() == -1 ) ) {
    port_B.set_inst_id(inst_id);
    port_B.set_addr(addr);
    port_B.set_data(data);
    port_B.set_data2(data2);
    port_B.set_data_half(data_half);
    port_B.set_data_byte(data_byte);
    port_B.set_data_size(data_size);
    port_B.set_reply();
    port_B.set_puid(puid);
}
```

どの要求に対する読み出しデータなのかを isim_processor に知らせるため、puid、inst_id には受信したときと同じ値を代入する必要がある。また、読み出しデータの送信要求であることを set_reply() によって通知する。if 文の条件に port_B.inst_id() == -1 を付加することによって、情報取得されていないパケットへの上書きを避けることができる。

B.6.1.4 外部システムがすべき具体的な処理内容

isim_processor からの要求に対して外部システムが行うべき処理について、パケットに記された情報別に説明する。

- 1byte データの読み出し要求 (rw==0、data_size==1) が発行されたとき
  - addr 番地からデータを読み出す
  - 読み出したデータを data_byte に代入して isim_processor に返す
- 2byte データの読み出し要求 (rw==0、data_size==2) が発行されたとき
  - addr 番地からデータを読み出す
  - 読み出したデータを data_half に代入して isim_processor に返す
- 4byte データの読み出し要求 (rw==0、data_size==4) が発行されたとき
  - Fetch and Dec を行う領域へのアクセス (addr >= FAD_MEM_TOP && addr <= FAD_MEM_BOTTOM)

        * addr 番地からデータを読み出す
        * 読み出したデータから 1 引いた値を addr 番地に書き込む
        * 読み出したデータを data に代入して isim_processor に返す
    - Test and Set を行う領域へのアクセス (addr >= TAS_MEM_TOP && addr <= TAS_MEM_BOTTOM)
        * addr 番地からデータを読み出す
        * 読み出したデータが 1 ならば addr 番地に 0 を書き込み、data に 0 を代入して isim_processor に返す
        * 読み出したデータが 0 ならば data に 1 を代入して isim_processor に返す
    - 通常のメモリアクセスを行う領域へのアクセス (上記条件以外の addr)
        * addr 番地からデータを読み出す
        * 読み出したデータを data に代入して isim_processor に返す

- 8byte データの読み出し要求 (rw==0、data_size==8) が発行されたとき
    - addr 番地と addr+4 番地からデータを読み出す
    - addr 番地から読み出したデータを data に、addr+4 番地から読み出したデータを data2 に代入して isim_processor に返す
- 1byte データの書き込み要求 (rw==1、data_size==1) が発行されたとき
    - addr 番地に data_byte の値を書き込む
- 2byte データの書き込み要求 (rw==1、data_size==2) が発行されたとき
    - addr 番地に data_half の値を書き込む
- 4byte データの書き込み要求 (rw==1、data_size==4) が発行されたとき
    - addr 番地に data の値を書き込む
- 8byte データの書き込み要求 (rw==1、data_size==8) が発行されたとき
    - addr 番地に data の値を、addr+4 番地に data2 の値を書き込む

以上の処理さえ正確に行えばよい。そのため、プロセッサ外部のネットワーク部やメモリシステムなどの記述は、ISIS のライブラリを使用しなくてもよい。

B.7 コマンドラインオプション

ISIS-SimpleScalar が提供するコマンドラインオプションをまとめる。

`-config [file]`
: [file] に記述されたコマンドラインオプションを読み込む

`-dumpconfig [file]`
: 指定したコマンドラインオプションを [file] に出力する

`-p [number]`
システムに搭載するプロセッサ数を指定する

`-redir:prog [file]`
実行結果を `[file]` に出力する

`-nice`
シミュレータプロセスの優先度を指定する

`-max:inst [number]`
シミュレータが `[number]` 個の命令を実行した時点で、実行を終了する

`-fetch:ifqsize [number]`
IFQ(fetch された命令を保持する) のサイズを指定する

`-fetch:speed`
命令 fetch 幅を指定する

`-bpred {nottaken|taken|bimod|2lev|comb}`
branch predictor を指定する

`nottaken` – 常に分岐しないと予測

`taken` – 常に分岐すると予測

`bimod` – 2 ビットカウンタによる予測

`2lev` – 2 レベル予測機構による予測

`comb` – bimodal predictor と 2-level predictor を結合した branch predictor

`-bpred:bimod [number]`
テーブルのサイズを指定する (※ `-bpred bimod` を指定した時のみ有効)

`-bpred:2lev [l1size] [l2size] [hist_size] [xor]`
2-level predictor の設定を行う (※ `-bpred 2lev` を指定した時のみ有効)

`l1size` – L1 テーブル (分岐履歴テーブル) のサイズ

`l2size` – L2 テーブル (2 ビットカウンタ) のサイズ

`hist_list` – 分岐履歴の幅

`xor` – L2 テーブルにおいて、履歴とアドレスの XOR を行うかどうか

`-bpred:comb [number]`
bimodal predictor と 2-level predictor で共有するテーブルのサイズを指定する

`-bpred:btb [num_sets] [assoc]`
Branch Target Buffer のの設定を行う

`num_sets` – セット数

`assoc` – 連想度

`-decode:width [number]`
命令 decode 幅を指定する

`-issue:width [number]`
命令 issue 幅を指定する

`-commit:width [number]`
命令 commit 幅を指定する

`-ruu:size [number]`
RUU(命令が fetch されてから commit されるまで命令を保持する) のサイズを指定する

`-lsq:size [number]`
LSQ(Load 命令/Store 命令を保持する) のサイズを指定する

`-cache:dl1 {<config|none>}`
L1 ローカルデータキャッシュを設定する

`<config> = <name>:<nsets>:<bsize>:<assoc>:<repl>`

`<name>` – キャッシュの名前 (識別子)

`<nsets>` – セット数

`<bsize>` – 大きさ (KByte)

`<assoc>` – 連想度

`<repl>` – 置換アルゴリズム (l:LRU,f:FIFO,r:random)

`-cache:dl1lat [number]`
L1 ローカルデータキャッシュにヒットしたときのレイテンシを指定する

`-cache:dl2 {<config|none>}`
L2 ローカルデータキャッシュを設定する

`-cache:dl2lat [number]`
L2 ローカルデータキャッシュにヒットしたときのレイテンシを指定する

`-cache:il1 {<config|none>}`
L1 ローカル命令キャッシュを設定する

`-cache:il1lat [number]`
L1 ローカル命令キャッシュにヒットしたときのレイテンシを指定する

`-cache:il2 {<config|none>}`
L2 ローカル命令キャッシュを設定する

`-cache:il2lat [number]`
L2 ローカル命令キャッシュにヒットしたときのレイテンシを指定する

`-cache:icompress`
64 ビットの命令アドレスを 32 ビットに置き替える

`-mem:lat [fist] [next]`
ローカルメモリへのアクセスレイテンシを指定する

`-mem:width [number]`
メモリのバス幅 (bytes)

`-tlb:itlb`
命令 TLB(アドレス変換バッファ) を設定する

`-tlb:dtlb`
データ TLB を設定する

`-tlb:lat [number]`
TLB ミス時のレイテンシを指定する

`-res:ialu [number]`
整数 ALU の数を指定する

`-res:imult [number]`
整数乗除算器の数を指定する

`-res:memport [number]`
ローカルメモリアクセスポートの数を指定する

`-res:fpalu [number]`
浮動小数点 ALU の数を指定する

`-res:fpmult [number]`
浮動小数点乗除算器の数を指定する

`-o [number]`
プロセッサが外部のシステムと通信するためのポート数を指定する

`-cache:flush`
特定部分統計データ取得時に限ったキャッシュに関する統計データを取得する (B.5.3 項 system_clock.c 参照)

- `0xd0000000` 番地へのアクセスがあったら特定部分統計データ取得用のキャッシュに切り替える
- `0xd0000100` 番地へのメモリアクセスがあったら、切り替える前のキャッシュに戻す

`-output [number]`
統計データの出力方法を指定する

0 – 統計データを出力しない
1 – 特定部分統計データのみ出力する
2 – 全時間の統計データのみ出力する
3 – 全時間の統計データと特定部分統計データの両方を出力する

B.8 デバッグ出力

ISIS-SimpleScalar が提供するプロセッサ `isim_processor` のデバッグをサポートする出力として、次のものが用意してある。

B.8.1 処理直前、直後の出力

各命令に関わる処理がなされるとき、その命令のプログラムカウンタ、命令名とともに、実行直前のオペランドの内容や実行結果、メモリアクセスアドレスなどを出力する。instruction.cc 内に

```
#define DEBUG_ISIM_PROCESSOR
```

inst_simple.cc 内に

```
#define OBSL_DEBUG_ISIM_PROCESSOR
```

と記述し、インストールすれば良い。

B.8.2 パイプライン内命令の出力

各命令がパイプラインの Stage を移るときに、その命令のプログラムカウンタ、命令名、命令 ID、使用するレジスタ番号を出力する。isim_processor.cc 内に

```
#define COUT_PIPELINE
```

と記述し、インストールすれば良い。