設計自動化 62－25
(1992. 5.28)

# ASIC CPU向きソフトウェア開発環境生成系の実現

博多 哲也[*1], 佐藤 淳[*2], Alaudain Y. Alomary[*3], 今井 正治[*3], 引地 信之[*4]

[*1]熊本電波工業高等専門学校 電子制御工学科 [*2]鶴岡工業高等専門学校 電気工学科
[*3]豊橋技術科学大学 情報工学系 [*4]ＳＲＡ

あらまし 筆者らはこれまで, ASIC マイクロプロセッサ(ASIP : Application Specific Integrated Processor)開発システムPEASについて研究を行ってきた. また,現在その処理系のプロトタイプを試作している. 本稿では, PEASシステムの一部であるソフトウェア開発環境生成系の実現方法について述べる.ASIP開発システムはCPUコアの生成と同時にコンパイラやシミュレータなどのソフトウェア開発環境を生成する. ASIPの命令セットは設計ごとに異なりうる. 命令セットごとに応用プログラム開発環境を作成するのは, 開発環境自体の開発工数と開発時間が必要となり, 効率的な解決方法とは言えない. そこで, 応用プログラム開発環境を系統的に自動生成する方法を採用した. また, この方法に基づいてＣコンパイラの自動生成システムを実現した.

# A Software Development Tool Generator for ASIC CPU

Tetsuya Hakata[*1], Jun Sato[*2], Alaudain Y. Alomary[*3], Masaharu Imai[*3], Nobuyuki Hikichi[*4]

[*1]Dept. of Electrical Control
Kumamoto National College of Technology

[*2]Dept. of Electrical Engineering
Tsuruoka National College of Technology

[*3]Dept. of Info. and Comp. Sciences
Toyohashi University of Technology

[*4]Software Tools and Tech. Division
Software Research Associates, Inc.

Abstract An ASIP (Application Specific Integrated Processor) development system PEAS has been proposed by the authors. PEAS system synthesizes the CPU core design of the ASIP as well as the application program development tools, such as compiler and simulator. This paper describes the application program development tool generator for PEAS system. Because the instruction set of ASIP's can vary from design to design, it is required to implement application program development tools for each design of ASIP. The application program development tool generator described in this paper is supposed to generate C compiler, simulator, and assembler automatically, according to the instruction set being supported by the ASIP CPU core.

## 1．はじめに

近年の半導体集積回路の集積度の向上およびASIC(特定用途向き集積回路)開発用ツールの発達により，特定用途向きCPUコアを内蔵した大規模な ASIP (Application Specific Integrated Processor)の開発が可能になってきた．

筆者らは，ASIPの自動生成の方法について研究を行ってきた．また，現在ASIP自動生成系のプロトタイプPEAS(Practical Environment for ASIP development)を試作している[1][3]．PEASシステムは，まず，ある特定分野の応用プログラムを解析する．次に，解析結果をもとにしてCPUのアーキテクチャを決定する．決定されたアーキテクチャの情報をもとにCPUコアを生成する．

さらに，PEASシステムの特徴の１つは，ハードウェアの自動生成と同時にコンパイラ，アセンブラ，シミュレータなどの応用プログラム開発環境の生成を自動的に行うことである．

ASIP用の応用プログラム開発環境を実現する場合，次の問題点を解決する必要がある．ASIPは，設計ごとに命令セットが異なる可能性がある．したがって，従来は命令セットごとに異なる応用プログラム開発環境を実現していた．応用プログラム開発環境はそれ自体，開発工数も多く長期にわたる開発時間が必要である．

この問題を解決するために，PEASシステムでは，命令セットアーキテクチャの可変範囲をある程度限定し，応用プログラム開発環境の自動生成の容易化をはかっている．

本稿では，PEASシステムにおける応用プログラム開発環境の実現方法について述べる．

## 2．ASIP開発システムPEASについて

### 2.1 システムの概要

PEASシステムでは，ASIP開発システムの対象ユーザが次に示す条件を満たしていることを前提としている．

(1) ある特定分野の応用プログラムを持ち，そのプログラムの処理効率の向上を計りたい．

(2) 専用のCPUを開発する必要がある．

これらの条件を満足するためには，CPUコアの生成と同時にソフトウェア開発環境を用意することが重要になってくる．また，(2)については既存のCPUコアの使用ライセンス上の問題などから汎用のCPUコアの使用を避けたい場合を想定している．

### 2.2 PEASシステムの構成

PEASシステムでの処理の流れを図１に示す．PEASシステムでの処理の概要は以下の通りである．

(1) まず，ある特定分野の応用プログラムの集合を静的および動的に解析を行い，それらの応用プログラムにおけるデータ処理の特徴を抽出する．

(2) 解析の結果に基づいてCPUのハードウェア構成および命令セットなどのアーキテクチャを決定する．

(3) 決定されたアーキテクチャ情報に基づいてCPUコアの生成およびソフトウェア開発環境の生成を行う．

PEASシステムは，次の４つのサブシステムから構成される．

(1) 応用プログラム解析部[2]

C言語で記述された応用プログラムおよびプログラムの実行時に用いられるデータを入力し，静的および動的解析を行う．解析項目は，

・C言語レベルの演算子
・ライブラリ関数およびユーザ定義関数
・データ構造
・データのアクセス方法

などで，演算子，関数，およびデータの使用形態や使用頻度などの統計情報を収集する．

(2) アーキテクチャ生成部[3]

解析部で収集したデータをもとに，与えられた応用プログラムとデータの集合に対して処理効率が最大になるような命令セットとアーキテクチャを決定する．制約条件は，チップ面積と消費電力である．

アーキテクチャ生成部では，CPUコア生成部用の情報とソフトウェア開発環境生成部用の情報を出力する．

(3) CPUコア生成部

アーキテクチャ情報をもとにCPUコアの設計情報を出力する．使用する可能性のあるハードウェア・モジュールをハードウェア記述言語(HDL)で記述してライブラリ化しておき，必要に応じてリンクし，制御部を生成する．

生成されたHDLによるCPUコアは既存の高位論理合成システム(High-level synthesis system)を用いて制御部を含む実際のCPUコアを作成する．必要に応じて，論理レベルのシミュレーションなどを行う．

(4) 応用プログラム開発環境生成部

アーキテクチャ情報をもとに，UNIXマシン上で動作するＣコンパイラ，アセンブラ，シミュレータなどの応用プログラム開発環境（クロス環境）を生成する．[5]

開発された応用プログラムのオブジェクトコードは，ローダによってASIP応用システム上にダウンロードされ，実行される．

## 3．アーキテクチャモデル

### 3.1 アーキテクチャの特徴

PEASシステムで生成されるCPUはRISCタイプの32ビットCPUである．生成されるCPUの主な特徴を以下に示す．

(1) ハーバードアーキテクチャを採用する
(2) 命令長は32ビットに固定する
(3) 命令へのアクセスはload/store命令を用い，レジスタとメモリ間での演算は行わない
(4) 3アドレス形式の命令をもつ
(5) 汎用レジスタをもつ

次に，ASIPのアーキテクチャにおいて可変である項目は以下の通りである．

(1) 汎用レジスタの個数
(2) 命令の実現方式は以下の３通りの方法から選択される．
・ハードウェアによる実現
・補助命令の組み合わせによる実現
・マイクロプログラムによる実現
(3) アドレスバスの幅

### 3.2 命令セットモデル

#### 3.2.1 ファンクショナリティ

Ｃ言語において，演算子と関数ははっきりと区別されている．しかし，PEASシステムが生成するCPUコアはハードウェアの自由度が高く，C言語の演算子と関数を実現方法によって区別することが困難である．例えば，'+'(加算演算子)は，演算対象が固定小数点数の場合にはハードウェア（ALU)で実現される．しかし，演算対象が浮動小数点数の場合にはハードウェアまたはソフトウェアで実現できる．また，三角関数なども専用の演算器によって実現可能である．

したがって，演算子と関数とを区別せず，統一して扱った方が議論しやすい．そこで，「ファンクショナリティ」をＣ言語の演算子と関数を総称する概念として定義する．[3]

PEASシステムが生成するCPUコアの命令セットはハードウェアあるいはマイクロプログラムによって実現されたファンクショナリティのみからなる．ハードウェアおよびマイクロプログラムによって実現されなかったファンクショナリティは，命令セット中に含まれるファンクショナリティを組み合わせて実現する．すなわち，これらのファンクショナリティは，ソフトウェアにより，ライブラリ，あるいはランタイムルーチンとして実現する．

#### 3.2.2 命令セットの階層化

PEASシステムでは，Ｃコンパイラを生成するためＣコンパイラの特徴を生かす命令セットを考える必要がある．まず，命令セットをＣ言語の演算子に対応するファンクショナリティと関数に対応するファンクショナリティに分類した．これは以下の理由による．

(1) 演算子の集合はＣ言語の構文規則によってあらかじめ与えられているので可能な実現方法を前もって決定できる．
(2) Ｃコンパイラは関数に対応する命令を生成するよりも演算子に対応する命令を生成する方が容易である．
(3) 関数に対応する命令はユーザ定義関数も含まれるのであらかじめ可能な実現方法を決定できない．

次に，演算子に対応するファンクショナリティを原始的(primitive)なファンクショナリティと基本的(basic)なファンクショナリティに分類した．原始ファンクショナリティはハードウェアで実現される．基本ファンクショナリティは，直接ハードウェアで実現することも可能であるが，原始ファンクショナリティを組み合わせることによってソフトウェアで実現してもよい．

以下に，３つの階層に分類した命令セットを示す．

(1) 原始ファンクショナリティ
Ｃ言語の演算子に対応するファンクショナリティのうちALU，1ビットシフタ，必要最小限の制御回路などのハードウェアで直接実現できるファンクショナリティが含まれる．

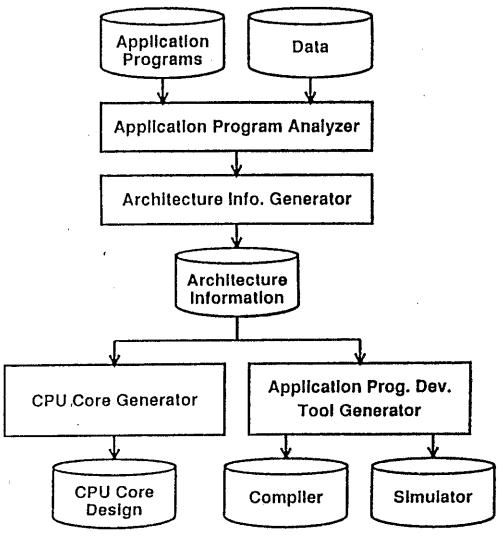


図　1　PEASシステムの構成

(2) 基本ファンクショナリティ
C言語の演算子に対応するファンクショナリティのうち原始ファンクショナリティ以外のものが含まれる.
(3) 拡張ファンクショナリティ
実行頻度が高いと思われるライブラリ関数およびユーザ定義関数に対応するファンクショナリティが含まれる.

## 4. Cコンパイラの実現

### 4.1 実現方法

PEASシステムが生成するCコンパイラはGNU Cコンパイラ (GCC)[4]をベースにしている. GCCは移植性にすぐれ, ターゲットマシンの変更が容易な設計になっている. ターゲットマシンを変更する場合には, 次のファイルを変更すればよい.

(1) GCCの中間言語 RTL ( Register Transfer Language)とターゲットとなるCPUの命令との対応関係を記述するファイル「md」
(2) アーキテクチャについて記述するヘッダファイル「tm.h」

この2つのファイルを生成する方法について, 4.3節および4.4節で述べる.

Cコンパイラ生成系の構成を図2に示す.

### 4.2 命令セット

PEASシステムが生成するプロセッサの命令セットはGCCの中間言語RTLをベースとしている. 命令セットは第3節で述べた基準に基づいて PRTL(Primitive RTL), BRTL(Basic RTL), XRTL(eXtended RTL)から構成される. PRTLおよびBRTLは基本となる命令で, XRTLは拡張命令である. PRTLはハードウェアによって直接実現され, 命令セット中に必ず含まれる. また, BRTLは, 実現方法をハードウェア, ソフトウェア (他の命令の組み合わせ), マイクロプログラムの中から選択する. さらに, PRTL, BRTLおよびXRTLに対応する補助命令 PAUX, BAUXおよびXAUXを用意している.

PAUXは直接ハードウェアによって実現される. 補助命令はCのファンクショナリティに直接対応していないがこれらの補助命令あるいは他の命令を組み合わせることによってBRTLおよびXRTLに含まれる命令を実現することができる. 表1にPRTL, BRTL, PAUXおよびBAUXに含まれる命令を示す.

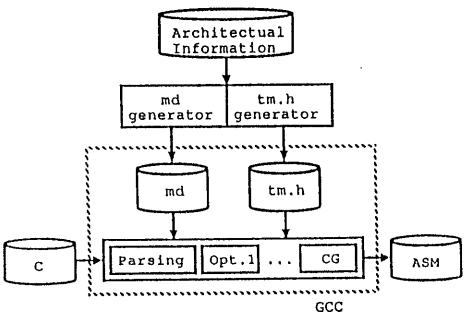


図 2 Cコンパイラ生成系の構成

### 4.3 mdファイルの記述

実際に, Cコンパイラによってファンクショナリティをソフトウェアによって実現する場合, 次の3つの方法が考えられる.

(1) ランタイム・ルーチン
(2) ライブラリ関数
(3) アセンブラによるマクロ展開

今回は, Cコンパイラの実現の容易性を考えて3番目の方法を選んだ. Cコンパイラは, ファンクショナリティに直接対応するコードを出力し, PEASシステムが生成するプロセッサの命令にないファンクショナリティはアセンブラによるマクロ展開を行う.[5]

また, 現在のバージョンでは, 演算対象を固定小数点数のみに限定する. PRTLおよびBRTLに含まれる命令はすべてmdファイル中に記述する.

### 4.4 tm.hファイルの記述

tm.hへの記述のうち現在のバージョンで特徴的な部分を以下に述べる.

(1) レジスタの用途
アーキテクチャの特徴からゼロレジスタとスタックポインタを, また, GCCおよびアセンブラが使用するためフレームポインタと2個の作業用レジスタを汎用レジスタの中から確保し用途を固定する. レジスタの割り当て方法を図3に示す.

(2) データアドレス空間
PEASシステムが生成するプロセッサのデータアドレス空間の大きさは可変である.

(3) 関数のプロログコードおよびエピログコード
PEASシステムが生成するプロセッサが備えるcall命令およびreturn命令は関数呼出し前の状態の保存および復帰の処理を行わない. その処理を行うプロログコードおよびエピログコードを出力するための記述を行う.

(4) スタートアップルーチン
プログラムカウンタに命令の開始番地を与え, スタックポインタにスタックフレームの開始番地を与える

スタートアップルーチンを出力するための記述を行う．

4.5 実行例

実行例として，

(1) ８個の汎用レジスタをもつ

(2) 16Mバイトのメモリ空間をもつ

PEASプロセッサ用のGCCのファイルを記述し，それを使って生成したGCCの出力例を図４に示す．

図４(b)に示すようにコンパイラはアセンブラコードのファイルを出力する．ファイルの先頭部分にスタートアップルーチンが記述されている．また，関数部分の先頭にプロログコードが，終わりの部分にエピログコードが記述されている．

５．GCC用ファイル生成コマンド

前節で述べたGCC用のファイルの記述を自動的に行うGCC用ファイル生成コマンドを作成した．コマンドへの入力は，汎用レジスタの個数とアドレスバスのサイズであり，既定値をそれぞれ８個と24ビットとする．コマンドの出力はmdファイルとtm.hファイルである．特に記述の自動化の対象となるのはtm.hファイルである．ファイルの自動生成を行うことによって，項目間の記述の矛盾などの入力によるミスを抑えることができる．

実際にこのコマンドを使った実行例を示す．

実行例：

```
$ gen.gcc
  number of general register ? [8] 16
  address bus size ? [24] 16
$ config.gcc peas
$ make
```

最初のコマンドにより，GCC用のファイルを生成する．コマンド実行時に汎用レジスタの個数およびアドレスバスのサイズを入力する．２，３番目のコマンドにより，GCCを生成する．

６．おわりに

本稿では，統合化ASIP開発システムPEASにおける応用プログラム開発環境生成系について述べた．PEASの応用プログラム開発環境生成系には，本稿で述べたCコンパイラ生成系の他にアセンブラ，シミュレータ生成系などがある．

現在の問題点として，実際の命令の変化分をアセンブラによるマクロ展開によって吸収しているため，必要の有無に関わらずワークレジスタを確保しなければならない．

また，PEASシステム全体のソフトウェアからみた場合の評価，つまり，シミュレータを使用して応用プロ

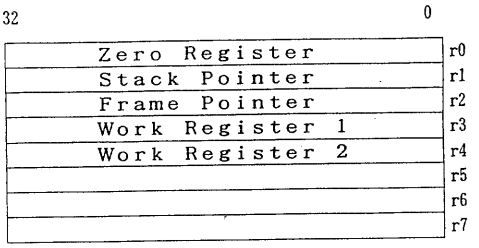


図3 汎用レジスタの最小構成

表1 命令セット

| 分類 | ニモニック | 機能 |
|---|---|---|
| PRTL | addsi | 加算 |
| | subsi | 減算 |
| | andsi | 論理積 |
| | iorsi | 論理和 |
| | xorsi | 排他論理和 |
| | one_cmplsi | １の補数（論理否定） |
| | nop | ノーオペレーション |
| | jump | ジャンプ |
| | branch | 分岐 |
| | call | サブルーチンコール |
| | return | サブルーチンからの復帰 |
| PAUX | ldsi | メモリからレジスタへのデータ転送 |
| | stsi | レジスタからメモリへのデータ転送 |
| | ashlsil | １ビットの算術左シフト |
| | ashrsil | １ビットの算術右シフト |
| | lshlsil | １ビットの論理左シフト |
| | lshrsil | １ビットの論理右シフト |
| | rotlsil | １ビットの左ローテイト |
| | rotrsil | １ビットの右ローテイト |
| | movsr | フラグレジスタと汎用レジスタ間の転送 |
| BRTL | mulsi | 乗算（符号付き） |
| | umulsi | 乗算（符号なし） |
| | divsi | 商（符号付き） |
| | udivsi | 商（符号なし） |
| | modsi | 剰余（符号付き） |
| | umodsi | 剰余（符号なし） |
| | ashlsi | 算術左シフト |
| | ashrsi | 算術右シフト |
| | lshlsi | 論理左シフト |
| | lshrsi | 論理右シフト |
| | negsi | ２の補数 |
| | cmpsi | 比較 |
| | tstsi | テスト |
| | truncsihi | 切り捨て（32ビットから16ビット） |
| | truncsiqi | 切り捨て（32ビットから8ビット） |
| | extendhisi | 符号拡張（16ビットから32ビット） |
| | extendqisi | 符号拡張（8ビットから32ビット） |
| | z_extendhisi | ゼロ拡張（16ビットから32ビット） |
| | z_extendqisi | ゼロ拡張（8ビットから32ビット） |
| BAUX | sethi | 汎用レジスタの上位16ビットの値をセット |
| | setlo | 汎用レジスタの下位16ビットの値をセット |
| | movpc | 汎用レジスタとPC間の転送 |
| | rotlsi | 左ローテイト |
| | rotrsi | 右ローテイト |

グラム解析部で解析した通りの性能が得られているか評価する作業が残っている.

さらに, GCCでより効率的なコードを行うためにRTLレベルでの静的および動的解析が必要である.

その他の今後の課題として, 浮動小数点数演算命令の追加, 拡張命令の追加, リンカおよびローダの生成があげられる.

謝辞　本研究について討論して頂いた豊橋技術科学大学VLSI設計研究室の諸兄に深謝致します.

参考文献

[1] Sato, J., Imai, M., Hakata, T., Alomary, A., and Hikichi, N. : "An Integrated Design Environment for Application Specific Integrated Processor," Proc. of ICCD'91, pp. 414-417, Oct., 1991.

[2] 佐藤　淳, 福田浩一, 市田真琴, 今井正治 : "特定用途向きＣＰＵコアの命令セットに関する一考察," 電子情報通信学会技術研究報告, VLD89-110, pp. 5-10, 1989.

[3] Alomary, A., Nakata, T., Honma, M., Imai, M., Sato, J., and Hikichi, N. : "An Integer Programming Approach to Instruction Set Selection Problem in ASIP Design Automation," 第５回 回路とシステム軽井沢ワークショップ講演集, pp. 267-272, 1992.

[4] Stallman, R. :Using and Porting GNU CC, Free Software Foundation, Version 1.4, 1991.

[5] 本間啓道 : 統合化ASIP開発環境(PEAS)用アセンブラ・ジェネレータの開発, 豊橋技術科学大学 情報工学課程 卒業論文, 1991.

[6] Athanas, P. and Silverman, H. : "An Adaptive Hardware Machine Architecture and Compiler for Dynamic Processer Reconfiguration", IEEE International Conference on Computer Design : VLSI in Computer & Processers(ICCD '91), pp. 397-400, 1991.

```
int
fact(n)
        int n;
{
        if (n == 0)
                return 1;
        else
                return (n * fact(n - 1));
}
```

(a) コンパイラへの入力

```
        sethi   sp      #0x00ff
        setlo   sp      #0xfffc
        sethi   fp      #0x00ff
        setlo   fp      #0xfffc
        ldsi    r5      #0
        ldsi    r6      #0
        ldsi    r7      #0
        call    _main
        jump    _exit
gcc_compiled.:
.text
.globl _fact
_fact:
        subsi   sp      sp      #12
        stsi    sp      #0      fp
        stsi    sp      #4      r5
        stsi    sp      #8      r6
        addsi   fp      sp      #12
        ldsi    r5      fp      #0
        tstsi   r5
        beq     L1
        ldsi    r3      #1
        jump    L2
L1:
        subsi   r6      r5      #1
        stsi    sp      #0      r6
        subsi   sp      sp      #4
        call    _fact
        addsi   sp      sp      #4
        mulsi   r3      r3      r5
L2:
        ldsi    r6      sp      #8
        ldsi    r5      sp      #4
        ldsi    fp      sp      #0
        addsi   sp      sp      #12
        return
```

(b) コンパイラからの出力

図4　生成されるコンパイラの出力例