超並列計算機 JUMP-1 の
キャッシュシステムの設計検証

舟本 一久

平成 8 年 2 月 15 日

内容概況

今日の計算機に必要不可欠なものとなりつつあるメモリアクセスバッファリング技術は、メモリアクセスをパイプライン処理し並列実行することで、メモリアクセスのレイテンシを隠蔽する技術である。現在我々が開発中の超並列計算機 JUMP-1 もいくつかのメモリアクセスバッファリングを実装する。

メモリアクセスバッファリングを実装する計算機では、そのパイプライン処理によりシステム内に同時に複数のメモリアクセスが存在するため複雑な制御が必要となる。

このような対象に対して、設計上クリティカルな入力系列を挙げ、そしてそれに対する正しい出力系列を机上で計算し、それらを元にシミュレーションすることで設計検証を行うのは、非常に困難である。

そこで、本稿で述べる手法では、人手でそのような系列を求めず、適当な入力に対する応答が、JUMP-1 キャッシュシステムと要素プロセッサの命令パイプラインとの間のインタフェースを規定するメモリモデルを満たしていることを確かめることで、設計検証を行う。

JUMP-1 のメモリモデルは、Weak Consistency と呼ばれるクラスに属する。このクラスに属するメモリモデルでは、メモリアクセスは、命令パイプラインにより発行された順序どおりにメモリに対しても実行されるとは限らない。つまり、メモリの状態は非決定的である。

普通、シミュレーションによる設計検証では、対象にテストベクトルを与えてシミュレーションを行い、得られた出力と予め用意しておいた期待値とを比較するという手法がとられる。

このような手法を、Weak Consistency をサポートする並列計算機に適用しようとすれば、メモリモデルの許す非決定的な出力を全て期待値として用意しなくてはならない。しかし、入力系列に応じて増大していく、この非決定的な出力の全てを静的に用意しておくことは、不可能に近い。

そこで、本稿で述べる設計検証法では、与えられた入力に対しメモリモデルを満たす出力全てを動的に用意するモジュール、'参照モデル' を用意する。そして、JUMP-1 キャッシュシステムと参照モデルに同一の入力を与え、JUMP-1 キャッシュシステムから返ってきた出力と、参照モデルが用意した正しい出力を比較することで、キャッシュシステムがメモリモデルを満たしているかを判断する。

参照モデルをモデリングする際に注意せねばならないことは、いかに効率良く非決定性を表現するかである。非決定的な状況が生ずる原因は、あるメモリアクセスの完了が不明の段階で、次のメモリアクセスを発行することにある。従って、メモリアクセスの完了を知る方法が重要となってくる。

本稿で述べる手法では、リクエスト 1 つ 1 つではなく、PEND 信号という信号により定義される '世代' と呼ばれる期間ごとにストア・リクエストを集合的に取り扱い、メモリアクセスの完了を評価する。この PEND 信号とは、それ以前に発行された全てのストア・リクエストの終了を保証するものである。

Weak Consistency というクラスに属するメモリモデルをサポートしている一般的な共有メモリ型マシンでは、同期のため、何らかの形で以前に発行されたリクエストの完了を知る機構を備えているはずである。つまり、PEND 信号のようなものを出力信号として持たないマシンでも、この種の情報を引き出すことは難しいことではなく、従って、本稿で述べる手法は一般性のあるものだといえる。

Design Verification
for the Cache System of the Massively Parallel Computer JUMP-1

FUNAMOTO Kazuhisa

Abstract

The memory-access-buffering technique which is going to be essential for today's computer systems efficiently hides the memory access latency by executing memory accesses by parallel by the memory-access-pipelining. And that technique is implemented in some place on JUMP-1, a distributed shared memory massively parallel computer that we are developing now.

A computer on that the memory-access-buffering technique is implemented needs complicated controls, bacause it deals with the memory-access by pipelining and so there are some memory-accesses at the same time on that systems.

To such system, it is almost impossible to list input sequences that are critical on the design and its expected output sequences. And it is almost impossible to simulate the design by these sequences.

So this report explains a design verification method to verify the cache system of JUMP-1 whether the cache system satisfies JUMP-1's memory model.

JUMP-1's memory model belongs to the weak consistency class. Memory models belonging to this class do not always guarantee the order of memory accesses issued by an instruction pipeline. Therefore it is impossible to define a certain order of memory access completion on a computer that supports weak consistency.

As usual, the ordinal verification method are used which verifies the system to be tested by comparing sequences obtained by giving test vectors to the system with expected outputsequences prepared in advance.

To apply this method to a parallel computer that supports the memory model which belongs to the weak consistency class, it is necessary to prepare all expected output sequences that allow by the memor model. But it is almost impossible to do because that sequences increase in number in propotion to the input sequences.

In order to deal with this ambiguousness, we have introduced the idea of **reference moel** that dynamically calculates all acceptable output sequences from input sequences. In this verification,the same input sequences are feeded to both JUMP-1 cache system and the reference model. Then, a set of acceptable output sequences calculated by reference model are compared with actual output sequences.

An important point in designing the reference model is how to deal with the ambiguous situations.

In order to deal with such situations, our reference model treats store requests not individually but as a collection of them which are issued during the period called 'generation' separated by PEND signal, which guarantees the completion of all store requests that are issued before.

Even if the processor does not have a signal like PEND, it is not so hard to get such information, because the shared memory parallel computer that supports a memory model which belongs to the weak consistency class should have a synchronization mechanism to know the completion of store requests issued before. Therefore the verification method described in this paper can be applied to the wide range of the shared memory system.

1　はじめに

メモリアクセスのバッファリング技術は、メモリアクセスのレイテンシを隠蔽するための工夫として、今日の計算機に一般的に実装されつつある。メモリアクセス・バッファリングとは、メモリアクセスを複数のステージに分割しパイプライン処理を行うことで、メモリアクセスを並列実行する技術である。現在我々は、共有メモリ型の超並列計算機 **JUMP-1** を開発中であるが、このJUMP-1のキャッシュ–メモリシステム (以下単にJUMP-1キャッシュシステム) も、いくつかのメモリアクセスバッファリング技法を実装する [1]。

メモリアクセス・バッファリングを実装する計算機では、そのパイプライン処理によりシステム内に同時に複数のメモリアクセスが存在するため複雑な制御が必要となり、その設計と検証が非常に困難になることが問題である。これらの計算機では、原則的に複数のメモリアクセスを並列に処理するが、それらのメモリアクセスが同一アドレスに対するものであった場合には、メモリアクセスのセマンティクスを守るために、逐次化して処理しなければならない。例えば、あるアドレスに対する操作が実行中である時に同一アドレスに対する操作が要求された場合には、その操作を一旦中断させ、先行する操作が完了した後に、再試行させる必要がある。このようなクリティカルな状況の場合の数は、要素プロセッサの台数に対して指数関数的に増加する。

このように複雑な対象に対して、上述のような設計上クリティカルな入力系列を (できれば) 全て挙げ、それに対する正しい出力系列を机上で求めることはほぼ不可能である。そこで本稿で述べる手法では、クリティカルな入力系列とそれに対する正しい出力系列を人手で求めるのではなく、JUMP-1のキャッシュシステムに対する適当な入力とそれに対する出力の関係がJUMP-1のメモリモデルを満たしていることを確かめることで、検証を行う。

メモリモデルとは、プログラムから観測されるメモリの振舞いを定義した仕様である。JUMP-1のキャッシュシステムがメモリモデルを満たすことが示されれば、JUMP-1のために書かれたプログラムが、メモリアクセスに関して正しく動作することが保証される。

本稿で述べる設計検証法の概略は、次のとおりである。即ち、検証の対象であるJUMP-1キャッシュシステムの他に、JUMP-1のメモリモデルをモデル化した参照モデルと呼ぶモジュールを用意し、JUMP-1キャッシュシステムと、参照モデルに同一の入力を与える。両者の応答を比較することで、JUMP-1キャッシュシステムがJUMP-1メモリモデルを満たしているかを判断する。

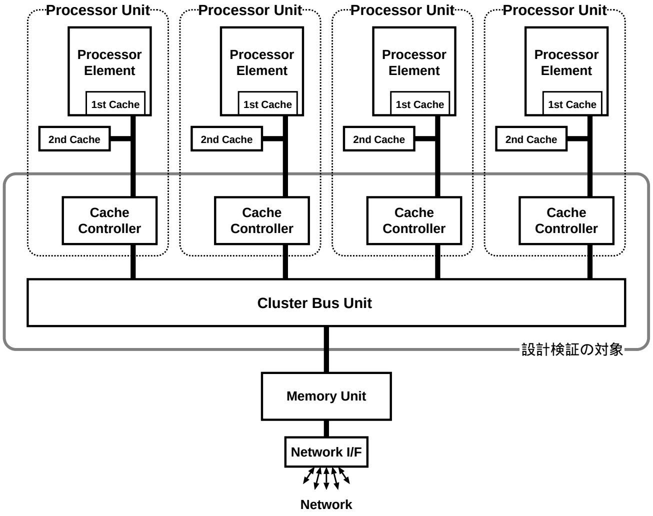

図 1: JUMP-1 のクラスタの構成

一般に共有メモリ型の並列計算機では、メモリアクセスに関して非決定的な状況が起こるため、メモリモデルにしたがった "正しい" 応答は1とおりではない。従って、参照モデルは返ってきうる正しい応答を全て用意できなければならない。

以下本稿では、第 2章で、設計検証の対象であるJUMP-1について述べ、第3章で検証手法の概略について説明する。今回の手法では、要素プロセッサと参照モデルのモデリングが特に重要である。第4章で要素プロセッサのモデリング、第5章で参照モデルのモデリングについて述べる。

2　設計検証の対象

本章では、検証の対象であるJUMP-1クラスタと、そのメモリモデルについて述べる。

2.1　JUMP-1 の構成

JUMP-1 はクラスタ構造を採用している。図 1に JUMP-1 のクラスタの構成を示す。クラスタは主に、4つのプロセッサ・ユニット、メモリ・ユニット、ならびに他クラスタとのインタフェースを司るでネットワーク・インターフェイス構成される [2][3]。プロセッサ・ユニットとメモリ・ユニットはクラスタバス・ユニット [4] によって接続される。プロセッサ・ユニットは、要素プロセッサ (以下、PE と呼ぶ)、2 次キャッシュ、および、キャッシュ・コントローラからなる。PE には、1 次キャッシュを内蔵する市販の高性能 RISC プロセッサ、SuperSPARC を用いる [5][6]。JUMP-1 のキャッシュ・システムの仕様を表 1にまとめる。

現在 私の研究室では、キャッシュ・コントローラとクラスタバス・ユニットの設計を行っている。したがって、その部分が本稿での設計検証の対象である。

表 1: キャッシュ・システムの仕様

キャッシュ	容量 (KB)	ブロックサイズ (KB)	連想度	書き戻し方式	コヒーレンス制御
1 次命令	20	32	5	WT	I
1 次データ	16	32	4	WT	I
2 次 命令/データ	10^3	32	1^a	WB^b	I/U

WT : ライト・スルー WB : ライト・バック I : 無効化 U : 更新

---

[a] Direct Map

[b] W・T も可

2.2 JUMP-1 のメモリモデル

本節では、JUMP-1 のメモリモデルについて述べる。以下では、まずメモリモデルの一般的な事柄に関して述べ、その後で JUMP-1 メモリモデルの説明を行う。

2.2.1 メモリモデル

メモリモデル [6] は、プログラムから観測されるメモリの振舞いを定義した仕様である。形式的に定義されたメモリモデルは、プログラムとハードウェア・インプリメンテーションのインターフェイスを厳密に定義する。メモリモデルを設定する目的は、プログラムとハードウェア・インプリメンテーションを独立に開発できるようにすることである。ハードウェア設計者は、メモリモデルを満たすようにハードウェアを設計すれば良く、メモリモデルの形式的記述によって、ハードウェアのクリティカルな部分が必要な性質を満足するかどうか推論することができる。

さて、一般に共有メモリ型並列計算機のメモリの状態は非決定的であり、メモリモデルも非決定的状況の発生を許すことが普通である。そのような非決定的状況においては、メモリモデルは、あるインプリメンテーションにおけるプログラムの実行結果が、メモリモデルに従って定義された概念的なマシン・モデルでそのプログラムを実行した場合に生じ得る結果の何れかになることを要求する。

2.2.2 JUMP-1 メモリモデルの定義

メモリモデルは、メモリ操作間の時相論理として形式的に表現することができるが、ここでは直観的に JUMP-1 のメモリモデルを説明する。なお、形式的定義については、本稿付録を参照されたい。

JUMP-1 のメモリモデルは、メモリを図 2のような N ポート・デバイスとしてモデル化する。この N は PE の命令パイプラインの数、即ち PE の数である。

命令パイプラインは、ポートを介してメモリ操作を開始する。各ポートにはストアバッファが用意され、命令パイプラインによるストア・リクエストは、一旦ここにストアされる。

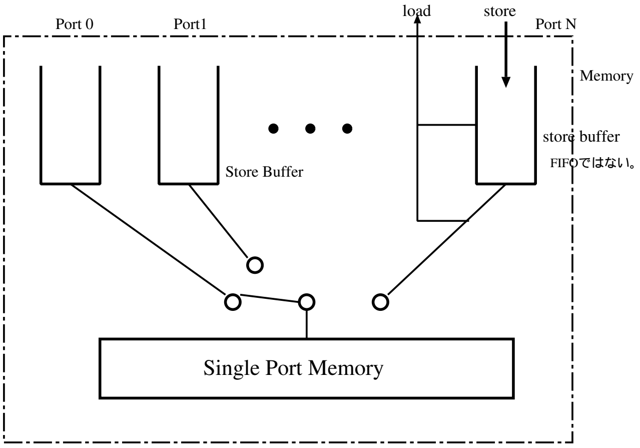

図 2: メモリモデルの直観的定義

メモリアクセスが終了するごとに、スイッチがシングルポートメモリを N 個のポートの中で任意のポートに接続する。このスイッチがメモリアクセスがメモリに対して実行される順序を決定するが、スイッチが 1 つのポートから他のポートへ接続を変更する順序は非決定的である。

JUMP-1 のメモリモデルは、Weak Consistency というクラスに属している。Weak Consistency に属するメモリモデルに従う計算機上では、メモリアクセスは、必ずしもプログラムに記述されたとおりには実行されない。つまり、ポートごとのストアバッファは FIFO ではなく、ストア・リクエストがプログラムに記述された順序でシングルポートメモリに実行されるという保証はない。ただし、1 つの命令パイプラインが発行したストア・リクエストは、当該命令パイプラインの発行したストア・バリア命令により分離された場合に限り、メモリに対してプログラムに記述された順序で実行されることが保証される。

一方、ロード・リクエストは、シングルポートメモリ、または、自らのストアバッファを読むことで実行される。これは、他の命令パイプラインのストアの結果にはシングルポートメモリを介してしか読めないが、自分のものは常に最新の結果が読めることを意味する。

2.3 検証の対象事項

今回の設計検証の対象である JUMP-1 のキャッシュ・コントローラとクラスタバス・ユニットは、JUMP-1 クラスタのキャッシュ・システムの中心をなす。今回の検証では、これらが JUMP-1 のメモリモデルを満たすことを確かめるのだが、プロセッサからのメモリアクセスは、実際には、キャッシュのコヒーレンス制御として実現さ

れる。したがって具体的には、主に以下の事項を検証することになる．

1. 2次キャッシュを、正しく無効化、あるいは，更新するか。
2. SuperSPARCの内蔵1次キャッシュを、正しく無効化するか。
3. 2次キャッシュの置き換え操作、および、ライトバック操作が正しく行われるか。
4. 同一アドレス(キャッシュ・ブロック)へのクリティカルなアクセスが正しく実行されるか。

3 検証手法

本章では、設計検証の方針、及びテストベンチについて述べる。また、3.3では，対象への入力について述べる。

3.1 設計検証の方針

通常、シミュレーションによる設計検証では、対象にテストベクトルを与えてシミュレーションを行い、得られた出力と予め用意しておいた期待値とを比較するという手法がとられることが多い。

しかし、この手法をJUMP-1のキャッシュシステムに適用することは難しい。今回は、JUMP-1キャッシュシステムがJUMP-1のメモリモデルを満たしていることを確かめるのだが、このメモリモデルの許す非決定的な出力を全て期待値として用意しておく必要があるからである。期待値の量は、入力に対して指数関数的に増加する。

そこで本検証では、検証対象に対してテストベンチを用いる手法をとる。この手法では、検証の対象とは別にテストベンチと呼ばれるモジュールを設計し、検証対象とテストベンチを接続した全体に対してシミュレーションを実行する。期待値は予め用意されるのではなく、シミュレーションの実行時にテストベンチにおいて動的に計算する。

3.2 テストベンチ

各モジュールのモデリングとシミュレーションには、Mentor Graphics社のIDEA System V8を用いた。このシステムでは、ハードウェア記述言語VHDL [7] によってモジュールの動作を記述し、シミュレーションすることができる。VHDLは、動作 (behavior) レベルなどの抽象度の高い記述をサポートする。

テストベンチはVHDLを用いて抽象的に記述する。テストベンチは物理的なハードウェアに直接結び付くものではないので、タイミングなどのハードウェアとしての動作イメージを意識して、詳細に記述する必要はない．テストベンチでのバグの発生を抑えるために、むしろモデルを簡略するなどして，簡潔に記述することが望ましい。テストベンチにバグが存在すると、間違った設計を“正しい”と判断してしまう可能性すらある。また、そうすることによって、シミュレーションの高速化を図ることもできる。

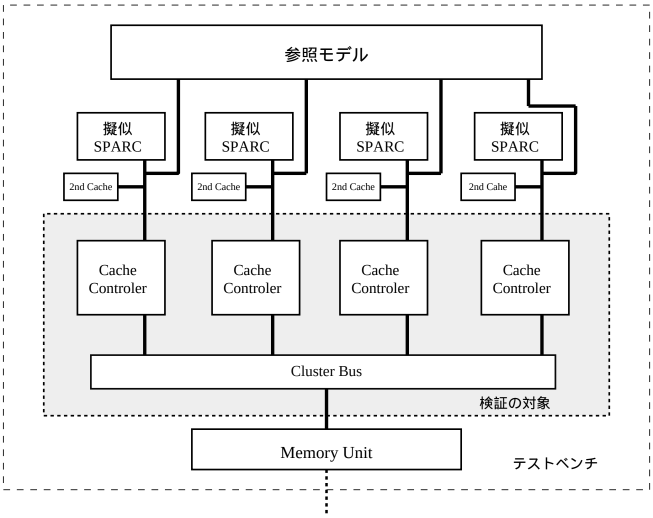

図 3: テストベンチ

図3に検証系の全体構成を示す。以下、図中のテストベンチを構成するモジュールについて説明する。

擬似 SPARC SuperSPARCをモデリングしたモジュール。JUMP-1キャッシュシステム、および、参照モデルに与える入力を動的に生成する。内部に、1次キャッシュに相当するモジュールを持つ。

参照モデル 疑似SPARCの発行するリクエストを入力として受け、メモリモデルを満たす全ての出力を計算する。

メモリ・ユニット、2次キャッシュ それぞれ、JUMP-1のメモリ・ユニットと2次キャッシュのデータ・アレイをモデリングする。

これらのモジュールを検証対象に接続し、シミュレーションを実行する。擬似SPARCは参照モデルと検証対象に同一の入力を与える。前述したように、メモリの非決定性により、メモリモデルが許す正しい応答は複数存在し得るが、参照モデルはそのすべての可能性を考慮して、1つ、または、複数の応答を用意する。検証対象から応答されるリプライ・データが、参照モデルが用意したデータと一致していれば、検証対象はその入力に対してメモリモデルを満たしているといえる。

なお、擬似SPARC、参照モデルのモデリングに関しては、それぞれ、4章、5章において詳述する。

3.3 検証対象への入力の生成

本節では、擬似 SPARC が生成する検証対象への入力について説明する。

今回の検証手法においては、ある入力系列と、それに対して応答された出力系列の関係のみから検証を行う。したがって検証対象への入力であるメモリアクセスの、

1. 種類 (ロード、または、ストア)
2. アドレス、および、
3. ストア・データ

に関しては、ランダムに生成して構わない。ただし、検証を行いたいクリティカルな系列の生起確率が十分に高くなるように、アクセスを行うアドレス空間を狭くしてやる必要がある。また、2 次キャッシュの置換え操作、書き戻し操作が発生する必要があるので、表 1に示した 2 次キャッシュの仕様を考慮して、図 4 において網のかかった 8 ブロックに対してアクセスを行うこととする。

また、理由は第 5 章にて詳述するが、正しいと思われるリプライの量を減らしていくため、ストアの際の書き込みデータは、以下に述べるルールにしたがったフォーマットである。

- 上位 2 ビットを PE の ID に従って決定する。即ち PE0 → 00、PE1 → 01、PE2 → 10、PE3 → 11。
- 下位 6 ビットに、(GC mod 64) を書き込む。

4 擬似 SPARC のモデリング

擬似 SPARC は、対象と参照モデルに対し入力を与えるモジュールである。また、内部に持つ 1 次キャッシュは、キャッシュシステムの一部である。

本章では、この擬似 SPARC のモデリングについて述べる。まず 4.1節で、実際の SuperSPARC について述べ、続く 4.2で、擬似 SPARC のモデリングについて説明する。

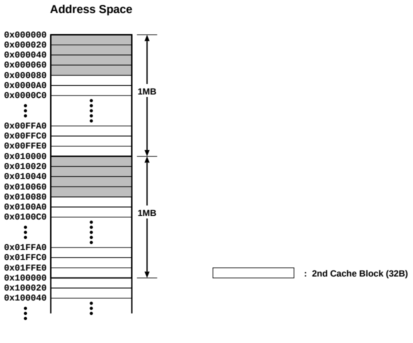

図 4: アクセスするアドレス空間

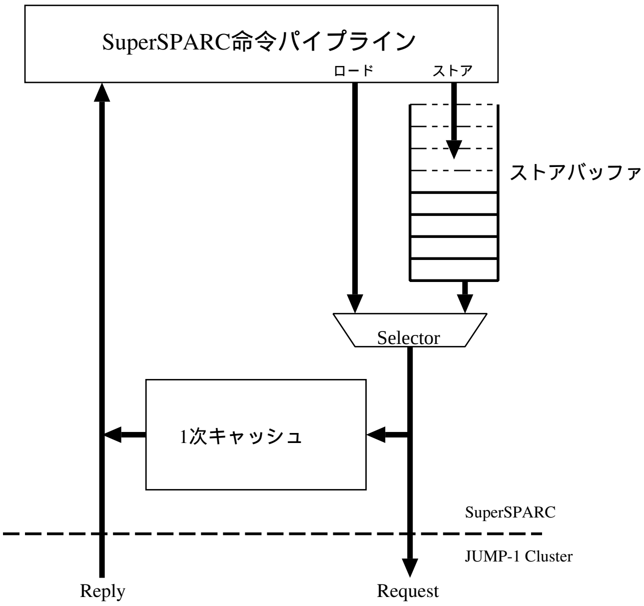

図 5: SuperSPARC の構成

4.1 SuperSPARC の構成

ここでは PE として用いる SuperSPARC の 1 次キャッシュ、ストアバッファを中心に説明する [5]。

図 5は、SuperSPARC の構成を簡単に示したものである。SuperSPARC は、内部に 1 次キャッシュ、ストアバッファを持つ。以下に、それらについての説明をする。

ストアバッファ SuperSPARC は長さ 8 のストアバッファを持ち、命令パイプラインの発行するストア・リクエストは一旦ここにストアされる。

通常、ストア・リクエストよりロードリクエストが優先される。しかし、ストアバッファ内に実行しようとするロードリクエストと同一アドレスに対するストア・リクエストが存在する場合には、ストア・リクエストが優先される。

1 次キャッシュ SuperSPARC は、4WAY の 1 次キャッシュを持つ。ストアスルー方式を採り、アロケーションポリシはライトノーアロケートである。

キャッシュシステムに対して発行する命令 命令パイプラインが、JUMP-1 キャッシュシステムに対して発行する命令は以下の 3 種である。

- ロード・リクエスト
- ストア・リクエスト
- ストア・バリア命令
  なお、ストア・バリア命令は、SuperSPARC チップ外部により実行途中のストアがないと知らされるま

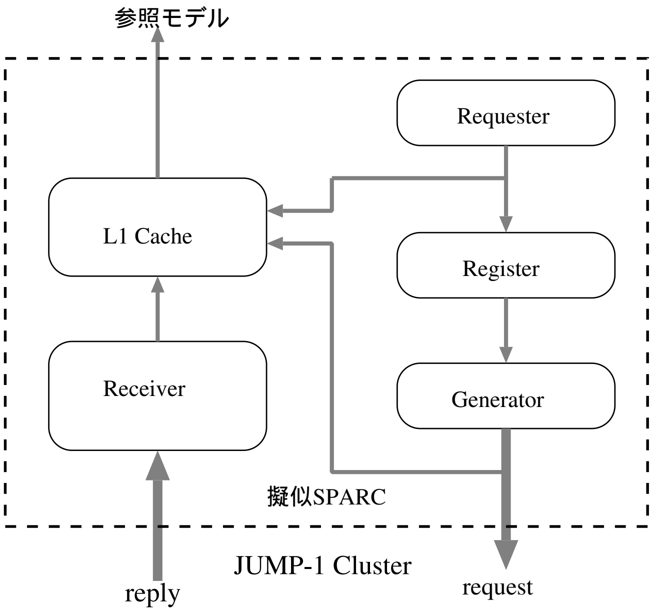

図 6: 擬似 SPARC の構成

でパイプラインをブロックすることで実現されるもので、厳密にいえば、外部に対してリクエストを発行するものではない。

4.2 擬似 SPARC の構成

本節では、SuperSPARC をモデリングしたモジュール ‘擬似 SPARC’ について説明する。

先述したように、メモリモデルは SuperSPARC の命令パイプラインとその外部、即ち JUMP-1 キャッシュシステムの間のインタフェースを規定するものである。

つまり、1 次キャッシュなどの、SuperSPARC の命令パイプライン外部は、メモリモデルによって規定される範囲である。従って、JUMP-1 クラスタがメモリモデルを満たしているか検証するためには、メモリモデルを正しく満たすように擬似 SPARC をモデリングせねばならない。

特に、1 次キャッシュに関しては、チップ外部からインバリデート・リクエストを受けるため、この操作が正しく確認できる仕様でなければならない。

検証用にモデル化した擬似 SPARC を図 6に示す。各部について説明する。

Requester ランダムにリクエストを生成し Register に送る。SuperSPARC の命令パイプラインとストアバッファに相当する。

Register 現在擬似 SPARC が行っている命令を保持する。

L1cache 擬似 SPARC が内部に持つ 1 次キャッシュである。

Generator Register で選択された命令を受け、SuperSPARC の外部バスの仕様に従って対象にリクエストを送信する。

Receiver 外からの入力を受け、L1cache を操作する。

SuperSPARC の持つストアバッファ、1 次キャッシュはキャッシュシステムの一部である。これらの、擬似 SPARC におけるモデリングについて以下に説明する。

4.2.1 ストアバッファのモデリング

実際の SuperSPARC はチップ内部にストアバッファを持つ。本来ならば擬似 SPARC にも実装せねばならないところだが、”今ロードしているアドレスと同一のアドレスに対してストアをしない”、という制限をつけることで仮想的にストアバッファを Requester に内在させることができる。この根拠を述べる。

- 命令パイプラインはロード・リクエストが完了するまでその後の命令を実行しないので、ロード・リクエスト実行中にキャッシュシステムに対して実行されるストア・リクエストは、そのロード・リクエストよりも先に命令パイプラインによって発行され、既にストアバッファに存在していたものであること。
- 前節で述べたように、実際の SuperSPARC では、通常、ロード・リクエストはストア・リクエストよりも優先的に処理される。しかし、ロードしようとしているアドレスと同一のアドレスに対するストア・リクエストがストアバッファ内部に存在した場合には、そのストア・リクエストが先に実行され、その後にロード・リクエストが実行される。

以上の 2 点よりロード・リクエストを実行中に、ロードしようとしているアドレスに対してストア・リクエストがキャッシュシステムに発行されることはない。従って、生成するリクエストに上に述べた制限を付加することで、ストアバッファを擬似 SPARC の Requester 内部に仮想的に内在させることができる。

ところで、先述したように、参照モデルは Requester の発行するリクエストを入力として受ける。すると、ストアバッファを Requester に内在させたことで、参照モデルはリクエストをプログラム・オーダではなく、ストアバッファを通過した順序で受けることになる。

メモリモデルは、命令パイプラインとキャッシュシステムの間のインタフェースを定めたものなので、本来ならば参照モデルはプログラム・オーダでリクエストを入力されねばならない。

しかし、本検証の場合では、参照モデルはストアバッファを通過した順序でリクエストを受けて問題は生じない。以下にその理由を述べる。

メモリアクセスのプログラム・オーダに対し、ストアバッファを通過した時点での順序が異なっているのは、ロード・リクエストが同一アドレスに対するストア・リクエストを追い抜かない範囲で前に移動している点のみである。

従って、2.2節で述べた、JUMP-1メモリモデルで定義されるマシンモデルに対し、リクエストをプログラム・オーダで与えた場合に返されうるリプライの集合と、ストアバッファを通過した順序で与えた場合に返されうるリプライの集合は等しい。

従って、参照モデルに対し、ストアバッファを通過した順序でリクエストを与えることに問題はない。

4.2.2 L1cacheのモデリング

1次キャッシュをモデリングして検証しなければならないことは、JUMP-1キャッシュシステムが正しく1次キャッシュをインバリデートできるか、ということである。

そこで、実際の1次キャッシュは4WAYのセットアソシアティブ方式であるが、設計を容易にするため、メモリと同じサイズを持ち、ブロック置換え操作の必要がないものにした。

なぜなら、1次キャッシュでミスヒットが生じた場合、SuperSPARCはキャッシュコントローラに対して当該ブロックのデータの転送を要求するが、その際キャッシュコントローラ側はブロック転送要求に答えるのみで、その処理は、置換えによって生じたミスヒットであろうと、キャッシュコントローラからのインバリデート・リクエストによって生じたミスヒットであろうと何ら異なるところはないからである。従って、(インバリデート・リクエストにより)1次キャッシュのミスヒットが生じるように設計されていれば、1次キャッシュでブロックの置換えが起こる必要はないのである。

1次キャッシュの置換え操作を擬似SPARC上で実現しないことで、擬似SPARCはモデリングが容易になり、かつデバッグのし易いものとなる。

5 参照モデルのモデリング

3章で述べたように、参照モデルはキャッシュシステムからの応答が正当なものであるか判断するために全ての正しいリプライを用意するモジュールである。従って、参照モデルは、メモリアクセスの非決定性を正しく表現できねばならない。

そのためには、何らかの形で過去に発行されたストア・リクエストの履歴を保持しておくことが必要となる。

また、メモリ使用量節約の観点から、非決定的状況が生じず、以降メモリに反映されている可能性の無くなったストア・リクエストの履歴を適当なタイミングで消去することが望ましい。

そこで、非決定的状況が生ずる原因は、あるPEの発行したストア・リクエストの完了が不明な時点で他のPEがリクエストを発行することにあるので、リクエストの発行と完了のタイミングを記録していけば、この非決定的状況の発生/消滅を効率良く表現することができる。

本稿では、これらのタイミングを計るための時計として、‘世代’という概念を用いる。

本章ではまず、非決定性について詳しく述べ、‘世代’の定義、‘世代’と非決定性の関係を示す。その後で、それらを利用した参照モデルのモデリングについて説明する。

5.1 非決定性

2.2節で述べたように、JUMP-1のメモリモデルにおいては、任意の2つのストア・リクエストのプログラムオーダとメモリオーダの一致が保証されるのは、ストア・バリア命令により隔てられた場合のみである。つまりそれ以外の場合、メモリに対して実行される順序は非決定的である。

そこで本節では、この非決定性をさらに明確にするためにロード、ストア、2つの非決定性を定義する。

定義 1 *2*つの*PE*の命令パイプラインが、ほぼ同じタイミングで、同一アドレスに対してストア・リクエストを発行すると、そのどちらのストア・リクエストがメモリに対して先に実行されるか決定できない。このような非決定的な状況をストアの非決定性と呼ぶ。

定義 2 ある*PE*の命令パイプラインがあるアドレスに対してストアを発行した際に、他の*PE*の命令パイプラインがそれとほぼ同じタイミングで同一アドレスに対してロードを発行すると、ストア前とストア後のどちらのデータが読めるか決定できない。このような非決定的な状況をロードの非決定性と呼ぶ。

ストアの非決定性のために、1つのアドレスに書き込まれているデータには、最大PE数分の可能性が生ずる。また、ロードの非決定性により、そのPE数分の可能性に対してさらに、現在実行中のストアによる書き込みデータと、もともとの書き込みデータ2つの可能性が生ずることになる。

なおここでは、敢えて‘ほぼ同じタイミングで’という曖昧な表現を使った。ここをいかに表現するかが、この設計検証の鍵となる。

5.2 世代

本手法では、‘世代’という概念を用いて前節で述べた2つの非決定性を表現する。

本節では、まず ‘世代’ の定義に用いる PEND 信号について説明し、その後で ‘世代’ の定義について述べる。

5.2.1　PEND 信号

PEND 信号とは、PE に対して、その PE が発行したストアで現在実行途中のものの存在を伝えるための信号である。

これは、ストアが完了した厳密な時間を知らせるものではないが、deassert された時にはそれまでにその PE が発行した全てのストアが完了していることを保証するものである。

この信号は本来、SuperSPARC がストア ・バリア命令を実行する際に利用されるものである。前述したが、命令パイプラインはストア ・バリア命令が実行されると、この PEND 信号が deassert されるまでその実行をブロックする。

PEND 信号を使用することの一般性　前述したように今日の一般的な共有メモリ型並列計算機は、Weak Consistency というクラスのメモリモデルをサポートしている。メモリ ・オーダとプログラム ・オーダが必ずしも一致しない Weak Consistency 型のマルチプロセッサマシンでは、同期操作のため以前に発行されたストア ・リクエストの完了を知る機構を備えている必要がある。

従って、JUMP-1 のようにキャッシュシステムから PE へ送信する信号という形で実装されていない他のマシンでも、PEND 信号と同等な信号を対象から引き出すことは容易であると思われる。

そういう意味では、PEND 信号を検証に用いることは一般性を欠く行為ではない。

5.2.2　世代

‘世代’ という概念を用いることで、前節で述べたロード、ストア 2 つの非決定性を表現することができる。

ここで、その ‘世代’ という概念の定義を行う。

定義 3 *PEND* 信号が一度 *deassert* されてから再び *deassert* されるまでの期間を、*1* つの世代という。

すると、ある世代において発行されたストア ・リクエストは、その世代が終るまでに完了していることが保証される。今回提案する参照モデルは、この世代をカウントすることで非決定性を表現する。

5.2.3　非決定性と世代

世代の概念を用いることで、前節で定義した 2 つの非決定的状況の発生、及び消滅を知ることができる。

以下に、2 つの非決定性と世代の関係について述べる。

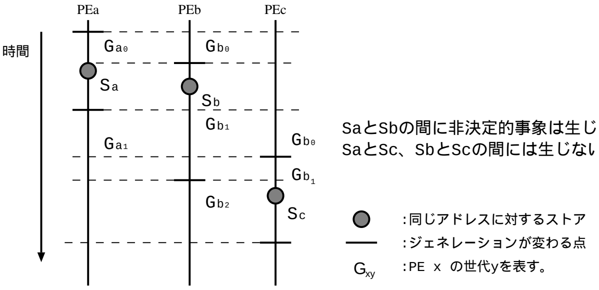

図 7: ストアの非決定性と世代

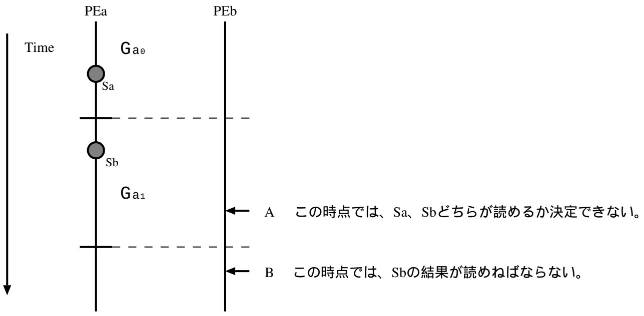

図 8: ロードの非決定性と世代

ストアの非決定性と世代　ある PE の命令パイプラインが発行したストア ・リクエストと、その PE の世代の変化した後で発行したストア ・リクエストとの間に、ストアの非決定性は存在しない。このことを図 7を用いて説明する。

図中、● は同一アドレスに対するストア ・リクエストを示し、–は、世代が変化する点を示す。下向きに時間が流れている。G_{x_y} は PEx の世代 y を意味する。

世代 G_{a_0}において、PEa の命令パイプラインがあるアドレスに対してストア ・リクエスト S_aを発行したものとする。このストア ・リクエスト S_aは、世代が G_{a_1} に変わるその完了が保証される。

この時、PEb がストア ・リクエスト S_aの完了の保証されない時点で同一アドレスに対してストア ・リクエスト S_bを発行すれば、ここに非決定的状況が生ずる。即ち、世代 G_{b_2}以降、S_aと S_bの 2 つのストア ・リクエストのどちらがメモリに反映されているか決定できない。

さらに、PEc が G_{c_1}において、図のタイミングでストア ・リクエスト S_cを発行したとする。この時、PEa は G_{a_1}、PEb は G_{b_2}に世代が変わっており、それぞれが発行したストア ・リクエスト S_a、S_bは既に完了が保証されている。

従って、S_a、S_b、S_cとの間には非決定的状況は生じない。

よって、F 点を過ぎた時点でストア S_a、S_bの履歴は破棄してよい。

ロードの非決定性と世代　同じPEが、あるアドレスに対し、再びストア・リクエストを発行すると、ロードの非決定的状況が生ずるが、そのPEの世代が変わった時点で、ロードの非決定的状況は消滅する。このことを図8を用いて説明する。

世代G_{a_0}が終った時点で、PEaによるSaのストア・リクエストの完了が保証される。この時点では、ストアの非決定性が生じなかった、つまりストア・リクエストが確実にメモリに反映できたものとする。その後、世代G_{a_1}で同一アドレスに対して、PEaがSbのストア・リクエストを発行する場合を考える。

ここで、PEbが、A点において、そのアドレスに対してロードを行うとする。この時点では、Sbのストアが完了しているかどうか分からない。ロードの非決定的状況が発生する。

一方、PEbが、B点において、同様にロードを行ったとする。この場合、世代がG_{a_1}に変わった時点で、Sbのストアの完了が保証されるので、ロードの非決定性は発生せず、必ずSbのストアの結果が読めねばならない。つまり、この場合、世代がG_{a_1}に変わった時点でSaのストアの履歴を破棄してよいといえる。

5.2.4　PEND信号の使用の是非

本検証で用いるPEND信号は、検証対象であるJUMP-1キャッシュシステムが生成するものである。このように、検証対象の生成する信号を用いて正当な設計検証が可能であるという根拠をここに述べる。

前小節で述べたように、本検証においては、正当と思われるリプライの数を減らすためだけにPEND信号を用いる。

もしそのPEND信号により減らされたリプライがクラスタより返された場合を考える。するとそれは、クラスタより返された誤ったリプライであるか、誤ったPEND信号により不当に消されたリプライであるので、どちらの場合も検証対象の誤りである。

つまり、PEND信号を用いることで検証を厳しくすることはあっても、検証を甘くすることはなく、使わないことで見つかる誤りが、使うことで見つからなくなることはない。

従って、この検証のような目的で対象が生成する信号を使うことに問題はない。

5.3　参照モデルの実装

本節では、以上述べた分析を元にモデリングした参照モデルの実装について述べる。

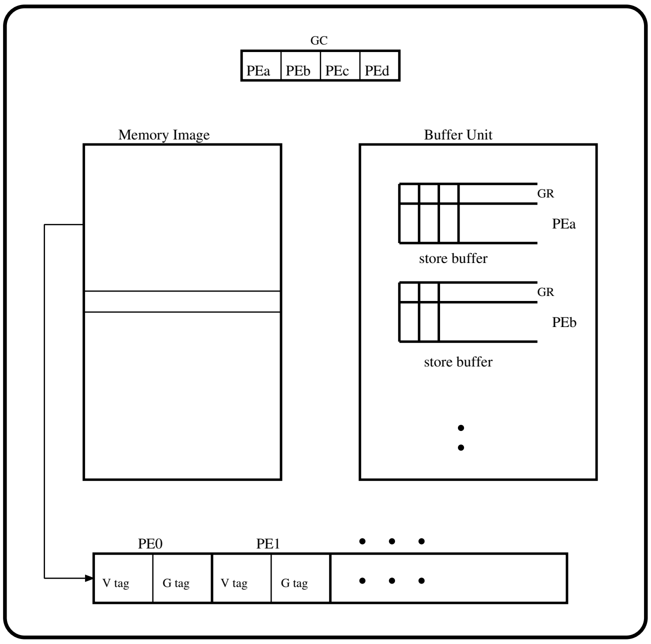

図9: 参照モデルのブロック図

5.3.1　参照モデルの外部仕様

参照モデルは、SuperSPARCの命令パイプラインに対応する擬似SPARCのRequesterが、その外部に対して発行したリクエスト信号と、JUMP-1キャッシュ・システムの出力を入力として受けとり、それらを比較し、エラーの有無を出力する。

Requesterの出力を直接受けとるのは、メモリモデルがSuperSPARCの命令パイプラインとJUMP-1キャッシュシステムのインタフェースを規定するものだからである。

5.3.2　参照モデルの構成

参照モデルのブロック図を図9に示す。参照モデルは、PE数分のバッファで構成される**Buffer Unit**とPE数分のタグで構成される**Memory Image**、及びPEごとのカウンタ**GC**からなる。

Buffer Unitは、完了の保証されていないストア・リクエストの履歴を保持するためのものである。先に述べたメモリモデルにおける、実行途中のストア・リクエストを貯めるバッファとは本質的に異なるものであることに注意されたい。

そして、Memory Imageは、既にメモリに対して実行完了が保証されているストア・リクエストの中で、現在メモリに反映されている可能性のあるものをまとめたテーブルである。

GCは、PEごとに世代を数えるカウンタである。

このような構成をとった根拠を述べる。Buffer UnitとMemory Imageの2つを用意し、ストア・リクエスト

を一旦 Buffer Unit にストアし、当該世代が終った時点、即ち GC がインクリメントされた時点で Memory Image に一斉に書き込む。こうすることで、完了の保証されていないストア ･リクエストと、完了が保証されたストア ･リクエストの最新のもの両方を保持しておくことができ、ロードの非決定性を表現できる。

そして、PE ごとの Buffer Unit と 1 つの Memory Image を用意し、Memory Image には PE ごとに、その PE が発行したストアがメモリに反映されている可能性、及びそのストアが発行された世代を示すタグを付加する。そして、Buffer Unit の内容を Memory Image に書き込む際に、当該アドレスに、その世代が始まる以前に行なわれたストアがメモリに反映されている可能性を消去する。こうすることで、PE ごとにストア ･リクエストを管理することができ、ストアの非決定性を表現できる。

以下に、それぞれについて詳しく説明する。

GC　GC は、PE ごとに設けられた、世代を数えるためのカウンタである。

PEND 信号が deassert される度にインクリメントされる。

Buffer Unit　Buffer Unit は、Store Buffer、及び Generation Regisiter(GR) で構成される。

Store Buffer は、PE ごとに用意されている、ストア ･リクエストを保持するためのバッファである。

GR は Store Buffer のエントリごとに設けられたレジスタで、当該リクエストが発行された時点での他の PE の世代を記録しておくためのものである。

Memory Image　Memory Image は、PE ごとのタグで構成されており、当該アドレスに対して、その PE の発行したストア ･リクエストが反映されている可能性があるか、そしてもしあるならばその内容を示している。

Memory Image は、V tag、G tag からなる。ストアされた内容はこれらのタグから計算できる仕様になっているので、特に書き込みデータを記録する領域は持たない。

V tag、G tag は、バイトごとに、PE 数分用意されているタグである。これらについて説明する。

V tag　各 PE によるストアが当該アドレスに反映されている可能性を示す。

G tag　V tag により示される、当該アドレスに反映されている可能性のあるストア ･リクエストが、どの世代において発行されたものであるかを示す。

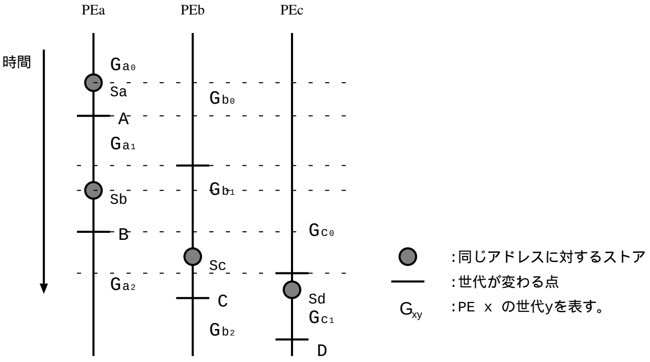

図 10: 動作の具体例 1

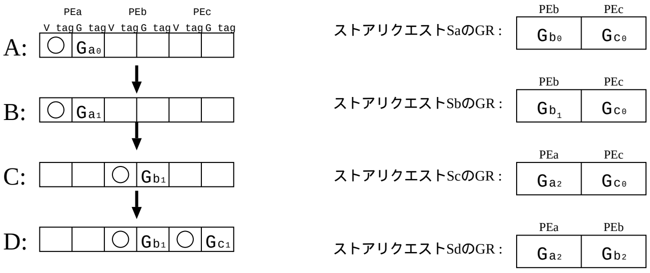

図 11: V tag、G tag、GR の様子

5.4　動作のアルゴリズム

5.4.1　リクエストに対する動作のアルゴリズム

ある PE の命令パイプラインからストア ･リクエストを受けとると、参照モデルは、他の PE の GC の値を GR に記録し、世代が終りそのストア ･リクエストの完了が保証されるまで、リクエストを Buffer Unit に保持しておく。

世代が終った時点で Buffer Unit の内容を Memory Image に書き込む。その際、そのアドレスに記載されているストア ･リクエストが、Buffer Unit 内のストア ･リクエストが発行される以前に完了したものであるならば、その履歴を消去する。

図 10に挙げたタイムチャートに対する参照モデルの動作を通して、このことを説明する。PEa の命令パイプラインが、世代 G_{a_0}において、ストア ･リクエスト Sa を発行する。この時、ロードの非決定性に対処するため、いきなり Memory Image には書き込まず、一旦 Buffer Unit に書き込む。同時に、対応する GR に他の PE の GC の値を記録する (図 11中 ストアリクエスト Sa の GR)。

ストア ･リクエスト Sa は、A 点で世代 G_{a_0}が終った時点で Memory Image に書き込まれる。この時、非決定的状況は生じず、A 点までにメモリに反映できたものとする。この時点でこのアドレスに対応する Memory Image の V tag、G tag は、図 11中 A のようになる。これは次のことを意味する。すなわち、このアドレスには PEa によるストア ･リクエストの結果が反映されており、そ

のストア・リクエストは世代 G_{a_0} において発行された。

次に、PEa が再び同じアドレスに対して、図 10のストア・リクエスト Sb を発行するとする。同様に、Buffer Unit に書き込み、同時に対応する GR に他の PE の GC の値を記録する (図 11中 ストアリクエスト Sb の GR)。

B 点で、PEa の世代 G_{a_1} が終り、それまでのストア・リクエストの完了が保証されると、PEa の Buffer Unit の内容は、Memory Image に書き込まれる。

この時、ストア・リクエスト Sa の履歴にストア・リクエスト Sb の履歴を上書きすることで、これ以降ロードできてはならないストア・リクエスト Sa の履歴を消滅することができる。

ここで、ストア・リクエスト Sa、Sb は、異なる世代において発行されているため、書き込む内容が異なることに注意されたい。すなわち、これ以降ストア・リクエスト Sa の結果が読めてはならないというを確認することができる。

PEa の Sc、PEc のストア・リクエスト Sd に対しても同様に対処する。(図 11 ストアリクエスト Sc の GR、ストアリクエスト Sd の GR)。

C 点で PEb の世代が変わると、PEb の Buffer Unit の内容を Memory Image に書き込む。この時、PEa の V tag が真になっているが、G tag に記録されている世代は GR に記録されている世代よりも古いので、ストア・リクエスト Sb は、ストア・リクエスト Sc が発行される以前に既に完了していることが分かる。そこで、PEa の V tag を偽にし、これ以降ストア・リクエスト Sb の結果がロードできてはならないようにする (図 11 C)。

そして、PEc の世代が G_{c_1} となり、ストア・リクエスト Sd の完了が保証されると、先程と同様に処理するが、ここでは、PEc の G tag とストア・リクエスト Sd の GR の値が等しいので、ストア・リクエスト Sd が発行された時点では、ストア・リクエスト Sc の完了が保証されていないことが分かる。従ってこの場合は、PEb の V tag、G tag はそのままに、PEc の V tag、G tag に書き込む (図 11 E)。

するとこの例では、D 点以降に当該アドレスに対してロードを行った場合、ストア・リクエスト Sc または Sd の結果がロードできれば良いと参照モデルは判断する。

5.4.2 リプライに対する動作のアルゴリズム

参照モデルは、JUMP-1 キャッシュシステムから出力を受けとると、2 つのことを実行する。即ち、1 つは、正しい出力を計算し実際の出力と比較すること、もう 1 つは、その出力を受けたことにより、それ以降出力されてはならない書き込みを行ったストア・リクエストの履歴を破棄することである。

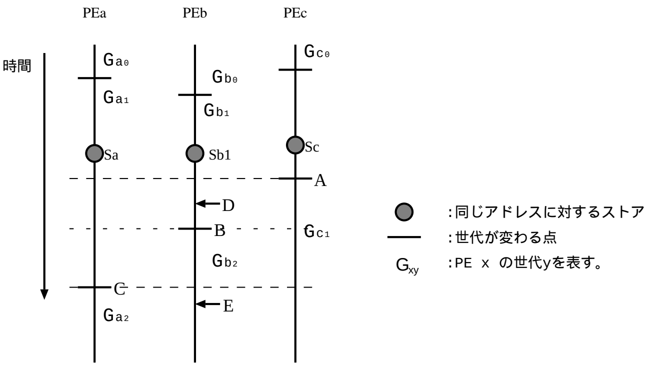

図 12: 動作の具体例 2

	PE0		PE1		PE2	
	V tag	G tag	V tag	G tag	V tag	G tag
A点でのV tag、G tagの様子					○	G_{c_1}
B点でのV tag、G tagの様子			○	G_{b_1}	○	G_{c_1}

図 13: V tag、G tag、GR の様子 2

リクエストの場合と同様に、以下に具体例を挙げながらアルゴリズムを示す。

図 12に示す状況では、Sa、Sb、Sc の同一のアドレスに対する 3 つのストア・リクエストの間にストアの非決定性が生じている。

ここで、これ以前には PEc の発行したストア・リクエストがメモリに反映していたとする。つまり、A 点以降 Sa、Sb、Sc 以外の結果はロードされないと仮定する。

これらのストア・リクエストが書き込みを行なうアドレスと同一のアドレスに対して、PEb が D、E のタイミングで、ロード・リクエストを発行する場合を考える。

A 点で、PEc の世代が G_{c_2} に変わった時点では、当該アドレスに対応する V tag、G tag は図 13上のようになる。即ち、既に完了の保証されたストア・リクエスト Sc の履歴は Memory Image 上に存在し、完了の保証されていないストア・リクエスト Sa、Sb の履歴は、Buffer Unit 上に存在する。

この時、PEb が D のタイミングでロードを行ったとする。この場合、ストア・リクエスト Sa、Sb、Sc による結果を正しい出力として用意する。

そしてもし、この時 Sa のストア・リクエストによる結果が JUMP-1 キャッシュシステムから出力されたならば、Buffer Unit 内のストア・リクエスト Sb の履歴、Memory Image 内のストア・リクエスト Sc の履歴を破棄する。

あるいはもし、ストア・リクエスト Sc による結果を出力された場合には、Buffer Unit 内のストア・リクエスト Sb の履歴を破棄する。

しかし、もしストア・リクエストSbによる結果が出力された場合には、どの履歴も破棄しない。

これらは、次に述べる根拠による。即ち、JUMP-1のメモリモデルでは、もしストアバッファ内に当該アドレスに対するストア・リクエストがあるならばその最新のものを読むことで、もし無いならば、シングルポートメモリに書かれている内容を読むことでロードは実行される。

従って、他のPEによるストア・リクエストの結果は、シングルポートメモリを介してしかロードすることはできない。

つまり、ストア・リクエストSaの結果が出力されたならば、それはメモリに反映されており、ストア・リクエストScの結果は書き換えられたことになる。

そして、ストア・リクエストScの結果が出力されたとするならば、ストア・リクエストSbがこれ以降メモリに反映されることはないことが分かる。

しかし、ストア・リクエストSbの結果が出力された場合には、シングルポートメモリの状態に関する情報は得られない。

次に、D点の代わりに、E点でロード・リクエストを発行したとする。E点では、Sa、Sb、Sc全てのストア・リクエストが完了しており、これらの履歴は全てMemory Imageに記録されている。この時のMemory Imageの状態を図13下に示す。

この場合も、Sa、Sb、Scのストア・リクエストによる結果を正しい出力として用意する。そして、この場合は、どのストア・リクエストによる結果が出力された場合にも、それ以外の履歴を破棄する。

その根拠を以下に述べる。まず、ストア・リクエストSa、Scによる結果が、出力された場合の処理の根拠は先に挙げた例の場合と同様である。そして、ストア・リクエストSbによる結果が出力された場合、このストア・リクエストはB点で既にその完了が保証されているので、これはシングルポートメモリよりロードされたものだと判断できる。従って、この場合は、Sa、Scのストア・リクエストの結果がメモリに反映される可能性はなくなるのである。

ところで、以上に述べたように、ストア・リクエストの履歴を破棄する判断を下す際、どのPEのストア・リクエストによる結果か判別できることが重要になる。そのため、先に述べたように、書き込みデータにPEのIDを記載したのである。

6 おわりに

本稿では、共有メモリ型超並列計算機JUMP-1のメモリモデルとメモリアクセスの非決定性について述べ、メモリモデルの関する設計検証手法について述べた。

一般に共有メモリ型並列計算機では、メモリアクセスが非決定的であり、予め、テストベクトルと出力の期待値ベクトルを用意するのは難しい。そのため、本稿では動的に期待値ベクトルを生成して検証する手法を示した。

動的に期待値ベクトルを作り出すためには、ストア・リクエストの履歴をとる必要があるが、効率良く検証を行なうためには適当なタイミングで過去の履歴を破棄する必要がある。

本稿で述べた手法では、ストア・リクエストをPEND信号により定義される世代で区切り集合的に扱うことで、非決定性を効果的に表現し、適切に過去の履歴を破棄することができる。

Weak Consistencyをサポートしている一般的な共有メモリ型マシンでは、同期のため、何らかの形で以前に発行されたリクエストの完了を知る必要がある。

本検証の対象であるJUMP-1の場合には、外部への信号という形で表に現れていたが、他のマシンでも、リクエストの完了に関する情報を管理するシステムが何らかの形で存在するはずであり、検証において、それを利用することは難しいことではない。従って、本稿で述べた手法は一般性のあるものだといえる。

謝辞

本研究の機会を与えて下さり、適切な御指導を賜わった富田眞治教授に深甚な謝意を表します。

また、貴重な御助言をいただいた中島浩助教授、森眞一郎助教授に深く感謝致します。

さらに、共同研究者である五島正裕氏、福島直人氏をはじめとして、日頃暖かく御鞭撻下さった京都大学工学部情報工学科富田研究室の諸兄に感謝致します。

参考文献

[1] 福島直人: 超並列計算機JUMP-1のクラスタバスにおけるメモリアクセスのバッファリング (1995).

[2] 要素プロセッサと2次キャッシュ: 文部省重点領域研究. 超並列原理に基づく情報処理基本体系, 第3回シンポジウム予稿集 (1993).

[3] 相互結合網とルータ: 文部省重点領域研究. 超並列原理に基づく情報処理基本体系, 第3回シンポジウム予稿集 (1993).

[4] Hiraki, K.: JUMP-1共通バス仕様書 ver 4.0 (1995).

[5] SPARC Technology Business: *SuperSPARC Multi-Cache Controller User's Manual* (1994).

[6] SPARC International Inc 相越克久, 田中長光, 多田好克訳: SPARC アーキテクチャ・マニュアル バージョン 8. トッパン (1992).

[7] Zainalabedin Navabi 佐藤一幸訳: VHDL の基礎, 日経 BP 出版センター (1994).

[付録 :メモリモデルの形式的定義]

本文では、メモリモデルに関して概念的な説明を与えたが、メモリモデルは、メモリ操作間の時相論理式として形式的に与えられる。本付録では、この形式的定義について述べる。

A 表記法

データのストア、ロードをそれぞれL、Sで表し、アトミックロード ·ストアを$[L;S]$で表す。ここで、$[]$はアトミック性を表す。そして、ストア ·バリア命令をBで表す。L、S、Bの上付添字はPEのIDを指し、L、Sの下付添字はメモリ ·ロケーションを指す。ただし、ストア ·バリア命令に関しては、全てのロケーションに対して実行されるため、下付添字は付加されない。Sの後の$\#n$は、Sにより書き込まれる値を示す。

$Val[]$は、Lにより返された値、またはSにより書き込まれた値を示す。$Val[]$は$[L;S]$全体そして、Bに対しては定義されない。

Opを、LまたはSの短縮形として定義する。Opは、アトミックロード ·ストアを含まないことに注意されたい。

最後に、(L、S、B)に対して2種類のオーダを定義する。

メモリ ·オーダ 単一のパーシャル ·オーダ$\leq$をメモリ ·オーダと呼ぶ。これは操作がメモリに対して実時間で実行される順序を示す。

プログラム ·オーダ プロセッサごとのトータル ·オーダ$;^i$をプログラム ·オーダという。これは、PEがメモリ操作を実行する順序を示す。

B JUMP-1 メモリモデルの形式的定義

前節で定義した表記法を用いて、JUMP-1メモリモデルの形式的定義を示す。

JUMP-1メモリモデルの形式的定義は、以下の6つの公理によって表される。

Order ストアに対してトータル ·オーダが存在する。

$(S_a^i \leq S_b^j) \vee (S_b^j \leq S_a^i)$

Atomicity アトミックロードストアは、ロードが先に実行され、かつそのロードとストアの間に他のストアは存在しない。

$[L_a^i; S_a^i] \Rightarrow (L_a^i \leq S_a^i) \wedge (\forall S_b^j : S_b^j \leq L_a^i \vee S_a^i \leq S_b^j)$

Termination 全てのストア及びアトミックロードストアは最終的に終了する。

$S_a^i \wedge (L_a^j;)\infty \Rightarrow \exists$ an L_a^j in $(L_a^j;)\infty$ such that $S_a^i \leq L_a^j$

Value ロードされるデータの値は、そのロケーションに対する最近のストアによるものである。

$Val[L_a^i] = Val[S_a^j | S_a^j = Max_{\leq}[S_a^k | S_a^k \leq L_a^i]]$

LoadOperation PEは、ロードの完了を待って、次のメモリ操作を実行する。

$L_a^i; Op_b^i \Rightarrow L_a^i \leq Op_b^i$

StoreStore ストアバリア命令に隔てられた2つのストアは、プログラム ·オーダとメモリ ·オーダが一致する。

$S_a^i; B^i; S_b^i \Rightarrow S_a^i \leq S_b^i$