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.

アセンブリコードを中間表現とする高位合成における関数の併合

高島　史明† 　石浦菜岐佐† 　織野　真琴† 　冨山　宏之†† 　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 1-1-1
††† 京都高度技術研究所 〒600–8813 京都市下京区中堂寺南町 134

あらまし　本稿では, アセンブリコードを中間表現とする高位合成における関数の併合手法を提案する. C プログラム中の指定された関数をハードウェアに合成する手法はいくつかの高位合成システムで実装されているが, 関数間の呼出しが頻繁に行われる場合には, 制御や引数, 返り値の受渡しのオーバヘッドが無視できなくなる. これに対し本稿では, 複数関数を併合して単一ハードウェアに合成することにより, 関数呼出しのオーバヘッドを削減する手法を提案する. 本手法では, goto 変換に基づいて, ハードウェア化する関数がソフトウェアからもハードウェアからも呼び出せるように, 高位合成の入力コードを変換する. Goto 変換には, 引数や返り値の受け渡しの他, レジスタの退避/復帰の処理の追加も必要になるが, コンパイラはアセンブリコード中にこれらの処理を行うコードを最適化した形で生成するので, これを利用して処理系を実装する. 関数の併合により, 関数呼出しのオーバヘッドが抑制できるだけでなく, 関数間で資源共有が行えるため, ハードウェアのコストも削減できる. 本手法を高位合成システム ACAP に実装し, 実験的な回路で評価を行った結果, 実行サイクル数を約 15%, FPGA 上の LUT 数を約 60% 削減することができた.
キーワード　高位合成, 関数呼出し, 関数併合, Goto 変換, ACAP

Merge of Functions in High-Level Synthesis Using Assembly Codes as Intermediate Representation

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

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

Abstract　This article presents a method of merging functions during high-level synthesis whose inputs are assembly codes generated by a compiler front-end. While synthesizing functions in programs into separate hardware modules is one of the major approaches in hardware/software codesign, the overheads for passing arguments, return values, and control will not be negligible when inter-function calls are made frequently. Our method attempts to reduce the overhead by merging multiple functions into a single hardware module. The functions are merged by "goto conversion" so that hardware function modules are callable both from the other hardware function modules and the software program. We take advantage of the fact that all the necessary tasks associated with function calls are incorporated as well as optimized in assembly codes generated by a front-end compiler, so that the major part of the merger tasks is done in source code level transformation. By the merger, hardware cost, as well as the overhead for inter-function calls, is reduced due to resource sharing among the functions. The preliminary experiments based on ACAP high-level synthesizer showed that the number of the execution cycles is reduced by 15% while the FPGA LUT count is reduced by about 60%.
Key words　high-level synthesis, function call, merge of function, goto conversion, ACAP

1. はじめに

近年の半導体技術の発展に伴う集積回路の大規模化・複雑化により, その設計に要する期間とコストが増大している. 設計の効率化を図る一技術として, 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ハードウェア記述を生成する高位合成 [1] の研究と実用化が進められている.

合成アルゴリズムの改良など, 高位合成により生成される回路の品質を向上させる研究が多く行われる一方で, 高位合成の適用範囲を広げる試みも多く行われている. 特に [2]〜[4] 等では, C コンパイラが生成する機械語, アセンブリコード, あるいは中間表現を入力として高位合成を行うことにより, C 言語のほぼ全ての構文にまで合成対象を拡張している.

これらの文献では, この合成法を用いて, 元々ソフトウェアとして開発されたプログラムの中で処理時間を要する部分をハードウェア化して性能改善を図ることも提案されている. ハードウェア化された部分は, 主記憶を介してデータや制御を受け渡すことにより, ソフトウェアとして CPU で実行されるプログラムから起動される. ハードウェア化する単位は, 小さい場合はコード片 [3], 大きな場合は関数 [2], [4] となる.

同じ仕組みで複数の関数をそれぞれハードウェアに合成すれば, ソフトウェアのより多くの部分をハードウェア化できる. しかし, ハードウェア化された関数間の呼び出しが多い場合には, 呼び出しのオーバヘッドが無視できなくなる.

そこで本研究では, 複数の関数を併合して一つのハードウェアに合成することにより, このオーバヘッドを削減する方法を提案する. 本手法は関数呼び出しを分岐処理に変更する goto 変換に基づくが, ハードウェア化する関数がソフトウェアからもハードウェアからも呼び出せるようにこれを改良する. この変換処理は, 関数の引数や返り値の受け渡し, およびレジスタの退避/復帰の処理がコンパイラの生成するアセンブリコードに含まれ, さらにそれが最適化されていることを利用して行う.

関数の併合により, 関数呼出しのオーバヘッドが抑制できるだけでなく, 関数間で資源共有が行えるため, ハードウェアのコストも削減できる. 本手法を高位合成システム ACAP に実装し, 実験的な回路で評価を行った結果, 実行サイクル数を約 15%, FPGA 上の LUT 数を約 60% 削減することができた.

2. ソースコード中の関数の高位合成

2.1 アセンブリ/機械語コードを中間表現とする高位合成

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, あるいはコンパイラの中間表現を入力として高位合成系を構成する手法が提案されている [2]〜[4]. これらは, それぞれ機械語, アセンブリ, 中間表現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系のバックエンドで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演算として合成を行うのため, 複雑なデータ型やポインタによるオブジェクトへのアクセスも合成可能である. レジスタに含まれてい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他, コンパイラの最適化機能を利用できる等の利点もある.

2.2 関数呼出しの高位合成

関数呼び出しを高位合成する方法は種々提案されている.

最も簡単な方法はインライン展開により呼び出し先の関数を呼び出し元に展開する方法である. 特に関数を意識することなく合成が行え, 関数呼び出しのオーバヘッドもなくせるが, 同じ関数が繰り返し呼び出されている場合には, 回路規模が増大する可能性がある.

これに対し, 同じ処理を複製することなく関数呼び出しを消去する方法として goto 変換がある [5]. これは関数呼び出しと関数からの戻りを通常の分岐処理に変換するものである. 関数呼び出しは無条件分岐に置き換えればよい. 関数が 1 箇所からしか呼ばれていなければ, 関数の戻りも無条件分岐に置き換えることができる. 関数が複数箇所から呼ばれている場合には, 呼び出し時にどこから呼び出したかを識別する情報を特定の変数に格納しておけば, 戻りはこの情報を使った条件分岐に変換できる. インライン展開とは異なり goto 変換では, このような制御の受け渡し以外に, 引数や返り値の受け渡し, レジスタ等の資源を呼び出し元と呼び出し先で共有する場合には退避/復帰コードを追加する必要がある.

一方, 与えられた動作記述全体をハードウェア化するのではなく, プログラム中の指定された関数だけを独立したハードウェアに合成する手法も提案されている [2]〜[4]. ソフトウェアとして実行される他の部分からハードウェア化された関数を呼び出す方法は種々考えられるが, 文献 [2]〜[4] で提案されている手法では, 主記憶を介して引数と返り値を渡し, ポーリングや割り込みにより制御の受け渡しを行っている.

図 1 は, この方法による関数呼び出しの処理の追加を, C プログラムのレベルで表現した例である. (a) はソフトウェアで実行するメインのコードであり, (b) のコードをハードウェアに合成するとする. (c)(d) は (b) を入力として得られる変換の出力である. (c) はハードウェアを起動するためのソフトウェアコードであり, (a) とリンクして実行される. 関数は (b) と同名であるが, 内容は, 1) 引数をグローバル変数 (`_ARG_` で始まる変数) に格納し, 2) ハードウェアの起動を制御するグローバル変数 (`_RUN_` で始まる変数) をセットし, 3) ハードウェアの実行終了をポーリングで待ち, 4) 返り値をグローバル変数 (`_RET_` で始まる変数) より読み出す, という処理に置き換えられている. 一方 (d) は, ハードウェアに合成されるコードであり, 元々の処理に, (c) とは逆の起動/終了の制御や引数/ 返り値の受け渡しが追加されている.

これまでに提案されている手法では, ハードウェア化の最大単位は関数であり, 図 1 のように複数の関数を合成対象とする場合には, 各関数に対して一つのハードウェアが合成される. ハードウェア化された各関数は, ソフトウェアから呼び出せるとともに, 他のハードウェア関数からも呼び出せる. しかし, ハードウェア関数間の呼び出しが頻繁におこる場合には, 引数, 返り値, 制御の授受のオーバヘッドが無視できなくなる.

main.c

```
 1 int f1(int a1);
 2 int f2(int a2);
 3 int f3(int a3);
 4
 5 int main(void)
 6 {
 7  int a1 = 1;
 8  int r1 = f1(a1);
 9
10  int a2 = 5;
11  int r2 = f2(a2);
12
13  int a3 = 10;
14  int r3 = f3(a3);
15
16  return 0;
17 }
```

(a) SW として実行するコード

sub.c

```
 1 int f1(int a1)
 2 {
 3  int x1 = a1 + 1;
 4  int r1 = f3(x1);
 5  return r1;
 6 }
 7
 8 int f2(int a2)
 9 {
10  int x2 = a2 + 10;
11  int r2 = f3(x2);
12  return r2;
13 }
14
15 int f3(int a3)
16 {
17  int r3 = a3 * 10;
18  return r3;
19 }
```

(b) HW 化するコード

sw_sub.c

```
 1 int _RUN_f1;
 2 int _ARG_f1_0;
 3 int _RET_f1;
 4 int _RUN_f2;
 5 int _ARG_f2_0;
 6 int _RET_f2;
 7 int _RUN_f3;
 8 int _ARG_f3_0;
 9 int _RET_f3;
10
11 int f1(int a1)
12 {
13  _ARG_f1_0 = a1;
14  _RUN_f1 = 1;
15  while(_RUN_f1){;}
16  return _RET_f1;
17 }
18
19 int f2(int a2)
20 {
21  _ARG_f2_0 = a2;
22  _RUN_f2 = 1;
23  while(_RUN_f2){;}
24  return _RET_f2;
25 }
26
27 int f3(int a3)
28 {
29  _ARG_f3_0 = a3;
30  _RUN_f3 = 1;
31  while(_RUN_f3){;}
32  return _RET_f3;
33 }
```

(c) HW 起動用の SW コード

hw_sub.c

```
 1 extern int _RUN_f1;
 2 extern int _ARG_f1_0;
 3 extern int _RET_f1;
 4 extern int _RUN_f2;
 5 extern int _ARG_f2_0;
 6 extern int _RET_f2;
 7 extern int _RUN_f3;
 8 extern int _ARG_f3_0;
 9 extern int _RET_f3;
10
11 void _HW_func1(void)
12 {
13  for(;;){
14   while(!_RUN_f1){;}
15   int a1 = _ARG_f1_0;
16   int x1 = a1 +1;
17   _ARG_f3_0 = x1;
18   _RUN_f3 = 1;
19   while(_RUN_f3){;}
20   int r1 = _RET_f3;
21   _RET_f1 = r1;
22   _RUN_f1 = 0;
23  }
24 }
25
26 void _HW_func2(void)
27 {
28  for(;;){
29   while(!_RUN_f2){;}
30   int a2 = _ARG_f2_0;
31   int x2 = a2 + 10;
32   _ARG_f3_0 = x2;
33   _RUN_f3 = 1;
34   while(_RUN_f3){;}
35   int r2 = _RET_f3;
36   _RET_f2 = r2;
37   _RUN_f2 = 0;
38  }
39 }
40
41 void _HW_func3(void)
42 {
43  for(;;){
44   while(!_RUN_f3){;}
45   int a3 = _ARG_f3_0;
46   int r3 = a3 * 10;
47   _RET_f3 = r3;
48   _RUN_f3 = 0;
49  }
50 }
```

(d) HW に合成するコード

図 1　関数呼び出しに対応したソースコード変換

3. 高位合成における関数の併合

本稿では, ハードウェア化する単位を単一の関数から複数の関数に拡張することにより, ハードウェア化される関数間の呼び出しのオーバヘッドを削減する手法を提案する. これは, goto 変換により複数の関数を一つのハードウェアに併合することにより実現する. Goto 変換にはコンパイラが生成するアセンブリコードを利用する. また, 命令のアドレスとハードウェアの制御系の状態の対応をとることにより, 関数からの戻りのオーバヘッドの削減を図る.

3.1 Goto 変換による関数の併合

本手法では, ハードウェア関数から別のハードウェア関数の呼び出しを全て goto 変換で分岐命令に置き換える. この際に, ソフトウェアからもハードウェアからも呼ばれる関数には, ハードウェアとソフトウェアの両方のインタフェースを提供するようにする.

図 2 にその変換の例を示す.

図 2(a) は, 図 1(d) の `hw_sub.c` 中の関数 `_HW_func1`, `_HW_func2`, `_HW_func3` をそれぞれ 1 つのハードウェアに合成したものを表す. `_HW_func1` の箱の中の Wait はポーリングによる待機を, Load ARG は引数のロードを, 続く Store ARG, Set RUN, Wait, Load RET は `_HW_func3` を呼び出すための引数の格納, 起動, 完了の待機, 返り値のロードを, そして最後の Store RET と Reset RUN はこの関数の返り値の格納と完了の書き込みを行うデータフローグラフ (DFG) である. dfg1 と dfg2 は `_HW_func1` の本体の計算を行う DFG であるとする. `_HW_func2`, `_HW_func3` についても同様である.

図 2(b) は, 図 2(a) における `_HW_func1` から `_HW_func3` の呼び出し, および `_HW_func2` から `_HW_func3` の呼び出しに goto 変換を適用したものである. Store ARG によって引数を主記憶に格納する代わりに, Set ARG で `_HW_func3` が引数を受け取るレジスタに値をコピーし, `_HW_func3` の本体の先頭の DFG (dfg5) に直接遷移する. `_HW_func2` から `_HW_func3` の呼び出しも同様に変換する. `_HW_func3` の本体の最後の DFG である dfg6 の実行が終了すると, 呼び出しがどこから行われたかに従って次の DFG に遷移する. 呼び出しが `_HW_func1` から行われていた場合には, Get RET で引数の転送を行った後に, dfg2 に遷移する.

この時点で, `_HW_func3` がソフトウェアから呼ばれることがなければ, `_HW_func3` の Wait, Load ARG, Store RET, Reset Run を消去できるが, この例のようにソフトウェアからも呼ばれることがある場合には, ソフトウェアとのインタフェースを維持しつつ他のハードウェア関数からの呼び出しに対応する必要がある.

`_HW_func3` が先頭の Wait でソフトウェアからの起動を待つ間に, 他のハードウェア関数からの呼び出しがあれば待ちを解除して dfg5 に遷移するようにするのも一案であるが, 本稿では, `hw_sub.c` 内の関数が同時に呼ばれることはないことに留意し, 図 2(c) のように待機状態を集約する. 即ち, ハードウェアの起動を表す変数は `_RUN_sub` に集約し, その値によってどの関数が呼び出されたかを判定するようにする. 図の例では, `_RUN_sub = 0` はハードウェアが待機中であることを, `_RUN_sub = 1, 2, 3` は, それぞれ `_HW_func1`, `_HW_func2`, `_HW_func3` が呼び出されて実行中であることを表す. `hw_sub.c` 内の関数は, ソフトウェアから呼び出された場合には Wait, Load ARG, Store RET, Reset RUN を経由して実行されるが, ハードウェアから呼び出された場合には DFG 間の直接の遷移で実行される.

3.2 アセンブリコードの利用による関数の併合

図 2(c) に対応する DFG の生成は, 1) 図 1(b) のプログラムを C 言語のソースコードレベルで変換し, 2) これをコンパイルして得られるアセンブリコードに goto 変換を施すことにより実現できる.

図 1(b) のプログラムは図 3 に示すように変換する.

図 3(a) の `sw_stub.c` は, 図 1(a) の `main.c` とリンクして実行するソフトウェア側のインタフェースコードである. 図 1(c) との違いは, ハードウェアの起動を制御する変数を`_RUN_sub` に集約したことである. 例えば, 関数 `f1`, `f2`, `f3` の起動は, それぞれ 12 行目, 20 行目, 28 行目にあるように, `_RUN_sub` に `1`, `2`, `3` を代入することにより行う.

図 3(b) の `hw_sub.c` は, ハードウェアに合成するコードである. `_HW_stub` (9〜28 行目) は, 図 2(c) 中のソフトウェアからの呼び出しとのインタフェースをとる部分に相当し, `_RUN_sub` の値に応じて待機と各関数の実行を選択する. 30〜48 行目の `f1`, `f2`, `f3` は, 図 1(b) と同じである. 即ち, `hw_sub.c` は元の `sub.c` にハードウェア側のインタフェース関数を追加することにより得られる.

図 3(b) の `hw_sub.c` を例えば MIPS 用のコンパイラでコンパイルすると, 図 4 のようなアセンブリコードが得られる. ここでは最適化を行っていないコードを掲載しているが, 最適化したコードを用いることもできる. `f1` からは 8 行目の `jal` (jump and link) 命令で `f3` を呼び出しているが, その前後に引数や返り値を転送するコードが生成されている. また, 呼び出し先と呼び出し元でレジスタを共有した場合に必要になるレジスタの退避/復帰コードも生成されている.

Goto 変換は, このアセンブリコードを CDFG に変換する際に, 関数呼び出しを行う 8 行目の `jal` 命令と, 呼び出し元への戻りを行う 14 行目の `jr` (jump register) 命令の機能を変更することにより実現できる. `jal` 命令は, 戻り番地を保存するレジスタ `ra` に, 番地ではなく戻り先の DFG (10〜15 行目に対応する DFG) の ID を保存し, 関数 `f3` の先頭の DFG に分岐する, という機能に変換する. また, `jr` 命令は, `ra` に保存されている ID の DFG に分岐するという機能に変換する.

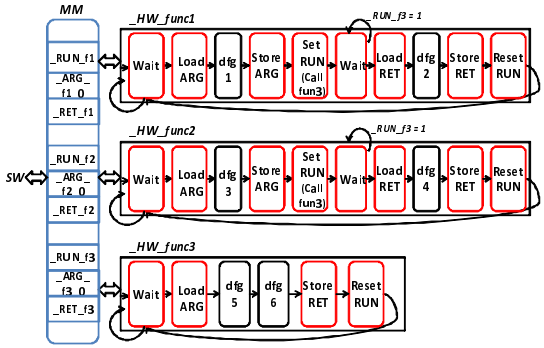
コンパイラは, 関数呼び出しに関して種々の最適化を行うので, 最適化されたアセンブリコードを利用すれば, 効率的なハードウェアを合成できる. 例えば, 引数/帰り値の受け渡しやレジスタの退避/復帰は必要なものだけに削減される. また, 関数呼び出しのインライン展開や goto 変換も, そのコンパイラ評価尺度に基づいて自動的に行われる場合がある.

3.3 戻り先アドレスの制御状態への変換

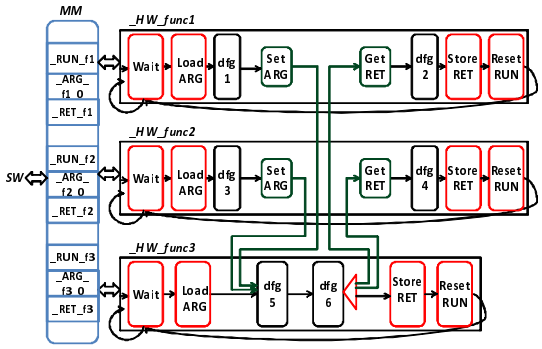
図 5(a) は, 前節の goto 変換の結果得られる CDFG の例である. dfg5 の次に実行される DFG は, dfg5 が dfg1 から呼び出されていた場合は dfg2, dfg3 から呼び出されていた場合は dfg4 であるとする. `ra` レジスタは呼び出し時に戻り先の DFG の ID を記憶しているので, dfg5 で ID の比較を行って, マルチウェイの分岐を行えば所望の DFG に戻ることができる.

しかし, この実装では, 呼び出し元が増えると, 比較器や遷移先を選択するマルチプレクサのハードウェアコストや遅延が無視できなくなる.

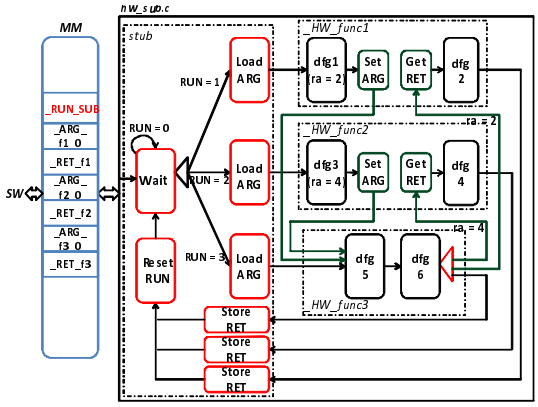
この問題は, アセンブリコードにおける命令のアドレスを制御状態 (のバイナリコード) に変換することにより解決できる. その概要を図 5(b) に示す. 関数呼び出しの際に, `ra` には戻り先 DFG の ID の代わりにその DFG の先頭状態のバイナリコードを代入しておく. そうすれば, dfg5 からの戻りは, 制御部の状態を表すレジスタ (`STATE`) に `ra` の値を代入するだけで実現できる.



(a) 1 関数 1 HW 化の合成



(b) Goto 変換による関数呼び出し



(c) SW から呼び出し可能な併合 HW

図 2　Goto 変換による関数の併合

sw_stub.c

```
 1 int _RUN_sub;
 2 int _ARG_f1_0;
 3 int _RET_f1;
 4 int _ARG_f2_0;
 5 int _RET_f2;
 6 int _ARG_f3_0;
 7 int _RET_f3;
 8
 9 int f1(int a1)
10 {
11  _ARG_f1_0 = a1;
12  _RUN_sub = 1;
13  while(_RUN_sub){;}
14  return _RET_f1;
15 }
16
17 int f2(int a2)
18 {
19  _ARG_f2_0 = a2;
20  _RUN_sub = 2;
21  while(_RUN_sub){;}
22  return RET_f2;
23 }
24
25 int f3(int a3)
26 {
27  _ARG_f3_0 = a3;
28  _RUN_sub = 3;
29  while(_RUN_sub){;}
30  return RET_f3;
31 }
```

(a) SW として実行するコード

hw_sub.c

```
 1 extern int _RUN_sub;
 2 extern int _ARG_f1_0;
 3 extern int _RET_f1;
 4 extern int _ARG_f2_0;
 5 extern int _RET_f2;
 6 extern int _ARG_f3_0;
 7 extern int _RET_f3;
 8
 9 void _HW_stub(void)
10 {
11  for(;;){
12   while(_RUN_sub==0){;}
13   if(_RUN_sub==1)
14   {
15    int a1 = _ARG_f1_0;
16    _RET_f1 = f1(a1);
17   } else if(_RUN_sub==2)
18   {
19    int a2 = _ARG_f2_0;
20    _RET_f2 = f2(a2);
21   } else if(_RUN_sub==3)
22   {
23    int a3 = _ARG_f3_0;
24    _RET_f3 = f3(a3);
25   }
26   _RUN_sub=0;
27  }
28 }
29
30 int f1(int a1)
31 {
32  int x1 = a1 + 1;
33  int r1 = f3(x1);
34  return r1;
35 }
36
37 int f2(int a2)
38 {
39  int x2 = a2 + 10;
40  int r2 = f3(x2);
41  return r2;
42 }
43
44 int f3(int a3)
45 {
46  int r3 = a3 * 10;
47  return r3;
48 }
```

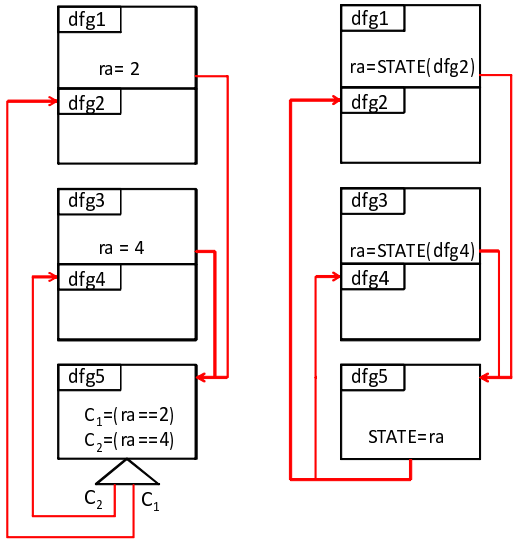
(b) HW に合成するコード

図 3　関数併合に対応したソースコード変換

```
<_HW_stub>:
 ......
0<f1>:
 1 addiu   sp,sp,-32
 2 sw      ra,28(sp)
 3 lw      v0,32(sp)
 4 sll     zero,zero,0x0
 5 addiu   v0,v0,1
 6 sw      v0,16(sp)
 7 lw      a0,16(sp)
 8 jal     <f3>
 9 sll     zero,zero,0x0
10 sw      v0,20(sp)
11 lw      v0,20(sp)
12 lw      ra,28(sp)
13 addiu   sp,sp,32
14 jr      ra
15 sll     zero,zero,0x0
16<f2>:
 ......
```

レジスタ退避コード
関数による計算
引数の設定(Set ARG)
ra ← 戻り先 DFG ID
Goto dfg(f3)
返り値の取得(Get RET)
レジスタ復帰コード
Goto dfg(ra)

図 4　中間表現となるアセンブリコード



(a) 遷移分岐によるリターン　(b) 制御状態への変換によるリターン

図 5　遅延を削減するリターン

4. 実装と実験

提案手法に基づく関数併合を高位合成システム ACAP [2] に実装した．ACAP は Perl (5.10.1) で実装されており，Linux, Mac OSX 及び Windows 上の Cygwin 環境で動作する．ソフトウェアを実行する CPU には MIPS を仮定している．入力された C プログラムは GCC (mips-elf-gcc 4.5.1) により MIPS のアセンブリに変換した後に CDFG に変換し，最終的に Verilog-HDL を出力する．

バインディングにはグリーディ法に基づく部分共有法 [6] を用いている．これは，バインディングの際に各演算器にもコストを定義し，マルチプレクサや結線のコストを減らせる場合にはスケジューリング時の制約以上の数の演算器を使用するというものである．論理合成には Xilinx ISE 12.2, シミュレーションには ModelSim-Altera 6.5b Starter Edition を用いた．

評価用の動作記述としては，関数が 9 個，関数呼び出し回数が 14 回，呼び出しの最大の深さが 9 であり，各関数では 1 回の加算を行う C プログラムを用いた．中間表現には動作記述を最適化なしでコンパイルして得られたアセンブリコードを用いた．

表 1 は生成した回路の演算数とサイクル数を併合前と併合後で比較した結果である．併合により，Load 演算数と Store 演算数はいずれも約 50%，実行サイクル数を約 15% 削減できている．

表 2 は生成した Verilog-HDL を論理合成した結果である．「Goto 変換」はリターンを比較と条件分岐により実装したものであり，「Goto 変換 (状態変換)」は戻り先アドレスを状態に変換してリターンを行うものである．併合により LUT 数と FF 数を大幅に削減できているが，これは関数間で演算器が共有されるようになったためである．遅延が増えているのは，演

算器の共有に伴いマルチプレクサの入力数が増加したためと考えられる．Goto 変換に比べ状態変換の遅延は減少しているが，これは状態遷移のためのマルチプレクサの入力数の減少によるものと考えられる．

表 1　演算数とサイクル数の評価

	Load 演算数	Store 演算数	総サイクル数
併合前	109	115	277
併合後	59 (54.1%)	54 (46.9%)	234 (84.4%)

表 2　論理合成結果

	LUT 数	FF 数	遅延 $_{(ns)}$
併合前	6363	1293	12.207
Goto 変換	2556	584	19.712
Goto 変換 (状態変換)	2447	357	17.130

5. む す び

本研究ではアセンブリコードを中間表現とする高位合成における関数の併合手法を提案した．本手法を高位合成システム ACAP に適用した結果実行サイクル数を約 15%, FPGA 上の LUT 数を約 60% 削減することができた．

今後の課題として，関数併合によるマルチプレクサの入力数を削減するためのバインディングアルゴリズムの改良が挙げられる．

謝　　辞

本研究を進めるにあたり，御助力・御討論を頂きました元立命館大学の中谷嵩之氏，立命館大学の安積祐子氏，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] 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 (2010 年 3 月).

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

[4] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Anderson, Stephen Brown, and Tomasz Czajkowsk: “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 (2011).

[5] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii: “Behavioral Synthesis of Double-Precision Floating-Point Adders with Function-Level Transformations: A Case Study,” in *Proc International Conference on Embedded Software and Systems (ICESS),* pp. 261–270 (May 2007).

[6] 田村真平, 丸谷亮祐, 石浦菜岐佐: “高位合成のバインディングにおける演算器の部分共有,” 電子情報通信学会ソサイエティ大会, A-3-6 (2011 年 9 月).