設計自動化 55－4
(1990.12.12)

ＢｅＳＩＭ：動作レベル・シミュレータ

高嶺 美夫，杉田 智彦，清水 嗣雄
㈱日立製作所

あらまし　動作レベル・シミュレーションにおいて最も多くの処理時間を要しているのは、ブール式や真理値表により記述された組合せ回路部分のイベント評価である。この点に着目し、イベント保留デマンド・ドリブン(ＲＥＤＤ)方式を考案し、高速な動作レベル・シミュレータＢｅＳＩＭを開発した。ＲＥＤＤ方式は、特にレジスタ前段にある組合せ回路における不要なイベント評価を抑止することにより、当該部分のシミュレーション時間を４０～９０％削減可能である。Sun SPARC station 370を用い、１００ｋゲート規模のＣＰＵモデルを対象とするシミュレーションをＢｅＳＩＭにより行った場合、ゲート・レベル論理シミュレータに比べ１桁以上高速な、１９.４サイクル／秒の処理速度を得た。

BeSIM: Behavioral Simulator

Yoshio Takamine, Toshihiko Sugita, Tsuguo Shimizu
Hitachi, Ltd.
Kokubunji, Tokyo 185, Japan

Abstract　A new simulation algorithm for high-speed behavioral simulation is described. In behavioral simulation, the process that consumes the most time is the evaluation of combinational logic circuits, which described by Boolean expressions or truth tables. To reduce the time, a reserved-event and demand-driven (REDD) algorithm was developed. It lessens useless evaluations, especially those of circuits located in front of registers. Incorporating this algorithm, a behavioal simulator called BeSIM has been developed. Using a Sun SPARC station 370, BeSIM can simulate a CPU system with about 100 kilogates at a speed of 19.4 cycles per second, which is over ten times faster than that of a gate-level simulator.

1．緒言

近年ますます大規模化、複雑化する計算機、ＶＬＳＩの開発において、その開発工数や開発期間の増加が、特に論理設計とその検証に関し問題となっている。この論理設計、検証に要する工数を削減することを目的に論理ＤＡの研究が盛んに行なわれているが、その中で論理自動生成システムと専用論理シミュレータが大きな効果をもたらしている[1,2]。論理自動生成システムはブール式、真理値表記述からゲート論理を自動生成するシステムであり、今日組合せ回路部分の設計においては、ゲート・レベルに代わりブール式レベルで設計を行うことが一般的となっている。またベクトル・プロセサや並列プロセサを利用しシミュレーションを高速実行する専用論理シミュレータの実用化により、設計検証においては従来に比べ格段に多くのテストを実施できるようになった。

このような論理ＤＡ技術の向上にもかかわらず論理設計、検証に要する工数は増え続け、今日では全開発工数の半分をはるかに超える割合を占めている。したがってこれらの工数を大幅に削減するためには、さらに抽象度の高い動作(レジスタ転送)レベルでの設計を可能とする高位論理ＤＡ技術の開発が必要である。

この分野では、動作レベル記述からブール式レベルの論理を生成する高位論理自動生成システムの研究が行われている[3]。また高位論理設計に対応した検証のツールとして、動作レベル・シミュレータも開発されている[4-9]。動作レベル・シミュレータは、設計の早いフェーズで検証を行うことにより早期に設計不良摘出を可能とし、摘出不良の対策に要する工数を削減する効果がある。高位論理設計においては、設計者が１人１台のエンジニアリング・ワークステーション(ＥＷＳ)を占有し、同一の枠組みの中で設計と検証とを進められるような環境を提供する必要がある。そのような環境において検証を効率良く実施するためには、動作レベル・シミュレータは動作レベル記述を直接取扱い可能で、かつＥＷＳ上で実行したとしても前述の専用論理シミュレータに匹敵する処理速度が得られる必要がある。

ＥＷＳ上で十分なシミュレーション速度を達成するために、いくつかの高速化方式が提案されている。デマンド・ドリブン・シミュレーション[6]は、シミュレーション結果が必要な箇所から時間、空間をさかのぼってシミュレーションを実行する方式である。無駄な評価を一切行なわないことより、処理時間を削減することが可能と考えられる。しかしながら再帰的な処理によって多くのメモリと、処理オーバヘッドを必要とする。またインクリメンタル・シミュレーション[7]は、過去に実施したシミュレーション結果を再利用することにより、シミュレーション時間を削減することが可能であるが、やはり多大な情報を格納する領域と、処理オーバヘッドを必要とする。

以下本報告では、動作レベル・シミュレーションの高速化方式について述べる。本方式はシミュレーション対象回路の動作に着目し、極めて小さな処理オーバヘッドで高速化を実現するものである。まず第２章に動作レベル記述とそのシミュレーションの特徴を述べる。続いて第３章では、動作レベル・シミュレーションの高速化における現状の問題点を示し、それを解決する新たなシミュレーション実行方式、すなわちイベント保留デマンド・ドリブン(ＲＥＤＤ)方式を提案する。第４章に本方式を実現した動作レベル・シミュレータＢｅＳＩＭの概要とその性能、およびＲＥＤＤ方式の効果を記す。

2．動作レベル記述とシミュレーション

2．1 動作レベル記述言語Ｂ²ＤＬ

計算機やマイクロ・プロセサ等において広く採用されているパイプライン方式のハードウェア記述が容易であることを特徴とする動作レベル記述言語Ｂ²ＤＬ(Block-diagram and Behavior oriented Description Language)の言語仕様の概要を図２．１に示す。またＢ²ＤＬによる記述例を付図１に示す。

Ｂ²ＤＬによるハードウェア記述は、構造記述部と動作記述部から成る。構造記述部ではメモリ、レジスタ、入出力信号等のハードウェア要素を宣言する。また組合せ回路部分の構造を論理式、真理値表等を用いて詳細に記述することが可能である。動作記述部では、構造記述部で宣言したハードウェア要素に対する操作を定義する。動作記述は並列動作が可能な複数のシーケンスから成る。シーケンスは、指定された開始条件が成立した後に連続して実行される複数のステップから成る。さらにステップは同時に実行すべき単位動作の集合である。単位動作とはレジスタ転送動作等であり、必要に応じて実行条件が指定される。あるステップを実行する時、各単位動作に与えられた実行条件が満たされていなければ、当該単位動作の実行は抑止される。またＢ²ＤＬでは記述対象となるハードウェアは同期回路であり、各ステップは対応するクロック信号の指定を含む。

動作レベル記述において留意すべき点は、組合せ回路に関する記述が非常に多いことである。実

際のハードウェアにおいてレジスタ間に多くの組合せ回路が存在することに対応して、B²DL記述においてもそれらの組合せ回路を表現する論理式、真理値表記述が多数存在する。特に動作記述部においては、シーケンスの開始条件、単位動作の実行条件、レジスタ転送動作における転送データ等を定義する論理式が随所に見られる。これらの論理式は、B²DL記述における記述量や、後述するように動作レベル・シミュレーションを行う際の処理速度に大きな影響を与えている。

2.2 動作レベル・シミュレーション

B²DLによるハードウェア記述をチェックし、早期に設計不良を摘出するために行う動作レベル・シミュレーションについて述べる。

動作レベル・シミュレーションは、実行可能な動作の選択と、当該動作の実行結果の算出という2つの処理の繰返しと考えられる。B²DLにより記述された同期回路を対象とする場合には、クロック信号が変化する毎に上記2つの処理を行う。実行可能な動作の選択では、全ての動作についてそれぞれの実行条件を調べる必要はない。B²DLによる動作記述はシーケンス、ステップ、単位動作という階層構造を持っているので、その階層に従った処理を行う。すなわちまず全シーケンスの開始条件を調べ、当該時刻に実行すべきステップを選択する。次に該当するステップに含まれる単位動作について指定された実行条件を調べる。

次に動作の実行結果の算出を行う。通常この時点で実行すべき単位動作は複数存在する。これらの単位動作は互いに独立であるので、シミュレーションにおいてもそれを保証する必要がある。すなわちいかなる順序でこれらの単位動作をシミュレートしたとしても結果が不変でなければならない。特にレジスタ転送動作において同一レジスタに複数のデータを同時に転送しようとした場合、シミュレーションの処理順序によっては結果が異なる可能性がある。したがってレジスタ転送動作をシミュレートする場合には、事前に転送先レジスタに関し衝突の有無を調べ、かつ全ての転送データを確定しておく必要がある。

```
BLOCK : ブロック・タイプ名;
  MEMORY:     メモリID;
  REGISTER:   レジスタID;
  INPUT:      入力信号ID;
  OUTPUT:     出力信号ID;
  SIGNAL:     内部信号ID;
  EXBLOCK:    外部定義ブロックID;
  EXMODULE:   外部定義モジュールID;
              (組合せ回路)
  CONNECT:    結線 or 組合せ回路;
                    (論理式)
  ARRAY:      組合せ回路c;
              (真理値表)
  SEQUENCE:   シーケンス
  END;
  SEQUENCE:   シーケンス
  END;
                .
                .
                .
END : ブロック・タイプ名;

シーケンス ::= シーケンスID
               << 開始条件 >>
               #1 : クロック信号   ステップ
               #2 : クロック信号   ステップ
               #3 : クロック信号   ステップ
                    .
                    .
                    .
  ステップ ::= 単位動作;
               単位動作;
                    .
                    .
                    .
```

構造記述部

動作記述部

図2.1 B²DL言語仕様

3. イベント保留デマンド・ドリブン方式

3.1 動作レベル・シミュレーションの問題点

動作レベル・シミュレーションは、入力となるハードウェア記述の抽象度が高いことから、一般にゲート・レベルあるいはブール式レベルのシミュレーションに比べ格段に高速な処理が可能と考えられている。確かにレジスタ転送等の動作をシミュレートする部分に関しては、実際のハードウェアにおける制御回路等の内部状態の変化を詳細にシミュレートする必要が無いため高速に処理が可能である。しかしながら実際の動作レベル・シミュレーションにおいては、期待される程の処理速度が得られないことが多い。

これは動作レベル・シミュレーションといえども、論理式や真理値表により記述された組合せ回路部分に関してゲート・レベルあるいはブール式レベルのシミュレーションと同様の処理、すなわち個々の演算子毎に出力信号値の算出を行っていることに起因する。前述のとおり動作レベル記述中には論理式、真理値表等による組合せ回路の記述が非常に多く存在し、結果的に通常の動作レベル・シミュレーションでは、その処理時間の90%以上が組合せ回路部分の評価に費やされている。

したがって動作レベル・シミュレーションの高速化のためには、組合せ回路部分の評価に関し、その処理時間の削減を図る必要がある。

従来からゲート～ブール式レベル・シミュレーションの高速化手法として、イベント・ドリブン方式が知られている。イベント・ドリブン方式では入力信号の値が変化した要素についてのみその出力値算出処理を行うことにより、明らかに無駄と判る処理の実行を抑止している。この方式は動作レベル・シミュレーションにおいても組合せ回路部分の評価に適用可能である。しかしながら例えばレジスタ前段の組合せ回路では無駄な処理が多く存在する。図３．１(ａ)に示した回路(太枠内を除く)のシミュレーションを行う場合について考える。イベント・ドリブン方式では、入力信号Ｘの値が変化する都度イベントが組合せ回路内に発生し、その出力Ｙの値を算出する。しかしながら同図(ｂ)に示すように、Ｙの値が参照されるのは後段のレジスタＲの制御信号Ｇが正で、クロック信号Ｋ０が立ち上がった時だけである。Ｇの値が負の場合、Ｙの値の算出は無効となる。これはイベント・ドリブン方式が入力値の変化のみに着目し、そのイベント評価結果が実際に必要とされているか否かを考慮していないためである。

前述のデマンド・ドリブン・シミュレーション、あるいは要求駆動型ＲＴレベル・シミュレーション[8]は、値の参照要求に基づいて処理を行うので、上述のような無効な評価は抑止される。しかしながらこれらのシミュレーション方式では、入力値が変化していないにもかかわらず同一の計算を繰返し実施するおそれがある。

３．２　ＲＥＤＤ方式の原理

シミュレーション対象回路の機能、すなわちレジスタ転送等の動作に着目し、組合せ回路部分の無効な評価を抑止するイベント保留デマンド・ドリブン(ＲＥＤＤ：Reserved-event and Demand -driven)方式を考案した。ＲＥＤＤ方式の原理を図３．１により説明する。本方式では、図中太枠内に示した仮想的なイベント・フラグ等を用いる。ＲＥＤＤ方式の処理は、以下の２つの手続きから成る。

(ａ)イベント保留手続き(図中①)：

組合せ回路の入力信号値、すなわちＸの値が変化した場合、イベントを仮想的なイベント・フラグにセットする。

(ｂ)イベント評価手続き：

以下の３ステップから成る。

(ｉ)参照要求検出(図中②)：

組合せ回路の出力信号値に対する参照要求が発生した場合、すなわち後段のレジスタＲに対する転送動作が実行される時、イベント・フラグを調べ、イベントが保留されていたらステップ(ii)に進み、そうでなければステップ(iii)に進む。

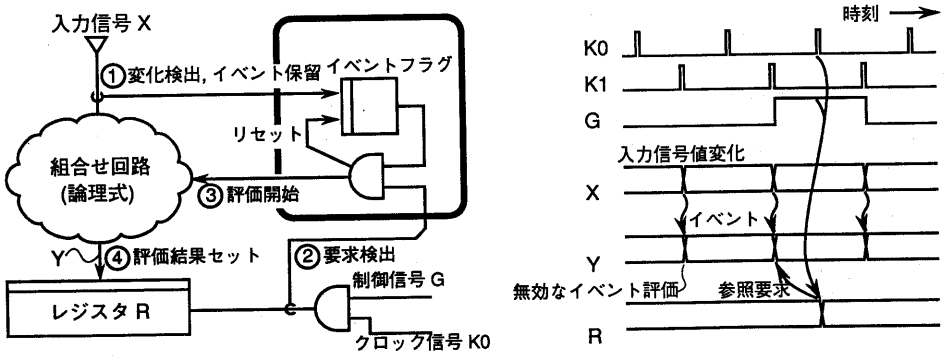
(ii)組合せ回路評価(図中③)：

イベント・フラグをリセットし、組合せ回路の出力Ｙの値を算出する。

(iii)出力値参照(図中④)：

この時点で組合せ回路の出力Ｙの最新の値が参照可能となっているので、それをレジスタＲにセットする。

ＲＥＤＤ方式は、イベント・ドリブン方式と同様入力信号値の変化に着目しつつ、イベントの評



(a) 回路例と制御メカニズム　　(b) 動作

図３．１ イベント保留デマンド・ドリブン(REDD)方式の原理

価は出力信号に対する参照要求が生じるまで行わない。このため、イベント・ドリブン方式で発生するような無効なイベント評価を抑止することが可能である。

3.3 実現方式とデータ構造

ＲＥＤＤ方式を実現するデータ構造について述べる。この場合、イベント・フラグを何に対応して設けるか、また参照要求発生時に評価をどのように行うかによって、以下の３通りのデータ構造が考えられる。

(1)論理式対応のイベント・フラグ：

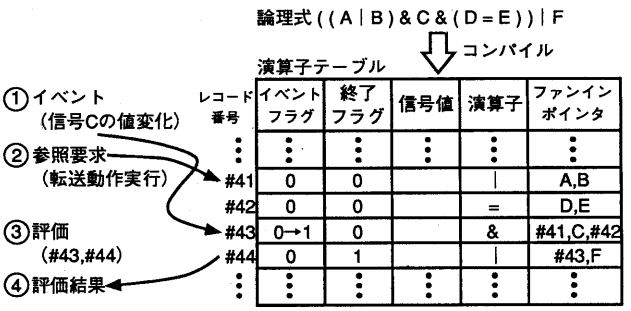
図３．１に示したＲＥＤＤ方式の原理をそのまま実現するものである。ある論理式について、その入力値のいずれかが変化したら、論理式に対応するイベント・フラグに”１”をセットする。また論理式の出力値に対する参照要求が発生した場合は、必ず論理式全体を評価する。

本データ構造では、論理式の中の各演算子についてはその入力値が変化していないものについても評価を実施するため、論理式が比較的大きな場合、あるいは信号の変化率が低い場合に、無効な演算子評価が多くなる可能性がある。

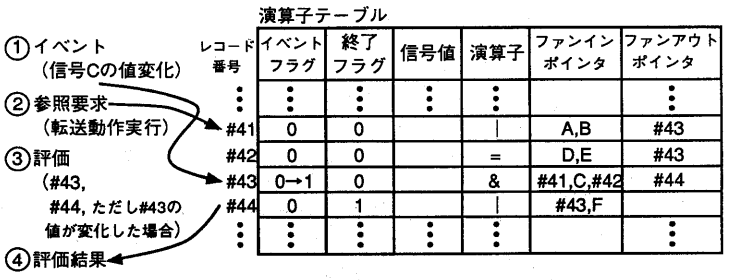
(2)演算子対応のイベント・フラグ、ファンアウト・ポインタ無し：

データ構造を図３．２(ａ)に示す。演算子テーブルの各レコードは組合せ回路、すなわち論理式を構成する各演算子に対応している。１つの論理式に属する一連のレコードをその評価順に従って並べることにより、論理式の出力値を算出する場合、演算子テーブル内の一連のレコードを順に処理することにより結果が得られるようにしておく。

イベント・フラグは、個々の演算子に対応して、演算子テーブルの各レコードに設ける。論理式の入力値が変化した場合、当該入力値を直接参照する演算子に対してのみ、対応するイベント・フラグに”１”をセットする(図中①)。論理式の出力値に対する参照要求が発生した場合は、対応する演算子テーブルの一連のレコードの先頭から順にイベント・フラグを調べ、”１”がセットされているレコー



(a) 演算子対応のイベント・フラグ, ファンアウト・ポインタなし



(b) 演算子対応のイベント・フラグ, ファンアウト・ポインタあり

図 3.2 REDD方式のデータ構造

ドが存在したら、当該レコード以降の演算子の評価を実施する(図中②,③,④)。

本データ構造は、(1)に比べやや処理が複雑になるが、入力値が変化していない演算子の評価は大幅に抑止できる。ただし図3.2(a)に示した例において、#44のレコードは、その入力値が実際に変化していなくても必ず評価される。

(3)演算子対応のイベント・フラグ
ファンアウト・ポインタ有り:

データ構造を図3.2(b)に示す。データ構造は(2)とほとんど同じであるが、演算子間の出力値参照関係をファンアウト・ポインタにより示している。

論理式の入力値が変化した場合、(2)と同様に当該入力値を直接参照する演算子に対して、対応するイベント・フラグに"1"をセットする。論理式の出力値に対する参照要求が発生した場合も、(2)と同様に対応する演算子テーブルの一連のレコードのイベント・フラグを調べる。本データ構造では、イベント・フラグに"1"がセットされているレコードについてのみ演算子の評価を行う。そのために演算子の出力値を算出し、値が変化した場合にはファンアウト・ポインタの内容に従ってその値を参照する演算子をたどり、対応するイベント・フラグに"1"をセットする。

本実現方式では、入力値が変化した演算子についてのみ評価を実施することにより、その評価数を最少とすることが可能である。

以上のデータ構造は、平均的な論理式の大きさと、信号値の変化率によって、その優劣が決定される。

4. 動作レベル・シミュレータBeSIM

4.1 システム構成

前章に述べたREDD方式を実現する動作レベル・シミュレータBeSIMを開発した。システム構成を図4.1に示す。BeSIMはB²DLによるハードウェア記述のほかに、シミュレーション・データを入力とする。シミュレーション・データはシミュレーション実行のためのテスト・パタン、シミュレーション結果をファイル等に出力すべき信号名の指示等を含む。またBeSIMはB²DLコンパイラと、シミュレータ本体から成る。B²DLコンパイラはシミュレーション実行用のテーブルを作成し、シミュレーション・マスタ・ファイルに出力する。シミュレータ本体は、シミュレーション・マスタ・ファイル、シミュレーション・データを入力し、シミュレーションを行い、結果を編集してシミュレーション結果ファイル、あるいはリストに出力する。B²DLコンパイラ、シミュレータ本体ともに、メイン・フレーム、あるいはEWS上で実行可能である。

4.2 実験結果

BeSIMのシミュレーション処理速度を測定するために、汎用計算機のCPUの一部(100kゲート規模)、および1kゲートのLSIを対象とするシミュレーションを行った。B²DLによるハードウェア記述はそれぞれ4700行、240行である。BeSIMのシミュレータ本体は、3.3節で述べたREDD実現方式のデータ構造(2)を組込んでいる。ただしB²DLの構造記述部において論理式、真理値表等により記述された組合せ回路の評価には従来からのイベント・ドリブン方式を用い、動作記述部に関する組合せ回路部分の評価に関し、REDD方式を適用している。

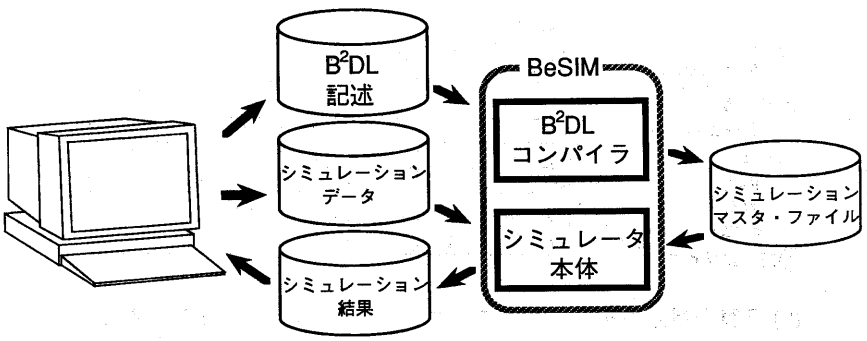


図4.1 システム構成

Ｓｕｎ　ＳＰＡＲＣ　ｓｔａｔｉｏｎ　３７０を用いたシミュレーション実行結果を表４．１に示す。ＣＰＵを対象とする場合において、１９．４サイクル／秒のシミュレーション速度を得た。これは同程度の性能を持つ計算機で実行したゲート・レベル論理シミュレータに比べ、１桁以上高速な値である。

またＲＥＤＤ方式の効果を評価するために、上記ＣＰＵモデルを対象とする実験を行った。前述のとおりＣＰＵモデルは４７００行のＢ²ＤＬ記述から成る。その４分の１が論理式、真理値表等による組合せ回路記述である。論理式は３５９個あり、その内１６３個が動作記述部に含まれ、ＲＥＤＤ方式の適用対象となるものである。

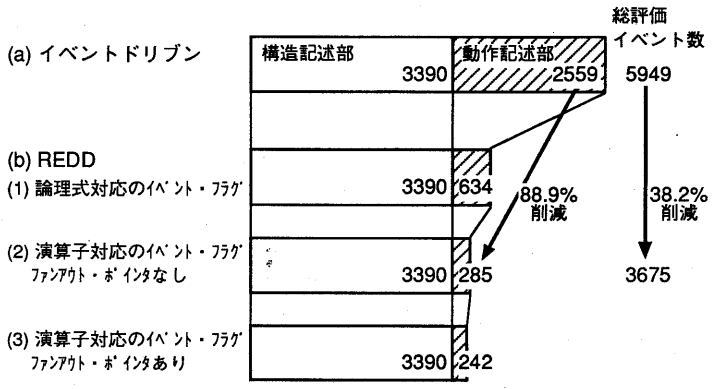


図４．２ REDD方式の効果 (評価演算子数)

実験結果を図４．２に示す。従来のイベント・ドリブン方式によるシミュレーションを実施した場合、演算子を単位とする評価実行数は５９４９回であり、その内２５５９回が動作記述部に含まれる論理式に関するものであった。ＲＥＤＤ方式を適用した場合、実現したデータ構造によって、動作記述部に関する演算子評価数は６３４回から２４２回に削減された。中でもデータ構造(２)は、(３)に比べて１７．８％しか評価数が増加しておらず、テーブル・サイズ、処理の複雑さを含めて考えた場合に、最も効果的な実現方法と考えられる。また論理式の出力値に対する参照要求の発生率は６．５％であり、ＲＥＤＤ方式の有効性が認められた。現在ＲＥＤＤ方式はＢ²ＤＬの動作記述部に含まれる論理式の評価に対してのみ適用しているが、構造記述部の論理式、真理値表等にまで適用範囲を拡大することによりより大きな効果が得られるものと期待される。

５．結言

動作レベル・シミュレーションの高速化を目的とするイベント保留デマンド・ドリブン(ＲＥＤＤ)方式について述べた。従来の動作レベル・シミュレーションにおいて、処理時間の９０％以上は組合せ回路部分の評価に占められている。ＲＥＤＤ方式はシミュレーション対象回路の動作に着目し、特にレジスタ前段にある組合せ回路に関し、結果が参照されないような無効なイベント評価を抑止する。また本方式を実現するいくつかのデータ構造の比較検討を行なった。

本方式を実現し、Ｂ²ＤＬによるハードウェア記述を入力とする動作レベル・シミュレータＢｅＳＩＭを開発した。評価の結果ＲＥＤＤ方式の有効性を確認した。

現在ＢｅＳＩＭでは、Ｂ²ＤＬ記述の動作記述部にて論理式により記述された組合せ回路の評価にのみＲＥＤＤ方式を適用している。適用範囲を構造記述部にまで拡大することが今後に残された課題である。

表４．１ BeSIMの性能

	ｹﾞｰﾄ数	B²DL 記述行数	実行 ｻｲｸﾙ数	CPU時間 [秒] *1		ｼﾐｭﾚｰｼｮﾝ 速度 [ｻｲｸﾙ/秒]
				ｺﾝﾊﾟｲﾙ	ｼﾐｭﾚｰｼｮﾝ *2	
CPU	100 k	4700	20	11.0	2.7(1.0)	20
LSI	1 k	240	40	0.7	1.4(0.6)	67

*1：Sun SPARC station 370 使用.
*2：テストデータ・コンパイル，結果出力を含む.
括弧内はシミュレーション実行のみ.

謝辞

研究の機会を与えて頂いた小澤時典、矢島章夫両氏に感謝致します。またＢｅＳＩＭの開発に協力頂いた高倉正博氏に感謝致します。

参考文献

[1] Y.Ohno et al., "Principles of Design Automation System for Very Large Scale Computer Design", 23rd. DA Conf. Proc., June, 1986, pp.354-359.

[2] Y.Kazama et al., "Algorithm for Vectorizing Logic Simulation and Evaluation of VELVET Performance", 25th. DA Conf. Proc., June, 1988, pp.231-236.

[3] T.Shimizu et al., "A Logic Synthesis Algorithm for the Design of a High Performance Processor", Proc. ISCAS'85, July, 1985, pp.407-410.

[4] R.Cheng et al., "Functional Simulation Shortens the Development Cycle of a New Computer", 20th. DA Conf. Proc., June, 1983, pp.515-519.

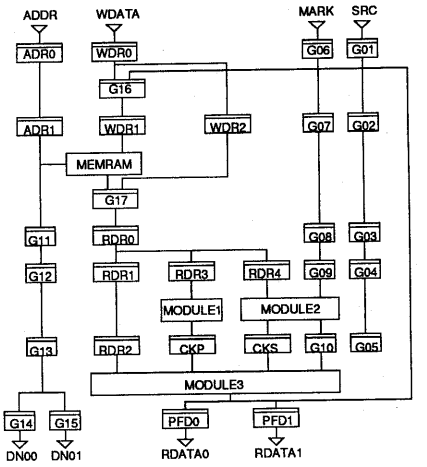
[5] K.Tham et al., "Functional Design Verification by Multi-level Simulation", 21st. DA Conf. Proc., June, 1984, pp.473-478.

[6] S.P.Smith et al., "Demand Driven Simulation BACKSIM", 24th. DA Conf. Proc., June, 1987, pp.181-187.

[7] S.Y.Hwang et al., "Incremental Functional Simulation of Digital Circuits", Proc. ICCAD'87, Nov., 1987, pp.392-395.

[8] 高橋, "ブール式を要求駆動で評価するＲＴレベルシミュレーション", 情報処理学会設計自動化研究会資料51-8, Feb., 1990.

[9] 水野他, "機能レベルシミュレーションの一手法", 情報処理学会設計自動化研究会資料50-5, Dec., 1989.



(a) ブロック図

```
BLOCK:     MCU ;
MEMORY:    MEMRAM RAM, ADDRESS=ADR1(12:25), DATA=WDR1 ;
REGISTER:  ADR0(0:31), ADR1(0:31),
           WDR0(0:71), WDR1(0:71), WDR2(0:71) ,
           RDR0(0:71), RDR1(0:71), RDR2(0:71) ,
           RDR3(0:71), RDR4(0:71), CKP(0:7), CKS(0:8),
           PFD0(0:71), PFD1(0:71) ;
EXMODULE:  MODULE1 : CODEGEN ;
             IN  :  DIN1=RDR3 ;
             OUT :  DOUT1;
           MODULE2 : SYNDROM ;
             IN  :  DIN1=RDR4, DIN2=REL(MARK) ;
             OUT :  DOUT1, DOUT2, DOUT3 ;
           MODULE3 : CORRECT ;
             IN  :  DIN1=RDR2, DIN2=CKP, DIN3=CKS,
                    DIN4=REL(MODULE2.DOUT1),
                    DIN5=REL(MODULE2.DOUT2) ;
             OUT :  DOUT1 ;
INPUT:     REQ, ADDR(0:31), CODE(0:8), SRC(0:2),
           WDATA(0:71), MARK(0:8), STYPE, TA ;
OUTPUT:    RTN0, DN00(0:2), RDATA0(0:71) = PFD0,
           RTN1, DN01(0:2), RDATA1(0:71) = PFD1 ;
SEQUENCE:  REQUEST  << REQ & !RDY(ADR0) >>
  *1:TA    ADR0 := ADDR ;
           IF CODE = 80H THEN START FETCH ENDIF ;
           IF CODE = 40H THEN
              WDR0 := WDATA ;
              IF STYPE THEN START FULL_STORE
              ELSE START PARTIAL_STORE ENDIF ENDIF ;
  END ;
SEQUENCE:  FETCH  << CALLED >>
  *2:TA    ADR1 := ADR0 ;
  *3:TA    ;
  *4:TA    RDR0 := MEMRAM ; START FOUT ;
  *5:TA    RDR1 := RDR0 ; RDR4 := RDR0 ;
  *6:TA    RDR2 := RDR1 ; CKS := MODULE2.DOUT3 ;
  END ;
SEQUENCE:  FOUT  << CALLED >>
  *1:TA    IF REL(SRC(0)) THEN RTN0 := PULSE FOR(1) ;
           IF REL(SRC(1)) THEN RTN1 := PULSE FOR(1) ;
  *2:TA    ;
  *3:TA    IF REL(SRC(0)) THEN PFD0 := MODULE3.DOUT1 ;
           IF REL(SRC(1)) THEN OFD1 := MODULE3.FOUT1 ;
  END ;
SEQUENCE:  FULL_STORE  << CALLED >>
  *2:TA    ADR1 := ADR0 ; WDR1 := WDR0 ;
  END ;
SEQUENCE:  PARTIAL_STORE  << CALLED >>
  *2:TA    ADR1 := ADR0 ; WDR2:=WDR0 ;
  *3:TA    RDR0 := WDR2 ;
  *4:TA    RDR1 := RDR0 ; RDR3:=RDR0 ; RDR0 := MEMRAM ;
  *5:TA    IF (REL(MARK)) THEN
              RDR1 := RDR0 ; RDR3 := RDR0 ENDIF ;
           RDR4 := RDR0 ;
  *6:TA    RDR2 := RDR1 ; CKP := MODULE1.DOUT1 ;
           CKS := MODULE2.DOUT3 ;
  *7:TA    WDR1 := MODULE3.DOUT1 ;
  END ;
END:       MCU ;
```

(b) B^{2}DLによるハードウェア記述

付図1 B^{2}DL記述例