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

永松 祐二† 石浦菜岐佐† 引地 信之††

† 関西学院大学 理工学部 〒669-1337 兵庫県三田市学園2-1
†† SRA 先端技術研究所 〒160-0004 東京都新宿区四谷3-12 丸正ビル5F
E-mail: †{nagamatsu,ishiura}@ksc.kwansei.ac.jp, ††hikichi@sra.co.jp

あらまし ASIP (Application Specific Instruction set Processor) を用いる上で課題となるのはソフトウェアの開発環境である．アセンブラ，リンカ，命令セットシミュレータ，コンパイラ，デバッガ等の開発ツールはプロセッサのアーキテクチャに強く依存し，その開発には多大な労力を要する．本稿では，新たなプロセッサが既存プロセッサの命令セットを拡張することにより設計される場合を想定し，既存プロセッサのGCC及びCGEN用マシン記述と追加命令の記述から拡張命令セットに対応したGCCとGNU Tool Chainによるクロス開発環境を効率的に構築する手法を提案する．
キーワード ASIP, ソフトウェア開発環境, GCC, GNU Tool Chain, リターゲッティング

Retargeting GCC and GNU Toolchain for Extended Instruction Set

Yuji NAGAMATSU†, Nagisa ISHIURA†, and Nobuyuki HIKICHI††

† School of Science & Technology, Kwansei Gakuin University, Sanda, 669-1337 Japan
†† SRA Key Technology Laboratory, Marusho Bldg, 3-12 Yotsuya, Shinjuku-ku, Tokyo, 160-0004 Japan
E-mail: †{nagamatsu,ishiura}@ksc.kwansei.ac.jp, ††hikichi@sra.co.jp

Abstract While ASIPs (Application Specific Instruction set Processors) are attractive components for embedded systems, constructing software developing environments is one of the challenges in utilizing them. Since tools like an assembler, a linker, a simulator, a compiler, and a debugger are sophisticated softwares that strongly depends on the processor architectures, it requires great efforts to construct the software developing environments. In this article, focusing on the case where new processors are designed by augmenting the instruction set of an existing processor, we propose an efficient method of building software cross developing environments for the new processors consisting of the GCC and the GNU Tool Chain from the GCC/CGEN machine descriptions for the original processor and those of the extended instructions.
Key words ASIP, softaware developing environment, GCC, GNU Tool Chain, retargeting

1. はじめに

ASIP (Application Specific Instruction set Processor) は，特定の応用に命令セットを最適化することによって性能，コスト及び消費電力のトレードオフを追求できるため，制約の厳しい組み込みシステムへの搭載が拡大している．

新たに開発したASIPを利用する際に課題となるのがソフトウェアの開発環境である．ソフトウェアを効率的に開発するためにはアセンブラ，リンカ，命令セットシミュレータ，コンパイラ，デバッガ等のツールが必須となるが，これらはプロセッサのアーキテクチャに強く依存する複雑なソフトウェアである．このため，ASIP自身の開発に加えてソフトウェア開発環境の構築にも多大な労力が必要となり，これがASIP利用の大きな障壁となり得る．

そこで，ASIP Meister (大阪大学) [1](注1)，LISA (米CoWare社)(注2)，CHESS/CHECKERS (ベルギーTARGET社)(注3)，Xtensa (米Tensilica社) (注4) 等のASIP設計システムでは，ASIPの設計とともにソフトウェア開発環境の構築をサポートしている．これらのシステムにはそれぞれ長短があるが，一般に，扱えるプロセッサ・アーキテクチャのクラスとソフトウェ

(注1)：http://www.eda-meister.org/asip-meister/
(注2)：http://www.coware.com/
(注3)：http://www.retarget.com/index.html
(注4)：http://www.tensilica.com/

ア環境のリターゲッティングの容易さ (あるいは自動化の程度やコンパイラの生成するコードの品質) にはトレードオフが存在すると考えられる. また得られるソフトウェア開発環境のライセンスも ASIP の利用形態によっては重要な問題となる.

我々は, ASIP に対して GCC 及び GNU Tool Chain によるクロス開発環境を構築することを試みている. これらのソフトウェアは GPL もしくはそれに準じたライセンスのもとで配布されているため, 新たな ASIP を既存のプロセッサの拡張や変更によって設計する場合, リソースの流用が可能である. しかし, GCC 及び GNU Tool Chain のリターゲッティングには, プロセッサに関する詳細な知識とともに, それを GCC や GNU Tool Chain への入力記述として与える方法に関し, 多くの知識と経験を必要とする.

そこで, 既に GCC や GNU Tool Chain が存在するプロセッサに専用命令を追加することによって新しい ASIP が設計される場合を想定し, 追加命令に関する記述から拡張命令セットに対応した GCC 及び GNU Tool Chain の構築を効率的に行う手法を提案する. これは, 各追加命令に対してその動作, ニーモニック等の情報を記述したファイルを作成し, ここから GCC 及び CGEN のマシン記述の拡張部分を生成して, 新たな ASIP 用の環境のビルドを行うというものである. 我々はこの処理手法の実装を行い, M32R [3] に対して, 12 の新たな命令を追加したプロセッサのクロス開発環境を構築する適用実験を行った.

以下, 2 章では GCC 及び GNU Tool Chain のリターゲッティングについて述べ, 3 章で我々の提案する手法を述べる. 4 章ではこの手法に基づくシステムの実装と, M32R を対象とした実験結果について述べる.

2. GCC, GNU Tool Chain, NEWLIB の構築

GCC [2] は GNU の提供するコンパイラ (gcc) である. また GNU Tool Chain は Binuitls, GDB 等からなる開発環境である. Binutils は, アセンブラ (gas), リンカ (ld), オブジェクトファイル操作コマンド (objcopy, objdump) 等を含むバイナリ・ユーティリティ群である. GDB はデバッガ (gdb) を提供するパッケージであるが, クロス開発環境ではソフトウェアの実行環境 (命令セットシミュレータ: ISS) として不可欠な役割も果たす. NEWLIB は I/O や数学関数等を含む C のライブラリセットである.

これらのソフトウェアによりクロス開発環境を構築する手順は以下の通りである.

(1) Binutils の構築.
(2) GCC の構築.
(3) GDB の構築.
(4) NEWLIB の構築.
(5) 構築した環境のテスト.

(2) の GCC 構築に先立ち (1) の Binutils が必要なのは, gcc コマンドが gas や ld といった Binutils の提供するコマンドを必要とするためである. NEWLIB は作成した Binutils と GCC によりビルドする. (5) のテストは [4] 等の C コンパイラ用テ

表 1 opcodes ライブラリ用マシン記述

†Binutils-2.16.1 の場合

ファイル名	行数
m32r-asm.c	761
m32r-desc.c	1,574
m32r-desc.h	240
m32r-dis.c	680
m32r-ibld.c	1,236
m32r-opc.c	1,835
m32r-opc.h	144
m32r-opinst.c	766

ストスイートを用いて行う.

GCC 及び GNU Tool Chain のリターゲッティングは, ターゲットアーキテクチャに関する情報の記述 (マシン記述) を作成することにより行う.

2.1 Binutils の構築

Binutils のリターゲッティングには bfd ライブラリ, opcodes ライブラリの構築, 及びアセンブラ, リンカ等のユーティリティのためのマシン記述がそれぞれ必要となる. 例えば, M32R プロセッサ向け opcodes ライブラリを構築するために必要なマシン記述は表 1 の通りである (Binutils-2.16.1 の場合). マシン記述は複数のファイルから構成されていて記述量も多いため, リターゲッティングは非常にコストの高い作業となる.

2.2 GCC の構築

GCC のリターゲッティングもマシン記述を作成することにより行う. GCC のマシン記述は主に以下のファイルから成る. ただし *machine* はターゲットプロセッサの名前で置き換えられる.

- *machine*.md: 命令パターンの定義
- *machine*.h: 機種の仕様を定義するマクロ, 及び機種固有のサブルーチンの宣言等
- *machine*.c: 機種固有のサブルーチンの本体

machine.md には各命令が利用する機能ユニットや遅延の定義, 及び命令パターンの定義等を記述する. *machine*.h にはレジスタ数やデータタイプのビット幅等プロセッサの仕様を定義するマクロの値のセットや, *machine*.md に記述するには複雑すぎる機種固有のサブルーチンの宣言を記述し, このサブルーチンの本体を *machine*.c に記述する.

2.3 GDB の構築

GDB のリターゲッティングもマシン記述の作成により行う. GDB のリターゲッティングには bfd ライブラリ, opcodes ライブラリの構築, 及びデバッガ, 命令セットシミュレータのリターゲッティングが必要である. ただし, bfd ライブラリと opcodes ライブラリは Binutils と同じものが利用できる. M32R プロセッサの ISS を構築するために必要なマシン記述の一覧を表 2 に示す. GDB のマシン記述はターゲット毎, ISA モデル毎に必要となるため, 作業コストが非常に高い.

2.4 CGEN

CGEN(注5) (米 RedHat 社) は独自のフォーマットで記述さ

(注5) : http://sources.redhat.com/cgen/

表 2 ISS (run) 用マシン記述

ファイル名	行数
m32r/arch.c	50
m32r/cpu.c	181
m32r/cpu.h	691
m32r/cpu2.c	197
m32r/cpu2.h	1,046
m32r/cpux.c	197
m32r/cpux.h	1,046
m32r/decode.c	2,113
m32r/decode.h	101
m32r/decode2.c	2,609
m32r/decode2.h	151
m32r/decodex.c	2,571
m32r/decodex.h	149
m32r/devices.c	107
m32r/m32r-sim.h	212
m32r/m32r.c	414
m32r/m32r2.c	311
m32r/m32rx.c	311
m32r/model.c	4,359
m32r/model2.c	3,253
m32r/modelx.c	3,071
m32r/sem-switch.c	2,615
m32r/sem.c	2,814
m32r/sem2-switch.c	6,622
m32r/semx-switch.c	6,654
m32r/sim-if.c	306
m32r/sim-main.h	94
m32r/traps-linux.c	1,392
m32r/traps.c	199

れたターゲットプロセッサのマシン記述から, Binutils, GDB のマシン記述の一部を生成する. CGEN は Guile (Scheme の GNU による実装) で実装されており, Binutils, GDB の構築時に Makefile から利用する. Binutils 及び GDB の必要とするマシン記述は非常に多くのファイルから構成され, その内容も複雑である. しかし, CGEN は少数のファイルからこれらのファイル全てを生成するため, 非常に効率的で保守性も高い.

CGEN のマシン記述は次のファイルからなる. *machine* はターゲットプロセッサ名で置き換えられる.

- *machine*.cpu: 命令パターン, データタイプ, ISA 等.
- *machine*.opc: opcode library 構築に補助的に必要な情報等.

ただし, CGEN のみで Binutils, GDB の完全なリターゲッティングを行うことはできない. CGEN は opcodes ライブラリ, アセンブラ, 命令セットシミュレータに関するマシン記述のみ生成し, bfd ライブラリやデバッガに関するマシン記述の生成は行わないからである.

3. 拡張命令セットに対するマシン記述の生成

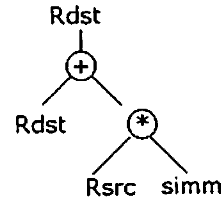
我々はあるプロセッサに対して GCC 及び CGEN を用いて構築された GNU Tool Chain が存在しており, このプロセッサに対して専用命令を追加することによって新しい ASIP が設計されるという場合を想定し, このような状況において新しい ASIP 用の GCC 及び GNU Tool Chain を効率的に構築する手法を提案する. ただし, 追加する命令に対して次の仮定を置く.

(1) 汎用レジスタや特殊レジスタの数, 種類は変更しない.

(2) パイプラインのステージ数の変更や演算器の追加は行わない.

(3) 命令フォーマットの追加や変更は行わない.

以上を仮定すると, GCC, CGEN とも追加命令に対する記述のみを元のマシン記述に追加するだけで開発環境の構築が実現で



Rdst=Rdst+Rsrc*simm16

図 1 命令 "adml" の動作

```
1:  (dni adml "adml"
2:       ((IDOC ALU))
3:       "adml $dr,$sr,$simm16"
4:       (+ OP1_12 OP2_5 dr sr simm16)
5:       (set dr (add dr (mul sr simm16)))
6:       ()
7:  )
```

図 2 CGEN の命令定義

きる.

3.1 CGEN を利用した Binutils 及び GDB 用マシン記述の生成

前述の制限のもとでは, 命令セット拡張の影響は opcodes ライブラリ, アセンブラ, 命令セットシミュレータに関するマシン記述にしか波及しないため, CGEN のみで Binutils, GDB のリターゲッティングが可能となる.

例えば, 図 1 のような命令 adml を拡張命令として M32R プロセッサに追加するためには, 図 2 に示すエントリを ".cpu" ファイルに追加すればよい. 1 行目はこのマッチング規則の定義する命令のニーモニック及びコメント, 2 行目はこの命令の属性, 3 行目は出力されるアセンブリコードのシンタックス, 4 行目は命令を構成するフィールドのリスト, 5 行目は命令のセマンティクスである. 6 行目は使用する機能ユニットや遅延の定義で, 何も指定しない場合はデフォルトのパイプラインや演算器が使用される.

3.2 define_peephole による GCC 用拡張命令定義

GCC に利用可能な命令を追加するためには define_insn マクロを利用する方法と define_peephole マクロを利用する方法等が考えられるが, 我々は define_peephole を利用する.

define_insn マクロは命令を定義する最も自由度の高い方法である. 例えば図 1 の命令 adml を M32R プロセッサの命令セットに追加する場合, define_insn マクロを利用すると図 3 のような記述になる. (ただし, この記述は不完全である. この記述ではレジスタ・アロケーションに失敗し gcc が異常終了することがある.) この定義は, プログラムから得られる RTL 式に 2~5 行目で示す式のパターンがマッチするならば, 6 行目の条件に反しない限り, 7 行目に示すテンプレートにしたがってアセンブリコードを出力するということを意味する. 8, 9 行目は命令の属性の定義であり, 命令スケジューリングに利用される. match_operand で始まる式はオペランドのプレースホルダであり, 左からオペランドのモード, 番号, 種類, データタイプを示

```
1: (define_insn
2:   [(set (match_operand:SI 0 "register_operand" "=r")
3:      (plus:SI (match_operand:SI 1 "register_operand" "0")
4:           (mult:SI (match_operand:SI 2 "register_operand" "r")
5:                    (match_operand:SI 3 "nonmemory_operand" "J"))))
6:   ""
7:   "adml %0,%2,%3"
8:   [(set_attr "type" "int4")
9:    (set_attr "length" "4")])
```

図 3 define_insn による GCC の命令定義 (不完全)

```
1: (define_peephole
2:   [(set (match_operand:SI 0 "register_operand" "")
3:         (mult:SI (match_operand:SI 1 "register_operand" "")
4:                  (match_operand:SI 2 "nonmemory_operand" "")))
5:    (set (match_operand:SI 3 "register_operand" "")
6:         (plus:SI (match_operand:SI 4 "register_operand" "")
7:                  (match_operand:SI 5 "register_operand" "")))]
8:   "REGNO (operands[0]) == REGNO (operands[5])"
9:   "adml %3,%1"
10:  [(set_attr "type" "int4")
11:   (set_attr "length" "4")])
```

図 4 define_peephole による GCC の命令定義

す制約子を指定する.

GCC は RTL 中間表現の生成時に define_insn 式を参照する. プログラムをパースした結果図 1 の木構造で表現される RTL 式が生成できれば, 図 3 の define_insn 式が示すを生成する. しかし, GCC はこの時点 (命令選択) では制約子を参照しない. このため, GCC は図 1 の木構造に対し, 制約子の内容に関わらずこの命令を生成する. ところが M32R の命令フォーマット (図 7) では, 引数として 32bit の即値を扱うことができない. このため, オペランドが 32bit 即値であった場合, レジスタ・アロケーション時にリロード・エラーが発生し, GCC は異常終了する.

この問題を回避する最も単純な方法は, オペランドのモードを HI (Half Integer) にして, 16bit 以上の即値ではこのエントリが使用されないようすることである. しかし, GCC は C プログラム中の整数即値を 32bit として処理してしまうため, この命令パターンは絶対にマッチしなくなる. 代わりに図 1 の即値オペランドが 16bit 以上の値であった場合には 2 命令に分割するようすればリロード・エラーを回避できるが, そのためにはプロセッサの利用できる命令や制約子をあらかじめ知る必要があり, 環境構築の自動化が難しくなる.

そこで我々は, define_peephole マクロで追加命令を定義する. define_peephole もプログラムから得られる RTL 式が, 特定の RTL 式の列にマッチしたとき, これを指定した命令に置き換えることを指示するが, define_peephole はレジスタアロケーション後に適用されるという点で define_insn と大きく異なる. 図 1 の命令を define_peephole で記述したものが図 4 である. 2 行目から 7 行目はマッチングすべき RTL 式列である. マッチする RTL 式列が現れた場合, 8 行目の適用条件が検査される. 適用条件は C の式で表され, 省略された場合, エントリは無条件に適用される. この例では 0 番目のオペランドと 5 番目のオペランドが同じレジスタであるときマッチングが成立し, 命令が生成される. 10 行目は出力されるアセンブリ・シンタックスである. これはニーモニック及びオペランドのプレースホルダからなり, 数字は図 4 の RTL 式中の番号に対応する.

生成される RTL 式はコンパイル時の最適化レベルによっても変化するため必ずしもこの定義が効果的に利用されるとは限らない. しかし, define_peephole で定義された命令は命令選択及びレジスタアロケーション後に適用されるため, オペランドに対する制約つまりリロードを気にする必要がない. またレジスタ番号が決定しているため, この命令が利用される条件をより直観的に定義することが可能となる.

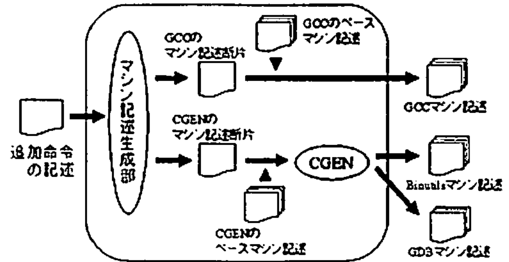


図 5 システムの構成

```
1:  <insn>
2:    <mnemonic> adml $dr,$sr,$simm16</mnemonic>
3:    <behavior> dr=dr+sr*simm16 </behavior>
4:    <format> OP1 OP2 dr sr simm16 </format>
5:    <opcode OP1=12 OP2=5 />
6:    <attribute length=4 type=int4 />
7:  </insn>
```

図 6 追加命令の記述例

4. 実装及び M32R プロセッサへの適用

4.1 実　　装

我々は, 3 章の手法に基づき, 追加命令の記述から新しいプロセッサのクロス開発環境を自動構築するシステムの実装を行った. 利用したバージョンは GCC が 3.3.4, Binutils が 2.16.1, GDB が 6.2, NEWLIB が 1.12.0 である. 図 5 に処理の流れを示す. 実装はマシン記述生成部とリターゲッティング部からなっており, マシン記述生成部は XML 形式で記述した入力ファイルをもとに GCC 及び CGEN のマシン記述の断片を生成し, これとベースプロセッサのマシン記述から拡張命令セットに対応した完全なマシン記述を生成する. 最後にリターゲッティング部がこのマシン記述から拡張命令セット対応のソフトウェアクロス開発環境を構築する. 処理系は Perl-5.8.4 で実装した.

M32R プロセッサに対し図 1 の adml 命令を追加する場合の入力ファイルの例を図 6 に示す. ユーザは追加命令のニーモニック, 命令フォーマット, 動作, オプコード, 命令の属性を記述する.

4.2 M32R プロセッサへの適用

M32R [3] はルネサス社製の 32bit RISC であり, 16bit と 32bit の複数の命令フォーマット (図 7) を持つ. 本研究ではこ

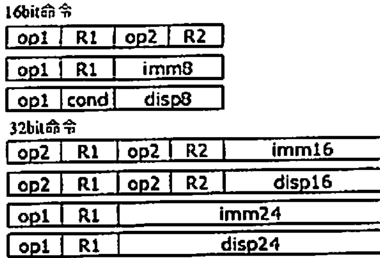


図 7 M32R の命令フォーマット

表 3 M32R に対する拡張命令

opcode		mnemonic	behavior
12	0	padd Rdst,Rsrc	Rdst=Rdst+Rsrc*Rsrc
12	1	psub Rdst,Rsrc	Rdst=Rdst-Rsrc*Rsrc
12	2	mlad Rdst,Rsrc,#simm16	Rdst=Rdst*Rsrc+simm16
12	3	adad Rdst,Rsrc,#simm16	Rdst=Rdst+Rsrc+simm16
12	4	sbsb Rdst,Rsrc,#simm16	Rdst=Rdst-Rsrc-simm16
12	5	adml Rdst,Rsrc,#simm16	Rdst=Rdst+Rsrc*simm16
12	6	sbml Rdst,Rsrc,#simm16	Rdst=Rdst-Rsrc*simm16
12	7	mlpad Rdst,Rsrc,#simm16	Rdst=Rdst*(Rsrc+simm16)
12	8	adrs Rdst,Rsrc,#uimm16	Rdst=(Rdst+Rsrc)≫uimm5
12	9	sbrs Rdst,Rsrc,#uimm16	Rdst=(Rdst-Rsrc)≫uimm5
12	10	adls Rdst,Rsrc,#uimm16	Rdst=(Rdst+Rstc)≪uimm5
12	11	sbls Rdst,Rsrc,#uimm16	Rdst=(Rdst-Rsrc)≪uimm5

Rdst, Rsrc: 汎用レジスタ　simm16: 符号付き 16bit 即値
uimm16: 符号無し 16bit 即値　uimm5: 符号無し 5bit 即値

の M32R プロセッサに対し, 表 3 に示す 12 の命令の追加を行った. 追加命令を記述した入力ファイルは 163 行で, ここから生成された追加マシン記述は GCC 用が 170 行, CGEN 用が 110 行である.

拡張マシン記述から構築したクロス開発環境の正当性を C コンパイラ用テストスイート [4](注6) で検査し, 全てのテストにパスすることを確認した. またこの際, 全ての拡張命令がアセンブリコード中で利用されていることも確認した.

5. む す び

本稿では, 新たなプロセッサが既存のプロセッサの命令セットを拡張することにより設計される場合を想定し, Binutils, GDB のマシン記述を CGEN により生成, GCC への追加命令の記述を define_peephole で行うことにより, 追加命令の記述のみから拡張命令セットに対応したクロス開発環境を自動的に構築する手法を提案した.

本手法自体は特定のプロセッサのアーキテクチャに依存するものではないが, 追加命令の記述, (図 6) において, オペランドやフォーマットの名前 (dr, sr, simm16, OP1, OP2) が既知であることを仮定している. また, コンフィギュラブルプロセッサのソフトウェア開発環境構築インフラストラクチャとしての使用を考えた場合には, レジスタ数の変更やパイプライン変更のような機能が必須であると考えられる. 今後はこのような点への対応を検討していく予定である.

(注6) : http://ist.ksc.kwansei.ac.jp/ ishiura/gcc/ipa/

文 献

[1] 小林悠記, 小林真輔, 坂主圭史, 武内良典, 今井正治: "コンフィギュラブル VLIW プロセッサの HDL 記述生成手法," 情報処理学会論文誌, vol. 45, no. 5, pp. 1311-1321 (2003 年 5 月).

[2] Richard M. Stallman and the GCC Developer Community: *GNU Compiler Collection Internals* (2002).

[3] ルネサステクノロジ: *M32R* ファミリソフトウェアマニュアル (2003).

[4] 内山裕貴, 引地信之, 永松祐二, 石浦菜岐佐: "*C* コンパイラ用テストスイートおよびその生成ツール," 情報処理学会関西支部大会, C-12 (2005 年 10 月).