ソフトウエアシミュレーションによるシステムLSI設計・検証ツールの評価

池辺 貞郎[1)]　平尾 智也[1)]　伊達 博[2)]
細川 利典[2)]　中西 恒夫[1)]　福田 晃[3)]

[1)] 奈良先端科学技術大学院大学情報科学研究科
[2)] 半導体理工学研究センター
[3)] 九州大学大学院システム情報科学研究院

概要

開発初期に各モジュールをハードウエア／ソフトウエアのいずれで実装するか定めにくいシステムLSIの開発において，設計ならびに検証を容易にするべく，ハードウエアならびにソフトウエア全体をソフトウエアシミュレーションすることが行われるようになっている．本稿では，このようなソフトウエアシミュレーションによるシステムLSI設計・検証ツールCoMETについて，ハードウエアまたはソフトウエアの切り分けの自動化の可能性，ならびにシミュレーションに要する時間を評価する．切り分けの自動化は、ハードウエアモジュールとソフトウエアモジュールのインタフェース生成については期待されるが，タイミングなどのハードウエア特性や，ハードウエアモジュールとソフトウエアモジュール間の同期については問題点が残るものであった．また、シミュレーション時間はモジュールの構成法によって大きく変わった．

Evaluation of a Software Simulation Based System LSI Design and Verification Tool

Sadao Ikebe[1)]　Tomoya Hirao[1)]　Hiroshi Date[2)]
Toshinori Hosokawa[2)]　Tsuneo Nakanishi[1)]　Akira Fukuda[3)]

[1)] Graduate School of Information Science
Nara Institute of Science and Technology
[2)] Semiconductor Technology Academic Research Center
[3)] Graduate School of Information Science and Electrical Engineering
Kyushu University

Abstract

It is a intractable task in earlier phases of system LSI design to make decision for each component whether be implemented as a hardware component or a software component. This software simulation of the whole system including both hardware and software components is an attractive approach for design and verification of the system LSI. This paper reports evaluation of Comet, a system LSI design and verification tool by software simulation of the whole system. Possibility of automatic hardware/software separation and time for simulation are evaluated. Automatic interface synthesis between hardware and software components was expectable, however, problems of automatic hardware/software separation were remained in hardware properties such as timing and synchronization between hardware and software components. Simulation time was largely dependent on organization and configuration of components.

1 はじめに

近年，LSI の集積度は年率 50%以上の勢いで伸びている．集積度の向上とともに複数の機能回路をひとつの LSI にまとめたシステム LSI や，システム全体をワンチップにまとめる SoC(System on Chip) が注目を集めている．システム LSI を構成するモジュールの実装方法はシステム全体の性能，回路規模などに大きな影響を与える．ハードウエアはその構成を容易には変更できないため，各々のモジュールをハードウエア，ソフトウエアのいずれで実装すべきかを，設計の初期の段階で決定することが望ましい．しかし，どのようにハードウエア／ソフトウエア分割を行うべきかは熟練した技術者の経験によるところが大きく，設計から完成まで，特に検証には時間がかかる．一方，携帯電話をはじめとして，電子機器の製品サイクルは，その機能の豊富さにもかかわらず極端に短くなってきており，開発工期の短縮が望まれている．

そこで，ハードウエアの設計が完了していない段階でハードウエア／ソフトウエアを含めたシステム全体をシミュレーションし，システム LSI 上で動くアプリケーションのアルゴリズムを評価し，ハードウエア／ソフトウエアの切り分けを決定することが行われるようになっている．このようなシミュレーションによるシステム LSI の評価は，非常に長いシミュレーション時間を必要とするため，シミュレーション時間をどれだけ短縮できるかが重要となる．

本稿では，システム全体のソフトウエアシミュレーションにより，ハードウエア／ソフトウエアの切り分けを円滑に行う手法を紹介する．また，この手法に基づくシステム LSI 設計･検証ツールである CoMET (VaST Systems Technology 社製) [1] を，ハードウエア／ソフトウエア切り分け自動化の可能性ならびにシミュレーションに要する時間について評価する．

本稿の構成を次に示す．第 2 節で CoMET の概要を述べる．第 3 節で CoMET を用いた自動ハードウエア化の可能性に関して考察する．第 4 節ではシミュレーション時間計測の結果を報告し，またシミュレーション時間短縮の可能性に関して考察する．第 5 節で結論を述べる．

2 評価対象ツール CoMET の概要

本節では CoMET におけるハードウエア／ソフトウエア分割の手順について述べる．

2.1 ハードウエア・ソフトウエアの分割

システムの各機能モジュールはハードウエアもしくはソフトウエアのいずれかで実現される．ハードウエアで実現されるものはハードウエア記述言語で，ソフトウエアで実現されるものはプログラミング言語で記述される．しかし，ハードウエアまたはソフトウエアのいずれで実現するのかが決定されていないモジュールは，ハードウエアともソフトウエアとも解釈できる記法で記述されなければならない．

ハードウエア，ソフトウエアのいずれで実装するか定まっていないモジュールをプログラミング言語で記述する方法が考えられる．ソフトウエアとして実現する場合にはそのモジュールをそのまま用い，ハードウエアとして実現する場合にはそのような “振る舞い (behavior)” をするハードウエアとして捉える方法である．

この方法の利点として，モジュールの振る舞いは確定しているが，ハードウエア実装が済んでいない段階，つまり，ハードウエア／ソフトウエア分割する以前の段階であっても，システム全体のシミュレーションが可能な点が挙げられる．

今回の評価に用いる CoMET では，ハードウエアで実装するモジュールは VerilogHDL，もしくは VHDL を用いて記述する．一方，ソフトウエアで実装するモジュール，もしくはハードウエアとして実装するかソフトウエアとして実装するかが確定していないモジュールは C/C++で記述する．

これらのモジュールをソフトウエアで実現する状況をシミュレートする際は，その C/C++のコードは VPM(Virtual Processor Model) と呼ばれる仮想的なプロセッサ上で動作するものとみなされる．一方，ハードウエアで実現する状況をシミュレートする際は，その C/C++コードはハードウエアの “振る舞い” を記述するものとみなされる．前者を VPM C コードと呼び，後者を Behavioral C コードと呼ぶ．

2.2 Behavioral C を用いたハードウエアモジュールの記述

前述のとおり，CoMET ではハードウエア／ソフトウエアのいずれで実装するかが決定されていないモジュールについては C/C++で記述する．

Behavioral C コードを用いたハードウエアモジュールを構成するとき，プログラマはモジュールのインタフェース，具体的にはポートのみをハードウエア記述言語により記述する．

モジュールの実装はハードウエア記述言語では記

述せず，Behavioral C コードの関数として記述する．ハードウエア記述言語では PLI(Programming Language Interface) による Behavioral C 関数の呼び出しのみを記述する（図 1）．

```
module HW_MODULE(trigger);
    input trigger;
    always@(posedge trigger)
    begin
        $vpa_function();
    end
endmodule

int function(){
    ...
}
```

図 1: ハードウエアモジュールと Behavioral C コードの対応

ハードウエアの振る舞いを記述するためには，Behavioral C コードからハードウエアモジュールのレジスタやポートなどにアクセスすることが不可欠であるが，C/C++にはそのような機能は存在しない．また，C/C++ではゲート遅延などのハードウエアの特性を記述できない．そこで，CoMET では C/C++からハードウエアモジュールへのアクセスや，C/C++によるハードウエア特性の記述を実現するため，さまざまな API を用意している．

2.3 VPM C から Behavioral C へ

当初ソフトウエアで実装するつもりで記述された VPM C コードを Behavioral C コードとして取り扱うためには，いくつかの変更が必要になる．

以降，VPM C コード caller() が呼び出す関数 callee() を Behavioral C に書き換える場合を説明する．

まず callee() を PLI 呼び出しするハードウエアモジュールを記述する必要がある．次に VPM C コードでは引数などを用いて行っていたモジュール間のデータの授受を，ハードウエアモジュールのポートを介して行うよう書き換えなければならない．この変更は caller()，callee() 双方に必要となる．さらに caller() 内で callee() を呼び出していた部分を，ハードウエアモジュールを起動するためのトリガをかけるルーチンに書き換えなければならない．

これらの変更はプログラミング言語が計算機を抽象化して捉えているのに対し，Behavioral C コードはハードウエアの構造を強く意識しなければならないことに起因する．

3 自動ハードウエア化の可能性の評価

すでに述べたとおり，VPM C と Behavioral C の間にはいくつかの相違点があり，VPM C から Behavioral C への変換には，手作業によるコードの変更が必要になる．本稿では，VPM C コードから Behavioral C コードへの変換することを，ハードウエア化という．ハードウエア化は比較的単純な作業であるにもかかわらず，作業量が大きく，自動化が望まれる．

本節では，ハードウエア化の際に検討すべきことをまとめ，それらの検討課題について自動化の可否を考察する．

3.1 ハードウエア化する際の検討事項

VPM C コードを Behavioral C コードを書き換える際に，下記の事柄を検討する必要がある．

1. ハードウエア特性の記述

   Behavioral C コードはハードウエアの実装方法を記述するものではない．そのため，ゲート遅延などのハードウエアの特性は API を用いて Behavioral C コード中に記述されなければならない．

2. **caller()** とハードウエアモジュールとのあいだの同期処理

   caller() が callee() を呼び出していた部分のコードを，ハードウエアモジュールの起動ルーチンに書き換える必要がある．

   このときに，caller() がハードウエアモジュールの処理の終了を待つべきか否かを決定する必要がある．また，ハードウエアモジュールの処理の終了を待つ必要がある場合は，どのような同期アルゴリズムを用いるかを決定する必要がある．

3. ハードウエアモジュール／ソフトウエアモジュール間のインタフェース

   ハードウエアモジュールは Behavioral C コードの関数を PLI 呼び出しするばかりでなく，周囲のハードウエアモジュールとのデータのやり取りを行うためのインタフェースとなるポートやレジスタを備えなければならない．

   ハードウエアモジュールの記述を行うためには，モジュールが備えるべきポートやレジスタの個

数，bit 幅などを決定する必要があり，そのためには callee() に対してどのようなデータの入出力があるのかを解析しなければならない．

3.2 自動ハードウエア化の可能性の評価

前小節に述べたハードウエア化に関する 3 つの検討事項について，自動化の可否の観点から考察する．

1. ハードウエア特性の記述

   ハードウエア特性はその実装に依存して変化するため，ハードウエア特性を機械的に推測し，自動的に記述することは困難である．しかし，ハードウエアの実装を意識しないシミュレーションを行うのであれば，ハードウエア特性を固定，あるいは無視することで問題を回避できる．しかし，設計者によるチューニングの有無により，シミュレーション結果に大きな違いが出る可能性がある．

2. **caller()** とハードウエアモジュールとのあいだの同期処理

   どのような方法で同期を取るべきかを機械的に判断することは難しく，プログラマによる判断が必要となる．そのため，適切な同期処理を自動的に組み込むことは困難である．この問題も同期アルゴリズムを固定することで問題を回避できる．この場合も，チューニングの有無によりシミュレーション結果に違いが出る可能性がある．

3. ハードウエアモジュール／ソフトウエアモジュール間のインタフェース

   Behavioral C コードの関数への入出力を解析しなければ，ハードウエアモジュールのポートやレジスタを記述できない．関数へのデータの入出力の手段としては戻り値，大域変数，引数がある．

   戻り値が関数からの出力であることは明らかである．C ソースコードの簡単な解析により，容易にハードウエア化することが可能である．

   大域変数はハードウエアで実装されたレジスタと考えることもできる．大域変数を用いてデータを入出力することでハードウエアモジュールのポートの数や入出力の方向を気にすることなく VPM-C コードと Behavioral C コードとの間のデータの入出力が可能である．

   関数の引数は，そのままハードウエアモジュールのポートであると考えてほぼ差し支えない．しかし，引数が参照渡しである場合，その引数が関数への入力であるのか出力であるのかはソースコードからは容易に解析できない．そのため，ハードウエアモジュールにどのようなポートを設けるべきかがわからず，モジュールの記述が不可能である．この問題は，設計者がプラグマの挿入やコメントアウトなど何らかの方法で引数が関数への入力／出力のいずれであるかを示すことで，自動的にハードウエアモジュール化することが可能になる．あるいは，behavior の RTL を得ることができれば，関数の引数が入力であるか出力であるかの判断が可能になるため，モジュールの記述が可能になる．

   戻り値，大域変数，引数のいずれの場合でも入出力として扱うことができるのはビット列である．ポインタや構造体などの複雑なデータを扱うことはできない．

結論として，ハードウェアモジュールとソフトウェアモジュールのインターフェース生成については期待されるが，タイミングなどハードウェア特性や，ハードウェアモジュールとソフトウェアモジュール間の同期については問題点が認められた．

4 シミュレーション時間の評価

本節では，CoMET のシミュレーション時間に関する評価を行う．

4.1 評価方法

以下では，簡単な音声のフィルタリング (DSP) を行うアプリケーションを用いて CoMET のシミュレーション時間に関する評価を行う．このアプリケーションは，フィルタ処理のタスクを 2 つ含み，これらのタスクはハードウエア，もしくはソフトウエアのモジュールとして実装される．

それぞれをハードウエアもしくはソフトウエアのいずれで実装するか，またその実装方法などに関して条件を変え，以下の 8 つのパターンについて，ホスト PC 上でのシミュレーションの実行時間を計測する．

2 つのフィルタは，それぞれ VPC C もしくは Behavioral-C における関数として記述される．以下ではそれぞれの関数を N，T と表記する．

表 1: シミュレーション時間の計測結果

	アドレス 変換無	アドレス 変換有	キャッシュ モデリング	D キャッシュ モデリング	I,D キャッシュ モデリング
モデル 1	240	1,412	14,736	18,416	31,164
モデル 2	76,995	81,487	103,233	108,700	126,847
モデル 3	122,511	126,471	148,242	154,176	171,822
モデル 4	127,453	131,408	149,530	150,916	165,552
モデル 5	178,561	185,707	209,926	197,879	215,645
モデル 6	179,437	185,711	210,447	198,600	215,519
モデル 7	224,002	229,900	255,336	244,656	260,589
モデル 8	227,421	231,727	253,419	254,926	270,954

(表中の値は，clock() 関数によって計測した実行時間)

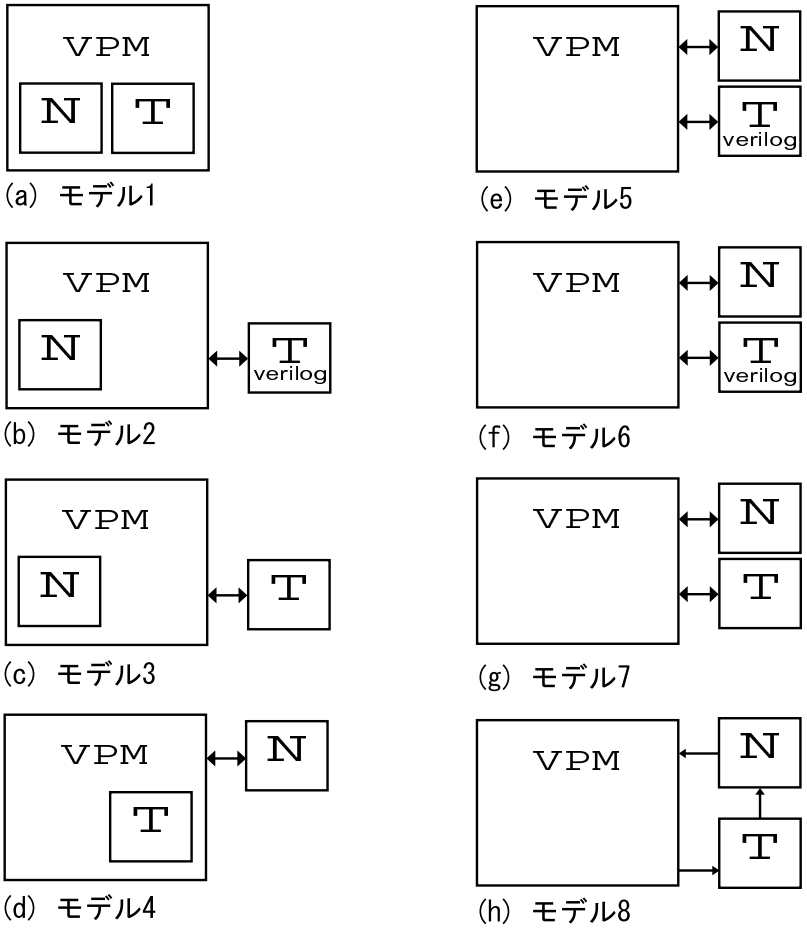

図 2: シミュレーション時間計測用のモデル

- モデル 1: すべてをソフトウエアで記述する場合. (図 2(a))
- モデル 2: T を Verilog でハードウエアとして記述し，タイミング用の信号線とデータ用の信号線でそれぞれのハードウエアを接続 (4-cycle signaling) して通信を行う場合. (図 2(b))
- モデル 3: T を Behavioral-C でハードウエアとして記述する場合. (図 2(c))
- モデル 4: N を Behavioral-C でハードウエアとして記述する場合. (図 2(d))
- モデル 5: N を Behavioral-C で，T を Verilog でそれぞれハードウエアとして記述する場合. (図 2(e))
- モデル 6: N を Behavioral-C で，T を Verilog でそれぞれハードウエアとして記述する場合. T に関しては，信号の立ち下がりを検出する際にに，Verilog の@(negedge) 表記を用いて検出するのではなく，Behavioral-C 版と同様にポーリングを行っている. (図 2(f))
- モデル 7: N，T ともに Behavioral-C でハードウエアとして記述する場合. (図 2(g))
- モデル 8: N，T ともに Behavioral-C でハードウエアとして記述する場合. ただし，N の結果を直接 T に渡すよう，ハードウエア同士で直接通信を行っている. (図 2(h))

4.2 実験結果

実行時間の計測には，ホスト PC の OS で提供されている clock() 関数を使用する．各モデルについて，アドレス変換の有無，キャッシュシミュレーションの有無に関しての条件を変え，5 パターンの計測を行う．

結果は表 1 の通りとなり，シミュレーション時間はモジュールの構成法に大きく依存している．

表のデータより，関数をハードウエア化したもののシミュレーションを行うと，大幅に速度が低下する．上の例では，1 つのフィルタ関数を 4-Cycle Signaling で通信を行うハードウエアに置き換えるごとに，約 100,000〜120,000 サイクルの速度低下が見られる．

また，Verilog で書いたハードウエアの動作速度と，Behavioral-C での動作速度を比較すると，約 1.5〜2.0 倍 Verilog のコードの方が高速に動作する．ハードウエアで実装することが確定しているモジュールについては，Behavioral C でなく，Verilog コードを使用することで，シミュレーション時間を大きく節約できる．

5 まとめ

本稿では，システム LSI 設計・検証ツールである CoMET において，プログラミング言語で記述した機能モジュールをハードウエア化する際の検討事項をまとめ，その自動化に関する可否を評価した．

機能モジュールのハードウエア化の際には，タイミング等のハードウエア特性の記述，ハードウエアモジュールとソフトウエアモジュールの間の同期処理と通信のためのインタフェースについて，検討する必要が生じる．自動ハードウエア化の際には，これらの検討を自動的に行うことが要求される．

ハードウエア特性の記述，ハードウエアモジュールとソフトウエアモジュール間の同期処理の記述に関しては，ハードウエア特性や同期アルゴリズムを固定するなど，記述に制限を加えない限り，自動的なハードウエア化は困難であった．一方，ハードウエアモジュールとソフトウエアモジュールの間の通信のためのインタフェースについては，CoMET の拡張か支援ツールによりデータフロー解析を行うか，プログラマによるヒントにより自動的なハードウエア化は可能と思われる．

さらに本稿では，実際に CoMET を用いたシミュレーションを行い，シミュレーションに要する時間を計測した結果，Behavioral C コードで記述されたハードウエアモジュールのシミュレーションに多くの時間が費やされることが確認された．

謝辞

今回の CoMET の評価の機会を設けていただいた株式会社半導体理工学研究センター（STARC）の吉田憲司部長，村岡道明室長ならびにアーベル・システムズ株式会社の鈴木文雄社長に謝意を表します．本研究開発は，新エネルギー・産業技術総合開発機構 (NEDO) から半導体理工学研究センターに委託された「SoC 先端設計技術の研究開発」の一部として実施されています．

参考文献

[1] http://www.vastsystems.com/