Vol.2010-EMB-19 No.6
2010/12/6

組込みソフトウェア開発支援のための命令セットシミュレータ (ISS) 作成手法の提案

東園修平[†1]　片山徹郎[†1]

組込みシステムの開発は、開発期間短縮のためハードウェアとソフトウェアの開発を並行して行うことが多い。その場合、ソフトウェアのテスト時にハードウェアが存在せず、ソフトウェアのテストが実施できないという状況が生じる。この状況の解決策の 1 つとして命令セットシミュレータ (ISS) を用いることがあるが、ISS の開発には手間がかかってしまうという問題がある。そこで本論文では、ISS 作成に伴う手間を削減するために、ISS 作成手法を提案する。提案する ISS 作成手法は、ISS 作成対象の命令セットアーキテクチャ(ISA) の情報を「ISA 定義ファイル」として記述し、記述した ISA 定義ファイルから ISS のソースコードを生成する。また、提案する ISS 作成手法の実現性と有用性を確認するために、提案する ISS 作成手法を実装し、実際に ISS を作成した。その結果、ISS 作成に必要なコード行数を 25.39%削減できた。

Proposal of a Generation Method of an Instruction Set Simulator to Support Embedded Software Development

Syuhei Higashizono[†1] and Teturo Katayama[†1]

The embedded systems often develop hardware and software in parallel to reduce their development time. In that case, the situation in which software cannot be tested may happen because hardware does not exit when the software is tested. An ISS(Instruction set simulator) as one solution to this situation sometimes is used, but it takes much time to develop an ISS. This paper proposes a generation method of an ISS to reduce the effort in generating an ISS. The proposed method defines "ISA Definition File" which is described information on ISA(Instruction Set Architecture), and generates the source code of the ISS from the ISA Definition File. Moreover, in order to confirm realization and usefulness of the proposed method, it is implemented and generated an ISS. As a result, lines of code needed to develop the ISS can be reduced 25.39%.

1. はじめに

組込みシステムは、産業用、家庭用と至るところに存在し、現代社会には無くてはならないものとなっている。通常、組込みシステムの開発は、開発期間を短縮するためにハードウェアの開発と並行してソフトウェアの開発を行うことが多い[1)]。開発したソフトウェアをテストするためには、ハードウェアが必要となる。しかし、並行して開発を行っているために、テスト時にハードウェアが存在しない状況が多く、ソフトウェアのテストが実施できないという問題が生じる。

この問題の解決策の 1 つとして、開発しているハードウェアのマイクロプロセッサの動作をシミュレートする命令セットシミュレータを用いることがある。命令セットシミュレータを用いることにより、実機が存在しなくてもソフトウェアのテストを実施できる。このことは、ソフトウェアテストを実施するタイミングを早めることができるので、結果として開発期間の短縮に繋がる。

しかし、命令セットシミュレータを作成するためには、作成対象のマイクロプロセッサの情報である命令セットアーキテクチャの知識が必要であるため、手間がかかる。

そこで本論文では、命令セットシミュレータの作成にかかる手間の削減を目的とし、命令セットアーキテクチャから命令セットシミュレータのソースコードを生成する手法を提案する。提案する手法は、作成対象とする命令セットシミュレータのマイクロプロセッサの命令セットアーキテクチャから、作成に必要な情報を「ISA(命令セットアーキテクチャ) 定義ファイル」として記述し、記述した ISA 定義ファイルから命令セットシミュレータのソースコードを生成する。

また、提案する手法の実現性と有用性を確認するために、C♯[2)] と Visual Studio 2005[3)] を用いて提案する手法を実装し、実際に COMETII[4)] の命令セットシミュレータを作成し考察を行う。

以下、本論文の構成は次の通りである。

2 章では、命令セットシミュレータについて説明する。3 章では、提案する命令セットシミュレータ生成手法について説明する。4 章では、提案した手法を実装し、実際に COMETII[4)] の命令セットシミュレータを作成する。5 章では、提案した手法について考察を行う。

†1 宮崎大学
University of Miyazaki

Vol.2010-EMB-19 No.6
2010/12/6

2. 命令セットシミュレータ

命令セットシミュレータ (Instruction set simulator, 以下 ISS と記す) とは、マイクロプロセッサの動作をシミュレートするプログラムのことである。この章では、ISS の作成に必要な命令セットアーキテクチャやその用途について説明する。

2.1 命令セットアーキテクチャ

マイクロプロセッサ上で利用できる命令コードの一覧のことを、命令セットと呼ぶ。命令セットはマイクロプロセッサごとに異なる。命令セットにプログラマ側から見たレジスタの構成やアドレッシングモード、メモリアーキテクチャ等を加えたものを命令セットアーキテクチャ(Instruction set architecture, ISA 以下、ISA と記す) と呼ぶ。

提案する手法では、ISS の作成対象であるマイクロプロセッサの ISA を ISA 定義ファイルとして記述し、ISA 定義ファイルから ISS のソースコードを生成する。

2.2 用　　途

ISS は、主に以下の用途に用いられる。

- ハードウェアが存在しない状況下での動作検証
  通常、組込みシステムの開発では、開発期間短縮のためハードウェアとソフトウェアの開発が並行して行われる。そのためソフトウェアのテスト時にハードウェアが存在しない状況がある。そのような状況下でテストを実施するための手段の 1 つとして、ISS を用いる。
- 実機とは異なるハードウェア用のプログラムの実行
  ある特定のハードウェア用に作られたプログラムは、一般に他のハードウェア上では動かない。この問題の解決策の 1 つとして、ISS を用いることがある。

3. ISS 作成手法の提案

本論文では、ISS の作成にかかる手間の削減を目的とし、ISS のソースコードを生成する手法を提案する。一般に、ISS の作成は、作成対象のマイクロプロセッサの ISA の情報を元に作成する。そこで提案手法では、作成対象のマイクロプロセッサの ISA の情報を「ISA 定義ファイル」として記述し、この記述した ISA 定義ファイルから ISS のソースコードを生成する。

提案する手法の流れを、図 1 に示す。提案する手法は、ISS 作成対象のマイクロプロセッサの ISA の情報を ISA 定義ファイルの書式に従って記述し、ISA 定義ファイルから ISS のソースコードを生成する。ISA 定義ファイルは、仕様記述部、レジスタ構成記述部、オペランド記述部、命令フォーマット記述部、コード記述部の 5 つで構成する。

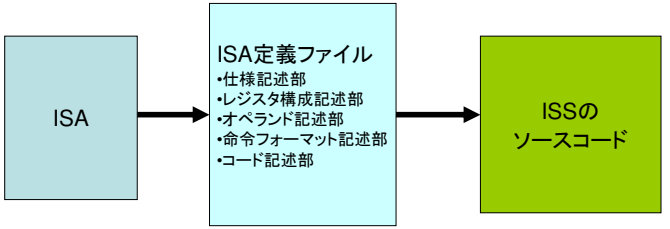

図 1 ISS 作成手法の流れ

リスト 1　ISA 定義ファイルテンプレート

```
1  <作成するISSのソースコードの名前>
2  WORD 1語長の長さ
3  MSIZE 主記憶の容量
4  FETCH 最初のフェッチで取ってくる語数
5  OPCODE オペコードの長さ
6  <REGISTER>
7  レジスタ名_サイズ:個数
8  <OPERAND>
9  オペランド名_サイズ:オペランドマスク 値
10 <FORMAT>
11 [命令フォーマット名]:語数 命令の構成
12 <CODE>
13 [命令の名前]命令モード指定番号
14 処理
15 オペコード:命令のフォーマット 引数の設定
16 <END>
```

以下、ISA 定義ファイルとソースコードの生成について説明する。

3.1 ISA 定義ファイル

この節では、ISA 定義ファイルについて説明する。

リスト 1 に提案する ISA 定義ファイルのテンプレートを、リスト 2 に記述例を、それぞれ示す。以下、このテンプレートに沿って説明する。

3.1.1 仕様記述部

リスト 1 のテンプレートにおける 1〜5 行目までは仕様記述部であり、記述項目とその順序は固定である。まず、1 行目には、⟨⟩ に挟まれた間にアルファベット大文字 2 文字以上で「作成する ISS の名前」を記述する。2 行目には、「WORD」の後にスペースを入れ、1 語長の長さを 32、16、8 のいずれかで記述する。3 行目には、「MSIZE」の後にスペースを入れ、

Vol.2010-EMB-19 No.6
2010/12/6

リスト 2　ISA 定義ファイルの記述例

```
1  <TEST>
2  WORD 8
3  MSIZE 0x100
4  FETCH 1
5  OPCODE 4
6  <REGISTER>
7  GR_8:4
8  SF_1:1
9  <OPERAND>
10 r_2:1 GR0,GR1,GR2,GR3
11 x_2:0 0,GR1,GR2,GR3
12 adr_8:0 IMM
13 <FORMAT>
14 [op_r]:1 op r
15 [op_adr_x]:2 op -- x adr
16 [op_r_adr_x]:2 op r x adr
17 <CODE>
18 [LD]6
19 Rs -> Rd
20 SF = Rr.7
21 0x1:op_r_adr_x Rd: r Rs: ( adr + x )
22 <END>
```

主記憶の容量を 10 進数または 16 進数 (16 進数の場合は 0x を付けて) で記述する。4 行目には、「FETCH」の後にスペースを入れ、最初のフェッチで取ってくる語数を記述する。5 行目には、「OPCODE」の後にスペースを入れ、オペコードの長さをビット数で指定する。

リスト 2 の例では、1 行目の「⟨*TEST*⟩」は、作成する ISS の名前が「TEST」であることを示している。2 行目の「WORD 8」は、1 語長の長さが 8 ビットであること示している。3 行目の「MSIZE 0x100」は、主記憶の容量が 256 語であることを示している。4 行目の「FETCH 1」は、最初のフェッチで 1 語取ってくることを示している。5 行目の「OPCODE 4」は、オペコードの長さが 4 ビットであることを示している。

3.1.2　レジスタ構成記述部

リスト 1 のテンプレートにおける 6 行目以降はレジスタ構成記述部であり、⟨REGISTER⟩ 以下の行に、レジスタの構成を記述する。レジスタの記述は、レジスタ名はアルファベット大文字 2 文字以上、レジスタのサイズはビット数で、かつ 1 語長の長さ以下で記述し、個数は 10 進数で「レジスタ名_レジスタのサイズ:個数」のように記述する。レジスタのサイズを 1 ビットと記述した場合、そのレジスタをフラグレジスタとして扱う。

リスト 2 の例では、6 行目の「⟨REGISTER⟩」から、レジスタ記述部であることを示し、7 行目の「GR_16:8」は、「8 ビットで名前が GR のレジスタを 4 個」、8 行目の「SF_1:1」は、「1 ビットで名前が SF のレジスタを 1 個」というレジスタの構成を示している。

表 1　オペランドマスクの値と意味

値	内容
0	読み込みのみ可能
1	読み込み、書き込み可能
2	書き込みのみ可能

3.1.3　オペランド記述部

リスト 1 のテンプレートにおける 8 行目以降はオペランド記述部であり、⟨OPERAND⟩ 以下の行に、オペランドを記述する。オペランド記述部は、まずオペランドの名前をアルファベット小文字または数字 1 文字以上で記述する。この際、先頭は必ずアルファベット小文字でなければならない。オペランドの名前の後に続けて「_」を記述後、オペランドのサイズをビット数で記述する。次に「:」を記述後、オペランドマスクを記述する。オペランドマスクに使用できる値を、表 1 に示す。その後、スペースを入れ、オペランドの値を記述する。オペランドの値として記述可能なのは、10 進数の数字、またはレジスタ定義部で設定したレジスタであり、区切り文字として「,」を使用する。なお、オペランドの値をそのまま使用するイミディエイトデータの場合には「IMM」と記述する。

リスト 2 の例では、9 行目の「⟨OPERAND⟩」からオペランド記述部であることを示し、10 行目の「r_2:1 GR0, GR1, GR2, GR3」は、サイズが 2 ビットで名前が「r」であり、r が 0 の時 GR0、r が 1 の時 GR1、r が 2 の時 GR2、r が 3 の時 GR3 であるというオペランドを設定している。11 行目の「x_2:0 0, GR1, GR2, GR3」は、サイズが 2 ビットで名前が「x」であり、x が 0 の時 0、x が 1 の時 GR1、x が 2 の時 GR2、x が 3 の時 GR3 であるというオペランドを設定している。12 行目の「adr_16:0 IMM」は、サイズが 8 ビットで名前が「adr」のイミディエイトデータであるオペランドを設定している。

3.1.4　命令フォーマット記述部

リスト 1 のテンプレートにおける 10 行目以降は命令フォーマット記述部であり、⟨FORMAT⟩ 以下の行に、命令のフォーマットを記述する。命令フォーマット記述部は、まず [] に囲まれた区間に命令フォーマットの名前を記述する。命令フォーマットの名前はアルファベット小文字、数字、または「_」を組み合わせた 2 文字以上で記述する。この際、先頭は必ずアルファベット小文字でなければならない。次に「:」を記述後、語数を記述する。その後、命令フォーマットの構成を記述する。命令フォーマットの構成を記述する際、先頭は必ずオペコードを意味する「op」でなければならない。「op」記述後、オペランド記述部で設定したオペランドを記述する。オペランドとして使用しない箇所には「-」をビッ

表 2 定義例の命令フォーマット

フォーマット	語数	1 語目 7〜4 ビット	1 語目 3〜2 ビット	1 語目 1〜0 ビット	2 語目
op_r	1	op	r	-	-
op_adr_x	2	op	-	x	adr
op_r_adr_x	2	op	r	x	adr

ト数分記述する。なお、「-」が命令フォーマットの最後尾にくる場合には省略可能である。

リスト 2 の例では、13 行目の「⟨FORMAT⟩」から命令フォーマット記述部であることを示している。14 行目の「[op_r]:1 op r」は、命令フォーマットの名前が「op_r」で 1 語で構成し、仕様記述部でオペコードの長さを 4 ビットと定義しているため、1 語目の先頭 4 ビットがオペコード「op」であり、次の 2 ビットがオペランド「r」であることを示している。15 行目の「[op_adr_x]:2 op - - x adr」は、命令フォーマットの名前が「op_adr_x」で 2 語で構成し、1 語目の先頭 4 ビットがオペコード「op」、次の 2 ビットは何も使用せず、次の 2 ビットがオペランド「x」、2 語目の先頭から 8 ビットはオペランド「adr」であることを示している。16 行目の「[op_r_adr_x]:2 op r x adr」は、命令フォーマットの名前が「op_adr_x」で 2 語で構成し、1 語目の先頭 4 ビットがオペコード「op」、次の 2 ビットはオペランド「r」、次の 2 ビットがオペランド「x」、2 語目の先頭から 8 ビットはオペランド「adr」であることを示している。表 2 に、この例で定義した命令フォーマットを示す。

3.1.5 コード記述部

リスト 1 のテンプレートにおける 12 行目以降は、⟨CODE⟩ 以下の行に、各命令の処理とオペコードを記述する。コード記述部は、まず [] に囲まれた区間に命令の名前をアルファベット大文字 2 文字以上で記述し、命令モード番号を設定する。命令モード番号は 3 ビットのマスク値であり、最上位ビットが 2 番、最下位ビットが 0 番である。命令モード番号の各ビットの意味を、表 3 に示す。次の行から命令の処理を記述する。命令の処理に記述できる要素を、以下に示す。

引数

引数が存在する場合、引数を表す「Rd」、「Rs」を使用できる。「Rd」はデスティネーションオペランドを示し、「Rs」はソースオペランドを示す。符号付きで使用する場合は「RdA」、「RsA」と記述する。また、命令モードの 2 ビット目が立っているときは Rd の変化後の値を表す「Rr」も使用できる。

演算子

11 種類の演算子を使用できる。使用できる演算子とその意味を、表 4 に示す。

ビットチェック

Rd、Rs、Rr は、各要素の右に「.n(n は十進数の数字)」を付けることによって、要素の右から n ビット目が立っているかを確認できる。例として、「Rd.2」ならば、Rd の 2 ビット目が立っているかどうかを確認できる。また「!」を Rd、Rs、Rr 前に付けることによって、ビットが立っているかどうかの結果を反転できる。

フラグレジスタ設定用要素

フラグレジスタに値を代入するために、「true」と「false」を使用できる。

フラグレジスタ用演算子

フラグレジスタ用の演算子として、AND 演算子「&&」、OR 演算子「||」、フラグに値を設定する「=」を使用できる。

レジスタ

レジスタを記述する場合は、レジスタ構成記述部で記述したレジスタ名を記述できる。なお、プログラムカウンタを使用する場合は「PC」と記述する。

条件分岐

条件分岐の記述のため、「if」、「if else」、「else」を使用できる。

命令の処理の記述後、オペコードの先頭に 0x を付け 16 進数で記述し、「:」を記述後、命令のフォーマットを記述する。この際、引数がある場合はスペースを空け引数を記述する。引数の設定には、命令フォーマットのオペランドが使用できる。また、オペランドを「()」で囲むと () 内の値が示す主記憶の番地の値を示す。

リスト 2 の例では、17 行目の「⟨CODE⟩」からコード記述部であることを示し、18 行目の「[LD]6」は命令の名前が「LD」であり、命令のモードは、ビット番号 1 と 2 が立っていることを示しており、19 行目の「Rs -⟩ Rd」は Rs の値を Rd に代入することを示している。20 行目の「SF = Rr.7」は、Rs の値を代入した Rd の左から 7 ビット目が立ってるかどうかの結果を、レジスタ「SF」に格納することを示している。21 行目の「0x1:op_r_adr_x Rd: r Rs: (adr + x)」は、オペコードが「0x1」、命令フォーマットが「op_r_adr_x」、引数 Rd に「r」を、Rs に「 (adr + x)」を設定することを示している。

3.2 制　　限

今回提案する手法には、現状いくつかの制限がある。以下にその制限を記す。

プログラムカウンタ

プログラムカウンタは、32 ビット固定である。

多語長演算の未対応

表 3 命令モード番号

ビット番号	意味
0	引数として Rd を持ち、命令の処理に利用する。
1	引数として Rs を持ち、命令の処理に利用する。
2	引数として Rd を持ち、命令の処理の結果を Rd に戻す。

表 4 コード記述部で使用できる演算子

演算子	説明	演算子	説明
+	足し算	-	引き算
〈〈	右シフト	〉〉	左シフト
-〉	代入	&	AND 演算
|	OR 演算	〉	大なり
〉=	以上	〈	小なり
〈=	以下		

仕様記述部「WORD 1 語長の長さ」で設定した 1 語長の長さでしか演算を取り扱えない。

予約語

予約語を以下に示す。予約語を各要素の名前に付けることはできない。

- 「PC」
- ISS のソースコードを生成する言語の予約語

入出力関数

生成する ISS のソースコードは、レジスタや主記憶用の入出力関数を持たない。そのため、ユーザが追加する必要がある。

4. 提案手法の実装と実行例

これまでに述べた手法の実現性を確認するために、C♯[2] と Visual Studio 2005[3] を用いて提案した手法を実装し、実際に ISS を記述し動作の確認を行った。

今回は、動作確認のために COMETII[4] の ISS を作成した。

4.1 ISA 定義ファイル

3.1 節で提案した ISA 定義ファイルテンプレートに基づいて、COMETII の命令セットアーキテクチャ定義ファイルを作成した。

ISS 作成支援ツールの入力となる COMETII の ISA 定義ファイルは、193 行となった。リスト 3 に、COMETII の ISA 定義ファイルの一部を示す。

リスト 3 COMETII の ISA 定義ファイルの一部

```
1  <COMETII>
2  WORD 16
3  MSIZE 0x10000
4  FETCH 1
5  OPCODE 8
6  <REGISTER>
7  GR_16:8
8  SP_16:1
9  OF_1:1
10 ZF_1:1
11 SF_1:1
12 <OPERAND>
13 r_4:1 GR0, GR1, GR2, GR3, GR4, GR5, GR6, GR7
14 r1_4:1 GR0, GR1, GR2, GR3, GR4, GR5, GR6, GR7
15 r2_4:1 GR0, GR1, GR2, GR3, GR4, GR5, GR6, GR7
16 x_4:0 0, GR1, GR2, GR3, GR4, GR5, GR6, GR7
17 adr_16:0 IMM
18 <FORMAT>
19 [op]:1 op
20 [op_r]:1 op r
21 [op_r1_r2]:1 op r1 r2
22 [op_adr_x]:2 op ---- x adr
23 [op_r_adr_x]:2 op r x adr
24 <CODE>
25 [NOP]0
26 0x00:op
27
28 [LD]6
29 Rs -> Rd
30 ZF = !Rr.15 && !Rr.14 && !Rr.13 && !Rr.12 && !Rr.11 && !Rr.10 && !Rr.9 && !Rr.8 && !Rr.7
       && !Rr.6 && !Rr.5 && !Rr.4 && !Rr.3 && !Rr.2 && !Rr.1 && !Rr.0
31 SF = Rr.15
32 OF = false
33 0x10:op_r_adr_x Rd: r Rs: ( adr + x )
34 0x14:op_r1_r2 Rd: r1 Rs: r2
```

4.1.1 動 作 確 認

今回提案した ISS 作成手法を用いて作成した COMETII の ISS の動作を確認するために、情報処理推進機構が配布している COMETII のシミュレータ[6] と比較する。今回提案した ISS 作成手法によって、ISA 定義ファイルから生成した COMETII のソースコードに文字を出力する命令を追加した。また、COMETII のためのアセンブラ言語である CASLII のアセンブラを作成した。比較のために、ハノイの塔を解くプログラムを使用した。

図 2 と図 3 に、それぞれの実行結果の出力画面を示す。それぞれの実行結果から、両者が一致していることが確認できる。

よって、提案した ISS 作成手法を用いて、実際に ISS を作成できることが確認できた。

5. 考　　察

4 章では、今回提案した手法を用いることによって、ISS を実際に作成できることが確認できた。この章では、今回提案した ISS 作成手法を用いることによって、ISS 作成にかかる

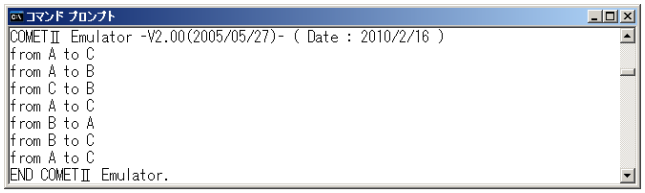

図 2 IPA の COMETII シミュレータの実行例

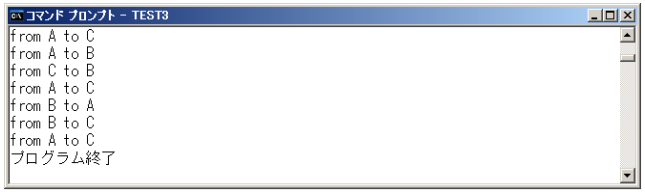

図 3 ISS 作成手法を用いて作成した COMETII シミュレータの実行例

手間の削減ができることを確認するために、提案した ISS 作成手法を使用せずに ISS を作成した場合との比較を行う。具体的には、ISS 作成のためのコード行数の削減率について述べる。また、関連研究と問題点についても述べる。

5.1 コード行数の削減率

今回提案した ISS 作成手法が、ISS 作成にかかる手間の削減ができることを確認するために、提案した手法を使用せずに ISS を作成した場合との比較を行う。この比較のために、COMETII の ISS を、今回提案した手法を使用せずに C♯ で記述した。COMETII の ISS の作成において、今回提案した ISS 作成手法を使用した場合と使用しなかった場合の記述に必要なコード行数を、表 5 に示す。

ここで、ISS 作成手法を使用した場合の行数は、「ISA 定義ファイルの行数」＋「動作確認のために追加したソースコードの行数」である。今回提案する手法は、ISS のソースコード生成後に、入出力関数のコードを追加する必要がある。すなわち、動作確認をするためには、入出力関数のコードを記述する必要がある。上記の「動作確認のために追加したソースコードの行数」とは、この記述した入出力関数のコードの行数である。

表 5 より、今回提案する手法を使用した場合は使用しなかった場合と比較して、COMETII の ISS を作成するために必要な行数を、25.39%削減できたことが分かる。

よって、今回提案した手法によって、ISS 作成にかかる手間の削減ができたと言える。

表 5 COMETII 作成のための行数と削減率

ISS 作成手法未使用の場合の行数	382
ISS 作成手法使用の場合の行数	285
ISA 定義ファイルの行数	193
動作確認のために追加したソースコードの行数	92
削減率 (%)	25.39

5.2 関 連 研 究

命令セットアーキテクチャの記述から ISS を生成するツールの 1 つとして、Campinas 大学で開発された ArchC[7] がある。ArchC は SystemC[8] に基づくソフトウェア開発環境構築ツールであり、アーキテクチャの記述からアセンブラ、リンカ、シミュレータといったソフトウェアの開発環境を構築することができる。ArchC を利用するためには、SystemC を学ぶ必要がある。これに対して、今回提案した ISS 作成手法は、命令セットアーキテクチャの仕様のデータをそのまま入力として使うので、SystemC を学ぶ必要がなくその分の手間を削減できる。

また、プロセッサ開発環境の一部として、ISS を生成する ASIP Meister[9] がある。ASIP Meister は特定用途向けプロセッサ開発ツールであり、プロセッサの仕様記述からプロセッサのハードウェア記述言語や SystemC 記述、プログラムの開発環境であるアセンブラやデバッガ、ISS を自動生成する。ASIP Meister はプロセッサの開発環境であるため、命令の動作の記述にパイプライン中のデータ処理を記述する言語を使用する。これに対して、今回提案した ISS 作成手法は、ISS の作成のみに重点を置いているため、命令の動作の記述にパイプライン中のデータ処理など複雑な記述は必要としない。そのため ISS を作成するだけならば、今回提案した ISS 作成手法の方が容易に ISS を作成することができる。

5.3 問 題 点

以下、今回提案した ISS 作成手法の問題点を挙げる。

- 未対応のレジスタ
  今回提案した ISS 作成手法は、16 ビットのレジスタ 1 個を 8 ビットのレジスタ 2 個のように扱うようなレジスタに対応していない。1 つの長いレジスタを複数の短いレジスタとして扱える CPU が多いため、ISS 作成支援ツールを改良してこれに対応し、ツールの適用範囲を拡大する必要がある。
- 多語長演算への対応
  今回提案した ISS 作成手法では、多語長演算に対応していない。多語長演算が行える

CPU は多いため、提案する ISS 作成手法でも対応する必要がある。

- 時間の概念
  今回提案した ISS 作成手法を用いて作成した ISS は、命令の実行時間をシミュレートしない。そのため、実機で実行した場合と ISS で実行した場合でプログラムの実行時間にずれが生じる可能性がある。組込みシステムでは時間の概念が重要であるため、実行時間をシミュレートする必要がある。

6. おわりに

本論文では、ISS の作成にかかる手間の削減を目的とし、ISA から ISS のソースコードを生成する ISS 作成手法を提案した。

今回提案した ISS 作成手法の実現性を確認するために、実際に C♯ と Visual Studio 2005 を用いて COMETII の ISS を作成し、動作を確認した。また、今回提案した手法の有効性を示すために、提案した手法を使用せずに作成した ISS と比較し、コード行数の削減率を調べた。削減率により、今回提案した手法を用いることによって、ISS 作成の手間が削減できることを確認した。

ISS 作成の手間が削減できることは、ソフトウェアテストのタイミングを早めることができ、組込みソフトウェアの開発期間の短縮に繋がる。

以下に、今後の課題を挙げる。

- 未対応のレジスタ
- 多語長演算への対応
- 時間の概念

参考文献

1) 社団法人日本システムハウス協会エンベデッド技術者育成委員会, 「エンベデッドシステム開発のための組込みソフト技術」, 電波新聞社 (2005).
2) Herbert Schildt(矢嶋 聡 監修/株式会社テック・インデックス 訳),「独習 C♯ 第二版」, 翔泳社 (2007).
3) Micorsof 社, 「Visual Studio 2005」, http://www.microsoft.com/japan/msdn/vstudio/2005
4) 独立行政法人 情報処理推進機構,「試験で使用する情報技術に関する用語・プログラミング言語 ver 1.0」, http://www.jitec.ipa.go.jp/1_00topic/topic_20081027_hani_yougo.pdf
5) Microsoft 社,「.NET Framwork」, http://msdn.microsoft.com.ja-jp/netframework/
6) 独立行政法人 情報処理推進機構,「CASLII シミュレータ」, http://www.jitec.ipa.go.jp/1_20casl2/casl2dl_002.html
7) The Computer Systems Laboratory (LSC) of the Institute of Computing at the University of Campinas (IC-UNICAMP) ,「ArchC」, http://www.archc.org/
8) Open SystemC Initiative (OSCI), 「SystemC」, http://www.systemc.org/home/
9) 今井 正治, 武内 良典, 塩見 彰睦, 佐藤 淳, 北嶋 暁 「特定用途向きプロセッサ開発システム ASIP Meister」, Technical report of IEICE. DSP 102(399) pp.39-44 (2002).