情報処理学会論文誌 Vol.57 No.8 1718–1736 (Aug. 2016)

命令セットシミュレータ生成フレームワークの設計と実装

奥田 勝己[1,a) 竹山 治彦[1]

受付日 2015年11月19日, 採録日 2016年5月17日

概要：命令セットシミュレータ（ISS：Instruction Set Simulator）は，組込みシステムの開発に不可欠なソフトウェアである．組込みシステムの開発では，新規アーキテクチャのプロセッサや専用プロセッサを含む多種多様なプロセッサが採用される．このため，組込みシステムの開発に ISS を適用するためには，ISS の効率的な開発手法が必要である．そこで，本論文では，ISS の開発効率化を目的とした ISS 生成フレームワークを提案する．本生成フレームワークでは，模擬対象プロセッサごとに作成された命令属性記述と命令動作記述を主な入力とし，ISS を自動生成することで，ISS の開発に必要なコード記述量を削減する．本生成フレームワークを用いて ARM，MIPS64，SH に対応した ISS を構築し，ハンドコーディングの ISS との比較でコード記述量を 40%削減できることを確認した．

キーワード：命令セットシミュレータ，ISS，エミュレータ，エミュレーション

Design and Implementation of Generation Framework for Instruction Set Simulators

Katsumi Okuda[1,a) Haruhiko Takeyama[1]

Received: November 19, 2015, Accepted: May 17, 2016

Abstract: Instruction set simulators (ISSs) are indispensable tools for the development of embedded systems. Various kinds of processors, including new architecture processors or specific purpose processors, are adopted in embedded systems. Hence, an efficient way to develop ISSs is indispensable in order to use them in the development of embedded systems. In this paper, we propose a generation framework for ISSs aimed at improvement of development efficiency. The framework is able to reduce the effort in coding ISSs by automatically generating them from the attribute and behavior descriptions of instructions. Our results on ARM, MIPS64, and SH show that the description amount is 40% smaller than for a hand-coded ISS.

Keywords: instruction set simulator, ISS, emulator, emulation

1. はじめに

ISS は，ホスト計算機上でターゲットプロセッサを模擬するソフトウェアである．近年，組込みシステムの開発環境として，ISS を組み込んだ仮想環境の適用が進んでいる．組込みシステムの開発に仮想環境を適用することで，ハードウェアとソフトウェアを同時に開発することができるため，開発期間を短縮することが可能となる．また，ISS 上では，任意の実行制御や故障注入など，実機では困難な操作を容易に実現できる．このため，組込みシステム開発者は，ISS を組み込んだ仮想環境を利用することで，効率的にソフトウェアを試験することが可能である．ISS を用いた試験の事例としては，ISS の実行制御機構を用いたマルチプロセッサ RTOS のテスト効率化手法 [1] などが報告されている．

しかし，組込みシステムの開発に ISS を適用するためには，ISS を開発するためのコストと期間が問題となる．組込み向けプロセッサの種類は豊富であるため，製品開発では機種ごとに異なるプロセッサが採用される場合がある．また，製品によっては，新しいアーキテクチャのプロセッサや専用プロセッサも適用される．さらには，開発の途中でプロセッサが変更となる場合もある．このため，組込み

[1] 三菱電機株式会社先端技術総合研究所
Advanced Technology R&D Center, Mitsubishi Electric Corporation, Amagasaki, Hyogo 661–8661, Japan
a) Okuda.Katsumi@eb.MitsubishiElectric.co.jp

© 2016 Information Processing Society of Japan

システムの開発にISSを適用するためには，異なる命令セット向けのISSを低コストかつ短期間で準備するための効率的な開発手法が要求される．

ISSの開発を効率化する従来技術としてISSを生成可能なプロセッサ記述言語 [2], [3], [4], [5], [6], [7], [8] が存在する．ISS開発者は，命令ごとの動作やフォーマット情報をプロセッサ記述言語で記述することでISSを構築できる．しかし，命令動作を記述するためにISS開発者がプロセッサ記述言語を習得する必要があるため，実開発への適用が困難であるという課題があった．そこで，本論文では，汎用プログラミング言語で命令の動作を記述可能なISS生成フレームワークを提案する．本生成フレームワークでは，簡素な命令属性記述とC++言語による命令動作記述を主な入力とし，インタプリタ方式のISSを生成する．本生成フレームワークでは，命令属性記述から命令デコード処理，命令フィールド抽出処理，プログラムカウンタ更新処理を自動生成するため，命令動作記述中ではこれらの処理記述を省略することができる．本論文の貢献は下記のとおりである．

- 複数の命令セットに対応可能な命令属性記述エントリを設計した．ISS生成フレームワークは，当該エントリの情報を用いて命令デコード処理，命令フィールド抽出処理，遅延スロット操作をともなうプログラムカウンタ更新処理を自動生成できる．
- 命令動作記述と被生成コードが連携するための記述ガイドラインを規定した．記述ガイドラインに従うことで，命令動作記述では命令フィールド抽出処理やプログラムカウンタ更新処理を記述する必要がない．
- C++言語による命令動作記述と連携可能なコードを生成するISS生成フレームワークを実装し，ARM，MIPS64，SH向けのISSを生成できることを確認した．ARMを対象とした実験では，ハンドコーディングのISSと比較し，40%少ない記述量かつ95%小さい平均サイクロマティック複雑度のコードからDMIPS値が2倍のISSを生成できることを確認した．

本論文の構成を以下に示す．まず，2章において関連研究について示し，本研究の位置付けを明確化する．3章では，本生成フレームワークの全体構成を示す．4章では本生成フレームワークの主な入力である命令属性記述と命令動作記述の詳細を示す．5章では，本生成フレームワークの実装について示す．6章では，本生成フレームワークを用いて構築したARM，MIPS64，SH向けのISSを用いた実験結果について示す．最後に8章では，まとめと今後の課題について示す．

2. 関連研究

2.1 ISS生成手法

ISSの開発手法に関する既存研究としては，プロセッサ記述言語からISSを生成する手法が提案されている．ISSを生成可能なプロセッサ記述言語としては，nML [2]，ISDL [3]，LISA [4]，EXPRESSION [5], [6]，ASIP Meister [7]，Harmless [8] が存在する．従来のプロセッサ記述言語を用いたISSの開発手法では，ISS開発者が専用のプロセッサ記述言語を習得する必要があった．これに対し，本生成フレームワークでは，入力とする命令動作記述，命令属性記述，資源記述のうち，命令動作記述と資源記述についてC++を用いることができる．このため，ISSの開発者が命令の動作とプロセッサの資源を記述するために専用言語を習得する必要がない．また，本生成フレームワークの命令属性記述では，CSVライクな文法を採用している．このため，本生成フレームワークでは，既存のプロセッサ記述 [2], [3], [4], [5], [6], [7], [8] と比較し，同じ情報を記述するためにISS開発者が習得すべき事柄が少ない．たとえば，既存プロセッサ記述言語 [2], [3], [4], [5], [6], [7], [8] の場合，命令フォーマットの情報を記述するためには，専用のキーワードを用いた構文を習得する必要がある．一方，本生成フレームワークでは，専用のキーワードを習得することなく命令フォーマットを記述できる．

文献 [9] では，汎用プログラミング言語であるC言語で命令の動作を記述することでISSを生成する手法が提案されている．文献 [9] で生成対象とするISSが静的コンパイル方式のISSであるのに対し，本研究で生成対象とするISSの実行方式はインタプリタ方式である．

ISSは，ディスアセンブラやデバッガなどのソフトウェアと同様に命令デコーダを含む．命令デコーダに関する既存研究としては，命令デコーダ生成手法の提案が存在する [10], [11], [12], [13], [14], [15]．本生成フレームワークでは，命令デコード関数の生成に文献 [14], [15] のアルゴリズムを使用している．

2.2 ISS実現方式

ISSの実現方式は，インタプリタ方式とコンパイル方式に大別される．本研究で生成対象とするISSは，インタプリタ方式である．インタプリタ方式のISSに関する既存研究としては，SimCore [16]，SimMips [17]，文献 [18]，GDB [19] が存在する．SimCoreとSimMipsは，それぞれAlphaとMIPSを対象としたISSであり，C++によるシンプルな記述で可読性を高めている点を特徴とする．SimCoreは，デコード結果を再利用するという点において，本生成フレームワークが生成するISSと類似している．しかし，SimCoreやSimMipsは複数の命令セットへの対応は考慮されておらず，ISSの処理全体がそれぞれの命令セット向けに記述されている．一方，本生成フレームワークでは，命令セット依存部のみを記述することで，個々の命令セットに対応したISSを生成することができる．文献 [18] では，インタプリタ方式ISSの高速化手法が提案されている．文

© 2016 Information Processing Society of Japan

献 [18] の改良 function 方式は，命令ごとに命令動作関数を用意する点で本生成フレームワークと同様である．しかし，改良 function 方式の命令動作関数は，命令語を引数とし，命令動作関数内で命令語からの命令フィールドの抽出やプログラムカウンタの更新を行う必要がある．一方，本生成フレームワークでは，命令動作関数は命令語から抽出済みの命令フィールド配列を引数とする．このため，本生成フレームワークでの命令動作関数は，命令フィールドの抽出処理を記述する必要がない．また，命令動作関数中でプログラムカウンタを操作する必要もない．本生成フレームワークでは，被生成コードが命令フィールドの抽出やプログラムカウンタの更新を行う．GDB [19] は，デバッガであるが，実機の代用となる ISS を構成要素として含む．GDB の ISS は，複数の命令セットに対応しているが，命令セットごとに異なる開発手法で開発されている．たとえば，ARM 向け ISS は，C 言語のみで実装されているのに対し，MIPS64 や SH 向け ISS は，専用記述言語を入力とするコード生成ツールで実装されている．さらに，SH 向け ISS は，MIPS64 向け ISS とは異なる専用のコード生成ツールで実装されている．これに対し，本生成フレームワークでは，ARM，MIPS64，SH 向けの ISS を単一の仕組みで生成することができる．

コンパイル方式は，模擬対象命令セットのバイナリ命令列をホストネイティブの命令列や命令模擬関数の呼び出し列に変換する方式である．コンパイル方式は，プログラム実行開始前に変換する静的コンパイルと実行時に変換する動的コンパイルに大別される．文献 [9], [20], [21] では，静的コンパイル方式のシミュレーションについての研究が報告されている．静的コンパイル方式の ISS は，プログラム実行時におけるプログラムの書き換えを模擬することができないことが問題である．たとえば，静的コンパイル方式では，JIT（Just In Time）方式の VM（Virtual Machine）を実行することや，組込みシステムで想定される動的なソフトウェア更新処理を実行することができない．一方，本研究で対象とするインタプリタ方式の ISS では，実行時にプログラムが書き換えられた場合であっても，再利用するために保持しておいたデコード結果を廃棄することで対応可能である．

動的コンパイル方式は，プログラム実行時に模擬対象命令列をホスト計算機の命令列に変換してから実行する方式である．動的コンパイル方式の ISS としては，Shade [22]，Embra [23]，ESPRIT/sim [24]，QEMU [25] があげられる．動的コンパイル方式の Shade や Embra では，模擬対象命令セットおよびホスト計算機の移植性について考慮がなされていない．これに対し，QEMU や ESPRIT/sim では，命令セット非依存な中間コードへの変換や命令セット非依存なマクロを用いた変換によって，移植性を向上させている．しかし，動的コンパイル方式の ISS では，模擬対象命令ごとの変換処理をハンドコーディングする必要がある．命令の変換処理は複雑であるため，一般にコンパイル方式の ISS の開発には，インタプリタ方式の ISS を開発する場合と比べて多大な労力を要する．

3. ISS 生成フレームワーク概要

3.1 ISS 生成フレームワークの構成

ISS 生成フレームワークは，共通コードを含む命令セット非依存部と命令セット依存部を生成するコード生成ツールから構成される．ISS の開発者は，ISS 生成フレームワークを用いることにより，模擬対象命令セットごとに命令セット依存部を記述するのみで，ISS を開発できる．ISS 生成フレームワークの概要を図 **1** に示す．

命令セット非依存部は，共通コードとメモリモデルなどから構成される．共通コードは，C++で記述されており，複数種類のプロセッサで共通の振舞いとデータを定義した Core クラスで定義される．本生成フレームワークでは，命令セットごとの ISS を Core クラスを継承したサブクラスとして実装する．また，Core クラスでは，命令セット共通処理を実現するためにプログラムカウンタ関連の変数を予約しており，サブクラスからアクセス可能なメンバ変数として保持する．

命令セット依存部は，模擬対象命令セットごとに記述が必要となるファイルである．命令セット依存部は，下記のファイルから構成される．

資源記述 Core クラスを継承したクラスの定義を含む．本サブクラスは，ステータスレジスタ，システムレジスタなどのプロセッサ資源をメンバ変数として持つ．プログラムカウンタは Core クラスで定義されるため，資源記述ではプログラムカウンタの定義を省略する．

命令属性記述 命令ごとの命令フォーマットや性質に関する情報を保持する．CSV（Comma-Separated Values）ライクな文法で記述される．

命令動作記述 命令ごとの動作を命令動作関数として記述する．C++の文法で記述される．命令動作関数は，資

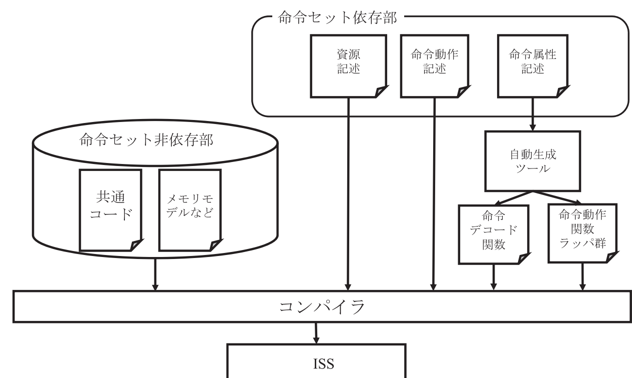

図 **1** ISS 生成フレームワーク
Fig. 1 Generation framework for ISS.

© 2016 Information Processing Society of Japan

源記述ファイルで定義したクラスのメンバ関数である．

命令属性記述と命令動作記述の詳細については，それぞれ 4.1 節と 4.2 節で示す．

ISS 生成処理は，命令属性記述を入力とし，ISS の処理を実現するために必要なコードを生成することで，資源記述で定義されたサブクラスを補完実装する．ISS 生成処理が生成するコードは下記のとおりである．

- 命令デコード関数
- 命令動作関数ラッパ群

命令デコード関数と命令動作関数ラッパ群は，資源記述で定義されたサブクラスのメンバである．命令デコード関数は，命令のフェッチとデコードを行い，命令動作関数ラッパは，命令動作関数の機能を補完する役割を担う．命令デコード関数と命令動作関数ラッパの詳細については，5 章で示す．

本生成フレームワークでは，命令動作記述，資源記述，命令セット非依存部に加えて被生成コードを C++コンパイラでビルドすることで，ホスト計算機上で実行可能な ISS を得る．

3.2 生成対象 ISS

本生成フレームワークが生成する ISS は，機能レベルのインタプリタ方式 ISS である．機能レベルの ISS は，パイプラインレベルでサイクル精度のシミュレーションを行うのではなく，命令単位のシミュレーションを行う．したがって，サイクル精度のシミュレータと比較して高速に動作することを特徴とする．また，インタプリタ方式は，実行対象プログラムを 1 命令ごとに解釈実行する方式である．

本生成フレームワークが生成する ISS の処理フローを**図 2** に示す．本 ISS の主な処理は，命令の実行を繰り返すループ処理である．図 2 の外側のループでは，終了条件を満たすまで命令の実行を繰り返す．本 ISS では，プログラムカウンタがあらかじめ設定した特定の値になることを終了条件としている．

本 ISS では命令の実行を以下の 2 ステップで行う．

- 命令デコード
- 命令実行

命令デコードステップでは，模擬プログラムカウンタのアドレスに存在する命令を識別し，命令フィールドの抽出までを行う．次に，命令実行ステップでは，命令デコードステップで識別子した命令を解釈実行する．

実行対象プログラムが不変の場合，命令デコードステップの実行結果は，プログラムカウンタの値が同じであれば不変である．そこで，本 ISS では，1 度デコードした結果を再利用できるようにホスト計算機上のメモリに保持する．本論文では，デコード結果を保持するホスト計算機上のメモリ領域をデコード結果キャッシュと呼ぶ．

本 ISS では，毎回命令デコードを行うのではなく，はじめにデコード結果キャッシュを参照し，プログラムカウンタが指す命令がデコード済みか否かを確認する．デコード済みの場合，デコード結果キャッシュのデコード結果を利用し，それ以外の場合，命令デコードを行う．このとき ISS は，デコード結果をデコード結果キャッシュに格納する．本 ISS では，デコード結果キャッシュを利用することで，命令実行ごとの命令デコードステップを不要化している．

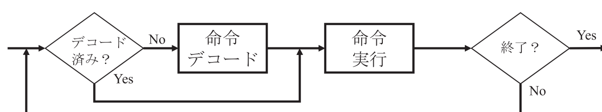

図 2 ISS の処理フロー

Fig. 2 Processing flow of ISS.

4. 入力記述

4.1 命令属性記述

コード生成ツールの入力となる命令属性記述ファイルは，対象とするプロセッサごとに存在し，命令フォーマットと命令の性質に関する情報を保持する．ここで，命令フォーマットとは命令のビットパターンと命令フィールドの名前，位置，サイズである．命令属性記述ファイルは，下記項目を区切り文字で連結したエントリから構成される．

(1) 命令名
(2) ビットパターン
(3) 命令フィールドの名前，位置，サイズ
(4) 除外条件
(5) 分岐命令か否か
(6) 条件分岐か否か
(7) Likely 命令か否か
(8) 遅延スロット数

上記のエントリは命令ごとに 1 つ存在する．(1)–(5) はすべての命令で必須の項目であり，(6)–(8) は，分岐命令の場合のみ必要な項目である．

(1) の命令名は命令ごとに一意な識別子である．本生成フレームワークは，命令ごとに必要な関数の名前を生成するために当該識別子を使用する．(2) のビットパターンは，命令デコード関数生成に必要な項目であり，命令固有の定数，すなわちオペコードの位置と値に関する情報を保持する．(3) の命令フィールド情報は，オペランドを指定する命令フィールドごとの名前と命令ビット列中における位置とサイズを保持する．(4) の除外条件 [14], [15] は，命令がとりえないビットパターンの条件を保持する．除外条件が存在しない場合，本項目は空である．(5) の分岐命令か否かは，エントリと対応する命令が分岐命令か否かを示す真偽値を保持する．

(6)–(8) の項目は，分岐命令に対するエントリの場合にのみ有効な項目である．(6) の条件分岐か否かは，命令が条件分岐命令か否かの真偽値を保持する．(7) の Likely 命

© 2016 Information Processing Society of Japan

表 1　命令属性記述エントリ例
Table 1　Example of entries in instruction attribute description.

	DADD 命令（MIPS64）	BEQL 命令（MIPS64）	BL 命令（ARM）
命令名	DADD	BEQL	BL
ビットパターン	000000xxxxxxxxxxxxxxx00000101100	010100xxxxxxxxxxxxxxxxxxxxxxxxxx	xxxx1011xxxxxxxxxxxxxxxxxxxxxxxx
命令フィールド	rs[25:21], rt[20:16], rd[15:11]	rs[25:21], rt[20:16], offset[15:0]	cond[31:28], imm24[23:0]
除外条件	-	-	cond = 1111
分岐命令	false	true	true
条件分岐	-	true	true
Likely 命令	-	true	false
遅延スロット数	-	1	0

令か否かは，(6) の条件分岐か否かが真の場合にのみ有効な項目である．(7) が真の場合，命令が Likely 命令であることを表す．Likely 命令とは，条件分岐命令のうち，分岐条件成立の場合にのみ遅延スロットを実行する命令のことである．(8) の遅延スロット数は，分岐命令の遅延スロット数を示す．MIPS，SH，SPARC などのプロセッサや DSP（digital signal processor）は，分岐命令実行後，分岐先の命令を実行する前に，遅延スロットの数だけ後続の命令を実行する．遅延スロットを持たない分岐命令の場合，遅延スロット数は 0 となる．

本生成フレームワークでは，プログラムカウンタを変更する命令を分岐命令として扱う．したがって，プログラムカウンタが汎用レジスタの 1 つである ARM のような命令セットの場合，汎用レジスタを更新するすべての命令が分岐命令となる．また，ARM では大部分の命令が条件付きで実行される．汎用レジスタを更新する命令が条件付きであれば，当該命令を条件分岐命令として扱う必要がある．

命令属性記述のエントリが含む情報の例として，MIPS64 の DADD 命令および BEQL 命令と ARM の BL 命令のエントリが含む情報を**表 1** に示す．ビットパターン中の各文字は命令語の各ビットと一致する．ビットパターン文字列中の各文字は，対応する命令語のビットが固定の値を持つ場合 0 または 1 であり，任意の値を持つ場合 x である．BEQL 命令のビットパターンは，最下位ビットから数えて 26 ビット目から 31 ビット目が固定の値として 2 進数表記の 010100 を持ち，それ以外のビットが任意の値を持つことを示す．命令フィールド情報は，DADD 命令の場合，命令フィールドとして rs，rt，rd が存在し，それぞれが命令語中で最下位ビットから数えて 21 ビット目から 25 ビット目，16 ビット目から 20 ビット目，11 ビット目から 15 ビット目の位置に存在することを示す．DADD 命令の場合，分岐命令でないため，分岐命令の項目は false となる．したがって，条件分岐，Likely 命令，遅延スロット数の項目は無効である．一方，BEQL 命令は分岐命令であるため，分岐命令の項目が true である．また，BEQL 命令は Likely 命令であるため，条件分岐の項目が true，Likely 命令の項目が true となる．BEQL 命令の遅延スロット数は 1

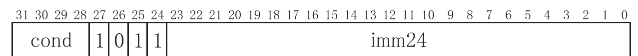

図 3　BL 命令（ARM）のエンコーディング図
Fig. 3　Encoding diagram of the BL instruction (ARM).

である．BL 命令は，除外条件を持つ命令である．ビット列が 28 ビット目から 31 ビット目の命令フィールド cond の全ビットが 1 の場合，当該ビット列は BL 命令のビットパターンと一致していても BL 命令ではないことを意味する．BL 命令は分岐命令であるが，BEQL 命令と異なり遅延スロットを持たない．このため，BL 命令の遅延スロット数は 0 である．

なお，命令属性記述で記述量が多い項目はビットパターンと命令フィールドである．しかし，ISS 開発者はプロセッサのマニュアルを参照することで，これらを容易に記述できる．これは，一般にプロセッサのマニュアルには，命令属性記述に近い形式でビットパターンと命令フィールドの情報が記載されるためである．例として，ARMv7 命令セットのマニュアル [26] の p.344 に掲載されている BL 命令の命令エンコーディング図を**図 3** に引用する．表 1 の BL 命令のパターン xxxx1011xxxxxxxxxxxxxxxxxxxxxxxx の数値部分 1011 は図 3 に記述されている数を転記することで記述可能である．同様に，命令フィールド cond[31:28] も図 3 の文字列 cond とその左上と右上にそれぞれ記述されている数 31 と 28 を転記することで記述可能である．プロセッサのマニュアルごとに若干の記法の差異は存在するが，ARM 以外の命令セットのビットパターンと命令フィールドもおおむね同じ要領で記述できる．

4.2　命令動作記述

本節では，本生成フレームワークを用いて ISS を開発する際に必要となる命令動作記述について示す．本生成フレームワークでは命令動作記述についての記述ガイドラインを規定することで，被生成コードと命令動作記述中のコードの連携動作を実現している．

命令動作記述ファイルには命令ごとの命令動作関数を定義する．各命令動作関数では，命令語中の即値やフラグ，

© 2016 Information Processing Society of Japan

レジスタ，メモリを入力オペランドとして演算し，結果をレジスタやメモリに反映する処理が記述される．一般にハンドコーディングでISSを開発する場合，上記の処理に加えて命令フィールドの抽出，プログラムカウンタの更新，遅延スロットの操作などを記述する必要がある．しかし，本生成フレームワークでは，被生成コードがこれらの処理を行うため，これらの処理を記述する必要がない．結果として，簡素なコードで命令の動作を記述することが可能である．

本生成フレームワークでは，命令動作関数が被生成コードと連携して動作できるように，命令動作関数の記述に以下のガイドラインを定めている．

- 命令動作関数のシグネチャ
- プログラムカウンタ関連の操作

以下本節では，命令動作関数のシグネチャに関する規則とプログラムカウンタ関連の操作に関する規則を順に示す．

4.2.1　命令動作関数のシグネチャ

命令動作関数は，命令属性記述のエントリごとに1つ存在する．命令動作関数のシグネチャの記述ガイドラインを以下に示す．

関数名　関数名は対象命令と対応する命令属性記述エントリに記された命令名と一致すること．

引数　命令動作関数は命令フィールドを引数リストとして持つ．引数の数と順序は，命令属性記述エントリの命令フィールドの数と順序と一致すること．

上記規則は，被生成コードから命令動作関数を呼び出すためのI/Fを規定している．命令動作関数には，命令語から抽出済みの命令語が引数として渡される．したがって，ISS開発者は命令フィールドの抽出処理を記述する必要がない．命令フィールドの抽出は，ビット操作をともない，ハンドコーディングのISSを複雑化している要因の1つである．本生成フレームワークでは，命令フィールドの抽出を記述不要とすることで，ISSのコードを簡素化している．

命令動作関数の引数が多い場合，すなわち命令が多数の命令フィールドを持つ場合，人手で引数リストを記述すると引数リストの記述順序を誤ることが懸念される．記述順序に誤りがある場合，ISSは命令フィールドを正しく解釈できないため，正常に動作しない．

そこで，本生成フレームワークでは，関数シグネチャの記述をサポートするためのヘルパマクロ `DEFINST` を命令属性記述から自動生成し，命令動作記述用に提供している．ヘルパマクロ `DEFINST` は命令名を引数とし，命令動作関数のシグネチャに展開される．ISSの開発者は，被生成マクロファイルをインクルードし，`DEFINST` を用いることで，容易にフレームワークが規定するシグネチャを記述することができる．また，ヘルパマクロが生成する関数シグネチャの引数名は対応する命令フィールドの名前と一致している．このため，関数ボディでは，命令属性記述中の

```
1  #define DEF_DADD void MIPS64Core::DADD(
       uint32_t rs, uint32_t rt, uint32_t rd)
2  #define DEFINST(x) inline DEF_##x
```

図4　ヘルパマクロの例
Fig. 4　Example of helper macros.

表2　予約変数一覧
Table 2　Reserved variables.

変数	説明
`m_NextPC`	次のプログラムカウンタ
`m_BranchResult`	分岐の成否
`m_PC`	現在のプログラムカウンタ

命令フィールド名で命令フィールドにアクセスできる．

本生成フレームワークが命令属性記述から生成するヘルパマクロの例としてMIPS64のDADD命令のエントリに対応するヘルパマクロを**図4**に示す．DADD命令の命令動作関数のシグネチャを得るには，`DEFINST(DADD)` と記述すればよい．プリプロセッサは，`DEFINST` の展開を2段階で行う．まず，プリプロセッサは，`DEFINST` を `DEF_命令名` の形式の `DEF_DADD` に展開する．次に `DEF_DADD` を `DADD` の関数シグネチャに展開する．`DEF_命令名`のマクロは命令ごとに生成されるが，生成される `DEFINST` マクロは単一である．シグネチャの引数rs，rt，rdは，表1の命令属性記述に記載された命令フィールド名である．関数ボディでは，命令属性中に記述した名前rs，rt，rdで命令フィールドにアクセスすることができる．

4.2.2　プログラムカウンタ関連の操作

本生成フレームワークでは，プログラムカウンタの更新と遅延スロットの実行を自動生成による命令動作関数ラッパが行う．したがって，命令動作関数ではこれらの処理の記述が不要である．

命令動作関数ラッパでは，現在のプログラムカウンタの値に命令属性記述から得られる命令長を加えることで，プログラムカウンタを更新することができる．しかし，分岐命令の命令動作関数ラッパは，分岐先アドレスや分岐の成否を命令動作関数の出力から得る必要がある．本生成フレームワークでは，命令動作関数と命令動作関数ラッパとの間でこれらの情報を伝達できるように，**表2**の変数を予約し，予約変数の使用法を規定している．

予約変数 `m_NextPC` は，分岐命令の命令動作関数が命令動作関数ラッパに分岐先アドレスを伝達するために使用する変数である．分岐命令の命令動作関数は，演算によって求めた分岐先アドレスを予約変数 `m_NextPC` に格納する必要がある．

条件分岐命令の命令動作関数は，分岐条件が成立したか否かを命令動作関数ラッパに伝達する必要がある．予約変数 `m_BranchResult` は，命令動作関数が分岐の成否を命令動作関数ラッパに伝達するための変数である．命令動作関

© 2016 Information Processing Society of Japan

```
1  DEFINST(DADD)
2  {
3    GPR[rd] = GPR[rs] + GPR[rt];
4  }
5
6  DEFINST(BEQL)
7  {
8    m_BranchResult = GPR[rs] == GPR[rt];
9    m_NextPC = m_PC + (sext16(offset) << 2);
10 }
```

図 5 命令動作関数の例
Fig. 5 Example of instruction behavior functions.

数は，分岐が成立した場合に m_BranchResult に true を出力する必要がある．

分岐命令など一部の命令は，命令実行時点でのプログラムカウンタをオペランドとして使用する．本生成フレームワークでは，プログラムカウンタの変数を m_PC として予約している．m_PC の更新は命令動作関数ラッパによって行われる．このため，本生成フレームワークでは命令動作関数中における m_PC の更新を禁止し，参照のみ許可している．

なお，本生成フレームワークでは，複数の命令セットで共通化が可能なプログラムカウンタ周りに限定して変数を予約している．命令セット固有のフラグやレジスタに関しては本生成フレームワークでは予約しないため，個々の命令セット向けの記述で対応する必要がある．たとえば，本生成フレームワークでは，ARM の条件付き実行のような命令セット固有の実行制御が命令動作記述中に記述されることを想定している．

4.2.3 命令動作記述例

C++言語による命令動作関数の記述例として，表 1 で命令属性記述を示した MIPS64 の DADD 命令と BEQL 命令の命令動作関数を図 **5** に示す．DADD 命令は，64 ビット汎用レジスタどうしを加算し，結果を 64 ビット汎用レジスタに格納する命令である．図 4 の DEFINST を用いることで，DADD 命令の命令動作関数の引数名は，命令属性記述中のフィールド名 rs，rt，rd となる．GPR は，プロセッサ資源記述ファイルで定義された 64 ビット整数型配列である．

BEQL 命令は，2 つのレジスタオペランドが同じ値の場合に分岐を行う条件分岐命令である．BEQL 命令の命令動作関数では，分岐の成否を予約変数 m_BranchResult に格納し，分岐先アドレスを予約変数 m_NextPC に格納する．分岐先アドレスの計算には，予約変数 m_PC の値を使用する．

5. 実装

ISS の主要処理は，終了条件を満たすまで命令の実行を繰り返すループ処理である．ループの内側処理フローとコードの対応を図 **6** に示す．ループの内側処理全体を実行

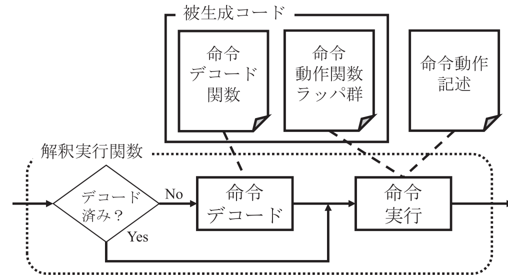

図 6 処理フローとコードの対応
Fig. 6 Mapping between the processing flow and the code.

```
1 void Core::InterpretInstruction()
2 {
3     DecodingResult* result = Lookup(m_PC);
4     if (!result) {
5         result = Decode(m_PC);
6         Register(m_PC, result)
7     }
8     (this->*result->wrapper)(result->fields)
9 }
```

図 7 解釈実行関数の定義
Fig. 7 Definition of the interpretation function.

するのは，命令セット非依存の解釈実行関数である．

被生成コードである命令デコード関数は，命令デコード処理を実行する．また，命令動作関数ラッパは，命令動作記述中の命令動作関数と連携動作することで，命令実行処理を行う．以下本章では，解釈実行関数，命令デコード関数，命令動作関数ラッパの実装を詳細に示す．

5.1 解釈実行関数

解釈実行関数は，現在のプログラムカウンタが指す命令を実行する関数である．解釈実行関数は，基本的にプログラムカウンタが指す 1 命令のみを実行するが，当該命令が遅延スロットを持つ場合，遅延スロット中の命令も連続して実行する．

本生成フレームワークでは，命令実行における処理を命令セット依存部と命令セット非依存部に分離することで，解釈実行関数自体を命令セット非依存の共通処理として実現している．本生成フレームワークでは，Template Method パターン [27] を採用しており，解釈実行関数を親クラスで定義し，解釈実行関数から呼び出される命令セット依存の関数をサブクラスで定義する．被生成コードである命令デコード関数と命令動作関数ラッパ群は，サブクラスの仮想関数である．

図 **7** に解釈実行関数を簡略化した C++コードを示す．解釈実行関数 InterpretInstruction は，プログラムカウンタ用の予約変数 m_PC が指す命令を実行する．また，このとき遅延スロットが存在すれば，遅延スロットの命令も実行する．

3–7 行目のコードは，必要に応じて命令デコードを行う処理である．3 行目では，はじめにプログラムカウンタ m_PC

© 2016 Information Processing Society of Japan

をキーとしてデコード済みの結果を検索する（関数 Lookup の呼び出し）．5 行目では，デコード結果が得られなかった場合のみ m_PC の命令をデコードする（関数 Decode 呼び出し）．また，1 度デコードした結果を後で再利用できるように m_PC と関連付けてデコード結果キャッシュに格納しておく（関数 Register の呼び出し）．なお，デコード結果キャッシュのデータ構造としてはハッシュテーブルを採用している．

命令デコード結果 result は下記のメンバから構成される．

wrapper 命令動作関数ラッパのポインタ

fields 命令フィールド配列

命令フィールド配列のサイズは，デコード対象命令の命令フィールド数に対応するため，可変である．

8 行目では，命令フィールド配列を引数とし，命令動作関数ラッパを呼び出すことで，命令ごとの処理を実行する．引数とする命令フィールド配列の要素数は，命令動作関数ラッパ側で引数の要素数が既知であるため，配列要素数を引数とする必要はない．

5.2 命令デコード関数

命令デコード関数 Decode は，解釈実行関数から呼び出される命令セット依存の被生成コードである．命令セットごとのサブクラスが命令デコード関数をメンバ関数として持つ．命令デコード関数は，以下の前半部分と後半部分から構成される．

前半 プログラムカウンタ位置の命令語の命令を識別する．

後半 特定した命令のデコード結果を作成する．

本生成フレームワークでは，文献 [14], [15] のアルゴリズムで命令デコード関数の前半部分を自動生成している．文献 [14], [15] のアルゴリズムを採用している理由は，ARM や RH850 などの変則的な命令セットに対応するためである．変則的でない命令セットのみを対象とする場合，文献 [12] などのアルゴリズムでも本生成フレームワークを実現できる．

命令デコード関数生成に必要な入力は，命令属性記述に含まれるエントリごとの命令名，ビットパターン，除外条件である．また，文献 [14], [15] のアルゴリズムの出力は，命令デコードツリーと呼ばれる決定木である．本生成フレームワークでは，決定木のプログラム表現として switch 文を採用している．switch 文は，命令のノードに対応し，case 文の処理が子ノードに対応する．各 switch 文では，プログラムカウンタが指す命令語の一部を検査し，次のノードを決定する．対応する case 文がネストされた switch 文の場合，子ノードはサブツリーのルートである．一方，case 文が switch 文でない場合，子ノードはリーフである．switch 文がリーフに到達するのは，プログラムカウンタが指す命令が特定された場合である．リーフに対応する処理では，

```
1  DecodingResult* p = AllocDecodingResult(3);
2  p->wrapper = &WrapperDADD;
3  p->fields[0] = Extract(i, 21, 5);
4  p->fields[1] = Extract(i, 16, 5);
5  p->fields[2] = Extract(i, 11, 5);
6  return p;
```

図 8 命令デコード関数の後半部分（MIPS64 DADD）
Fig. 8 The second half of the instruction decoding function (MIPS64 DADD).

命令のデコード処理後半を行う．なお，決定木用の switch 文は，コンパイラによるテーブルジャンプ最適化でルックアップテーブルに変換される．このため，命令デコード関数で 1 つの switch 文における処理の計算量は $O(1)$ である．

命令のデコード処理後半では，命令語から命令フィールドを抽出し，命令デコード結果のデータレコードを作成する．本生成フレームワークでは，命令動作関数ラッパのアドレス取得に必要な関数名を命令属性記述中の識別子から作成する．また，命令属性記述中の命令フィールドの位置，サイズの情報を用いて命令語から各命令フィールドを抽出する処理を生成する．

生成される命令デコード関数後半部分の例として，**図 8** に MIPS64 の DADD 命令に対応する処理を示す．図 8 のコードは，命令デコード関数で命令が DADD 命令に特定された後に実行される処理である．1 行目では，デコード結果保持用の領域をヒープから割り当てる（AllocDecodingResult 呼び出し）．ここで AllocDecodingResult の引数は，命令フィールド配列の要素数である．2 行目では，命令動作関数ラッパのアドレスを DADD 命令の命令動作関数ラッパ WrapperDADD のアドレスに設定している．WrapperDADD の名前は，命令属性記述中の命令名 DADD から作成される．3–5 行目は，命令フィールドを命令語の変数 i から抽出する処理である．関数 Extract は，第 1 引数の値から第 2 引数と第 3 引数で指定されるビット列を抽出する処理である．ここで，第 2 引数は抽出するビット列の最下位ビットの位置であり，第 3 引数は抽出するビット列のビット幅である．

5.3 命令動作関数ラッパ

本生成フレームワークは，命令属性記述から命令動作関数ラッパを生成する．命令動作関数ラッパは，命令属性記述に含まれる命令ごとに 1 つ生成される．解釈実行関数は命令動作関数ラッパを呼び出すことで，命令固有の処理を行う．命令動作関数ラッパは，命令デコード結果の一部として返される関数であり，以下の役割を持つ．

- 命令フィールド配列を引数リストに分解し，命令動作関数を呼び出す．
- プログラムカウンタの操作や遅延スロットの操作を補完する．

命令動作関数の引数の数が命令ごとに異なるのに対し，

© 2016 Information Processing Society of Japan

命令動作関数ラッパは，命令フィールド配列を唯一の引数とする．このため，呼び出し側では，実行する命令によらず共通のI/Fで命令動作関数を呼び出すことができる．命令動作関数ラッパは，命令フィールド配列の各要素を，適切な順序で命令動作関数の引数に設定し，命令動作関数を呼び出す．

命令動作関数は，命令依存の処理のみを含んでおり，プログラムカウンタの更新処理や遅延スロットの処理を含まない．命令動作関数ラッパは，プログラムカウンタの更新や遅延スロットの処理を補完し，命令動作関数を用いることで，命令ごとの処理を完結させる．

プログラムカウンタの操作や遅延スロットの操作は，下記の命令の種類ごとに共通化される．

- 非分岐命令
- 分岐命令
  - 無条件分岐命令
  - 条件分岐命令
    - 非likely命令
    - likely命令

本生成フレームワークは，命令属性記述の情報から命令ごとに上記のいずれに該当するかを判定し，命令ごとに適切なコードを生成する．以下本節では，上記の分類ごとに被生成コードの詳細を示す．

5.3.1 非分岐命令

非分岐命令の場合の命令動作関数ラッパの処理手順は下記のとおりである．

step 1 プログラムカウンタ m_PC を後続命令のアドレスに更新する．

step 2 命令動作関数を実行する．

step 1では，次の命令の実行に備えてプログラムカウンタ m_PC の値を更新する．模擬プログラムカウンタの新しい値は，プログラムカウンタの値に命令長を加えた値である．本生成フレームワークでは，命令動作関数ラッパの生成時，命令属性記述の命令ビットパターンのビット長から命令長を取得する．

step 2では，命令動作関数を呼び出すことで，命令固有の処理を行う．命令動作関数ラッパは，引数の命令フィールド配列の各要素を命令動作関数の引数として渡す．

本生成フレームワークでは，コード生成時のオプションで，step 1のプログラムカウンタの更新とstep 2の命令動作関数実行の順序を入替え可能としている．この入替えオプションは，非分岐命令のみでなく分岐命令の命令動作関数ラッパの生成にも同様に影響する．プログラムカウンタの更新を命令実行の前にすべきか後にすべきかは，命令セットに依存する．MIPSのように命令実行時点のプログラムカウンタが次の命令のアドレスとなっている場合，前者が適切である．一方，命令実行時点のプログラムカウンタが実行中の命令のアドレスとなる命令セットの場合，後者が適切である．以降の説明では，前者を前提とする．

```
 1 void MIPS64Core::WrapperDADD(uint32_t* fields)
 2 {
 3     /* step 1 */
 4     m_PC += 4;
 5
 6     /* step 2 */
 7     uint32_t rs = fields[0];
 8     uint32_t rt = fields[1];
 9     uint32_t rd = fields[2];
10     GPR[rd] = GPR[rs] + GPR[rt];
11 }
```

図 9 非分岐命令動作関数ラッパの例

Fig. 9 Example of an instruction behavior function wrapper for a non-branch instruction.

非分岐命令の命令実行手順の例として**図 9**にMIPS64のDADD命令に対応する命令動作関数ラッパを示す．DADD命令は，64ビット汎用レジスタどうしの加算命令である．4行目では，step 1に対応する処理であり，プログラムカウンタを次の命令のアドレスに進める．7–10行目は，命令動作関数の呼び出しをインライン展開した結果を示している．10行目が命令動作関数に記述されているコードである．

5.3.2 無条件分岐命令

分岐命令用の命令動作関数ラッパは，プログラムの更新方法が非分岐命令の場合と異なる．命令が分岐命令かつ無条件分岐命令の場合の命令動作関数ラッパの処理手順は下記のとおりである．

step 1 プログラムカウンタ m_PC を後続命令のアドレスに更新する．

step 2 命令動作関数を実行する．

step 3 遅延スロットの命令列を実行する．

step 4 分岐先 m_NextPC でプログラムカウンタ m_PC を更新する．

step 1とstep 2は，非分岐命令の場合と同様である．step 3では，命令の遅延スロットの数だけ，後続の命令を実行する．命令の遅延スロット数が0の場合，step 3は省略される．命令動作関数ラッパは，解釈実行関数を再帰的に呼び出すことで，遅延スロットの命令を実行する．一般にプロセッサは遅延スロットに分岐命令が入る場合の動作を未定義としている．このため，通常遅延スロットには，非分岐命令のみが入る．したがって，遅延スロット中の非分岐命令の実行時には遅延スロットがないため，通常では再帰のネスト数を1までと想定してよい．

step 4では，プログラムカウンタ m_PC を分岐先のアドレス m_NextPC で更新する．分岐先のアドレスは，step 2の命令動作関数の実行結果として予約変数 m_NextPC に格納されている．

無条件分岐命令の命令動作関数ラッパの例として，MIPS64のJ命令に対応する命令動作関数ラッパを**図 10**に示す．図10のコメントは上記のstep 1–step4と対応し

© 2016 Information Processing Society of Japan

```
1  void MIPS64Core::WrapperJ(uint32_t* fields)
2  {
3      /* step 1 */
4      m_PC += 4;
5
6      /* step 2 */
7      uint32_t instr_index = fields[0];
8      m_NextPC = ((m_PC >> 28) << 28) | (instr_index
           << 2);
9
10     /* step 3 */
11     InterpretInstruction();
12
13     /* step 4 */
14     m_PC = m_NextPC;
15 }
```

図 10 無条件分岐命令動作関数ラッパの例

Fig. 10 Example of an instruction behavior function wrapper for an unconditional branch instruction.

ている．7–8 行目は J 命令に対応する命令動作関数の呼び出しを展開したコードであり，8 行目が命令動作関数中に記述されているコードである．11 行目で前述の解釈実行関数 `InterpretInstruction` を再帰的に呼び出すことで，遅延スロットを実行している．

5.3.3 非 Likely 命令

非 Likely 命令は，条件分岐命令のうち分岐の成否によらず後続遅延スロットの命令を実行する命令である．非 Likely 命令の命令動作関数ラッパの処理手順を以下に示す．

step 1 分岐の成否 `m_BranchResult` を false に設定する．

step 2 プログラムカウンタ `m_PC` を後続命令のアドレスに更新する．

step 3 命令動作関数を実行する．

step 4 遅延スロットの命令列を実行する．

step 5 分岐の成否 `m_BranchResult` が true の場合，分岐先 `m_NextPC` にプログラムカウンタ `m_PC` を更新する．

無条件分岐命令との相違は，step 1 で分岐の成否 `m_BranchResult` を false に設定する点と step 5 でプログラムカウンタ `m_PC` の更新を選択的に行う点である．

非 lileky 命令の場合，step 3 の命令動作関数では，分岐成立の場合に分岐の成否 `m_BranchResult` が true に設定されることを記述ガイドラインで定めている．step 1 で分岐の成否 `m_BranchResult` を事前に false に設定しておくことで，命令動作関数が分岐不成立時に `m_BranchResult` の設定を省略することができる．

step 5 では，分岐の成否 `m_BranchResult` の値が true の場合のみプログラムカウンタを更新する．プログラムカウンタに設定すべき値は，step 3 の命令動作関数の実行で `m_NextPC` に格納されている．

非 Likely 命令の命令動作関数ラッパの例として，MIPS64 の BEQ 命令の命令動作関数ラッパを**図 11** に示す．図 11 中のコメントは，上記 step 1–step 5 との対応を表している．10–14 行目は，BEQ 命令の命令動作関数の呼び出しをイン

```
1  void MIPS64Core::WrapperBEQ(uint32_t* fields)
2  {
3      /* step 1 */
4      m_BranchResult = false;
5
6      /* step 2 */
7      m_PC += 4;
8
9      /* step 3 */
10     uint32_t rs = fields[0];
11     uint32_t rt = fields[1];
12     uint32_t offset = fields[2];
13     m_BranchResult = GPR[rs] == GPR[rt];
14     m_NextPC = m_PC + (SignExtend16(offset) << 2);
15
16     /* step 4 */
17     InterpretInstruction();
18
19     /* step 5 */
20     if (m_BranchResult) {
21         m_PC = m_NextPC;
22     }
23 }
```

図 11 非 Likely 命令動作関数ラッパの例

Fig. 11 Example of an instruction behavior function wrapper for a non-likely instruction.

ライン展開したコードである．BEQ 命令は，分岐の成否によらず 17 行目で解釈実行関数 `InterpretInstruction` を再帰的に呼び出すことで，遅延スロットの命令を実行する．20–21 行目では，分岐の成否 `m_BranchResult` が true の場合のみ `m_PC` を `m_NextPC` で更新する．

5.3.4 Likely 命令

Likely 命令は，条件分岐命令のうち分岐成立時にのみ遅延スロットを実行し，分岐不成立時には遅延スロットの実行をスキップする命令である．Likely 命令の場合，遅延スロット実行の要否を分岐の成否によって選択的に行う必要がある．Likely 命令に対応する命令動作関数ラッパの処理手順を以下に示す．

step 1 分岐の成否 `m_BranchResult` を false に設定する．

step 2 プログラムカウンタ `m_PC` を後続命令のアドレスに更新する．

step 3 命令動作関数を実行する．

step 4 分岐の成否 `m_BranchResult` が true の場合，遅延スロットの命令列を実行し，それ以外の場合，遅延スロットの命令列をスキップする．

step 5 分岐の成否 `m_BranchResult` が true の場合，分岐先 `m_NextPC` にプログラムカウンタ `m_PC` を更新する．

非 Likely 命令との相違は，step 4 で分岐の成否によって遅延スロットの処理を切り替える点である．step 4 では，命令動作関数の出力から分岐の成否を判定し，遅延スロットの処理を選択的に行う．分岐成立の場合，非 Likely 命令の場合と同様に解釈実行関数 `InterpretInstruction` の再帰呼び出しで遅延スロットの命令を実行する．一方，分岐不成立の場合，遅延スロット中の命令列をスキップする．step 4 と step 5 における `m_BranchResult` による分岐は，

© 2016 Information Processing Society of Japan

```
1  void MIPS64Core::WrapperBEQL(uint32_t* fields)
2  {
3      /* step 1 */
4      m_BranchResult = false;
5
6      /* step 2 */
7      m_PC += 4;
8
9      /* step 3 */
10     uint32_t rs = fields[0];
11     uint32_t rt = fields[1];
12     uint32_t offset = fields[2];
13     m_BranchResult = GPR[rs] == GPR[rt];
14     m_NextPC = m_PC + (SignExtend16(offset) << 2);
15
16     /* step 4 */
17     if (m_BranchResult) {
18         InterpretInstruction();
19     }
20     else {
21         SkipInstruction();
22     }
23
24     /* step 5 */
25     if (m_BranchResult) {
26         m_PC = m_NextPC;
27     }
28 }
```

図 12 非 Likely 命令動作関数ラッパの例
Fig. 12 Example of an instruction behavior function wrapper for a likely instruction.

単一のステップにまとめることも可能である．しかし，それぞれ独立したステップとするほうが命令動作関数ラッパの生成プログラムを簡素化できる．

Likely 命令の動作関数ラッパの例として，MIPS64 の BEQL 命令に対応する命令動作関数ラッパを**図 12** に示す．BEQL 命令は，BEQ 命令の Likely 命令版である．10–14 行目は，BEQL 命令の命令動作関数の呼び出しをインライン展開したコードである．17–22 行目の処理では，遅延スロットの実行を選択的に行う．21 行目は，命令の実行をスキップする関数の呼び出しである．関数 `SkipInstruction` は，プログラムカウンタが指す命令語の命令長分だけプログラムカウンタを進めることで，命令をスキップする．

6. 実験結果

本生成フレームワークの有用性を確認することを目的に，本生成フレームワークで ARM，MIPS64，SH の 3 つの命令セットを対象に ISS を開発し，生産性・保守性および性能の観点で評価を行った．生産性と保守性の評価では，開発した ISS のコードメトリクスの計測結果を指標とし，性能評価では，ベンチマークソフトウェア実行時の性能計測結果を指標とした．また，同一の計測を GDB [19] に付属の ARM 用 ISS に対しても行い，計測結果を本生成フレームワークと比較することで，有用性を確認した．比較対象として GDB の ISS を採用した理由は，当該 ISS が汎用プログラミング言語で記述されたインタプリタ方式の実用 ISS であるためである．GDB のソースコードのうち ARM 向けの ISS は C 言語でハンドコーディングされており，本生成フレームワークのコードと同一の計測ツールでメトリクスを取得することができる．また，ソースコードに性能計測用コードを入れることで，本生成フレームワークと同一条件で性能を計測することが可能である．GDB の MIPS64 向け ISS と SH 向け ISS は，それぞれ異なる記法で記述されており，汎用の計測ツールでメトリクスを取得できないため，今回計測対象外とした．なお，比較に使用した GDB のバージョンは，本論文執筆時点で最新の 7.10 である．

以下本章では，コードメトリクスと性能の計測・比較の結果と考察を示す．

表 3 コードメトリクス一覧
Table 3 Code metrics.

	GDB	提案フレームワーク		
	ARM	ARM	MIPS64	SH
実装命令数	170	175	234	152
コード行数	3,461	2,111	1,257	870
命令あたりの平均コード行数	20	12	5	6
関数数	23	194	294	157
関数あたりの平均コード行数	143	11	4	6
最大サイクロマティック複雑度	1,092	17	3	15
平均サイクロマティック複雑度	60	3	1	1
合計サイクロマティック複雑度	1,374	615	316	193

6.1 コードメトリクス

ISS を開発するために必要となる記述のコードメトリクスを計測し，ハンドコーディングで開発された GDB の ISS と比較した．計測結果のコードメトリクス一覧を**表 3** に示す．コード行数，関数数，関数あたりの平均コード行数，各種サイクロマティック複雑度 [28] の値は，コード解析ツール Understand [29] による計測結果である．サイクロマティック複雑度は，保守性の指標である．一般にサイクロマティック複雑度が大きいほど保守性は低下する．プロセッサが拡張された場合，ISS に対しても命令の追加・修正などの保守開発が必要である．命令セットは下位互換を保ちつつ拡張されることが多い．このため，ISS の保守性は生産性とともに重要である．

GDB の結果は，整数命令実装ファイル armemu.c を対象として計測した結果である．また，本生成フレームワークの計測結果は，命令動作記述ファイルを対象として計測した結果である．なお，同一項目で比較できない命令属性記述はメトリクスの計測対象外である．命令属性記述の場合，プロセッサのマニュアルから必要項目を転記する作業が主である．今回作成した ARM，MIPS64，SH ともに命令属性記述の作成に要した工数は 1 人日以内である．

表 3 の実装命令数は，対象とするファイルが実装している命令の数である．armemu.c では命令デコードと命令実行が単一のファイルで記述されているため，実装されて

© 2016 Information Processing Society of Japan

いる命令数をコードから読み取ることが困難である．そこで，ARMv7 の整数命令の全 271 命令をひととおり GDB で実行し，未定義命令と出力された 101 命令を除いた命令数を記載している．GDB のようにハンドコーディングで複数の命令の処理が単一の関数で記述されている場合，実装されている命令の洗い出しや命令の実装箇所の特定が困難である．このため，GDB に対する命令の追加や修正は難易度が高いといえる．一方，本生成フレームワークでは 1 命令ごとに命令動作関数が存在するため，命令実装数は，命令動作関数の数と一致する．本生成フレームワークで命令を追加する場合，命令属性記述にエントリを追加し，命令動作関数を記述すればよい．

表 3 のコード行数は，コメントを除いたコードの行数である．命令あたりの平均コード行数は，コード行数と実装命令数の商である．ARM を対象とした場合の GDB と本生成フレームワークとの比較では，本生成フレームワークの命令あたりの平均コード行数は，GDB より 40%小さい 12 行である．これは，GDB では命令デコード処理の記述が必要である一方，本生成フレームワークでは命令デコード処理と命令フィールドの抽出処理の記述が不要なためである．

本生成フレームワークのコード行数を異なる命令セット間で比較した場合，MIPS64，SH，ARM でそれぞれ平均コード行数は 4，6，11 と異なる．これはアーキテクチャの複雑性に起因するものである．ARM は，大部分の命令が条件付き命令である点や，シフト付きオペランドをとる命令が多い点などで，SH と MIPS64 より複雑である．結果として，ARM の場合，命令あたりの平均コード行数が 12 と多くなる．一方，MIPS64 の命令あたりの平均コード行数が 4 と少ない原因としては，MIPS64 にはプロセッサのステータスフラグが存在しないことが考えられる．ARM や SH では演算のキャリーやオーバフローの有無などを保持するステータスフラグの更新が必要である一方，MIPS64 ではステータスフラグが存在しない．したがって，MIPS64 の命令動作記述では，これらの処理の記述が不要である．

表 3 の関数数は，計測したファイルに含まれる関数の数である．GDB の計測対象ファイル armemu.c は，大部分の処理を行う関数 `ARMul_Emulate26` とサブ関数群の合計 23 個の関数を含む．一方，本生成フレームワークの場合，命令動作記述ファイルは，命令動作関数とそのサブ関数群の合計 194 個の関数を含む．ARM 向け ISS のサイクロマティック複雑度の比較では，本生成フレームワークの最大サイクロマティック複雑度は GDB より 98%以上小さい 17 であり，平均サイクロマティック複雑度は GDB より 95%小さい 3 である．また，合計サイクロマティック複雑度 615 は，本生成フレームワークのほうが GDB より 60%小さい．合計サイクロマティック複雑度の差異は，命令デコード処理の記述有無が主な要因であると考えられる．

ARM，MIPS64，SH のアーキテクチャ間におけるサイクロマティック複雑度の比較では，ARM が最も多い．ARM の最大サイクロマティック複雑度が 17 で他の命令セットより大きい原因は，ARM の命令が条件付きであるためである．条件付き命令は，17 通りの実行条件を持つが，命令の実行条件を確認するサブ関数が最大サイクロマティック複雑度 17 となっている．また，平均および合計サイクロマティック複雑度が ARM の方が MIPS64 と SH より大きい原因は，命令が条件付きであることや，命令動作関数の多くがフラグによる処理の切替えを含むためである．命令動作記述において処理の切替えは分岐文で実装されるため，複雑度が増加する．

以上より，本生成フレームワークでは，命令セットごとに総記述量の差異はあるものの，ハンドコーディングよりも少ないコード記述量と複雑度で ISS を開発することができると考えられる．

6.2 性能

本節では，本生成フレームワークで開発した ISS の性能評価結果と考察を示す．本計測では，ベンチマークソフトウェアである Dhrystone [30] と CHStone [31] を本生成フレームワークによる ISS 上と GDB 付属の ISS 上で実行し，処理性能とデコード結果キャッシュの消費メモリを計測した．また，ARM をターゲットとする GDB 付属の ISS との比較を行った．CHStone を選択した理由は，標準ライブラリやシステムコールの呼び出しを含まないため，命令の実行性能や消費メモリの計測および比較に適しているためである．

性能評価に用いた実験環境を**表 4** に示す．ベンチーマークプログラムのビルドに用いたクロスコンパイラは，ターゲットが ARM，MIPS64，SH いずれの場合も GCC 4.8.3 である．また，本生成フレームワークによる ISS と GDB 付属 ISS のビルドには，GCC 4.7.2 を使用し，同一の最適化オプションを指定した．

6.2.1 実行性能

Dhrystone と CHStone の各プログラム実行時の計測結果一覧を**表 5** に示す．表 5 の先頭行は Dhrystone の実行結果であり，以降は CHStone の各プログラムの実行結果である．なお，計測時に Dhrystone の実行パラメータである繰返し回数には 1,000 回を指定した．表 5 における性能指

表 4 実験環境
Table 4 Experiment environment.

ホストコンパイラ	GCC 4.7.2
クロスコンパイラ	GCC 4.8.3
ホストプロセッサ	Intel Core i5-4590 3.3 GHz
ホストメモリ	8 GB
ホスト OS	Debian Linux 7.9 64 bit

© 2016 Information Processing Society of Japan

表 5 ベンチマーク実行結果一覧
Table 5 Results of benchmarks.

	GDB		提案フレームワーク					
	ARM		ARM		MIPS64		SH	
プログラム	CPI	命令数	CPI	命令数	CPI	命令数	CPI	命令数
dhrystone	62.76	257,044	32.00	257,044	24.24	306,080	24.04	307,084
adpcm	57.41	84,653	38.28	84,653	28.47	97,555	28.17	203,787
aes	50.74	28,030	30.33	28,030	25.92	24,132	24.08	69,258
blowfish	52.47	559,361	31.36	559,361	24.17	747,366	26.42	854,894
dfadd	61.68	5,969	38.57	5,969	24.62	3,135	28.91	9,325
dfdiv	61.06	11,352	36.38	11,352	23.63	2,187	28.87	16,350
dfmul	63.25	2,746	38.08	2,746	23.50	1,393	27.44	7,600
dfsin	59.22	460,738	36.91	460,738	26.31	92,281	29.73	569,297
gsm	65.15	14,546	30.31	14,546	23.34	16,462	24.76	26,181
jpeg	50.71	2,030,356	31.83	2,030,356	24.62	2,492,796	25.07	3,897,467
mips	49.21	19,156	27.82	19,156	22.28	20,829	23.51	33,409
motion	53.45	2,287	33.13	2,287	28.84	1,446	28.45	2,954
sha	44.66	545,360	30.28	545,360	22.81	662,064	23.06	779,983

標は，CPI（Cycles Per Instruction）である．CPI は ISS 上でベンチマークプログラムを実行した際におけるホスト計算機の実行サイクル数と ISS の実行命令数の商である．ISS が同じ命令セットを対象とする場合，CPI の値が小さいほど ISS は高性能である．

一部のプログラムでは，命令セットとの相性があり，dfdiv のように実行命令数が大きく異る場合が存在する．dfdiv は 64 ビットの整数演算を多数含むため，64 ビットの演算命令を備える MIPS64 の場合のみ極端に命令実行数が少ない．プログラムと命令セットに特定の相性がある場合を除くと，ベンチマークプログラムの相違による CPI の大小は，命令セットによらず類似の傾向である．たとえば，プログラムが mips の場合，本生成フレームワークの ARM と MIPS64 の CPI はそれぞれ最小であり，GDB と本生成フレームワークの SH はそれぞれ 2 番目に小さい値である．

本生成フレームワークによる ISS の Dhrystone および CHstone 実行時の CPI は，MIPS64 と SH とで大差がないのに対し，ARM の場合のみ MIPS との比較で 1.15（motion）倍–1.62（dfmul）倍である．この原因は，ARM の命令あたりのコード行数が多い原因と同様，ARM の命令が条件付きであるためや命令がシフト付きオペランドを持つためであると考えられる．命令あたりのコード行数と CPI は関連しており，命令あたりのコード行数が多いほど CPI も大きい．

本生成フレームワークの ISS と GDB との比較では，本生成フレームワークの CPI のほうが GDB よりも 32%（sha）–53%（gsm）小さく高性能である．本生成フレームワークのほうが GDB よりも高性能である主な原因は，本生成フレームワークではデコード結果をキャッシュするため，デコード処理を省略できるためであると考えられる．

Dhrystone の実行結果から算出した DMIPS 値を**表 6** に示す．本生成フレームワークによる ISS 間の比較では ARM が 194 で最も小さいが，GDB との比較では約 2 倍の性能である．本生成フレームワークによる ARM ISS の CPI は Dhrystone 実行の場合で MIPS64 の 1.32 倍であるが，DMIPS 値の性能低下は 10%以内にとどまっている．これは，ARM の Dhryston 命令の実行回数が MIPS64 の場合より 16%少ないためである．ARM の命令は高機能であるため，32 ビット演算が主体のプログラムを対象とする場合，MIPS64 や SH よりも命令数が少なくなると考えられる．表 3 に示したとおり，ARM の ISS は命令あたりのコード行数が MIPS64 と SH より多い．このため，ARM の方が高機能といえる．

表 6 Dhrystone 実行結果
Table 6 Results of Dhrystone.

	GDB	提案フレームワーク		
	ARM	ARM	MIPS64	SH
DMIPS	99	194	215	216

Dhrystone の結果より，異なる命令セットの ISS 間における CPI の相違は，プログラムのシミュレーション時間には影響しないと考えられる．このことは，**表 7** に示したホスト計算機の実行サイクル数からも確認することができる．表 5 では ARM の CPI はすべてのベンチマークプログラムで SH より大きい．しかし，ARM の実行サイクル数は，13 個のプログラムのうち 11 個で SH より小さい．したがって，異なる命令セット間での実行サイクル数の相違は，CPI の相違だけでなく，プログラムと命令セットの相性による影響が大きいと考えられる．

Dhrystone と CHstone の実行結果より，命令セットとプログラムの相性によってシミュレーション時間に相違はあるものの，本生成フレームワークは命令セットによらず高

© 2016 Information Processing Society of Japan

表 7 ホスト計算機の実行サイクル数一覧
Table 7 Numbers of host CPU clock cycles.

	ARM	MIPS64	SH
dhrystone	8,226,138	7,418,139	7,381,506
adpcm	3,240,765	2,777,598	5,741,259
aes	850,050	625,566	1,667,856
blowfish	17,540,631	18,061,374	22,586,709
dfadd	230,220	77,190	269,547
dfdiv	412,953	51,672	471,993
dfmul	104,577	32,733	208,560
dfsin	17,005,752	2,427,879	16,922,628
gsm	440,844	384,216	648,294
jpeg	64,629,264	61,374,951	97,705,707
mips	532,989	464,124	785,526
motion	75,759	41,697	84,027
sha	16,514,727	15,098,637	17,985,702

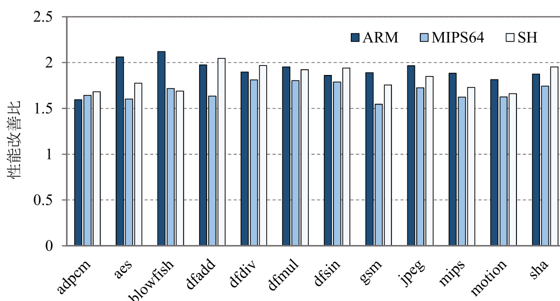

図 13 デコード結果キャッシュによる性能改善比
Fig. 13 Speed up with decoding cache.

性能な ISS を構築することができるといえる．

6.2.2 デコード結果キャッシュの利用有無による差異

本生成フレームワークでは，デコード結果キャッシュを用いて 1 度デコードした結果を再利用する方式を採用している．デコード結果キャッシュの利用による性能改善の効果を確認するため，デコード結果キャッシュの利用有無での性能比較を行った．

デコード結果キャッシュの利用による性能改善比を**図 13** に示す．プログラムごとの性能改善比は命令セット間で共通の傾向は見られない．たとえば，ARM では blowfish の場合に最も性能改善比が良いが，MIPS64 では dfdiv の場合が最も性能改善比が良い．これは命令のデコードの複雑性は命令の演算種類に無関係であるためと考えられる．しかし，全体的には，対象とするプログラムと命令セットによらず性能が 1.5 倍以上に改善されている．このため，デコード結果キャッシュの利用は有用な高速化手法であるといえる．

6.2.3 消費メモリサイズ

デコード結果キャッシュを用いる本生成フレームワークによる ISS では，デコード結果をメモリ中に保持する．デコード結果キャッシュによるメモリ消費とプログラムサイズとの関連を確認するため，CHStone 実行時におけるデコード結果キャッシュによる消費サイズを計測した．CHStone はプログラムの実行に不要なコードをほとんど含まない．したがって，CHStone のビルド結果のコード領域のサイズと実行時のキャッシュのサイズを取得することで，コードサイズとキャッシュサイズの関係をおおむね把握することができる．

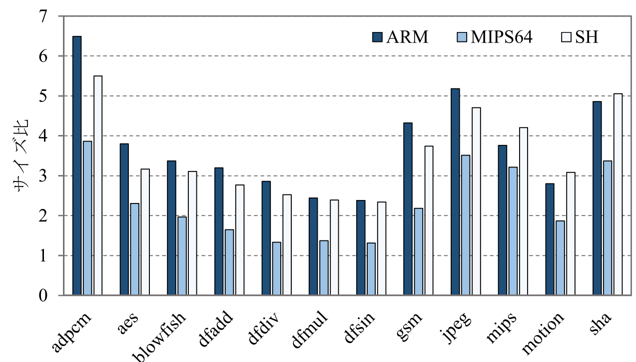

図 14 CHStone コードサイズ比
Fig. 14 Code size ratios of CHStone.

デコード結果キャッシュのサイズとプログラムのコード領域とのサイズ比を**図 14** に示す．プログラムのコード領域のサイズは，ベンチマークプログラムをビルドして得られた実行可能ファイルの .text セクションから取得したサイズである．ベンチマークプログラムの差異によるサイズ比の大小は命令セットに依存せず同じ傾向となっている．命令セット間で比較すると ARM が最もサイズ比が大きい．これは，ARM の命令が MIPS64 や SH と比べて多くの命令フィールドを持つためである．たとえば，ARM の命令では条件コード用の命令フィールドやシフト量，シフトタイプのための命令フィールドが多数の命令に存在する．SH は 2 番目にサイズ比が大きい．SH と MIPS64 では命令オペランドの数に大差はないが，MIPS64 の命令長が 32 ビットであるのに対し，SH の命令長は 16 ビットである．このため，コードサイズ比で比べた場合，SH の方が MIPS64 より大きくなっている．

プログラムごとに相違はあるものの，すべてのケースで命令キャッシュのサイズは，コードサイズの 7 倍以内に収まっている．多くの場合，今日の開発環境のホスト計算機のメモリ容量は，組込みシステムのコード領域のサイズの 10 倍以上である．したがって，デコード結果キャッシュをメモリ上に保持しておくことは，多くの場合に有用であるといえる．

7. 議論

前章まででは，複数の命令セットに対応可能な ISS 生成フレームワークの設計と実装について示した．本章では，評価対象とした ARM，MIPS64，SH 以外のプロセッサへの適用について議論する．

© 2016 Information Processing Society of Japan

7.1 可変長命令セットへの対応

一部の組込みプロセッサは可変長の命令セットを持つ．可変長の命令セットとしては，MIPS16，ARM Thumb，PowerPC VLE などが存在する．本生成フレームワークはこれらの命令セットに対しても対応可能な見込みである．

本生成フレームワークが生成する ISS において命令長が関係する部分は，命令デコード関数と命令動作関数ラッパである．命令デコード関数ではプログラムカウンタのアドレスに存在する命令を識別し，命令動作関数ラッパではプログラムカウンタに命令長を加算する．以下では，命令デコード関数と命令動作関数ラッパそれぞれの生成について，可変長命令セットへの対応の観点で言及する．

7.1.1 命令デコード関数の生成

本生成フレームワークが採用している命令デコード関数の生成アルゴリズム [14], [15] は，入力が可変長命令セットの場合も擬似的に固定長命令セットとして扱うことで，命令デコード関数を生成できる．具体的には，本生成フレームワークは命令デコード関数生成時に以下の前処理を行う．

step 1 命令属性記述中のビットパターンの最大長を求める．

step 2 命令属性記述中のすべてのビットパターンを命令の最大長まで任意値でパディングする．

たとえば，16 ビットと 32 ビットの 2 種類の命令長が混在する ARM Thumb の場合，上記 step 1 で命令の最大長として 32 が得られる．Thumb の命令属性記述に対して上記 step 2 を適用した結果，16 ビット長 ADD 命令のビットパターン `10101xxxxxxxxxxx` は，下位 16 ビットが任意値でパディングされ `10101xxxxxxxxxxxxxxxxxxxxxxxxxxx` となる．また，生成フレームワークは，ビットパターンのパディングと同時に命令フィールド位置も更新し，命令デコード関数の命令フィールド抽出処理を生成する．

7.1.2 命令デコード関数ラッパの生成

本生成フレームワークは，命令動作関数ラッパの生成時にプログラムカウンタに命令長を加算するコードを生成する．このため，可変長命令セットの場合も，ISS による命令動作関数ラッパの呼び出しごとにプログラムカウンタは次に実行すべき命令のアドレスを指すことができる．

ISS 生成フレームワークが命令動作関数ラッパ生成時に用いる命令長はパディングなしのビットパターンの長さである．たとえば，Thumb の 16 ビット ADD 命令の命令動作関数ラッパがプログラムカウンタに加算する値は 2 である．本生成フレームワークは，パディング前のビットパターン `10101xxxxxxxxxxx` のバイト長である 2 を 16 ビット ADD 命令の命令長として使用する．

7.2 命令セット切替えへの対応

組込みで用いられる一部のプロセッサは，動的な命令セットの切替えが可能である．命令セットの切替えが可能なプロセッサは命令セット切替え命令の実行を契機として命令セットを切り替える．一般に命令セット切替え命令は，分岐命令の一種である．プログラムは，命令セット切替え命令を用いることで，関数単位で命令セットを切り替えることができる．切替えの例としては，MIPS64 命令セットと MIPS16 命令セット，ARM 命令セットと Thumb 命令セット，PowerPC 命令セットと VLE 命令セットの相互切替えがあげられる．本生成フレームワークを命令セットの切替えが可能なプロセッサに対応させるためには下記 2 点の拡張が必要となる．

- 命令セット切替え I/F の提供
- 命令属性記述の項目追加

以下本節では，それぞれの拡張について詳細を示す．

7.2.1 命令セット切替え I/F の提供

本生成フレームワークを命令セットの切替えに対応させるためには，命令セット切替え命令に対応した命令動作関数を記述できるようにする必要がある．そこで，生成フレームワークが命令セット切替え用の I/F を命令動作関数に対して提供することを考える．

プロセッサによる命令セットの切替えは，命令デコードロジックの切替えに相当する．このため，ISS は命令デコード関数の切替えで命令セットの切替えを模擬できる．命令デコード関数の切替えを実現するには，解釈実行関数の命令デコード関数呼び出しを関数ポインタを介した間接的な呼び出しに変更したうえで，命令セット切替え I/F で当該関数ポインタを変更できるようにすればよい．

命令セット切替え I/F の実現方法の 1 つは，ISS 生成フレームワークが命令セットごとに当該命令セットへの切替え関数を生成することである．この場合，命令セット A の切替え関数は，命令セット A の命令デコード関数を先述の関数ポインタに設定する．ISS 開発者は，命令動作記述の作成時，命令セット A への切替えが必要な部分で命令セット A の切替え関数を使用する．切替え関数の命名規則を命令動作記述の記述ガイドラインに追加しておくことで，ISS 開発者は適切な切替え関数を選択できる．

7.2.2 命令属性記述の項目追加

本生成フレームワークは，命令属性記述ごとに命令デコード関数を生成する．このため，切替え対象となる命令セットごとに命令属性記述を用意することを前提とする場合，命令セット切替え I/F の提供のみで命令セットの切替えに対応可能である．

しかし，命令セットごとに命令属性記述を持つ場合，プロセッサによっては記述効率が悪くなるという問題がある．一部のプロセッサは，切替え対象とする複数の命令セット間で共通した命令を持つ．たとえば，PowerPC e200 コアのプロセッサでは，PowerPC 命令セットと VLE 命令セットとで共通の命令を持つ．このようなプロセッサ向けに命令セットごとの命令属性記述を作成する場合，一部の命令

を命令属性記述間で重複して記述する必要が生じる．

命令の重複記述を回避するためには，命令属性記述の各エントリが属する命令セット群を ISS 開発者が指定可能とし，命令ごとにエントリを 1 つだけ記述するようにすればよい．エントリが属する命令セット群を指定可能とする方法の 1 つは，命令属性記述の項目に命令セット名のリストを追加することである．この場合，ISS 生成フレームワークは，命令属性記述に含まれる命令セット名ごとに命令デコーダと切替え関数を生成する必要がある．ISS 開発者が適切な切替え関数を選択できるようにするには，切替え関数の名前の一部に命令セット名を含むようにし，この命名規則を命令動作記述の記述ガイドラインに加えるとよい．

8. おわりに

本論文では，ISS の開発効率化を目的とする ISS 生成フレームワークについて示した．本生成フレームワークは，命令属性記述と命令動作記述を主な入力とし，インタプリタ方式の ISS を自動生成する．命令属性記述は，命令ごとのエントリから構成され，各エントリは ISS を生成するために必要な情報を保持する．命令動作記述の記述言語は C++言語である．ISS 開発者は，専用のプロセッサ記述言語を習得することなく，命令の動作を記述することができる．本生成フレームワークは，命令属性記述から命令デコード処理，命令フィールド抽出処理，プログラムカウンタ更新処理を自動生成するため，命令動作記述中ではこれらの処理記述を省略可能である．GDB の付属 ISS との比較では，本生成フレームワークの場合，40%少ない記述量かつ 95%小さい平均サイクロマティック複雑度のコードから DMIPS 値が 2 倍の ISS を生成できることを確認した．今後の課題は，本生成フレームワークを VLIW（Very Large Instruction Word）の命令セットに対応させることである．

参考文献

[1] 一場利幸，森　孝夫，高瀬英希，鴫原一人，本田晋也，高田広章：命令セットシミュレータの実行制御機構を用いたマルチプロセッサ RTOS のテスト効率化手法，電子情報通信学会論文誌 D，情報・システム，Vol.95, No.3, pp.387–399 (2012).

[2] Hartoog, M.R., Rowson, J.A., Reddy, P.D., Desai, S., Dunlop, D.D., Harcourt, E.A. and Khullar, N.: Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign, *Proc. 34th Annual Design Automation Conference*, *DAC '97*, pp.303–306, ACM (online), DOI: 10.1145/266021.266110 (1997).

[3] Hadjiyiannis, G., Russo, P. and Devadas, S.: A Methodology for Accurate Performance Evaluation in Architecture Exploration, *Proc. 36th Annual ACM/IEEE Design Automation Conference*, *DAC '99*, pp.927–932, ACM (online), DOI: 10.1145/309847.310100 (1999).

[4] Pees, S., Hoffmann, A., Zivojnovic, V. and Meyr, H.: LISA Machine Description Language for Cycle-accurate Models of Programmable DSP Architectures, *Proc. 36th Annual ACM/IEEE Design Automation Conference*, *DAC '99*, pp.933–938, ACM (online), DOI: 10.1145/309847.310101 (1999).

[5] Khare, A., Savoiu, N., Halambi, A., Grun, P., Dutt, N. and Nicolau, A.: V-SAT: A visual specification and analysis tool for system-on-chip exploration, *Proc. 25th EUROMICRO Conference*, Vol.1, pp.196–203 Vol.1 (online), DOI: 10.1109/EURMIC.1999.794466 (1999).

[6] Reshadi, M., Dutt, N. and Mishra, P.: A retargetable framework for instruction-set architecture simulation, *ACM Trans. Embed. Comput. Syst.*, Vol.5, No.2, pp.431–452 (online), DOI: 10.1145/1151074.1151083 (2006).

[7] Okuda, K., Kobayashi, S., Takeuchi, Y. and Imai, M.: A Simulator Generator Based on Configurable VLIW Model Considering Synthesizable HW Description and SW Tools Generation, *Proc. Workshop on Synthesis and System Integration of Mixed Information Technologies*, pp.152–159 (Apr. 2003).

[8] Kassem, R., Briday, M., BéChennec, J.-L., Savaton, G. and Trinquet, Y.: Harmless, a hardware architecture description language dedicated to real-time embedded system simulation, *J. Syst. Archit.*, Vol.58, No.8, pp.318–337 (online), DOI: 10.1016/j.sysarc.2012.05.001 (2012).

[9] Engel, F., Nuhrenberg, J. and Fettweis, G.: A generic tool set for application specific processor architectures, *Proc. 8th International Workshop on Hardware/Software Codesign, 2000*, *CODES 2000*, pp.126–130 (2000).

[10] Jeremiassen, T.: Sleipnir-An Instruction-Level Simulator Generator, *Proc. 2000 IEEE International Conference on Computer Design: VLSI in Computers & Processors*, *ICCD '00*, pp.23–31, IEEE Computer Society (2000).

[11] Ratsiambahotra, T., Cassé, H. and Sainrat, P.: A versatile generator of instruction set simulators and disassemblers, *Proc. 12th International Conference on Symposium on Performance Evaluation of Computer & Telecommunication Systems*, *SPECTS '09*, pp.65–72, IEEE Press (2009).

[12] Theiling, H.: Generating Decision Trees for Decoding Binaries, *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, *LCTES '01*, pp.112–120, ACM (online), DOI: 10.1145/384197.384213 (2001).

[13] Fournel, N., Michel, L. and Pétrot, F.: Automated Generation of Efficient Instruction Decoders for Instruction Set Simulators, *Proc. International Conference on Computer-Aided Design*, *ICCAD '13*, pp.739–746, IEEE Press (2013).

[14] 奥田勝己，竹山治彦：変則的な命令セットに対応した命令デコーダの自動生成手法，情報処理学会研究報告，Vol.2015-EMB-038, No.2, pp.1–8 (2015).

[15] Okuda, K. and Takeyama, H.: Decision tree generation for decoding irregular instructions, *2016 Design, Automation & Test in Europe Conference & Exhibition*, *DATE 2016*, pp.1592–1597, IEEE (2016).

[16] 吉瀬謙二, 片桐孝洋, 本多弘樹, 弓場敏嗣：SimCore/Alpha Functional Simulator の設計と実装，電子情報通信学会論文誌 D-I，情報・システム，I-情報処理，Vol.88, No.2, pp.143–154 (2005).

[17] 藤枝直輝, 渡邉伸平, 吉瀬謙二：教育・研究に有用な MIPS システムシミュレータ SimMips，情報処理学会論文誌，Vol.50, No.11, pp.2665–2676 (2009).

© 2016 Information Processing Society of Japan

[18] 近江谷康人，天野英晴：C 言語実装を用いたインタプリタ方式の命令エミュレーション性能の向上（コンピュータシステム），電子情報通信学会論文誌 D，情報・システム，Vol.91, No.2, pp.413–434 (2008).

[19] Free Software Foundation: GDB: The GNU Project Debugger (online), available from ⟨https://www.gnu.org/software/gdb/⟩.

[20] Pees, S., Hoffmann, A. and Meyr, H.: Retargetable Compiled Simulation of Embedded Processors Using a Machine Description Language, *ACM Trans. Des. Autom. Electron. Syst.*, Vol.5, No.4, pp.815–834 (online), DOI: 10.1145/362652.362662 (2000).

[21] Zhu, J. and Gajski, D.D.: A Retargetable, Ultra-fast Instruction Set Simulator, *Proc. Conference on Design, Automation and Test in Europe, DATE '99*, ACM (online), DOI: 10.1145/307418.307509 (1999).

[22] Cmelik, R.F. and Keppel, D.: Shade: A Fast Instruction Set Simulator for Execution Profiling, Technical Report, Mountain View, CA, USA (1993).

[23] Witchel, E. and Rosenblum, M.: Embra: Fast and Flexible Machine Simulation, *Proc. 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '96*, pp.68–79, ACM (online), DOI: 10.1145/233013.233025 (1996).

[24] 近江谷康人，天野英晴：性能評価シミュレータ ESPRIT/sim における動的バイナリー変換アクセラレータの簡易実装（コンピュータシステム），電子情報通信学会論文誌 D，情報・システム，Vol.91, No.10, pp.2449–2465 (2008).

[25] Bellard, F.: QEMU, A Fast and Portable Dynamic Translator, *Proc. Annual Conference on USENIX Annual Technical Conference, ATEC '05*, pp.41–46, USENIX Association (2005).

[26] ARM Architecture Reference Manual ARMRv7-A and ARMRv7-R edition (2007).

[27] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995).

[28] Gill, G.K. and Kemerer, C.F.: Cyclomatic Complexity Density and Software Maintenance Productivity, *IEEE Trans. Softw. Eng.*, Vol.17, No.12, pp.1284–1288 (online), DOI: 10.1109/32.106988 (1991).

[29] Scitools: Understand Static Code Analysis Tool, (online), available from ⟨http://scitools.com/⟩.

[30] Weicker, R.P.: Dhrystone: A Synthetic Systems Programming Benchmark, *Comm. ACM*, Vol.27, No.10, pp.1013–1030 (online), DOI: 10.1145/358274.358283 (1984).

[31] Hara, Y., Tomiyama, H., Honda, S., Takada, H. and Ishii, K.: CHStone: A benchmark program suite for practical C-based high-level synthesis, *IEEE International Symposium on Circuits and Systems, 2008, ISCAS 2008*, pp.1192–1195 (online), DOI: 10.1109/ISCAS.2008.4541637 (2008).

付 録

本付録では，命令動作記述について ARM，MIPS64，SH 向けの実例を示す．いずれの例にも一般的な 5 種類の命令分類，すなわち算術演算，論理演算，シフト演算，制御演算，ロード/ストアから 1 つずつ命令が含まれるように命令動作記述を抜粋した．

A.1 ARM 用記述

ARM の命令動作記述の抜粋を図 **A·1** に示す．ヘルパマクロ `DEFINST` の引数中の番号は，ARM のユーザーズマニュアル [26] で命令が記載されている節 A.8.6.n の番号 n と一致している．たとえば，`ARM138_A1_ROR__immediate_` は，A.8.6.138 に記載の ROR 命令と対応している．

各命令動作関数では，命令フィールド `cond` の値に基づいて実行条件を満たすか否かを判定する共通の if 文が含まれる．これが，ARM の命令動作記述量が MIPS64 や SH に比べて多くなる一因である．

ARM 用の共通マクロ `SetResult` は，変数捕捉を利用したマクロになっており，`result`，`carry`，引数の値をそれぞれ宛先レジスタ，キャリーフラグ，オーバフローフラグへと反映させる．

関数 `SetR/GetR` は汎用レジスタのアクセッサである．ARM の汎用レジスタ 15 は，プログラムカウンタであるため，関数 `GetR` では，第 1 引数でレジスタ 15 が指定された場合，予約変数 `m_PC` を読み出す．また，関数 `SetR` では，汎用レジスタ 15 が指定された場合，予約変数 `m_NextPC` に書き込む．ARM の条件付き命令のうち汎用レジスタ 15 が宛先レジスタになりうる命令は，条件分岐命令と見なせる．このため，関数 `SetPC` では，レジスタ 15 が指定された場合，予約変数 `m_BranchResult` も true に更新する．

A.2 MIPS64 用記述

MIPS64 用の命令動作記述の抜粋を図 **A·2** に示す．BLTZ の命令動作関数で使用しているサブ関数 `Branch` は，他の条件分岐命令と共通の関数である．関数 `Branch` は，第 1 引数が非ゼロの場合に予約変数 `m_BranchResult` を true に更新し，予約変数 `m_NextPC` に分岐先アドレスを設定する．

A.3 SH 用記述

SH 用の命令動作記述の抜粋を図 **A·3** に示す．

© 2016 Information Processing Society of Japan

```
1  DEFINST(ARM006_A1_ADD__register_)
2  {
3    if (IsConditionPassed(cond)) {
4      uint32_t shift_t;
5      uint32_t shift_n = DecodeImmType(type, imm5, &
           shift_t);
6      uint32_t shifted = Shift(GetR(Rm), shift_t, shift_n,
           APSR.C);
7      uint32_t carry;
8      uint32_t overflow;
9      uint32_t result = AddWithCarry(GetR(Rn), shifted,
           0U, &carry, &overflow);
10     SetResult(overflow);
11   }
12 }
13
14
15 DEFINST(ARM012_A1_AND__register_)
16 {
17   if (IsConditionPassed(cond)) {
18     uint32_t carry;
19     uint32_t shift_t;
20     uint32_t shift_n = DecodeImmType(type, imm5, &
           shift_t);
21     uint32_t shifted = Shift_C(GetR(Rm), shift_t, shift_n
           , APSR.C, &carry);
22     uint32_t result = GetR(Rn) & shifted;
23     SetResult(APSR.V);
24   }
25 }
26
27
28 DEFINST(ARM138_A1_ROR__immediate_)
29 {
30   if (IsConditionPassed(cond)) {
31     uint32_t shift_t;
32     uint32_t shift_n = DecodeImmType(SRType_ROR,
           imm5, &shift_t);
33     uint32_t carry;
34     uint32_t result = Shift_C(GetR(Rm), SRType_ROR,
           shift_n, APSR.C, &carry);
35     SetResult(APSR.V);
36   }
37 }
38
39
40 DEFINST(ARM025_A1_BX)
41 {
42   if (IsConditionPassed(cond)) {
43     SetArmPC(GetR(Rm));
44   }
45 }
46
47
48 DEFINST(ARM058_A1_LDR__immediate__ARM_)
49 {
50   if (IsConditionPassed(cond)) {
51     uint32_t imm32 = imm12;
52     uint32_t offset_addr = U ? GetR(Rn) + imm32 :
           GetR(Rn) − imm32;
53     uint32_t address = P ? offset_addr : GetR(Rn);
54     uint32_t data = Load32(address);
55     if (!P | W) {
56       SetR(Rn, offset_addr);
57     }
58     SetR(Rt, data);
59   }
60 }
```

図 A·1 ARM 用命令動作記述抜粋

Fig. A·1 Excerpt of instruction behavior description for ARM.

```
1  DEFINST(ADD)
2  {
3      GPR[rd] = SignExtend32(GPR[rs] + GPR[rt]);
4  }
5
6  DEFINST(AND)
7  {
8      GPR[rd] = GPR[rs] & GPR[rt];
9  }
10
11 DEFINST(SLLV)
12 {
13     GPR[rd] = SignExtend32(GPR[rt] << (GPR[rs] & 0
           x1F));
14 }
15
16 DEFINST(BLTZ)
17 {
18     Branch(Signed(GPR[rs]) < 0, offset);
19 }
20
21 DEFINST(LW)
22 {
23     GPR[rt] = SignExtend32(Load32(GPR[base] +
           SignExtend16(offset)));
24 }
```

図 A·2 MIPS64 用命令動作記述抜粋

Fig. A·2 Excerpt of instruction behavior description for MIPS64.

```
1  DEFINST(ADD)
2  {
3      R[n] += R[m];
4  }
5
6  DEFINST(AND)
7  {
8      R[n] &= R[m];
9  }
10
11 DEFINST(SHLD)
12 {
13   if (int32_t(R[m]) > 0) {
14     R[n] <<= (R[m] & 0x1F);
15   }
16   else if ((R[m] & 0x1F) == 0) {
17     R[n] = 0;
18   }
19   else {
20     R[n] >>= ((~R[m] & 0x1F) + 1);
21   }
22 }
23
24 DEFINST(BT)
25 {
26   if (m_BranchResult = (T == 1)) {
27     m_NextPC = m_PC + 4 + (sext8(d) << 1);
28   }
29 }
30
31 DEFINST(MOVLL)
32 {
33     R[n] = Load32(R[m]);
34 }
```

図 A·3 SH 用命令動作記述抜粋

Fig. A·3 Excerpt of instruction behavior description for SH.

© 2016 Information Processing Society of Japan

奥田 勝己 （正会員）

1980 年生．2003 年立命館大学理工学部情報学科卒業．2005 年同大学大学院理工学研究科情報システム学専攻修士課程修了．同年三菱電機株式会社入社．現在，同社先端技術総合研究所で組込みシステム開発技術に関する研究開発に従事．ACM 会員．

竹山 治彦 （正会員）

1971 年生．1993 年大阪市立大学工学部電気工学科卒業．1995 年同大学大学院工学研究科電気工学専攻修士課程修了．同年三菱電機株式会社入社．現在，同社先端技術総合研究所で組込みシステムに関する研究開発に従事．電子情報通信学会会員．

© 2016 Information Processing Society of Japan