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.

組込みプロセッサの命令セット拡張に適したソフトウェア開発ツール生成手法

久村　孝寛[†,††]　多賀惣一朗[†††]　石浦菜岐佐[†††]　武内　良典[††]　今井　正治[††]

† 日本電気株式会社　〒216–8666 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
††† 関西学院大学　〒669–1337 兵庫県三田市学園 2-1

あらまし　組込みプロセッサの命令セット拡張およびレジスタの追加に適したソフトウェア開発ツール生成手法を提案する．本手法では，基本となるソフトウェア開発ツールに対してプラグインを追加することによって，ベースプロセッサへ追加された命令セットやレジスタを適切に扱えるソフトウェア開発ツールを生成する．生成可能なソフトウェア開発ツールは，コンパイラ，アセンブラ，逆アセンブラ，命令レベルシミュレータなどである．本手法で生成されるコンパイラでは，追加命令に一対一に対応する組込み関数 (intrinsic function) を用いて記述された C 言語のソースコードに対してコード生成を行う．従来のツール生成手法と比較して，本手法では，GNU ツールチェーンに対応したプロセッサをベースのプロセッサとして使用可能であること，コンパイラを含むツールチェーンを生成可能であること，が特徴である．本手法の有用性を示すために，組込み向けプロセッサである V850 に SIMD 命令を追加する場合について評価を行った．SIMD 命令をアセンブラで直接記述したプログラムと，組込み関数を使って C 言語で記述したプログラムとについて，本手法で生成したソフトウェア開発ツールを用いて実行命令数を比較した．実験の結果，本手法を用いて生成されたコンパイラは，人手によるアセンブラ記述と比較して，実行命令数が平均でわずか 7%程度の増加という良質なコードを生成できることが確認された．

キーワード　アーキテクチャ記述言語，ツール生成，プラグイン

Software Development Tool Generation Method Suitable for Instruction Set Extension of Embedded Processors

Takahiro KUMURA[†,††], Soichiro TAGA[†††], Nagisa ISHIURA[†††], Yoshinori TAKEUCHI[††], and Masaharu IMAI[††]

† NEC Corporation　Shimonumabe 1753, Nakahara-ku, Kawasaki City, 216–8666 Japan
†† Osaka University　Yamadaoka 1–5, Suita City, Osaka, 565–0871 Japan
††† Kwansei Gakuin University　Gakuen 2–1, Mita City, Hyogo, 669–1337 Japan

Abstract　This paper proposes a method of software development tool generation suitable for instruction set extension of existing embedded processors. The key idea in the proposed method is to enhance a base processor's toolchain by adding plugins, which are software components that handle additional instructions and registers. The proposed method can generate a compiler, assembler, disassebler, and instruction set simulator. Generated compilers with the plugins provide intrinsic functions that are translated directly into the new instructions. To demonstrate that the proposed method works effectively, this paper presents an experimental result of the proposed method in the study of adding SIMD instructions to the embedded microprocessor V850. In the experiment, by using intrinsic functions, the compiler generated good code with only 7% increase in the number of instructions against the hand-optimized assembly codes.

Key words　Architecture description language, Tool generation, Plug-in

1. はじめに

マルチメディア処理や無線通信処理などの組込みシステムにおいて、特定用途向けプロセッサ (ASIP, application specific instruction set processor) が広く活用されている．ASIP は，ソフト書き換えにより仕様変更や機能追加に対応できる柔軟性と，ターゲットアプリケーション向けにチューニングされた命令セットによる高い性能と，を兼ね備えている．近年，ASIP は専用の設計ツールを使って開発されるようになってきた．ASIP 設計ツールは，ASIP のハードウェア記述言語 (HDL, hardware description language) のソースコードや，ソフトウェア開発ツールを生成するツールである．ASIP 設計ツールに関しては，これまでに，nML, LISA, EXPRESSION といったアーキテクチャ記述言語 (ADL, architecture description language) を使って，プロセッサのマイクロアーキテクチャや命令セットを検討する研究が数多くある [1]〜[4].

ASIP の活用における重要な課題は C コンパイラなどのソフトウェア開発ツールをアーキテクチャ設計段階の早期に提供することである．優れた C コンパイラを早期に提供可能にするために，多くの ASIP は基本的な命令セットを備えたベースプロセッサとアプリケーション専用の機能ユニットとで構成されている．このような構成の ASIP は Tensilica 社などいくつかの企業から提供されており，組込みシステムの市場で受け入れられている．さらに，ASIP 設計ツールも CoWare 社や Target Compiler 社などからリリースされている．

ASIP のソフトウェア開発ツールの生成に注目すると，従来の ASIP 設計ツールや関連研究の多くはツール全体をスクラッチから生成することに注力し，ベースプロセッサ向けの既存のツールを再利用することや，生成されたツールを人間が改良することを考慮していない．既存ツールを再利用することは，既存の組込みプロセッサを ASIP のベースプロセッサとして使う場合には特に重要である．なぜなら，既存の組込みプロセッサには，最適化された C コンパイラを含むソフトウェア開発ツールがそろっているからである．その開発ツールにはプロファイラなどの機能を備えたシミュレータが含まれるだろう．こうした既存の開発ツールの機能は，従来の ASIP 設計ツールが生成するツールでは必ずしも利用可能になるとは限らない．

ASIP 用ソフトウェア開発ツール生成におけるもう一つの重要な点は，ツール生成のための基礎となるツールチェーンである．従来の ASIP 設計ツールや関連研究の多くは，独自のコンパイラ，アセンブラ，シミュレータなどにもとづいてソフトウェア開発ツールを生成する [1], [3]〜[5], [8]．一方，GNU ツールチェーンにもとづいてソフトウェア開発ツールを生成する手法もいくつか提案されている [6], [7], [9], [11]．GNU ツールチェーンはオープンソースのツールチェーンであり，組込み分野でも広く使われている．GNU ツールチェーンは数多くの組込みプロセッサに対応しているので，既存の組込みプロセッサにもとづいた ASIP のためのソフトウェア開発ツールを生成するのに適している．しかしながら，Xtensa 向けのツール生成を除いて，GNU ツールチェーンにもとづいたこれまでのツール生成手法は，主にアセンブラなどの GNU Binutils に注目し，コンパイラを含む GNU ツールチェーン全体を扱っていなかった．Xtensa 向けのツール生成手法は Xtensa 向けのコンパイラを含む GNU ツールチェーン全体を生成するが，この手法のターゲットプロセッサは Xtensa に限られ，他のプロセッサには対応していない．

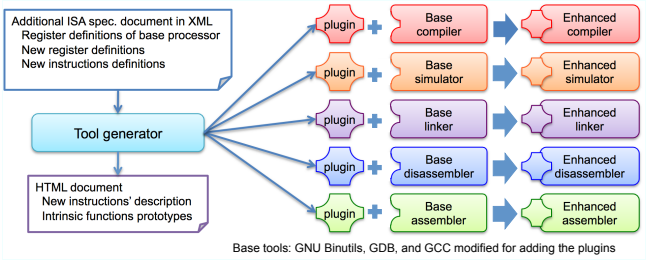

図 1　Concept flow underlying tool generation

そこで，GNU ツールチェーンをもとにした，複数のベースプロセッサに対応可能な新たなツール生成手法を本論文は提案する．本手法は，ベースプロセッサのツールチェーンに対して，プラグインと呼ばれるソフトウェアコンポーネントを追加することにより，追加命令セットや追加レジスタに対応したターゲットプロセッサ向けのソフトウェア開発ツールを生成する．プラグインはベースプロセッサに対する追加命令セットや追加レジスタなどに関する仕様情報にもとづいて生成される．ベースプロセッサのツールチェーンにプラグインを追加するために必要な修正は，プラグインを接続するためのソケットを追加することだけである．本論文では，このツール生成手法に関して，ベースプロセッサへの追加要素の仕様を如何に記述するのか，追加要素を扱うためにどのようなプラグインが生成されるのか，などを中心に述べる．さらに，本手法を使った，SIMD(single instruction multiple data) 命令セットを備えた V850 プロセッサのためのソフトウェア開発ツールを生成する実験についても述べる．

2. プラグインにもとづいたツール生成

我々の目的は，既存の組込みプロセッサをベースプロセッサとして，それに命令セットやレジスタを追加したターゲットプロセッサのために，ソフトウェア開発ツールを効率よく生成する手法を提供することである．この目標を達成するために，プラグインにもとづいたツール生成手法を提案する．プラグインとは，ベースプロセッサ用ツールチェーンの機能を拡張するためのソフトウェアコンポーネントである．例えば，追加命令セットの字句解析，命令エンコード，命令デコードなどの機能をプラグインとして追加する．その結果，既存のツールチェーンを活用して，コンパイラ/アセンブラ/シミュレータなどのターゲットプロセッサ向けのソフトウェア開発ツールを生成することができる．

2.1 ツール生成フロー

提案手法によるツール生成の流れを図 1 に示す．まず，ツール生成は XML(extensible markup language) でターゲットプ

ロセッサの仕様を記述することから始まる．提案手法において，ターゲットプロセッサの命令セットは，変更不可能なベースプロセッサの基本命令セットと，自由に変更が可能な追加命令セットで構成されると仮定する．XML 文書には，ターゲットプロセッサの追加命令セットの情報を記述する．図 2 の命令に対する XML 文書の記述例を図 3 に示す．図 3 の<insn>というタグが図 2 の命令 MYADD の仕様を表す．

この追加命令セットの情報には，ベースプロセッサの名前，ベースプロセッサのレジスタに関する情報，ベースプロセッサへ追加されるレジスタや命令の情報，が含まれる．ただし，XML 文書には，ベースプロセッサの基本命令セットの情報を記述しない．ターゲットプロセッサの設計者はこの XML 文書を作成し，ツールジェネレータに与える．

つづいて，ツールジェネレータは，ベースプロセッサのツールチェーン (ベースツール) へ追加するためのプラグインを生成する．プラグインを生成する前に，ベースツールを予め修正して，プラグインを接続できるようにしておく．予め修正されたベースツールへ生成されたプラグインが結合され，ターゲットプロセッサ用のツールチェーンとなる．生成されたツールチェーンは，ターゲットプロセッサの追加命令のアセンブル，逆アセンブル，リロケーション，実行，などの機能を備える．さらに，生成されたコンパイラでは追加命令に一対一に対応した組込み関数 (intrinsic function) を利用することができる．ただし，本手法で生成されたコンパイラは追加命令を自動的に利用することはない．追加命令をコンパイラに出力させるためには，プログラマが組込み関数を明示的に記述する必要がある．

2.2 生成されたプラグインの構成

図 4 は GNU ツールチェーン向けのプラグイン生成プロセスを表す．ツールジェネレータは，予め用意されたテンプレートへ追加命令や追加レジスタの情報やコード断片を追加することによって，プラグインを生成する．テンプレートファイルは全ての追加命令や追加レジスタに共通して使用可能な関数やデータ構造を含んでいる．共通な関数の例としては，追加命令のエンコード，デコード，アセンブル，逆アセンブルを行う関数がある．これらの関数は追加命令に関する情報にもとづいて動作する．その情報は，XML 文書にもとづいて，データ配列として生成される．そして，そのデータ配列は追加命令に関する一種のデータベースとして使われる．

Binutils や GDB 向けのプラグインに対しては，ツールジェネレータは，アセンブル/逆アセンブル/リンク/命令実行などに必要な命令情報 (ニモニック，シンタックス，命令フィールド構造など) をデータ配列として生成する．さらに，命令動作を表す関数をコード断片として生成する．生成されたデータ配列やコード断片はテンプレートと結合されて Binutils や GDB 向けのプラグインとなる．

一方，GCC 向けのプラグインに対しては，ツールジェネレータは，レジスタに関する各種マクロと，追加命令のマシン記述 (RTL テンプレート) とイントリンジック関数のプロトタイプ定義，などをコード断片として生成する．生成されたコード断片はテンプレートと結合されて GCC 向けのプラグインとなる．

Syntax: `MYADD reg1, reg2, reg3`

Fields:	31 27	26 16	15 11	10 5	4 0
	reg3	opc1	reg2	opc0	reg1

Name	Type	Bits	Description
reg1	GPR	5	Register index of GPR registers
reg2	GPR	5	Register index of GPR registers
reg3	GPR	5	Register index of GPR registers
opc0	opcode	6	Bit pattern to identify the inst.
opc1	opcode	11	Bit pattern to identify the inst.

図 2 Instruction field structure of MYADD

```
 1: <Processor>
 2: <nickname>cpu</nickname>
 3: <register_type length="32">GPR_type</register_type>
 4: <register_bank type="GPR_type" size="32"
 5:                prefix="R" base="true">GPR</register_bank>
 6: <insn_length>32</insn_length>
 7: <insn>
 8:   <mnemonic>MYADD</mnemonic>
 9:   <syntax>MYADD %reg1, %reg2, %reg3</syntax>
10:   <field type="GPR"    length="5">reg1</field>
11:   <field type="opcode" value="0b11_1111"
12:                        length="6">opc0</field>
13:   <field type="GPR"    length="5">reg2</field>
14:   <field type="opcode" value="0b11111001000"
15:                        length="11">opc1</field>
16:   <field type="GPR"    length="5">reg3</field>
17:   <input>
18:     <operand type="GPR" width="32">reg1</operand>
19:     <operand type="GPR" width="32">reg2</operand>
20:   </input>
21:   <output>
22:     <operand type="GPR" width="32">reg3</operand>
23:   </output>
24:   <description>
25:   This instruction calculates the sum of the contents of
26:   registers reg1 and reg2, and stores the sum to register reg3.
27:   </description>
28: </insn>
29: <gdb>
30: <regnum name="R0">0</regnum>
31: ....
32: <regnum name="R31">31</regnum>
33: </gdb>
34: <behavior>cpu-isa.c</behavior>
35: </Processor>
```

図 3 Example of an XML document with additional ISA specification

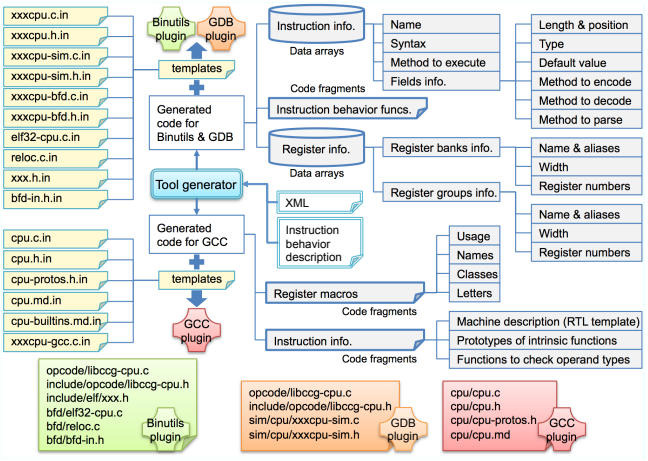

図 4 Generating plugins for GNU Binutils, GDB, and GCC

2.3 ベースツールに対するプラグインの組込み

ターゲットプロセッサ用ツールチェーンにおいて，プラグインをどのように使用するかを図 5 に示す．図 5 は，ターゲットプロセッサ用ツールチェーンにおけるアセンブル/逆アセンブル/命令エンコード/命令デコード/命令実行等の処理を単純化したフローチャートである．ターゲット用ツールチェーンにお

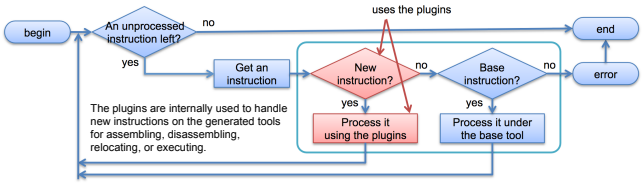

図 5　Tool internal flow for enhanced assemblers, disassemblers, linkers, and simulators working with plugins

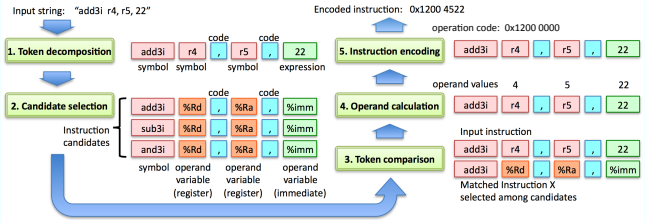

図 6　Assembling instructions on the plugins

```
1: ;; syntax:  myadd reg1, reg2, reg3
2: (define_insn "builtin_cpu_MYADD"
3: [
4: (set
5:   (match_operand:SI 0 "cpu_gpr_regs_operand" "=r")
6:   (unspec:VOID [
7:     (match_operand:SI 1 "cpu_gpr_regs_operand" "r")
8:     (match_operand:SI 2 "cpu_gpr_regs_operand" "r")
9:     ]  UNSPEC_BUILTIN_MYADD))
10: ]
11: ""
12: "myadd %1, %2, %0"
13: [(set_attr "length" "4")]
14: )
```

図 7　A generated instruction pattern

けるこれらの処理において，命令は，ベースツールの処理パスか，プラグインによる処理パス，のいずれかで処理される．まず，ターゲット用ツールは，与えられた命令が追加命令であるか否かを判定する．つづいて，その命令が追加命令であればプラグインによる処理パスでその命令は処理され，そうでなければベースツールの処理パスで処理される．ターゲット用ツールチェーンをこのように動作させるために，ベースツールを予め修正しておく必要がある．

2.4　追加命令のアセンブル方法

ターゲットプロセッサ用アセンブラが追加命令をどのようにアセンブルするかについて説明する．ターゲットプロセッサ用アセンブラに組込まれるパーサプラグインは，ニモニック形式，関数形式，算術形式，の三つの命令記述スタイルに対応している．ニモニック形式は，ニモニックの後に複数のオペランド記述する形式である．関数形式は，C 言語の関数の戻り値を出力オペランドへ代入するように命令を記述する形式である．算術形式は，算術演算子や論理演算子を使って命令を記述する形式である．パーサプラグインの命令処理手順を図 6 に示す．命令をアセンブルした後で，もし未解決のシンボルを含むオペランドがあれば，ターゲットプロセッサ用アセンブラはその未解決シンボルに対するリロケーション情報を作成する．

1. トークン分解:　パーサプラグインは入力文字列をトークンに分解する．トークンは，シンボル，数式，コードのいずれかのトークンに分類される．図 6 では，トークンは 6 個ある．

2. 命令候補選択:　パーサプラグインは入力文字列と同じ数のトークンをもつ命令の候補を選択する．図 6 では，三つの命令が候補として選択されている．

3. トークン比較:　選択された各命令候補について，パーサプラグインは，対応する位置にあるトークンを比較し，入力命令と同じトークンのパターンをもつ命令 X を候補の中から見つける．図 6 では，命令 `add3i` が入力命令と同じトークンのパターンをもつ命令 X である．

4. オペランド計算:　命令 X において`%xxx` というトークンはオペランド変数を意味する．パーサプラグインはオペランド変数を表すトークンから命令 X のオペランドの値を計算する．例えば，オペランド変数を表すトークンがレジスタ名を表すならば，パーサプラグインはそのレジスタ名からレジスタの番号を求める．

5. 命令エンコーディング:　パーサプラグインは，命令 X のオペコードとオペランドの値とを使って，入力命令をエンコードする．

2.5　追加命令のマシン記述

ツールジェネレータは追加命令のマシン記述を生成する．マシン記述とは，追加命令の命令パターンであり，GCC のプロセッサ機種依存部で使用されるものである．命令パターンは，GCC 独自の中間言語である RTL(register transfer language) で記述される．図 7 は生成されたマシン記述の例である．

一般的には，命令パターンには命令の動作が記述される．命令パターンに命令の動作を記述することによって，その命令がどんな演算を実行するのかを GCC に伝えるのである．しかし，ツールジェネレータが生成する追加命令の命令パターンには，意味のある動作を記述せずに，単に入出力のオペランドだけを記述する．このようにすることで，命令パターンの生成が単純化され，さらに追加命令のスケジューリングに最低限必要な入出力オペランドを GCC に認識させることができる．この場合，動作記述のない命令パターンをもつ追加命令を GCC が自動的に使用することはないが，組込み関数を提供するにはそれで十分である．

もしターゲットプロセッサが追加レジスタへの転送命令をもつなら，ツールジェネレータはその転送命令のための命令パターンを生成する．転送命令の命令パターンは転送を意味する動作記述をふくんでおり，GCC がその命令パターンを認識して，コード生成プロセスにおいて使えるようにしている．

2.6　GCC に対する組込み関数の追加

GCC に組込み関数を追加するには，`TARGET_INIT_BUILTINS` と `TARGET_EXPAND_BUILTIN` という二つのマクロが使われる．`TARGET_INIT_BUILTINS` は組込み関数の初期化を行う関数の名前を表す．この初期化関数は，各組込み関数の名前，戻り値，引数，ID 番号，などの情報を GCC に登録する．ツールジェネレータは，この登録処理のステートメントを各組込み関数ごとに生成する．生成されたステートメントは `TARGET_INIT_BUILTINS` で表される関数の中に挿入される．

TARGET_EXPAND_BUILTIN は与えられた組込み関数から命令パターンを生成する関数の名前を表す．この命令パターン生成関数は，与えられた組込み関数に対してどの命令パターンを使うべきかを判断し，組込み関数の引数を命令パターンのオペランドに変換する．ツールジェネレータは，与えられた命令パターンから命令パターンとそのオペランドタイプを得るためのテーブルを生成する．

GCC では，組込み関数の代わりにインラインアセンブラを使うことによって，追加命令などを C コードに記述することもできる．しかし，以下の点で，組込み関数の方がインラインアセンブラよりも優れている．

- 組込み関数に対応する命令のレイテンシや命令語長などの情報が GCC に与えられていれば，その命令を GCC はスケジューリング/最適化できる可能性がある．しかし，インラインアセンブラで記述された命令を GCC が最適化したり効率よくスケージューリングすることはない．
- 組込み関数に対しては，GCC は入出力オペランドの型を検査する．一方，インラインアセンブラに対しては，GCC はそのような検査を行わない．
- インラインアセンブラで記述したコードはターゲットプロセッサでのみで動作するのに対して，組込み関数は非ターゲットプロセッサ上でもエミュレーションを使って動作させられる可能性がある．

こうした理由から，提案手法では，インラインアセンブラではなく，組込み関数を選んだ．

3. 実　　験

ここでは提案手法を用いたツール生成に関する実験について述べる．実験では，組み込み向けマイクロコントローラ V850 [10] をベースプロセッサとして使用する．V850 へ SIMD(single instruction multiple data) 命令セットと 64 ビットレジスタを追加したものをターゲットプロセッサとして，コンパイラ等のソフト開発ツールをツールジェネレータを使って生成する．生成されたコンパイラと SIMD 拡張向けの組込み関数を使って，生成されたコンパイラのコード効率を示す．

ここで，SIMD 拡張を活用する方法として，組込み関数の代わりにインラインアセンブラを使うことはできないことに注意して欲しい．なぜなら，この実験では 64 ビットレジスタを追加するが，それらのレジスタをベースプロセッサ向けのコンパイラで扱うことはできないからである．提案手法が生成するプラグインをベースプロセッサ向けコンパイラへ追加することによってはじめて，64 ビットレジスタを含む SIMD 拡張をコンパイラが活用できるようになる．

ターゲットプロセッサの構成を図 8 と表 8 に示す．V850 に対して追加する SIMD 拡張は，32 個の 64 ビットレジスタと，64 ビット幅のパックドデータを扱う SIMD 演算器と，147 個の SIMD 命令セットからなる．利用可能なデータタイプは 16 ビット ×4, 32 ビット ×2, 64 ビット ×1，の三種類である．SIMD 命令セットは，加算/減算/乗算などの算術演算，型変換，ロード/ストア，論理演算，データインターリーブ，などを含む．

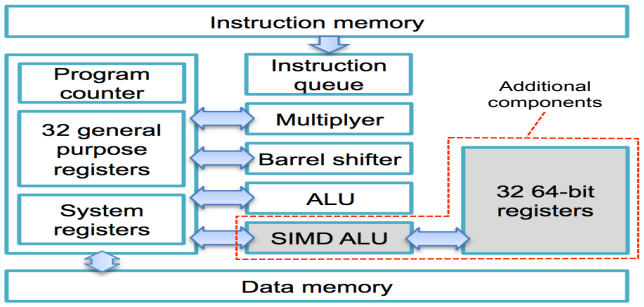

図 8　Block diagram of the V850 processor with SIMD extension

表 1　Architecture summary of the V850 microcontroller with SIMD extension

Base architecture	RISC processor for emmbedded systems. Harvard architecture. Single cycle instruction execution. Compact code size allowed by 2-byte insts. 32 32-bit general purpose registers
SIMD extension	32 64-bit registers Data types: 16 bits×4, 32 bits×2, 64 bits×1. Packed arithmetic insts. Load/store insts. Data type conversion. Logical operation.

表 2　The code amount of the generated plugins for the SIMD extension.

Base tools	GNU Binutils 2.17 GDB 6.6 GCC 3.4.6
# of instructions	179
# of lines of the input XML file	5934
# of lines of the input behavior description	4436
# of lines of the assembler plugin	8903
# of lines of the simulator plugin	7699
# of lines of the compiler code fragments	13496

3.1　SIMD 拡張を備えた V850 プロセッサ向けに生成されたツールチェーン

このような SIMD 拡張の命令セットとレジスタに関する仕様を XML で記述し，ツールジェネレータを使って，コンパイラ等のソフトウェア開発ツールを生成した．ツールジェネレータが生成したプラグインの規模を表 2 に示す．ベースプロセッサ V850 へ追加した SIMD 命令の数は 179 個で，ツールジェネレータへの入力ファイル (XML と動作記述の C コード) は合計で 10370 行である．一方，ツールジェネレータから生成されたプラグインは合計で 30098 行である．入力ファイルの規模は出力ファイルよりも非常に小さいわけではないが，ひとつの仕様記述からアセンブラ/シミュレータ/コンパイラ等のツールチェーンが生成されることは非常にメリットがある．

3.2　組込み関数を使ったコード生成

本実験で生成されたコンパイラのコード生成効率を調べるために，(1) 組込み関数を使ってコンパイラが生成したコードと，(2) 手書きアセンブラコードと，を比較する．両方とも SIMD 命令セットを使用するが，(1) では C コードの中に SIMD 命令セットに一対一に対応する組込み関数を記述し，(2) ではアセンブラコードの中に SIMD 命令セットを記述する．フィルタ，FFT などの基本的な信号処理とソートなどの 28 個のプログラムについて，二種類のプログラムを作成し，シミュレータを使って，関数内で実行された命令の数を計測した．

図 9 は手書きアセンブラコードに対するコンパイラ生成コードの命令数の比率を表す図である．図 9 によると，手書きアセ

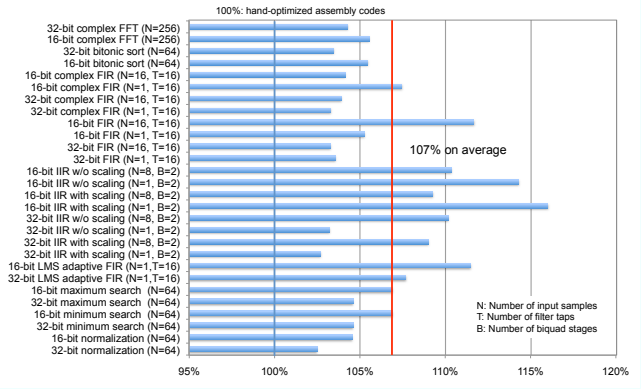

図 9　Increase in the number of executed instructions of compiler-generated codes using intrinsic functions against that of hand-optimized assembly codes

ンブラコードに比べてコンパイラ生成コードの命令数は平均で7%増加する．手書きアセンブラコードに対するコンパイラ生成コードの明らかなメリットは，レジスタ割当てをコンパイラに任せられることである．つまり，組込み関数を使用する場合には，手書きアセンブラコードに対して 7%程度の命令数増加を許容することで，プログラマがレジスタ割当てから解放される．ソースコードのメンテナンス性を高めるという点で，7%増加は十分に許容できると考える．この平均 7%の命令数増加の要因は以下のとおりである．

ループ回数検査　コンパイラ生成コードでは，ループ突入前にループ回数が変数である場合に 0 か否か必ずチェックする．手書きアセンブラコードではチェックしない．

冗長なレジスタ間転送　コンパイラのコードには，引数や戻り値に関する冗長な転送命令がある．引数，戻り値，ローカル変数に対するレジスタ割当てが別々に行われることが原因．回避困難と考えられる．

4. 関連研究と考察

GNU Binutils の retargetting に関する関連研究と，本提案とを比較し，本提案の位置付けについて議論する．Tensilica は 1990 年代後半にカスタマイズ可能なプロセッサ Xtensa [12] を開発し，Xtensa 向けのソフト開発ツールを生成するシステムを提供した．ターゲットアーキテクチャは Xtensa に限られるが，レジスタ数や演算器や命令セットをカスタマイズ可能であり，カスタマイズした Xtensa のための GNU Binutils，GDB，シミュレータ，コンパイラを生成できる．文献 [12] によると，残念ながら新たなリロケーションの追加については記載がない．

Red Hat 社は 2000 年に，GNU Biutils や GDB の移植を容易にするためのツール CGEN [6] をオープンソースソフトウェアとしてリリースした．CGEN は GNU Biutils 全体をリターゲッティングするものではなく，命令のエンコードやデコードを行なうライブラリ opcode や命令実行の関数群だけをプロセッサ仕様記述から生成する．

Abbaspour は 2002 年に，仕様記述にもとづいて GNU Binutils を生成するツール rbinutils を開発した [7]．しかしながら，rbinutils が生成可能なのは GNU Binutils だけで，シミュレータやコンパイラを生成することはできなかった．Baldassin は 2003 年から，仕様記述にもとづいて GNU Binutils やシミュレータを生成するツール ArchC を開発し，オープンソースソフトとして提供している．文献 [9] において、ArchC を使って複数の CPU で GNU Binutils のリターゲッティングとシミュレータ生成が可能であることが報告された．

一方，提案手法は，Xtensa 用ツールジェネレータとほぼ等価な機能を，一般的な GNU ツールチェーンに対応するプロセッサに提供するものである．さらに，本提案は追加命令のための新たなリロケーションタイプを追加したり，追加命令と等価な演算を行なうエミュレーション関数を追加したり，することができる．本提案以外はニモニック形式の命令シンタックスだけが使用可能であるが，本提案はニモニック形式だけでなく，関数形式や算術形式の命令シンタックスを使用することができる．実験には詳細を記述しなかったが，V850 プロセッサの他に，ARM および MIPS をベースプロセッサとする場合についても、数個の命令を追加するような簡易な実験によって、ツール生成が可能であることも確認している．

5. おわりに

組込みプロセッサの命令セット拡張およびレジスタの追加に適したソフトウェア開発ツール生成手法を提案した．本手法では，基本となるソフトウェア開発ツールに対してプラグインを追加することによって，ベースプロセッサへ追加された命令セットやレジスタを適切に扱えるソフトウェア開発ツール (アセンブラ，逆アセンブラ，リンカ，シミュレータ，コンパイラ) を生成する．組込み向けプロセッサ V850 に SIMD 命令を追加する実験をとおして，本手法により，SIMD 拡張を備えた V850 向けのツールチェーンを生成可能であることを示した．さらに，本手法を用いて生成されたコンパイラは，手書きアセンブラコードと比較して，実行命令数が平均でわずか 7%程度の増加という良質なコードを生成できることが確認された．

文　献

[1] A. Fauth, J. Van Praet, and M. Freericks, “Describing Instruction Set Processors Using nML,” Proceedings of the European Design and Test Conference, pp. 503-507, March 1995.

[2] Paul C. Clements, “A Survey of Architecture Description Languages,” Proceedings of the 8th International Workshop on Software Specification and Design, pp. 16, March 22-23, 1996

[3] Vojin Zivojnovic, Stefan Pees, and Heinrich Meyr, “LISA-machine description language and generic machine model for HW/SW co-design,” Proceedings of the IEEE Workshop on VLSI Signal Processing IX, pp. 127-136, 1996.

[4] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau, “EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability,” Proceedings of Design Automation and Test in Europe (DATE 99), pp. 485-490, 1999.

[5] S. Kobayashi, Y. Takeuchi, A. Kitajima, M. Imai, “Compiler Generation in PEAS-III: an ASIP Development System,” International Workshop on Software and Compilers for Embedded Processors (SCOPES), 2001.

[6] CGEN, the Cpu tools GENerator, see http://sources.redhat.com/cgen/ .

[7] Maghsoud Abbaspour, Jianwen Zhu, “Retargetable Binary Utilities,” Proceedings of the 39th Design Automation Conference, pp. 331-336, 2002.

[8] Manuel Hohenauer, Hanno Scharwaechter, Kingshuk Karuri, Oliver Wahlen, Tim Kogel, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Gunnar Braun, and Hans van Someren, “A Methodology and Tool Suite for C Compiler Generation from ADL Processor Models,” Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'04), 2004.

[9] Alexandro Baldassin, Paulo Centoducatte, and Sandro Rigo, “An Open-Source Binary Utility Generator,” ACM Transactions on Design Automation of Electronic Systems, Vol. 13, No. 2, Article 27, April 2008.

[10] NEC Electronics Corp., “User's Manual: V850 Family for Architectur,” Document No. U10243EJ7V0UM00, March 2001, available at http://www.necel.com/ .

[11] Ricardo E. Gonzalez, “Xtensa: A Configurable and Extensible Processor,” IEEE Micro, Volume 20, Issue 2, pp. 60-70, March-April 2000.

[12] Tensilica, Inc., “Tensilica Instruction Extension (TIE) Language Reference Manual,” Nov. 2006.