情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2010-SLDM-144 No.58
Vol.2010-EMB-16 No.58
Vol.2010-MBL-53 No.58
Vol.2010-UBI-25 No.58
2010/3/28

MIPS アセンブリを中間表現とする高位合成システムの実装

入 谷 賢 孝[†1]　池 上 達 也[†2]　石浦 菜岐佐[†1]
神 原 弘 之[†3]　冨 山 宏 之[†4]

本稿では, C プログラム中の指定された関数を他の関数から呼び出し可能なハードウェアに合成するための一手法として, アセンブリからのハードウェア合成とソースコードレベルでのプログラム更新に基づく方法を提案する. C プログラムをコンパイルして得られるアセンブリ/機械語を入力として, CPU と同じ動作をするハードウェアを合成することにより, より広範な C プログラムをハードウェア化の対象とできる. ハードウェア化した関数を他の関数から呼び出すためには, そのインタフェースをとるために呼び出し元の関数を修正する必要が生じるが, 本手法では, この修正をアセンブリ/機械語レベルではなく, C 言語のソースコードレベルで行うことにより処理を単純化する. 本方式に基づき, MIPS 用 GCC を用いて得られるアセンブリを経由してハードウェアに合成した関数を, MIPS R3000 互換プロセッサ上の関数から呼び出せることを確認した.

Implementation of a High-Level Synthesis System which Uses MIPS Assembly Programs as Intermediate Representation

Yoshitaka Iritani,[†1] Tatsuya Ikegami,[†2] Nagisa Ishiura,[†1]
Hiroyuki Kanbara[†3] and Hiroyuki Tomiyama[†4]

This article presents a high-level synthesis flow, which uses assembly programs as intermediate representation. By synthesizing CPU compatible hardware from assembly/binary programs generated by compilers, wider class of C programs can be compiled into hardware. In order for the “hardware functions” to be called from the other functions, the callers must be updated so that they may interface with the callee hardware. In our method, this task is simplified by modifying caller code in source program level instead of in assembly or binary level. Through RTL simulation, it is verified that a hardware function, synthesized through assembly code generated by GCC for MIPS, are succesfully called from a main function running on an MIPS R3000 compatible CPU.

1. はじめに

プログラミング言語等による動作記述からレジスタ転送レベルの回路を合成する高位合成[1) は, 設計に要する期間やコストを削減する技術の一つとして実用化が進められている. ハードウェア設計用に拡張/制限した C 言語から品質の高いハードウェアを合成する技術の研究が進められる一方で, 最近ではハードウェアとソフトウェアからなるシステムの設計に高位合成を適用する研究が盛んに行われている[2)]. 特に, 既存のプログラムの一部を自動的にハードウェア化して性能や消費電力を改善する技術は, 高位合成の有望な応用の一つと考えられる.

このような応用に着目した場合には, いかに広範な C プログラムを合成の対象にできるかが, 高位合成に求められる新たな指標となる. ハードウェア化を意識せずに開発された C プログラムには, ポインタによる動的オブジェクトのアクセスや可変配列等, 高位合成では従来扱いが難しいとされる構文が多く含まれる. ソフトウェアを出発点とするハードウェア合成を実用化するためには, できる限り広いクラスの C プログラムをそのままハードウェアに合成できることが重要になる.

プログラムの動作を忠実にハードウェア化するための一つのアプローチとして, C プログラムではなく, アセンブリや機械語を入力とする高位合成手法が提案されている[4)–6)]. これは, CPU を制御するアセンブリ/機械語を入力として, そのコードを実行する CPU と同じ動作のハードウェアを合成するというものである. この方法では, C 言語からの合成に比べて得られる回路の品質が劣る可能性はあるが, C コンパイラの利用により, より広いクラスの C プログラムを容易にハードウェア化することが可能となる.

このような点で, アセンブリ/機械語からの高位合成は, ソフトウェアからのハードウェア合成という目的に適していると考えられるが, ソフトウェアとハードウェアのリンク処理に

†1 関西学院大学
Kwansei Gakuin University
†2 NTT データ アウラ
NTT DATA AURA
†3 京都高度技術研究所
ASTEM RI
†4 名古屋大学
Nagoya University

© 2010 Information Processing Society of Japan

Vol.2010-SLDM-144 No.58
Vol.2010-EMB-16 No.58
Vol.2010-MBL-53 No.58
Vol.2010-UBI-25 No.58
2010/3/28

課題が残る. 文献[6] の処理系では, リンク前のアセンブリから抽出した関数をハードウェアに合成し, ポーリングによってこれを他のソフトウェア関数から起動するが, ハードウェア化する関数中でのグローバル変数はシンボルで表現されているため, そのシンボル解決の処理が必要になる. Stitt らの Binary Synthesis[5] では, リンク済みの機械語を入力としてその一部をハードウェア化するが, ソフトウェア側においてリンカーのリロケーションと同様のアドレス修正が必要になる.

本稿では, このような課題を解決する一つのアプローチとして, プログラムのハードウェア化はアセンブリを入力として行い, ハードウェアとソフトウェアのインタフェースを実現するためのコードの更新は C ソースコードレベルで行うという手法を提案する. 本手法は, 与えられた C プログラムに対し関数単位で指定された部分をハードウェア化する. ハードウェア関数の起動のために必要となるコードの修正は, C プログラムに対して行う. 修正したプログラムをコンパイル・リンクし, 生成された機械語コードを逆アセンブルして得られるアセンブリから当該関数を抽出してハードウェアに合成する. 本方式に基づき高位合成システムを実装した結果, MIPS 用 GCC を用いて得られるアセンブリを経由してハードウェアに合成した関数を, MIPS R3000 互換プロセッサ上の関数から呼び出せることを確認できた.

2. アセンブリ/機械語プログラムからの高位合成

アセンブリ/機械語プログラムを入力とする高位合成は, より広いクラスの C プログラムが容易に合成できる, 現存する機械語コードをそのままハードウェア化できる等のメリットがあり, いくつかの研究が行われている.

Mittal ら[3] は, DSP のアセンブリを入力とし, そのコードを実行する DSP と等価なハードウェアを合成する手法を提案している. 尾形ら[4] は PIC プロセッサの実行コードからのハードウェア合成を提案している. これらはいずれもハードウェア単体の合成に主眼を置いてるため, ソフトウェアとの協調動作は考慮していない.

Stitt ら[5] は, 機械語プログラム中の関数またはより小さなセクションのプログラムをハードウェアに合成し, これを残りのソフトウェアから起動する手法 “Binary Synthesis” を提案している. ソフトウェアとハードウェアの制御の受け渡しは, グローバル変数のポーリングにより実現する. 処理の流れを図 1 に示す. 入力となるのはリンク済みの機械語プログラムであり, その中でハードウェア化したい部分を指定する. ハードウェア化の対象となるプログラムは CDFG (control dataflow graph) に変換し, ソフトウェアからの起動をポーリングにより待つ演算や, 処理の完了を通知するストア演算を追加した後, 従来と同様の手法でハードウェアに合成する. 一方, ソフトウェア側では, ハードウェア化するコードを除去し, 代わりにハードウェアの起動, ハードウェアの実行完了の待機, 返り値の受取りのための命令を挿入する. この書き換え処理自体は単純だが, リンク済みの機械語に対して命令の削除・挿入を行うため, コード全体に渡ってアドレスの修正が必要になる. この処理は “binary update” と呼ばれるが, リンカーが行うリロケーション処理の一部を含んでおり, プロセッサによっては非常に複雑になる. また, この処理は機械依存であるため, 他のアーキテクチャのプロセッサにこの手法を適用しようとすれば, 合成系のみならず, binary update の処理系も作成しなければならない.

文献[6] の高位合成手法では, リンク前のアセンブリを入力としているため, グローバル変数のシンボル解決が課題となる. ハードウェア化する関数がグローバル変数を参照している場合, リンク前にはアドレスがシンボルのままなので, このシンボルを呼び出し元のソフトウェア中のアドレスで置換する必要がある. また, この際にもリロケーションの処理が必要になることがある.

このように, アセンブリ/機械語プログラムからの高位合成においては, リンカーに関わる処理に課題が残る.

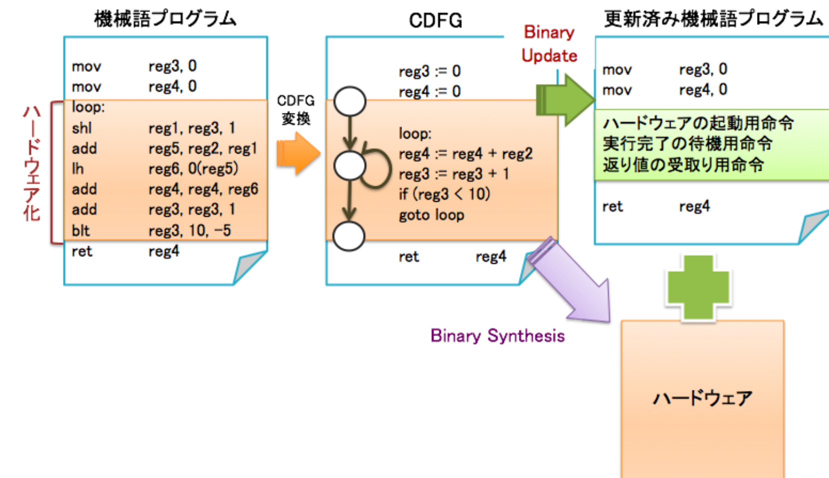


図 1　Binary Synthesis の処理の流れ

© 2010 Information Processing Society of Japan

Vol.2010-SLDM-144 No.58
Vol.2010-EMB-16 No.58
Vol.2010-MBL-53 No.58
Vol.2010-UBI-25 No.58
2010/3/28

3. ソースコード更新に基づく MIPS アセンブリからの高位合成

3.1 提案手法の概要

本稿では, アセンブリを入力とする高位合成と, ハードウェアの起動に必要なソフトウェアのソースコードレベルでの更新に基づいて, 与えられた C プログラム中の指定された関数をハードウェア化する手法を提案する. 本手法の狙いは, 元々の C プログラムが実行されるプロセッサのコンパイラとリンカを利用することにより, ハードウェアの合成や, ソフトウェアとハードウェアのリンク処理を単純化することにある.

本手法の処理の流れを図 2 に示す. 与えられた C プログラムの中で, ハードウェア化する関数が指定されているとする. まず, ハードウェア化される関数をソフトウェア関数から呼び出すために必要な書き換えを, C プログラムのレベルで行う. 更新は,

(1) ポーリング, およびデータの授受に必要なグローバル変数の宣言
(2) 被呼び出し側の起動待ち, データの授受, 完了通知のコードの追加
(3) 呼び出し側の起動, データの授受, 完了待ちを行うためのインタフェース関数の追加

である. 呼び出し側のソフトウェア関数そのものは一切変更しない. こうして更新した C プログラムをコンパイル・リンクして実行可能コードを得る. CPU では, このコードを実行する. ハードウェアの合成も, この実行可能コードから行う. このコードを逆アセンブルして得られるアセンブリコードから当該関数を切り出し, これを合成ツールの入力とすることによりハードウェアを得る.

3.2 MIPS アセンブリからの CDFG 生成

アセンブリコードからの合成の手法は, 基本的に文献[6] と同様である. まず, アセンブリコードを基本ブロックに分割し, 各基本ブロックをデータフローグラフ (DFG) に変換する. これは, 各命令を DFG の 1 つあるいは複数の演算節点に変換し, データ依存があればその間に依存枝を設定するという処理である. 図 3 は MIPS アセンブリの基本ブロックの DFG への変換例である. 例えば, `lw` (1 語のロード) 命令は, 実効アドレスの計算を行うための加算とその結果をアドレスとしてメモリを読む演算 (lod) に変換する. nop 命令 (`sll zero,zero,0x0`) は削除する. 分岐命令は, DFG 間の遷移に変換する.

レジスタへのアクセスは, 一旦抽象的な「値」へのアクセスに変換し, バインディング段階で, 関数ローカルなレジスタに再割り付けする. このため, 合成後のレジスタはアセンブリ中のレジスタとは必ずしも対応しない. また, 静的単一代入 (SSA) 変換と等価な処理を行うため, アセンブリでは同じレジスタに格納されている値が合成後には別のレジスタに格納されることもある.

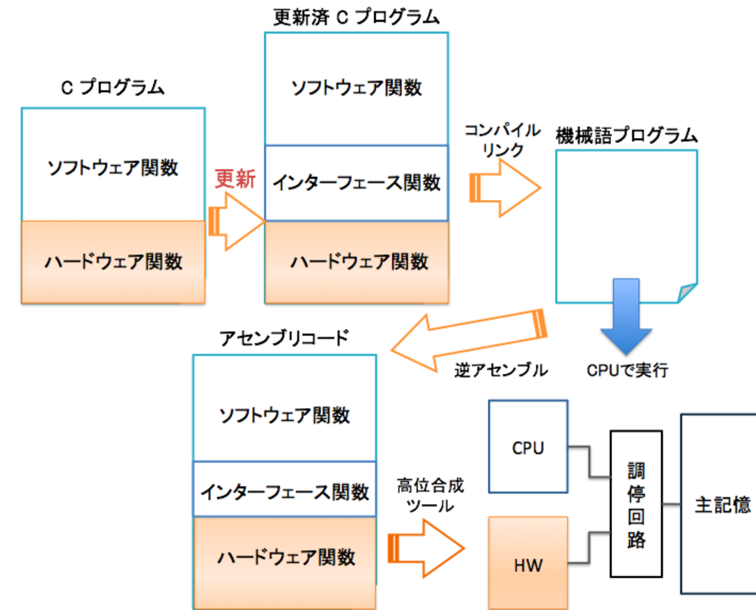


図 2 提案手法の処理の流れ

ソフトウェア関数とハードウェア関数間での値の授受に関して留意を要するレジスタは下記の通りである.

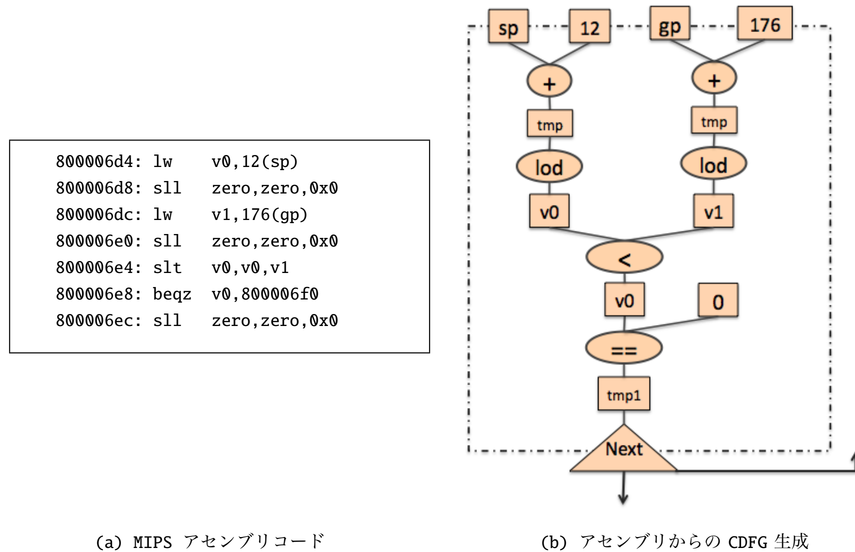
(1) 引数レジスタ (`$a0`〜`$a3`) と返り値レジスタ (`$v0`, `$v1`)
本手法では, 次節で述べる通り, 引数や返り値は全てグローバル変数を介して受け渡すようにソースコードを書き換えるため, これらのレジスタが (一時利用以外の用途で) ハードウェア側のアセンブリコード中に現れることはない.

(2) グローバルポインタ (`$gp`)
`$gp` はグローバル変数にアクセスするためのコードに出現する. `$gp` の値は, 起動後に更新されることはないため, アセンブリコード中のスタートアップルーチン `.crt0` から初期値を取得し, DFG 中の全ての `$gp` をその値で置き換える.

(3) フレームポインタ (`$fp`)

© 2010 Information Processing Society of Japan

Vol.2010-SLDM-144 No.58
Vol.2010-EMB-16 No.58
Vol.2010-MBL-53 No.58
Vol.2010-UBI-25 No.58
2010/3/28



```
800006d4: lw     v0,12(sp)
800006d8: sll    zero,zero,0x0
800006dc: lw     v1,176(gp)
800006e0: sll    zero,zero,0x0
800006e4: slt    v0,v0,v1
800006e8: beqz   v0,800006f0
800006ec: sll    zero,zero,0x0
```


(a) MIPS アセンブリコード　　(b) アセンブリからの CDFG 生成

図 3　MIPS アセンブリからの CDFG 生成

コンパイラに対する指示★1により消去する.

(4)　スタックポインタ ($sp)

スタックフレームの管理に必要になるため, 次節で述べる方法により, グローバル変数を介してソフトウェアからハードウェアに受け渡す.

3.3　ソースコード更新

ソフトウェアとして実行される関数からハードウェア化される関数を起動するために, 下記の 3 箇所の更新を, C プログラムのレベルで行う.

(1)　グローバル変数

次の 4 種類のグローバル変数を使用する. ただし, *callee* はハードウェア化する関数の名前である.

- _RUN_*callee*

  関数 *callee* の起動と終了をポーリングにより制御するための変数. この変数の値が 1 の場合は「実行中」を, 0 の場合は「非実行中」を表す.

- _ARG_*callee*_0, _ARG_*callee*_1, _ARG_*callee*_2, ⋯

  引数を格納する変数.

- _RET_*callee*

  返り値を格納する変数.

- _SP

  スタックポインタの値を格納する変数.

(2)　インターフェース関数 (ソフトウェア側)

関数 `callee` の呼び出しは変更せず, 関数 `callee` の内容を次のように更新する.

- 本体の文を全て削除する.
- 冒頭に次の文を追加する;

  `_SP = &第 1 引数;`

  これは, `$sp` の値をグローバル変数 _SP に格納してハードウェア側に渡すためのものである. 関数が呼び出された時点での `$sp` の値が, 第 1 引数のアドレスに等しいことを利用している.

- 引数の値を _ARG_*callee*_0, _ARG_*callee*_1, _ARG_*callee*_2, ⋯ にストアする.
- _RUN_*callee* をループでポーリングし, 値が `0` になる (ハードウェアの実行が完了する) のを待つ.
- 最後に次の文を追加し, 返り値を返す.

  `return _RET_`*callee*`;`

(3)　ハードウェア化する関数

ハードウェア関数 *callee* のコピー hw_*callee* という関数を作成し, これを合成対象とする. hw_*callee* は引数を持たず, 値も返さない.

- hw_*callee* の冒頭に次の処理を追加する.
  - _RUN_*callee* をループでポーリングし, 値が 1 になる (呼び出し側が起動する) のを待つ.
  - 引数の値を _ARG_*callee*_0, _ARG_*callee*_1, _ARG_*callee*_2, ⋯ よりロードする.
- hw_*callee* 中の `return` *exp*`;` 文をすべて下記で置換する.
  - `{_RET_`*callee* `=` *exp*`; _RUN_`*callee* `= 0;}`

図 4 (a) のプログラムに対し, 関数 `vprod` をハードウェア化した場合の更新結果を図 4 (b) に示す.

★1 gcc の -fomit-frame-pointer オプション等

© 2010 Information Processing Society of Japan

```
/* main はソフトウェアで実行 */
 1: int main (void) {
 2:     int n=4, a[4], b[4];
 3:     int s = vprod(n, a, b);
 4:     return 0;
 5: }

/* ハードウェア化する関数 */
 6: int vprod (int n, int *a, int *b) {
 7:
 8:     int i, s=0;
 9:     for (i=0; i<n; i++) {
10:         s += a[i] * b[i];
11:     }
12:     return s;
13: }
```

(a) 元のプログラム

```
/* main はソフトウェアで実行 */
 1: int main (void) {
 2:     int n=4, a[4], b[4];
 3:     int s = vprod(n, a, b);
 4:     return 0;
 5: }

/* (1) グローバル変数の宣言 */
 6: static int _RUN_vprod;
 7: static int _ARG_vprod_0;
 8: static int *_ARG_vprod_1;
 9: static int *_ARG_vprod_2;
10: static int _RET_vprod;
11: static void *_SP;

/* (2) インターフェース関数 (ソフトウェア側) */
12: int vprod(int n, int *a, int *b) {
13:     _SP = &n;
14:     _ARG_vprod_0 = n;
15:     _ARG_vprod_1 = a;
16:     _ARG_vprod_2 = b;
17:     _RUN_vprod = 1;
18:     while(_RUN_vprod) {;}
19:     return RET_vprod;
20: }

/* (3) ハードウェア化する関数 */
21: void hw_vprod(void) {
22:     while(!_RUN_vprod) {;}
23:     int i, n, *a, *b, s=0;
24:     n = _ARG_vprod_0;
25:     a = _ARG_vprod_1;
26:     b = _ARG_vprod_2;
27:     for (i=0; i<x; i++) {
28:         s += a[i] * b[i];
29:     }
30:     _RET_vprod = s; _RUN_vprod = 0;
31: }
```

(b) ソースコード更新済プログラム

図 4 ソースコード更新をした C プログラム

表 1 実験結果

	LUT 数	遅延 (ns)	サイクル数
CPU 単体	8241	26.925	221
CPU + HW	8301	26.228	199

なお, 呼び出され (ハードウェア) 側において _SP の値を $sp にロードするためのソースコードは現れていない. この処理は, アセンブリから CDFG を生成する際に直接対応する節点を挿入することにより実現する. _SP のアドレスは, 実行可能コードを逆アセンブルして得られるシンボル表より取得し, これに基づきハードウェアを合成する.

4. 高位合成システムの実装と実験

4.1 実　装

提案手法に基づく高位合成システムを, 文献[6] の処理系を拡張する形で実装した. 実装したシステムの構成を図 5 に示す.

入力となる C プログラムにソースコード更新を行い, コンパイル・リンクして得られる実行可能コードは, そのまま MIPS R3000 互換プロセッサ[7] で実行する. 指定された関数は, 実行可能コードから逆アセンブルを経て抽出する. これを CDFG に変換したものを高位合成系のバックエンドでハードウェア化し, CPU と結合して実行する. 以上の処理系は Perl 5 で実装しており, Linux, Mac OSX, Windows 上の Cygwin 等の環境で動作する. また, コンパイラには mips-elf-gcc (3.4.6) を用いた.

4.2 実　験

前節のシステムで合成したハードウェアと CPU を結合し, シミュレータ上で動作させた. また, Xilinx 社の ISE (Integrated Software Environment) 上で論理合成で回路規模や遅延の評価を行った. 2 つの 1 次元配列 (要素数 4) の内積を計算する関数をハードウェア化の対象とし, これを main から呼び出すというプログラムを用いた.

実験結果を表 1 に示す. CPU 単体で実行した場合に比べ, CPU と合成したハードウェアの組合せで実行した場合には, LUT 数 0.72 % の増加で, サイクル数を 9.95% 削減することができた.

5. む す び

アセンブリからのハードウェア合成とソースコードレベルでのプログラム更新に基づき, 与えられた C プログラム中の指定された関数をハードウェア化する手法を提案した. MIPS

© 2010 Information Processing Society of Japan

Vol.2010-SLDM-144 No.58
Vol.2010-EMB-16 No.58
Vol.2010-MBL-53 No.58
Vol.2010-UBI-25 No.58
2010/3/28

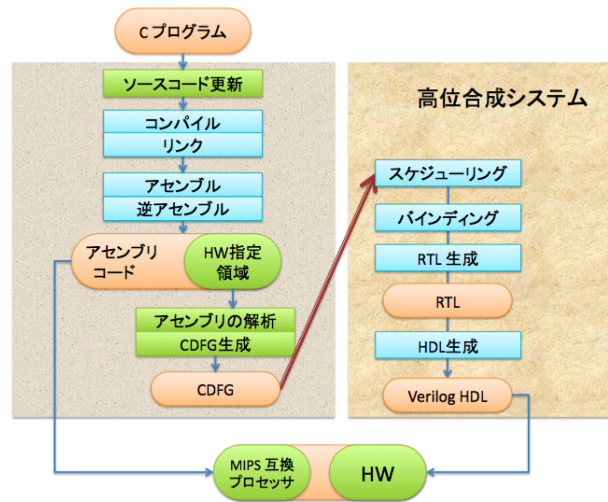


図 5　アセンブリを中間表現とする高位合成システム

互換プロセッサを用いた実験により, 動作を確認した.

$sp レジスタの値の受け渡しが機械依存部分として残っているため, これを改善する枠組みを考えることが重要な課題として考えられる. また, 実用的なプログラムに対する実験を通じて, 本手法が適用できる C プログラムの範囲/本手法の限界を明確にすることも今後の課題である.

謝　辞

本研究に際し, ハードウェア実装や動作検証などの面で多大なご助言とご助力を頂いた元立命館大学の中谷嵩之氏に感謝いたします. また, ACAP の開発に携わった関西学院大学の戸田勇希氏をはじめ, 石浦研究室の諸氏に感謝します.

参 考 文 献

1) D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S .Y.-L. Lin: *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).
2) 本田晋也, 冨山宏之, 高田広章: “システムレベル設計環境：SystemBuilder,” 信学論, vol.J88-D-I No.2, pp.163–174 (Feb. 2005).
3) G. Mittal, D. C. Zaretsky, and X. Tang: “Automatic translation of software binaries onto FPGAs, ” in *Proc. 41st DAC*, pp. 389–394 (June 2004).
4) 尾形秀範, 北川章夫: “アセンブリレベル合成法,” 信学技報, CAS2004-78, pp.35-40 (Jan. 2005).
5) G. Stitt and F. Vahid: “Binary synthesis,” *ACM Trans. on Design Automation of Electronic Systems*, vol. 12, no. 3, article 34 (Aug. 2007).
6) 池上達也: “MIPS アセンブリを中間表現とする高位合成,” 情処平成 20 年度関西支部大会, A-03 (Dec. 2008).
7) 神原弘之, 金城良太, 戸田勇希, 矢野正治, 小柳滋: “パイプラインプロセッサを理解するための教材 RUE-CHIP1,” 情処平成 21 年度学会関西支部大会, A-09 (Sept. 2009).

© 2010 Information Processing Society of Japan