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

2005－EVA－12（6）
2005／3／23

SCE-MI 仕様協調エミュレーション環境および エミュレーション用モジュール自動生成環境の開発/評価

大山将城[†]　近藤信行[†]
清水尚彦[†]　星野民夫[††]

SoC 開発における検証は大規模化、複雑化の一途を辿っている．これまでソフトウェアとハードウェアの検証を同時に行う協調検証環境は標準と呼べる環境が無く、移植性の点で問題があった．
accellera 社は協調エミュレーションにおけるテストベンチ開発の標準化/容易化を目的に SCE-MI(Standard Co-Emulation Modeling Interface) 仕様を提案した．本稿では SCE-MI 仕様を踏まえた独自協調エミュレーション環境の開発と評価について述べる．

Development and Evaluation of Co-Emulation Environment and Automatic Hardware Generation Environment featuring SCE-MI

Masashiro Ohyama ,[†] Nobuyuki Kondoh ,[†] Naohiko Shimizu [†]
and Tamio Hoshino [††]

Recentry, the scale of verification for SoC gets larger and larger. But there is a problem of portability of Co-Verification enviromnent.
accellera proposed a Co-Emulation environment named SCE-MI(Standard Co-Emulation Modeling Interface) for standardize of test bench development. In this report, we present detailed development and evaluation of our original Co-Emulation environment featuring SCE-MI.

1. はじめに

SoC 開発においてハードウェア-ソフトウェア協調検証が盛んに行われるようになった．accellera 社は協調エミュレーション用テストベンチの設計/使用を容易にすることを目的に SCE-MI(Standard Co-Emulation Modeling Interface) を提案した[1)]．

SCE-MI は DUT(Device/Design Under Test) とテストベンチ間のインタフェース仕様である (図 1)．図 1 の SW model は DUT 検証用のアプリケーション/テストベンチ,DUT はユーザが設計した CPU やメモリなどの IP コアである．DUT は Transactor と呼ばれるインタフェースモジュールを介してソフトウェアサイドと通信する．SCE-MI はテストベンチ側のインタフェースとして 6 つの API を提供し,DUT の入出力信号のタイミング等について隠蔽する．

† 東海大学
Tokai University
†† 株式会社 アプリスター
Applistar Corporation

SCE-MI を利用することで,

- SCE-MI 定義の 6 つの API 仕様を理解するのみでテストベンチ開発が可能である
- DUT をサブモジュールとするエミュレーション用ハードウェア論理は自動で生成される
- DUT のインタフェース仕様を SCE-MI が吸収するので DUT が違っても同じ方法で実装できる

といったメリットがある．

我々は SCE-MI 仕様に沿った協調エミュレーション環境とエミュレーション用モジュールの自動生成環境を独自の実装仕様で構築し，Linux と FPGA ボードを用いて実装/評価をした．

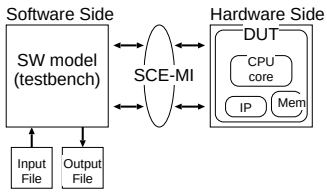

図 1　SCE-MI 概念図

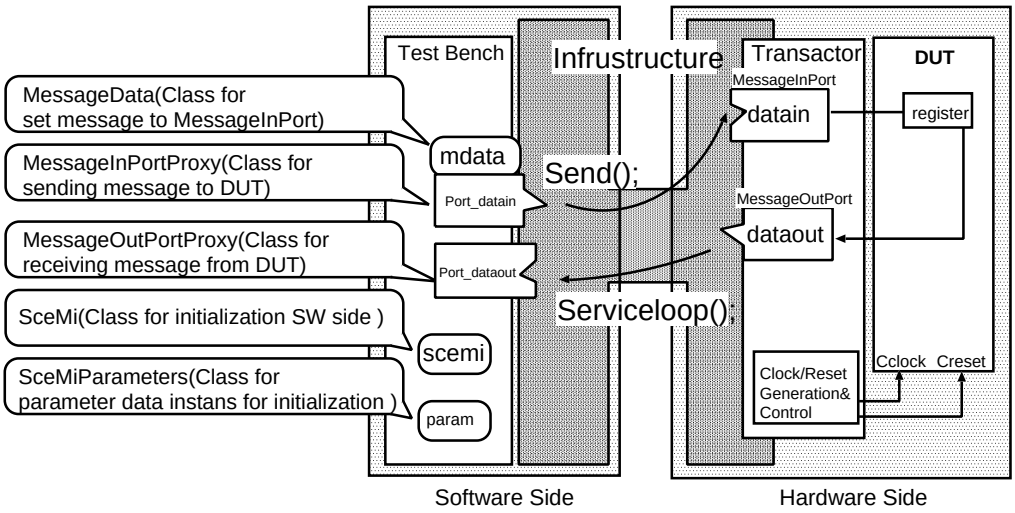

図 2 SCE-MI の実装例

表 1 ソフトウェアサイドのクラス仕様概要

クラス名	機能概要
SceMiEC	エラーハンドリング用クラス．このクラスはエラー箇所，エラー内容などの情報を保持するための構造体を持つ．
SceMi	SCE-MI ソフトウェアサイドの主要な処理を行うクラス．SCE-MI ソフトウェアサイドの初期化が呼び出されると，SceMiParameters クラスが生成するパラメータセットに基づいてハードウェアサイドの Message Port に対応する Message Port Proxy のインスタンスを生成する．
SceMiMessageInPortProxy	ハードウェアサイドの MessageInPort と接続する MessageInPortProxy を提供するクラス．
SceMiMessageOutPortProxy	ハードウェアサイドの MessageOutPort と接続する MessageOutPortProxy を提供するクラス．
SceMiParameters	Infrastructure Linker が出力したパラメータファイルを読み込み，SceMi クラスが初期化する際に使用するパラメータセットを生成する．
SceMiMessageData	送信 Message を保存するクラス．送信 Message をこのクラスのインスタンスに格納する．

2. SCE-MI 仕様概要

SCE-MI の仕様概要を説明する (実装例を図 2 に示す).

ソフトウェアサイドの Message Port Proxy とハードウェアサイドの Message Port は SCE-MI Infrastructure を介して 1 対 1 で接続する．各 Message Port Proxy と Message Port は通信方向が入力または出力に固定され、Message 単位でデータを送受信する．テストベンチから操作する場合、DUT への Message 送信では MessageData クラスのインスタンスにデータをセットし、MessageInPortProxy の Send メソッドをコールする．DUT から Message を受信する場合は,SceMi クラスのインスタンスの Seviceloop メソッドをコールする.Seviceloop メソッドは DUT に Transactor 内の ClockPort を介して 1 クロックの供給を行い、全ての MessageOurPort からの Message を読み出す. 図 2 の左側にはテストベンチ開発時に使用するクラスを, 図 3 にテストベンチの例を示す. 基本的には各クラスのインスタンスを生成するだけで DUT との通信ができる状態になる.

```
int main(void){
  try{
  SceMiEC *errhandler = new SceMiEC;

  SceMiParameters *scepam = new SceMiParameters("param.para",NULL);

  SceMi *scemi = SceMi::Init(versionchk,scepam,NULL);

  SceMiMessageInPortBinding inbinding = {NULL,isready,sceClose};

  SceMiMessageOutPortBinding outbinding = {NULL,receiveRdy,sceClose};

  SceMiMessageInPortProxy *Port_datain =
    scemi->BindMessageInPort ("TransactorC","datain", &inbinding, NULL);

  SceMiMessageOutPortProxy *Port_dataout =
    scemi->BindMessageOutPort("TransactorC","dataout",&outbinding,NULL);

  SceMiMessageData *mdata = new SceMiMessageData(*Port_datain, NULL);

  unsigned int mDat = 1;

  mdata->Set(0, mDat, NULL);

  Port_datain->Send(*mdata, NULL);

  scemi->Serviceloop();
  }
}
```

図 3 テストベンチの例

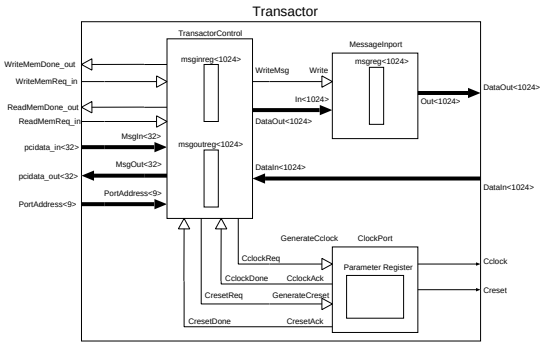

図 4 Transactor のブロック図

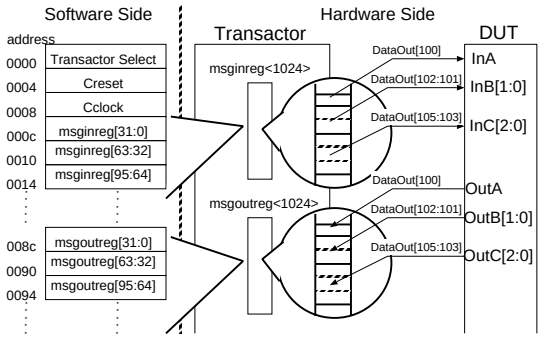

図 5 Transactor 内部のレジスタを PCI のメモリ空間,DUT の入出力ピンにマッピングした様子

3. 実 装 仕 様

3.1 SCE-MI ハードウェアサイド

SCE-MI ハードウェアサイドモジュール (Transactor) の実装仕様を以下のように決めた.

(1) SCE-MI Infrastructure には PCI インタフェースを用いる☆
(2) トランザクションレベルではなくクロックアキュレイトレベルの通信をサポートする
(3) DUT の外部インタフェースは入力または出力とし双方向の端子は使用できない
(4) 1Transactor は入出力端子を各々合計 1024 ビットまでサポートし、最大 4Transactor までとする

設計した Transactor のブロック図を図 4 に示す. Transactor は送信用に 1024 ビットのレジスタを 1 本, 受信用に 1024 ビットのレジスタを 1 本, クロックとリセットの供給用モジュールとコントローラにより構成する. 送受信用の内部レジスタを 32 ビット毎に PCI のメモリ空間にマップし, レジスタを Transactor の入出力端子に接続する. これにより PCI のアドレス空間にアクセスすることで対応したレジスタにアクセスをし、結果として DUT の入出力ピンのアクセスが可能となる (図 5). また、クロックとリセットの供給についてもメモリマップし、同様のアクセス方式で行うようにした. この方式で SCE-MI における Message の送受信の概念を実現した.

Transactor の生成と PCI インタフェース,DUT との接続,PCI のアドレス空間へのマッピングは後述する Infrastructure Linker が自動で行う. また, マッピングしたアドレスの情報についてはパラメータファイルと

☆ PCI インタフェースモジュール/ドライバは研究室所有 IP を使用[2)3)]

```
//インポートプロキシへのポインタを用意
//自分が必要なトランザクタのポートをバインドして, そのポインタをもらう
SceMiMessageInPortProxy *Port_datain =
  scemi->BindMessageInPort("TransactorC", "datain", &inbinding, NULL);

//メッセージデータインスタンス生成
SceMiMessageData *mdata =
  new SceMiMessageData(*Port_datain, NULL);

//メッセージデータセット
mdata->Set(0, mDat, NULL);

//データ送信
InP1->Send(*mdata, NULL);
```

図 6 Message 送信に関わるソースコード (抜粋)

```
//アウトポートプロキシへのポインタ用意
//自分が必要なトランザクタのポートをバインドして, そのポインタをもらう
SceMiMessageOutPortProxy *Port_dataout =
  scemi->BindMessageOutPort("TransactorC", "dataout", &outbinding, NULL);

//受信確認
scemi->Serviceloop();
```

図 7 Message 受信に関わるソースコード (抜粋)

いう形で Infrastructure Linker が出力し、これをソフトウェアサイドが初期化で使用し MessagePortProxy を生成する.

3.2 SCE-MI ソフトウェアサイド

SCE-MI 仕様に従って SCE-MI ソフトウェアサイド (API) を C++言語用いて設計した (表 1).

MessageInPort を介して DUT に Message を送信する際は SceMiMessageData クラスのインスタンスにデータをセットし, SceMiMessageInPortProxy クラスの Send メソッドを使用して行う (図 6).

MessageOutPort から Message を受信する際は SceMi クラスの Serviceloop メソッドを使用する (図 7). Serviceloop メソッドは以下の処理を行う.

(1) ClockPort より 1 クロックを供給する
(2) 全ての MessageOutPort に対して Message の読みだしを行う (読み出しは Port 毎にユーザ定義の関数を登録できる)

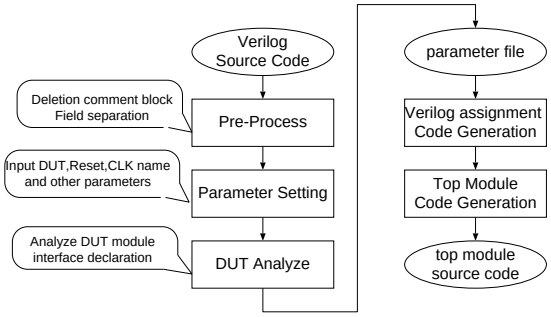

図 8 Infrastructure Linker の処理フロー

```
1
1
1
1
1,TransactorA,datain,32,0,0
2,TransactorA,dataout,32,0,1024,0
3,m_clock,33000,33000,50,50,0,5
4,TransactorA,m_clock
```

図 9 自動生成したパラメータファイル

3.3 Infrastructure Linker

Infrastructure Linker は Transactor とソフトウェアサイドの初期化用パラメータファイルを DUT のソースコードより生成するプログラムである. Infrastructure Linker により SCE-MI API を使用したテストベンチと DUT のソースコードのみでエミュレーションが可能となる.

Infrastructure Linker は DUT の Verilog ソースコードを読み込み, クラスファイルで使用するパラメータファイルと DUT,PCI インタフェースを合成可能な Verilog コードとして生成する. 前処理部, 解析部, コード生成部の 3 部で構成する (図 8).

前処理部ではコメント部の削除, フィールドの分割など解析部が処理しやすい形にソースコードを変形する. 解析部ではユーザが指定した情報 (top モジュール名,clock ポート名,reset ポート名) を元に DUT インタフェース記述の解析を行う. コード生成部では解析部の出力したパラメータファイルを元に Transactor モジュールの生成を行う.

パラメータファイルは図 10 のフォーマットである. 図 10 のように,

- 1 行目 Message InPort の数
- 2 行目 Message OutPort の数
- 3 行目 ClockPort の数
- 4 行目 ClockBind の数
- 5 行目以降 MessageInPort,MessageOutPort ClockPort,ClockBind の詳細な設定情報

のレコードで構成する.

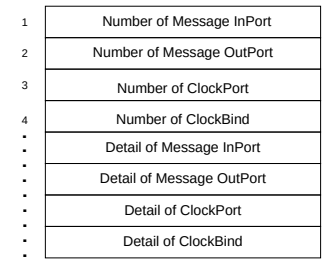

図 10 パラメータファイルのフォーマット

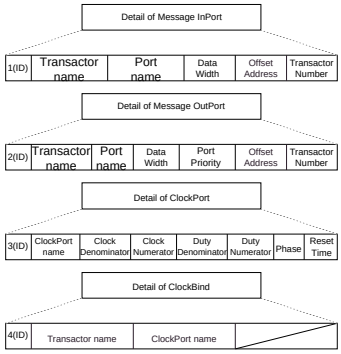

図 11 InPort,OutPort,ClockPort,ClockBind レコードのフォーマット

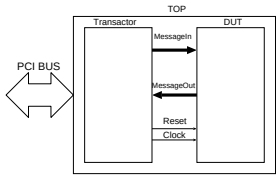

図 12 自動生成する top モジュールのインタフェース仕様

MessageInPort, MessageOutPort, ClockPort, ClockBind 各レコードの詳細な設定情報について図 11 に示す. 図 11 のうち、MessageIn/OutPort の名前とその Port が存在する Transactor の名前についてはテストベンチの開発で必要になる. PCI メモリ空間にマップされた各 Port のアドレスについてはソフトウェアサイド SceMi クラスが管理し、テストベンチの開発者が知る必要は無い.

パラメータファイルを元に SceMiParameters クラスは SceMi クラスの初期化用パラメータセットを生成する. また、パラメータファイルを元に SCE-MI ハードウェアサイドモジュールを生成する (図 12). 図 12 よりエミュレーション用のモジュールは PCI のインタフェース部のみを外部インタフェースとして持つ. このためユーザは使用する評価ボード, 論理合成ツール用に PCI インタフェース部のみのピンアサインファイルを用意すれば実装が可能となる.

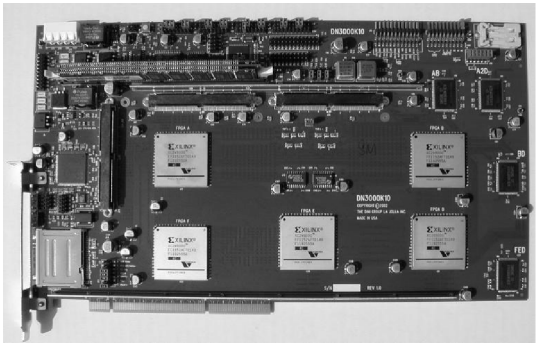

図 13 使用した FPGA ボード DN3000K10

表 2 合成結果

Device	Virtex2 XC2V4000
Module Name	slice
top	408

```
----------------------------------------
input data = 2
writting now!
receiveRdy called
ReceiveOutData = 2
----------------------------------------
```

図 14 エミュレーションテストのテキストログ (抜粋)

4. エミュレーションテスト

開発した SCE-MI によるエミュレーションテストを Red Hat Linux7.3 上で行った．ハードウェアサイド (PCI インタフェース，Transactor，DUT) は Xilinx ISE6.0 を使用し XC2V4000 に実装した (図 13).

実装結果を表 2 に示す．論理規模は 408[slice], 最高動作周波数は 95[MHz] であった.

エミュレーションテストではテストベンチから DUT ☆ の入力ピンに対して 1 サイクル毎にデータを書き込み，その時の出力ピンから Message の読み出しを行った．エミュレーションテストのテキストログを図 14 に示す．正常に Message の送受信が行えているのが確認できた.

4.1 エミュレーション性能評価

エミュレーション性能についての評価を行った．性能評価に用いたマシンのスペックを表 3 に示す.

評価は 1 つの MessageInPort と 1 つの MessageOutPort を持つ DUT に対し、Message の送信 (Send メソッド) と Serviceloop(クロックの供給と全 MessageOutPort の読み出し) を交互に 10 万回繰りかえした時の、実時間の計測で行った (図 15)．パラメータとして MessageInPort,MessageOutPort のデータ幅を 16,32,64,128,256,512,768,1024[bit] と変化させた場合の結果を表 4 に示す. 表 4 において時間とは Send と Serviceloop を 1 回ずつ実行した時の平均時間である．表 4 よりデータ幅が 16[bit] 時に 131[KHz]、1024[bit] 時に 29.6[KHz] でエミュレーションが行えることが分かる (グラフを図 16 に示す).

MessagePort のデータ幅を変えた場合、PCI ドライバに渡すデータバファのサイズのみが変化しその他

☆ ソフトウェアサイドから正常な Message の書き込みが行えているかを確認するために書き込まれた Message をそのまま返すモジュールを DUT とした.

表 3 評価用マシンのスペック

CPU	Celeron 1.8[GHz]
Memory	PC2100 128[MB]
OS	RedHat Linux 7.3

```
for(int time = 0 ; time < 100000 ; time++) {
  //メッセージデータセット
  mdata->Set(0 , mDat , NULL);

  //データ送信
  InP1->Send(*mdata , NULL);

  //受信確認
  scemi->Serviceloop();

  //メッセージデータ更新
  mDat++;
}
```

図 15 時間計測用のソースコード (抜粋)

表 4 時間計測結果

データ幅 [bit]	時間 [us]	周波数 [KHz]	転送バンド幅 [MB/s]
16	7.76	131	0.258
32	7.99	125	0.501
64	8.27	121	0.967
128	9.90	101	1.62
256	13.32	75.1	2.40
512	20.08	49.8	3.19
768	26.96	37.1	3.56
1024	33.80	29.6	3.79

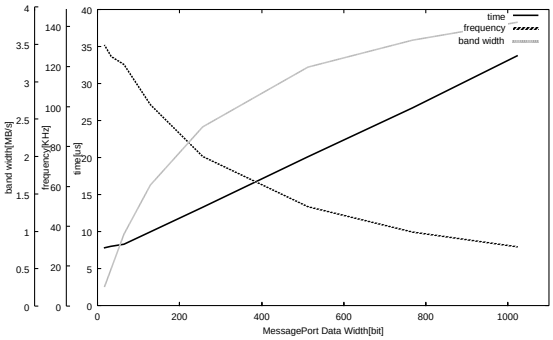

図 16 MessagePort のデータ幅の変化に対するエミュレーション実行時間, 周波数, 転送バンド幅の変化

表 5 ドライバ内の時間計測結果 (read)

データ幅 [bit]	時間 [us]	周波数 [KHz]	転送バンド幅 [MB/s]
16	0.783	1277	2.55
32	0.849	1178	4.71
64	1.38	725	5.80
128	2.77	361	5.78
256	5.32	188	6.02
512	10.6	94.3	6.04
768	15.9	62.8	6.04
1024	21.3	46.9	6.01

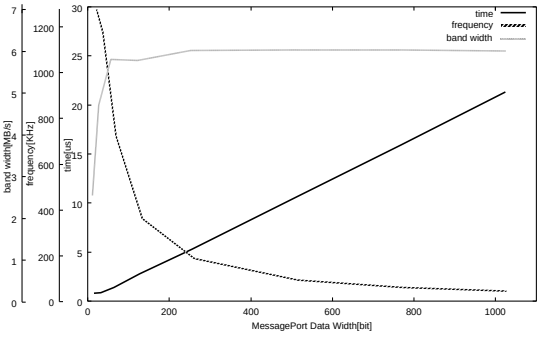

図 17 MessagePort のデータ幅の変化に対する read システムコール実行時間, 周波数, 転送バンド幅の変化

表 6 ドライバ内の時間計測結果 (write)

データ幅 [bit]	時間 [us]	周波数 [KHz]	転送バンド幅 [MB/s]
16	1.42	704	1.41
32	1.55	645	2.58
64	1.25	800	6.40
128	1.49	671	10.7
256	2.02	495	15.8
512	3.09	324	20.7
768	4.18	239	23.0
1024	5.25	190	24.4

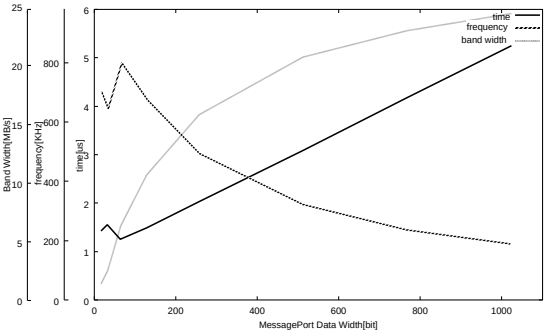

図 18 MessagePort のデータ幅の変化に対する write システムコール実行時間, 周波数, 転送バンド幅の変化

の処理には影響しない. このため、PCI ドライバ部分だけの実行時間の変化について計測を行った. 表 5, 6 に PCI ドライバ内の read,write それぞれの時間計測結果を示す ☆. 表 5 より PCI ドライバの read では 16[bit] 時に 1277[KHz],1024[bit] 時に 46.9[KHz] であった (グラフを図 17 に示す). 表 6 より PCI ドライバの write では 16[bit] 時に 704[KHz],1024[bit] 時に 190[KHz] であった (グラフを図 18 に示す).

この結果より,MessagePort のデータ幅が大きい場合、特に Message read が遅くなることが分かった. 現在の read 実装では unreachable な PCI 空間に 4Byte 単位で読み書きを行っており、ストアバッファが利用できる write に対し 4 倍の時間がかかったものと思われるが、詳細原因について現在調査中である.

5. ま と め

SCE-MI 仕様に沿ったハードウェア論理 (Transactor), クラスライブラリ, 実装用ハードウェアモジュールとソフトウェアサイド初期化用パラメータファイルを自動生成する Infrastructure Linker を開発した. そして, これらを使用し自動生成したモジュールとパラメータファイルによる FPGA 上でのエミュレーションを行い正常動作を確認した. また, エミュレーション性能の評価より現在のドライバ仕様では DUT の入出力ピン数が多い場合に性能が低下する結果となった.

今後はドライバ内での性能をさらに詳細に計測し性能低下の原因特定と改善、複数 MessagePort を持つ DUT によるエミュレーション動作の確認と評価を行う予定である.

参 考 文 献

1) accellera:"SCE-MI Reference Manual DRAFT",2003
2) 早坂, 志水, 横山, 孕石:"第 9 回 ASIC デザインコンテスト 規定課題 B PCI バスインタフェース" 第 22 回パルテノン研究会,2003
3) ALESSANDRO RUBINI "LINUX デバイスドライバ" O'REILLY ジャパン,1998

謝 辞

本研究において SCE-MI の実装評価は株式会社アプリスターにおいて、同社所有の FPGA 評価ボード DN3000K10 と Linux 搭載 PC により行った. 評価の協力に感謝する.

☆ 表 5, 6 はドライバ内での時間であり、実際にはシステムコール呼び出しによるオーバーヘッドが加わる. ドライバ内で何も実行せずシステムコールを呼ぶだけで消費されるクロックは約 1800[clk] であったので各 read,write に対し 1[us] のオーバーヘッドとなる.