設 計 自 動 化 77－6
(1995. 10. 19)

Ｓｕｐｅｒ Ｓｃａｌａｒ ＭＰＵの論理検証

†指宿 剛、足立 英樹
富士通 第二ＴＭＰ開発部
〒２０６ 東京都稲城市大丸１４０５
E-mail: ibusuki@tmp.pad.cs.fujitsu.co.jp
Ｂａｂｕ Ｔｕｒｕｍｅｌｌａ
ＨＡＬ Ｃｏｍｐｕｔｅｒ Ｓｙｓｔｅｍｓ
１３１５ Ｄｅｌｌ Ａｖｅｎｕｅ、Ｃａｍｐｂｅｌｌ、ＣＡ ９５００８、Ｕ．Ｓ．Ａ
E-mail: babu@hal.com

あらまし ハードウエアモデルを使った論理シミュレーションではテストベクターおよび各種検証プログラムを走行し論理設計を検証することが一般的である．その検証プログラムの中でもランダムに生成されたプログラムやＯＳカーネルあるいはアプリケーションプログラムに効果があることは周知の事実であり、我々もいくつかのランダム検証プログラムを開発した．しかし、この種のプログラムの運用では、膨大な実行サイクルとチェックアビリティの問題を解決しなければならなかった．我々は６４ビットＳＰＡＲＣプロセッサとして開発したＨＡＬプロセッサの論理シミュレーション・システムにシグネチャ比較とアサーション・チェックを導入することでこの問題を解決し検証効率を向上した．

和文キーワード： プロセッサ、論理シミュレーション、論理検証、ランダム・プログラム、シグネチャ、アサーション

Ｌｏｇｉｃ Ｖｅｒｉｆｉｃａｔｉｏｎ ｆｏｒ Ｓｕｐｅｒ Ｓｃａｌａｒ ＭＰＵ

†Ｔａｋｅｓｈｉ Ｉｂｕｓｕｋｉ、Ｈｉｄｅｋｉ Ａｄａｃｈｉ
２ＴＭＰ ＤＥＰＴ． Ｆｕｊｉｔｓｕ ＬＴＤ．
１４０５ Ｏｈｍａｒｕ Ｉｎａｇｉ ２０６、 Ｔｏｋｙｏ
E-mail: ibusuki@tmp.pad.cs.fujitsu.co.jp
Ｂａｂｕ Ｔｕｒｕｍｅｌｌａ
ＨＡＬ Ｃｏｍｐｕｔｅｒ Ｓｙｓｔｅｍｓ
１３１５ Ｄｅｌｌ Ａｖｅｎｕｅ、Ｃａｍｐｂｅｌｌ、ＣＡ ９５００８、Ｕ．Ｓ．Ａ
E-mail: babu@hal.com

Ａｂｓｔｒａｃｔ This paper provides the logic simulation methodology adopted on HaL processor design verification. The effectiveness of random verification program and application programs are well known as a good verification tool for the logic simulation. We developed some random verification programs which are well focussed on hardware behavior. It ,however, takes billions of machine cycles to run those programs until the design fix. In order to solve the problem, we embedded the efficient method , signature-compare and assertion-check , into our logic simulation environment. With the unique ideas, we ran a variety of programs in limited period , and fulfilled our goal.

英文ｋｅｙ ｗｏｒｄｓ：Processors、Logic Simulation, Logic Verification, Random programs, Signature, Assertion

1．はじめに

検証プログラムの検証能力については、ハードウエアのビヘイビア・モデルから網羅的に検証ポイントを抽出し、それを実現する命令列の自動生成［1］が研究されているが、今のところ簡易モデルで実用の可能性が証明された段階で、ハンドメイドの検証プログラムおよびランダム生成プログラムに頼っているのが現状である．後者においては様々な手法［2］が研究されており、我々もＢＩＡＳ命令列［3］を組み込んだランダム・プログラムをはじめとして各種のランダム検証プログラムを開発した．さらに実マシンで使用するＯＳカーネルやアプリケーションプログラムの一部を検証プログラムとして併用し効果を得た．

本論文はまず初めにＨＡＬプロセッサ(ＳＰＡＲＣ６４ビットアーキテクチャ)、その論理検証に使用した検証プログラムの概要を説明してから、ランダム検証プログラムおよびアプリケーション・プログラムの実行サイクルを格段に向上した．

2．ＨＡＬプロセッサ

ＨＡＬプロセッサ(図１)は６４ビットＳＰＡＲＣ－Ｖ９仕様［4］を初めてインプリしたプロセッサであり、ＣＰＵ［5］、２次キャッシュ［6］、ＭＭＵ［7］がＭＣＭにパッケージされる．２次キャッシュは命令キャシュ、データキャッシュそれぞれ２個（１つが６４ＫＢ）を装備している．

ＳＰＡＲＣ－Ｖ９仕様の特徴として ６４ビット命令セット、ユーザーモードでのＶ８オブジェクト完全互換、４トラップレベル、*Relaxed Memory Order*（ＲＭＯ）、さらに*big-endian*と*little-endian*をサポートしている．

マイクロ・アーキテクチャの特徴としては、*Dynamic Branch prediction*、*register renaming*、*out of order execution*［5］、*non-blocking cache*［6］が挙げられる．

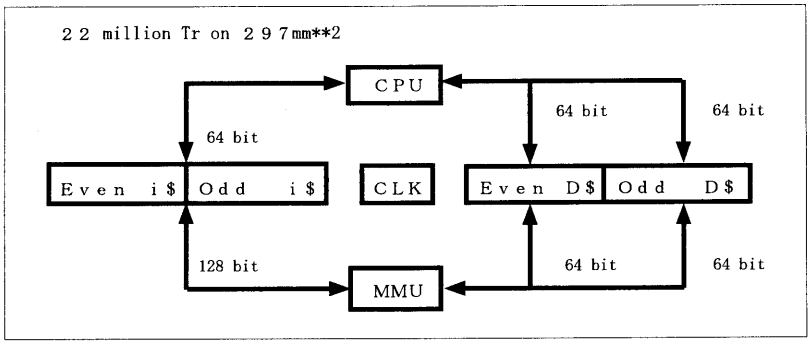

図１　ＨａＬプロセッサＭＣＭ

3．Ｈａ Ｌ検証ストラテジ

我々の検証ストラテジはプロセッサを構成する最小構成のBlockレベル、Unitレベル、Chipレベル、Processorレベル（＝ＭＣＭ）へと段階を踏んで行うもので、Blockレベル、Unitレベルは設計者がテスト・ベクタを用意し、自分の設計の正しさを検証する．Chipレベル、Processorレベルの検証では、ハンド・メイドあるいはランダム生成による検証プログラムを設計モデルで走行する．

我々は４種類のシミュレータ（表１）を用いて設計モデルの検証を行った．Ｃｏｓｉｍはシミュレーション言語で展開された設計モデルを高速に実行するアクセラレータ（社内製）でありＳＵＮワークステーションのアダプタとして位置する．ＳｏｆｔｓｉｍはＣｏｓｉｍのソフト版でありＳＵＮワークステーション上のアプリケーションとして動作する．ＨａＬｓｉｍは命令レベル・シミュレータであり、検証プログラムやＯＳカーネルをデバッグするための高速版（400K ins/sec）とよりハードを意識した論理検証の正解値モデル版（50K ins/sec）がある．

表1　シミュレーションの用途と性能

シミュレータ	用途	性能
Ｃｏｓｉｍ	設計モデル走行	＊1　８．４　inst/sec
Ｓｏｆｔｓｉｍ	設計モデル走行	＊1　０．７５　inst/sec
ＨａＬｓｉｍ　Ａ	検証正解値作成	５０　Kinst/sec
ＨａＬｓｉｍ　Ｂ	ソフト・デバグ	４００　Kinst/sec

＊1　ＩＰＣ＝１．４として換算

３．１　検証プログラム

先に述べたようにChipレベル、Processorレベルの検証ではハンドメイドの検証プログラム、ランダム検証プログラム、ＯＳカーネルそしてアプリケーション・プログラムを使用した．ここではそれら検証プログラムの特徴および検証ポイントを説明する．

３．１．１ ハイブリッドＩＶＰ (Impremention Verification Program)

これは純粋にアーキテクチャ仕様から見える部分(命令、trap、MMU仕様)以外、すなわちマイクロ・アーキテクチャの動作が仕様通りに動作することを詳細に検証する．このプログラムはアセンブラ言語とｐｅｒｌで構成されており、先に述べたＳｏｆｔｓｉｍ上でのみ走行させる．図２から分かるようにハードモデルの状態設定・参照はMACRO_STARTから始まるｐｅｒｌで記述している．この例ではｏｒ命令の実行タイミングでレベル１１の外部割り込みを発生している．

```
                    add %g0, 1, %g4
          loop:     or  %g1, %g2, %g3
MACRO_START
                    if (&cpu_issued(&pc_at(loop))
                              { &gen_interrupt (level11));}
MACRO_END
```

図２ ハイブリッド機能試験プログラム

３．１．２　ランダム検証プログラム

我々は検証ポイントを効果的に試験するために以下の特徴をそれぞれ持つランダム検証プログラムを開発した．

(1) *ＥＰ／ＳＰＡＲＣ*

このプログラムはＳＰＡＲＣアーキテクチャの特徴であるネスティング・トラップ、そしてＣＰＵ、Ｃａｃｈｅ、MMUのマイクロ・アーキテクチャ全般を検証している．特にパイプラインが詰まった状態での様々なキャンセル処理を主要な検証ポイントとした．このプログラムの命令列は完全なランダム生成列（②）とハード検証者作成のＢＩＡＳ命令列（③）のハイブリッド構成である．　ＢＩＡＳ命令列は設計を熟知するハード検証者が意図したテスト・ケースを冗長記述したもの（①）に乱数を掛け合わせて命令展開させる．前後の完全ランダム命令列と相互作用して検証者の気付かないコーナーケース・バグを多く検出した．さらにこのプログラムのトラップハンドラは以下の機能を持ち、プロセッサ内部のマシン・ステイタスをダイナミックに変化させ検証向上に寄与した．

●トラップ処理中のランダムなトラップ生成（レベル３まで可能）
●命令列へ復帰前に実行ＰＣ、アクセスアドレスの空間遷移
●Ｃａｃｈｅ、MMU内部状態の変更（cache ,TLB purge）

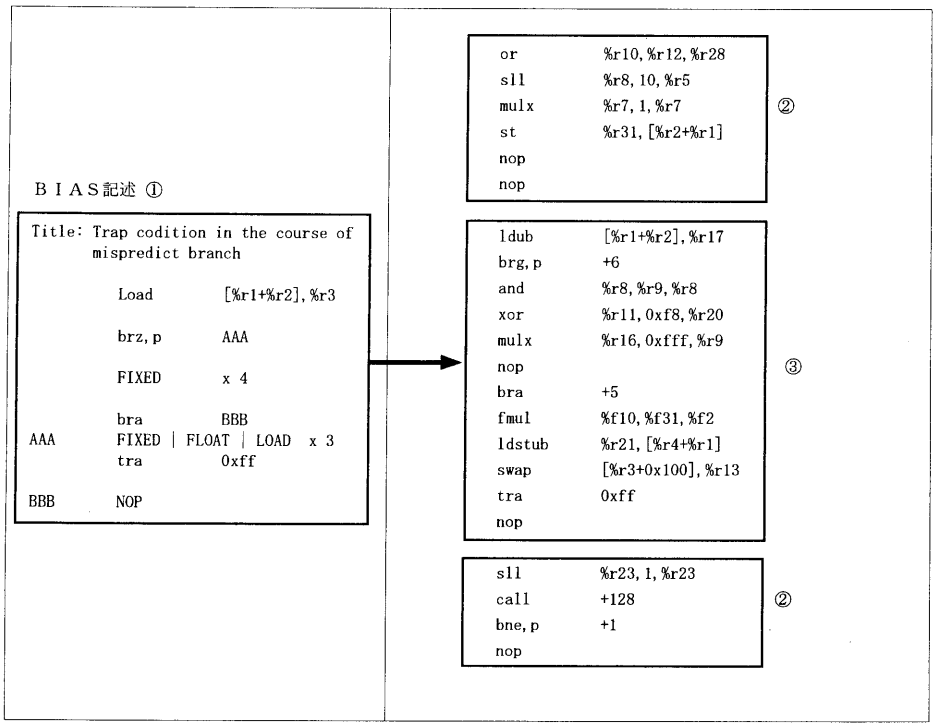

図３　*EP／SPARC*

（2）*Brand*

次は分岐命令をランダムに生成する*Brand*（図４）である．ＨＡＬプロセッサは予測分岐のヒット率を高める工夫を随所にしているが、この予測分岐の決定と予測が外れたときのリカバリ処理の設計がもっとも難しい．　*Brand*は分岐命令と遅延命令のバリエーション、予測分岐失敗時のキャンセル処理を検証ポイントとして効果を得た．　その特徴は図１の①に見られるように、暴走を防ぐために分岐が成功してもしなくても同一アドレス（ランダムに決定）に無条件分岐するようにしている点である．

```
                    . . . . . .
Ai+1:               FLOAT           x 2
                    bra             GOTO_END
                    . . . . . .
START               FIXEDcc         %rs,%rd
                    BRANCH          Ai
                    FLOAT           %fs,%fd
                    bra             Ai          <--①
                    nop
Ai:                 FIXED           %rs,%rd
                    MEMORY          [%gl+simm],%rd
                    BRANCH          Ai+1
                    FIXEDcc         %rs,%rd
                    bra             Ai+1        <--①
                    nop
```

図４　*Brand*

（3）*Blockbuster*

コンパイラの生成する命令列の多くにバック・ループがあり、なかにはネストするケースもある．このようなシーケンスを自動的にランダム生成する仕組みは検討されているが、未だ解がない．我々は*Blockbuster*［８］でその問題を解決した．　図５は*Blockbuster*のサンプルコ

ードであり、どのようなブロック・オペレーションで命令は何を使うかという指示スクリプト（①）を入力情報としてコンパイラが生成する典型的なアセンブラ命令列の近似（④）を自動生成している．前処理としてＢｌｏｃｋや命令の定義（②）をしてから、Ｂｌｏｃｋ内記述（③）で生成したい命令列のイメージを記述する． 生成されたアセンブラ命令列の⑤は*if then else*、⑥は *for* 文に相当する．スクリプトのＡＮＹ＿Ｂという記述がアセンブラ命令ではｌｄｕｂというメモリ・オペレーションに展開されている．*Ｂｌｏｃｋｂｕｓｔｅｒ*を利用することで さらに複雑にネストしたバック・ループも容易に生成できる．

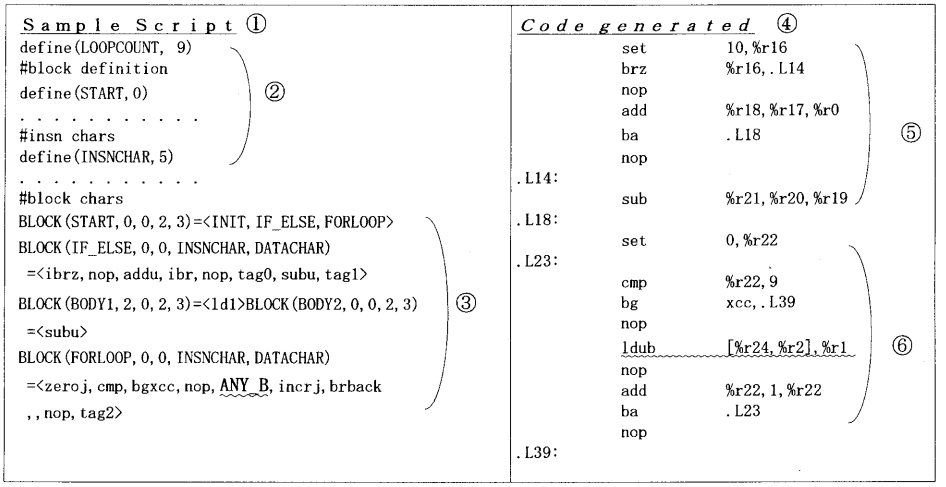

図５ *Ｂｌｏｃｋｂｕｓｔｅｒ*

３．１．３　その他の検証プログラム

検証専門のプログラム以外に我々はＯＳカーネルや一般アプリケーションを論理検証プログラムとして走行した． 後述する検証ステイタス（表２）から判断できるようにこれらのプログラムは検証ツールとして極めて有効であることが証明されている．それぞれのプログラムサイズは400K命令で長い冗長なループを除くものを５０ｓｔｒｉｐ走行した．ＯＳカーネルはシングルユーザーで動作する部分、アプリケーションは以下を使用した．

●ＳＰＥＣベンチマーク
●Ｉｎｆｏｒｍｉｘ、Ｏｒａｃｌｅ
●コンパイラ（ｇｃｃ）、リンカー
●各種Ｕｎｉｘユティリティ

３．２ 検証ステイタス

表２は各検証プログラムが１Ｋサイクルあたり何件のハードバグを検出したかを表している．ハンドメイドの機能レベル検証プログラム（*ＩＶＰ*）の方がランダム検証プログラム（*ＥＰ／ＳＰＡＲＣ*、*Ｂｒａｎｄ*、*Ｂｌｏｃｋｂｕｓｔｅｒ*）よりもはるかにいい値を出すのが一般的だが、ＨＡＬではある程度機能検証プログラムが走行すると並行してランダム検証プログラムの走行を開始したためこの結果になったと捉えている．しかし、それを差し引いてもランダム検証プログラムやＯＳカーネル走行の検証効率は無視できない．ランダム検証プログラムの検証効率を高めた要因は４章で説明するようにプログラム自身からセルフ・チェックを排除することで冗長ステップが不要になったためである．

このセルフ・チェック無しでも高い検証能力に貢献したのが、４章で説明するシグネチャ比較方式、アサーション・チェックである．

表２　検証プログラムの性能比較

検証プログラム	Weekly Kcycles	Total Defect／Kcycles
IVP	1879	2．20
EP／SPARC	9160	0．60
Blockbuster	2776	0．61
Brand	3180	0．54
OS Kernel & Application	9933	0．53

４．検証効率の向上

これまでのランダム検証プログラムの結果確認は、ソフトシミュレータで走行した結果(これが正解値となる)をダンプしておき、ハードウエアモデルに検証プログラムと同時に持ち込みダイナミックに比較するか、あるいはハードモデルの結果を最後にダンプして両者を比較するかのどちらかであった．いずれの場合もあるタイミング(以後、チェックポイントと呼ぶ)でアーキテクチャ・レジスタやメモリのサム・チェックをこまめに比較するのがより精度の高い確認方法である．

しかしこのチェック方法の欠点はハードウエア・モデル上で一定あるいは不定期のチェックポイントで（検証プログラムが）アーキテクチャ・レジスタを正解値と比較するオペレーション（以後、セルフ・チェック）が必然的に発生することである．ＲＩＳＣのようにレジスタ数が多いアーキテクチャでは本処理は無駄なクロックを消費していると言える．とくにＳＰＡＲＣアーキはウィンドウ・レジスタが特徴であり、汎用レジスタに関しては９６個、浮動小数点レジスタが３２個、ＰＣ／ｎＰＣ、トラップ関連レジスタまで含めると総数１５０個を超えるレジスタの比較が必要となる．１個のチェックに少なくとも６命令かかるとしても約９００命令を要する．実際はウインドウを変更したりする手間がかかるため１０００命令を必要とする．実機ならともかく論理シミュレーションでこのステップ数は無視できる数値ではない．

４．１　シグネチャ比較

先に述べたＣｏｓｉｍに展開されるＨＡＬのハードウエアモデルには本来の回路以外にアーキテクチャ・レジスタごとにシグネチャ［９］を生成する回路が組み込まれている．当然この回路は実際のチップでは存在しない．シグネチャ生成論理は一般に知られているＸＯＲの組合せで実現しており、図６にその式を紹介する．ビット０、２、３では余分なＸＯＲによりall '0',all 'f' が連続したとしても履歴が蓄積されるよう考慮されている．

```
Q[63:0]：signature RAMの現在の値
D[63:0]：signature RAMの新しい値
X[63:0]：新しいregister/PCの値

        D[0] xor Q[63]
        D[1] <- X[1] xor Q[0] xor Q[63]
        D[2] <- X[2] xor Q[1]
        D[3] <- X[3] xor Q[2] xor Q[63]
        D[4] <- X[4] xor Q[3] xor Q[63]
        D[5] <- X[5] xor Q[4]
        D[6] <- X[6] xor Q[5]
        ...
        D[62] <- X[62] xor Q[61]
        D[63] <- X[63] xor Q[62]
```

図６　シグネチャ生成式

このシグネチャは命令が完了しレジスタが変更されたことを契機に更新される．我々の運用では図７のように１０００クロック(可変に設定可能)毎にハードウエア・モデル（Ｃｏｓｉｍ）が生成したシグネチャ（以後、*hard-signature*）（１）とソフト・シミュレータ（ＨａＬｓｉｍ）が同じロジッで生成したシグネチャ（以後、*soft-signature*）（４）を両者に介在する論理シミュレーション・デバガ（以後、ＬＤＢ）が比較（５）する．このＬＤＢはハードウエア・モデルが所定クロックに到達するとハードウエア・モデルを停止、*hard-signature*をアップロード（２）、ソフトシミュータに完了した命令数

分を実行するよう指示する（３）．ソフト・シミュレータはシーケンシャル・マシンであるため実行クロック数ではなくハードウエア・モデルで何命令が完了したかが意味を持つ．ＬＤＢは両者のシグネチャを比較（５）し、不一致があればシステムを停止する．問題がなければハードウエア・モデルを再開させる．この間、ハードエア・モデルは停止しているが、*hard-signature*のアップロードから始まりソフト・シミュレータの実行、*soft-signature*との比較に到る動作時間はハードウエア・モデルの実行時間に比較すると殆ど無視出来るためシミュレーション時間には影響がない．

実行サイクルの削減以外に大きな効果は、セルフ・チェックを不要にしたことでＯＳコードや一般アプリケーションを論理シミュレーションにかけられることである．検証プログラムは通ったがチップが出来てからある特定アプリケーションではフェイルするというのは良くある話である．我々はＯＳカーネル、コンパイラ（ｃｃ、ｇｃｃ）、データベース等をテープ・アウトまでに流し込み、ファースト・シリコンでのＯＳブートを１日で実現した．

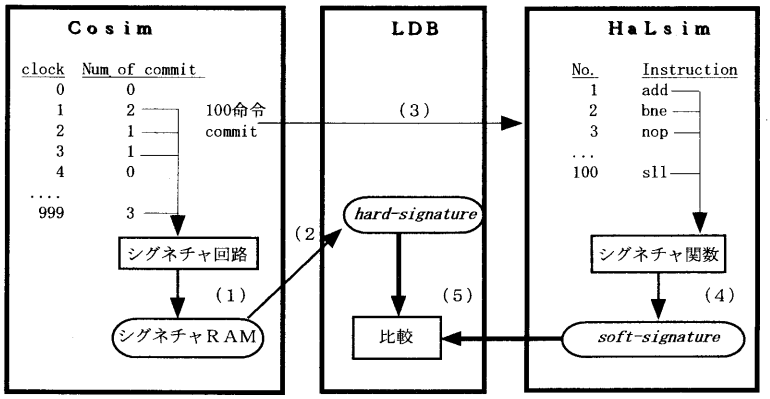

図７　シグネチャ・チェック方式

４．２　アサーション・チェック

ハードウエア・モデルには先に述べたシグネチャ回路以外に仕様に対する矛盾をチェックするアサーション・チェックと呼ぶロジックが随所に組み込まれている．これもシグネチャ回路同様に実際の回路とは独立して動作し、あるサイクル毎にロジックの矛盾を監視する．

ＶＨＤＬでは***ａｓｓｅｒｔ***文でタイミングの矛盾チェックが記述できるが、我々は同様のことをマイクロ・アーキテクチャの仕様検証に用い、これまでにクリティカルなバグを２５件ほど検出している．図８の例は「コミット命令をカウントするカウンタＣＳＮは１サイクルで８個進んではならない」というマイクロ・アーキテクチャ仕様を検証するアサーション・チェックである．

```
# Check that CSN never advances by more than 8 in any cycle
            if ($CSN - $OLDCSN > 8) {
            printf("CSN skipped too much¥n");
            &assert_fail;
            }
```

図８　アサーション・チェック

この機能はランダム検証プログラムのようにメモリ・モデルのコヒーレンシイをまめにチェックできない物に対しては、検証力の補完の意味で効果は大きい．

5. おわりに

シグネチャ比較、アサーション・チェックというメカニズムを論理シミュレーション環境に取り込むことで検効率、検証レベルが向上したことでＨＡＬプロセッサの実機レベルでの立ち上げがスムースに運んだことを述べた．シミュレーションでランダム検証プログラムを始め多くのアプリケーションを流し込んだことにより、低負荷状態での障害は殆ど見られなかた．この点に関しては我々の取り組みの正しさが証明されたものと認識している．

現在は非同期な要因を含んだ(例えばタイマーやｉ／ｏ割り込み)検証においてもシグネチャ比較を適用できるようハードウエア・モデルとＬＤＢに特別なインターフェイスを構築中である．

今後の大きな取り組みはマルチＣＰＵ環境におけるシグネチャ比較の適用であるが、これに関してもメモリ制御ユニット（MMU）モデルでアクセス順序の正当性を保証しておき、その順序は正しいという前提でソフト・シミュレータにメモリ・オペーションのオーダーを伝えればシングルＣＰＵ同様にシグネチャ比較が使える見通しである．　これが実現すればマルチＣＰＵ構成で常に問題となるキャッシュ・コヒーレンシイ矛盾を論理検証可能である．

『参考文献』

[1] 中田、古渡、岩下、広瀬：論理シミュレーションをベースとしたプロセッサ制御の効率的検証手法 FTS94-67, VLD94-86(1994-10)

[2] J. Miyake, G. Brown, M. Ueda and T. Nishiyama: Automatic Test Generation for Functional Verification of Microprocessor;Third Asian Test Symposium

[3] A. Aharon, A. Bar-David, B. Dorfman, E. Gofman, M. Leibowitz, V. Schwartburd: Verification of the IBM RISC System/6000 by a dynamic biased pseudo-random test program generator ; IBM SYSTEMS JOURNAL VOL 30. NO 4, 1991

[4] The SPARC ArchitectureManual-Version9 : David L. Waever and TomGermond, Editors ;PTR Prentice Hall, 1994 ISBN 0-099227-5

[5] N. Patkar, A. Katuno, S. Li, T. Maruyama, S. Savkar, M. Simone, G. Shen, R. Swami and D. Tovey: Microarchitecture of the HaL PM1 CPU; Compcon Proceedings 1995

[6] C. Chen, Y. Lu and A. Wong: The Microarchitecture of the HaL Cache; Compcon Proceedings 1995

[7] D. Chen, D. Lyon, C. Chen, L. Peng, M. Masaumi, M. hakimi, S. Iyengar, E. Li and R. Remedios: The Microarchitecture of the HaL's Memory Management; Compcon Proceedings 1995

[8] Rajeev Bharadhwaj: Blockbuster--a CPU Test Generator Design Specification Sep 15, 1995 Preliminary

[9] Babu Turumella, Aiman Kabakibo, Manjunath Bogadi, Karunakara Menon, Shalesh Thusoo, Long Nguyen, Nirmal Saxena, Michael Chow : Design Verification of a Super-Scalar RISC Processor