計算機アーキテクチャ 145−12
（２００１．１１．２９）

PLIを用いたネットワークインタフェースコントローラとホストプログラムの協調シミュレーション

山本淳二[†] 渡邊幸之介[††] 宮脇達朗[†††]
西宏章[†] 工藤知宏[†] 天野英晴[††]

Verilog HDL で記述された ASIC の論理と、この ASIC を利用する C++ で記述されたホストプログラムの協調シミュレーション環境について報告する。Verilog の PLI 機能を使用して新しいシステムタスクを作成し、Verilog コードから UNIX の通信機能を使用できるようにした。また、いくつかのライブラリや C++ によるクラスを用意し、ホストプログラムがシミュレーション時に Verilog コードと通信をすることを可能にした。これらを用いて、実機動作時と基本的に同じホストプログラムのコードを使用できるシミュレーション環境を構築した。また、Martini と呼ぶネットワークインタフェースコントローラ ASIC をこのシミュレーション環境を用いて実際に開発した。

Co-simulation with host program and Verilog HDL by PLI

Junji Yamamoto,[†] Kounosuke Watanabe,[††]
Tatsuaki Miyawaki,[†††] Hiroaki Nishi,[†] Tomohiro Kudoh[†]
and Hideharu Amano[††]

In this report, cooperative simulation environment of ASIC logic written in Verilog HDL, and host processors program written in C++ language is introduced. In this environment, process which executes Verilog simulation and another process which executes a host program communicate with each other using a pipe provided by UNIX operating system. For the Verilog process, the PLI function of Verilog HDL is used to realize such communication. For the host program, C++ libraries and operation overloading is used for communication so as to use the same host program code as the real machine. Using such simulation environment, a network interface controller ASIC called Martini is developed.

1. はじめに

ASIC の開発では、設計段階での検証が非常に重要である。

ASIC の開発には通常 HDL が用いられ、HDL シミュレータにより動作の確認を行なう。ところが、PCI カード上に実装される ASIC などでは、ASIC 単体のシミュレーションだけでなく、PCI バスを介して接続されるホストプロセッサの動作も含めたシミュレーションが必要である。

ホストプロセッサの動作を含めて HDL シミュレーションを行なえば、もっとも正確なシミュレーションができるが、シミュレーションする回路の規模が非常に大きく、現実的な時間でシミュレーションを行なうことが困難であるし、通常ホストプロセッサの動作の HDL 記述を入手することは困難である。

一方、シミュレーション対象となる ASIC へのアクセスのリストという形でホストプロセッサの動作を静的に切り出す手法では、協調動作によって変化する動的な状態のシミュレーションができない。

そこで我々は、 PCI を通じて使用するネットワークインタフェース用 ASIC Martini[1)~4)] の開発にあたり、Verilog HDL[5)] のシミュレータと C 言語で記述したホストプログラムを協調実行させるシミュレーション環境を構築した。

Verilog HDL 側では PLI により UNIX の通信機能を用い、ホストプログラムにはいくつかのライブラリや C++ によるクラスを用意して Verilog シミュレータと通信することができるようにした。

このような環境を用いると、ソフトウェア開発者もシミュレーションの段階でプログラミングすることができ、ソフトウェアから見た問題点を容易に洗い出すことが可能になった。

† 新情報処理開発機構
Real World Computing Partnership
†† 慶應義塾大学
Keio University
††† NEC 情報システムズ
NEC Informatec Systems

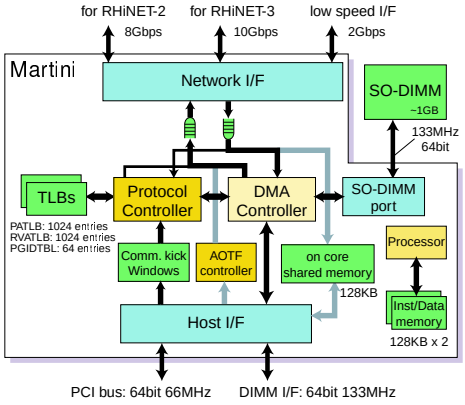

図 1　Martini ブロック図

2. ネットワークインタフェースコントローラ Martini

Martini は RHiNET[2),6)] と呼ばれるネットワークシステムで使われるネットワークインタフェースの核となるコントローラチップである。RHiNET ではユーザプロセスが直接通信を起動するユーザプロセス通信をサポートする。また、ネットワークインタフェースが直接ユーザプロセスのメモリ間でデータをコピーするゼロホストコピー通信を基本とする。

図 **1** に Martini のブロック図を示す。

Martini は 2 種類のホストインタフェースと 3 種類のネットワークをサポートする。基板に実装される際はそれらのうちから 1 つずつを選んで使用する。

PCI バスに接続されるネットワークインタフェースカード RHiNET/NI で使う場合、Martini は PCI I/F をホストインタフェースとして使用する。一方、DIMM スロットに挿入するネットワークインタフェースカード DIMMnet-1[7)~9)] 時には DIMM ホスト I/F をホストインタフェースとして使用する。ネットワーク側は、使用するネットワークに応じて 3 種類から選択する。

通信はホストプロセッサ上で動作するユーザプロセスから起動する。ユーザプロセスは Martini 内に存在する window と呼ぶ領域を通じて Martini に指示を出す。Martini は例えば、PUSH (リモートメモリライト) の指示を受け付けると、TLB を使用してアドレス変換を行い、ホスト I/F を通してホストメモリにアクセスし、データを取り出す。同時にネットワークに対してパケットヘッダを送り出し、その後ボディとしてデータを送信する。パケット受信時はヘッダに書かれている内容に従い、TLB を引き、ボディにあるデータをホスト I/F を通じてホストメモリに格納する。

このほかにも、Martini では少量のデータを高速に転送するための AOTF や BOTF 機構をサポートする。

また、ハードワイヤードでは実装していない機能の実現をサポートするためのオンチッププロセッサを持つ。

3. 検証環境

Martini などの ASIC の HDL 記述とホストプログラムの協調シミュレーションを行なう場合、最も問題となるのは、ASIC を記述している言語と、ホストプログラムを記述している言語が異なることである。

そこで、Verilog シミュレータとホストプログラムのそれぞれにライブラリを組み込むことで、UNIX の通信機能を用いた協調実行を可能にする環境を構築した。

この環境でのホストプログラムと Verilog 側の関係を図 **2** に示す。ホストプログラム側と Verilog 側は最下層のライブラリ間に張られたパイプにより接続される。

ホスト上で実行されるユーザプログラムのコード中では、実機上で動作するプログラム同様に RHiNET の API を用いて通信を記述する。通信に関与するメモリ領域 (Martini がマップされた領域を含む) への書き込みは、オペレータの多重定義 (overloading) を用いることにより、ユーザプログラム中では実機上で操作するプログラム同様に単純な代入により記述できる。

API および演算子定義のルーチン内で、必要に応じてアドレス変換を行ない、メモリや Martini をアクセスするトランザクションを Verilog 側と通信して処理する。

Verilog 側は PLI ルーチン群からなるライブラリ “host I/F” を使用して要求を受け取る。この要求は Verilog コードで記述されている controller が処理し、また、必要に応じて PCI のシミュレーションモデルである Synopsys 社の SmartModel PCI や Martini にアクセスする。

4. PLI による Verilog の拡張

Verilog HDL は PLI (Programming Language Interface) と呼ばれるインタフェースを提供しており、この PLI を使用することで Verilog 自身を機能拡張できる。PLI ルーチン自体は C 言語で記述するが、このルーチンは Verilog から呼び出される。そのため、シミュレーション自体は Verilog 側が主導権を持つことになる。

PLI では、Verilog コードが直接使用するシステムタスクや関数の他、特定の信号が変化した場合に Verilog シミュレータから呼び出されるコールバック関数や、初期化や後処理のための関数を定義することができる。

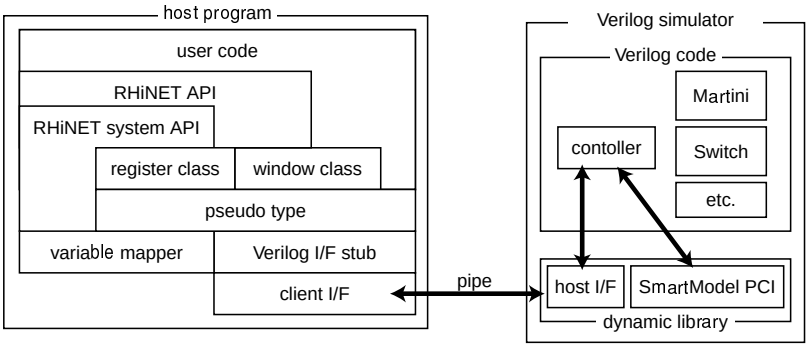

図 **2** シミュレーション環境階層図

PLI には次のような使用例が想定されている[5]。

- Verilog コードのデータ構造を動的にスキャンすることで遅延計算を行い、また、各インスタンスの遅延を変更する C 言語のコード
- ファイルからテストベクタやそのほかのファイルを読み込み Verilog コードに渡す C 言語のコード
- Verilog コードのための波形表示やデバッガの提供
- コンパイル済みの Verilog コードを解析して逆コンパイルするコード
- C 言語で書かれたシミュレーションモデル
- シミュレーションと協調動作するハードウェアとのインタフェース

PLI には世代によりいくつかの別のインタフェースが存在する。

- 第一世代は Task/function (TF) ルーチンとよばれ、`tf_` をプレフィックスとして持つ関数群である。TF ルーチンにはユーザ定義のタスクや関数の引数の解析や出力関数などがある。また、現在でもユーティリティ関数として利用される。
- 第二世代は Access (ACC) ルーチンで、`acc_` をプレフィックスとして持つ関数群である。ACC ルーチンは Verilog HDL の構造に対して直接アクセスする機能を提供する。また、遅延の値や論理信号の値などを変更することもできる。
- 第三世代は Verilog Procesdual Interface (VPI) ルーチンとよばれ、`vpi_` をプレフィックスとして持つ関数群である。VPI ルーチンは Verilog の構造のほか、動作記述に関するオブジェクトにもアクセスできるルーチンを提供する。VPI は TF, ACC に対するスーパーセットとなっている。

シミュレーションは PLI 機能を用いて拡張した Verilog シミュレータのプロセス (以下、Verilog プロセス) と、C++言語で記述されたホストプログラムおよび通信ライブラリからなるプロセス (以下、ホストプロセス) が通信しながら実行される。

Verilog プロセス内にある PLI ルーチンでは Verilog の時刻管理機構を使用できるため、信号に対して遅延付きで代入を行ったり、ある信号が変化した場合に処理をするルーチンを実装することができる。

しかし、ホストプロセス中では直接 Verilog の時刻管理機構を利用できないため、別の方法で時刻を管理する必要がある。

4.1 時 刻 管 理

実機上での実行を想定したホストプログラムには単に計算内容のみが記述されている。しかし、シミュレーションでは計算に要する時間をプログラム中に明示的に関数を用いて記述する。

本シミュレーション環境では、Verilog プロセス側では図 **3** 示す記述が実行の中心となる。Verilog プロセスとホストプロセスとはパイプを介して通信する。実行開始時には Verilog プロセス側からホストプロセスが起動される (図 3(1))。ホストプロセス側では、Martini との通信はメモリ領域へのアクセス時に多重定義された演算子により検出され、パイプを通じて Verilog プロセスに渡される。これを通信イベントと呼ぶ。また、プログラム中で明示的に記述された計算時間も、計算時間を記述した関数が実行される際にパイプを介して Verilog プロセスに渡される。これを遅延イベントと呼ぶ。

Verilog プロセスでは、図 3 のメインループ中で `$get_host_request()` (図 3(2)) によりパイプを通じてホストプロセスから通信イベントまたは遅延イベントを受けとり、それぞれ以下の処理を行なう。

通信イベント受領時: ホスト側からの書き込みは、相手がホストメモリか Martini かによりイベントの種類が異なる。これは、ホストメモリへのアクセスは直接メモリモデルのインスタンスにアクセスで実現されるのに対して、Martini へは PCI トランザクションを起動してアクセスする必要があるためである。しかし、どの場合でも SmartModel PCI が提供している API を使用して、メモリもしくは PCI バスにアクセスする (図 3(3))。

ただし、ホスト側が読み出しのイベントを通知した場合、読み出した値を`$put_host_result()` によりホスト側に通知する (図 3(4))。

いずれも処理が終ると、その時点での仮想時刻で

```
$init_host_program_cnf(...);          (1)

while( 1 ) begin
  $get_host_request(..., cmd, ...); (2)

  case( cmd )
    `MARTINI_HOST_PCI_WRITE: begin
      pcimaster_write_cycle(...);     (3)
    end
    `MARTINI_HOST_PCI_READ: begin
      pcimaster_read_cycle(...);      (3)
      pcimaster_read_rslt(...);       (3)
      $put_host_result(...);          (4)
    end
    `MARTINI_HOST_MEM_WRITE: begin
      pcislave_set_addr(...);         (3)
    end
    `MARTINI_HOST_PCI_READ: begin
      pcimaster_addr_req(...);        (3)
      pcimaster_read_rslt(...);       (3)
      $put_host_result(...);          (4)
    end
    `MARTINI_HOST_WAIT:
      repeat( arg32 )                 (5)
        @(posedge clk);
    ...
  endcase
end
```

図 3　イベント処理ループ

メインループの先頭に戻る。

遅延イベント受領時: 遅延イベントに記述された時間だけ経過した仮想時刻にメインループの先頭に戻る (図 3(5))。

実際の動作例を図 4 に示す。

5. 実　　装

本節ではシミュレーション環境の実装について、ホストプロセス側と Verilog プロセス側に分けて説明する。

5.1 ホストプロセス

ホストプロセスは、ソフトウェア開発者が記述するユーザコードとそれをサポートするライブラリ群から構成される。

ユーザコード: ソフトウェア開発者が直接コーディングする部分である。シミュレーションのためのコードには、計算時間を記述する関数が含まれるが、この関数は実機で実行する際にはなにも行なわないので、シミュレーション環境と実機実行環境で同じコードを使用できる。シミュレーションでは、通信はオペレータの多重定義により検出され、適当な処理が行なわれる。

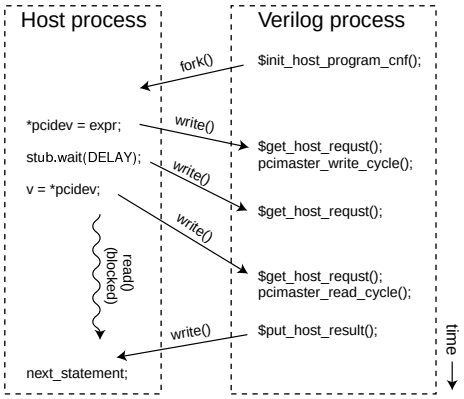

図 4　動作例

RHiNET API: RHiNET API ではユーザプロセスが通信に使用する API を提供する。シミュレーション環境でも実機実行環境でも同じコードを使用する。この RHiNET API 部でも、シミュレーション時には通信はオペレータの多重定義により検出される。

RHiNET system API: メモリ領域のピンダウンや Martini 内部のページテーブルへの登録など実機では OS やそれに準じるプログラムによって処理される API である。シミュレーションでは、必要な動作を記述した関数におきかえられる。

window class, register class: window, register ともに Martini がホスト側に提供する資源である。実機ではプロセス空間上にマップされたメモリとして見え、union へのポインタとして定義される。しかし、シミュレーション環境では、後述の pseudo type をメンバとして持たなくてはならない。コンストラクタを持つクラスを union のメンバにはできないという C++の制限があるため、特別なクラスを用意している。

pseudo type: ユーザコード中でも直接使用される通信領域で使われるクラスである。シミュレーションではこれらへのアクセスは多重定義されたオペレータより検出され、下層のライブラリを通じて Verilog プロセスと通信する。

variable mapper: プログラム中のデータの仮想アドレスと Verilog プロセスにあるメモリデバイスのアドレス (物理アドレス) の変換テーブルを管理する。ページ単位で管理しているため、実機と同様、仮想アドレスで同一ページ上にある変数は物理アドレスでもやはり同一ページとなる。

Verilog I/F stub: ユーザコードでのアクセスは変数単位で行われる。そのため、IA32 であれば最大 4byte 単位のアクセスとなる。しかし、実機で

はチップセットがいくつかのアクセスをまとめることがあり、PCI トランザクションとしてはバーストアクセスとなることが多い。
そこで、この階層で連続アドレスへのライト要求をまとめる。実際のライト要求を下層に伝えるタイミングは、連続しないアドレスへのライト要求が発生した場合とライト以外の要求、すなわちリードやシミュレーション時刻の更新要求などが発生した場合である。

client I/F: Verilog プロセス側と繋がるパイプのホストプロセス側の窓口を受け持つ。

5.2 Verilog プロセス

Verilog プロセスは Verilog HDL で記述されたコード群と、ダイナミックライブラリに格納されている PLI ルーチンの 2 つに分けられる。

Verilog HDL で記述されたコードには検証のターゲットである Martini そのものの他、シミュレーション全体を制御する制御部が含まれる。

Martini: 開発した ASIC そのものの Verilog 記述である。

Switch, etc.: 別途開発された Martini に接続されるスイッチやネットワークのモデルなどの記述である。これらに Martini を接続することで複数のノードを使用した場合のシミュレーションを可能にしている。

制御部: シミュレーション全体を制御しているコード。ライブラリ中の host I/F 部からホストプロセスからの要求を受け取り、それを Martini や SmartModel PCI に伝達する。また、シミュレーションに先立ち、各部の初期化などもこのコードで行う。

一方、拡張機能には PCI バスやメモリデバイスなどのシミュレーションモデルを提供する SmartModel PCI と、ホストプロセスとの連携を図るための host I/F が含まれる。

SmartModel PCI: Synopsys 社の PCI バスのシミュレーションモデルライブラリである。

host I/F: ホストプロセスとのインタフェースを受け持つライブラリコードである。Verilog に対してシステムタスクの形でホストプロセスとのインタフェースを提供する。

5.2.1 Host I/F

Verilog プロセス側のホストプロセスとのインタフェースは、host I/F とよぶライブラリ群により提供される。Host I/F は Verilog の機能拡張のためのインタフェース PLI (Programming Language Interface) と VPI (Verilog Procedual Interface) を使って記述されている。

Host I/F は次の 5 つのシステムタスクを提供する。

$init_host_program: 引数で与えられたホストプロセスを起動する。ホストプロセスには Verilog コードから渡されるホストメモリの物理アドレスやサイズも引数として渡される。

$init_host_program_cnf: プログラム名や引数を外部のファイルから読み出す。

$get_host_request: ホストプロセスからのリクエストを受け取り、引数で与えられた配列に情報を返すタスクである。リクエストの実際の処理は Verilog コードで行う。

$put_host_result: ホストプロセスにリプライを返すためのシステムタスク。

$set_host_signum: 引数で指定された信号線を割込線として登録する。この信号線が指定された極性に変化した時、ホストプロセスにシグナルを送る。

Host I/F で提供するタスク自体は PCI に依存している部分はないため、DIMM がホスト I/F となる DIMMnet-1 のシミュレーション時でも host I/F を使用してホストプロセスとの協調実行が可能である。

6. ホストプログラムの例

本節では本環境で実行されるホストプログラムの記述例を使用して、コードを記述する上での注意点や特徴を述べる。

図 **5** に Martini を使った通信プログラムの例を示す。

図中、線で囲っている部分で、`win` というポインタを通していくつかのフィールドに値をセットしているが、これは Martini の window へのアクセスである。最後に `type` フィールドにタイプ (この例ではリモートメモリライト) をセットすることで Martini が処理を開始する☆。

図中、下線を施した部分が、シミュレーションのために追加・変更したコードである。

`sendbuf, recvbuf` は送受信データバッファである。通信時に Martini はこのバッファにアクセスする。

そのため、これらの領域へのアクセスは Verilog プロセスに通信イベントとして通知する必要がある。そこで基本型の int ではなく、pseudot type の Int クラスを使用する。

`simulate_init()` 関数は Verilog シミュレータ側から引数として渡される情報を元にホストプロセスの環境の初期化を行う。

`stub.wait(5)` は遅延イベントを発生するコードで、この例では 5 PCI clock 分の時間、シミュレーション時刻を進めること指示している。ここでは、受信バッファへのアクセスが Verilog シミュレーション上連続しないようにするために使用している。

このようにシミュレーション用のコードにするための作業では特に手間は必要とならない。また、実機の

☆ 実際のプログラムでは `rn_push()` というライブラリ中の関数を呼び出すが、ここでは説明のため内容を展開した。

```
#include "rhinet.h"
int main( int argc, char **argv )
{
  Int sendbuf[128];
  Int recvbuf[128];
  STATUS status;
  int i;

  simulate_init( &argc, argv );
  ...
  for( i = 0; i < 128; i++ )
    sendbuf[i] = i;

  for( i = 0; i < 128; i++ )
    recvbuf[i] = 0;

  status = 0;
  win->iva = (unsigned)sendbuf;
  ...
  win->size = 128*sizeof(Int);
  win->type = PrimPush;

  while( recvbuf[127] == 0 )
    stub.wait(5);

  return 0;
}
```

図 5　通信プログラムの例

ための環境でもヘッダファイルやクラス定義の工夫により、このユーザコードをそのまま実機で使用できる。

7. おわりに

本稿では、Verilog HDL で記述された ASIC の検証に、C++ で記述されたホストプログラムを利用する協調シミュレーション環境について述べた。

この環境を使用することで、ASIC の設計と並行してソフトウェアの開発も進めることができた。

そのため、Verilog HDL では記述が困難と思われるテストプログラムの使用が可能だったため、特定の条件下でデータを返すタイミングを誤るバグをシミュレーション時に発見することができた。

また、実チップが完成するよりも前に、実機でも使用できるテストプログラム群やメッセージ交換機能を実現するライブラリ群を実装できたため、実チップの完成と同時にプログラムによるテストを開始することができた。

現在、この環境で検証したネットワークインタフェース向けチップ Martini のボードへの実装が行われ、実チップでの検証が進んでいる。

参考文献

1) 山本淳二, 田邊昇, 西宏章, 土屋潤一郎, 渡邊幸之介, 今城英樹, 上嶋利明, 金野英俊, 寺川博昭, 慶光院利映, 工藤知宏, 天野英晴: 高速性と柔軟性を併せ持つネットワークインタフェース用チップ:Martini, デザインガイア 2000 (2000).
2) 山本淳二, 土屋潤一郎, 寺川博昭, 田邊昇, 渡邊幸之介, 今城英樹, 西宏章, 工藤知宏: RHiNET の特徴と Martini の設計／実装, *SWoPP 2001* (2001).
3) 宮脇達朗, 山本淳二, 工藤知宏: RHiNET のソフトウェアレイア, *SWoPP 2001* (2001).
4) 渡邊幸之介, 山本淳二, 土屋潤一郎, 田邊昇, 西宏章, 今城英樹, 寺川博昭, 上嶋利明: RHiNET/MEMOnet ネットワークインタフェース用コントローラチップ Martini の予備評価, *SWoPP 2001* (2001).
5) IEEE: *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Hanguate*, IEEE std 1364-1995 edition.
6) Kudoh, T., Nishimura, S., Yamamoto, J., Nishi, H., Tatebe, O. and Amano, H.: RHiNET: A network for high performance parallel processing using locally distributed computers, *IWIA 99* (1999).
7) 田邊昇, 山本淳二, 工藤知宏: メモリスロット搭載型ネットワークインタフェース DIMMnet-1 における細粒度通信機構, 情報処理学会研究報告, HOKKE 2000 (2000).
8) 田邊昇, 山本淳二, 今城英樹, 上嶋利明, 濱田芳博, 中條拓伯, 工藤知宏, 天野英晴: DIMM スロット搭載型ネットワークインタフェース DIMMnet-1 の試作, *SWoPP 2001* (2001).
9) 田邊昇, 山本淳二, 今城英樹, 上嶋利明, 濱田芳博, 中條拓伯, 工藤知宏, 天野英晴: A prototype of high bandwidth low latency network interface plugged into a DIMM slot, *SSGRR 2001* (2001).