コンフィギュラブルプロセッサの命令セット拡張に対応したGCCの自動生成

吉田　昌平† 　久村　孝寛†† 　石浦菜岐佐† 　池川　将夫†† 　今井　正治†††

† 関西学院大学 理工学部　〒669–1337 兵庫県三田市学園 2–1
†† NEC　〒211–8666 神奈川県川崎市中原区下沼部 1753
††† 大阪大学　〒565–0871 大阪府吹田市山田丘 1–5

あらまし　本稿では, コンフィギュラブルプロセッサの命令セット拡張に対応する GCC を自動生成する手法を提案する. 本手法に基づく「C コンパイラジェネレータ」は, ベースプロセッサに対して追加する命令およびレジスタセットの記述から, その命令を intrinsic 関数 (コンパイラ組込み関数) として利用できる GCC のマシン記述を自動生成する. 追加したレジスタセットに対する変数のレジスタ割当も可能であり, そのレジスタセットに対する転送演算の自動挿入も行える. 本手法に基づく C コンパイラジェネレータ (CCG) を実装した結果, V850, ARM に, ベースプロセッサが持つレジスタを用いた命令追加を行えることを確認した. また Brownie32STD には, 命令追加及びレジスタ追加を行えることを確認した.

キーワード　コンフィギュラブルプロセッサ, GCC, intrinsic 関数

Automatic Generation for GCC for Insturction Set Extension on Configurable Processor

Shohei YOSHIDA†, Takahiro KUMURA††, Nagisa ISHIURA†, Masao IKEKAWA††, and Masaharu IMAI†††

† Kwansei Gakuin University,　Gakuen 2–1, Sanda, Hyogo, 669–1337, Japan
†† NEC Corporation,　Simonumabe 1753, Nakahara-ku, Kawasaki, Kanagawa, 211–8666, Japan
††† Osaka University,　Yamadaoka 1–5, Suita, Osaka, 565–0871, Japan

Abstract　This article proposes a method of autogenerating GCCs for extended instruction sets of configurable processors. A “C compiler generator” based on the method takes the specification of newly added instructions and newly added register sets so that it generates machine description files of a GCC which allows the use of the new instructions via intrinsic functions (built-in functions). The new register sets are available for register allocation of variables and necessary data transfer instructions associated with the register set are automatically inserted by the compiler. An implemented C compiler generator (CCG) successfully generated GCCs for V850 and ARM extensions which have new instructions that access their original general purpose registers. It also generated a GCC for a Brownie32STD processor augmented with a new set of registers as well as new instructions.

Key words　configurable processor, GCC, intrinsic functions

1. はじめに

近年のディジタル映像・音響機器等の高性能化に伴い, 組込みシステムのプラットフォームには益々高い処理能力を低コスト・低消費電力で実現することが求められている. このような要求に応えるために, 命令セットやレジスタ構成をカスタマイズ可能なコンフィギャラブルプロセッサが提案されている [1]. コンフィギュラブルプロセッサは, 利用目的に

応じてカスタマイズすることで, 応用に最適なプロセッサの探索にも利用できる. コンフィギュレーションをどれだけ変更できるかにも依存するが, 一般に個々の応用に対して専用プロセッサを一から設計するよりもコンフィギュラブルプロセッサを利用する方が, 開発期間やコストを抑制することができる. また, ソフトウェア開発のためには, アセンブラ, コンパイラ, リンカ, シミュレータ等のソフトウェア開発ツールが必要となるが, ベースプロセッサの資源を再利用できるコンフィギュラブルプロセッサは, これらのツールの開発の点でも有利と言える.

コンパイラはソフトウェア開発に不可欠なツールであるが, その開発はプロセッサとコンパイラに関する深い知識を必要とし, 非常に難易度が高い. コンフィギュラブルプロセッサが, ベースプロセッサとカスタマイズ部分から成り, ベースプロセッサのコンパイラが既に存在していれば, それをカスタマイズ部分へ対応させることによって, コンパイラを開発できる. カスタマイズ部分への対応は, ベースプロセッサのコンパイラの開発に比べれば容易であり, カスタマイズの制限を与えることで, 自動化することも可能となる.

プロセッサのアーキテクチャ仕様記述からコンパイラを自動生成する研究は既に行われている. Cosy (蘭 ACE 社)(注1), Lisatek (米 Coware 社)(注2)はアセンブラ, リンカ, シミュレータ等とともにコンパイラを生成することができる. しかし, 利用者コミュニティの規模やライセンスの制約から, 生成されたツールのチューニングや配布が強く限定されてしまう. Xtensa (米 Tensillica 社)(注3)プロセッサ用コンパイラジェネレータは命令セット拡張に対応した GCC (GNU Compiler Collection) の自動生成を行うが, 対象となるプロセッサは Xtensa プロセッサに限定される. [2] は GCC が存在しているプロセッサに対し, 覗き穴最適化を利用して追加命令を選択する GCC の自動生成を行うが, 追加命令が必ずしも意図通りに利用できないという問題がある.

本研究では, GCC が既に移植されているベースプロセッサに対し, 命令セット拡張における新たな命令に, 一対一で対応する intrinsic 関数が利用可能となる GCC を自動生成する手法を提案する. いくつかのプロセッサをベースプロセッサとし, 提案手法によって命令セット拡張に対応した GCC を生成できることを示す. intrinsic 関数は, コンパイラが生成する命令を直接制御できる組込み関数であり, これにより追加命令を意図通り利用した効率的なソフトウェア開発が可能となる. 提案手法はベースプロセッサへの命令追加だけでなく, レジスタ追加にも対応している. 提案手法によって生成された GCC は追加レジスタに関するレジスタ割り当てを行うことができる.

(注1) : http://www.ace.nl/
(注2) : http://www.coware.com/
(注3) : http://www.tensilica.com

```
1: (define_insn "mulsi3"
2: [(set (match_operand:SI 0 "r_operand" "=d")
3:       (mult:SI
4:        (match_operand:SI 1 "r_operand" "d")
5:        (match_operand:SI 2 "r_operand" "d")))]
6:  ""
7:  "mul\\t%0,%1,%2"
8:  [(set_attr "type" "alu")
9:  (set_attr "mode" "SI")])
```

図 1　define_insn の記述例

2. GCC と intrinsic 関数

GCC は GNU プロジェクトによって開発が行われているコンパイラ群である. GCC は GPL (GNU Public License) で配布されているため, ユーザが自由にチューニングし, 配布することができる. また GCC は多くのアーキテクチャに対応しており, 多くのプロセッサをベースプロセッサとして利用することができる.

機種依存の情報は「マシン記述」と呼ばれる以下のファイルに記述する (以下 *machine* はプロセッサ名で置き換える).

- *machine*.md: 命令パターンの定義
- *machine*.h: 機種仕様の定義
- *machine*.c: *machine*.h では補えない情報の記述

新しいアーキテクチャに GCC を対応させるには, これらのファイルを定義して GCC を構築すればよい. しかし, プロセッサのマシン記述作成は, プロセッサと GCC の詳細にわたる多くの知識を必要とするため, 新規プロセッサに GCC をリターゲッティングすることは難易度の高い作業となる.

プロセッサへの命令追加は, その命令情報を上記のファイルに追加することにより行う. レジスタクラスを追加する場合には, レジスタクラスの定義の他に, それに伴い必要となる転送命令の定義を行う必要がある.

2.1 GCC への命令追加

GCC への命令追加の方法としては, define_insn による命令パターンの定義, インラインアセンブラの利用, intrinsic 関数の利用がある. インラインアセンブラを利用する場合は, コンパイラが追加レジスタや追加命令に関する情報を知る必要がないためコンパイラの再構築をする必要がないが, その他の手法では必要な記述をマシン記述に追加した後にコンパイラを再構築する必要がある.

（1） define_insn

GCC は, 各機械命令の命令パターンと, 中間言語として生成された RTL (Register Transfer Language) とのマッチングを行うことにより, 出力するアセンブリ命令を選択する. RTL テンプレートとは, 機械命令が GCC からどのように見えるかを示すものであり, RTL とのマッチング及び各オペランドの記憶要素の割り当てに利用される. 命令パターン (演算の木構造表現) は define_insn を用いて定義し, 命令の名前, RTL テンプレート, パターンにマッチするかを最終的

```
 1: void foo(int a, int b)
 2: {
 3:     int x0, x1, x2;
 4:     x0 = 100;
 5:     asm volatile(
 6:          "ADDSUB %0,%1,R6,R7"
 7:          : "=r"(x1)
 8:          : "=r"(x2));
 9:     asm volatile(
10:          "ADD3  %0,R6,R7"
11:          : "=r"(x0));
12:     x0 = x0 - x1 - x2;
13:     return x0;
14:  }
```

図 2　インラインアセンブラの例

に決定する条件, 出力テンプレート, 命令の属性から成る.

図 1 に乗算命令の定義例を示す. 2–5 行目が RTL テンプレートであり, オペランド 1, 2 の乗算結果をオペランド 0 に格納することを意味する. RTL とこの命令パターンがマッチした場合に 7 行目のアセンブリが出力される. 図 1 のように単純な構成を持つ命令は define_insn で定義できるが, マルチメディア命令などの複雑な構成を持つ命令の場合は, マッチさせることが難しくなる.

（2） インラインアセンブラ

インラインアセンブラは, C 言語などにアセンブリ言語を埋め込むことを可能とするコンパイラの機能である. 図 2 にインラインアセンブラの例を示す. 以下, ADDSUB は 2 つのオペランドの和と差をそれぞれ別々にレジスタに格納するという命令, ADD3 は 3 つのオペランドの加算を行うという命令とする. 5–8 行目と 9–11 行目でインラインアセンブラを用い, これらの命令の利用を指定している. 7, 8, 10 行目のようにインラインアセンブラでは変数を利用することもできるが, その場合にはオペランド制約の記述が必要になる.

インラインアセンブラの利用は最適化の影響で問題を引き起こすことがある. インラインアセンブラで組み込んだ命令は命令スケジューリングの影響を受けるため, プログラマの意図とは異なる場所で実行される場合がある. そのため, 組み込まれた命令が副作用を伴う場合には, 実行結果が意図と異なる可能性が生じる. これは volatile 修飾子を付加して, コンパイラに最適化の対象としないことを指示することにより防止できるが, この部分はスケジューリングの対象外となってしまう上, 誤った volatile 修飾子の利用は他の部分の最適化に悪影響を及ぼし, コード効率が悪化する恐れがある.

インラインアセンブラを用いる場合には, コンパイラの再構築は不要である. しかし新しい命令の語長やレイテンシなどの情報をコンパイラに伝えることができないため, インラインアセンブラで記述されたコードに対して, コードサイズの計算や正しいスケジューリングが行えない等の問題が生じる. また, 追加レジスタに対してレジスタ割り当てを行うこともできない.

```
1: int function(int a, int b) {
2:   int x0, x1, x2;
3:   x0 = 100;
4:   x0 = __builtin_ADD3(x0, a, b);
5:   __builtin_ADDSUB(x1, x2, a, b);
6:   x0 = x0 - x1 - x2;
7:   return x0;
8: }
```

図 3　Intrinsic 関数の利用例

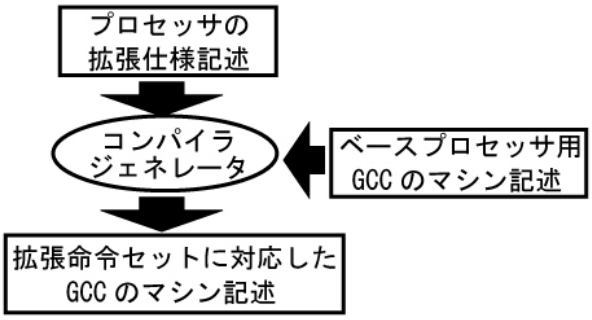

図 4　C コンパイラジェネレータの概念

（3） Intrinsic 関数

Intrinsic 関数はコンパイラが直接そのコード生成を制御できる組込み関数であり, x86 の拡張命令セット MMX, SSE, SSSE3(注4)などの利用にも用いられている.

Intrinsic 関数を用いて記述した図 2 と等価なプログラムを図 3 に示す. 4 行目の __builtin_ADD3 が ADD3 命令に対応しており, 変数 x0, a, b の和を x0 に格納する. 5 行目の __builtin_ADDSUB は ADDSUB 命令に対応し, 変数 a, b の和を x1, 差を x2 にそれぞれ格納する.

Intrinsic 関数をコンパイラに組み込むためには, コンパイラの再構築が必要になるが, 命令のスケジューリングに関わる情報や, 使用するレジスタをコンパイラが知っているため, 正しい命令スケジューリングとレジスタ割り当てが行われる.

2.2 GCC へのレジスタクラスの追加

レジスタを単純に追加するだけでなく, GCC が追加レジスタに関するレジスタ割り当てを行えるようにするためには, レジスタクラスの定義と, それに伴い必要となる転送命令の定義をマシン記述に行う必要がある. 転送命令は汎用レジスタとの転送, メモリ間との転送, 即値の代入, 同一レジスタ間のコピーに使われる転送命令を定義する. これにより GCC は追加レジスタに関するレジスタ割り当てを行えるようになり, これらのレジスタをオペランドとして用いる命令の追加が可能になる.

3. 命令セット拡張に対する GCC の自動生成

3.1 概　　要

本稿では命令セット拡張に対応した GCC のマシン記述を

(注4)：http://www.intel.com/

```
 1: (define_insn "builtin_ADD3"
 2: [
 3: (set (match_operand:SI 0 "general_regs" "=d")
 4:    (unspec:VOID
 5:     [
 6:     (match_operand:SI 1 "general_regs" "0")
 7:     (match_operand:SI 2 "general_regs" "d")
 8:     (match_operand:SI 3 "general_regs" "d")
 9:     ]  UNSPEC_BUILTIN_ADD3))
10: ]
11: ""
12: "ADD3 %0 %2 %3"
13: [])
```

図 5　命令追加に必要となる記述 (1)

```
 1: tree ftype_ADD3 = build_function_type_list(
 2:                  integer_type_node,
 3:                  integer_type_node,
 4:                  integer_type_node,
 5:                  integer_type_node,NULL_TREE);
 6:
 7: builtin_function("__builtin_ADD3",
 8:                 ftype_ADD3,
 9:                 CODE_FOR_builtin_ADD3,
10:                 BUILT_IN_MD,NULL,NULL_TREE);
```

図 6　命令追加に必要となる記述 (2)

自動生成する「C コンパイラジェネレータ」を提案する. 図 4 にその概念を示す. C コンパイラジェネレータは, ベースプロセッサの GCC のマシン記述と, 仕様の拡張記述を入力とし, その拡張に対応した GCC のマシン記述を自動生成する. ここでプロセッサの仕様拡張には, 命令の追加及びレジスタクラスの追加が許される. 自動生成された GCC は, 追加命令を intrinsic 関数として利用することができ, 追加レジスタに関するレジスタ割り当てを行うことができる. なお, GCC は C 言語以外に C++, Java, Fortran などに対応しているが, 本研究では C コンパイラのみを扱うものとする.

3.2　命令追加

追加命令を intrinsic 関数として定義するために以下の記述を行う必要がある

（1） 命令パターンの定義
（2） Intrinsic 関数の定義
（3） 命令を組み込むための関数呼出し

(1) は通常の命令と同様 define_insn を用いて記述する. Intrinsic 関数と追加命令は一対一に対応しているため, コンパイラが命令選択に利用する命令のセマンティックスの情報は不要である. そのため, この RTL テンプレートでは単一の節点のみからなる演算の木とオペランドの制約のみを記述する. (2) は引数や返り値の型を定義するものである. (3) は intrinsic 関数の名前, 利用する命令パターンを指定し, GCC に組み込むための関数呼出しである.

3 つのオペランドの加算を行う ADD3 を intrinsic 関数として定義する例を図 5, 図 6 に示す. 図 5 は命令パターンの定義であり, *machine*.md に記述する. 図 6 は関数定義と intrinsic 関数として GCC に組み込むための関数呼び出しであり, *machine*.c に記述する.

```
 1: (define_insn "movsi_general"
 2: [(set
 3:   (match_operand:SI 0 "nonimmediate_operand"
 4:                       "=d,d,m,d,d,c,m,c,c")
 5:   (match_operand:SI 1 "general_operand"
 6:                        "i,m,d,d,c,d,c,m,i"))]
 7: ""
 8: "@
 9:  addi %0, r0, %1
10:  lw %0, %1
11:  sw %1, %0
12:  add %0, r0, %1
13:  m2g %0, %1
14:  g2m %1, %0
15:  med2mem %1, %0
16:  mem2med %0, %1
17:  mem2med %0, %1"
18: [])
```

図 7　自動生成される転送命令の命令パターン

これらの記述は命令のニーモニック, 各入出力オペランドの記憶要素, 出力するアセンブリとそのフォーマットから自動生成できる.

3.3　レジスタクラス及び転送命令の追加

レジスタクラスの追加は *machine*.h と *machine*.c に, そのレジスタクラスに関するマクロ, 及びそれらが指定する値と関数群を定義することにより行える. 表 1 に必要となる記述を示す. 追加レジスタを用いた命令を追加するためには, 転送命令が必要になる. 転送命令は関数呼出しの引数や返り値の受渡し, レジスタのスピル・リロードのためにコンパイラによって自動的に挿入される. データ転送, スピル・リロード, 即値の代入を行うために, それらに用いられる転送命令を定義する必要がある.

図 7 に転送命令の定義に必要となる記述を示す. 3–4 行目で転送先, 5–6 行目で転送元を指定している. 8–17 行目で各組合せで出力するアセンブリフォーマットを定義している. 転送命令の記述は *machine*.md に行う.

レジスタクラスの追加を行うためレジスタクラスの名前, ビット長, レジスタ本数, プレフィックスが必要となる. 転送命令の追加には転送元・転送先の記憶要素の情報, 転送に必要となるアセンブリ命令が必要となる. 汎用レジスタ, メモリとのデータ転送, 即値の代入, 同一レジスタでの転送に用いる命令を定義する必要がある.

4.　C コンパイラジェネレータとプロセッサの拡張仕様記述

以上の考察に基づき, C コンパイラジェネレータ (CCG)

表 1　レジスタ追加に必要な記述

必要となる記述	内 容
reg_class	すべてのレジスタクラスの値を定義する列挙型
REGISTER_NAMES	マシンレジスタのアセンブリでの名前を定義するマクロ
REG_CLASS_NAMES	レジスタクラス名を文字列として定義するマクロ
REG_CLASS_CONTENTS	レジスタクラスの内容をビットマスクである整数として定義するマクロ
REG_CLASS_FROM_LETTER	レジスタクラスについてのオペランド制約文字を定義するマクロ
HARD_REGNO_MODE_OK	モードとレジスタ番号を受け取り, そのモードで指定されたレジスタが利用可能かを判定するマクロ.
FIRST_PSEUDO_REGISTER	コンパイラに知らせるべきハードウェアレジスタの個数を示すマクロ.
FIXED_REGISTERS	決まった目的で使用され, 一般利用できないレジスタを指定するマクロ.
CALL_USED_REGISTERS	関数内であるレジスタが利用される場合, 破壊されるか否かを指定するマクロ.

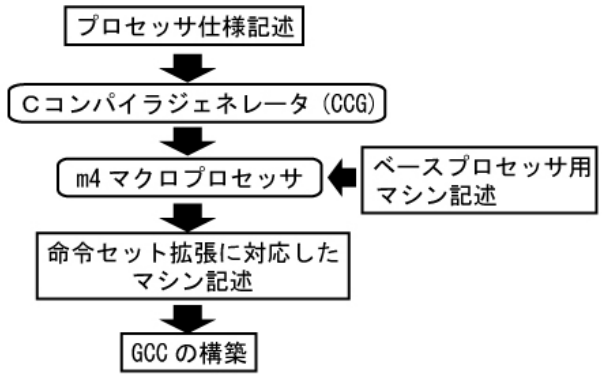

図 8　CCG の処理の流れ

及び入力となる拡張仕様記述の設計を行った. CCG の処理の流れを図 8 に示す. CCG はプロセッサ仕様記述 (XML) を入力とし, 命令セット拡張に必要となるマシン記述の断片を m4 マクロ形式で出力する. m4 はマクロプロセッサの一種であり, ベースプロセッサのマシン記述に挿入されたマクロを展開する. そして得られたマシン記述を用いてコンパイラを構築することにより, 命令セット拡張に対応した GCC を得る. 以下では, CCG の入力ファイルであるプロセッサの拡張仕様記述と CCG の出力について説明する。

```
<name>v850</name>
<reg_type length="32">GR</reg_type>
<reg_type length="32">MM</reg_type>
<reg_bank type="GR" size="32" prefix="r">
   GENERAL
</reg_bank>
<reg_bank type="MM" size="4" prefix="m">
   MEDIA
</reg_bank>
```

図 9　プロセッサ情報

4.1 仕 様 記 述

プロセッサの拡張仕様記述はプロセッサ情報, 命令情報, 転送命令情報より成る.

（1） プロセッサ情報

プロセッサ情報の記述例を図 9 に示す. reg_type タグはレジスタの種類を示し, reg_bank タグは各レジスタの本数とプレフィックスを指定する. 図 9 ではそれぞれ 32 ビットの GR レジスタ, MM レジスタから構成されるレジスタバンク GENERAL, MEDIA を定義している.

```
<insn>
 <mnemonic>ADD3</mnemonic>
 <syntax>ADD3 %rd %rm %rn</syntax>
 <input>
  <operand type="GENERAL" width="32">rd</operand>
  <operand type="GENERAL" width="32">rm</operand>
  <operand type="GENERAL" width="32">rn</operand>
 </input>
 <output>
  <operand type="GENERAL" width="32">rd</operand>
 </output>
</insn>
```

図 10　命 令 情 報

```
<insn move_insn="yes">
 <mnemonic>M2G</mnemonic>
 <syntax>M2G %rd %rm</syntax>
 <input>
   <operand type="MEDIA">rm</operand>
 </input>
 <output>
   <operand type="GENERAL">rd</operand>
 </output>
</insn>
```

図 11　転送命令の XML 仕様 (一部属性省略)

（2） 命令情報

3 つの変数の加算を行う *ADD3* の例を図 10 に示す. mnemonic タグは命令の名前. syntax タグはアセンブリのシンタックスで %に続く記号は operand タグの情報により置き換えられる. input, output タグはそれぞれ命令の入力オペランド, 出力オペランドを示す.

（3） 転送命令情報

転送命令の定義は基本的に intrinsic 関数の定義と同じであり, insn タグの move_insn 属性を “yes” とすることで転送命令であることを指示すればよい. input タグで転送元,

```
m4_define(<<<<__HARD_REGNO_MODE_OK__>>>>,
<<<<
if( mode == SI && regno >= 0 && regno < 32)
      return 1;
>>>>)
m4_define(<<<<__enum_reg_class__>>>>,
<<<<
enum reg_class{
  NO_REGS,
  GENERAL_REGS,
  ALL_REGS,
  LIM_REG_CLASSES
};
>>>>)
```

図 12　CCG の出力 (一部)

```
1:   mov     r3, r0
2:   mov     r0, #100
3:   ADDSUB  ip, r2, r3, r1
4:   ADD3    r0, r3, r1
5:   rsb     r0, r2, r0
6:   rsb     r0, ip, r0
```

図 13　図 3 のコンパイラ出力の一部 (ARM)

output タグで転送先, syntax タグで転送の際に使われるアセンブリ命令を示す. 図 11 に MEDIA レジスタから汎用レジスタへの転送命令の定義を示す. input タグで転送元, output タグで転送先, syntax タグで利用する命令を定義している.

4.2 出力情報

CCG からの出力の一部を図 12 に示す. 必要となる記述をそれぞれ m4 マクロとして出力する. ユーザはベースプロセッサ用マシン記述にマクロを挿入することで, 命令セット拡張に対応した GCC を得ることができる.

5. 実　　装

プロセッサ仕様記述から命令セット拡張に必要な記述を自動生成する CCG を C++を用いて実装した. Mac OSX, Debian GNU/Linux, Cygwin (Windows XP) で動作することを確認した. 対応する GCC のバージョンは 3.4.6 から 4.0.2 である.

命令追加とレジスタ追加を 32 ビット RISC プロセッサ BrownieSTD32 [1] に適用し, 命令及びレジスタ割り当てが行われることを確認した. V850, ARM などその他のアーキテクチャではベースプロセッサが持つレジスタを用いた命令追加のみ行えることを確認した. ARM に ADD3, ADDSUB 命令を追加し, 図 3 のプログラムをコンパイルして, 図 13 に示す正しいコードが得られた (最適化オプション "–O2").

現在の CCG はベースプロセッサの GCC への命令追加が可能であり, 一部のプロセッサではレジスタ追加も行える.

6. ま と め

本稿ではコンフィギュラブルプロセッサの命令セット拡張に対応した GCC の自動生成手法を提案した. 本手法により, コンフィギュラブルプロセッサ向けに迅速にコンパイラが開発でき, 効率的なソフトウェア開発を行うことができる. 本コンパイラジェネレータは追加命令の効率的な評価を可能にするため, コンフィギュラブルプロセッサのアーキテクチャ探索ツールとしても有用であると考える. GCC は GPL で配布されているため自由にチューニングや, 自由に配布を行うことができる.

現在の CCG では, 追加できるレジスタクラスの制限, 複雑な制御を行う転送命令を持つベースプロセッサへのレジスタ追加, 即値の扱いなどに関して制限がある. 今後は多くのベースプロセッサへの対応と任意の命令, レジスタの追加への対応を検討していく予定である.

謝　　辞

本研究を進めるにあたり, 御助言, 御指導頂いた大阪大学 武内良典准教授, NEC 石原希実氏, 大阪大学 小林悠記氏, 稗田拓路氏, 渡辺愛子氏はじめ大阪大学今井研究室のみなさんに感謝致します. また支援と助言を頂いた内山裕貴氏, 森本剛徳氏はじめ関西学院大学石浦研究室の関係諸氏に感謝致します.

文　　献

[1] 岩戸宏文, 稗田拓路, 田中浩明, 佐藤淳, 坂主圭史, 武内良典, 今井正治: "ASIP 短期開発のための高い拡張性を有するベースプロセッサの提案," 信学技報 VLD2007-92, pp. 19–24 (Nov. 2007).

[2] 永松祐二, 石浦菜岐佐, 引地信之 : "命令セット拡張に対する GCC 及び GNU Tool Chain のリターゲッティング," 信学技報 VLD2005-103, pp. 37–41 (Jan. 2006).

[3] Richard M. Stallman and the GCC Developer Community: *GNU Compiler Collection Internals* (2007).