社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

ソフトウェアと再リンク可能なハードウェアの高位合成

織野　真琴[†]　石浦菜岐佐[†]　冨山　宏之[††]　高島　史明[†]　神原　弘之[†††]

† 関西学院大学 理工学部 〒 669–1337 兵庫県三田市学園 2-1
†† 立命館大学 理工学部 〒 525–8577 滋賀県草津市野路東 1 丁目 1-1
††† 京都高度技術研究所 〒 600–8813 京都府京都市下京区中堂寺南町 134 番地

あらまし　本稿では, 高位合成を利用したハードウェア/ソフトウェア協調設計において, ソフトウェアと再リンク可能なハードウェアの合成手法を提案する. 近年, 機械語やアセンブリコードを中間表現として用いる高位合成において, ソフトウェアの一部を高位合成でハードウェア化することにより, ソフトウェアとハードウェアからなるシステムを合成する手法が研究されている. しかしこの手法では, ソフトウェアに変更があった場合, ハードウェアとソフトウェアが共有する変数のアドレスが変わるため, ハードウェアの再合成が必要となったり, 生成したハードウェアが他のソフトウェアとリンクすることができないという問題があった. そこで本稿では, ソフトウェアの変更に依存しないハードウェアを合成する手法を提案する. これはリンク前コードからの高位合成において, ハードウェアとソフトウェアが共有する変数のアドレス情報を表形式にしてソフトウェアから渡す枠組みを追加することにより実現する. このアドレス表の生成および読み出しはソースコード変換により追加できる. RTL シミュレータを用いて実験を行った結果, 合成されたハードウェアは, 初期化のために数十サイクルを要するが, ソフトウェア側でグローバル変数のアドレスが変わっても再合成することなく動作し, 別のソフトウェアともリンクして動作することが確認できた.
キーワード　高位合成, ハードウェア/ソフトウェア協調設計, 再リンク可能ハードウェア, ACAP

High-Level Synthesis of Hardware Relinkable to Software

Makoto ORINO[†], Nagisa ISHIURA[†], Hiroyuki TOMIYAMA[††], Fumiaki TAKASHIMA[†], and Hiroyuki KANBARA[†††]

† Kwansei Gakuin University, 2-1 Gakuen, Sanda, Hyogo, 669–1337, Japan
†† Ritsumeikan University, 1-1-1 Noji-Higashi Kusatsu, Shiga, 525–8577, Japan
††† ASTEM RI/KYOTO, 134 Chudoji Minamimachi, Shimogyo-ku, Kyoto-City, Kyoto, 600–8813, Japan

Abstract　This article presents a method of synthesizing *relinkable* hardware for hardware/software codesign utilizing high-level synthesis. Recent development of high-level synthesis through binary codes or assembly codes has enabled synthesis of functions in software programs into hardware modules callable from the software. In this scheme, however, hardware description is susceptible to the changes on the software, especially the changes on the addresses of the variables shared by software and hardware, so that the small changes on the software will lead to resynthesis of hardware or the hardware can not be linked with the other software programs. To solve this problem, we propose a method of synthesizing hardware which is less sensitive to the software changes and thus linkable to modified or different software programs without resynthesis. This is realized by synthesizing hardware from unlinked codes instead of linked codes, and a table of the addresses of the shared variables is passed from the software to the hardware. Since the task of creating and passing the address table is added by source code modification, little modification is needed on the synthesis system. We synthesized hardware modules according to the proposed method to confirm they are immune to the changes on the software part, and the hardware modules are linkable to different main programs, though extra cycles to pass the address tables are needed during initialization of the hardware modules.
Key words　High-Level Synthesis, hardware/software codesign, relinkable hardware, ACAP

1. はじめに

近年, 大規模化・複雑化するハードウェアの設計を効率化するための一技術として, C 言語などのプログラミング言語で書かれた動作記述からレジスタ転送レベルの回路記述を自動合成する高位合成技術 [1] の研究およびその実用化が進められている.

ハードウェア用に制限/拡張された C 言語による動作記述からハードウェアを合成する高位合成技術は成熟してきており, 優れたツールも多く実用に供されている [2]〜[4]. 一方で最近では, 元々ソフトウェアとして書かれた C プログラムを入力として, その一部をそのままハードウェア化する試みが行われている. 例えば, 文献 [5], [7] では, C プログラムの翻訳処理は通常のコンパイラに任せ, 機械語あるいはアセンブリ言語を入力とする高位合成のバックエンド処理系を実装することにより, 入力プログラムを実行するプロセッサと等価なハードウェアを合成している. ハードウェアを指向した動作記述からの合成に比べて, 生成される回路の品質は必ずしも高くないが, 既存のソフトウェア資産の性能や消費電力の改善が容易に図れるというメリットがある.

これらの手法では, 与えられたプログラム全体をコンパイル, アセンブル, リンクして得られるリンク済みの実行可能コードの中から, ハードウェア化する部分を抽出して合成する. ソフトウェアとハードウェアのデータの授受 (共有) は, グローバル変数を介して行われる. メモリアクセスに必要なアドレスはリンク時に決定されているため, ハードウェアのグローバル変数へのアクセスは容易に合成できる.

しかし一方で, リンク済みコードを利用する方法では, ソフトウェアに変更が生じ, それに伴って共有するグローバル変数のアドレスも変わると, 先に合成したハードウェアはそのまま使うことができなくなり, 再合成が必要になる. また, 合成されたハードウェアを別のソフトウェアとリンクして使用することもできない.

これに対し, 本稿では, リンク前のコードを入力として, ソフトウェアと再リンク可能なハードウェアを合成する手法を提案する. アドレスの解決は, ハードウェアが使用するグローバル変数のアドレス表を, ハードウェアの起動時にソフトウェアから渡すことにより実現する. これらの処理に必要な変換は, 全て C 言語のソースコードレベルで行えるため, コンパイルされたコードをそのまま高位合成系の入力とすることにより, 再リンク可能なハードウェアが生成できる.

高位合成ツール ACAP [7] にリンク前アセンブリコードを処理する機能を追加し, RTL シミュレータを用いて実験を行った結果, 合成されたハードウェアは, 初期化に数十サイクルを要するが, ソフトウェア側でグローバル変数のアドレスが変わっても再合成することなく動作し, 別のソフトウェアともリンクして動作することが確認できた.

2. 関数のハードウェア化とその呼び出し

2.1 アセンブリ/機械語からの高位合成

高位合成で処理可能な動作記述のクラスを拡大する試みの一つとして, アセンブリコードや機械語コードを入力とする手法が提案されている [5], [7]. これらは, アセンブリや機械語を制御データフローグラフ (CDFG) に変換し, それを従来の高位合成手法によりハードウェア化するというものである. コード中のロードストア命令はそのまま主記憶へのアクセスを行うように合成するので, 複雑なデータ型やポインタによるデータのアクセスも自然に合成できる. レジスタに含まれる番地への分岐やトラップなどの特殊な命令以外はほぼハードウェア化できるため, 高位合成の適用範囲を大幅に広げることができる上, コンパイラの最適化機能を利用できる等の利点もある. アセンブリや機械語コードではなく LLVM の中間表現を利用した処理系も発表されている [6].

このような高位合成系では, 与えられたプログラム全体をハードウェア化することも可能であるが, 合成されるハードウェアの性能コスト比の観点から, 文献 [5]〜[7] いずれの手法でも, プログラムのうちプロセッサでの処理がボトルネックとなる部分だけを選択してハードウェア化することが提案されている. ハードウェア化する単位は, 原理的には特に限定されないが [5], 関数を一つの単位とすることが多い [6], [7].

2.2 ハードウェア関数の呼び出し

ハードウェア化した関数をソフトウェアから呼び出す方法は種々考えられるが, 最も簡単な方法は [5]〜[7] でも記述されているポーリングによる方法である. これは, 元のプログラムをソースコードレベル [7], 機械語レベル [5], あるいはコンパイラや高位合成系の中間言語レベル [7] で変換することにより実現できる.

図 1 に変換の一例を示す. (a) が与えられたプログラムであり, この中の関数 `main` はソフトウェアとして CPU 上で実行し, そこから呼び出している関数 `f1` をハードウェア化したいとする.

これをソースコードレベルで変換したものが (b) である. 関数 `main` からは変換前と同様に関数 `f1` を呼び出している (10 行目) が, `f1` (14–21 行目) はハードウェアとのインタフェースを取るだけのもの (wrapper あるいは stub) に置き換えられている. 即ち, `f1` は, 引数を渡し (16–17 行目), ハードウェアの起動を指示する変数をセットし (18 行目), 関数の実行の終了をポーリングで待ち (19 行目), 返り値を読み出して返す (20 行目) だけであり, 実際の計算は関数 `_HW_f1` の方で行う. `_HW_f1` は, 自分が起動されるのを待って (27 行目), 引数をロードし (30–31 行目), 計算を行った後に, 返り値を書き込み (33 行目), 実行の終了を変数に書き込む (35 行目), という操作を繰り返し行う (25–36 行目).

(b) をコンパイルして得られるコード (シンボルに対してアドレスが割り当てられたリンク済みコード) から, `_HW_f1` に対応する部分を抽出し, 高位合成系に入力すれば, ソフトウェアから呼び出し可能なハードウェアを合成することができる.

しかし, ソフトウェア側に変更があった場合には, ハードウェアの再合成が必要になる場合がある. グローバル変数へのアクセスは, リンカが解決したアドレスを用いて行われるため, ソフトウェア側でグローバル変数の数や宣言順を変更した場合に

```
 1 int g;
 2
 3 int main(void) // SW として実行
 4 {
 5  ...
 6  int v = f1(x,y);
 7  ...
 8 }
 9
10 int f1(int a, int b) // HW 化対象
11 {
12  ...
13  return r;
14 }
```

(a) 与えられたプログラム

```
 1 int g;
 2 int _RUN_f1;
 3 int _ARG_f1_0;
 4 int _ARG_f1_1;
 5 int _RET_f1;
 6
 7 int main(void) // SW として実行
 8 {
 9  ...
10  int v = f1(x,y);
11  ...
12 }
13
14 int f1(int a, int b) // SW として実行
15 {
16  _ARG_f1_0 = a;
17  _ARG_f1_1 = b;
18  _RUN_f1 = 1;
19  while(_RUN_f1){;}
20  return _RET_f1;
21 }
22
23 void _HW_f1(void) // HW に合成
24 {
25  for(;;)
26  {
27   while(!_RUN_f1){;}
28   {
29     int a,b;
30     a = _ARG_f1_0;
31     b = _ARG_f1_1;
32     ...
33     _RET_f1 = r;
34   }
35   _RUN_f1 = 0;
36  }
37 }
```

(b) 変換後のプログラム

図 1　ハードウェア関数の呼び出しのための変換

は, そのアドレスを変更したハードウェアの合成が必要になる. また, 合成したハードウェアを別のソフトウェアとリンクして利用する場合にも, グローバル変数のアドレスが同じであることは保証されないので, 再合成を行う必要が生じる.

3.　再リンク可能なハードウェアの合成

3.1　概　　要

本手法では, ハードウェアとソフトウェアで共有するすべての変数のアドレスを, ソフトウェアからハードウェアに渡す枠組みを追加する. これは, 図 2 に示すように, 変数のアドレスを格納した表をソフトウェアが主記憶上に作成し, ハードウェアが起動時にこの表を読み出すことにより実現する. ただし, ソフトウェアとハードウェア間で制御とアドレス表を受け渡すための変数 (_RUN_f1) のアドレスのみはあらかじめ決定しているものとする.

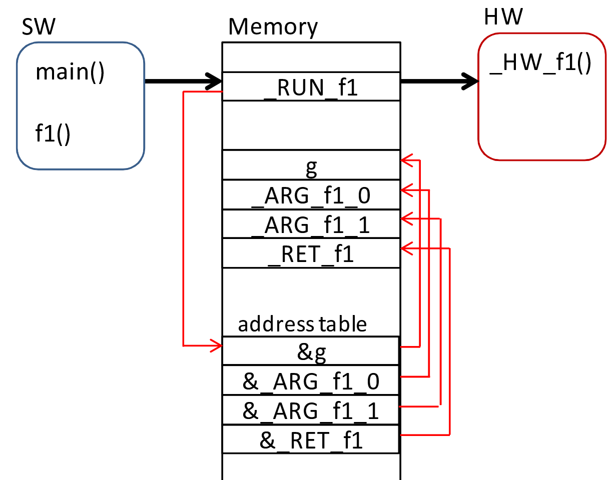


図 2　提 案 手 法

3.2　ソースコード変換による C プログラムの変換

アドレス表の授受の処理を追加する方法は種々考えられるが, 本稿ではソースコードレベルの変換による手法を示す.

変換の例を図 3 に示す. (a) の `main` が (c) の `f1` を呼ぶプログラムがあり, `f1` を高位合成によりハードウェア化するものとする.

まず (c) からハードウェアとのインタフェース部である (d) を生成する. 従来の処理に加え, 初期化関数 (7–18 行目) を追加している. 初期化関数 `_INI_f1` は, ソフトウェアとハードウェアで共有する変数のアドレスをアドレス表 `addr_table` に格納し (9–15 行目), その先頭番地を制御用の変数 `_RUN_f1` に渡して (16 行目) ハードウェアの初期化終了を待つ (17 行目). `_RUN_f1` は `void*` 型として宣言し, コンパイラの最適化によるポーリングの削除を防ぐため `volatile` 修飾する (1 行目).

次に (c) からハードウェアの本体の処理部 (e) を生成する. ここでは, 起動をポーリングで待って (6 行目), `_RUN_f1` の指す配列から変数のアドレスを読み出す (7–10 行目). すべての変数のアドレスを受け取ると, ソフトウェア側に制御を返し (11 行目), 通常の待機状態に戻る (16 行目). ハードウェアが呼び出された時には, 起動時に受け取ったアドレスから引数を読み込み (19–20 行目), 計算結果の書き込み (22 行目) を行う.

最後に (a) の `main` を (b) のように書き換え, 動作開始時に初期化関数を実行するようにする (13 行目).

3.3　複数ハードウェアにおける変換

本稿の手法は, 複数の関数をそれぞれ合成して複数のハードウェアに変換する場合にも適用可能である. 本稿では, 1 つのファイルにつき 1 つのアドレス表を生成し, すべてのアドレス表の初期化とハードウェアへの受け渡しをソフトウェアの起動時に行うことにより, 複数のハードウェアに対応する.

図 4 に例を示す. 図 4 では, (a) の `main` から (b) の `f1` と (c) の `f2` を呼び出し, さらに (b) の `f1` から (c) の `f2` を呼び出している. この (b) と (c) をハードウェア化するものとする.

```
1 int f1(int, int);
2 int g;
3
4 int main(void)
5 {
6  ...
7  f1(x,y);
8  ...
9 }
```

(a) main 関数

```
1 int f1(int, int);
2 int g;
3
4 int _main(void)
5 {
6  ...
7  f1(x,y);
8  ...
9 }
10
11 int main(void)
12 {
13  _INI_f1(void);
14  _main(void);
15  return 0;
16 }
```

(b) main 関数の変換

```
1 extern int g;
2
3 int f1(int a,int b)
4 {
5  ...
6  return r;
7 }
```

(c) ハードウェア化する関数

```
1 void* volatile _RUN_f1;
2 extern int g;
3 int _ARG_f1_0;
4 int _ARG_f1_1;
5 int _RET_f1;
6
7 void _INI_f1(void)
8 {
9  void* volatile addr_table[]=
10  {
11   &g,
12   &_ARG_f1_0,
13   &_ARG_f1_1,
14   &_RET_f1,
15  };
16  _RUN_f1=(void*)addr_table;
17  while(_RUN_f1){;}
18 }
19
20 int f1(int a,int b)
21 {
22  _ARG_f1_0 = a;
23  _ARG_f1_1 = b;
24  _RUN_f1 = (int *)1;
25
26  while(_RUN_f1){;}
27
28  return _RET_f1;
29 }
```

(d) ソフトウェア変換部分

```
1 extern void* volatile _RUN_f1;
2
3 void _HW_f1(void)
4 {
5  /* 初期化 */
6  while(!_RUN_f1){;}
7  int* _g_f1    = *((int**)_RUN_f1);
8  int* _arg_f1_0= *((int**)_RUN_f1+1);
9  int* _arg_f1_1= *((int**)_RUN_f1+2);
10  int* _ret_f1  = *((int**)_RUN_f1+3);
11  _RUN_f1 = (void*)0;
12
13  /* 通常の処理 */
14  for(;;)
15  {
16   while(!_RUN_f1){;}
17   {
18    int a,b;
19    a = *_arg_f1_0;
20    b = *_arg_f1_1;
21    ...
22    *_ret_f1 = r;
23   }
24   _RUN_f1 = (void*)0;
25  }
26 }
```

(e) ハードウェア変換部分

図 3　再リンクのためのソースコード変換

(b) のソースコードを変換した結果が (e) と (g) である. (e) は (a) とリンクして実行されるソフトウェアのインタフェース部である. ここでは `f1` にアクセスするためのグローバル変数だけでなく, `f2` のプロトタイプ宣言から, `f2` の引数と返り値を授受するためのグローバル変数を抽出して宣言し (5–7 行目), すべてのグローバル変数をアドレス表に追加する (13–17 行目). (a) と (c) ではグローバル変数 `A` を使用しているが, (b) では使用していないため, `f1` のアドレス表に `A` は入らない.

(b) のハードウェア部分である (g) では, `f2` の呼び出し部分をハードウェア呼び出しのためのポーリング処理に変更する (22–25 行目). 値の受け渡しには, 初期化で取得したアドレスを使用し (22, 25 行目), `f2` の制御変数である `_RUN_f2` を使用して処理を待つ (24 行目).

同様に, (c) を変換したソフトウェア部分が (f), ハードウェア部分が (h) である. (b) とはファイルに含まれるグローバル変数が異なるため, 異なるアドレス表が生成される.

`f2` は `f1` が呼び出された段階で既に初期化されていなければ, グローバル変数 `A` のアドレスを取得することができない. そのため, すべてのハードウェアの初期化を `main` 実行の前に行うよう, (a) を (d) のように変更する. ソフトウェアの本体の実行前に初期化を終えることにより, 各ハードウェアは呼び出し元に依存せず動作を行える. また, ファイルごとに初期化することで, ハードウェアを追加, 拡張した場合においても再合成は不要となる.

4. 実装と評価実験

高位合成ツール ACAP [7] にリンク前アセンブリコードを処理する機能を追加し, ソースコード変換を施した本手法のプログラムを入力として合成を行った. ソフトウェア部分は MIPS R3000 互換プロセッサ [8] で動作する. CPU 及びハードウェアは FPGA Xilinx Spartan3E 上に Xilinx ISE 12.3 で論理合成し, ModelSim Xilinx Edition III 6.4b を使用して動作の確認を行った.

再リンクの動作は, シミュレータを利用してメモリに書き込まれる値やハードウェアが出力する値から確認した. ソフトウェアから呼び出されたハードウェアは, 引数や返り値, グローバル変数に正しくアクセスすることができた. また, このハードウェアを別のソフトウェアから起動して実行したところ, ハードウェアは新しいアドレスからデータを読み込み, 処理を実行していることが確認できた.

次に, ハードウェアが 2 つあって, ハードウェア同士の呼び出しを含む場合について動作の確認を行った. ソフトウェアがハードウェア A を呼び出し, ハードウェア A がハードウェア B を呼び出す場合に, A から B の引数, 返り値, 起動変数へのアクセスおよび, B からソフトウェアで定義したグローバル変数へのアクセスが正しく行えていることが確認できた.

表 1 は, 2 つの 1 次元配列とその要素数を引数として受け取り, その内積を計算して返す関数をハードウェア化したときのサイクル数 (Cycle), 遅延 (Delay), フリップフロップ数 (FFs), LUT 数 (LUTs) を比較したものである. Cycle は総サイクル

```
1 int f1(int,int);
2 int f2(int);
3 int A;
4
5 int main(void)
6 {
7  ...
8  f1(x,y);
9  ...
10 f2(z);
11 ...
12 return 0;
13 }
```

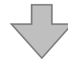
(a) ハードウェアを呼ぶ main 関数

```
1 int f2(int);
2
3 int f1(int a, int b)
4 {
5  ...
6  s = f2(t);
7  ...
8  return r;
9 }
```

(b) ハードウェア化する関数 f1

```
1 extern int A;
2 int f2(int a)
3 {
4  ...
5  return r;
6 }
```

(c) ハードウェア化する関数 f2



```
1 int f1(int,int);
2 int f2(int);
3 int A;
4
5 int _main(void)
6 {
7  ...
8  f1(x,y);
9  ...
10 f2(z);
11 ...
12 return 0;
13 }
14
15 int main(void)
16 {
17  _INI_f1();
18  _INI_f2();
19  _main();
20  return 0;
21 }
```

(d) 変換後 main 関数

```
1 void* volatile _RUN_f1;
2 int _ARG_f1_0;
3 int _ARG_f1_1;
4 int _RET_f1;
5 extern void* volatile _RUN_f2;
6 extern int _ARG_f2_0;
7 extern int _RET_f2;
8
9 void _INI_f1(void)
10 {
11  void* volatile addr_table[]=
12  {
13   &_ARG_f1_0,
14   &_ARG_f1_1,
15   &_RET_f1,
16   &_ARG_f2_0,
17   &_RET_f2
18  };
19  _RUN_f1=(void*)addr_table;
20  while(_RUN_f1){;}
21 }
22
23 int f1(int a,int b)
24 {/*図 3(d) と同様*/}
```

(e) f1 のソフトウェア部分

```
1 void* volatile _RUN_f2;
2 extern int A;
3 int _ARG_f2_0;
4 int _RET_f2;
5
6 void _INI_f2(void)
7 {
8  void* volatile addr_table[]=
9  {
10   &A,
11   &_ARG_f2_0,
12   &_RET_f2
13  };
14  _RUN_f2=(void*)addr_table;
15  while(_RUN_f2){;}
16 }
17
18 int f2(int a)
19 { … }
```

(f) f2 のソフトウェア部分

```
1 void* volatile _RUN_f1;
2 void* volatile _RUN_f2;
3
4 void _HW_f1(void)
5 {
6 while(!_RUN_f1){;}
7 int* _arg_f1_0= *((int**)_RUN_f1);
8 int* _arg_f1_1= *((int**)_RUN_f1+1);
9 int* _ret_f1  = *((int**)_RUN_f1+2);
10 int* _arg_f2_0= *((int**)_RUN_f1+3);
11 int* _ret_f2  = *((int**)_RUN_f1+4);
12 _RUN_f1 = (void*)0;
13
14 for(;;)
15 {
16  while(!_RUN_f1){;}
17  {
18   int a = *_arg_f1_0;
19   int b = *_arg_f1_1;
20   ...
21   /*f2 call*/
22   *_arg_f2_0 = t;
23   _RUN_f2 = (void*)1;
24   while(_RUN_f2){;}
25   s = *_ret_f2;
26   *_f1_ret = r;
27   }
28  _RUN_f1 = (void*)0;
29  }
30 }
```

(g) f1 のハードウェア部分

```
1 void* volatile _RUN_f2;
2
3 void _HW_f2(void)
4 {
5  while(!_RUN_f2){;}
6  int* _g_f2    = *((int**)_RUN_f2);
7  int* _arg_f2_0= *((int**)_RUN_f2+1);
8  int* _ret_f2  = *((int**)_RUN_f2+2);
9  _RUN_f2 = (void*)0;
10
11  for(;;)
12  {
13   while(!_RUN_f2){;}
14   {
15    int a=_arg_f2_0;
16    ...
17    *_ret_f1 = r;
18   }
19  _RUN_f2 = (void*)0;
20  }
21 }
```

(h) f2 のハードウェア部分

図 4　複数ハードウェアへの対応

数 (Total) の他に, ソフトウェア部分 (SW), ハードウェアの初期化 (HW(init)), ハードウェア本体 (HW(body)) に要したサイクル数を示している. 提案手法では, サイクル数が約 30 サイクル増加し, 回路面積も増加しているが, これはハードウェア初期化のオーバーヘッドによるものである.

表 1　内積の比較

	Cycle				Delay[ns]	FFs	LUTs
	Total	SW	HW (init)	HW (body)			
従来	557	121	-	436	13.1	547	1222
提案	588	159	15	414	12.9	681	1651

5. む す び

本稿では, ソースコード変換によりソフトウェアと再リンク可能なハードウェアの合成手法を提案した. RTL シミュレータを用いて実験を行った結果, 本手法で合成したハードウェアは, 再合成を必要とせず動作することが確認できた.

ソースコード変換は現在手動で行なっているが, 現在これを自動で行う処理系を実装中である. これと並行して, リンカの出力するシンボル表からアドレス表を抽出し, ハードウェアを部分的に再合成する枠組みについても検討を進めていく予定である.

謝　　辞

本研究を行う上で多くのご助言, ご協力頂いた元立命館大学の中谷嵩之氏, 京都大学の矢野正治氏に感謝します. また, 本研究に関してご協力, ご討議頂いた関西学院大学石浦研究室の諸氏に感謝します.

文　　献

[1] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu, and Steve Y-L Lin: *High-Level Synthesis: Introduction to Chip and System Design,* Kluwer Academic Publishers (1992).

[2] Takashi Kambe, Akihisa Yamada, Koichi Nishida, Kazuhisa Okada, Mitsuhisa Ohnishi, Andrew Kay, Paul Boca, Vince Zammit, and Toshio Nomura: "A C-Based Synthesis System, Bach, and its Application," in *Proc. ASP-DAC 2001,* pp. 151–155 (Jan. 2001).

[3] Handel-C (http://www.celoxica.com/).

[4] Kazutoshi Wakabayashi: "CyberWorkBench: Integrated Design Environment Based on C-based Behavior Synthesis and Verification," in *Proc. 2005 IEEE VLSI-TSA International Symposium,* pp. 173–176 (Apr. 2005).

[5] Greg Stitt and Frank Vahid: "Binary Synthesis," *ACM Trans. Design Automation of Electronic Systems,* vol. 12, no. 3, article 34 (Aug. 2007).

[6] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Anderson, Stephen Brown, and Tomasz Czajkowski: "LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems," in *Proc. 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays,* pp. 33–36 (Feb. 2011).

[7] 入谷賢孝, 池上達也, 石浦菜岐佐, 神原弘之, 冨山宏之: "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," 情報処理学会研究報告, 2010-SLDM-144-58 (Mar. 2010).

[8] 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," 情報処理学会関西支部大会, A-9 (Sept. 2009).