社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2008－SLDM－133（32）
2008 / 1 / 17

高速HW-SW協調検証モデル向けCtoHDL変換コンパイラ

伊藤 康宏† 菅原 豊† 平木 敬†

† 東京大学大学院情報理工学系研究科
E-mail: †{ladiemon,sugawara,hiraki}@is.s.u-tokyo.ac.jp

あらまし 組み込みシステムでは回路規模の増加と対象ソフトウェアの複雑化のため動作検証のコストが増し，開発期間を圧迫している．検証環境に於いてはサイクル精度の高速な検証，同一コードからの検証モデルとRTLの生成による工数削減が求められている．高速な検証手法の一つとして，Callbackをベースにしてハードウェアモデルを構築する手法が既に存在する．Callbackとはある信号の変化に対し，0からNサイクル後に登録した関数を呼ぶ仕組みを指す．この手法は，既存のSystemC等ベースの検証環境と比べ高い検証速度を持つがRTLの生成能力を持たないため，RTLと検証モデルは別個に実装され，工数が多くなってしまう．本研究では，Callbackベース検証モデルで高い検証速度とRTL生成能力の両立を目標とし，Cで記述されたCallback関数からVerilogHDLへのコード変換器を開発した．評価ではCPU，メモリ，通信モジュールを持つSoCをCallbackベース検証モデルとVerilogの両方で実装し，それぞれの検証速度とRTLの規模を比較した．Callbackベース検証モデルを用いた場合，Verilogによる場合に比べ検証速度は24倍，回路規模及び周波数は同等であった．

C to HDL compiler for rapid HW-SW co-simulation models

Yasuhiro ITO†, Yutaka SUGAWARA†, and Kei HIRAKI†

† The University of Tokyo Graduate School of Information Science and Technology
E-mail: †{ladiemon,sugawara,hiraki}@is.s.u-tokyo.ac.jp

Abstract The importance of verification for embedded systems increases as the scale of circuit and complexity of software increase, and the time for verification step also increases in development period. The verification environments should have the following two aspects: One is high-speed verification with cycle level accuracy. The other is to reduce man-hour by generating both the verification model and RTL from the same code. There is a proposed approach for high speed verification , which constructs hardware models with callbacks and calls registered callback functions after 0 or more delays reacting on change of the trigger signal. This approach have higher verification speed compared to existing approaches such as SystemC. However, it cannot generate RTL, thus it requires more man-hour for constructing verification model and RTL separately. We implemented a code converter for generating Verilog code from callback functions described in C. It is aimed to achieve both high verification speed and the RTL generative capacity. We implemented a SoC for evaluation by using both the callback-based verification model and Verilog. We measured the verification speed and the scale of RTL of both methods. With our method, the verification speed is four times faster than that of Verilog, with equal clock frequency and circuit scale. We show that the verification speed of our method was twenty four times faster than Verilog.

1. はじめに

組み込みシステムのハードウェア規模とその上で実行されるソフトウェアの大型化のため，動作検証にかかるコストは増大し続けている．その一方で組み込み製品の開発期間は短縮され続けているため，組み込み検証環境の高速化，効率化が課題となっている．サイクル精度の検証を高速に行う事により実行可能な検証パターンを増やす事，製品となるRTLを生成するソースコードに手を入れる事無く検証を可能にする事により検証自体にかかる工数を減少させる事が求められている．

現在数多くの組み込みシステム検証環境が存在する．SystemCやHDLをベースとした検証はRTL生成と検証を一つのソー

スコードで賄えるものの, 単位時間ごとに各モジュールのイベントを評価する検証モデルを持つため, 速度が低く十分な数のテストパターンを実行する事ができない.

一方, 高速な検証環境の一つとして,Callback ベースでハードウェアモデルを構築する方法が提案されている [1].Callback ベース手法とは, 入力の信号またはレジスタの値の変化が起こった際, 0から Nnsec の遅延後に事前に登録された Callback 関数を呼ぶ仕組みを組み合わせるものである. これは前者の検証方法に比べ, 極めて高い検証速度を持つが,RTL の生成能力を持っていない.

以上のように, 高精度の高速シミュレーションと検証工数の削減を両立した検証環境は未だに存在していない. 高速な Callback ベース検証モデルに RTL 生成能力を持たせる事で, 上記の要求を満たす事が可能である. しかし Callback ベース検証手法ではイベント駆動にモデル内の状態や信号の遷移を記述するため,RTL の記述モデルとは異なる. RTL を得るためには, 文法規則の置換のみでは不十分であり, 複雑なモデルの変換を必要としている.

我々は Callback ベース検証手法のイベント駆動型モデルから RTL のビヘイビアモデルへと変換を行うコンパイラを設計, 実装した. 我々のコンパイラが,Callback ベース検証モデルの高い検証速度を損なわずに, 高速かつ小サイズの RTL が出力できる事を示すため, 以下の評価を行った. 評価には Callback ベース検証手法と Verilog の両方で記述された同じ動作をする組み込みシステムのモデルを用いた. Callback ベース手法と RTL ベース手法の検証速度の比較を行った. Callback ベースモデルを我々のコンパイラで Verilog に変換したものと, 人手で記述された Verilog を合成し, 動作周波数と回路規模を比較した.

以下 2 章では, 組み込みシステムの手法, 対象などを説明し, 3 章で今回実装したコンパイラの内部を説明し, 4 章で実装したコンパイラの性能評価を行い, 5 章で検証環境の高効率化と関連した先行研究との比較を行い, 6 章でまとめる.

2. 組み込み検証環境

2.1 組み込み検証環境への要求

組み込みシステムは携帯端末, 車載コントローラとあらゆるところにあり, CPU, メモリ, バスを中心としてアナログ機器とのインターフェイスや通信機器を内蔵する, 独立した計算機である. ハードウェアの高速化により汎用ソフトウェアを組み込みシステム上で動作可能になり, 組み込み Linux カーネルを動かした上でユーザアプリケーションなどを動作させる例が増加している. このように, 組み込みシステムの使用される分野, 頻度が高まっている. それに従って, クロックレベルでの周辺機器のタイミング検証と, 大量の資源を必要とする複雑なユーザーアプリケーションのデバッグまたはチューニングを行いたいという要求が高まっている. 故に, 種々の高速化手法が研究されている.

2.2 HDL ベース検証手法

既存の環境の検証実行モデルについて概説する. 検証はシミュレータに与えられた単位時間ごとにカウンタの値を進められていく. 検証環境に読み込まれた各モデルは, それぞれ個別にスレッドを生成させる. 個々のスレッドは, カウンタの値が更新されるごとにイベントの評価を行い, それぞれの内部状態を更新していく. 実行前に与えられた単位時間ごとにしか検証は進まないため, 内部状態が変更する状態, 何も状態が変わらない状態のいずれも, 一定の時間を消費してしまう. これは, 長い待ちのステートを伴うメモリコントローラ, 通信用周辺機器の検証において検証速度向上のオーバーヘッドを招いてしまう. このため, 通信機器, メモリコントローラなどを伴う組み込みシステムの検証において, HDL ベースの検証環境では実際の大きなアプリケーションを動かすには検証速度が不足である.

2.3 Callback ベース検証手法

Callback によるシミュレーションモデルは高速な検証の手法として提案されている. [1] 本稿ではこの手法の例として VaST Systems Technology の CoMET [2] と互換性をもつ,Callback モデルの構築と検証を取り上げる.

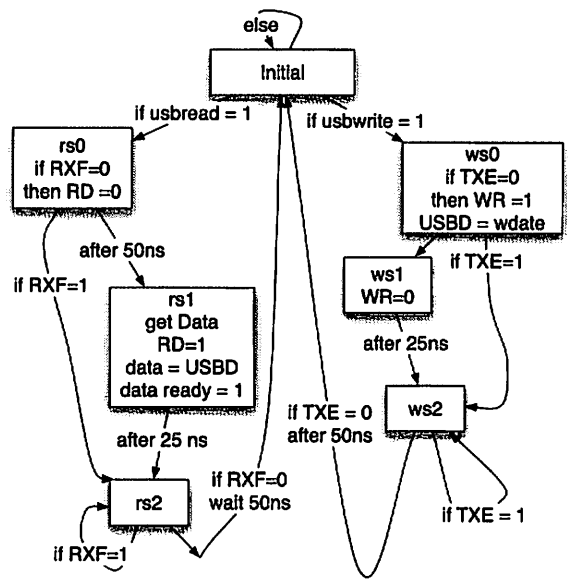

図 1 USB コントローラ模式図

2.3.1 Callback ベースモデルの構築方法

Callback ベース検証手法はイベントドリブンに各変数の遷移を記述してモデルを構築する. 信号の処理や状態遷移等を Callback 関数内で記述し, 入力ポートやレジスタの信号変化をトリガとして Callback 関数を呼ぶ, 所定の遅延の後に Callback 変数をスケジュールすることにより状態遷移が表現される.

今回, 例として, 図 1 に示される状態遷移を持つ USB コントーラを使用する. これの Callback モデルの記述例が図 2,3,4 である, それぞれはモデルを構築する信号, ステート,Callback の抽象データ構造の宣言, 宣言した各データ構造の初期化,Callback 関数の記述に対応している. Callback ベース検証モデルは, クロック, リセット信号, 入出力ポート, レジスタ等の各要素のハンドル, Callback のハンドル, ステート管理用変数,Callback 関数

を組み合わせて構築する. ハンドルとは各要素の関数ポインタ, 変数名等を持つ抽象データ構造である. 初期化メソッドでは各ハンドルを初期化する. ここでは各信号の bit 長の指定, 信号が変更した時に呼ばれる Callback 関数の指定, Callback ハンドルに Callback 関数とこの Callback の同期クロックの対応付けを行う. Callback 関数の内部では, ステートの遷移, 出力ポートの値の変化, 内部状態の変更を記述する. また Callback ハンドル, 呼び出し遅延, 遷移先のステートを引数として,Callback 関数を Schedule する事ができる. 例では ScheduleCallbackTime を使っている. 入出力ポート, レジスタの値は各ハンドルへの参照へアクセスによって通常の変数のように値の読み書きが可能である. 以上の工程を組み合わせる事により Callback ベースの検証モデルは構築される.

2.3.2 Callback ベースモデルの検証過程

今回取り上げた Callback ベース検証システムは, イベントの実行されるタイムスタンプを優先度とした Callback 関数のポインタの Queue を持っており, 検証は Queue の中にある Callback 関数をタイムスタンプが若い順から実行するモデルで実現されている. Callback イベントの実行によってポートやレジスタ値が変更され, これに反応して SlaveCallback 関数が連鎖的に呼ばれる. この間はシミュレーション時間は進まない. また Callback の Schedule は関数ポインタと実行されるべきタイムスタンプの Queue への挿入と言う形で表現される. 以上のように,Callback ベース検証モデルは途中何もイベントの無い箇所での処理を完全に省く事ができ, 対象ハードウェアの振る舞いを登録された Callback 関数の呼び出しで表現できるため, スレッドを作るオーバーヘッドを発生させる事なく検証の高速化を可能としている.

```
struct usbcont{
  STATE State;
  SlavePort *RE;
  SlavePort *WR;
  SlavePort *Data_in;
  MasterPort *Data_out;
  MasterPort *busy;
  MasterSlavePort * USB_D;
  MasterPort      * USB_WR;
  MasterPort      * USB_RDX;
  SlavePort       * USB_TXEX;
  SlavePort       * USB_RXFX;
  Register        * OUT_DATA;
  ScheduleCallback *WaitWRHold;
};
```

図 2 USB モデル構造体宣言

2.3.3 Callback ベースモデルと RTL の相違点

Callback ベース検証モデルと実際のハードウェアでは複数の相違点があり,Callback ベースから RTL の変換への課題となる. 以下に相違点を列挙する.

```
0:void InitModuleInstance(){
1: struct usbcont * MOD;
2: MOD -> State=InitModuleParentalState(s0,rs0,
3:                                      rs1,rs2,ws0,ws1,ws2);
4: MOD->CLK=InitCLKSlaveHandle();
5: InitModuleTargetSlaveClockTickINns(MOD->CLK,20);
6: MOD->RST=InitRSTSlaveHandle(Reset);
7: MOD->RE      =InitSlaveHandle(,REchange);
8: MOD->WR      =InitSlaveHandle(,WRchange);
9: MOD->Data_in =InitSlaveHandle();
10: MOD->Data_out=InitMasterHandle(8);
11: MOD->busy    =InitMasterHandle();
12: MOD->USB_D   =InitMasterSlaveHandle(8);
13: MOD->USB_WR  =InitMasterHandle(,0xA);
14: MOD->USB_RDX =InitMasterHandle(,0xA);
15: MOD->OUT_DATA=InitRegisterHandle(8);
16: MOD->WWRHold =InitTaskCallback(WRHold,MOD->CLK);
17: MOD->WTXEXSet=InitTaskCallback(TXEXSet,MOD->CLK);
18:}
```

図 3 USB 検証モデル初期化メソッド

```
0:void REchange(struct usbcont * MOD){
1:  if(MOD->State == s0){
2:    if(MOD->USB_RXFX == 0){
3:      MOD->USB_RDX = 0;
4:    }
5:    else{
6:     MOD->State = rs2;
7:   }
8: }
9:}
10:
11:void WRHold(struct usbcont *MOD){
12:  if(MOD->State == ws0){
13:    MOD->USB_WR = 0;
14:    ScheduleCallbackTime(MOD->WTXEXSet,25);
15:     MOD->State = ws1;
16:  }
17:}
```

図 4 USB モデル Callback 関数

(1) Callback ベースの検証環境では, ハードウェアのビヘイビアが Callback のみにより記述される. しかし, 実際のハードウェアモデルは, 組み合わせ回路と順序回路によって構成される.

(2) Callback の遅延時間について, 検証モデルでは時間単位とクロック tick 単位の双方をサポートしているが, 実際のハードウェアではクロック同期のフリップフロップが使用されるため, すべての遅延はクロックサイクルによってのみ表現される.

(3) Callback ベース検証モデルでは, 図 2,3,4 の用に Callback 関数を Schedule によって接続するイベントドリブン型の

記述である. 実際のハードウェアでは, 各フリップフロップごとの, Enable 条件と条件ごとに代入される値の列挙の集合である. このため,Callback ベースの検証モデルから RTL への変換は syntax を変更する簡易な C マクロだけでは不十分であり, 構文の解析を伴ったコンパイラの実装が必要である.

3. Callback to HDL コンパイラ

高速なシミュレーションと高速で小サイズな RTL の生成を一つのソースで賄う手法としては, 検証モデルから RTL 出力するアプローチとその逆の二つがある. 前述のように検証の高速化のためには, 検証するモデルの内部状態が変わらない時間の省略する事の効果が高いが,RTL から検証モデルを生成する場合, この待ちステートの抽出は変換コンパイラの自動最適化に委ねられるため. 最適な検証速度を保証する事が難しい.

Callback ベース検証手法では, 検証モデルはイベントに同期して内部状態の変更を記述するスタイルであり, これはクロックイベントに同期してレジスタやポートの値を更新する RTL の記述スタイルに類似するものであり, 最適化が容易である. しかし Callback ベースのモデルでは Callback 関数の中に複数の信号の変化を記述するのに対し,RTL では各信号ごとに遷移条件と変更する値を記述するモデルを持つため, Callback ベース検証手法から RTL の生成を可能にするためには, この記述モデル間の変換を行う必要が有る.

今回, 我々が実装したコンパイラは (1) 組み合わせ回路と順序回路の判別, (2) 代入先オペランドによる構文木のソート, (3) 時間からクロックへの待ちステートの変換を行う事で, このモデルの変換を実現し, RTL 合成が可能な Verilog ソースコードの出力を可能にした. 以下に本コンパイラの行う変換を解説する.

3.1 入力仕様

我々のコンパイラは C 言語で記述されたハードウェア検証モデルのコードを検証時に使用されるライブラリへのリンクをせずに受け取り, これを直接 Verilog のハードウェアモデルへと変換する. 変換の行程としては, まず 2,3,4 の様に記述された検証モデルのソースコードをそのまま字句・構文解析にかける. 字句構文解析機は C 言語の文法規則のサブセットと前章で解説した Callback モデルを構成する要素のみを受け取るこのため, 今回我々の実装した C コンパイラは Callback 検証モデル専用であり一般の C 言語を受容しない.

3.2 変換工程

入力されたソースコードは, 構文解析によって構文木へと変換される. 構文木は変数名と型の対応表, 変数とオプションのノード Callback 関数のリストの 3 つの大きなノードにより構成される. Callback 関数リストの構造変換を行う事で Verilog への変換を行う.

3.3 組み合わせ回路, 順序回路の推論

2.3.3 節で言及したように Callback ベースの検証モデル構築では, 組み合わせ回路も順序回路も同じく Callback によって表現される. よってこれらを自動で判別する必要がある. 我々のコンパイラでは, 以下の条件をすべて満たす変数代入の表現を組み合わせ回路と推論し, それ以外を組み合わせ回路と推論する.

(1) 値の変化時に呼ばれる Callback 関数内で記述される.

(2) 代入の destination が他の Callback 関数に出現しない.

この条件を満たしているとき,destination の変数は wire 変数として, それ以外は reg 変数として Verilog に変換される.

3.4 時間からクロックへの変換

Callback ベースの検証モデルでは遅延時間をナノ秒単位でも指定可能だが, コンパイラでこれをクロックサイクル数に変換する. 図 3 の初期化メソッド 5 行目のように, InitModuleTargetSlaveClockTickINns を用いてクロックサイクルの長さをナノ秒単位で指定できる. これで遅延時間を除算し端数切り上げを行い, 遅延サイクル数相当の自動生成ステートとステート遷移を構文木に追加する. 記述例の図 4 の 14,15 行目は 25nsec 後に TXEX を呼び出し, ステートを ws1 に進めるという記述である. Callback ハンドル WTXEXSet は CLK に関連づけられており,CLK のサイクル長は 20nsec なので,25nsec の遅延は 2 サイクルである. 以上から, 構文木のこの 2 行に相当する部分は

```
if(State == ws0)
  State = WaitTXEXSet_State_ws1_0;
else  if(State == WaitTXEXSet_State_ws1_0)
  State = ws1;
```

に置き換えられ,WaitTXEXSet_State_ws1_0 がステートのリストに追加される.

3.5 構文木の組み替え

Callback ベース検証モデルの記述の場合,State 変数等一つの変数が複数の Callback 関数内に出現する. HDL ではこのような表現をサポートしないため, 各 Callback 関数に対し HDL への syntax 変換を行っただけの HDL では合成が不可能である. よって我々のコンパイラでは, 各変数ごとに Process 文または条件代入文に変換されるように, 構文木に対して代入先オペランドによるソートを行う. 手順としては, まず Callback 関数の記述を再帰的にたどっていき,{代入先オペランド, 条件節, 代入表現}と言う形にしてハッシュテーブルに登録する. 次にハッシュのエントリを代入先オペランドでソートしグループにまとめる. これにより HDL に変換可能な構文の構造が得られる. 記述例の図 4 の 6,15 行目では異なる Callback 関数で同じ State 変数に対する代入が出現している. まず各代入表現は

```
{MOD->State,
     (MOD->State==s0&&not(MOD->USB_RXFX==0)),rs2},
```

{MOD->State,(MOD->State==ws0),ws1} へと変換されハッシュテーブルに登録されソートの後グループ化され, 最終的に

```
{MOD->State,[
  {(MOD->State==s0 && not (MOD->USB_RXFX==0)),rs2}
  {(MOD->State==ws0),ws1}
    ...
  ]
}
```

というデータ構造に変換され, 後はこのデータ構造ごとに Verilog の Process 文もしくは組み合わせ回路の条件代入文へと変換される. 以上の変換過程により, 我々のコンパイラは Callback ベース検証モデルからの Verilog ソースへの変換を行う.

4. 性能評価

本コンパイラが,Callback ベース検証モデルの高い検証速度と, 性能の高い RTL の出力を両立させている事を示すため, 以下の評価を行った. 比較対象として, 我々の手法と同じく, 既存の同一ソースから検証と RTL の合成が可能な手法の中から Verilog で記述されたモデルを Modelsim で検証する手法を選択した. 他に SystemC,VHDL を利用可能であったが, 出力されるソースのフォーマットを統一して RTL 合成の評価を適正にするために除外した. また検証環境は Callback ベースと同じ動作条件とするため Windows 上で動き, サイクル精度で高速な検証が可能な検証環境として代表的な ModelSim を選択した.
評価アプリケーションには実際に用いられる組み込みシステムの要素である, 通信と演算の機能を持たせるため CPU,RAM, 通信モジュールを組み込んだシステムを設計し, これを Callback ベースと Verilog の双方で実装し, それぞれのソースコードを評価に用いた.
まず Callback ベースと Verilog ベースで記述された各モデルの検証速度の比較を行い,Callback ベースが 24 倍の検証速度を持つ事を示した. 次に Callback ベースモデルを我々のコンパイラで Verilog に変換したものと, 人手で記述された Verilog のモデルをそれぞれ合成し, 動作周波数と回路規模を比較し同程度の回路が得られる事を示した.

4.1 評価対象モデル

検証速度と生成された RTL の性能評価のため,CPU, メモリ,USB 通信モジュールを搭載したハード上で, データ転送のアプリケーションを動かすシステムの検証を想定した. CPU は 16bit で加減乗算, 分岐,load,store,USB 通信モジュールとの通信命令を搭載している. メモリは IDT71T75602 を想定したコントローラと SRAM のモデルを実装した. USB モジュールは FT245BM を想定したモデルを実装した. それぞれのモデルの通信レイテンシは SRAM モデルは読み書きとも,4 クロック/1 ワード,USB は読み書きともに 100 クロック/1 バイトとした. これらを組み合わせたシステムの上で 260Kbyte のデータを USB 経由で受け取り, メモリに値を格納した後に, 再度 USB 経由で送り返すアプリケーションを動作させた. 以下の図は今回評価に用いたテスト環境の概要図である.

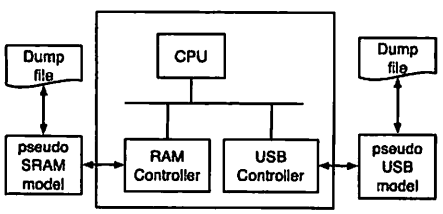

図 5 評価対象の検証モデル

4.2 評価環境

Callback モデルの構築および検証の実行には VaST Systems Technology 社の CoMET5, Verilog の検証環境には Mentor Graphics 社の ModelSim SE6.3,HDL の RTL 合成には Xilinx 社の ISE 8.2 をそれぞれ使用した. また,CoMET5 でサポートされている syntax と, 我々のコンパイラでサポートする C の syntax の間に相違があったため, 人手で修正を行った.

4.3 結　果

まずそれぞれの手法の工数を示すため, ソースコードの量を比較する. 以下の表 1 は Callback ベース検証モデルでのソースコード, 我々のコンパイラにより生成された Verilog ソースコード, 人手による Verilog ソースコードの順で表示している.
従来の Callback ベース検証手法では Callback 関数の記述と,

表 1 モデル記述量比較

記述手法	行数 (行)	文字数 (文字)
Callback モデル	684	13396
Callback モデル変換後 Verilog	663	14260
人手による Verilog	681	13845

さらに HDL によるハードウェアモデルの記述の両方が必要であった. 我々のコンパイラにより, 人手による Verilog の記述を省略が可能になり, 表 1 の例では約 700 行,14 キロバイト分の工数削減を可能にした.
次に Callback ベースの検証環境と Verilog 検証環境による検証の速度を比較する. まず USB モデルから 260Kbyte のデータを読み CPU を通じて RAM に格納し, 再度 RAM から読み出して USB モデルへと送信するアプリケーションを実行し, その場合のシミュレーション環境内部での実行時間と, 実際のシミュレータの実行時間の比較を行った. 結果は表 2 に示されるように, 最新の Verilog 検証環境に比べ Callback ベース検証環境は 40.02 倍の速度比がある事が示された.

表 2 検証速度比較 (USB 通信レイテンシ 100 クロック)

検証手法	検証時間 (ナノ秒)	実行時間 (秒)
Callback モデル	927,994,080	2.141
Verilog(最適化有効)	927,214,480	85.7
Verilog(最適化無効)	927,214,480	293.1

表 3 検証速度比較 (USB 通信レイテンシ 1 クロック)

検証手法	検証時間 (ナノ秒)	実行時間 (秒)
Callback モデル	1,886,084,320	1.968
Verilog(最適化有効)	1,884,818,960	47.6
Verilog(最適化無効)	1,884,818,960	137.6

次に,USB 通信のレイテンシを 1 クロックにし, 再度速度評価を行った. 結果は表 3 に示されるように,Verilog 検証環境に比べ 24.18 倍の速度差にとどまった. これは USB 通信のレイテンシが減少した事により, シミュレーション中に占める内部状態の変わらないステートの割合が減少したためである. 最後に Callback ベース検証モデルを我々のコンパイラで変換した Verilog のソースと, 人手による Verilog のソースを Xilinx ISE8.2 上で実際の FPGA をターゲットとして合成した. 両手法によって記述された CPU, メモリコントローラ,USB コントローラをそれ

表 4 RTL 合成結果比較値

モジュール	手法	周波数 (MHz)	回路規模 (Slice)	回路規模 (gate)
CPU	Callback モデル変換後 Verilog	64.3	967	247552
CPU	人手による Verilog	66.7	877	224512
USB	Callback モデル変換後 Verilog	287.77	14	3584
USB	人手による Verilog	270.416	7	1792
Memory	Callback モデル変換後 Verilog	263.223	23	5888
Memory	人手による Verilog	270.3	18	4608

ぞれ合成した際の最大動作周波数と, 回路規模を比較した. 合成結果は Slice という単位で与えられるため,1slice=256gate と換算して表記した. 結果は表 4 に示されるように, 今回構築した評価対象の組み込みシステムのすべての部品で我々のコンパイラが Callback ベースモデルから変換して出力した Verilog ソースが人手により記述されたそれと同程度の動作周波数, 回路規模を持つ事が確認された. 上記の結果をまとめると我々は CPU,RAM, 通信モジュールをもつ典型的な組み込みシステムを Callback ベース手法と HDL ベース手法の両方で構築し, それぞれの検証速度と生成される RTL の性能比較を行った. 今回使用した Callback ベースモデルは HDL ベースに比べ 24 倍の検証速度を持ち, 我々のコンパイラを用いて Verilog に変換した場合, 人手により記述された Verilog コードと同程度の性能の RTL が合成可能であった. これにより, 我々のコンパイラは Callback ベース検証モデルの高い検証速度を保ちつつ, 本来別々に実装する必要があった RTL と同程度の性能の Verilog ソースを Callback ベース検証モデルから生成可能である事が示された.

5. 関連研究

RTL の生成能力と検証速度を両立させるほかのアプローチとしては,HDL から高速なシミュレーションを行うというものが考えられる. Cynergy System Design 社の Afterburner [4] は,HDL で記述されたモデルから C 言語で記述されたシミュレーションモデルを生成し, これに検証用ライブラリをリンクさせる事で高速な検証を可能にしている. これは C で記述された検証モデルから合成可能な HDL を出力する我々の手法とは逆のアプローチだが, 同じく検証速度と RTL の生成能力を両立させる手法である.

また HDL シミュレータの Modelsim6.3 では,HDL で記述されたシミュレーション環境全体を受け取り, それをモジュールの階層をなくした検証モデルにコンパイルという過程を経て最適化する. これにより 2 の例では最適化前に対して, 3.42 倍の高速化を達成している. これらの手法は [5] で言及されているように, コンパイラの自動最適化による省略可能なステートの抽出を行っている. Callback ベース検証環境のモデリング手法ではユーザが明示的に省略可能なステートを挿入する. 従って, 待ちステートの数が多いモデルを含む環境下では検証速度の面で有利となる.

6. 結　　論

我々は, 単一のソースコードから効率の良い RTL と, 高速なシミュレーションモデルの双方の生成を可能にするコンパイラを開発した. これによって同じ時間に実行できるテストパターン数を増加させ, シミュレーションモデルと RTL を別個に実装する手間を削減した. 我々のコンパイラは,C 言語で記述された Callback 関数からなる検証モデルを Verilog のハードウェアモデルに変換するものである. 我々のコンパイラは, 以下の変換を行う.

（1） Callback 関数から同期クロックの抽出により順序回路か組み合わせ回路の自動判別を行う.

（2） 秒単位で与えられた Callback 関数の呼び出し遅延を, パラメタとして与えられた動作周波数に相当するクロック数に変換する.

（3） Callback 関数の Schedule メソッドを, ハードウェアモデル上の待ちステートへと変換する.

評価として, 通信機能をもつ典型的な組み込みシステムのシミュレーションモデルを HDL と Callback ベースモデルの両方で実装し, 性能比較を行った. 我々の Callback モデルは Verilog で記述されたモデルに対し,24 倍の検証速度で同程度の動作周波数と回路規模の RTL が合成可能であった. この結果により,Callback ベース検証モデルにおいて高速な動作検証速度と高性能の RTL 合成を両立可能な事が示された.

以降の研究では, 現在は未実装である Callback 関数のパイプライン化されたロジックへの自動変換を追加する事によって, 信号処理に対する記述の容易さを高める, 本コンパイラの適用範囲を通信用周辺機器に限らず, アクセラレータ DSP 等にも広げていく.

文　　献

[1] CPU Model-based Hardware/Software Co-designfor Real-Time Embedded Control Systems, Makoto Ishikawa D. J. McCune and George Saikalis Shigeru Oho, SAE World Congress,2007.

[2] CoMET5, VaST Systems Technology, http://www.vastsystems.com/

[3] ModelsimSE6.3c, Mentor Graphics,http://www.model.com

[4] Afterburner,http://www.cqpub.co.jp/dwm/eventreport/edsf2001/adac/cynergy2.pdf

[5] Compiler Optimizations in ModelSim - Balancing Performance and Visibility Mentor Graphics,Application Note for ModelsimSE6.3