(19)日本国特許庁(JP)　(12)特　許　公　報(B2)　(11)特許番号
特許第6418696号
(P6418696)
(45)発行日　平成30年11月7日(2018.11.7)　(24)登録日　平成30年10月19日(2018.10.19)

(51)Int.Cl.　F I
*G06F 9/455 (2006.01)*　G06F 9/455

請求項の数 9　(全 72 頁)

(21)出願番号　特願2015-145452(P2015-145452)
(22)出願日　平成27年7月23日(2015.7.23)
(65)公開番号　特開2017-27375(P2017-27375A)
(43)公開日　平成29年2月2日(2017.2.2)
審査請求日　平成30年7月3日(2018.7.3)

早期審査対象出願

(73)特許権者　304021417
国立大学法人東京工業大学
東京都目黒区大岡山２丁目１２番１号
(74)代理人　110000316
特許業務法人ピー・エス・ディ
(74)代理人　110001807
特許業務法人磯野国際特許商標事務所
(72)発明者　一色　剛
東京都目黒区大岡山２－１２－１　国立大学法人東京工業大学内

審査官　多賀　実

最終頁に続く

(54)【発明の名称】命令セットシミュレータおよびそのシミュレータ生成方法

(57)【特許請求の範囲】
【請求項１】
機械語プログラムをプログラムソースコードに変換してシミュレータ実行プログラムが生成される命令セットシミュレータであって、
前記機械語プログラムに含まれるサブルーチンを検出するサブルーチン検出手段と、
前記機械語プログラムに含まれる命令語のうち分岐先アドレスを有する分岐命令を検出する分岐命令検出手段と、
前記機械語プログラムに含まれる命令語のうちサブルーチン呼出し先アドレスを有するサブルーチン呼出し命令を検出するサブルーチン呼出し命令検出手段と、
前記機械語プログラムを前記サブルーチン検出手段で検出した各サブルーチン単位のプログラムソースコードを出力するサブルーチンソースコード出力手段と、
前記分岐先アドレスを示す識別子を前記プログラムソースコードの分岐先の命令に付加する識別子付加手段と、
前記機械語プログラムの前記分岐命令を、前記プログラムソースコードの前記識別子をもつ命令への無条件分岐命令にして出力する無条件分岐命令出力手段と、
前記機械語プログラムのサブルーチン呼出し命令を、前記プログラムソースコードのサブルーチン呼出し命令にして出力するサブルーチン呼出し命令出力手段と、
を備え、
前記分岐命令検出手段は、
前記機械語プログラムに含まれる命令語のうち分岐先アドレスが特定できる単純分岐命

令を検出する単純分岐命令検出手段と、
前記機械語プログラムに含まれる分岐先アドレスがレジスタ値またはメモリ値で決定されるデータ依存分岐命令を検出するデータ依存分岐命令検出手段と、
を有し、
前記機械語プログラムの前記データ依存分岐命令の分岐先アドレスを、ジャンプテーブル情報記憶部に記録するジャンプテーブル記録手段と、
前記ジャンプテーブル情報記憶部から、前記データ依存分岐命令のアドレスを基に、該当するジャンプテーブル情報を検索し、検索されたジャンプテーブル情報を用いて、前記プログラムソースコードの前記無条件分岐命令を生成するデータ依存分岐命令生成手段と、
をさらに備えることを特徴とする命令セットシミュレータ。

【請求項２】
機械語プログラムをプログラムソースコードに変換してシミュレータ実行プログラムが生成される命令セットシミュレータであって、
前記機械語プログラムに含まれるサブルーチンを検出するサブルーチン検出手段と、
前記機械語プログラムに含まれる命令語のうち分岐先アドレスを有する分岐命令を検出する分岐命令検出手段と、
前記機械語プログラムに含まれる命令語のうちサブルーチン呼出し先アドレスを有するサブルーチン呼出し命令を検出するサブルーチン呼出し命令検出手段と、
前記機械語プログラムを前記サブルーチン検出手段で検出した各サブルーチン単位のプログラムソースコードを出力するサブルーチンソースコード出力手段と、
前記分岐先アドレスを示す識別子を前記プログラムソースコードの分岐先の命令に付加する識別子付加手段と、
前記機械語プログラムの前記分岐命令を、前記プログラムソースコードの前記識別子をもつ命令への無条件分岐命令にして出力する無条件分岐命令出力手段と、
前記機械語プログラムのサブルーチン呼出し命令を、前記プログラムソースコードのサブルーチン呼出し命令にして出力するサブルーチン呼出し命令出力手段と、
を備え、
前記サブルーチン呼出し命令検出手段は、
前記機械語プログラムに含まれる命令語のうち呼出し先アドレスが特定できる単純サブルーチン呼出し命令を検出する単純サブルーチン呼出し命令検出手段と、
前記機械語プログラムに含まれる呼出し先アドレスがレジスタ値またはメモリ値で決定されるデータ依存サブルーチン呼出し命令を検出するデータ依存サブルーチン呼出し命令検出手段と、
を有し、
サブルーチン名とサブルーチン機械語アドレスを対とした情報に関するサブルーチン機械語アドレステーブルを生成するサブルーチン機械語アドレステーブル生成手段と、
サブルーチン機械語アドレスから前記プログラムソースコード上のサブルーチンを検索して前記プログラムソースコード上のサブルーチンアドレスを取得するサブルーチンアドレス検索処理のプログラムを生成するサブルーチンアドレス検索処理命令生成手段と、
前記サブルーチンアドレス検索処理の命令によって前記プログラムソースコードのデータ依存サブルーチン呼出し命令の呼出し先サブルーチンを特定して、これを呼び出す処理を行うデータ依存サブルーチン呼出し命令生成手段と、
をさらに備えることを特徴とする命令セットシミュレータ。

【請求項３】
請求項２に記載の命令セットシミュレータにおいて、機械語プログラムにシンボル情報が欠如しているがために、前記サブルーチン検出手段においてすべてのサブルーチンを検出できない場合に対処する手段として、
前記シミュレータ実行プログラムの実行において前記サブルーチン機械語アドレステーブルに登録されていない機械語アドレスがデータ依存サブルーチン命令によって呼び出さ

れた場合には、当該未登録機械語アドレスを記録した後に、前記シミュレータ実行プログラムを強制終了させる未登録機械語アドレス検出手段と、
未登録機械語アドレスのサブルーチンについて、前記手段によりプログラムソースコードを追加生成する未登録サブルーチンプログラムソースコード生成手段とを、
備えることを特徴とする命令セットシミュレータ。
【請求項４】
請求項１から請求項３のうちの何れか一項に記載の命令セットシミュレータにおいて、
前記機械語プログラムのレジスタ値を、前記プログラムソースコード上のサブルーチン引数変数またはローカル変数として記述するレジスタ変数展開手段を備える
ことを特徴とする命令セットシミュレータ。
【請求項５】
請求項１から請求項４のうちの何れか一項に記載の命令セットシミュレータにおいて、
前記プログラムソースコードは、Ｃ言語のプログラムであり、プログラムカウンタに関するswitch文と各命令アドレスに関するcase文のコード構造を用いずに、プログラムソースコード上の識別子を持つ命令への無条件分岐命令とサブルーチン呼出し命令が用いられていて、前記機械語プログラムのサブルーチンと前記プログラムソースコードのサブルーチンとが対応しており、前記機械語プログラムにおけるサブルーチンの階層が前記プログラムソースコードにおけるサブルーチンの階層に復元され、
前記シミュレータ実行プログラムは、前記プログラムソースコードがコンパイルされることによって生成される
ことを特徴とする命令セットシミュレータ。
【請求項６】
機械語プログラムをプログラムソースコードに変換してシミュレータ実行プログラムが生成される命令セットシミュレータのシミュレータ実行プログラム生成方法であって、
前記命令セットシミュレータは、サブルーチン検出手段と分岐命令検出手段とサブルーチン呼出し命令検出手段とサブルーチンソースコード出力手段と識別子付加手段と無条件分岐命令出力手段とサブルーチン呼出し命令出力手段とを備え、
前記サブルーチン検出手段は、前記機械語プログラムに含まれるサブルーチンを検出し、
前記分岐命令検出手段は、前記機械語プログラムに含まれる命令語のうち分岐先アドレスを有する分岐命令を検出し、
前記サブルーチン呼出し命令検出手段は、前記機械語プログラムに含まれる命令語のうちサブルーチン呼出し先アドレスを有するサブルーチン呼出し命令を検出し、
前記サブルーチンソースコード出力手段は、前記機械語プログラムを前記サブルーチン検出手段で検出した各サブルーチン単位のプログラムソースコードを出力し、
前記識別子付加手段は、前記分岐先アドレスを示す識別子を前記プログラムソースコードの分岐先の命令に付加するとともに、前記無条件分岐命令出力手段は、前記機械語プログラムの前記分岐命令を、前記プログラムソースコードの前記識別子をもつ命令への無条件分岐命令にして出力し、
前記サブルーチン呼出し命令出力手段は、前記機械語プログラムのサブルーチン呼出し命令を、前記プログラムソースコードのサブルーチン呼出し命令にして出力し、
前記命令セットシミュレータは、ジャンプテーブル記録手段とデータ依存分岐命令生成手段とを備え、
前記ジャンプテーブル記録手段は、前記機械語プログラムに含まれる分岐先アドレスがレジスタ値またはメモリ値で決定されるデータ依存分岐命令の分岐先アドレスを、ジャンプテーブル情報記憶部に記録し、
前記データ依存分岐命令生成手段は、前記ジャンプテーブル情報記憶部から、前記データ依存分岐命令のアドレスを基に、該当するジャンプテーブル情報を検索し、検索されたジャンプテーブル情報を用いて、前記プログラムソースコードの前記無条件分岐命令を生成する

ことを特徴とする命令セットシミュレータのシミュレータ実行プログラム生成方法。
【請求項７】
機械語プログラムをプログラムソースコードに変換してシミュレータ実行プログラムが生成される命令セットシミュレータのシミュレータ実行プログラム生成方法であって、
前記命令セットシミュレータは、サブルーチン検出手段と分岐命令検出手段とサブルーチン呼出し命令検出手段とサブルーチンソースコード出力手段と識別子付加手段と無条件分岐命令出力手段とサブルーチン呼出し命令出力手段とを備え、
前記サブルーチン検出手段は、前記機械語プログラムに含まれるサブルーチンを検出し、
前記分岐命令検出手段は、前記機械語プログラムに含まれる命令語のうち分岐先アドレスを有する分岐命令を検出し、
前記サブルーチン呼出し命令検出手段は、前記機械語プログラムに含まれる命令語のうちサブルーチン呼出し先アドレスを有するサブルーチン呼出し命令を検出し、
前記サブルーチンソースコード出力手段は、前記機械語プログラムを前記サブルーチン検出手段で検出した各サブルーチン単位のプログラムソースコードを出力し、
前記識別子付加手段は、前記分岐先アドレスを示す識別子を前記プログラムソースコードの分岐先の命令に付加するとともに、前記無条件分岐命令出力手段は、前記機械語プログラムの前記分岐命令を、前記プログラムソースコードの前記識別子をもつ命令への無条件分岐命令にして出力し、
前記サブルーチン呼出し命令出力手段は、前記機械語プログラムのサブルーチン呼出し命令を、前記プログラムソースコードのサブルーチン呼出し命令にして出力し、
前記命令セットシミュレータは、サブルーチン機械語アドレステーブル生成手段とサブルーチンアドレス検索処理命令生成手段とデータ依存サブルーチン呼出し命令生成手段とを備え、
前記サブルーチン機械語アドレステーブル生成手段は、サブルーチン名とサブルーチン機械語アドレスを対とした情報に関するサブルーチン機械語アドレステーブルを生成し、
前記サブルーチンアドレス検索処理命令生成手段は、サブルーチン機械語アドレスから前記プログラムソースコード上のサブルーチンを検索して前記プログラムソースコード上のサブルーチンアドレスを取得するサブルーチンアドレス検索処理のプログラムを生成し、
前記データ依存サブルーチン呼出し命令生成手段は、前記サブルーチンアドレス検索処理の命令によって前記プログラムソースコードのデータ依存サブルーチン呼出し命令の呼出し先サブルーチンを特定して、これを呼び出す処理を行う
ことを特徴とする命令セットシミュレータのシミュレータ実行プログラム生成方法。
【請求項８】
請求項７に記載の命令セットシミュレータのシミュレータ実行プログラム生成方法において、
前記命令セットシミュレータは、未登録機械語アドレス検出手段と未登録サブルーチンプログラムソースコード生成手段とを備え、
前記未登録機械語アドレス検出手段は、前記シミュレータ実行プログラムの実行において前記サブルーチン機械語アドレステーブルに登録されていない機械語アドレスがデータ依存サブルーチン命令によって呼び出された場合には、当該未登録機械語アドレスを記録した後に、前記シミュレータ実行プログラムを強制終了させる機能を有し、
前記未登録サブルーチンプログラムソースコード生成手段は、未登録機械語アドレスのサブルーチンについて、前記手段によりプログラムソースコードを追加生成する
ことを特徴とする命令セットシミュレータのシミュレータ実行プログラム生成方法。
【請求項９】
請求項６から請求項８のうちの何れか一項に記載の命令セットシミュレータのシミュレータ実行プログラム生成方法において、
前記命令セットシミュレータは、レジスタ変数展開手段を備え、

前記レジスタ変数展開手段は、前記機械語プログラムのレジスタ値を、前記プログラムソースコード上のサブルーチン引数変数またはローカル変数として記述する
ことを特徴とする命令セットシミュレータのシミュレータ実行プログラム生成方法。
【発明の詳細な説明】
【技術分野】
【０００１】
本発明は、機械語からプログラムソースコードに変換したり、プログラムソースコードから機械語に変換することで生成される命令セットシミュレータおよびそのシミュレータ生成方法に関する。
【背景技術】
【０００２】
従来、本発明に係る技術に関連する既存技術として逆コンパイラと命令セットシミュレータがある。
逆コンパイラとは、コンピュータが直接実行する実行バイナリ（機械語）からプログラマ等が作成するプログラムソースコードに変換する技術である。つまり、逆コンパイラとは、プログラムソースコードから実行バイナリに変換するコンパイラの逆変換を行うものである。
【０００３】
逆コンパイラ技術の開発は、1960年代から始まり、当初は、あるプラットフォーム用のプログラムを別のプラットフォームに移植する作業の補助として使用されていた。その後、逆コンパイラ技術は、誤って失われたソースコードの復元の補助、プログラムのデバッグ、ウィルスプログラムの発見・解析、プログラム内容の解析・理解、プログラム全体の高位処理構造の抽出等に応用されている。
【０００４】
一方、命令セットシミュレータは、プロセッサの動作を模擬して実行バイナリを実行するツールであり、逆コンパイラ技術が目的とする「可読性の高いソースプログラム復元」と異なる目的を持つ。命令セットシミュレータには、下記の翻訳方式、静的コンパイル方式、動的コンパイル方式の3種類の方式がある。
【０００５】
以下、3種類の命令セットシミュレータ方式を説明する。なお、以下で「ホストCPU」とは、命令セットシミュレータを実行するCPUを指す。
翻訳（Interpreter）方式（非特許文献3）：実行バイナリから機械語命令を一つずつデコード（解釈）しながらCPU動作を模擬する。デコード処理がオーバーヘッドとなりシミュレーション速度が遅い。
【０００６】
静的コンパイル（Static compile）方式（非特許文献4、5、8、9、10）：実行バイナリすべてをまとめてデコードし、ホストCPUの機械語命令に変換した後に実行する。また、ホストCPUの機械語命令に変換する過程で、C言語等のソースコードに変換する場合もある。
【０００７】
動的コンパイル（Dynamic compile）方式（非特許文献6、7）：翻訳方式の改良として、一度デコードしたホストCPUの機械語命令系列をメモリに保存してこれを実行することにより、同じ命令を実行するたびに毎回デコードするオーバーヘッドを軽減する。この方式は、ターゲットCPU命令セットからホストCPU命令セットへの変換を行うバイナリ変換（Binary translation）のため、命令セット変換機構をすべて実装する必要があり、ツール実装は非常に複雑になる。
【先行技術文献】
【非特許文献】
【０００８】
【非特許文献１】Michael Van Emmerik, "Static Single Assignment for Decompilation

", PhD Thesis, The University of Queensland, 2007（http://www.backerstreet.com/decompiler/vanEmmerik_ssa.pdf）
【非特許文献２】http://www.program-transformation.org/Transform/DeCompilation
【非特許文献３】D. Burger, T. M. Austin, "The simplescalar tool set, version 2.0", Computer Architecture News, pp.13-25, 1997
【非特許文献４】C. Mills, S. Ahalt, J. Fowler, "Compiled instruction set simulation", Software - Practice and Exprerience, 21(8), pp.877 - 889, 1991
【非特許文献５】Jianwen Zhu, and Daniel D. Gajski, "An Ultra-Fast Instruction Set Simulator", IEEE Trans. VLSI Systems, Vol.10, No.3, pp.363 - 373, 2002
【非特許文献６】A. Nohl, G. Braun, O/ Schliebusch, R. Leupers, H. Meyr, A. Hoffmann, "A universal technique for fast and flexible instruction-set architecture simulation", Proceedings of DATE 2002
【非特許文献７】M. Reshadi, P. Mishra, N. Dutt, "Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation", Proceeding of DAC 2003, pp. 758 - 763, 2003
【非特許文献８】M. Bartholomeu, R. Azevedo, S. Rigo, G. Araujo, "Optimizations for compiled simulation using instruction type information", Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04), pp.74 - 81, 2004
【非特許文献９】Joseph D'Errico, Wei Qin, "Constructing Portable Compiled Instruction-set Simulators - An ADL-driven Approach", Proceeding of DATE 2006, pp.112 - 117, 2006
【非特許文献１０】S. Bansal, A, Aiken, "Binary translation using peephole superoptimizers", Proceedings of OSDI'08, pp.177 - 192, 2008
【非特許文献１１】Wikipedia: Instruction Set Simulator http://ja.wikipedia.org/wiki/命令セットシミュレータ http://en.wikipedia.org/wiki/Instruction_set_simulator
【非特許文献１２】Wikipedia: Binary Translation http://en.wikipedia.org/wiki/Binary_translation
【発明の概要】
【発明が解決しようとする課題】
【０００９】
ところで、上述の既存の逆コンパイラの課題として、以下の技術的課題がある（非特許文献1のP15～P28参照）。
第１に、データ型の復元が困難である。つまり、実行バイナリから元のソースコードのデータ型を推定することが技術的に非常に難しい。
【００１０】
第２に、「間接分岐」のプログラム制御フローの解析が困難である。具体的には、レジスタ値をPC（プログラムカウンタ）に代入する命令は、C言語でのswitch文（分岐命令）、case文のコード実装で見られる他、関数ポインタを使った関数呼出しでも、レジスタ値をPCに代入する。このようにプログラム制御フローがデータ依存（レジスタの値でプログラム実行遷移が変わる）となる場合の解析が非常に難しい。
【００１１】
第３に、関数引数・戻り値の復元が困難である。つまり、関数の引数形式や戻り値形式は、ターゲットプロセッサの命令セットに依存するだけでなく、Calling Convention（関数呼出しにおける引数形式や戻り値形式の取り決め）に依存する。そのため、このような予備情報無しに関数の引数や戻り値を復元することが難しい。
【００１２】
第４に、完全性という点で逆コンパイラによって出力されたソースコードは以下の２つの課題がある。なお、非特許文献１のP26～P27では、2002年当時で何らかの成果を挙げて

いるのは、dcc decompilerとREC(Reverse Engineering Compiler)のたった2つだけである、としている。

【００１３】

一つ目の課題として、手動編集の必要性（REC）がある。多くの既存の逆コンパイラは手動編集なしにコンパイル可能形式にすることが出来ない。具体的には、出力ソースコードの可読性に重きをおいているため、コンパイル可能形式を保証する逆コンパイラは非常に少ない。

【００１４】

二つ目の課題として、プログラム複雑度の限界（dcc）がある。具体的には、生成ソースコードのコンパイル可能形式を保証する場合でも、プログラム複雑度の制限により扱えるプログラムが限定される。

【００１５】

非特許文献1の手法と解決した課題として、逆コンパイラの中間言語にSSA（static single assignment）形式を採用し、高度なデータ依存性解析を行うことで可読性の高いソース出力を行っている。その他、複数の個別の機械語命令を連結した「式」（expression）を可読性の高い表記形式でソースコードに出力できる、関数引数・戻り値の正確な抽出、変数・関数引数・戻り値のデータ型の高精度な推測、間接分岐のプログラム制御フローの正確な解析が挙げられる。

ただし、「手動編集の必要性」を排除または削減する効果についての言及は特になく、逆コンパイラの完全性に関する効果は限定的であると考えられる。

【００１６】

そこで、逆コンパイラの完成度を上げるための技術構築が望まれる。前記した技術課題（一部は非特許文献1で解決）を克服するためには、 コンパイラ環境と同程度の技術構築が必要となり、非常に複雑な処理工程が数多く必要となる。一方、コンパイラは逆コンパイラと比べて圧倒的に需要が大きく、その開発リソース（コンパイラ開発に従事する研究者・技術者、技術フォーラム等）も圧倒的に多い。そのため、逆コンパイラの技術成熟度はコンパイラ技術に比べかなり劣ることが大きな課題である。

【００１７】

一方、前記した命令セットシミュレータは、逆コンパイラ技術の課題である「完全性」（どのようなプログラムでも確実にシミュレーション可能であるということ）については満足できている。

通常は、静的コンパイル方式が実行バイナリすべてをまとめてデコードすることから、最も高速なシミュレーションが可能とされている。しかし、機械語命令のデコード結果をCソースコードに変換する方式の場合（非特許文献4、8、9）、元のプログラムを直接ホストCPUで実行する（ネイティブ実行）と比べ、14倍～54倍のシミュレーション時間がかかるというデータが非特許文献8で報告されている。

【００１８】

これに対して、非特許文献5、10では、変換結果を直接ホストCPUの機械語命令として出力することで、様々なコード最適化が可能となり、ネイティブ実行時間とほぼ匹敵するシミュレーション時間が実現できるとしている。一方、実行バイナリと等価動作をするソースプログラムの生成を行うことができる方式は、静的コンパイル方式の中でもホストCPUの機械語命令に変換する前にソースプログラムを出力する方式（非特許文献4、8、9）だけである。

【００１９】

なお、非特許文献5、10では、変換結果を直接ホストCPUの機械語命令として出力する方式は、一のプラットフォームを他のプラットフォームに移植できない、変換対象のプログラムの解析ができない、ウイルスの発見、解読、解析ができない等の難点がある。

そのため、実際の使用に供されるのは、機械語命令のデコード結果をCソースコードに変換する方式（非特許文献4、8、9）のみである。

【００２０】

しかし、機械語命令のデコード結果をCソースコードに変換する方式（非特許文献4、8、9）は、前記したように、シミュレーション時間がネイティブ実行と比べ、14倍～54倍かかり、シミュレーション時間が長いという問題がある。

また、非特許文献4、8、9の方式は、後記するように、変換したCソースコードの解析が困難であるという問題がある。そのため、ウイルスの発見、解読、解析が困難である。

【００２１】

ここで、本発明の構成を明らかにするために、本発明に最もアプローチが類似している非特許文献4、8、9が有する課題（本発明が解決すべき課題）について、Cコード出力形式を示して具体的に説明する。

【００２２】

非特許文献4、8、9は、前記のStatic compile方式命令セットシミュレータの中で、実行バイナリの変換結果をC言語等の高級プログラミング言語による記述で出力する技術である。これらのCコード出力のアプローチは、最初の非特許文献4で提案された方式を後続の非特許文献8、9でも採用している。

【００２３】

図53、図54に、非特許文献4で説明されているCコード出力形式を示す。図53は、非特許文献4の機械語命令用Cマクロ定義であり、図54は、非特許文献4の実行バイナリと等価動作するCコード出力である。

図54の４行目の「register UL * S = M, *F = H, A = 0, P = START;」は、以下を意味する。

【００２４】

Sはスタックポインタ、Fはフレームポインタ、Aはアキュムレータ、PはPC（プログラムカウンタ）にそれぞれ対応した関数内部のローカル変数であり、4つのレジスタだけを持つ極めて単純なCPUモデルである。execute関数からリターンする（図54の下から３行目）ときは、SAVESTATEマクロ（図54の下から４行目）によって、これらCPUレジスタ値をグローバル変数に保存する。

【００２５】

図54の６行目以降の「for(;;) switch(P)[ … ]」は、以下を意味する。

命令セットシミュレーションのメインループであり、HALT命令(cpu停止命令)かdefaultラベル（未定義命令アドレスに分岐、エラー処理）に到達するまで実行を繰返す。

switch(P)は、機械語命令アドレスに対応する各case文（Cマクロ内部に埋込まれている）に分岐する構造を持っている。

【００２６】

各機械語命令用Cマクロは、アドレス値をcaseラベルとして持ち、そのアドレスの機械語命令動作のC記述を含む。

分岐命令（CBLS：条件分岐, JSR：サブルーチン呼出し, UNLK：リターン）のCマクロには、プログラムカウンタの更新処理とbreak文を含んでおり、その機械語命令処理の後、switch文に制御が戻り、次命令アドレスに分岐する構造である。

【００２７】

一つのswitch文に含まれるcase文の数には上限があるため、大きなプログラムに対応するためには、プログラムを分割して複数のswitch文を使用して実現する必要がある。（後記の非特許文献８でもプログラム分割を行っている）

【００２８】

図55A、図55Bには、非特許文献4の手法を引用した非特許文献8におけるCコード構造を示す。

図55Aの1行目の「Region3()」は、特定アドレス範囲の機械語命令の実行動作の記述である。以下、各命令を説明する。

図55Aの3行目の「switch文」は、PC（プログラムカウンタ：図中ac_pc）の値によって、該当する命令アドレスのcase文に分岐する。

【００２９】

図55Aの5、11行目の「case文」は、caseのラベル値が命令アドレスに対応しており、そのアドレスの命令の動作がCコードで記述されている。次命令へのプログラム制御（PCの更新）も行い、末尾の各break文によってswitch文に戻る。

図55Aの18行目の「default文」は、Region3()の範囲外の命令アドレスの場合は関数を終了する。

図55Bの1行目の「Execute()」は、シミュレータのメイン関数で、プログラムカウンタの上位ビットによって、該当するプログラム領域に対応するRegionXX()関数を呼ぶ。

【００３０】

非特許文献8のCコードの最適化手法（非特許文献8の5. Optimization Techniques参照）は、以下2つの最適化手法を適用してCコード記述に反映させる。なお、これらの最適化手法は、非特許文献4で示されたCコードでは既に実現されている。

Optimization 1（最適化1）：PCが不連続な変化（分岐，コール等）をしない通常の命令では、次命令は直後のcase文になるので、無駄なbreak文を削除する。

Optimization 2（最適化2）：PCが不連続な変化（分岐，コール等）をするときのみ、PCの更新を行う。

【００３１】

上述の非特許文献8の命令セットシミュレータは、当時の既存技術よりも最も速いとしており、それでもネイティブ実行と比べ14倍～54倍のシミュレーション実行時間がかかると報告されている。

以下に、上記非特許文献4、8、9の手法で得られる命令セットシミュレータのCコード（以降「命令セットシミュレータソースコード」と呼ぶ）に関する課題を説明する。

【００３２】

第１の課題として、各命令アドレスに対応したcase文が必要となる理由（非特許文献7：3ページ目、右欄第1パラグラフ：Each instruction of the decoded application corresponds to respective behavior function call in the generated simulator. These functions must be accessed randomly, as all of them can be a potential target of a jump instruction. ... The switch is based on the Program Counter.）として、実行バイナリは、アセンブリプログラムと異なり、分岐先命令ラベルの情報が欠如している。従って、どの命令が分岐先アドレスとなり得るかを判断する情報がない。このため、「すべての」命令が分岐先アドレスとなり得る、という前提に基づき、switch文から各case文の命令動作コードに直接実行が遷移できるような構造が必要となる。

【００３３】

第２の課題として、シミュレーション実行時間がネイティブ実行に比べ14倍～54倍かかるため、命令セットシミュレータソースコードを別CPUへポーティングする目的で使うことは大きなオーバーヘッドを伴うため実用的でない。シミュレーション実行時間が大きくなる要因は、PCに関するswitch文と各機械語命令のアドレスに対応するcase文の構造から来ている。

【００３４】

第３の課題として、最適化の問題がある。

switch文から任意のcase文へ分岐するコード構造では、分岐先情報が欠如していることからcompilerにおけるコード最適化が非常に掛かりにくくなる。

特に、ここで重要となるコード最適化は、後に使用されない命令を除去する「デッドコード除去」（dead code elimination）であり、シミュレーション実行時間に大きく影響を与える。命令セットシミュレータの動作記述では、 PCや状態フラグ（ゼロフラグ，符号フラグ，オーバーフローフラグ）が最も頻繁に更新され、これらの更新処理の記述が全体の動作記述の大きな部分を占めており、これらが最適化されないため、ネイティブ実行に比べて大きな処理オーバーヘッドとなる原因となっている。

【００３５】

第４の課題として、階層構造の欠如の問題がある。

分岐命令では、PC更新処理の後、break文によってプログラム制御がswitch文に戻った

後にcase文に分岐する一連の処理も、ジャンプテーブル（メモリ）へのアクセスが発生するため、この部分も処理オーバーヘッドの原因となる。つまり、実行バイナリの関数階層構造が出力Cソースコードに反映されていないため、処理が長く時間がかかる原因となっている。

【００３６】

第５の課題として、可読性が低いという問題がある。

また、switch文に含むcase文の数の上限からプログラムを機械的に分割したCコード構造は、ソースプログラムとしては著しく可読性が低い。そのため、プログラムのプログラムのエラーを修正するデバッグ作業や、プログラムの機能拡張などのコード改変作業などが著しく困難になっている。

【００３７】

第６の課題として、プログラム制御フロー解析の欠如という問題がある。

命令セットシミュレーション技術は、プロセッサのシミュレーション機能を実装する部分に対して焦点が当てられ、コンパイラ・逆コンパイラ技術と十分な連携を取らないまま発展してきた経緯がある。そのため、コンパイラ技術のプログラム制御フロー解析の仕組みを取り入れた命令セットシミュレータは存在しない。

【００３８】

図56、図57に、従来の組込みシステム用ソフトウエア開発環境のプログラムを示し、各ケースのメリット、デメリットについて説明する。

図56に、従来の命令セットシミュレータを用いない組込みシステム用ソフトウエア開発環境を示す。

【００３９】

従来、命令セットシミュレータを用いない組込みシステム用ソフトウエア開発環境は、ターゲットCPU搭載組み込みシステム製品（実機）の場合と、ホストCPU搭載コンピュータの場合とがある。

ソフトウエア開発環境がターゲットCPU搭載組み込みシステム製品（実機）の場合、ターゲットCPU用のプログラムソースコードまたはアセンブリコードは、ターゲットCPU用コンパイラで機械語の実行バイナリファイルに変換される。実行バイナリファイルは、ターゲットCPU搭載組み込みシステム製品のメモリに格納される。

【００４０】

このケースでは、開発されたプログラムソースコードまたはアセンブリコードが最終製品のターゲットCPU搭載組み込みシステム製品（実機）に組み込まれるので、詳細な動作が可能である。しかし、実機であるので、各種条件のテストケースを設定するのが困難であり、エラー修正のためのデバッグ作業が困難である。

【００４１】

一方、ソフトウエア開発環境がホストCPU搭載コンピュータの場合、ターゲットCPU用のプログラムソースコードまたはアセンブリコードは、ホストCPU用コンパイラで機械語の実行バイナリファイルに変換される。実行バイナリファイルは、ホストCPU搭載コンピュータに格納される。

【００４２】

このケースでは、開発されたプログラムソースコードまたはアセンブリコードがホストCPU搭載コンピュータに組み込まれ実機ではないので、詳細な動作の再現が無理となる。しかし、ホストCPU搭載コンピュータであるので、各種条件のテストケースを設定することが可能で、エラー修正のためのデバッグ作業が容易である。

【００４３】

図57に、従来の命令セットシミュレータを用いた組込みシステム用ソフトウエア開発環境を示す。

ターゲットCPU用のプログラムソースコードまたはアセンブリコードは、ターゲットCPU用コンパイラで機械語の実行バイナリファイルに変換される。実行バイナリファイルは、命令セットシミュレータで、ホストCPU用の実行バイナリファイルに変換される。ホストC

PU用の実行バイナリファイルは、ホストCPU搭載コンピュータで実行される。ターゲットCPU用の実行バイナリファイルから命令セットシミュレータで、ホストCPU用の実行バイナリファイルに変換されるので、ホストCPU搭載コンピュータでの詳細動作の再現が可能である。

【００４４】

ターゲットCPU用の実行バイナリファイルがホストCPU用の実行バイナリファイルに変換されるので機械語の文法が異なり、シミュレーション時間が長い。

【００４５】

以上のことから、従来技術の解決すべき課題をまとめると、従来技術として、プログラムソースコードを機械語に変換する逆コンパイラ技術と、プロセッサの動作を模擬して実行バイナリを実行する命令セットシミュレータとがある。

【００４６】

逆コンパイラ技術は以下の課題がある。

第１に、データ型の復元が困難である。

第２に、「間接分岐」のプログラム制御フローの解析が困難である。

第３に、関数引数・戻り値の復元が困難である。

第４に、完全性という点で逆コンパイラによって出力されたソースコードは以下の２つの課題がある。一つ目の課題として、多くの既存の逆コンパイラは手動編集なしにコンパイル可能形式にすることが出来ない。二つ目の課題として、プログラム複雑度の制限により扱えるプログラムが限定される。

【００４７】

非特許文献1は、可読性の高いソース出力で上記第１のデータ型の復元が困難な問題を解決し、第２の「間接分岐」のプログラム制御フローの解析が困難、第３の関数引数・戻り値の復元が困難等の問題を解決している。しかし、第４の完全性の問題は未解決である。

【００４８】

一方、命令セットシミュレータでは、逆コンパイラ技術で未解決な第４の完全性については満足できている。しかし、命令セットシミュレータは、元のプログラムを直接ホストCPUで実行する（ネイティブ実行）と比べ、14倍～54倍のシミュレーション時間がかかるという問題がある。

【００４９】

ターゲットCPUの機械語命令を直接ホストCPUの機械語命令として出力する非特許文献5、10では、ネイティブ実行時間とほぼ匹敵するシミュレーション時間が実現できる。しかし、機械語命令を他の機械語命令として出力する非特許文献5、10では、一のプラットフォームを他のプラットフォームに移植できない、変換対象のプログムの解析ができない、ウイルスの発見、解読、解析ができない等の問題がある。

【００５０】

そのため、結果的に実際の使用に供されるのは、非特許文献5、10の問題である一のプラットフォームを他のプラットフォームに移植できない、変換対象のプログムの解析ができない、ウイルスの発見、解読、解析ができない等を解消できる、機械語命令のデコード結果をCソースコードに変換する方式（非特許文献4、8、9）のみである。

しかし、前述したように、非特許文献4、8、9の命令セットシミュレータでは、次の解決すべき課題がある。

【００５１】

第１の課題として、どの命令が分岐先アドレスとなり得るかを判断する情報がない。

第２の課題として、シミュレーション実行時間がネイティブ実行に比べ14倍～54倍かかる。

第３の課題として、compilerにおけるコード最適化が非常に掛かりにくい。

【００５２】

第４の課題として、階層構造が欠如している。

第５の課題として、復元されるソースプログラムがサブルーチンの構造さえ実現していないので、ソースプログラムとしては著しく可読性が低い。
第６の課題として、プログラム制御フロー解析の仕組みが欠如している。
そこで、本願は実際の使用に供される非特許文献4、8、9の上述の第１～第６の課題を解決するものである。

【００５３】

本発明は上記実状に鑑み創案されたものであり、実行バイナリからソースプログラム記述ファイル（命令セットシミュレータソースコード：C言語記述）の復元を確実に行え、解析が容易なソースプログラム記述ファイルを出力でき、高速な命令セットシミュレーション環境を構築できる命令セットシミュレータおよびそのシミュレータ生成方法の提供を目的とする。

【課題を解決するための手段】

【００５４】

前記課題を解決するため、従来考慮されていなかった以下に詳述する新しい８つの処理技術を導入した。
第１の本発明の命令セットシミュレータは、機械語プログラムをプログラムソースコードに変換して生成される命令セットシミュレータであって、前記機械語プログラムに含まれるサブルーチンを検出するサブルーチン検出手段と、前記機械語プログラムに含まれる命令語のうち分岐先アドレスを有する分岐命令を検出する分岐命令検出手段と、前記機械語プログラムに含まれる命令語のうちサブルーチン呼出し先アドレスを有するサブルーチン呼出し命令を検出するサブルーチン呼出し命令検出手段と、前記機械語プログラムを前記サブルーチン検出手段で検出した各サブルーチン単位のプログラムソースコードを出力するサブルーチンソースコード出力手段と、前記分岐先アドレスを示す識別子を前記プログラムソースコードの分岐先の命令に付加する識別子付加手段と、前記機械語プログラムの前記分岐命令を、前記プログラムソースコードの前記識別子をもつ命令への無条件分岐命令にして出力する無条件分岐命令出力手段と、前記機械語プログラムのサブルーチン呼出し命令を、前記プログラムソースコードのサブルーチン呼出し命令にして出力するサブルーチン呼出し命令出力手段とを備えている。

【００５５】

第９の本発明の命令セットシミュレータのシミュレータ生成方法は、第１の本発明の命令セットシミュレータを実現する方法である。

【００５６】

第１の本発明または第９の本発明によれば、サブルーチン検出手段と分岐命令検出手段とサブルーチン呼出し命令検出手段と識別子付加手段とを有するので、どの命令が分岐先アドレスとなり得るかを判断する情報を有している（前記第１の課題の解決）。
機械語プログラムの分岐命令がプログラムソースコードで無条件分岐命令にして出力されるとともに機械語プログラムに含まれるサブルーチンがプログラムソースコードのサブルーチンとして生成される。機械語プログラムの階層構造を復元できる（前記第４の課題の解決）。
そのため、コンパイラの最適化が効果的に行え（前記第３の課題の解決）、処理速度が速い（前記第２の課題の解決）。また、命令セットシミュレータの完全性を実現できる。

【００５７】

第２の本発明の命令セットシミュレータは、第１の本発明において、前記分岐命令検出手段は、前記機械語プログラムに含まれる命令語のうち分岐先アドレスが特定できる単純分岐命令を検出する単純分岐命令検出手段と、前記機械語プログラムに含まれる分岐先アドレスがレジスタ値またはメモリ値で決定されるデータ依存分岐命令を検出するデータ依存分岐命令検出手段とを有している。

【００５８】

第２の本発明によれば、単純分岐命令を単純分岐命令検出手段で検出し、データ依存分岐命令をデータ依存分岐命令検出手段で検出でき、プログラムソースコードでの無条件分

岐命令化を円滑かつ容易に行える（前記第１、前記第６の課題の解決）。

【００５９】

第３の本発明の命令セットシミュレータは、第２の本発明において、前記機械語プログラムの前記データ依存分岐命令の分岐先アドレスを、ジャンプテーブル情報記憶部に記録するジャンプテーブル記録手段と、前記ジャンプテーブル情報記憶部から、前記データ依存分岐命令の前記分岐先アドレスを基に、該当するジャンプテーブル情報を検索し、検索されたジャンプテーブル情報を用いて、前記プログラムソースコードの前記無条件分岐命令を生成するデータ依存分岐命令生成手段とを備えている。

【００６０】

第１０の本発明の命令セットシミュレータのシミュレータ生成方法は、第３の本発明の命令セットシミュレータを実現する方法である。

【００６１】

第３の本発明または第１０の本発明によれば、機械語プログラムの分岐命令からジャンプテーブル情報を用いてプログラムソースコードの無条件分岐命令を円滑かつ容易に生成できる（前記第１、第６の課題の解決）。

【００６２】

第４の本発明の命令セットシミュレータは、第１から第３のうちの何れかの本発明において、前記サブルーチン呼出し命令検出手段は、前記機械語プログラムに含まれる命令語のうち呼出し先アドレスが特定できる単純サブルーチン呼出し命令を検出する単純サブルーチン呼出し命令検出手段と、前記機械語プログラムに含まれる呼出し先アドレスがレジスタ値またはメモリ値で決定されるデータ依存サブルーチン呼出し命令を検出するデータ依存サブルーチン呼出し命令検出手段とを有している。

【００６３】

第４の本発明によれば、単純サブルーチン呼出し命令を単純サブルーチン呼出し命令検出手段で検出し、データ依存サブルーチン呼出し命令をデータ依存サブルーチン呼出し命令検出手段で検出でき、プログラムソースコードでのサブルーチン呼出し命令を円滑かつ容易に生成できる（前記第１、第６の課題の解決）。

【００６４】

第５の本発明の命令セットシミュレータは、第４の本発明において、前記サブルーチン呼出し命令出力手段は、サブルーチン名とサブルーチン機械語アドレスを対とした情報に関するサブルーチン機械語アドレステーブルを生成するサブルーチン機械語アドレステーブル生成手段と、サブルーチン機械語アドレスから前記プログラムソースコード上のサブルーチンを検索して前記プログラムソースコード上のサブルーチンアドレスを取得するサブルーチンアドレス検索処理のプログラムを生成するサブルーチンアドレス検索処理命令生成手段と、前記サブルーチンアドレス検索処理の命令によって前記プログラムソースコードのデータ依存サブルーチン呼出し命令の呼出し先サブルーチンを特定して、これを呼び出す処理を行うデータ依存サブルーチン呼出し命令生成手段とを備えている。

【００６５】

第１１の本発明の命令セットシミュレータのシミュレータ生成方法は、第５の本発明の命令セットシミュレータを実現する方法である。

【００６６】

第５の本発明または第１１の本発明によれば、プログラムソースコード上でデータ依存サブルーチン呼出し先のサブルーチンを特定して、これを呼び出すことが記述できる（前記第４の課題の解決）。

【００６７】

第６の本発明の命令セットシミュレータは、第１の本発明から第５の何れかの本発明において、前記機械語プログラムのレジスタ値を、前記プログラムソースコード上のサブルーチン引数変数またはローカル変数として記述するレジスタ変数展開手段を備えている。

【００６８】

第１２の本発明の命令セットシミュレータのシミュレータ生成方法は、第６の本発明の

命令セットシミュレータを実現する方法である。
【００６９】
第６の本発明または第１２の本発明によれば、レジスタ変数展開手段により機械語プログラムのレジスタ値を前記プログラムソースコード上のサブルーチン引数変数またはローカル変数として記述する（前記第３、第４の課題の解決）ので、命令セットシミュレータの処理速度を速くすることができる（前記第２の課題の解決）。
【００７０】
第７の本発明の命令セットシミュレータは、第１の本発明から第６の何れかの本発明において、前記プログラムソースコードは、Ｃ言語のプログラムであり、プログラムカウンタに関するswitch文と各命令アドレスに関するcase文のコード構造を用いずに、プログラムソースコード上の識別子を持つ命令への無条件分岐命令とサブルーチン呼出し命令が用いられていて、前記機械語プログラムのサブルーチンと前記プログラムソースコードのサブルーチンとが対応しており、前記機械語プログラムにおけるサブルーチンの階層が前記プログラムソースコードにおけるサブルーチンの階層に復元されている。
【００７１】
第７の本発明の命令セットシミュレータによれば、プログラムソースコードは、C言語のプログラムであり、プログラムカウンタに関するswitch文と各命令アドレスに関するcase文のコード構造を用いずに、プログラムソースコード上の識別子を持つ命令への無条件分岐命令とサブルーチン呼出し命令が用いられていられるので、最適化を効果的に行うことができる。また、機械語プログラムのサブルーチンとプログラムソースコードのサブルーチンとが対応しており、機械語プログラムにおけるサブルーチンの階層がプログラムソースコードにおけるサブルーチンの階層に復元される（前記第４の課題の解決）ので最適化が効果的に行え（前記第６の課題の解決）、命令セットシミュレータの処理速度が速い（前記第２の課題の解決）。また、プログラムソースコードの可読性が高く解析が容易である。そのため、マルウェア、ウィルス等の解析が容易に行える（前記第５の課題の解決）。
【００７２】
第８の本発明の命令セットシミュレータは、第１の本発明から第７の何れかの本発明において、機械語プログラムにシンボル情報が欠如しているがために、前記サブルーチン検出手段においてすべてのサブルーチンを検出できない場合に対処する手段として、前記サブルーチン機械語アドレステーブルに登録されていない機械語アドレスがデータ依存サブルーチン命令によって呼び出された場合には、当該未登録機械語アドレスを記録した後に、命令セットシミュレータを強制終了させる未登録機械語アドレス検出手段と、未登録機械語アドレスのサブルーチンについて、前記手段によりプログラムソースコードを追加生成する未登録サブルーチンプログラムソースコード生成手段とを、備えている。
【００７３】
第１３の本発明の命令セットシミュレータのシミュレータ生成方法は、第８の本発明の命令セットシミュレータを実現する方法である。
【００７４】
第８の本発明または第１３の本発明によれば、機械語プログラムにシンボル情報が欠如しているがために、サブルーチン検出手段においてすべてのサブルーチンを検出できない場合に対処できる（前記第６の課題の解決）。
【発明の効果】
【００７５】
本発明によれば、実行バイナリからソースプログラム記述ファイルの復元を確実に行え、解析が容易なソースプログラム記述ファイルを出力でき、高速な命令セットシミュレーション環境を構築できる命令セットシミュレータおよびそのシミュレータ生成方法を提供することができる。
【図面の簡単な説明】
【００７６】

【図１】Cソースコードのサンプルを示す図。
【図２】各機械語命令のアドレスと32ビット機械語命令の16進表示と逆アセンブルしたアセンブリ命令を示す図。
【図３】本実施形態によって出力されるCソース記述を示す図。
【図４Ａ】本発明による実行バイナリのC記述出力を示す図。
【図４Ｂ】本発明による実行バイナリのC記述の再コンパイル後のホストPC（X86命令セット）の機械語命令を重ね合わせて表示したものを示す図。
【図５】第１実施形態の命令セットシミュレータを用いた組込みシステム用ソフトウエア開発環境を示す図。
【図６】ELFファイルの全体構造の概要を示す図。
【図７】第１実施形態の機械語命令解析部の機能ブロック図。
【図８】switch文を含んだ関数jump_testのサンプルCプログラムを示す図。
【図９】データ依存分岐情報抽出部の機能ブロック図。
【図１０】図8のjump_test関数のARMv5命令セットの機械語命令とアセンブリ命令を示す図。
【図１１】jump_test関数のx86(64-bit)命令セットの機械語命令とアセンブリ命令を示す図。
【図１２】命令セットシミュレータの構成を、解析対象の実行バイナリファイル５、命令セットシミュレータが出力する命令セットシミュレータプログラムソースコードとともに示す図。
【図１３】ARMv5用命令セットシミュレータプログラムにおけるCPUリソースのデータ構造記述の一例を示す図。
【図１４】CPUリソース参照用マクロ定義と命令実行条件判定用マクロ定義を示す図。
【図１５】"jump_test"関数のARMv5命令セット機械語・アセンブリ記述の4命令を示す図。
【図１６】図15の４つのARMv5命令セット機械語命令に対するマクロ呼出し記述を示す図。
【図１７】SUBマクロ定義とCMPマクロ定義を示す図。
【図１８】_SUB_マクロ定義を示す図。
【図１９】_SUB_マクロ内部で呼ばれるその他のマクロを示す図。
【図２０】図16で呼ばれるSTRマクロ定義を示す図。
【図２１】_ADDR_,_m8_，_m16_，_m32_，D_CACHE_SIM各マクロ定義を示す図。
【図２２】LDRマクロ定義を示す図。
【図２３】図16の4つの機械語命令プログラム生成マクロ呼出しによって生成されるプログラム記述を示す図。
【図２４】サブルーチンプログラムソース生成部の機能ブロック図。
【図２５】"jump_test"関数のサブルーチン定義記述出力を示す図。
【図２６】無条件分岐命令を示す図。
【図２７】無条件分岐命令のプログラム記述を示す図。
【図２８】サブルーチン呼出し命令を示す図。
【図２９】サブルーチン呼出し命令のプログラム記述を示す図。
【図３０】条件付きデータ依存分岐命令と直後の無条件分岐命令を示す図。
【図３１】条件付きデータ依存分岐命令と直後の無条件分岐命令のプログラム記述を示す図。
【図３２】データ依存サブルーチン呼出しを実現する２つの機械語命令を示す図。
【図３３】データ依存サブルーチン呼出し命令のプログラム記述を示す図。
【図３４】図10の"jump_test"関数の機械語命令記述に対応するプログラム記述の出力を示す図。
【図３５】シンボルアドレス情報を格納する_FP_INFO_構造体を示す図。
【図３６】_FP_INFO_構造体の初期値設定用プログラム記述を示す図。

【図３７】_FP_INFO_構造体のポインタを返す_GET_FPI_関数の定義を示す図。
【図３８】メモリ初期化プログラム記述の例を示す図。
【図３９】命令セットシミュレータの最上位関数のプログラム記述例１を示す図。
【図４０】命令セットシミュレータの最上位関数のプログラム記述例２を示す図。
【図４１】main関数の引数情報をCPUメモリに書き込むプログラム記述を示す図。
【図４２】第2実施形態の機械語命令解析部の機能ブロック図を示す図。
【図４３】第2実施形態の命令セットシミュレータの構成を示す図。
【図４４】図11の"jump_test"関数のARMv5機械語命令の入出力レジスタを示す図。
【図４５】サブルーチン引数抽出部の機能ブロック図。
【図４６】CPUリソース参照用マクロ定義と命令実行条件判定用マクロ定義を示す図。
【図４７】"jump_test"関数のサブルーチン定義記述出力を示す図。
【図４８】サブルーチン呼出し命令のプログラム記述を示す図。
【図４９】関数ポインタデータ型の定義を示す図。
【図５０】データ依存サブルーチン呼出し命令のプログラム記述を示す図。
【図５１】第3実施形態の命令セットシミュレータを用いた組込みシステム用ソフトウエア開発環境を示す図。
【図５２】_FP_INFO_構造体のポインタを返す_GET_FPI_関数の定義を示す図。
【図５３】従来の非特許文献4の機械語命令用Cマクロ定義を示す図。
【図５４】従来の非特許文献4の実行バイナリと等価動作するCコード出力を示す図。
【図５５Ａ】非特許文献4の手法を引用した非特許文献8におけるCコード構造を示す図。
【図５５Ｂ】非特許文献4の手法を引用した非特許文献8におけるCコード構造を示す図。
【図５６】従来の命令セットシミュレータを用いない組込みシステム用ソフトウエア開発環境を示す図。
【図５７】従来の命令セットシミュレータを用いた組込みシステム用ソフトウエア開発環境を示す図。
【発明を実施するための形態】
【００７７】
以下、本発明の実施形態について、適宜図面を参照しながら詳細に説明する。
＜本発明の概要＞
本発明の命令セットシミュレータは、逆コンパイル技術で構築されてきたプログラムフロー制御解析機能やデータ依存性解析機能をStatic compile（静的コンパイル）方式に適用したものである。これにより、命令セットシミュレータ技術の持つ「完全性」を担保し、シミュレーション速度をネイティブ実行同等まで向上させられる。また、復元ソースコードの一定の可読性を担保する。
なお、「完全性」とは、どのようなプログラムでも確実にシミュレーション可能など実行バイナリを「確実」に実行できることである。
【００７８】
＜＜第1実施形態＞＞
本発明の具体的内容を明らかにするため、第1実施形態の命令セットシミュレータのCソースコード出力について説明する。
【００７９】
＜本発明による実行バイナリと等価動作するCソースコード出力＞
図1に、Cソースコードのサンプルを示す。
図1に示すCソースコードサンプルを、C Compiler（gcc：GNU Compiler Collection）で組み込みプロセッサのARMv5命令セットの機械語命令に変換したものを図2に示す。図2には、各機械語命令のアドレスと32ビット機械語命令の16進表示と逆アセンブルしたアセンブリ命令を示している。図2において、左側4桁の英数字が命令アドレスであり、中央8桁の英数字が16進表示機械語命令であり、右側がアセンブリ命令である。
【００８０】
図2に示すARMv5命令セットの特徴を以下に示す。

・16個のレジスタ（r0，…，r9，sl，fp，ip，sp，lr，pc）：プログラムカウンタ(pc)も通常の命令で書き換えることができるようになっている。また、pcを更新する各種命令によって様々なプログラム制御が実現できる。

【００８１】

・実行条件：すべての命令は5ビットの実行条件フラグを含んでおり、実行条件が成立しない場合は実行をスキップする。図2において、popgt、 beq、 ble命令のサフィックス（gt，eq，le）が実行条件（greater than，equal，less or equal）であり、実行条件サフィックスがない命令は常に実行される（al：always）。

【００８２】

・定数プール：32ビット即値（定数）をレジスタにロードする仕組みとしてプログラム領域に埋込む「定数プール」を採用している．0x8234，0x8238，0x8428の各アドレスのデータ（0x1b308，0x1b404，0x1b308）が定数プールに該当する。これら定数をロードするときは、pcをベースアドレスにしたロード命令（0x8218，0x8220，0x83c4）で実現する。

【００８３】

図3に、本実施形態によって出力されるCソース記述を示す。

以下に本実施形態の命令セットシミュレータによって出力されるCソース記述の特徴を説明する。

【００８４】

一つ目の特徴として、一つの機械語命令が一つのCマクロ呼出し命令（LDR，ADD，CMP等）と一対一に対応している。つまり、基本的に、機械語のソースコードの記述の順番に処理する逐次処理を遂行するという特徴がある。実際の命令動作は各Cマクロ記述の内部で定義されている。（非特許文献４も同様のCマクロ記述を利用している：図53、図54参照）

【００８５】

二つ目の特徴は、元のCソースコード（図１）の関数と、実行バイナリから本実施形態の命令セットシミュレータで生成されるC記述の関数が一対一に対応している。つまり、元のCソースコードの関数階層が、本実施形態の命令セットシミュレータＳで出力されるCソース記述にそのまま復元される。

【００８６】

C記述の関数呼出し構造の特徴は以下の通りである。

第1に、関数の引数（例えばr0）は、CPUレジスタに対応したパラメータ名で引き渡される。関数引数に現れないその他のCPUレジスタは関数のローカル変数として宣言される。図3において、r0，r1，…，r9，sl，fp，…，pcまでがCPU汎用レジスタ変数（ローカル変数）であり、cv，cc，…，cleがCPU状態レジスタ変数（後記）である。

【００８７】

第2に、C記述の関数呼出し機構を使うため、CPU動作として関数呼出しの際に必要となるPC（プログラムカウンタ）とリンクレジスタ（lr：リターンアドレスを格納するレジスタ）の更新処理は必要ない。そのため、ネイティブ実行と同程度の関数呼出しの処理オーバーヘッドに抑えることができる。

【００８８】

第3に、呼出し関数側でスタックフレーム確保が必要な場合は、スタックポインタ（sp）を関数引数として引き渡している。

【００８９】

第4に、関数戻り値は、関数呼出し規則（コンパイラ依存）で規定されるレジスタ（上記例ではr0）を介して引き渡す（図3の12行目）。元のC関数がvoid型（戻り値がない関数）であっても、r0を引き渡すような記述を出力しているが、プログラム動作上の不具合は発生しない。何故なら、r0は関数呼出しの前後で値が書き換えられることをコンパイラは前提とするため、r0に「空」（値不定の）戻り値が代入されても参照されないので問題ない。

【００９０】

三つ目の特徴は、分岐先命令の前にCラベル（L_083f0，L_083fc，L08400等）（識別子）を配置し、分岐命令は無条件分岐命令のgoto文で記述する。

これにより、従来の静的コンパイル方式命令セットシミュレータ（非特許文献4、8、9）が生成していたPC（プログラムカウンタ）に関するswitch文と各命令アドレスに関するcase文のコード構造を用いずに済む。そして、実行バイナリのプログラム制御フローをそのままC言語の記述で表現できる。

【００９１】

このような本来のプログラム制御フローと同等の構造をC言語の記述で表現することにより、コンパイラの最適化処理が最大限に効果を発揮できる。そのため、シミュレーション実行時間を大幅に短縮できる。

【００９２】

＜本実施形態による命令セットシミュレータ実行時間＞

本実施形態（本発明）によるCソースコード構造の特質は、実際に本実施形態の命令セットシミュレータによるCソースコードをコンパイルして実行することで確認できる。図3のCソースコードに出現するCマクロの内部構造は、複雑なCPU動作を忠実にC言語の記述で表現するためにかなり複雑なコードで構成されている。ここでは、複雑なCPU動作のC記述にも関わらず、本命令セットシミュレータが出力するCソースコードが如何に効率的な実行コード（ホストCPUの実行バイナリ）を生成するかについて説明する。ホストCPUとは、命令セットシミュレータ等のソフトウエアツールを実行するプロセッサ（デスクトップ・ノートブックPCに搭載されているプロセッサ等）をいう。

【００９３】

図4A、図4Bに、本発明による実行バイナリのC記述出力と、その再コンパイル後のホストPC（X86命令セット）の機械語命令を重ね合わせて表示したものを示す。図4A、図4B中の罫線で囲った部分が再コンパイル後のホストPC（X86命令セット）の機械語命令である。

【００９４】

・_my_put_prime_の関数呼出し（2ヶ所）は，インライン展開されている．
・ARMv5コードサイズ：26 + 7 * 2 = 40命令
　get_primes関数：26命令
　put_prime関数：7命令（インライン展開2ヶ所）
・X86コードサイズ：42命令

【００９５】

図2のARMv5と、図4A、図4BのX86の異なる命令セットを直接比較はできないが、少なくとも、図4A、図4Bに示す複雑なCマクロで構成されているにも関わらず、数の上で一つのARMv5命令と一つのX86命令がほぼ対応している。結果、本命令セットシミュレータによるCソースコードのコンパイラ最適化が最大限に効いていることが分かる。ホストPC（X86命令セット）の機械語命令が少ないほど実行時間が短くなり、シミュレーション時間が短くなるので、本実施形態の命令セットシミュレータを用いた場合のシミュレーション時間が短かくできることが分る。

【００９６】

また、実際にネイティブ実行時間（元のCソースコードを直接コンパイル・実行）と、本命令セットシミュレータSによるシミュレーション実行時間（実行バイナリから命令セットシミュレータＳでCソースコード出力し、これをホストCPUでコンパイルして実行）を比べると以下の結果が得られている。

【００９７】

テストプログラムの図2のコードを含む素数計算プログラムを、
ホストCPU：Intel Xeon 3.4GHz (Quad core), 3.25GB memory
で実行すると、ネイティブ実行時間：0.7580秒であった。

これに対して、本実施形態（本発明）の命令セットシミュレータSによるテストプログラムの図2のコードを含む素数計算プログラムのシミュレーション実行時間は、0.7699秒

であり、対ネイティブ実行時間：0.7580秒の比で1.016倍であった。

【００９８】

これより、従来のシミュレーション実行時間がネイティブ実行に比べ14倍～54倍であったものが、本実施形態の命令セットシミュレータSによれば、1.016倍と大幅に短縮できることが明らかである。

【００９９】

＜命令セットシミュレータを用いた組込みシステム用ソフトウエア開発環境＞

次に、命令セットシミュレータSを用いた組込みシステム用ソフトウエア開発環境について説明する。

図5に、第1実施形態の命令セットシミュレータを用いた組込みシステム用ソフトウエア開発環境を示す。

【０１００】

ターゲットCPUとは、実行バイナリの機械語命令を実行するプロセッサ（組込みシステムに搭載されているプロセッサ等）をいう。

ホストCPUとは、命令セットシミュレータ等のソフトウエアツールを実行するプロセッサをいう。例えば、デスクトップ・ノートブックPCに搭載されているプロセッサ等をいう。

【０１０１】

ターゲットCPU用のプログラムソースコードまたはアセンブリコードは、ターゲットCPU用コンパイラでターゲットCPU用の実行バイナリファイルに変換される。実行バイナリファイルは、ターゲットCPU搭載組み込みシステム製品のメモリに格納され実行される。

【０１０２】

命令セットシミュレータSでのシミュレーションに際しては、命令セットシミュレータSの機械語命令解析部1がターゲットCPU用の実行バイナリファイルを読み込み、機械語命令解析処理を実行し、シンボル情報リストr1、分岐先アドレスリストr2、ジャンプテーブル情報リストr3を出力する。

【０１０３】

命令セットシミュレータSのプログラムソースコード出力部2は、ターゲットCPU用の実行バイナリファイルと、シンボル情報リストr1、分岐先アドレスリストr2、ジャンプテーブル情報リストr3とを読み込む。そして、プログラムソースコード出力部2は、命令セットシミュレータプログラムソースコード（命令セットシミュレータＳのプログラムソースコード）を出力する。なお、本実施形態において、命令セットシミュレータプログラムソースコードは、前記したC言語ソースコードが相当する。

【０１０４】

命令セットシミュレータプログラムソースコードは、ホストCPU用コンパイラにより、ホストCPU用の実行バイナリファイルに変換される。ホストCPU用の実行バイナリファイルは、ホストCPU搭載コンピュータのメモリにロードされ、実行される。

命令セットシミュレータSは、ソフトウェアを用いて実現される。なお、命令セットシミュレータＳのうちの少なくとも一部をハードウェアで構成してもよい。

【０１０５】

次に、第1実施形態の命令セットシミュレータSの構成について詳細に説明する。

＜実行バイナリデータ構造＞

最初に、実行バイナリ（機械語）のデータ構造について説明する。

図6に、ELFファイルの全体構造の概要を示す。

【０１０６】

プログラムの実行バイナリデータ（機械語データ）のフォーマットは、コンパイラやOSによって数種類存在するが、ここではELF(Executable and Linkable Format)ファイルを例に、その内部構造を説明する。なお、下記で「格納位置」とは、ELFファイルの先頭からのオフセット値を指し、ELFファイル中に格納されている位置をバイト数で指定する。

【０１０７】

（1）ELFヘッダー：ELFファイルの先頭に位置し、ELF内部に保存されている各種データを取得するために必要な情報をすべて含んでいる。主な項目は以下の通りである。
ファイル種類は、実行可能ファイル、リンク可能ファイル、共有ライブラリ等である。
CPU機種・バージョン、エントリーポイント（プログラム実行時の最初のアドレス）、プログラムヘッダーテーブルの格納位置、セクションヘッダーテーブルの格納位置のELFヘッダーのサイズ、プログラムヘッダーテーブルの１つのエントリーのサイズとエントリー数、セクションヘッダーの１つのエントリーのサイズとエントリー数、セクション名文字列テーブルが格納されているセクションヘッダーテーブルインデックス等がある。
【０１０８】
（2）プログラムヘッダーテーブルは、ELFファイルをCPUで実行するときに、OS等がメモリ領域確保・メモリ初期化等の準備をするために必要となる情報を含む。1つのエントリーは1つのセグメント（1つ以上のセクションからなる）に関する以下の情報を主要項目として含む。
セグメント型として読込み可能セグメント、動的リンクセグメント等がある。その他、セグメントの格納位置、セグメントがメモリ上に配置される仮想アドレスと物理アドレス、セグメントのファイルサイズとメモリサイズ、セグメントのアクセス権（実行可、読出し可、書込み可の組合せ）等がある。
【０１０９】
（3）バイナリデータ部は、機械語命令データ（プログラムメモリデータ）やデータメモリの初期値データが格納されている。
【０１１０】
（4）セクションヘッダーテーブル
セクションとは、メモリ領域（プログラムメモリ・データメモリ）の構成単位として、また補足情報の格納単位として存在している。セクションヘッダーテーブルの1つのエントリーが1つのセクションに関する以下の情報を主要項目として含む。
セクション名文字列情報としてセクションヘッダー内のセクション名文字列テーブルセクションへのインデックスがある。
セクション型としてプログラム定義情報（メモリ領域の構成単位、デバッグ情報他）、シンボルテーブル（プログラムアドレス情報、変数アドレス情報）、文字列デーブル、再配置情報、動的リンク情報等がある。
セクション属性としてメモリ領域占有（読出し可能）、書込み可能、実行可能等がある。
その他、セクションのメモリアドレス、セクションの格納位置、セクションのサイズがある。
【０１１１】
（5）シンボルテーブル
シンボルとは、主にメモリに格納されるプログラムアドレス情報と変数情報を含み、いずれかのセクションに属する。
シンボルテーブルは、セクションヘッダーテーブルの１つのエントリーとして格納される。シンボルは以下の情報を主要項目として含む。
【０１１２】
シンボル名文字列情報として、セクションヘッダー内の「シンボル名文字列テーブル」へのインデックスがある。
シンボルのメモリアドレス、シンボルのサイズがある。
シンボル属性として、バインド属性、タイプ属性、可視性属性がある。特に、タイプ属性では、関数や他の実行可能命令に関連付けられるシンボルの属性を「FUNC型」と呼ぶことにする。
セクションインデックスとして、シンボルが属するセクションのインデックスがある。
【０１１３】
（6）特殊なシンボル：「シンボルテーブル」で定義されたシンボルの中で、後述の「機

械語命令解析処理」において、特別な扱いが必要なものが含まれる。
「エントリーシンボル」：ELFヘッダーで定義された「エントリーポイント」（プログラム実行時の最初のアドレス）を「シンボルメモリアドレス」として持つ「シンボル」のことを指す。この「エントリーシンボル」の実行を終了することで、プログラム実行が終了するが、通常の終了手段としては、CPU動作を停止する「停止命令」もしくは、次の「プログラム終了処理シンボル」へのサブルーチン呼出し命令を実行する。
「プログラム終了処理シンボル」：プログラム実行を終了する処理を行うシンボルを指す。例えば、C言語ではexit()関数が「プログラム終了処理シンボル」である。
【０１１４】
＜実行プログラムのメモリ領域の情報＞
前記ELF構造において、実行プログラムが格納されているメモリ領域に関する情報は、幾つかの階層で定義されている。
・エントリーポイント（ELFヘッダー項目）：プログラム起動後に最初に実行される命令が格納されたアドレス
【０１１５】
・「実行可能」アクセス権を持つセグメントのアドレス領域がプログラムヘッダーテーブルで定義される。
・「実行可能」属性を持つセクションのアドレス領域がセクションヘッダーテーブルで定義される。
【０１１６】
・「実行可能」属性を持つセクションのアドレス領域に含まれるシンボルのメモリアドレスがセクションヘッダーテーブル内で定義されたシンボルテーブルで定義される。
ここで、実行ファイルで定義されていることが保証されている情報はエントリーポイントと実行可能セグメントのアドレス領域であり、セクションヘッダーテーブルは省略可能であるため、セクション情報やシンボル情報が欠如する場合もあり得る。
【０１１７】
＜機械語命令解析処理のためのデータ構造＞
まず、機械語命令解析処理で必要となるデータ構造を説明する。
（1）「シンボルアドレス一時リスト」：実行可能属性を持ち、関数等に関連付けられた実行可能シンボルのアドレスのリスト構造をもつ。機械語命令解釈処理が進行するに従って更新される。
【０１１８】
（2）「シンボル情報リスト」：「シンボル情報」のリスト構造をもつ。
「シンボル情報」とは、実行可能シンボルに関するシンボルアドレスと命令アドレスリストとの情報項目を格納する。
（A）シンボルアドレス
（B）命令アドレスリスト：シンボルアドレスから到達可能な全命令のアドレスのリスト構造をもつ。
【０１１９】
（C）シンボル名文字列：「シンボルテーブル」に含まれる「シンボル名文字列情報」から生成する。「シンボルテーブル」に存在しないシンボルや、「シンボルテーブル」自体が存在しない場合は、重複しない文字列を生成する任意の命名規則に従って「シンボル名文字列」を生成する。
（D）「データ依存サブルーチン命令フラグ」：シンボルアドレスから到達可能な全命令のなかで「データ依存サブルーチン命令」が含まれているか（１）否か（０）を示すフラグ。
【０１２０】
（3）「命令アドレス一時リスト」：処理対象の機械語命令が格納され、機械語命令解析部１の処理が進行するに従い更新される（処理対象の命令アドレスが読み込まれ削除される）命令アドレスのリスト構造をもつ。

【０１２１】
（4）「分岐先アドレスリスト」：条件付き分岐命令や無条件分岐命令の分岐先アドレスのリスト構造をもつ。
【０１２２】
（5）「次命令情報一時リスト」：「次命令情報」のリスト構造をもつ。
「次命令情報」とは、ある命令が実行されてから次に実行し得る命令（以降「次命令」と呼ぶ）に関する情報で、以下のデータ項目からなる。
A）次命令アドレス
B）次命令タイプは、以下の３タイプのいずれかである。
【０１２３】
1）「直後」型：その命令語が格納されているメモリ上の直後のアドレスに格納されている命令を「直後命令」と呼ぶことにする。PC（プログラムカウンタ）に作用をしない演算命令（ADD、SUB等）・ロード命令（LDR）・ストア命令（STR）等は「直後命令」が次命令となる。
2）「分岐」型：分岐命令における分岐先アドレス
3）「呼出し」型：サブルーチン呼出し命令における呼出し先アドレス
また、上記3つの「次命令タイプ」に対応して、「直後型次命令」、「分岐型次命令」、「呼出し型次命令」とそれぞれを呼ぶことにする。
【０１２４】
（6）「ジャンプテーブル情報リストr3」：「ジャンプテーブル情報」のリスト構造である。
「ジャンプテーブル情報」とは、後記の「データ依存分岐情報抽出部2の処理」によって生成される情報で、以下のデータ項目からなる。
A）データ依存分岐命令アドレス
B）ジャンプテーブルサイズ（データ依存分岐情報抽出部2の処理が成功しなかった場合は、ジャンプテーブルサイズは「0」となる）ジャンプテーブル情報リストに情報がないことを意味する。
C）分岐先アドレステーブル（1つのエントリーは、分岐先アドレスを格納する。「エントリー数」＝「ジャンプテーブルサイズ」である。）
【０１２５】
＜リスト構造に対する操作用語説明＞
（1）リストへの「（非重複）追加」：追加するデータと同一データがリストに存在していない時のみ、リストの最後尾にデータを追加することをいう。以降では、基本的にすべてのリストが保持するデータは「非重複的」であるので、リストへの「追加」という表現は、特別な断りがない限り「非重複的追加」を意味する。
（2）リストからの「取出し」：リストの最後尾のデータを取出し、この最後尾データを削除することをいう。
【０１２６】
＜＜第1実施形態＞＞
次に、第1実施形態の命令セットシミュレータSの機械語命令解析部1について説明する。
図7に、第1実施形態の機械語命令解析部の機能ブロック図を示す。
【０１２７】
第1実施形態の機械語命令解析部1は、対象CPUで実行される分析対象の機械語をCコードに変換するために、機械語を解析する役割をもつ。
機械語命令解析部1は、シンボルアドレス前処理部1a、シンボルアドレス取得部1b、命令アドレス取得部1c、次命令情報抽出部1d、アドレスリスト更新部1e、およびデータ依存分岐抽出部2を有している。
【０１２８】
機械語命令解析部1は、解析対象の機械語からCコードを作成するために、シンボル情報

リストr1と分岐先アドレスリストr2とジャンプテーブル情報リストr3とを出力する。
ここでは、実施形態1の機械語命令解析部1の処理として、セクションヘッダーテーブルが存在し、さらにその中にシンボルテーブルも定義されているという前提が成立する場合での処理の流れを説明する。
【０１２９】
＜シンボルアドレス前処理部1a＞
シンボルアドレス前処理部1aは、「シンボルアドレス一時リスト」を作成する。
シンボルアドレス前処理部1aは、セクションヘッダーテーブル内で定義されたシンボルテーブルにおいて、実行可能属性を持つセクションのアドレス領域に含まれたシンボルであって、関数や他の実行可能命令に関連付けられたタイプ属性を持つシンボル（前記の「FUNC型シンボルタイプ属性」参照）を「実行可能シンボル」と見なす。シンボルアドレス前処理部1aは、「実行可能シンボル」のアドレスをすべて「シンボルアドレス一時リスト」に「追加」することで「シンボルアドレス一時リスト」を作成する。
その後、シンボルアドレス取得部1bへ進む。
【０１３０】
＜シンボルアドレス取得部1b＞
（1）「シンボルアドレス一時リスト」が空でない場合
A）シンボルアドレス取得部１ｂは、「シンボルアドレス一時リスト」から、シンボルアドレスを一つ「取出す」（以降、このシンボルを「現シンボル」と呼ぶ）。
B）現シンボルに関する「シンボル情報」が既に「シンボル情報リスト」に存在する場合（現シンボルは処理済み）は、以降の処理すべてをスキップし、直ちにシンボルアドレス取得部１ｂの処理のスタートに戻る。
【０１３１】
C）現シンボルに関する「シンボル情報」が「シンボル情報リスト」に存在しない場合、取出した現シンボルの「シンボル情報」を以下のようにして生成する。なお、このシンボル情報を、以降「現シンボル情報」という。
1）「シンボル情報」の「シンボルアドレス」に「現シンボル」アドレスを設定する。
2）「シンボル名文字列」を生成する。シンボルが「シンボルテーブル」に登録されている場合は、「シンボルテーブル」内の「シンボル名文字列情報」を取得して生成する。一方、シンボルが「シンボルテーブル」に登録されていない場合は、任意の命名規則によりシンボル名として重複しない文字列を生成する（一例としては、「シンボルアドレス」を基にシンボル名を生成する：シンボルアドレスが0x1234の場合に、「func_1234」とする、など）。
3）「データ依存サブルーチン命令フラグ」を「0」に設定する。
【０１３２】
D）現シンボルのアドレスを「命令アドレス一時リスト」に「追加」する。そして、下記の命令アドレス取得部1cに進む。
【０１３３】
（2） 一方、「シンボルアドレス一時リスト」が空の場合、シンボルアドレス取得部1bは、「機械語命令解析部1の処理」を終了する。
【０１３４】
＜命令アドレス取得部1c＞
（1）「命令アドレス一時リスト」が空でない場合、命令アドレス取得部1cは「命令アドレス一時リスト」から命令アドレスを一つ「取出す」。以降、これを「現命令アドレス」と呼ぶ。
「現命令アドレス」が、「現シンボル情報」の「命令アドレスリスト」に既に存在する場合（現命令アドレスは処理済み）は、以降の処理すべてをスキップし、直ちに命令アドレス取得部1cの処理のスタートに戻る。
一方、「現命令アドレス」が、「現シンボル情報」の「命令アドレスリスト」に存在し

ない場合、「現命令アドレス」を「現シンボル情報」の「命令アドレスリスト」へ「追加」し、下記の次命令情報抽出部1dの処理へ進む。

【０１３５】

（２）一方、「命令アドレス一時リスト」が空の場合、「現シンボル情報」の「命令アドレスリスト」を昇順にソートする。「現シンボル情報」を「シンボル情報リストr1」に「追加」する。

そして、シンボルアドレス取得部1bの処理に戻る。

【０１３６】

＜次命令情報抽出部１ｄ＞

次命令情報抽出部1dは、「現命令アドレス」を基に、「実行バイナリファイル」の「バイナリデータ部」に格納されているバイナリデータを読出し、このバイナリデータを機械語命令と見なす。そして、次命令情報抽出部1dは、この機械語命令をデコード（解釈）して、命令種別に対応した以下（1）～（4）の処理を実行し「次命令情報」を生成し、「次命令情報一時リスト」に「追加」する。

【０１３７】

（1）「単純分岐命令」（命令語のみで分岐先アドレスが特定できる分岐命令）の場合、次命令情報抽出部1dは分岐先アドレスから「分岐型」次命令情報を生成する。

（2）「データ依存分岐命令」（分岐先アドレスがレジスタ値・メモリ値で決定される分岐命令）の場合、命令語のみでは分岐先を直接特定できないので、以下の処理を実行する。

まず、後記の「データ依存分岐情報抽出部2」を実行して、ジャンプテーブル情報リストr3の「ジャンプテーブル情報」を抽出する。そして、次命令情報抽出部1dは、抽出された「ジャンプテーブル情報」の「分岐先アドレスリストr2」に格納されたすべての分岐先アドレスについて、「分岐型」次命令情報を生成する。

【０１３８】

（3）「単純サブルーチン呼出し命令」（命令語のみで呼出し先アドレスが特定できるサブルーチン呼出し命令）の場合、次命令情報抽出部1dは、呼出し先アドレスから「呼出し型」次命令情報を生成する。

（4）「データ依存サブルーチン呼出し命令」（呼出し先アドレスがレジスタ値・メモリ値で決定されるサブルーチン呼出し命令）の場合、次命令情報抽出部1dは、「現シンボル情報」の「データ依存サブルーチン命令フラグ」を「１」に設定する。なお、「データ依存サブルーチン呼出し命令」の呼出し先アドレスの解決は、後述の別手段で行う。

【０１３９】

（5）「直後命令」を実行する可能性のあるすべての命令について：「直後命令」を実行する可能性のある命令は、以下の場合を除いたすべての命令である。

A）無条件に実行される「単純分岐命令」または「データ依存分岐命令」：いずれかの分岐先アドレスが次命令アドレスになるため。

B）リターン命令：サブルーチンから復帰するときの「リターン命令」は、復帰先アドレスが一意に定まらないため。

C）CPU停止命令：CPUの実行動作を停止する命令。

【０１４０】

D）「現シンボル」が「エントリーシンボル」であって、「プログラム終了処理シンボル」（前記のプログラム実行を終了する処理を行うシンボル）への「サブルーチン呼出し命令」の場合：「エントリーシンボル」から 「プログラム終了処理シンボル」を呼出した場合、その「サブルーチン呼出し命令」が「エントリーシンボル」の最後の命令となるため。

E）非機械語命令：命令セットで規定されたいずれの命令にも当てはまらない場合。

上記以外のすべての命令（条件付き分岐命令、条件付きデータ依存分岐命令、サブルーチン呼出し命令を含むことに注意）は、すべて「直後命令」を実行する可能性があるので、「直後命令アドレス」から「直後型」次命令情報を生成する。

その後、アドレスリスト更新部1eに進む。

【０１４１】

＜アドレスリスト更新部1e＞

次命令情報抽出部1dで生成された「次命令情報一時リスト」から、以下のように、各種アドレスリストの更新処理を行う。

（1）「直後型」次命令情報による更新は、「次命令情報一時リスト」から、「直後型」次命令情報をすべて「取出し」、これらを命令アドレス一時リストに追加する。

（2）「分岐型」次命令情報による更新は、「次命令情報一時リスト」から、「分岐型」次命令情報をすべて「取出し」、これらを分岐先アドレスリストr2と命令アドレス一時リストとに「追加」する。

【０１４２】

（3）「呼出し型」次命令情報による更新は、「次命令情報一時リスト」から、「呼出し型」次命令情報をすべて「取出し」、これらをシンボルアドレス一時リストに追加する。シンボルアドレス前処理部1aに説明の方法で抽出した「実行可能シンボル」以外のアドレスをサブルーチン呼出し先アドレスとするサブルーチン呼出し命令が、稀に存在することがあり、この場合に対応するための処理である。呼び出されるすべてのサブルーチンを漏れなく解析することを目的とする。

その後、命令アドレス取得部1cに戻る。

【０１４３】

＜データ依存分岐情報抽出部2の処理の原理＞

データ依存分岐情報を生成するためのデータ依存分岐情報抽出部2の処理原理について説明する。

図8に、switch文を含んだ関数jump_testのサンプルCプログラムを示す。

図8のサンプルCプログラムにおいて、jump_test関数内のswitch文（switch(n)[ case 3: … case 8: … default: …]）が変数nの値に関するデータ依存分岐命令として実装される。

【０１４４】

＜データ依存分岐命令の動作原理＞

データ依存分岐命令は、以下の命令要素とデータ要素を予め準備して動作する。

（1）case文の整数定数の最小値と最大値：図8の場合は、最小値3、最大値8である。以後、min_case_value = 3, max_case_value = 8と表記する。

【０１４５】

（2）ジャンプテーブル：switch文内の各case文に対応する先頭命令アドレスを格納したアドレステーブルである。ジャンプテーブルのサイズは、通常max_case_value － min_case_value + 1（図8の場合8 － 3 + 1 = 6）で与えられる。ジャンプテーブルの最初のエントリーには、case min_case_value: に対応する先頭命令アドレスが格納される。最後のエントリーには、case max_case_value: に対応する先頭命令アドレスが格納されている。

また、min_case_value <= idx <= max_case_valueの範囲の定数idxでcase文に出現しない値については、default文が存在する場合はdefault文に対応する先頭命令アドレスが格納され、default文が存在しない場合はswitch文直後の先頭命令アドレスが格納されている。

【０１４６】

（3）ジャンプテーブルオフセット値：通常、switch(n)の処理において、jt_offset = n － min_case_valueがジャンプテーブルオフセット値として計算される。

（4）ジャンプテーブルオフセット値の範囲判定命令：前記ジャンプテーブルオフセット値jt_offsetについて、jt_offset >= 0かつjt_offset <= max_jt_index = max_case_value － min_case_valueであるかの範囲判定を行う。もし、前記範囲判定が「真」の場合、データ依存分岐先アドレスはジャンプテーブル内に存在し、範囲判定が「偽」の場合は、データ依存分岐先アドレスはジャンプテーブル内に存在しない。

【０１４７】
（5）ジャンプテーブルオフセット値の範囲判定による条件付き分岐命令：前記範囲判定結果が「偽」の時に分岐する条件付き分岐命令である。この時の分岐先アドレスは、default文が存在する場合はdefault文に対応する先頭命令アドレスであり、default文が存在しない場合はswitch文直後の先頭命令アドレスとなる。
【０１４８】
（6）ジャンプテーブル格納メモリ読出し命令とデータ依存分岐命令：ジャンプテーブルオフセット値の範囲判定結果が「真」の場合、ジャンプテーブルが格納されているメモリアドレスjt_addrに対して、(jt_addr + jt_offset * a_size)で計算されるメモリのアドレスに格納されているデータ依存分岐先アドレスを読出す。ここで、a_sizeはアドレス値の語長（ワードサイズのバイト長）であり、32-bit命令セットであれば4（バイト）、64-bit命令セットであれば8（バイト）である。このようにして取得した分岐先アドレス値をPC（プログラムカウンタ）に代入することで、データ依存分岐が実現する。命令セットによっては、ジャンプテーブル格納メモリ読出し命令とデータ依存分岐命令が一つの機械語命令で実装されている場合もある（ARMv5命令セット、x86命令セットなど）。
【０１４９】
＜データ依存分岐命令における関連命令の配置＞
（1）ジャンプテーブルオフセット値の範囲判定命令
（2）ジャンプテーブルオフセット値の範囲判定による条件付き分岐命令
（3）ジャンプテーブル格納メモリ読出し命令
（4）データ依存分岐命令
なお、（3）、（4）が一つの機械語命令となる場合もある（前記）。（2）は（4）の直後に来る場合もある。
【０１５０】
＜データ依存分岐情報抽出部2＞
図9に、データ依存分岐情報抽出部の機能ブロック図を示す。
データ依存分岐情報抽出部2は、以下の一連の処理を行う。データ依存分岐命令のアドレスは既に前記の次命令情報抽出部1d（図7）で与えられているとする。
（1）「ジャンプテーブル格納メモリ読出し命令」抽出部2a
データ依存分岐命令そのものがメモリ読出しを行う場合は、この命令が「ジャンプテーブル格納メモリ読出し命令」である。また、レジスタ値をPCに代入するデータ依存分岐命令の場合は、このデータ依存分岐命令が実行される前のレジスタにメモリロードする命令が「ジャンプテーブル格納メモリ読出し命令」となる。
【０１５１】
（2）ジャンプテーブル格納メモリアドレス解析部2bとジャンプテーブルオフセットレジスタ抽出部2c
ジャンプテーブル格納メモリ読出し命令において、メモリアドレスが「ベースアドレス値（固定アドレス）＋オフセット値」の形式で計算されている場合、「ベースアドレス値」を「ジャンプテーブル格納メモリアドレス値」と特定し、オフセット値を格納するレジスタを「ジャンプテーブルオフセットレジスタ」と特定する。
【０１５２】
なお、「ベースアドレス値」は、「絶対アドレス」（アドレス値そのものの情報を機械語命令が指定する）として与えられる場合と、「PC相対アドレス」（現命令アドレスに加算するオフセット値を機械語命令が指定する）として計算される場合がある。
【０１５３】
なお、「ジャンプテーブル格納メモリアドレス値」と「ジャンプテーブルオフセットレジスタ」のいずれかの特定ができない場合は、データ依存分岐情報抽出が不可能であると判断し、「データ依存分岐情報抽出部2の処理」を終了する。
【０１５４】
（3）ジャンプテーブルオフセット範囲判定命令抽出部2dとジャンプテーブルサイズ抽出

部2e

「ジャンプテーブル格納メモリ読出し命令」が実行される前に位置する「ジャンプテーブルオフセットレジスタに対する範囲判定命令」を特定する。この範囲判定命令の比較対象定数値が前述のmax_jt_indexであり、max_jt_index + 1 = max_case_value - min_case_value + 1がジャンプテーブルサイズであることが特定できる。「ジャンプテーブルオフセットレジスタに対する範囲判定命令」が特定できない場合や、範囲判定命令の比較対象が定数値でない場合は、データ依存分岐情報抽出部2の処理が不可能であると判断し、「データ依存分岐情報抽出部2の処理」を終了する。

【０１５５】

（4）分岐先アドレス情報読出し部2f

前記で特定した「ジャンプテーブル格納メモリアドレス値」と「ジャンプテーブルサイズ」の情報から、ジャンプテーブルに格納される分岐先アドレスをすべて読み出し、分岐先アドレステーブル（機械語アドレステーブル）に格納する。

【０１５６】

（5）ジャンプテーブル情報生成部2g

予め与えられた「データ依存分岐命令」と、上記（1）～（4）の工程で得られた「ジャンプテーブルサイズ」と「分岐先アドレステーブル」からなる「データ依存分岐ジャンプテーブル情報」を生成し、「データ依存分岐情報リスト」に「追加」する。生成したデータ依存分岐ジャンプテーブルの情報は、前記した次命令情報抽出部1dで使用される。

【０１５７】

＜＜機械語命令解析処理の実施例1（ARMv5命令セットの場合）＞＞

図10に、図8のjump_test関数のARMv5命令セットの機械語命令とアセンブリ命令を示す。各行は、「命令アドレス」（16進表記）、「機械語命令データ」（16進表記）、「アセンブリ命令記述」の順で表記されており、PC（プログラムカウンタ）に作用する命令については、その命令種別も表記している。

以下の説明では、「シンボルアドレス一時リスト」から“jump_test”のアドレス0x8340を取得したことを前提に、その後の機械語命令解析処理について説明する。

【０１５８】

表1に、0x8340 － 0x834cまでの機械語命令解析処理を示す。

【表１】

0x8340 - 0x834cまでの機械語命令解析処理

| 処理部 | 処理概要 | 更新される情報・リスト等 |
|---|---|---|
| シンボルアドレス取得部 **1b** | **0x8340**を取得 | 現シンボルアドレス：**0x8340** |
| | | 現シンボルの命令アドレスリスト：{ 空 } |
| | | 命令アドレス一時リスト：**{ 0x8340 }** |
| 命令アドレス取得部 **1c** | **0x8340**を取得 | 現シンボルの命令アドレスリスト：**{ 0x8340 }** |
| | | 命令アドレス一時リスト：{ 空 } |
| 次命令情報抽出部 **1d** | 命令解析：通常命令 | 次命令情報一時リスト：**{(0x8344,“直後”)}** |
| アドレスリスト更新部 **1e** | **0x8344**を追加 | 命令アドレス一時リスト：**{ 0x8344 }** |
| 命令アドレス取得部 **1c** | **0x8344**を取得 | 現シンボルの命令アドレスリスト：**{ 0x8340, 0x8344 }** |
| | | 命令アドレス一時リスト：{ 空 } |
| 次命令情報抽出部 **1d** | 命令解析：通常命令 | 次命令情報一時リスト：**{(0x8348,“直後”)}** |
| アドレスリスト更新部 **1e** | **0x8348**を追加 | 命令アドレス一時リスト：**{ 0x8348 }** |
| 命令アドレス取得部 **1c** | **0x8348**を取得 | 現シンボルの命令アドレスリスト：**{ 0x8340, 0x8344, 0x8348 }** |
| | | 命令アドレス一時リスト：{ 空 } |
| 次命令情報抽出部 **1d** | 命令解析：通常命令 | 次命令情報一時リスト：**{(0x834c,“直後”)}** |
| アドレスリスト更新部 **1e** | **0x834c**を追加 | 命令アドレス一時リスト：**{ 0x8348c }** |
| 命令アドレス取得部 **1c** | **0x834c**を取得 | 現シンボルの命令アドレスリスト：**{ 0x8340, 0x8344, 0x8348, 0x834c }** |
| | | 命令アドレス一時リスト：{ 空 } |
| 次命令情報抽出部 **1d** | 命令解析：データ依存分岐命令 | （データ依存分岐情報抽出部へ） |

【０１５９】

A）最初の部分の機械語命令解析処理では、シンボルアドレス0x8340の命令を基に、次命令情報抽出を逐次的に行っている。0x8340，0x8344，0x8348の各命令はPCに作用しない「通常命令」のため、「直後命令」のみが次命令となる。これら次命令アドレスは、アドレスリスト更新部1eで「命令アドレス一時リスト」に「追加」されたあと、命令アドレス取得部1cで直ちに取出される。

B）0x834cの命令は「データ依存分岐命令」のため、この部分の次命令情報抽出処理は、データ依存分岐情報抽出部2において行われる。

【０１６０】

表2に、0x834cにおけるデータ依存分岐情報抽出部2の処理を示す。

【表２】

0x834cにおけるデータ依存分岐情報抽出処理

| 処理部 | 処理概要 |
|---|---|
| ジャンプテーブル格納メモ読出し命令抽出部 2a | 解析対象命令：**834c:979ff103 ldrls pc, [pc, r3, lsl #2]** |
| | 解析結果：**ldrls**命令は条件付きロード命令であり、**PC**へのロードであるため、「データ依存分岐命令」であると同時に、「ジャンプテーブル格納メモリ読出し命令」でもあることが特定できる。 |
| ジャンプテーブル格納メモリアドレス解析部 2b | 解析対象アドレス式：**[pc, r3, lsl #2] : MEM[pc + (r3 << 2)]** |
| | 解析結果：ベースアドレス値 **= pc** |
| | 解説：ここで、**pc**がロード命令で参照される場合、現命令アドレス**(0x834c)**に**+8**した値が使われる（**ARMv5**命令セット仕様より）。<br>**→ ジャンプテーブル格納メモリアドレス = 0x8354** |
| ジャンプテーブルオフセットレジスタ抽出部 2c | 解析対象アドレス式：**[pc, r3, lsl #2] : MEM[pc + (r3 << 2)]** |
| | 解析結果：オフセット値 **= (r3 << 2) →** オフセットレジスタ **= r3** |
| | 解説： **ARMv5**命令セットは**32-bit**（**4**バイト）アドレスのため、<br>オフセット値が**r3 << 2 = r3 * 4**で与えられる。 |
| ジャンプテーブルオフセット範囲判定命令抽出部 2d<br><br>（「ジャンプテーブル格納メモ読出し命令」が実行されるよりも前に位置する命令を解析する） | 解析対象命令：**8348:e3530005 cmp r3, #5** |
| | 解析結果：「ジャンプテーブル格納メモリ読出し命令」の直前の**cmp**命令は、オフセットレジスタ**r3**と**5**との比較を行っている<br>**→ cmp**命令が「ジャンプテーブルオフセット範囲判定命令」<br>**→ ジャンプテーブルサイズ = 5+1 = 6** |
| | 解説：**cmp**命令の比較内容は直後の**ldrls**命令の実行条件**ls**（**lower-or-same**：符号なし比較）に反映されており、**((unsigned) r3) <= 5**がデータ依存分岐命令の実行条件である。 |
| 分岐先アドレス情報読出し部 2f | 読出し対象メモリ：ジャンプテーブル格納メモリアドレス**(0x8354)**、<br>ジャンプテーブルサイズ**(6)**から、以下の**6**データを読出す<br>**8354:00008384 andeq r8, r0, r4, lsl #7**<br>**8358:0000839c muleq r0, ip, r3**<br>**835c:0000836c andeq r8, r0, ip, ror #6**<br>**8360:000083a4 andeq r8, r0, r4, lsr #7**<br>**8364:0000836c andeq r8, r0, ip, ror #6**<br>**8368:00008384 andeq r8, r0, r4, lsl #7**<br>（注：右部分のアセンブリ命令記述は本来の意味ではない） |
| | 解説：図**8**の**switch**文内の**case**文・**default**文との対応<br>**n == 3 (case 3:)** の分岐先：**0x8384 (f = ftab[1];)**<br>**n == 4 (default:)** の分岐先：**0x839c (return 0;)**<br>**n == 5 (case 5:)** の分岐先：**0x836c (f = ftab[0];)**<br>**n == 6 (case 6:)** の分岐先：**0x83a4 (f = ff1;)**<br>**n == 7 (case 7:)** の分岐先：**0x836c (f = ftab[0];)**<br>**n == 8 (case 8:)** の分岐先：**0x8384 (f = ftab[1];)** |
| ジャンプテーブル情報生成部 2g | 生成されるジャンプテーブル情報：<br>・ データ依存分岐命令アドレス：**0x834c**<br>・ ジャンプテーブルサイズ：**6**<br>・ 分岐先アドレステーブル：**{ 0x8384, 0x839c, 0x836c, 0x83a4, 0x836c, 0x8384 }** |

【０１６１】

データ依存分岐情報抽出部2は、まず0x834cにおける「ジャンプテーブル格納メモリ読

出し命令」がデータ依存分岐命令と同一であることが特定され、そこからジャンプテーブル格納メモリアドレスが0x8354であり、ジャンプテーブルサイズが6であることが特定される。最終的に、アドレス0x8354から6つの分岐先アドレス[0x8384，0x839c，0x836c，0x83a4，0x836c，0x8384]を読出し、「ジャンプテーブル情報」を生成する。

【０１６２】

表３に、0x834cの次命令情報抽出部1dの処理と0x8350の機械語命令解析部1の処理を示す。

【表３】

0x834cの次命令情報抽出処理と0x8350の機械語命令解析処理

| 処理部 | 処理概要 | 更新される情報・リスト等 |
|---|---|---|
| データ依存分岐情報抽出部 **2** | **0x834c**のデータ依存分岐命令のジャンプテーブル情報生成 | 抽出された分岐先アドレステーブル：**{ 0x8384, 0x839c, 0x836c, 0x83a4, 0x836c, 0x8384 }** |
| 次命令情報抽出部 **1d** | ジャンプテーブル情報の分岐先アドレステーブルと、直後命令（データ依存分岐命令が条件付き実行のため）から次命令情報を生成 | 次命令情報一時リスト：**{(0x8384,“分岐”), (0x839c,“分岐”), (0x83a4,“分岐”), (0x836c,“分岐”), (0x8350,”直後”)}** |
| | | 解説：ジャンプテーブル情報の分岐先アドレステーブルでは、同一アドレスが複数回現れることも可能だが、次命令情報一時リストへのアドレス情報の「追加」は「非重複的」であるため、重複する次命令情報は追加されない。 |
| アドレスリスト更新部 **1e** | 分岐型次命令の更新 | 分岐先アドレスリスト：**{ 0x8384, 0x839c, 0x836c, 0x83a4 }** |
| | 分岐型・直後型次命令の更新 | 命令アドレス一時リスト：**{ 0x8384, 0x839c, 0x836c, 0x83a4, 0x8350 }** |
| 命令アドレス取得部 **1c** | **0x8350**を取得 | 現シンボル命令アドレスリスト：**{ 0x8340, 0x8344, 0x8348, 0x834c, 0x8350 }** |
| | | 命令アドレス一時リスト：**{ 0x8384, 0x839c, 0x836c, 0x83a4 }** |
| 次命令情報抽出部 **1d** | 命令解析：無条件分岐 | 次命令情報一時リスト：**{(0x839c,“分岐”)}** |
| アドレスリスト更新部 **1e** | **0x839c**を追加 | 解説：**0x839c**は分岐先アドレスリストと命令アドレス一時リストにいずれも存在しているので、両リストは更新されない。 |

【０１６３】

A）0x834cの命令に対してデータ依存分岐情報抽出部2で生成されたジャンプテーブル情報から6つの分岐先アドレスを取得し、さらに0x834cのデータ依存分岐命令が条件付き実行命令（ldrls）であるため、直後命令（0x8350）も次命令として次命令情報一時リスト

を生成する。ここで、6つの分岐先アドレスには重複するアドレスも含まれるが、次命令情報一時リストには、重複する次命令情報は追加されないことに注意する。アドレスリスト更新部1eでは、4つの（重複しない）分岐先アドレスが「分岐先アドレスリスト」に「追加」され、5つの次命令アドレス（分岐型・直後型）が「命令アドレス一時リスト」に「追加」される。

【０１６４】

B）次に、命令アドレス取得部1cは、命令アドレス一時リストの最後尾アドレス0x8350を「取出」す。この命令が「無条件分岐命令」であることから、次命令情報抽出部1dは、分岐先アドレス0x839cを次命令アドレスとして抽出する。そして、アドレスリスト更新部1eが分岐先アドレス0x839cを「分岐先アドレスリスト」と「命令アドレス一時リスト」に追加する。ここでは、0x839cはいずれのリストにも存在するため、これらリストは更新されない。

【０１６５】

表4に、0x83a4 － 0x83b4までの機械語命令解析処理を示す。

【表４】

0x83a4 - 0x83b4までの機械語命令解析処理

| 処理部 | 処理概要 | 更新される情報・リスト等 |
|---|---|---|
| 命令アドレス<br>取得部 **1c** | **0x3a4**を取得<br>（ジャンプテーブル分岐先） | 現シンボルの命令アドレスリスト：**{ 0x8340, 0x8344, 0x8348, 0x834c, 0x8350, 0x83a4 }** |
| | | 命令アドレス一時リスト：**{ 0x8384, 0x839c, 0x836c }** |
| 次命令情報抽出部 **1d** | 命令解析：通常命令 | 次命令情報一時リスト：**{(0x83a8,“直後”)}** |
| アドレスリスト<br>更新部 **1e** | **0x83a8**を追加 | 命令アドレス一時リスト：**{ 0x8384, 0x839c, 0x836c, 0x83a8 }** |
| **0x83a8 – 0x83b0**の処理：省略<br>（**0x83ac**の「データ依存サブルーチン呼出し命令」は「通常命令」と同じ扱いをする） | | |
| 命令アドレス<br>取得部 **1c** | **0x83b4**を取得 | 現シンボルの命令アドレスリスト：**{ 0x8340, 0x8344, 0x8348, 0x834c, 0x8350, 0x83a4, 0x83a8, 0x83ac, 0x83b0, 0x83b4 }** |
| | | 命令アドレス一時リスト：**{ 0x8384, 0x839c, 0x836c }** |
| 次命令情報抽出部 **1d** | 命令解析：リターン | 次命令情報一時リスト：**{** 空**}→**リスト更新なし |

【０１６６】

これらのアドレスの命令は「直後命令」のみを次命令とする「通常命令」が続く。最後

のアドレス0x83b4はリターン命令であり、次命令が存在しないので、命令アドレス一時リストは更新されない。なお、途中のアドレス0x83a8，0x83acで以下の2つの命令が出現する。

【０１６７】

表5に、0x83ac、0x83a8の機械語命令とＣ言語命令を示す。

【表５】

| | | | | |
|---|---|---|---|---|
| **83a8:** | **e1a0e00f** | **mov** | **lr, pc** | **(lr ← 0x83a8 + 8)** |
| **83ac:** | **e12fff13** | **bx** | **r3** | **(**データ依存サブルーチン呼出し**)** |

ここで、0x83acはレジスタr3の値にジャンプする「データ依存分岐命令」であるが、その直前の0x83a8に、pc（実際の値は現命令アドレス0x83a8 + 8）をリンクレジスタlrに代入する命令があるため、0x83acの命令は実際には「データ依存サブルーチン呼出し命令」であることが判別できる。

【０１６８】

図8のサンプルCプログラムで、func_pointer f; ... f(n)の部分に対応しており、変数fは関数ポインタ（変数）であり、f(n)は関数ポインタを介したサブルーチン呼出しである。「データ依存分岐命令」における「ジャンプテーブル情報」は、メモリ上の固定化されたデータであるため抽出可能であるのに比べ、「データ依存サブルーチン呼出し命令」における「呼出し先アドレス」は、プログラム実行中に設定されることもあり、呼出し先アドレスの抽出は一般的には困難である。

【０１６９】

従って、ここでは、データ依存サブルーチン呼出し命令の次命令情報は「直後命令」のみとし、データ依存サブルーチン呼出し先アドレスの解決は、後述の別手段で対応する。つまり、「データ依存サブルーチン呼出し命令」の次命令情報抽出処理は、「通常命令」と同じ扱いとなる。

【０１７０】

表6に、その他の部分の機械語命令解析処理を示す。

【表６】

その他部分の機械語命令処理

| 処理部 | | 更新される情報・リスト等 |
|---|---|---|
| 0x836直前 | 現シンボルの命令アドレスリスト | { 0x8340, 0x8344, 0x8348, 0x834c, 0x8350, 0x83a4, 0x83a8, 0x83ac, 0x83b0, 0x83b4 } |
| | 命令アドレス一時リスト | { 0x8384, 0x839c, 0x836c } ← 末尾取出し |
| A | 解析対象命令アドレス | 0x836c →(直後) 0x8370 →(直後) 0x8374 →(直後) 0x8378 →(直後) 0x837c →(直後) 0x8380 (リターン) |
| | 現シンボルの命令アドレスリスト | { 0x8340, 0x8344, 0x8348, 0x834c, 0x8350, 0x83a4, 0x83a8, 0x83ac, 0x83b0, 0x83b4, 0x836c, 0x8370, 0x8374, 0x8378, 0x837c, 0x8380 } |
| | 命令アドレス一時リスト | { 0x8384, 0x839c } ←末尾取出し |
| B | 解析対象命令アドレス | 0x839c →(直後) 0x83a0 (リターン) |
| | 現シンボルの命令アドレスリスト | { 0x8340, 0x8344, 0x8348, 0x834c, 0x8350, 0x83a4, 0x83a8, 0x83ac, 0x83b0, 0x83b4, 0x836c, 0x8370, 0x8374, 0x8378, 0x837c, 0x8380, 0x839c, 0x83a0 } |
| | 命令アドレス一時リスト | { 0x8384 } ←末尾取出し |
| C | 解析対象命令アドレス | 0x8384 →(直後) 0x8388 →(直後) 0x838c →(直後) 0x8390 →(直後) 0x8394 →(直後) 0x8398 (リターン) |
| | 現シンボルの命令アドレスリスト | { 0x8340, 0x8344, 0x8348, 0x834c, 0x8350, 0x83a4, 0x83a8, 0x83ac, 0x83b0, 0x83b4, 0x836c, 0x8370, 0x8374, 0x8378, 0x837c, 0x8380, 0x839c, 0x83a0, 0x8384, 0x8388, 0x838c, 0x8390, 0x8394, 0x8398 } |
| | 命令アドレス一時リスト | { 空 } → シンボル内全命令解析終了 |
| D | 昇順ソート後の現シンボルの命令アドレスリスト | { 0x8340, 0x8344, 0x8348, 0x834c, 0x8350, 0x836c, 0x8370, 0x8374, 0x8378, 0x837c, 0x8380, 0x8384, 0x8388, 0x838c, 0x8390, 0x8394, 0x8398, 0x839c, 0x83a0, 0x83a4, 0x83a8, 0x83ac, 0x83b0, 0x83b4 } |

【０１７１】

解析対象命令アドレス0x836以降の機械語命令解析処理では、表4と同様の命令系列（複数の「通常命令」と「リターン命令」）に対する命令解析処理が続く。

解析対象命令アドレス0x836直前の命令アドレスリストと命令アドレス一時リストとは、表6の上欄に示す如くである。

【０１７２】

表6のA欄の解析対象命令アドレス（0x836c - 0x8380）では、命令アドレス一時リストの末尾アドレス0x836cを取出し、アドレス0x836cの命令から、次命令（直後命令）アドレスを連続的に解析していき、0x8380のリターン命令（次命令なし）に到達する。

【０１７３】

表6のB欄の解析対象命令アドレス（0x839c - 0x83a0）では、命令アドレス一時リストの末尾アドレス0x839cを取出し、アドレス0x839cの命令から、次命令（直後命令）アドレスを解析し、0x83a0のリターン命令（次命令なし）に到達する。

【０１７４】

表6のC欄の解析対象命令アドレス（0x8384 - 0x8398）では、命令アドレス一時リストの末尾アドレス0x8384を取出し、アドレス0x8384の命令から、次命令（直後命令）アドレスを連続的に解析していき、0x8398のリターン命令（次命令なし）に到達する。

【０１７５】

表6のC欄の最後で、命令アドレス取得部1cで、「命令アドレス一時リスト」が「空」の状態（シンボル内全命令解析終了）を検知し、「現シンボル情報」の「命令アドレスリスト」を昇順にソートする。

【０１７６】

＜＜機械語命令解析処理の実施例２（x86(64-bit)命令セットの場合）＞＞

図11に、jump_test関数のx86(64-bit)命令セットの機械語命令とアセンブリ命令を示す。各行は、左から、「命令アドレス」（６桁の16進表記）、「機械語命令データ」（左から右への順番で16進数2桁ごとに区切られている）、「アセンブリ命令記述」の順で表記されている。なお、PC（プログラムカウンタ）に作用する命令については、その命令種別も表記している。

【０１７７】

以下の説明では、シンボルアドレス取得部１ｂが「シンボルアドレス一時リスト」からjump_testのアドレス0x400560を取得したことを前提に、その後の機械語命令解析処理について説明する。

【０１７８】

表7に、図11のjump_test関数の0x400560 － 0x40056eまでの機械語命令解析処理部１の処理を示す。

【表７】

0x400560 - 0x40056eまでの機械語命令解析処理

| 処理部 | 処理概要 | 更新される情報・リスト等 |
|---|---|---|
| シンボルアドレス取得部**1b** | **0x400560**を取得 | 現シンボルアドレス：**0x400560** |
| | | 現シンボルの命令アドレスリスト：{ 空 } |
| | | 命令アドレス一時リスト：{ **0x400560** } |
| 命令アドレス取得部**1c** | **0x400560**を取得 | 現シンボルの命令アドレスリスト：{**0x400560**} |
| | | 命令アドレス一時リスト：{ 空 } |
| 次命令情報抽出部**1d** | 命令解析：通常命令 | 次命令情報一時リスト：{(0x400563,“直後”)} |
| アドレスリスト更新部**1e** | **0x400563**を追加 | 命令アドレス一時リスト：{ **0x400563** } |
| 命令アドレス取得部**1c** | **0x400563**を取得 | 現シンボルの命令アドレスリスト：{ **0x400560, 0x400563** } |
| | | 命令アドレス一時リスト：{ 空 } |
| 次命令情報抽出部**1d** | 命令解析：通常命令 | 次命令情報一時リスト：{(0x400567,“直後”)} |
| アドレスリスト更新部**1e** | **0x400567**を追加 | 命令アドレス一時リスト：{ **0x400567** } |
| 命令アドレス取得部**1c** | **0x400567**を取得 | 現シンボルの命令アドレスリスト：{ **0x400560, 0x400563, 0x400567** } |
| | | 命令アドレス一時リスト：{ 空 } |
| 次命令情報抽出部**1d** | 命令解析：通常命令 | 次命令情報一時リスト：{(0x40056a,“直後”)} |
| アドレスリスト更新部**1e** | **0x40056a**を追加 | 命令アドレス一時リスト：{ **0x40056a** } |
| 命令アドレス取得部**1c** | **0x40056a**を取得 | 現シンボルの命令アドレスリスト：{ **0x400560, 0x400563, 0x400567, 0x40056a** } |
| | | 命令アドレス一時リスト：{ 空 } |
| 次命令情報抽出部**1d** | 命令解析：条件付き分岐命令 | 次命令情報一時リスト：{(0x400575,“分岐”), (0x40056c,“直後”)} |
| アドレスリスト更新部**1e** | 分岐先アドレス追加 | 分岐先アドレスリスト：{ **0x400575** } |
| | 次命令アドレス追加 | 命令アドレス一時リスト：{ **0x400575, 0x40056c** } |
| 命令アドレス取得部**1c** | **0x40056c**を取得 | 現シンボルの命令アドレスリスト：{ **0x400560, 0x400563, 0x400567, 0x40056a, 0x40056c** } |
| | | 命令アドレス一時リスト：{ **0x400575** } |
| 次命令情報抽出部**1d** | 命令解析：通常命令 | 次命令情報一時リスト：{(0x40056e,“直後”)} |
| アドレスリスト更新部**1e** | **0x40056e**を追加 | 命令アドレス一時リスト：{ **0x400575, 0x40056e** } |
| 命令アドレス取得部**1c** | **0x40056e**を取得 | 現シンボルの命令アドレスリスト：{ **0x400560, 0x400563, 0x400567, 0x40056a, 0x40056c, 0x40056e** } |
| | | 命令アドレス一時リスト：{ **0x400575** } |
| 次命令情報抽出部**1d** | 命令解析：データ依存分岐命令 | （データ依存分岐情報抽出部へ） |

【０１７９】

A）最初は、シンボルアドレス0x400560の命令を基に、次命令情報抽出を逐次的に行っており、0x400560，0x400563，0x400567の各命令はPCに作用しない「通常命令」のため、

「直後命令」のみが次命令となる。
B）0x40056aの命令は「条件付き分岐命令」のため、分岐先アドレス0x400575と直後命令アドレス0x40056cが次命令アドレスとなる。
C）0x40056eの命令は「データ依存分岐命令」のため、この部分の次命令情報抽出部1dの処理は、データ依存分岐情報抽出部2において行われる。
【０１８０】
表8に、図11の0x40056eにおけるデータ依存分岐情報抽出処理を示す。

【表８】

0x40056eにおけるデータ依存分岐情報抽出処理

| 処理部 | 処理概要 |
|---|---|
| ジャンプテーブル格納メモ読出し命令抽出部 2a | 解析対象命令：**40056e:ff 24 c5 38 07 40 00 jmpq *0x400738(,%rax,8)** |
| | 解析結果：この**jmpq**命令はメモリデータによって分岐先を決定する「データ依存分岐命令」であると同時に、「ジャンプテーブル格納メモリ読出し命令」でもあることが特定できる。 |
| ジャンプテーブル格納メモリアドレス解析部 2b | 解析対象アドレス式：***0x400738(,%rax,8) : MEM[0x400738 + (rax*8)]*** |
| | 解析結果：ベースアドレス値＝0x400738<br>→ ジャンプテーブル格納メモリアドレス **= 0x400738** |
| ジャンプテーブルオフセットレジスタ抽出部 2c | 解析対象アドレス式：***0x400738(,%rax,8) : MEM[0x400738 + (rax*8)]*** |
| | 解析結果：オフセット値 **= (rax*8)** → オフセットレジスタ **= rax** |
| | 解説：**x86(64-bit)**命令セットは**64-bit**（8バイト）アドレスのため、オフセット値が**rax*8**で与えられる。 |
| ジャンプテーブルオフセット範囲判定命令抽出部 2d<br><br>（「ジャンプテーブル格納メモ読出し命令」が実行されるよりも前に位置する命令を解析する） | 解析対象命令1：**40056c:89 c0　　mov　　%eax,%eax** |
| | 解析結果：「ジャンプテーブル格納メモリ読出し命令」の直前の**mov**命令はオフセットレジスタ**eax**（**64-bit**レジスタ**rax**の下位**32-bit**）に作用し、下位**32-bit**はそのままにして、上位**32-bit**をクリアする動作を行う→オフセット範囲判定命令ではない |
| | 解析対象命令２：**40056a:77 09　　ja　　400575** |
| | 解析結果：前記**mov**命令の直前の**ja**命令は、ジャンプテーブルの範囲外の場合に、データ依存分岐命令を回避するための条件付き分岐命令である。→オフセット範囲判定命令ではない |
| | 解析対象命令3：**400567:83 f8 05　　cmp　　$0x5,%eax** |
| | 解析結果：前記**ja**命令の直前の**cmp**命令は、オフセットレジスタ**eax**と**5**との比較を行っている<br>→ **cmp**命令が「ジャンプテーブルオフセット範囲判定命令」<br>→ ジャンプテーブルサイズ **= 5+1 = 6** |
| | 解説：**cmp**命令の比較内容は**2**命令後の**ja**命令の分岐条件（**above**：符号なし比較）に反映されており、**((unsigned) eax > 5)**が**ja**の分岐条件である。 |
| 分岐先アドレス情報読出し部 2f | 読出し対象メモリ：ジャンプテーブル格納メモリアドレス**(0x400738)**、ジャンプテーブルサイズ**(6)**から、以下の**6**データを読出す<br>**400738:91 05 40 00 00 00 00 00　　0000000000400591**<br>**400740:75 05 40 00 00 00 00 00　　0000000000400575**<br>**400748:80 05 40 00 00 00 00 00　　0000000000400580**<br>**400750:9a 05 40 00 00 00 00 00　　000000000040059a**<br>**400758:80 05 40 00 00 00 00 00　　0000000000400580**<br>**400760:91 05 40 00 00 00 00 00　　0000000000400591** |
| | 解説：図**8**の**switch**文内の**case**文・**default**文との対応<br>**n == 3 (case 3:)** の分岐先：**0x400591 (f = ftab[1];)**<br>**n == 4 (default:)** の分岐先：**0x400575 (return 0;)**<br>**n == 5 (case 5:)** の分岐先：**0x400580 (f = ftab[0];)**<br>**n == 6 (case 6:)** の分岐先：**0x40059a (f = ff1;)**<br>**n == 7 (case 7:)** の分岐先：**0x400580 (f = ftab[0];)**<br>**n == 8 (case 8:)** の分岐先：**0x400591 (f = ftab[1];)** |
| ジャンプテーブル情報生成部 2g | 生成されるジャンプテーブル情報：<br>・　データ依存分岐命令アドレス：**0x40056e**<br>・　ジャンプテーブルサイズ：**6**<br>・　分岐先アドレステーブル：**{ 0x400591, 0x400575, 0x400580, 0x40059a, 0x400580, 0x400591 }** |

【０１８１】

ここでは、まず「ジャンプテーブル格納メモリ読出し命令」がデータ依存分岐命令と同一であることが特定され、そこからジャンプテーブル格納メモリアドレスが0x400738であ

り、ジャンプテーブルサイズが6であることが特定される。最終的に、アドレス0x400738から６つの分岐先アドレス[0x400591，0x400575，0x400580，0x40059a，0x400580，0x400591]を読出し、「ジャンプテーブル情報」を生成する。

【０１８２】

表9に、0x40056eの次命令情報抽出処理を示す。

【表９】

0x40056eの次命令情報抽出処理

| 処理部 | 処理概要 | 更新される情報・リスト等 |
|---|---|---|
| データ依存分岐情報抽出部 **2** | **0x40056e**のデータ依存分岐命令のジャンプテーブル情報生成 | ジャンプテーブル情報の分岐先アドレスリスト：**{ 0x400591, 0x400575, 0x400580, 0x40059a, 0x400580, 0x400591 }** |
| 次命令情報抽出部 **1d** | ジャンプテーブル情報の分岐先アドレステーブルから次命令情報を生成 | 次命令情報一時リスト：**{(0x400591,“分岐”), (0x400575,“分岐”), (0x400580,“分岐”), (0x40059a,“分岐”) }** |
| | | 解説：ジャンプテーブル情報の分岐先アドレステーブルでは、同一アドレスが複数回現れることも可能だが、次命令情報一時リストへのアドレス情報の「追加」は「非重複的」であるため、重複する次命令情報は追加されない。 |
| アドレスリスト更新部 **1e** | 分岐型次命令の更新 | 分岐先アドレスリスト：**{ 0x400575, 0x400591, 0x400580, 0x40059a }** |
| | 分岐型次命令の更新 | 命令アドレス一時リスト：**{ 0x400575, 0x400591, 0x400580, 0x40059a }** |

0x40056eの命令に対してデータ依存分岐情報抽出部2で生成されたジャンプテーブル情報から6つの分岐先アドレスを取得し、次命令情報一時リストを生成する。ここで、６つの分岐先アドレスには重複するアドレスも含まれるのが、次命令情報一時リストには、重複する次命令情報は追加されないことに注意する。アドレス更新部では、４つの（重複しない）分岐先アドレスが「分岐先アドレスリストr2」と「命令アドレス一時リスト」に「追加」されるが、次命令アドレス0x400575は、両リストに存在するので、実際には３つのアドレスが追加される。

【０１８３】

表10に、その他の部分の機械語命令解析部1の処理を示す。

【表１０】

その他部分の機械語命令処理

| 処理部 | | 更新される情報・リスト等 |
|---|---|---|
| | 現シンボルの<br>命令アドレスリスト | { 0x400560, 0x400563, 0x400567, 0x40056a, 0x40056c, 0x40056e } |
| | 命令アドレス一時リスト | { 0x400575, 0x400591, 0x400580, 0x40059a } ←末尾アドレス取出し |
| A | 機械語命令解析対象アドレス | 0x40059a →(直後) 0x40059f →(分岐) 0x400587 →(直後) 0x400589 →(直後) 0x40058d →(直後) 0x400590 (リターン) |
| | 現シンボルの<br>命令アドレスリスト | { 0x400560, 0x400563, 0x400567, 0x40056a, 0x40056c, 0x40056e, 0x40059a, 0x40059f, 0x400587, 0x400589, 0x40058d, 0x400590 } |
| | 分岐先アドレスリスト r2 | { 0x400575, 0x400591, 0x400580, 0x40059a, 0x400587 } |
| | 命令アドレス一時リスト | { 0x400575, 0x400591, 0x400580 } ←末尾取出し |
| B | 機械語命令解析対象アドレス | 0x400580 →(直後) 0x400587 (処理済み) |
| | 現シンボルの<br>命令アドレスリスト | { 0x400560, 0x400563, 0x400567, 0x40056a, 0x40056c, 0x40056e, 0x40059a, 0x40059f, 0x400587, 0x400589, 0x40058d, 0x400590, 0x400580 } |
| | 命令アドレス一時リスト | { 0x400575, 0x400591 } ←末尾取出し |
| C | 機械語命令解析対象アドレス | 0x400591 →(直後) 0x400598 →(分岐) 0x400587 (処理済み) |
| | 現シンボルの<br>命令アドレスリスト | { 0x400560, 0x400563, 0x400567, 0x40056a, 0x40056c, 0x40056e, 0x40059a, 0x40059f, 0x400587, 0x400589, 0x40058d, 0x400590, 0x400580, 0x400591, 0x400598 } |
| | 命令アドレス一時リスト | { 0x400575 } ←末尾取出し |
| D | 機械語命令解析対象アドレス | 0x400575 →(直後) 0x400577 →(直後) 0x40057b (リターン) |
| | 現シンボルの<br>命令アドレスリスト | { 0x400560, 0x400563, 0x400567, 0x40056a, 0x40056c, 0x40056e, 0x40059a, 0x40059f, 0x400587, 0x400589, 0x40058d, 0x400590, 0x400580, 0x400591, 0x400598, 0x400575, 0x400577, 0x40057b } |
| | 命令アドレス一時リスト | { 空 } → シンボル内全命令解析終了 |
| | 昇順ソート後の命令アドレス<br>リスト | { 0x400560, 0x400563, 0x400567, 0x40056a, 0x40056c, 0x40056e, 0x400575, 0x400577, 0x40057b, 0x400580, 0x400587, 0x400589, 0x40058d, 0x400590, 0x400591, 0x400598, 0x40059a, 0x40059f } |

0x40059aの直前には、命令アドレスリスト、命令アドレス一時リストとは、表10の上欄に示す如くである。

【０１８４】

0x40059aの以降の機械語命令解析処理では、一つの次命令を伴う「通常命令」、「無条件分岐命令」、「データ依存サブルーチン呼出し命令」のいずれかが続いた後に、「リターン命令」または「処理済み命令」に到達する命令系列に対する命令解析処理が続く。なお、「データ依存サブルーチン呼出し命令」の呼出し先アドレスの解決については、別手段で行う。

【０１８５】

表10のA欄の解析対象命令アドレス（0x40059a，0x40059f，0x400587，0x400589，0x40058d，0x400590）では、命令アドレス一時リストの末尾アドレス0x40059aを取出し、次命令アドレス（直後型、分岐型）を連続的に解析していき、0x400590のリターン命令（次命令なし）に到達する。

【０１８６】

表10のB欄の解析対象命令アドレス（0x400580，0x400587）では、命令アドレス一時リストの末尾アドレス0x400580を取出し、次命令アドレスを解析し、直後の0x400587が「処理済み」（現シンボルの命令アドレスリストに存在する）と判定される。

【０１８７】

表10のC欄の解析対象命令アドレス（0x400591，0x400598，0x400587）：命令アドレス一時リストの末尾アドレス0x400591を取出し、次命令アドレス（直後型、分岐型）を連続的に解析していき、0x400587が「処理済み」（現シンボルの命令アドレスリストに存在する）と判定される。

【０１８８】

表10のD欄の解析対象命令アドレス（0x400575，0x400577，0x40057b）：命令アドレス一時リストの末尾アドレス0x400575を取出し、次命令アドレス（直後型）を連続的に解析していき、0x40057bのリターン命令（次命令なし）に到達する。

【０１８９】

表10のD欄の最後で「命令アドレス取得部1c」で「命令アドレス一時リスト」が「空」の状態（シンボル内全命令解析終了）を検知し、「現シンボル情報」の「命令アドレスリスト」を昇順にソートする。

【０１９０】

＜命令セットシミュレータプログラムソースコード出力部3＞

図12に、命令セットシミュレータの構成を、解析対象の実行バイナリファイル、命令セットシミュレータが出力する命令セットシミュレータプログラムソースコードとともに示す。

命令セットシミュレータSは、機械語命令解析部１と命令セットシミュレータプログラムソースコード出力部3とを備えている。

【０１９１】

命令セットシミュレータプログラムソースコード出力部3は、機械語命令解析部2で抽出された情報をもつシンボル情報リストr1、分岐先アドレスリストr2、ジャンプテーブル情報リストr3から命令セットシミュレータプログラムソースコードを出力して生成する。

命令セットシミュレータプログラムソースコード出力部3は、サブルーチンプログラムソース生成部3aと、シンボルアドレス情報テーブルソース生成部3bと、メモリ初期化プログラム記述生成部3cと、メイン関数プログラム記述生成部3dとを有している。

【０１９２】

命令セットシミュレータプログラムソースコード4は、サブルーチンプログラムソース4a、シンボルアドレス情報テーブルソース4b、メイン関数ソース4c、メモリ初期化処理ソース4dと、さらに予め準備された機械語命令プログラム生成マクロ定義ソース4eとを有している。

【０１９３】

＜命令セットシミュレータプログラムにおけるCPUリソースのデータ構造記述＞

ここでは、「命令セットシミュレータプログラムソースコード4」を生成するための準備として必要となるCPUリソースのデータ構造記述について説明する。

【０１９４】
ここで、「CPUリソース」とは、メモリとCPUレジスタ（汎用レジスタ、内部状態レジスタ等）を指し、これら「CPUリソース」の状態が命令セットシミュレータプログラム上で、実際のハードウエアとしてCPU動作を忠実にシミュレータプログラム上で再現させる必要がある。図13には、ARMv5用命令セットシミュレータプログラムにおけるCPUリソースのデータ構造記述の一例を示し、図14には、CPUリソース参照用マクロ定義と命令実行条件判定用マクロ定義を示す。
【０１９５】
（1）メモリ：char型（8ビット）の配列として宣言している。MEM_SIZE（メモリサイズ）は、実行バイナリファイルに含まれる「プログラムヘッダーテーブル」等から得られる情報から必要なメモリサイズ（必要なスタック領域も確保する）を特定する。
【０１９６】
（2）汎用レジスタ：16個の32ビット配列（符号なし）として宣言している。その中で、r[13]はスタックポインタ（sp）として、r[14]はリンクレジスタ（lr：サブルーチンの復帰アドレスを格納）として、r[15]はプログラムカウンタ（pc）として使用される。
【０１９７】
（3）状態レジスタ：cr，cc，cv，cls，cge，cleからなる６つの３２ビット変数（符号なし）として宣言している。これらは、ARMv5命令における「命令実行条件」を判定するためのCPUの状態を示す情報に使われる。実際のARMv5命令セットを実行するCPUでは、通常４つの内部レジスタビット（Z：zero，N：negative，V：overflow，C：carry（けた上がり））の組合せによって14種類の実行条件を判断する機構が実装される。リソース記述として、同様の4ビットによる状態レジスタを実現することも可能であるが、ここでは、6つの３２ビット変数として宣言することによって、「命令実行条件」の判定のためのソフトウエア上での演算量を削減することを意図している（後述）。
【０１９８】
（4）CPUリソース参照用マクロ定義：
A）_r_(n)： n番目の汎用レジスタを参照するためのマクロ
B）_R_,_C_,_V_,_LS_,_LE_,_GE_：６つの状態レジスタ変数（cr，cc，cv，cls，cge，cle）を参照するためのマクロ
C）_m8_，_m16_，_m32_：メモリから８ビットデータ、１６ビットデータ、３２ビットデータをそれぞれ「読出し」・「書込み」するためのマクロ
【０１９９】
（5）命令実行条件判定用マクロ定義：前記の６つの状態レジスタ変数参照マクロによる命令実行条件判定をプログラム記述上で行う。命令実行条件は以下の１５種類である。（下記は、通常のZ/N/V/Cの4ビットによる状態レジスタの条件判定式と図14の６つの状態レジスタ変数参照マクロによる条件判定式を比較して記載している。）
A）AL（無条件）：常に命令を実行する。
B）EQ（equal条件）、NE（not equal条件）
1）Z/N/V/C判定式：EQ(Z==1)，NE(Z==0)
2）6状態変数判定式：EQ(_R_==0)，NE(_R_!=0)
【０２００】
C）CS（carry set条件）、CC（carry clear条件）
1）Z/N/V/C判定式：CS(C==1)，CC(C==0)
2）6状態変数判定式：CS(_C_!=0)，CC(_C_==0)
D）MI（minus条件）、PL（plus条件）
1）Z/N/V/C判定式：MI(N==1)，PL(N==0)
2）6状態変数判定式：MI(((S32)_R_)<0)，PL(((S32)_R_)>=0)
【０２０１】
E）VS（overflow条件）、VC（no overflow条件）
1）Z/N/V/C判定式：VS(V==1)，VC(V==0)

2）6状態変数判定式：VS(((S32)_V_)<0), VC(((S32)_V_)>=0)
F）HI（higher条件）、LS（lower or same条件）：符号なし
1）Z/N/V/C判定式：HI(C==1 && Z==0), LS(C==0 || Z==1)
2）6状態変数判定式：HI(_LS_==0), LS(_LS_!=0)

【０２０２】

G）GE（greater or equal条件）、LT（less than条件）：符号付き
1）Z/N/V/C判定式：GE(N==V), LT(N!=V)
2）6状態変数判定式：GE(_GE_!=0), LT(_GE_==0)
H）GT（greater than条件）、LE（less or equal条件）：符号付き
1）Z/N/V/C判定式：GT(Z==0 && N==V), LE(Z==1 && N!=V)
2）6状態変数判定式：GT(_LE_==0), LE(_LE_!=0)

【０２０３】

通常のZ/N/V/Cの４ビットによる状態レジスタの場合、ALを除く14種類の命令実行条件判定には、1～3ビットからなる判定式であるのに対し、６状態変数判定式では、それぞれ１つの状態変数による判定式であり、プログラム実行時により少ない演算数で判定できる利点がある。

【０２０４】

＜機械語命令プログラム生成マクロ定義ソース4e＞

「機械語命令プログラム生成マクロ」とは、幾つかの引数（変数、定数、文字列）に基づいて、実行プログラム記述を部分的に生成する機能を持つプログラミング記述手法のことである。

【０２０５】

この「機械語命令プログラムマクロ」は使用されるすべての機械語命令種別それぞれについて定義する（このマクロ定義は、当然のことながら、ターゲットCPUの命令セットに特化した記述となる）。 なお、機械語命令プログラム生成マクロ定義ソース４ｅは、解析対象の「実行バイナリファイル」には依存しない情報であるため、事前に準備しておく。

図15に示す４つの機械語命令に対応する「機械語命令プログラムマクロ」呼出し記述を図16に示す。以降、各機械語命令プログラムマクロについて説明する。

【０２０６】

＜SUBマクロ記述（sub命令）とCMPマクロ記述（cmp命令）＞

図17に、SUBマクロ定義とCMPマクロ定義を示す。

sub命令に対応してSUBマクロを呼び、cmp命令に対応してCMPマクロを呼んでいる。

【０２０７】

（１）SUBマクロとCMPマクロで使われる引数：
・c（実行条件を表す文字列）：Z/N/V/Cの４ビットによる状態レジスタ
・v（命令ビットフィールド）：主に内部レジスタの更新条件を示す命令属性
・rd（出力レジスタID）：計算結果を格納するレジスタ番号
・rn（第１オペランドレジスタID）：第１オペランドを格納するレジスタ番号
・sh（シフト属性）：第２オペランドに対するシフト処理属性
・rm（第２オペランドレジスタID）：第２オペランドを格納するレジスタ番号
・val（即値データ）：第２オペランドが即値（定数）の場合の即値データ
・pc（命令アドレス）：ここでは命令動作プログラムには関係ない

【０２０８】

（2）図18に示す_SUB_マクロ：SUBマクロとCMPマクロがさらに呼出す内部マクロである。
ここで、SUBマクロとCMPマクロで使われる引数の意味は以下の通りである。
・w_rd（出力属性）：１の場合出力レジスタに格納し、０の場合格納しない。
・cin（キャリー入力）：減算のキャリー入力
・swap_op（オペランド入れ替え属性）：１の場合第１・第２オペランドを入れ替え、０の場合はそのまま。

・c，v，rd，rn，sh，rm，val：SUB/CMPマクロ引数と同じ意味
【０２０９】
（3）_COND_マクロ：_SUB_マクロがさらに呼出す内部マクロであり、図14に定義されている。「_COND_(c)」は「if　_COND___c___」としてマクロ定義されている。「__」は「文字列連結プリプロセッサ」であるため、例えば_COND_(AL)は「if　_COND_AL_」に変換され、さらに「_COND_AL_」はそのマクロ定義により「(1)」に変換される。
【０２１０】
（4）状態レジスタ変数更新記述：_fS_(v)が１の場合、６つの状態レジスタ変数が更新される。
【０２１１】
（5）_SUB_マクロ内部で呼ばれるその他のマクロを図19に示す。
・IMM(rm，val)：図16のSUBマクロ呼出しとCMPマクロ呼出しでは、これらのマクロ引数shとして「IMM」が与えられている。図18における「sh(rm，val)」の記述は、「IMM(rm，val)」に変換され、このIMMマクロ定義では、最終的に「(val)」に変換される。
【０２１２】
・_fS_(v)：((v >> 2) & 1)に変換される。ここでは、vの下位2ビット目（0ビット目から数えて）を取り出す演算と等価である。
・__fB_(v)，_fH_(v)，_fP_(v)，_fU_(v)，_fW_(v)：_fS_(v)と同様に、vの特定ビットを取り出す演算を定義する。後述のSTRマクロで呼び出される。
【０２１３】
＜STRマクロ記述（push命令）＞
図16で呼ばれるSTRマクロ定義を図20に示し、そのマクロ内部で呼び出される_ADDR_，_D_CACHE_SIMマクロ定義を図21に示す。（補足：元の機械語のアセンブリ命令記述では「push [lr]」（図15参照）となっているが、これは、スタックポインタsp（arm.r[13]）を「デクリメント」（1データワード分のアドレスを減算）した値をメモリアドレスとして、リンクレジスタlr（arm.r[14]）をメモリに書込む動作をし、ARMv5におけるstr命令として実装される。）
【０２１４】
（1）STRマクロで使われる引数：
・c（実行条件を表す文字列）：SUBマクロやCMPマクロと同様
・v（命令ビットフィールド）：アドレスオフセット計算方法、アドレスレジスタの更新の有無、データ型情報等の命令属性
・rd（書込みデータレジスタID）：書込みデータを格納するレジスタ番号
・rn（ベースアドレスレジスタID）：ベースアドレスを格納するレジスタ番号
・sh（シフト属性）：オフセットオペランドに対するシフト処理属性
・rm（オフセットレジスタID）：オフセット値を格納するレジスタ番号
・val（即値データ）：オフセット値が即値（定数）の場合の即値データ
・pc（命令アドレス）：現命令アドレス（ベースアドレスレジスタがPC（プログラムカウンタ）の場合、現命令アドレスがアドレス計算で使われる）
【０２１５】
（2）_ADDR_マクロ：STRマクロ内部で呼ばれ、メモリ書込みアドレスの計算を行う以下のプログラムを生成する。_ADDR_から呼び出されるマクロ_fU_(v)，_fP_(v)，_fW_(v)は、アドレスのオフセット計算方法やベースアドレスレジスタ_r_(rn)の更新の有無などを指定する命令属性である。
【０２１６】
（3）プログラムカウンタの作用：ARMv5命令セット仕様より、プログラムカウンタ（pc）がベースアドレスとして使われるときは、「現命令アドレス + 8」がベースアドレスとなる（_ADDR_マクロで定義）。また、書込みデータレジスタとして使われるときは、「現命令アドレス + 12」が書込みデータになる（STRマクロの変数dへの代入文で定義）。
【０２１７】

（4）書込みデータ型： _fB_(v), _fH_(v)（図20参照）で与えられる命令属性ビットに従い、８ビット整数型（U8）、１６ビット整数型（U16）、３２ビット整数型（U32）のそれぞれのメモリ書込み動作（図14の_m8_, _m16_, _m32_の各マクロで定義）を記述している。

【０２１８】

（5）キャッシュモデル： D_CACHE_SIMマクロは、キャッシュシミュレーションが有効の時（ENABLE_CACHE_MODELが別途定義されている場合）に、データキャッシュシミュレーション関数D_Cache_Sim（別途定義される）（図21参照）を呼び出す仕組みを提供する。なお、キャッシュシミュレーションが有効でない場合は、D_CACHE_SIM(addr, isRead)の記述（図21参照）は「空文字列」に変換される。

【０２１９】

＜LDRマクロ記述（pop命令）＞

図16で呼ばれるLDRマクロ定義を図22に示す。（補足：元の機械語のアセンブリ命令記述では「pop [pc]」となっているが、これは、スタックポインタsp（arm.r[13]）をメモリアドレスとして、前記push [lr]で記録された「戻り命令アドレス値」をメモリから読出し、スタックポインタを「インクリメント」（1データワード分のアドレスを加算）する動作をし、ARMv5におけるldr命令として実装される。）

【０２２０】

ここで、LDRマクロ（メモリ読込み）の引数や、内部で呼び出されるマクロは、STRマクロ（メモリ書込み）とほぼ共通している。一方、メモリ読出し固有の動作は以下の通りである。

【０２２１】

（1）読出しデータの上位ビット拡張：_fS_(v) == 1の場合「符号付き」データを表し、_fS_(v) == 0の場合「符号なし」データを表す。８ビット型データや１６ビット型データにおいて、「符号付き」データは上位ビットを「符号拡張」（符号ビットで埋める）し、「符号なし」データは上位ビットを「０挿入」する。

【０２２２】

（2）プログラムカウンタ（pc）へのロード動作：LDRマクロでは、プログラムカウンタへのロード命令は、「リターン命令」であると見なし、「return文」を実行するプログラム記述を生成する。前記した「データ依存分岐命令」もldr命令によるpcへのロードを実行するが、後述の「データ依存分岐命令プログラム記述生成部3a8」によって、LDRマクロを使用しないプログラム記述で「データ依存分岐命令」を実装する。

【０２２３】

＜マクロ呼出し記述のマクロ展開後のプログラム記述＞

前述の４つの命令プログラム生成マクロ（SUB, STR, CMP, LDR）の呼出しは、最終的に、図23に示すプログラム記述を生成する。

【０２２４】

＜プログラム記述上のサブルーチン命名規則＞

図12に示すサブルーチンプログラムソース4aのプログラム記述上における「サブルーチン命名規則」について説明する。

サブルーチン命名規則で注意すべき点は、例えば、C言語では、main関数やexit関数のように特別な意味を持つサブルーチン名が予約されているため、予約サブルーチン名と重複しないようにする必要がある。

【０２２５】

そこで、予め定めた「プリフィックス文字列」（例えば"_my_"など）を、「シンボル情報」に含まれる「シンボル名文字列」の先頭に連結して「サブルーチン名文字列」とする。例えば、シンボル名"jump_test"に対応する「サブルーチン名文字列」は、「プリフィックス文字列」"_my_"とした場合、"_my_jump_test"となる。

【０２２６】

また、実行バイナリファイル中の「シンボルテーブル」に登録されていないシンボルや

、「シンボルテーブル」自体が存在しない場合は、前記したようなシンボル名生成規則で生成される。この場合も同じ「プリフィックス文字列」で構わない。例えば、シンボルアドレスが0x1234である「未登録シンボル」名がfunc_1234と生成された場合、そのサブルーチン名文字列を"_my_func_1234"としてよい。

【０２２７】

＜サブルーチンプログラムソース生成部3a＞

ここでは、図16に示すような機械語命令プログラム生成マクロ呼出し記述を含み、サブルーチンプログラムソースを生成する「サブルーチンプログラムソース生成部3a」（図24）について説明する。

【０２２８】

＜サブルーチン情報取得部3a１＞

前記の機械語命令解析部１で生成したシンボル情報リストｒ１からシンボル情報を一つ取り出す。以後、ここで取り出したシンボル情報を「現シンボル情報」と呼ぶ。

【０２２９】

＜サブルーチン定義記述生成部3a2＞

「現シンボル情報」に含まれる「サブルーチン名文字列」を基に、図25に示すような「サブルーチン定義記述」を生成する。「第１実施形態」では、戻り値がなく（void型）、引数を取らないサブルーチンの定義の記述を生成する。

【０２３０】

＜命令アドレス取得部3a3＞

命令アドレス取得部3a3は、「現シンボル情報」の「命令アドレスリスト」の先頭から「命令アドレス」を順次読み出す。読み出した「命令アドレス」を、以後「現命令アドレス」と呼ぶ。

【０２３１】

＜プログラムラベル記述生成部3a4＞

「現命令アドレスが分岐先アドレスリストr2（機械語命令解析部1で生成）に存在する場合、プログラム分岐記述（goto文等）を可能とするために、命令（コマンド）を識別するための「プログラムラベル記述」を生成する。前記したARMv5命令セットのjump_test関数の機械語命令解析処理で得られた「分岐先アドレスリスト」は、[ 0x8384, 0x839c, 0x836c, 0x83a4]である（表３参照）。これらの分岐先アドレスに対応するC言語のラベル記述は、例えば、"L_08384:", "L_0839c:", "L_0836c:", "L_083a4:"と生成すればよい。

【０２３２】

＜機械語命令種別判定部3a5＞

機械語命令種別判定部3a5では、「現命令アドレス」を基に、「実行バイナリファイル」の「バイナリデータ部」に格納されているバイナリデータを読出す。このバイナリデータを機械語命令と見なし、この機械語命令をデコード（解釈）して（前記の次命令情報抽出部1dの機械語命令デコード処理と同じ手順）、以下の命令種別を判定する。

【０２３３】

・「単純分岐命令」（命令語のみで分岐先アドレスが特定できる分岐命令）の場合：単純分岐命令プログラム記述生成部3a6に進む。

・「単純サブルーチン呼出し命令」（命令語のみで呼出し先アドレスが特定できるサブルーチン呼出し命令）の場合：単純サブルーチン呼出し命令プログラム記述生成部3a７に進む。

【０２３４】

・「データ依存分岐命令」（分岐先アドレスがレジスタ値・メモリ値で決定される分岐命令）の場合：データ依存分岐命令プログラム記述生成部3a8に進む。

・「データ依存サブルーチン呼出し命令」（呼出し先アドレスがレジスタ値・メモリ値で決定されるサブルーチン呼出し命令）の場合：データ依存サブルーチン呼出し命令プログラム記述生成部3a9に進む。

・上記以外の場合：機械語命令プログラム生成マクロ呼出し記述生成部3a10に進む。

【０２３５】
＜機械語命令プログラム生成マクロ呼出し記述生成部3a10＞
ここでは、機械語命令の動作に関する以下の詳細情報を取得する。
・命令種別：機械語命令プログラム生成マクロ名を特定する。図16におけるSUB，STR，CMP，LDRなどの　マクロ名を生成するための情報となる。
【０２３６】
・作用レジスタ番号：オペランドレジスタ番号や演算結果格納レジスタ番号を特定する。図16の例では　、以下が作用レジスタ番号に対応する
SUB(AL,0x020，3，0，IMM,-1，...)（ 作用レジスタ番号：3，0
STR(AL,0x009,14,13，IMM,-1，...)（作用レジスタ番号：14，13
CMP(AL,0x024，0，3，IMM,-1，...)（作用レジスタ番号：0，3
LDR(AL,0x012,15,13，IMM,-1，...)（作用レジスタ番号：15，13
【０２３７】
なお、上記で「-1」の引数も各マクロの「rm」（第２オペランドレジスタID：第２オペランドを格納するレジスタ番号）引数に対応する（SUB/CMP：第２オペランドレジスタID、STR/LDR：オフセットレジスタID）が、第２オペランドまたはオフセットオペランドが定数のため、対応するレジスタが無いことに対応している。
【０２３８】
・　なお、上記で「-1」の命令属性情報：命令動作の詳細を規定する様々な属性情報を抽出する。図16　の例では、以下のデータがこれら命令属性情報に対応する。
【０２３９】
SUB(AL,0x020，3，0，IMM,-1,0x00000003，0x08340)
AL：実行条件（always：無条件で実行）
0x020：命令ビットフィールド引数vに対応（内部レジスタ更新なし）
IMM：シフト属性引数shに対応（第２オペランドが即値）
0x00000003：即値データ
【０２４０】
STR(AL,0x009,14,13，IMM,-1,0x00000004，0x08344)
AL：実行条件（always：無条件で実行）
0x009：命令ビットフィールド引数vに対応（アドレスオフセット計算方法、アドレスレジスタ更新の有無、データ幅情報等を含む）
IMM：シフト属性引数shに対応（アドレスオフセットが即値）
0x00000004：即値データ
【０２４１】
CMP(AL,0x024，0，3，IMM,-1,0x00000005，0x08348)
AL：実行条件（always：無条件で実行）
0x024：命令ビットフィールド引数v（図17）に対応
IMM：シフト属性引数shに対応（第２オペランドが即値）
0x00000005：即値データ
【０２４２】
LDR(AL,0x012,15,13，　IMM,-1,0x00000004，0x08380)
AL：実行条件（always：無条件で実行）
0x012：命令ビットフィールド引数vに対応
IMM：シフト属性引数shに対応（アドレスオフセットが即値）
0x00000004：即値データ
これらの機械語命令に関する詳細情報に基づき、「機械語命令プログラム生成マクロ呼出し記述」を生成する。
【０２４３】
＜単純分岐命令プログラム記述生成部3a6＞
命令語のみで分岐先アドレスが特定できるような「単純分岐命令」については、goto文

（無条件分岐命令）などを使ったプログラム分岐の記述を生成する。
以下では、図10の機械語命令記述における図26の無条件分岐命令（図10の6行目）について説明する。
【０２４４】
ここで、分岐先アドレスが0x839cであり、前記のプログラムラベル記述生成部3a4によってラベル文L_0839c:が0x839cの命令プログラム記述（図10の25行目）の前に挿入されるので、このラベルに分岐する図27のプログラム記述が生成される。
また、条件付き分岐の場合でも、AL（always：無条件で実行）の代わりに、対応する実行条件の文字列（EQ，NEなど）を挿入するだけで実現できる。
【０２４５】
＜単純サブルーチン呼出し命令プログラム記述生成部3a7＞
命令語のみで呼出し先アドレスが特定できるような「単純サブルーチン呼出し命令」については、図28に示すサブルーチン呼出しを行うプログラム記述を生成する。
以下では、jump_test関数を呼び出す図29の機械語命令記述について説明する。
【０２４６】
ここで、呼出し先の「サブルーチン名文字列」を、呼出し先アドレスを基に「シンボル情報リスト」から該当するシンボル情報を検索し、そのシンボル情報に含まれる「シンボル名文字列」から、前記した「サブルーチン命名規則」を適用して「サブルーチン名文字列」を生成した後、これを呼び出す図29のプログラム記述が生成される。
また、条件付き実行のサブルーチン呼出し命令の場合でも、AL（always：無条件で実行）の代わりに、対応する実行条件の文字列（EQ，NEなど）を挿入するだけで実現できる。
【０２４７】
＜データ依存分岐命令プログラム記述生成部3a8＞
分岐先アドレスがレジスタ値・メモリ値で決定されるような「データ依存分岐命令」については、機械語命令解析部1で生成した「ジャンプテーブル情報リスト」から、「データ依存分岐命令」のアドレスを基に、該当する「ジャンプテーブル情報」を検索し、その中に格納されている「分岐先アドレステーブル」を用いて、「データ依存分岐命令プログラム記述」を生成する。
【０２４８】
以下では、図10の機械語命令記述における図30のデータ依存分岐命令（図10の５、６行目）について説明する。
ここで、「データ依存分岐命令」のアドレス0x834c対応する「ジャンプテーブル情報」は以下の通りである。
【０２４９】
・　ジャンプテーブルサイズ：6
・　分岐先アドレステーブル：[ 0x8384，0x839c，0x836c，0x83a4，0x836c，0x8384 ]
このジャンプテーブル情報と、この「データ依存分岐命令」の実行条件（ls: less or same条件）を用いて、図31のようなプログラム記述が生成される。
ここで、各case文後に出てくるgoto文のラベルは、「ジャンプテーブル情報」の「分岐先アドレステーブル」の各分岐先アドレスのラベルに対応している。なお、case 0:とcase 5:に対応する条件分岐は、defaultラベルに対応している。
【０２５０】
＜データ依存サブルーチン呼出し命令プログラム記述生成部3a9＞
呼出し先アドレスがレジスタ値・メモリ値で決定されるような「データ依存サブルーチン呼出し命令」については、後記のシンボルアドレス情報テーブルソース生成部3bで生成する「シンボルアドレス情報」を使って、データ依存サブルーチン呼出しをプログラム記述上で実現する。
【０２５１】
以下では、図10の機械語命令記述における図32のデータ依存サブルーチン呼出し命令について説明する。

ここでは、前記したように、0x83acはレジスタr3の値にジャンプする「データ依存分岐命令」であるが、その直前の0x83a8に、pc（実際の値は現命令アドレス0x83a8 + 8）をリンクレジスタlrに代入する命令があるため、0x83acの命令は実際には「データ依存サブルーチン呼出し命令」であることが判別できる。0x83a8と0x83acの２命令に対応するプログラム記述は図33のようになる。

【０２５２】

また、条件付き実行のデータ依存サブルーチン呼出し命令の場合でも、AL（always：無条件で実行）の代わりに、対応する実行条件の文字列（EQ, NEなど）を挿入するだけで実現できる。

なお、ここで生成された図33のプログラム記述の内容については、後記するシンボルアドレス情報テーブルソース生成部3bで説明する。

【０２５３】

＜サブルーチンプログラムソース出力例＞

図10の"jump_test"関数のARMv5命令セットの機械語命令記述に対応するサブルーチンプログラム記述の出力例を図34に示す。

【０２５４】

＜シンボルアドレス情報テーブルソース生成部3b＞

ここでは、前記したデータ依存サブルーチン呼出し命令プログラム記述生成部3a9によって生成されるプログラム記述を実行するための情報を生成し、プログラムソースに出力する処理について説明する。この処理は、「シンボル情報」の「データ依存サブルーチン呼出し命令フラグ」が「１」であるものが「シンボル情報リスト」に含まれる場合にのみ必要となる。

【０２５５】

まず、図33にある_FP_INFO_は、図35に示すように、予め定義されたシンボルアドレス情報を格納するデータ構造体である。

また、図33にある_GET_FPI_(_r_(3))は、_r_(3)の値が示す「シンボルアドレス」を基に、後記の「シンボルアドレス情報テーブル」から該当する「シンボルアドレス情報」のデータ構造体へのポインタを返す関数の呼出しである（後述）。つまり、fpiが指す「シンボルアドレス情報」のデータ構造体へのポインタを使い、このデータ構造体に格納されているサブルーチンのアドレス（func）を使うことで、fpi->func()というデータ依存サブルーチン呼出しをプログラム上で実現している。

【０２５６】

次に、「シンボルアドレス情報」をテーブル形式で格納した「シンボルアドレス情報テーブル」のプログラムソース出力方法を説明する。例えば、表11のような「シンボル情報リスト」が生成されているとする。

【表１１】

「シンボル情報リスト」の例
（「命令アドレスリスト」は省略）

| シンボルアドレス | シンボル名 |
|---|---|
| **0x8218** | **“f0”** |
| **0x8220** | **“f1”** |
| **0x8238** | **“ff1”** |
| **0x8338** | **“ff2”** |
| **0x8340** | **“jump_test”** |
| **0x8448** | **“main”** |

この「シンボル情報リスト」の情報を、前述の_FP_INFO_構造体の配列変数の初期値設定として、プログラム上で参照可能にするためのプログラム記述を図36に示す。

【０２５７】

図36において、例えば[0x8218, my_f0]の記述では、0x8218が図35に示す_FP_INFO_構造体のメンバー変数addrの初期値であり、_my_f0がメンバー変数funcの初期値である。_my_f0はシンボル名"f0"に「プリフィックス文字列」"_my_"を持つサブルーチン名文字列であり、実際は、対象サブルーチンのアドレス値を指している。このようにして、実行バイナリファイルにおける「シンボルアドレス」と、プログラム記述上の「サブルーチンアドレス」を対応付けるためのデータ構造体の初期値を設定することができる。

【０２５８】

次に、図33にある_GET_FPI_関数（「シンボルアドレス」を基に、これに該当する_FP_INFO_構造体を探索してそのポインタを返す関数）は、図37に示すような簡単なプログラムで実現できる。

また、シンボルアドレス情報テーブルに含まれるシンボル数が非常に多いと、_GET_FPI_関数の処理時間が長くなる懸念はあるが、その場合は、「２分木探索法」や「ハッシュ探索法」などを使って、シンボルアドレスを基とした探索処理の高速化手法を適用すればよい。

【０２５９】

＜メモリ初期化プログラム記述生成部3c＞

メモリ初期化プログラム記述生成部3c（命令セットシミュレータプログラムソースコード出力部3）では、命令セットシミュレータSのプログラム記述におけるCPUのメモリ領域の初期化を行うためのプログラム記述を生成する。

メモリ領域の初期化の対象は、初期値が定義されているデータ領域の他にも、プログラム領域にも初期値データが含まれる場合がある。そこで、プログラム領域も初期化の対象とする。

【０２６０】

メモリ初期化データは、実行バイナリファイル5の前記のバイナリデータ部に存在し、各メモリ領域と「バイナリデータ部」の位置関係の情報は、「プログラムヘッダーテーブル」に含まれる前記の「セグメント」の単位で、または、前記の「セクションヘッダーテーブル」に含まれる「セクション」の単位で、それぞれ定義されている。

【０２６１】
メモリ初期化プログラム記述生成部3bのメモリ初期化処理のプログラム記述は、これらいずれかの単位での処理を定義する。図38に「メモリ初期化プログラム記述」の一例を示す。ここでは、「セクション単位」のメモリ初期化処理を記述している。
【０２６２】
初期化データを持つ「.rodata」セクションが、0x1260c － 0x127e7のアドレス領域で定義されており、その具体的初期化データはunsigned char data[476]の初期化データとして定義されている。この配列データをmemcpy関数によって&arm._mem[0x1260c]のアドレスにコピーするプログラム記述となっている。
【０２６３】
＜メイン関数プログラム記述生成部3d＞
ここでは、最終的に生成する命令セットシミュレータＳのプログラム記述の最上位関数（main関数）のプログラム記述について説明する。最上位関数（main関数）のプログラムはメイン関数プログラム記述生成部3dが作成する。
前記したように、実行バイナリファイルで定義されている「エントリーポイント」（プログラム実行時の最初のアドレス）を「シンボルアドレス」として持つ「エントリーシンボル」が最初に実行するサブルーチンに対応している。そこで、「命令セットシミュレータ」のプログラム記述の最上位関数から、「エントリーシンボル」に対応するサブルーチンを呼び出すプログラム記述を生成する。
【０２６４】
例えば、エントリーシンボルに対応するサブルーチン名が"my__mainCRTStartup"の場合、命令セットシミュレータSの最上位関数のプログラム記述は、図39のようになる。図39に、命令セットシミュレータの最上位関数のプログラム記述例1を示す。
【０２６５】
また、別の方法として、実行バイナリファイルの中のmain関数を直接呼び出すことも可能である。この場合、スタックポインタをメモリのサイズに設定し、必要ならば、main関数の引数int argc, char* argv[]をCPUのメモリ上に書込んだ上でmain関数を呼び出す（図40）。　図40に、命令セットシミュレータの最上位関数のプログラム記述例2を示す。図41に、main関数の引数情報をCPUメモリに書き込むプログラム記述を示す。
【０２６６】
＜命令セットシミュレータＳによるプログラム記述構造の特徴＞
以上により生成される命令セットシミュレータプログラム記述構造は以下の特徴を持つ。
【０２６７】
（1）図34に示すように、「実行バイナリファイル」中のサブルーチンがそのまま「プログラム記述」上のサブルーチンに対応しており、「実行バイナリファイル」が持つ「関数階層構造」が命令セットシミュレータＳが出力するプログラム記述にそのまま反映されている。
また、単純分岐命令においてgoto文とラベル文が明確に対応し、データ依存分岐命令においてswitch文とcase文が明確に対応しており、プログラム制御の流れも理解しやすい。
従って、プログラム記述の可読性が高く、プログラムのデバッグ作業やプログラムの機能拡張などのコード改変作業を効率的に行うことができる。
【０２６８】
（2）プログラム制御は、goto文、call文、return文により明示的に実現されており、プログラムカウンタ（PC）の更新処理が存在しない。従来の静的コンパイル方式命令セットシミュレータでは必須であった「PCに関するswitch文と各命令アドレスに関するcase文のコード構造」を使わない。
【０２６９】
そのため、プログラム制御の処理オーバーヘッドが従来よりも大幅に少ない。また、プログラム制御構造が、従来技術のものよりも極めて単純になるため、コンパイラの最適化

処理（冗長命令削除等）が非常に効きやすくなる。
これらのことから、命令セットシミュレータＳの処理速度が大幅に向上する。
【０２７０】
以上のことから、第1実施形態の命令セットシミュレータＳの構成によれば、下記の効果を奏する。
1．命令セットシミュレータＳを用いた組込みシステム用ソフトウエア開発環境によれば、詳細動作の再現が可能である。
【０２７１】
2．実行バイナリと等価機能をもつソースプログラム記述ファイル（命令セットシミュレータソースコード：C言語記述）を出力し、これをコンパイル・実行することで、上述したように、高速な命令セットシミュレーション環境を構築できる。ネイティブ実行時間（例えば、ターゲットCPU用のプログラムソースをコンパイルして機械語にしてCPUで実行する時間）にほぼ匹敵するシミュレーション実行時間をもつ高速な命令セットシミュレータが得られる。
【０２７２】
3．アプリケーションソフトウエア開発検証環境での、ターゲットCPU用で動作させるプログラムのデバッグ・動作検証・性能検証が行える。
【０２７３】
4．プロセッサアーキテクチャ開発検証環境での、CPU高性能化のためのアーキテクチャ変更等を反映した命令セットシミュレータを生成できる。
【０２７４】
5．プラットフォーム変換（ポーティング）が可能である。あるターゲットCPU用の実行バイナリから等価機能C記述を出力し、別CPU用にプログラム移植できる。
【０２７５】
6．マルウェア解析が可能である。プログラムソースの解析が容易であるので、ウィルス等のマルウェア解析が容易に行え、ウィルス駆除ソフト・ファイヤウォールの早期開発に資することができる。
【０２７６】
7．ソフトウエア知財保護に役立つ。実行バイナリと等価機能をもつソースプログラム記述ファイルの解析が容易であるので、侵害の立証がスムーズに行え、著作権侵害・特許侵害の検証で有効に活用できる。
【０２７７】
8．実行バイナリからソースプログラム復元を「確実に」行える完全性をもつ。
【０２７８】
9．高速性と完全性とを実現できる簡便なツールフレームワーク構造の命令セットシミュレータSを得られる。
【０２７９】
前記したように、実際の使用に供される非特許文献4、8、9は、下記の解決すべき課題があった。
第１の課題として、どの命令が分岐先アドレスとなり得るかを判断する情報がない。
第２の課題として、シミュレーション実行時間がネイティブ実行に比べ14倍～54倍かかる。
第３の課題として、compilerにおけるコード最適化が非常に掛かりにくい。
第４の課題として、階層構造が欠如している。
第５の課題として、ソースプログラムとしては著しく可読性が低いという
第６の課題として、プログラム制御フロー解析の仕組みが欠如している。
前述したように、第1実施形態の命令セットシミュレータSの構成によれば、分岐先アドレスが分岐先命令にラベル（識別情報）として付加されたり、ジャンプテーブル等があるので、どの命令が分岐先アドレスとなり得るかを判断する情報がある。
また、シミュレーション実行時間がネイティブ実行時間とほぼ同等である。

また、バイナリファイルの階層構造がプログラムソースコードに復元されるので、compilerにおけるコード最適化がかかり易い。
また、バイナリファイルから復元されるソースプログラムは、バイナリファイルの各サブルーチンが関数の構造を持ち、階層構造を有するので可読性が高い。
さらに、単純分岐命令、データ依存分岐命令、データ依存サブルーチン等を解析してジャンプテーブル等で分岐先を明示するので、プログラム制御フロー解析の仕組みを有している。
以上のことから、第１実施形態（本願発明）により、実際の使用に供される非特許文献4、8、9の問題を解決することができる。

【０２８０】

＜＜第2実施形態＞＞

第2実施形態の命令セットシミュレータ2S（図43参照）は、第1実施形態の命令セットシミュレータSに「サブルーチン引数抽出機能」を追加したものである。
その他の構成は、第１実施形態の命令セットシミュレータSと同様であるから同様な構成要素には同一の符号を付して示し、詳細な説明は省略する。

【０２８１】

第2実施形態の命令セットシミュレータ2Sは、第1実施形態の命令セットシミュレータSに、「サブルーチン引数抽出機能」を追加し、CPUのレジスタをサブルーチン内のローカル変数及び引数変数に置き換えることにより、命令セットシミュレーションのさらなる高速化を図ったものである。

【０２８２】

図42に、第2実施形態の機械語命令解析部の機能ブロック図を示す。
第2実施形態の機械語命令解析部21は、第1実施形態のシンボルアドレス前処理部1a、シンボルアドレス取得部1b、命令アドレス取得部1c、次命令情報抽出部1d、アドレスリスト更新部1e、およびデータ依存分岐抽出部2に、新たにサブルーチン引数抽出部1fを加えて構成したものである。

【０２８３】

サブルーチン引数抽出部1fは、出力するソースコードにCPUのレジスタ内のデータをローカル変数及び引数変数として記述する働きをする。
図43に、第2実施形態の命令セットシミュレータ2Sの構成を、解析対象の実行バイナリファイル5、命令セットシミュレータ2Sが出力する命令セットシミュレータプログラムソースコード4とともに示す。

【０２８４】

第2実施形態の命令セットシミュレータ2Sは、機械語命令解析部21と命令セットシミュレータプログラムソースコード出力部23とを備えている。
命令セットシミュレータ2Sは、実行バイナリファイル5を入力として、高級言語の命令セットシミュレータプログラムソースコード4を出力する。

【０２８５】

命令セットシミュレータプログラムソースコード出力部23は、サブルーチンプログラムソース生成部3a、シンボルアドレス情報テーブルソース生成部3b、メモリ初期化プログラム記述生成部3c、およびメイン関数プログラム記述生成部3dに、サブルーチン引数共通化処理部3eが新たに加わった構成である。

【０２８６】

なお、機械語プログラムの実行バイナリファイル5に、データ依存サブルーチン呼出し命令がある場合のみ、シンボルアドレス情報テーブルソース生成部3bとサブルーチン引数共通化処理部3eとが用いられる。

【０２８７】

以下、新たに加わったサブルーチン引数抽出部1f（図42参照）について説明する。
＜サブルーチン引数抽出部1f＞
サブルーチン引数抽出部1fによる処理は、命令アドレス取得部1cが「現シンボル情報」

を「シンボル情報リスト」に「追加」する前に、実行する（図42、図7参照）。
【０２８８】
＜サブルーチン引数抽出部1fの処理の原理＞
サブルーチンの「引数」は、サブルーチン呼出し前にCPUレジスタに設定されたデータであり、プログラムの命令実行がサブルーチンに入った後に参照される（引数がスタックメモリに格納される場合はここでは考慮しない）。どのレジスタが「サブルーチン引数」であるかを解析するためには、命令実行がサブルーチンに入った後にいかなる命令によっても更新されずに参照されるレジスタを見つければよい。この解析のためには、各命令がどのレジスタを参照し、どのレジスタを更新するかを予め解析（把握）しておき、命令実行順序を「逆順」に辿ることで、「サブルーチン引数」を特定することができる。
【０２８９】
図10のARMv5機械語命令の「入出力レジスタ」の情報を図44に示す。ここで、「入出力レジスタ」とは、機械語命令がオペランドとして使用するレジスタ（入力レジスタ）と、機械語命令によって値が更新されるレジスタ（出力レジスタ）を指す。入力・出力レジスタの対象は、pc（プログラムカウンタ）を除き（pcは引数抽出処理には関係がないので）、「状態レジスタ」を含むとする。また、複数の「状態レジスタ」を便宜上１つのレジスタ「CC」と見なし、レジスタIDを「16」と表記することにする。　また、spのレジスタIDを「13」，lrのレジスタIDを、前記したように「14」と表記している。
【０２９０】
（1）入力レジスタIDリスト（図44でI(..)と表記）：機械語命令がオペランドとして使用するレジスタIDのリスト
（2）出力レジスタIDリスト（図44でO(..)と表記）：機械語命令がオペランドとして使用するレジスタIDのリスト
【０２９１】
（3）入出力レジスタの例：
A）sub　　r3，r0，#3：第１オペランド がr0なのでI(0)となり、結果格納レジスタがr3なのでO(3)となる。
B）push　　[lr]：ベースアドレスがスタックポインタ(sp: r[13])であり、書込みデータがリンクレジスタ(lr: r[14])なのでI(13,14)となり、ベースアドレスレジスタが更新されるのでO(13)となる。
【０２９２】
C）cmp　　r3，#5：第１オペランドがr3であるのでI(3)となり、状態レジスタ(CC)が更新されるのでO(16)となる。
D）ldrls　　pc，[pc，r3，lsl #2]：ベースアドレスがpc（入力とみなさない）でオフセットがr3であり、条件付き実行命令であるので状態レジスタ(CC)を参照するのでI(3,16)となる。出力レジスタpcは除かれるのでO()となる（出力レジスタはない）。
【０２９３】
（4）「生存入力レジスタ解析」：サブルーチン内のすべての「リターン命令」から命令実行の逆順に辿り、「生存入力レジスタIDリスト」を更新する。「生存入力レジスタ」とは、ここでは、ある命令に着目し、その命令実行後にサブルーチン内で参照されるレジスタであって、その命令実行前にサブルーチン内で更新されないレジスタのことを指す。「生存入力レジスタ解析」処理は、以下の手順を各命令で実行する。
【０２９４】
A）「出力レジスタIDリスト」に含まれる各レジスタIDについて、「生存入力レジスタIDリスト」にそのレジスタIDが存在する場合、そのレジスタIDを「生存入力レジスタIDリスト」から削除する。
B）「入力レジスタIDリスト」のすべてのレジスタIDを「生存入力レジスタIDリスト」に（非重複的に）「追加」する。
【０２９５】

【表１２】

入出力レジスタの解析例（0x8380のリターン命令から解析開始）

| 命令 | 命令入出力レジスタID | 生存入力レジスタID |
|---|---|---|
| **8380: pop {pc}** | **I(13), O(13)** | **I(13)** |
| **837c: add r0, r0, #1** | **I(0), O(0)** | **I(0, 13)** |
| **8378: bx r3** | **I(3), O()** | **I(0, 3, 13)** |
| **8374: mov lr, pc** | **I(), O(14)** | **I(0, 3, 13)** |
| **8370: ldr r3, [r3]** | **I(3), O(3)** | **I(0, 3, 13)** |
| **836c: ldr r3, [pc, #68]** | **I(), O(3)** | **I(0, 13)** |
| **834c: ldrls pc, [pc, r3, lsl #2]** | **I(3,16), O()** | **I(0, 3, 13, 16)** |
| **8348: cmp r3, #5** | **I(3), O(16)** | **I(0, 3, 13)** |
| **8344: push {lr}** | **I(13,14), O(13)** | **I(0, 3, 13, 14)** |
| **8340: sub r3, r0, #3** | **I(0), O(3)** | **I(0, 13, 14)** |

【０２９６】

表12は、0x8380のリターン命令からシンボルの開始命令（0x8340）まで逆順に辿って行った時の「生存入力レジスタIDリスト」を示しており、これらの解析の結果、シンボルの開始命令における「生存入力レジスタIDリスト」は(0，13，14)となり、「生存入力レジスタ」がr0，sp，lrであることがわかる。同様の解析処理を他のすべてのリターン命令から行うことで、最終的に「生存入力レジスタIDリスト」として（0，13，14)を得る。

【０２９７】

（5）サブルーチン引数レジスタの決定：前記の解析処理で得られる「生存入力レジスタIDリスト」から、以下の規則に従って「サブルーチン引数」を決定する。

A）リンクレジスタ（lr：r[14]）：「サブルーチン引数」と見なさない。なぜなら、CPU動作上で「サブルーチン呼出し処理」と「サブルーチン復帰処理」を実現するためのレジスタの情報は、プログラム記述上でこれらサブルーチンに関する制御を直接行う「当発明技術」には不要であるからである。

【０２９８】

B）スタックポインタ（sp：r[13]）：「サブルーチン引数」と見なす。通常スタックポインタの更新はサブルーチン呼出し処理とサブルーチン復帰処理の前後で実行され、CPUリソースで管理されているが、本第２実施形態では、スタックポインタを含めたすべてのCPUレジスタをローカル変数またはサブルーチン引数として扱うためである。

C）その他のレジスタ：「サブルーチン引数」と見なす。

【０２９９】

＜サブルーチン引数抽出部1fの処理で使用するデータ構造＞

サブルーチンの引数抽出部1fの処理では、以下のデータ構造を使用する。

（1）命令情報リスト：「生存入力レジスタ」を解析するための命令情報リスト。

「命令情報」は以下のデータ項目からなる。

A）命令情報ID：各命令情報に固有のID番号

B）命令アドレス

C）入力レジスタIDリスト

【０３００】
D）出力レジスタIDリスト
E）生存入力レジスタIDリスト（命令毎の「生存入力レジスタ」）
F）次命令情報リスト：前記の「次命令情報一時リスト」と同じ情報
G）「前方向命令情報IDリスト」：その命令を「次命令」するすべての命令の命令情報IDからなるリスト（命令実行の逆順に辿る時に必要となる）
【０３０１】
（2）「命令情報ID一時リスト」：命令実行の逆順解析処理のための命令情報IDの一時リスト。
【０３０２】
（3）「サブルーチン生存入力レジスタID一時リスト」：「生存入力レジスタ」のIDからなる一時リスト。
【０３０３】
（4）「サブルーチン引数レジスタIDリスト」：最終的に、サブルーチン引数に対応するレジスタIDを格納するリスト
また、前記した「シンボル情報」に「サブルーチン引数レジスタIDリスト」を追加する。
「第２実施形態のシンボル情報」のデータ項目：
A）シンボルアドレス
B）命令アドレスリスト
C）シンボル名文字列
D）データ依存サブルーチン命令フラグ
E）サブルーチン引数レジスタIDリスト（第2実施形態で新たに加わったもの）
【０３０４】
＜サブルーチン引数抽出部1fの構成＞
図45に、サブルーチン引数抽出部1fの機能ブロック図を示す。
サブルーチン引数抽出部1fは、入出力レジスタ解析部1f1、前方向命令解析部1f2、サブルーチン引数抽出前処理部1f3、命令情報取得部1f4、生存入力レジスタIDリスト更新部1f5、およびサブルーチン引数レジスタ決定部1f6を有している。以下、サブルーチン引数抽出部1fの各部について説明する。
【０３０５】
＜入出力レジスタ解析部1f1＞
「現シンボル情報」の「命令アドレスリスト」から「命令アドレス」を一つずつ順次取得し、その命令アドレスに対応する「命令情報」を生成する。そして、「命令情報」に「命令情報ID」を適宜（重複しないように）設定し、その命令の「命令アドレス」を設定し、「入力レジスタIDリスト」、「出力レジスタIDリスト」、「次命令情報リスト」を作成する。なお、「次命令情報リスト」の作成には前記の次命令情報抽出部1dを使う。この時点では、各「命令情報」の「生存入力レジスタIDリスト」と「前方向命令情報IDリスト」は「空」にする。すべての「命令アドレス」に対する処理が終了後、前方向命令解析部1f2の処理へ進む。
【０３０６】
＜前方向命令解析部1f2＞
「命令情報リスト」の各「命令情報」において（以降この「命令情報」を「現命令情報」と呼ぶことにする）、「次命令情報リスト」に含まれる「直後型次命令」と「分岐型次命令」の各「次命令アドレス」を基に、該当する「命令情報」を探索する。命令情報の命令アドレスが次命令アドレスと一致するものを探索する。以降探索した「命令情報」を「次命令情報」と呼ぶ。この「次命令情報」の「前方向命令情報IDリスト」に「現命令情報」の「命令情報ID」を（非重複的に）「追加」する。すべての「命令情報」に対する処理が終了後、サブルーチン引数抽出前処理部1f3へ進む。
【０３０７】

＜サブルーチン引数抽出前処理部1f3＞
サブルーチン引数抽出処理前処理部1f3の前処理として、以下を実行する。
「命令情報リスト」に含まれる命令情報のうち、「リターン命令」に該当する「命令情報」の「命令情報ID」をすべて「命令情報ID一時リスト」に（非重複的に）「追加」する。
「サブルーチン生存入力レジスタID一時リスト」を「空」にする。
その後、命令情報取得部1f4へ進む。
【０３０８】
＜命令情報取得部1f4＞
「命令情報ID一時リスト」から「命令情報ID」を「取り出し」、該当する「命令情報」を取得し、生存入力レジスタIDリスト更新部1f5へ進む。以降、これを「現命令情報」と呼ぶ。「命令情報ID一時リスト」が「空」の場合、サブルーチン引数レジスタ決定部1f6へ進む。
【０３０９】
＜生存入力レジスタIDリスト更新部1f5＞
以下の処理を実行し、命令情報取得部1f4の処理へ戻る。
（1）「現命令情報」の「出力レジスタIDリスト」の各「出力レジスタID」について、「サブルーチン生存入力レジスタID一時リスト」に含まれる場合は、そのレジスタIDを「サブルーチン生存入力レジスタID一時リスト」から削除する。
（2）「現命令情報」の「入力レジスタIDリスト」の各「入力レジスタID」を「サブルーチン生存入力レジスタID一時リスト」に（非重複的に）「追加」する。
【０３１０】
（3）「サブルーチン生存入力レジスタID一時リスト」のすべてのレジスタIDが「現命令情報」の「生存入力レジスタIDリスト」に含まれる場合、以降の処理をスキップし、直ちに命令情報取得部1f4の処理に戻る。
【０３１１】
（4）「サブルーチン生存入力レジスタID一時リスト」のあるレジスタIDが「現命令情報」の「生存入力レジスタIDリスト」に含まれない場合、以降の処理を実行する。
A）「サブルーチン生存入力レジスタID一時リスト」の各レジスタIDを「現命令情報」の「生存入力レジスタIDリスト」に非重複的に「追加」する。つまり、重複する場合、レジスタIDは「生存入力レジスタIDリスト」に追加しない。
B）「現命令情報」の「前方向命令情報IDリスト」の各「命令情報ID」を「命令情報ID一時リスト」に非重複的に「追加」する。つまり、重複する場合、命令情報IDは「命令情報ID一時リスト」に追加しない。
【０３１２】
＜サブルーチン引数レジスタ決定部1f6＞
「サブルーチン生存入力レジスタID一時リスト」で、「サブルーチン復帰アドレス」を格納する「リンクレジスタ」のレジスタID以外を「サブルーチン引数レジスタIDリスト」に書き込む。
【０３１３】
＜サブルーチン引数共通化処理部3e＞
次に、命令セットシミュレータプログラムソースコード出力部23におけるサブルーチン引数共通化処理部3eについて説明する。
サブルーチン引数共通化処理部3eは、サブルーチン引数抽出部1fの処理を、「シンボル情報リスト」に含まれるすべての「シンボル情報」について実行したのちに実行する（図42参照）。この処理は、「シンボル情報」の「データ依存サブルーチン呼出し命令フラグ」が「１」であるものが「シンボル情報リスト」に含まれる場合にのみ必要となる。この処理が必要な理由は、データ依存サブルーチンをプログラム記述で実現する場合に、全サブルーチンに「共通」の関数ポインタデータ型（引数の個数も共通化）が必要となるからである。
【０３１４】

「空」の「共通サブルーチン引数レジスタIDリスト」を作成する。
「シンボル情報リスト」に含まれるすべての「シンボル情報」について、その「サブルーチン引数レジスタIDリスト」に含まれるすべてのレジスタIDを「共通サブルーチン引数レジスタIDリスト」に非重複的に「追加」する。つまり、共通サブルーチン引数レジスタIDリストには、レジスタIDは重複して存在しない。
【０３１５】
「シンボル情報リスト」に含まれるすべての「シンボル情報」について、「共通サブルーチン引数レジスタIDリスト」をその「シンボル情報」の「サブルーチン引数レジスタIDリスト」に置き換える。
【０３１６】
＜第２実施形態のその他の処理の変更点＞
（1）CPUリソース参照マクロ定義：図14のCPUリソース参照マクロ定義は、CPUレジスタ変数がCPUリソースデータ構造体（図13参照）で定義されていることを前提としているが、「第２実施形態」では、CPUレジスタ変数はサブルーチン引数変数またはローカル変数として定義する。そこで、CPUリソース参照マクロを図46のように変更する。
【０３１７】
（2）「サブルーチン定義記述生成部」：第２実施形態では、サブルーチンの関数型が第１実施形態の場合と異なる。
A）サブルーチン「戻り値」：第１実施形態では戻り値のない(void型)関数であったが、第２実施形態ではCPUレジスタのデータ型の戻り値をとる関数にする。（32ビットCPUの場合U32型、64ビットCPUの場合U64型となる）
B）サブルーチン引数：「サブルーチン引数抽出部」及び（必要に応じて実行される）「サブルーチン引数共通化処理部3e（図43参照）で生成した「サブルーチン引数レジスタIDリスト」によりサブルーチン引数の個数を決定する。それぞれの引数のデータ型は「戻り値」と同様にCPUレジスタのデータ型と同じにする。また、引数名は、レジスタ番号やレジスタ別名（r[13]の別名sp等）を基に生成する。
【０３１８】
C）CPUレジスタのローカル変数定義記述：「サブルーチン引数」に出現しないその他のCPUレジスタをローカル変数として、サブルーチンの最初の部分に定義する（図47参照）。図47に、"jump_test"関数のサブルーチン定義記述出力を示す。
図48に、第２実施形態のサブルーチン定義記述例を示す。ここでは、「サブルーチン引数共通化処理部3e」実行後もサブルーチン引数の個数が変化しないことを想定している。
【０３１９】
（3）単純サブルーチン呼出し命令プログラム記述生成部3a7（図24参照）：第２実施形態では、サブルーチンの関数型が異なるため、サブルーチン呼出し記述が変わる。
A）サブルーチン引数引き渡し記述：「呼出し先シンボル情報」の「サブルーチン引数レジスタIDリスト」を取得し、引数変数またばローカル変数のレジスタを使って呼出しサブルーチンの引数引き渡し記述を生成する。
【０３２０】
B）サブルーチン戻り値：サブルーチン戻り値をレジスタ変数に格納する記述を生成する。（実際のサブルーチンの戻り値がない場合でもプログラム実行上の不都合は発生しない）
図50に第２実施形態のサブルーチン呼出しプログラム記述を示す。
【０３２１】
（4）「シンボルアドレス情報テーブル」における「関数ポインタデータ型」の変更：「第１実施形態」では、戻り値のなく、引数もない関数ポインタデータ型をシンボルアドレス情報の_FP_INFO_構造体（図34～図36参照）で使用していたが、第２実施形態で図49示す定義に変更する。なお、「共通サブルーチン引数レジスタIDリスト」が、サブルーチン「my_jump_test」の「サブルーチン引数レジスタIDリスト」と同一であると想定している。

【０３２２】
（5）データ依存サブルーチン呼出し命令プログラム記述生成部3a9（図24参照）：第２実施形態では、サブルーチン呼出し記述が変わるため、「データ依存サブルーチン呼出しプログラム記述」も変更される。この場合、「共通サブルーチン引数レジスタIDリスト」により、引数変数またはローカル変数のレジスタを使って呼出しサブルーチンの引数引き渡し記述を生成する。図50に、第2実施形態のデータ依存サブルーチン呼出し命令プログラム記述を示す。前記同様、「共通サブルーチン引数レジスタIDリスト」が、サブルーチン「my_jump_test」の「サブルーチン引数レジスタIDリスト」と同一であると想定している。

【０３２３】
＜＜第3実施形態＞＞
図51に、第3実施形態の命令セットシミュレータを用いた組込みシステム用ソフトウエア開発環境を示す。
第3実施形態の命令セットシミュレータ3Sは、第1実施形態と第2実施形態とにおいて、シンボルテーブルがない場合の機械語命令解析部31を含むものである。
第1実施形態および第2実施形態では、実行バイナリファイルに「シンボルテーブル」が含まれていることを前提にしている。ここでは、「シンボルテーブル」が存在しない場合に対応するための第3実施形態について説明する。

【０３２４】
＜シンボルテーブル」が存在しない場合の対処法の概要＞
第3実施形態では、第1・第2実施形態の機械語命令解析部1（21）と命令セットシミュレータプログラムソースコード出力部3とが、機械語命令解析部31、命令セットシミュレータプログラムソースコード出力部33に変更される。

【０３２５】
（1）機械語命令解析部31の変更点は下記である。
（ア）機械語命令解析部31は、未解析シンボルアドレス検出部３１aをもつ。
「シンボルテーブル」が存在しない場合、機械語命令の位置を特定する唯一の情報は「エントリーポイント」（プログラム起動後に最初に実行される命令アドレス）である。従って、まず、未解析シンボルアドレス検出部３１aが「エントリーポイント」のみを「シンボルアドレス一時リスト」に追加する。

【０３２６】
（イ）機械語命令解析部31は、シンボル候補アドレスリスト３１a1をもつ。「シンボル候補アドレスリスト」とは、「シンボル候補」のアドレスリストである。
シンボル候補アドレスリスト３１a1の初期状態は「空」であり、「未解析」のシンボルアドレスが見つかった場合に未解析シンボルアドレス検出部３１aは、シンボル候補アドレスに追加する（後述）。この「シンボル候補アドレスリスト」の各アドレスを「シンボルアドレス一時リスト」に追加する。

【０３２７】
（2）命令セットシミュレータプログラムソースコード出力部33の変更点：「データ依存サブルーチン呼出し命令」のプログラム記述で呼び出される_GET_FPI_関数の定義を図52のように変更する。図52は、_FP_INFO_構造体のポインタを返す_GET_FPI_関数の定義を示す。

【０３２８】
変更ポイントとしては、
「データ依存サブルーチン呼出し命令」による呼出し先アドレスに該当する「シンボル情報」がない（そのアドレスのシンボルが「未解析」であるということ）場合に、命令セットシミュレータを「強制終了」させる仕組みがある。
この特徴により、「未解析」シンボルが呼び出された時に、命令セットシミュレータを「異常終了」させる（exit(-1)。そして、更新された「シンボル候補アドレスリスト」を使って機械語命令解析部31、命令セットシミュレータプログラムソースコード出力部33を

再度実行し、生成された命令セットシミュレータプログラムソースコード34を再度コンパイル・実行する。

【０３２９】

上記構成によれば、実行バイナリファイルにシンボルテーブルがない場合も、命令セットシミュレータ3Sが対応できる。

また、第1実施形態と同様な作用効果を奏し、詳細動作の再現が可能で、シミュレータの処理速度が速い。

【０３３０】

＜＜その他の実施形態＞＞

1．なお、前記した実施形態1～3では、実行バイナリから逆コンパイルさせる高級プログラム言語としてC言語を例示したが、逆コンパイルさせる言語をC言語以外の高級プログラム言語に代替できるのは勿論である。

【０３３１】

2．なお、特許請求の範囲のプログラムソースコードとは、実施形態で説明した命令セットシミュレータS、2S、3Sによって機械語から変換されるC言語等を指し、プログラムソースコードはC言語以外のプログラミング言語でもよいのは勿論である。

【符号の説明】

【０３３２】

1c　命令アドレス取得部（サブルーチン検出手段、サブルーチン呼出し命令検出手段）

1d　次命令情報抽出部（サブルーチン検出手段、サブルーチン呼出し命令検出手段、分岐命令検出手段、単純分岐命令検出手段、単純サブルーチン呼出し命令検出手段、データ依存分岐命令検出手段、データ依存サブルーチン呼出し命令検出手段）

1e　アドレスリスト更新部（サブルーチン検出手段、単純サブルーチン呼出し命令検出手段、データ依存サブルーチン呼出し命令検出手段）

1f　サブルーチン引数抽出部（サブルーチン検出手段、レジスタ変数展開手段）

2　データ依存分岐情報抽出部（サブルーチン検出手段、分岐命令検出手段、データ依存分岐命令検出手段、データ依存サブルーチン呼出し命令検出手段、分岐先アドレス情報読出し部）

2f　（サブルーチン機械語アドレステーブル生成手段、）

2g　ジャンプテーブル情報生成部（ジャンプテーブル記録手段、サブルーチン機械語アドレステーブル生成手段）

3　命令セットシミュレータプログラムソースコード出力部（データ依存分岐命令生成手段）

3a　サブルーチンプログラムソース生成部（サブルーチンソースコード出力手段、サブルーチン呼出し命令出力手段）

3a4　プログラムラベル記述生成部（識別子付加手段）

3a5　機械語命令種別判定部（無条件分岐命令出力手段）

3a6　単純分岐命令プログラム記述生成部（無条件分岐命令出力手段、サブルーチン呼出し命令出力手段）

3a7　単純サブルーチン呼出し命令プログラム記述生成部（サブルーチンソースコード出力手段、サブルーチン呼出し命令出力手段）

3a8　データ依存分岐命令プログラム記述生成部（無条件分岐命令出力手段、データ依存分岐命令生成手段、サブルーチン呼出し命令出力手段、データ依存サブルーチン呼出し命令生成手段）

3a9　データ依存サブルーチン呼出し命令プログラム記述生成部（サブルーチンソースコード出力手段、サブルーチン呼出し命令出力手段、サブルーチンアドレス検索処理命令生成手段）

3e　サブルーチン引数共通化処理部（サブルーチンソースコード出力手段、レジスタ変数展開手段）

23　命令セットシミュレータプログラムソースコード出力部（データ依存分岐命令生成手段、レジスタ変数展開手段）

33　命令セットシミュレータプログラムソースコード出力部（サブルーチンソースコード出力手段）

r3　ジャンプテーブル情報リスト（ジャンプテーブル情報記憶部、機械語アドレステーブル）

S、2S、3S　命令セットシミュレータ

【図１】

```
int prime_count = 0;
int primes[MAX_COUNT];
void put_prime(int i)
{
  primes[prime_count ++] = i;
}
```

```
void get_primes()
{
  int i;
  put_prime(2);
  i = 3;
  while(prime_count < MAX_COUNT){
    if(check_prime(i)){ put_prime(i);}
    i += 2;
  }
}
```

【図２】

```
00008218 <put_prime>:
    8218:   e59f3014    ldr     r3, [pc, #20]      ; 8234
    821c:   e5932000    ldr     r2, [r3]
    8220:   e59fc010    ldr     ip, [pc, #16]      ; 8238
    8224:   e2821001    add     r1, r2, #1
    8228:   e78c0102    str     r0, [ip, r2, lsl #2]
    822c:   e5831000    str     r1, [r3]
    8230:   e12fff1e    bx      lr
    8234:   0001b308    andeq   fp, r1, r8, lsl #6
    8238:   0001b404    andeq   fp, r1, r4, lsl #8

000083c0 <get_primes>:
    83c0:   e92d4070    push    {r4, r5, r6, lr}
    83c4:   e59f505c    ldr     r5, [pc, #92]      ; 8428
    83c8:   e3a00002    mov     r0, #2
    83cc:   e3a06a7a    mov     r6, #499712        ; 0x7a000
    83d0:   ebffff90    bl      8218 <put_prime>
    83d4:   e2866f47    add     r6, r6, #284        ; 0x11c
    83d8:   e5953000    ldr     r3, [r5]
    83dc:   e2866003    add     r6, r6, #3
    83e0:   e1530006    cmp     r3, r6
    83e4:   c8bd8070    popgt   {r4, r5, r6, pc}
    83e8:   e3a04003    mov     r4, #3
    83ec:   ea000003    b       8400 <get_primes+0x40>
    83f0:   e5953000    ldr     r3, [r5]
    83f4:   e1530006    cmp     r3, r6
    83f8:   c8bd8070    popgt   {r4, r5, r6, pc}
    83fc:   e2844002    add     r4, r4, #2
    8400:   e1a00004    mov     r0, r4
    8404:   ebffff8c    bl      823c <check_prime>
    8408:   e3500000    cmp     r0, #0
    840c:   0afffff7    beq     83f0 <get_primes+0x30>
    8410:   e1a00004    mov     r0, r4
    8414:   ebffff7f    bl      8218 <put_prime>
    8418:   e5953000    ldr     r3, [r5]
    841c:   e1530006    cmp     r3, r6
    8420:   dafffff5    ble     83fc <get_primes+0x3c>
    8424:   e8bd8070    pop     {r4, r5, r6, pc}
    8428:   0001b308    andeq   fp, r1, r8, lsl #6
```

【図３】

```
typedef unsigned int U32;
U32 _my_put_prime_ (U32 r0)
{
  U32 r1=0, r2=0, r3=0, r4=0, r5=0, r6=0, r7=0, r8=0, r9=0, sl=0, fp=0;
  U32 ip=0, sp=0, lr=0, pc=0, cv=0, cc=0, cr=0, cls=0, cge=0, cle=0;

  LDR(AL,0x01a, 3,15,  IMM,-1,0x00000014,0x0023,0x08218)
  LDR(AL,0x01a, 2, 3,  IMM,-1,0x00000000,0x0024,0x0821c)
  LDR(AL,0x01a,12,15,  IMM,-1,0x00000010,0x0025,0x08220)
  ADD(AL,0x020, 1, 2,  IMM,-1,0x00000001,          0x08224)
  STR(AL,0x02a,  0,12,  LSL_I, 2,                  2,0x000c,0x08228)
  STR(AL,0x00a,  1,  3,  IMM,-1, 0x00000000,0x000d,0x0822c)
  _COND_(AL){ return r0;}
}
U32 _my_get_primes_ (U32 sp)
{
  U32 r0=0, r1=0, r2=0, r3=0, r4=0, r5=0, r6=0, r7=0, r8=0, r9=0, sl=0;
  U32 fp=0, ip=0, lr=0, pc=0, cv=0, cc=0, cr=0, cls=0, cge=0, cle=0;

  STM(AL,0x009,13,                 0x00004070,0x0015,0x083c0)
  LDR(AL,0x01a, 5,15,  IMM,-1,0x0000005c,0x0045,0x083c4)
  MOV(AL,0x020, 0, 0,  IMM,-1,0x00000002,          0x083c8)
  MOV(AL,0x020, 6, 0,  IMM,-1,0x0007a000,          0x083cc)
  _COND_(AL){ r0 = _my_put_prime_ (r0);}                  /// bl 8218 <put_prime>
  ADD(AL,0x020, 6, 6,  IMM,-1,0x0000011c,          0x083d4)
  LDR(AL,0x01a, 3, 5,  IMM,-1,0x00000000,0x0046,0x083d8)
  ADD(AL,0x020, 6, 6,  IMM,-1,0x00000003,          0x083dc)
  CMP(AL,0x004, 0, 3,LS0_I,  6,                    0,          0x083e0)
  LDM(GT,0x013,13,                 0x00008070,0x0047,0x083e4)
  MOV(AL,0x020, 4, 0,  IMM,-1,0x00000003,          0x083e8)
  _COND_(AL){ goto L_08400;}                              /// b 8400
L_083f0:
  LDR(AL,0x01a, 3, 5,  IMM,-1,0x00000000,0x0048,0x083f0)
  CMP(AL,0x004, 0, 3,LS0_I, 6,                     0,          0x083f4)
  LDM(GT,0x013,13,                 0x00008070,0x0049,0x083f8)
L_083fc:
  ADD(AL,0x020, 4, 4,  IMM,-1,0x00000002,          0x083fc)
L_08400:
  MOV(AL,0x000, 0, 0,LS0_I, 4,                     0,          0x08400)
  _COND_(AL){ r0 = _my_check_prime_ (r0, sp);} /// bl 823c <check_prime>
  CMP(AL,0x024, 0, 0,  IMM,-1,0x00000000,          0x08408)
  _COND_(EQ){ goto L_083f0;}                              /// beq 83f0
  MOV(AL,0x000, 0, 0,LS0_I, 4,                     0,          0x08410)
  _COND_(AL){ r0 = _my_put_prime_ (r0);}                  /// bl 8218 <put_prime>
  LDR(AL,0x01a, 3, 5,  IMM,-1,0x00000000,0x004a,0x08418)
  CMP(AL,0x004, 0, 3,LS0_I, 6,                     0,          0x0841c)
  _COND_(LE){ goto L_083fc;}                              /// ble 83fc
  LDM(AL,0x013,13,                 0x00008070,0x004b,0x08424)
}
```

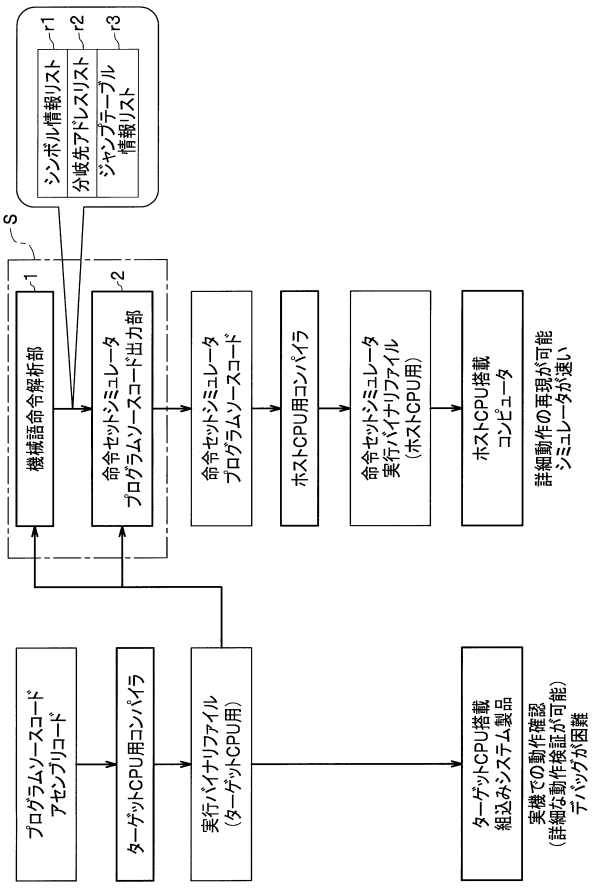
【図４Ａ】

```
U32 _my_get_primes_ (U32 sp)
{
  U32 r0=0, r1=0, r2=0, r3=0, r4=0, r5=0, r6=0, r7=0, r8=0, r9=0, sl=0;
  U32 fp=0, ip=0, lr=0, pc=0, cv=0, cc=0, cz=0, cn=0, cls=0, cge=0, cle=0;

  STM(AL,0x009,13,          0x00004070,0x0015,0x083c0)
  LDR(AL,0x01a, 5,15,  IMM,-1,0x0000005c,0x0045,0x083c4)
  MOV(AL,0x020, 0, 0,  IMM,-1,0x00000002,      0x083c8)
  MOV(AL,0x020, 6, 0,  IMM,-1,0x0007a000,      0x083cc)
  _COND_(AL){ r0 = _my_put_prime_ (r0);}                  /// bl 8218 <put_prime>
012B15D0 A1 4C 66 32 01   mov   eax,dword ptr ds:[0132664Ch]
012B15D5 8B 88 18 E4 31 01   mov   ecx,dword ptr _arm (131E418h)[eax]
012B15DB 8B 15 50 66 32 01   mov   edx,dword ptr ds:[1326650h]
012B15E1 53   push   ebx
012B15E2 57   push   edi
012B15E3 8B 3D 40 68 32 01   mov   edi,dword ptr ds:[1326840h]
012B15E9 C7 84 8A 18 E4 31 01 02 00 00 00 mov   dword ptr _arm (131E418h)[edx+ecx*4],2
012B15F4 41   inc   ecx
  ADD(AL,0x020, 6, 6,  IMM,-1,0x0000011c,      0x083d4)
  LDR(AL,0x01a, 3, 5,  IMM,-1,0x00000000,0x0046,0x083d8)
  ADD(AL,0x020, 6, 6,  IMM,-1,0x00000003,      0x083dc)
  CMP(AL,0x004, 0, 3,LS0_I, 6,          0,      0x083e0)
012B15F5 8B 1F A1 07 00   mov   ebx,7A11Fh
012B15FA 89 88 18 E4 31 01   mov   dword ptr _arm (131E418h)[eax],ecx
012B1600 39 9F 18 E4 31 01   cmp   dword ptr _arm (131E418h)[edi],ebx
  LDM(GT,0x013,13,          0x00008070,0x0047,0x083e4)
012B1606 7E 08   jle   my_get_primes_+40h (12B1610h)
012B1608 5F   pop   edi
012B1609 B8 02 00 00 00   mov   eax,2
012B160E 5B   pop   ebx
}
012B160F C3   ret
012B1610 56   push   esi
  MOV(AL,0x020, 4, 0,  IMM,-1,0x00000003,      0x083e8)
012B1611 BE 03 00 00 00   mov   esi,3
L_08400:
  MOV(AL,0x000, 0, 0,LS0_I, 4,          0,      0x08400)
  _COND_(AL){ r0 = _my_check_prime_ (r0, sp);}
012B1616 8B C6   mov   eax,esi
012B1618 E8 33 FE FF FF   call   _my_check_prime_ (12B1450h)
  CMP(AL,0x024, 0, 0,  IMM,-1,0x00000000,      0x08408)
  _COND_(EQ){ goto L_083f0;}                              /// beq 83f0
012B161D 85 C0   test   eax,eax
012B161F 75 0C   jne   L_083f0+0Ch (12B162Dh)
  _COND_(AL){ goto L_08400;}                              /// b 8400
L_083f0:
  LDR(AL,0x01a, 3, 5,  IMM,-1,0x00000000,0x0048,0x083f0)
  CMP(AL,0x004, 0, 3,LS0_I, 6,          0,      0x083f4)
012B1621 39 9F 18 E4 31 01   cmp   dword ptr _arm (131E418h)[edi],ebx
  LDM(GT,0x013,13,          0x00008070,0x0049,0x083f8)
012B1627 7E 2B   jle   L_083fc (12B1654h)
012B1629 5E   pop   esi
012B162A 5F   pop   edi
012B162B 5B   pop   ebx
}
012B162C C3   ret
```

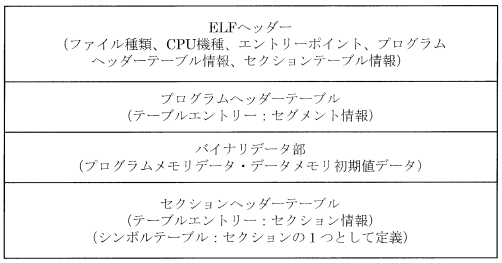
【図４Ｂ】

```
  MOV(AL,0x000, 0, 0,LS0_I, 4,          0,      0x08410)
  _COND_(AL){ r0 = _my_put_prime_ (r0);}                  /// bl 8218 <put_prime>
012B162D A1 4C 66 32 01   mov   eax,dword ptr ds:[0132664Ch]
012B1632 8B 88 18 E4 31 01   mov   ecx,dword ptr _arm (131E418h)[eax]
012B1638 8B 15 50 66 32 01   mov   edx,dword ptr ds:[1326650h]
012B163E 89 B4 8A 18 E4 31 01 mov   dword ptr _arm (131E418h)[edx+ecx*4],esi
012B1645 41   inc   ecx
012B1646 89 88 18 E4 31 01   mov   dword ptr _arm (131E418h)[eax],ecx
  LDR(AL,0x01a, 3, 5,  IMM,-1,0x00000000,0x004a,0x08418)
  CMP(AL,0x004, 0, 3,LS0_I, 6,          0,      0x0841c)
012B164C 39 9F 18 E4 31 01   cmp   dword ptr _arm (131E418h)[edi],ebx
  _COND_(LE){ goto L_083fc;}                              /// ble 83fc
012B1652 7F 05   jg   L_083fc+5 (12B1659h)
L_083fc:
  ADD(AL,0x020, 4, 4,  IMM,-1,0x00000002,      0x083fc)
012B1654 83 C6 02   add   esi,2
012B1657 EB BD   jmp   L_08400 (12B1616h)
  LDM(AL,0x013,13,          0x00008070,0x004b,0x08424)
012B1659 8B C6   mov   eax,esi
012B165B 5E   pop   esi
012B165C 5F   pop   edi
012B165D 5B   pop   ebx
}
012B165E C3   ret
```

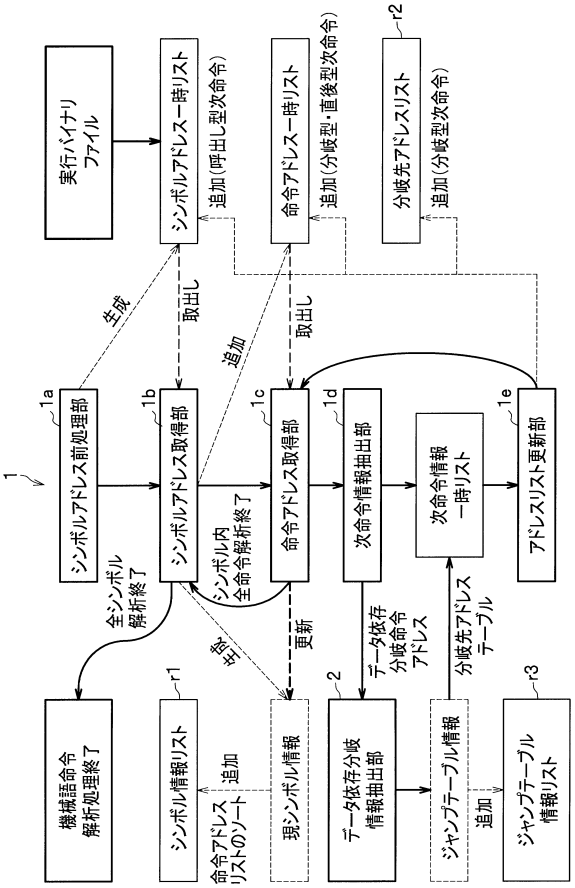
【図５】



【図６】




【図７】

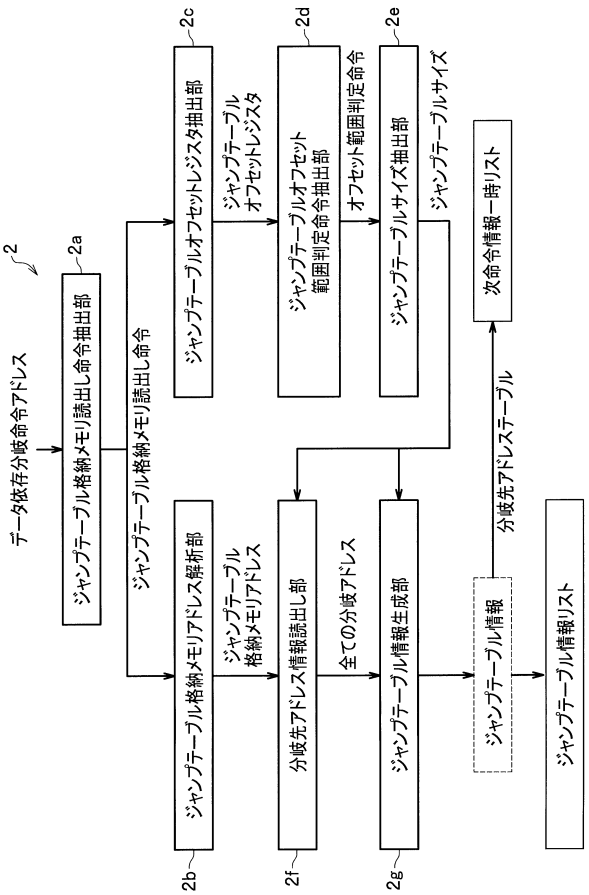



【図８】

サンプルCプログラム1

```
typedef int (* func_pointer)(int);

int f0(int n){ return n + 1;}
int f1(int n){ return 5 / n + 1;}
func_pointer ftab[2] = { f0, f1};
int ff1(int n){ return n << 3;}

int jump_test(int n)
{
        func_pointer f;
        switch(n){
        case 3:         f = ftab[1];    break;
        case 8:         f = ftab[1];    break;
        case 5:         f = ftab[0];    break;
        case 7:         f = ftab[0];    break;
        case 6:         f = ff1;
        break;
        default:        return 0;
        }
        return f(n) + 1;
}
```

【図９】



【図１０】

図8の”jump_test”関数のARMv5命令セットの機械語命令とアセンブリ命令

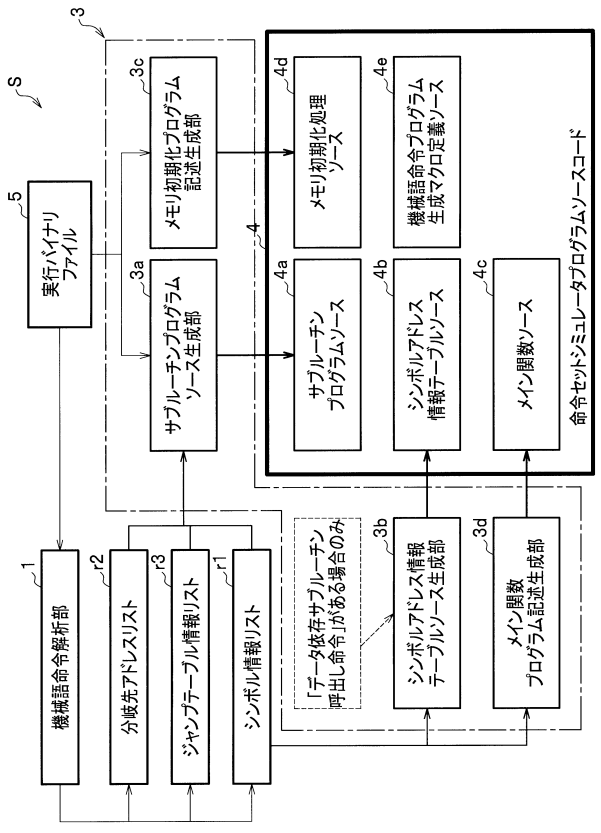
```
00008340 <jump_test>:
  8340:  e2403003  sub     r3, r0, #3
  8344:  e52de004  push    {lr}
  8348:  e3530005  cmp     r3, #5
  834c:  979f f103  ldrls   pc, [pc, r3, lsl #2]        (条件付きデータ依存分岐)
  8350:  ea000011  b       839c                         (無条件分岐)
  8354:  00008384  andeq   r8, r0, r4, lsl #7
  8358:  0000839c  muleq   r0, ip, r3
  835c:  0000836c  andeq   r8, r0, ip, ror #6
  8360:  000083a4  andeq   r8, r0, r4, lsr #7
  8364:  0000836c  andeq   r8, r0, ip, ror #6
  8368:  00008384  andeq   r8, r0, r4, lsl #7
  836c:  e59f 3044  ldr     r3, [pc, #68]
  8370:  e5933000  ldr     r3, [r3]
  8374:  e1a0e 00f  mov     lr, pc              (lr ← 0x8374 + 8)
  8378:  e12f f f 13  bx      r3                  (データ依存サブルーチン呼出し)
  837c:  e2800001  add     r0, r0, #1
  8380:  e49df 004  pop     {pc}                (リターン)
  8384:  e59f 302c  ldr     r3, [pc, #44]
  8388:  e5933004  ldr     r3, [r3, #4]
  838c:  e1a0e00f  mov     lr, pc              (lr ← 0x838c + 8)
  8390:  e12f f f 13  bx      r3                  (データ依存サブルーチン呼出し)
  8394:  e2800001  add     r0, r0, #1
  8398:  e49df 004  pop     {pc}                (リターン)
  839c:  e3a00000  mov     r0, #0
  83a0:  e49df 004  pop     {pc}                (リターン)
  83a4:  e59f 3010  ldr     r3, [pc, #16]
  83a8:  e1a0e00f  mov     lr, pc              (lr ← 0x83a8 + 8)
  83ac:  e12f f f 13  bx      r3                  (データ依存サブルーチン呼出し)
  83b0:  e2800001  add     r0, r0, #1
  83b4:  e49df 004  pop     {pc}                (リターン)
  83b8:  0001a8e0  andeq   sl, r1, r0, ror #17
  83bc:  00008238  andeq   r8, r0, r8, lsr r2
```

【図１１】

図8の”jump_test”関数のx86(64-bit)命令セットの機械語命令とアセンブリ命令

```
0000000000400560 <jump_test>:
 400560:  8d 47 fd                lea    0xf f f f f f f f f f f f f f f d(%rdi),%eax
 400563:  83 f8 05                cmp    $0x8,%rsp
 400567:  77 09                   ja     400575                    (条件分岐)
 40056a:  89 c0                   mov    %eax,%eax
 40056c:  ff 24 c5 38 07 40 00    jmpq   *0x400738(,%rax,8)        (データ依存分岐)
 400575:  31 c0                   xor    %eax,%eax
 400577:  48 83 c4 08             add    $0x8,%rsp
 40057b:  c3                      retq                             (リターン)
 40057c:  0f 1f 40 00             nopl   0x0(%rax)
 400580:  48 8b 05 99 05 20 00    mov    2098585(%rip),%rax  # 600b20 <ftab>
 400587:  ff d0                   callq  *%rax  (データ依存サブルーチン呼出し)
 400589:  48 83 c4 08             add    $0x8,%rsp
 40058d:  83 c0 01                add    $0x1,%eax
 400590:  c3                      retq                             (リターン)
 400591:  48 8b 05 90 05 20 00    mov    2098576(%rip),%rax# 600b28 <ftab+0x8>
 400598:  eb ed                   jmp    400587                    (無条件分岐)
 40059a:  b8 40 05 40 00          mov    $0x400540,%eax
 40059f:  eb e6                   jmp    400587                    (無条件分岐)
 4005a1:  0f 1f 80 00 00 00 00    nopl   0x0(%rax)
 4005a8:  0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
 4005af:  00

 400738:  91 05 40 00 00 00 00 00      0000000000400591
 400740:  75 05 40 00 00 00 00 00      0000000000400575
 400748:  80 05 40 00 00 00 00 00      0000000000400580
 400750:  9a 05 40 00 00 00 00 00      000000000040059a
 400758:  80 05 40 00 00 00 00 00      0000000000400580
 400760:  91 05 40 00 00 00 00 00      0000000000400591
```

【図１２】




【図１３】

ARMv5用命令セットシミュレータプログラムにおけるCPUリソースのデータ構造記述

```
typedef unsigned int        U32;
typedef signed int          S32;
typedef unsigned short      U16;
typedef signed short        S16;
typedef unsigned char       U8;
typedef signed char         S8;

/// CPUリソースデータ構造
typedef struct
{
        char  _mem[MEM_SIZE];
        U32   r[16];          // r[13] → sp
                              // r[14] → lr
                              // r[15] → pc
        U32   cr, cc, cv, cls, cge, cle;
} ARMv5;
extern ARMv5 arm;
```

【図１４】

CPUリソース参照用マクロ定義と命令実行条件判定用マクロ定義

```
#define _r_(n)          arm.r[n]
#define _R_             arm.cr
#define _C_             arm.cc
#define _V_             arm.cv
#define _LS_            arm.cls
#define _GE_            arm.cge
#define _LE_            arm.cle

#define _m8_(n)         ¥
                (*(U8*) (&arm._mem[n]))
#define _m16_(n)        ¥
                (*(U16*)(&arm._mem[n]))
#define _m32_(n)        ¥
                (*(U32*)(&arm._mem[n]))

#define _COND_AL_       (1)
#define _COND_EQ_       (!_R_)
#define _COND_NE_       (_R_ != 0)
#define _COND_CS_       (_C_ != 0)
#define _COND_CC_       (!_C_)
#define _COND_MI_       (((S32)_R_) < 0)
#define _COND_PL_       (((S32)_R_) >= 0)
#define _COND_VS_       (((S32)_V_) < 0)
#define _COND_VC_       (((S32)_V_) >= 0)
#define _COND_HI_       (!_LS_)
#define _COND_LS_       (_LS_ != 0)
#define _COND_GE_       (_GE_ != 0)
#define _COND_LT_       (!_GE_)
#define _COND_GT_       (!_LE_)
#define _COND_LE_       (_LE_ != 0)
#define _COND_(c)       if _COND_##c##_
```

【図１５】

図10の”jump_test”関数のARMv5命令セット機械語・アセンブリ記述の4命令

```
00008340 <jump_test>:
   8340:        e2403003 sub     r3, r0, #3
   8344:        e52de004 push    {lr}
   8348:        e3530005 cmp     r3, #5
/// <省略>
   8380:        e49df004 pop     {pc}                    (リターン)
```

【図１６】

図15の４つのARMv5命令セット機械語命令に対するマクロ呼出し記述

```
   SUB(AL,0x020, 3, 0,   IMM,-1,0x00000003, 0x08340)
   STR(AL,0x009,14,13,  IMM,-1,0x00000004, 0x08344)
   CMP(AL,0x024, 0, 3,   IMM,-1,0x00000005, 0x08348)
/// <省略>
   LDR(AL,0x012,15,13,  IMM,-1,0x00000004, 0x08380)  /// リターン
```

【図１７】

SUBマクロ定義とCMPマクロ定義

```
#define SUB(c, v, rd, rn, sh, rm, val, pc)      _SUB_(1, 0, 0, c, v, rd, rn, sh, rm, val)
#define CMP(c, v, rd, rn, sh, rm, val, pc)      _SUB_(0, 0, 0, c, v, rd, rn, sh, rm, val)
```

【図１８】

_SUB_マクロ定義（SUBマクロとCMPマクロから呼び出される）

```
#define _SUB_(w_rd, cin, swap_op, c, v, rd, rn, sh, rm, val)                    ¥
_COND_(c){ U32 op1, op2, res;                                                   ¥
   if(swap_op)  { op2 = _r_(rn); op1 = sh(rm, val);}                            ¥
   else         { op1 = _r_(rn); op2 = sh(rm, val);}                            ¥
   res = op1 - op2 - cin;                                                       ¥
   if(_fS_(v))   {                                                              ¥
         _R_  = (res);                                                          ¥
         _C_  = (cin) ? (op1 > op2) : (op1 >= op2);                             ¥
         _V_  = ((op1 ^ op2) & (op1 ^ res));                                    ¥
         _LS_ = (cin) ? (op1 < op2) : (op1 <= op2);                             ¥
         _GE_ = (cin) ? (((S32)op1) > ((S32)op2)) : (((S32)op1) >= ((S32)op2)); ¥
         _LE_ = (cin) ? (((S32)op1) < ((S32)op2)) : (((S32)op1) <= ((S32)op2)); ¥
   }                                                                            ¥
   if(w_rd)      { _r_(rd) = res;}                                              ¥
}
```

【図１９】

その他のマクロ定義

```
#define IMM(rm, val)                    (val)

#define _fS_(v)                         ((v >> 2) & 1)
/// 以下は、STRマクロ内部で引用される
#define _fB_(v)                         ((v >> 8) & 1)
#define _fH_(v)                         ((v >> 6) & 1)
#define _fP_(v)                         ((v >> 3) & 1)
#define _fU_(v)                         ((v >> 1) & 1)
#define _fW_(v)                         ((v >> 0) & 1)
```

【図２０】

STRマクロ定義

```
#define STR(c, v, rd, rn, sh, rm, val, pc)                              ¥
_COND_(c){ U32 ofs, rn2, addr, d;                                       ¥
   _ADDR_(v, rn, sh, rm, val, pc)                                       ¥
   d = (rd == 15) ? pc + 12 : _r_(rd);                                  ¥
   if(_fB_(v))          { _m8_(addr)  = (U8)  (d & 0xff);}              ¥
   else if(_fH_(v))     { _m16_(addr) = (U16) (d & 0xffff);}            ¥
   else                 { _m32_(addr) = d;}                             ¥
   D_CACHE_SIM(addr, 0)                                                 ¥
}
```

【図２１】

_ADDR_, D_CACHE_SIM各マクロ定義

```
#define _ADDR_(v, rn, sh, rm, val, pc)                                  ¥
   if(rn == 15)         { _r_(15) = pc + 8;}                            ¥
   ofs = sh(rm, val);                                                   ¥
   if(_fU_(v))          { rn2 = _r_(rn) + ofs;}                         ¥
   else                 { rn2 = _r_(rn) - ofs;}                         ¥
   addr = (_fP_(v)) ? rn2 : _r_(rn);                                    ¥
   if(_fW_(v) || !_fP_(v))     { _r_(rn) = rn2;}

#if defined(ENABLE_CACHE_MODEL)
void D_Cache_Sim(unsigned int addr, int isRead);
#define D_CACHE_SIM(addr, isRead)       D_Cache_Sim(addr, isRead);
#else
#define D_CACHE_SIM(addr, isRead)       // 「空文字列」に変換
#endif
```

【図２２】

LDRマクロ定義

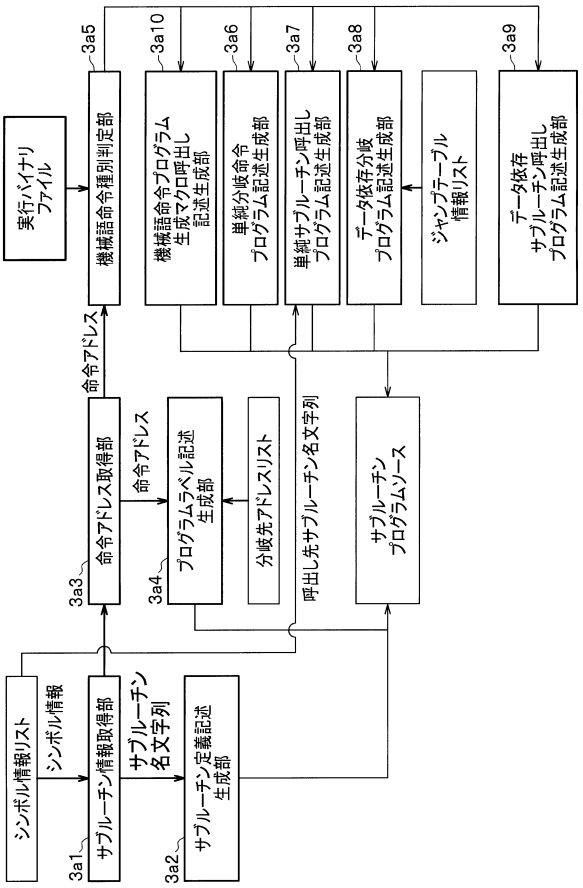
```
#define LDR(c, v, rd, rn, sh, rm, val, pc)                              ¥
_COND_(c){ U32 ofs, rn2, addr;                                          ¥
   _ADDR_(v, rn, sh, rm, val, pc)                                       ¥
   if(_fS_(v))          {                                               ¥
         if(_fH_(v))          { _r_(rd) = (U32)((S16)_m16_(addr));}     ¥
         else                 { _r_(rd) = (U32)((S8)_m8_(addr));}       ¥
   }                                                                    ¥
   else{                                                                ¥
         if(_fB_(v))          { _r_(rd) =  _m8_(addr); }                ¥
         else if(_fH_(v))     { _r_(rd) =  _m16_(addr);}                ¥
         else                 { _r_(rd) =  _m32_(addr);}                ¥
   }                                                                    ¥
   D_CACHE_SIM(addr, 1)                                                 ¥
   if(rd == 15){ return;}                                               ¥
}
```

【図２３】

図16の4つの機械語命令プログラム生成マクロ呼出しによって生成されるプログラム記述

```
/// SUB(AL,0x020, 3, 0, IMM,-1,0x00000003,_ _0x08340)
if (1){
  U32 op1, op2, res;
  if(0) { op2 = arm.r[0]; op1 = (0x00000003);}
  else  { op1 = arm.r[0]; op2 = (0x00000003);}
  res = op1 - op2 - 0;
  if(((0x020 >> 2) & 1)) {
        arm.cr = (res);
        arm.cc = (0) ? (op1 > op2) : (op1 >= op2);
        arm.cv = ((op1 ^ op2) & (op1 ^ res));
        arm.cls = (0) ? (op1 < op2) : (op1 <= op2);
        arm.cge = (0) ? (((S32)op1) > ((S32)op2)) : ((((S32)op1) >= ((S32)op2));
        arm.cle = (0) ? (((S32)op1) < ((S32)op2)) : ((((S32)op1) <= ((S32)op2));
  }
  if(1) { arm.r[3] = res;}
}
/// STR(AL,0x009,14,13, IMM,-1,0x00000004,_ _ 0x08344)
if (1){
  U32 ofs, rn2, addr, d;
  if(13 == 15) { arm.r[15] = 0x08344 + 8;}
  ofs = (0x00000004);
  if(((0x009 >> 1) & 1))      { rn2 = arm.r[13] + ofs;}
  else                        { rn2 = arm.r[13] - ofs;}
  addr = (((0x009 >> 3) & 1)) ? rn2 : arm.r[13];
  if(((0x009 >> 0) & 1) || !((0x009 >> 3) & 1)) { arm.r[13] = rn2;}
  d = (14 == 15) ? 0x08344 + 12 : arm.r[14];
  if(((0x009 >> 8) & 1))      { (*(U8*) ((&arm._mem[addr]))) = (U8) (d & 0xff);}
  else if(((0x009 >> 6) & 1)){ (*(U16*)((&arm._mem[addr]))) = (U16) (d & 0xffff);}
  else                        { (*(U32*)((&arm._mem[addr]))) = d;}
}
/// CMP(AL,0x024, 0, 3, IMM,-1,0x00000005,_ _0x08348)
if (1){
  U32 op1, op2, res;
  if(0) { op2 = arm.r[3]; op1 = (0x00000005);}
  else  { op1 = arm.r[3]; op2 = (0x00000005);}
  res = op1 - op2 - 0;
  if(((0x024 >> 2) & 1)) {
        arm.cr = (res);
        arm.cc = (0) ? (op1 > op2) : (op1 >= op2);
        arm.cv = ((op1 ^ op2) & (op1 ^ res));
        arm.cls = (0) ? (op1 < op2) : (op1 <= op2);
        arm.cge = (0) ? (((S32)op1) > ((S32)op2)) : ((((S32)op1) >= ((S32)op2));
        arm.cle = (0) ? (((S32)op1) < ((S32)op2)) : ((((S32)op1) <= ((S32)op2));
  }
  if(0) { arm.r[0] = res;}
}
//<省略>
//LDR(AL,0x012,15,13, IMM,-1,0x00000004,0x08380)
if (1){
  U32 ofs, rn2, addr;
  if(13 == 15) { arm.r[15] = 0x08380 + 8;}
  ofs = (0x00000004);
  if(((0x012 >> 1) & 1))                { rn2 = arm.r[13] + ofs;}
  else                                  { rn2 = arm.r[13] - ofs;}
  addr = (((0x012 >> 3) & 1)) ? rn2 : arm.r[13];
  if(((0x012 >> 0) & 1) || !((0x012 >> 3) & 1)) { arm.r[13] = rn2;}
  if(((0x012 >> 2) & 1)) {
        if(((0x012>> 6) & 1)) { arm.r[15] = (U32)((S16)(*(U16*)((&arm._mem[addr]))));}
        else                  { arm.r[15] = (U32)((S8)(*(U8*)((&arm._mem[addr]))));}
  }
  else{
        if(((0x012 >> 8) & 1))          { arm.r[15] = (*(U8*)((&arm._mem[addr])));}
        else if(((0x012 >> 6) & 1))     { arm.r[15] = (*(U16*)((&arm._mem[addr])));}
        else                            { arm.r[15] = (*(U32*)((&arm._mem[addr])));}
  }
  if(15 == 15){ return;}
}
```

【図２４】




【図２５】

”jump_test”関数のサブルーチン定義記述出力

```
void _my_jump_test(){
        /// 機械語命令プログラム記述部分
}
```

【図２８】

サブルーチン呼出し命令

```
8454:    ebffffb9    bl    8340 <jump_test> (jump_testの呼出し)
```

【図２６】

無条件分岐命令

```
8350:        ea000011 b   839c                    (無条件分岐)
```

【図２９】

サブルーチン呼出し命令のプログラム記述

```
_COND_(AL){ _my_jump_test();} /// if(1){ _my_jump_test();}に変換される
```

【図２７】

無条件分岐命令のプログラム記述

```
_COND_(AL){ goto L_0839c;} /// if(1){ goto L_0839c;}に変換される
```

【図３０】

条件付きデータ依存分岐命令と直後の無条件分岐命令

```
834c:    979ff103 ldrls     pc, [pc, r3, lsl #2]    (条件付きデータ依存分岐)
8350:    ea000011 b         839c                    (無条件分岐)
```

【図３１】

条件付きデータ依存分岐命令と直後の無条件分岐命令のプログラム記述

```
_COND_(LS){
  switch(_r_(3)){
        case  1: goto L_0839c;
        case  2: goto L_0836c;
        case  3: goto L_083a4;
        case  4: goto L_0836c;
        default: goto L_08384;
  }
}
_COND_(AL){ goto L_0839c;} /// if(1){ goto L_0839c;}に変換される
```

【図３２】

データ依存サブルーチン呼出しを実現する２つの機械語命令

```
83a8:       e1a0e00f mov    lr, pc      (lr ← 0x83a8 + 8)
83ac:       e12fff13 bx     r3          (データ依存サブルーチン呼出し)
```

【図３３】

データ依存サブルーチン呼出し命令のプログラム記述

```
_COND_(AL){ _FP_INFO_ * fpi = _GET_FPI_(_r_(3)); fpi->func();}
```

【図３４】

図11の”jump_test”関数の機械語命令記述に対応するプログラム記述の出力例

```
void _my_jump_test_ ()
{
   SUB(AL,0x020, 3, 0,  IMM,-1,0x00000003, 0x08340)
   STR(AL,0x009,14,13,  IMM,-1,0x00000004, 0x08344)
   CMP(AL,0x024, 0, 3,  IMM,-1,0x00000005, 0x08348)
   _COND_(LS){
              switch(_r_(3)){
              case  1: goto L_0839c;
              case  2: goto L_0836c;
              case  3: goto L_083a4;
              case  4: goto L_0836c;
              default: goto L_08384;
              }
   }
   _COND_(AL){ goto L_0839c;}
L_0836c:
   LDR(AL,0x01a, 3,15,  IMM,-1,0x00000044, 0x0836c)
   LDR(AL,0x01a, 3, 3,  IMM,-1,0x00000000, 0x08370)
   MOV(AL,0x000,14, 0,LS0_I,15,       0, 0x08374)
   _COND_(AL){ _FP_INFO_ * fpi = _GET_FPI_(_r_(3)); fpi->func();}
   ADD(AL,0x020, 0, 0,  IMM,-1,0x00000001, 0x0837c)
   LDR(AL,0x012,15,13,  IMM,-1,0x00000004, 0x08380) /// リターン命令
L_08384:
   LDR(AL,0x01a, 3,15,  IMM,-1,0x0000002c, 0x08384)
   LDR(AL,0x01a, 3, 3,  IMM,-1,0x00000004, 0x08388)
   MOV(AL,0x000,14, 0,LS0_I,15,       0, 0x0838c)
   _COND_(AL){ _FP_INFO_ * fpi = _GET_FPI_(_r_(3)); fpi->func();}
   ADD(AL,0x020, 0, 0,  IMM,-1,0x00000001, 0x08394)
   LDR(AL,0x012,15,13,  IMM,-1,0x00000004, 0x08398) /// リターン命令
L_0839c:
   MOV(AL,0x020, 0, 0,  IMM,-1,0x00000000, 0x0839c)
   LDR(AL,0x012,15,13,  IMM,-1,0x00000004, 0x083a0) /// リターン命令
L_083a4:
   LDR(AL,0x01a, 3,15,  IMM,-1,0x00000010, 0x083a4)
   MOV(AL,0x000,14, 0,LS0_I,15,       0, 0x083a8)
   _COND_(AL){ _FP_INFO_ * fpi = _GET_FPI_(_r_(3)); fpi->func();}
   ADD(AL,0x020, 0, 0,  IMM,-1,0x00000001, 0x083b0)
   LDR(AL,0x012,15,13,  IMM,-1,0x00000004, 0x083b4) /// リターン命令
}
```

【図３５】

シンボルアドレス情報を格納する_FP_INFO_構造体

```
typedef void (* _FP_TYPE_)(); /// 関数ポインタデータ型
typedef struct ST_FP_INFO_{
   unsigned int addr;    /// シンボルアドレス
   _FP_TYPE_    func;    /// シンボルに対応するサブルーチンのアドレス
} _FP_INFO_;
```

【図３６】

_FP_INFO_構造体の初期値設定用プログラム記述

```
#define _FP_SIZE_   6 /// シンボル情報テーブルに格納されたシンボル情報の個数

_FP_INFO_ fp_info_array[_FP_SIZE_] = {
            {0x8218, _my_f0},
            {0x8220, _my_f1},
            {0x8238, _my_ff1},
            {0x8338, _my_ff2},
            {0x8340, _my_jump_test},
            {0x8448, _my_main},
};
```

【図３７】

_FP_INFO_構造体のポインタを返す_GET_FPI_関数の定義

```
_FP_INFO_  _GET_FPI_(unsigned int addr)
{
  int i;
  _FP_INFO_ * fpi;
  for(i = 0; i < _FP_SIZE_; i ++){
       if(fp_info_array[i].addr == addr){
           return &fp_info_array[i]; /// addrに対応する_FP_INFO_構造体のポインタ
       }
  }
  return 0; /// addrに対応するサブルーチンは存在しない場合、0を返す
}
```

【図３８】

メモリ初期化プログラム記述の例

```
void install_mem()
{
   /// <省略：他のセクションの初期化プログラム記述>
   ////    (1260c - 127e7) : .rodata
   {
            static unsigned char data[476] = {
            0x25, 0x64, 0x20, 0x2d, 0x3e, 0x20, 0x25, 0x64,
            0x0a, 0x00, 0x00, 0x00, 0x24, 0xa9, 0x01, 0x00,
            /// <省略>
            };
            memcpy(&arm._mem[0x1260c], data, 476);
   }
   /// <省略：他のセクションの初期化プログラム記述>
}
```

【図３９】

命令セットシミュレータの最上位関数のプログラム記述例１

```
int main()
{
   install_mem(); /// メモリ初期化
   my__mainCRTStartup();
   return 0;
}
```

【図４０】

命令セットシミュレータの最上位関数のプログラム記述例２

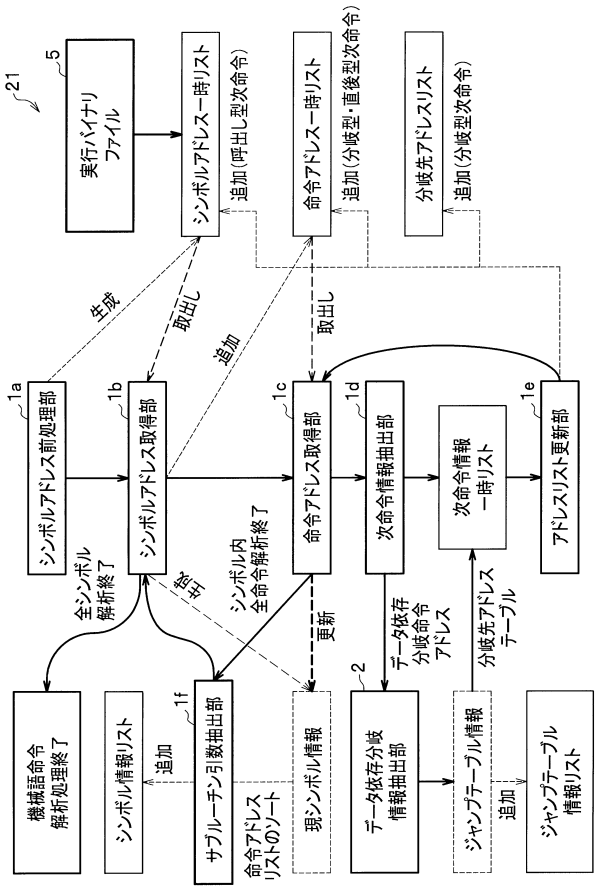
```
int main(int argc, char * argv[])
{
   install_mem(); /// メモリ初期化
   install_main_param(arg, argv);
   my_main();
   return _r_(0); /// main関数の戻り値：arm.r[0]に格納
}
```

【図４１】

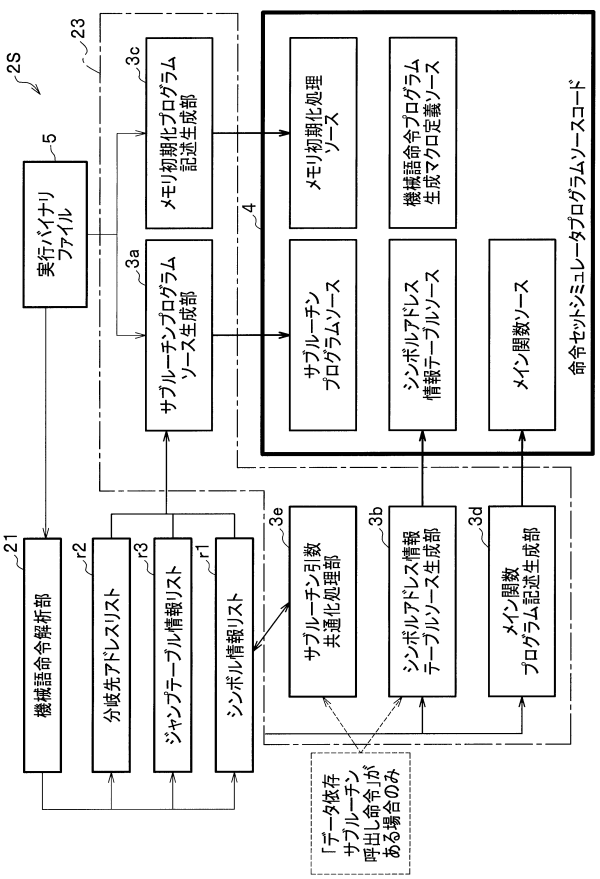
main関数の引数情報をCPUメモリに書き込むプログラム記述

```
void install_mem_param(int argc, char * argv[])
{
   int i, str_size = 0;
   int str_addr, p_str_addr;
   for(i = 0; i < argc; i ++){ str_size += strlen(argv[i]) + 1;}
   str_addr = MEM_SIZE - str_size; /// argvの文字列データ領域確保
   p_str_addr = str_addr - argc * 4; /// argvの文字列ポインタデータ領域確保
   _r_(0) = argc; /// main 第１引数 (argc)
   _r_(1) = p_str_addr; /// main 第２引数 (argv)
   _r_(13) = p_str_addr; /// スタックポインタ設定
   for(i = 0; i < argc, i ++){
              int len = strlen(argv[i]) + 1;
              memcpy(&arm._mem[str_addr], argv[i], len); /// argvの文字列データコピー
              _m32_(p_str_addr) = str_addr;  /// argvの文字列アドレスデータコピー
              str_addr += len;
              p_str_addr += 4;
   }
}
```

【図４２】




【図４３】

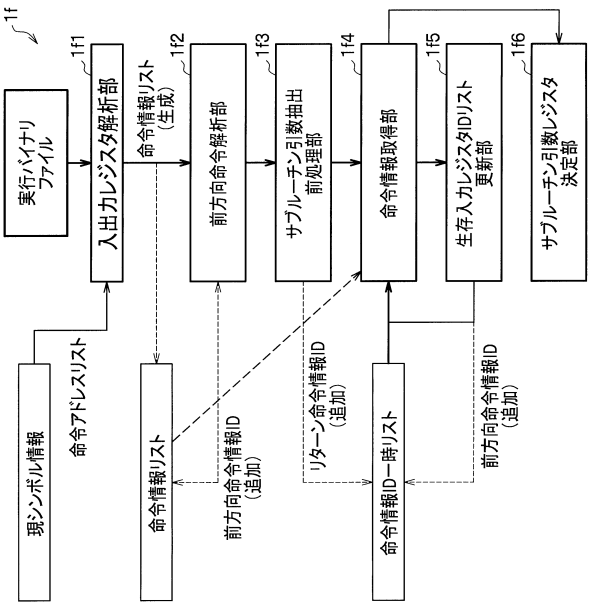



【図４４】

図10の”jump_test”関数のARMv5機械語命令の入出力レジスタ

```
00008340 <jump_test>:
   8340:e2403003  sub     r3, r0, #3                  I(0), O(3)
   8344:e52de004  push    {lr}                        I(13,14), O(13)
   8348:e3530005  cmp     r3, #5                      I(3), O(16)
   834c:979ff103  ldrls   pc, [pc, r3, lsl #2]        I(3,16), O()
          ///   分岐先：{0x8384, 0x839c, 0x836c, 0x83a4}
   8350:ea000011  b       839c                        I(), O()
/// 8354－8368: ジャンプテーブル領域
   836c:e59f3044  ldr     r3, [pc, #68]               I(), O(3)
   8370:e5933000  ldr     r3, [r3]                    I(3), O(3)
   8374:e1a0e00f  mov     lr, pc                      I(), O(14)
   8378:e12fff13  bx      r3                          I(3), O()
   837c:e2800001  add     r0, r0, #1                  I(0), O(0)
   8380:e49df004  pop     {pc}                        I(13), O(13)
   8384:e59f302c  ldr     r3, [pc, #44]               I(), O(3)
   8388:e5933004  ldr     r3, [r3, #4]                I(3), O(3)
   838c:e1a0e00f  mov     lr, pc                      I(), O(14)
   8390:e12fff13  bx      r3                          I(3), O()
   8394:e2800001  add     r0, r0, #1                  I(0), O(0)
   8398:e49df004  pop     {pc}                        I(13), O(13)
   839c:e3a00000  mov     r0, #0                      I(), O(0)
   83a0:e49df004  pop     {pc}                        I(13), O(13)
   83a4:e59f3010  ldr     r3, [pc, #16]               I(), O(3)
   83a8:e1a0e00f  mov     lr, pc                      I(), O(14)
   83ac:e12fff13  bx      r3                          I(3), O()
   83b0:e2800001  add     r0, r0, #1                  I(0), O(0)
   83b4:e49df004  pop     {pc}                        I(13), O(13)
```

【図４５】



【図４６】

CPUリソース参照用マクロ定義と命令実行条件判定用マクロ定義

```
#define _r_(n)   r##n
#define r13      sp  // r13の別名をspとする

#define _R_      cr
#define _C_      cc
#define _V_      cv
#define _LS_     cls
#define _GE_     cge
#define _LE_     cle
```

【図４７】

”jump_test”関数のサブルーチン定義記述出力（第２実施形態）

```
U32 _my_jump_test(U32 sp, U32 r0){
  U32 r1, r2, r3, r4, r5, r6, r7, r8;   /// r0は引数なので除外
  U32 r9, r10, r11, r12, r14, r15;   /// sp (r13)は引数なので除外
  U32 cr, cc, cv, cls, cge, cle;

            /// 機械語命令プログラム記述部分
}
```

【図４８】

サブルーチン呼出し命令のプログラム記述（第２実施形態）

```
_COND_(AL){ r0 = _my_jump_test(sp, r0);}
```

【図４９】

関数ポインタデータ型の定義（第２実施形態）

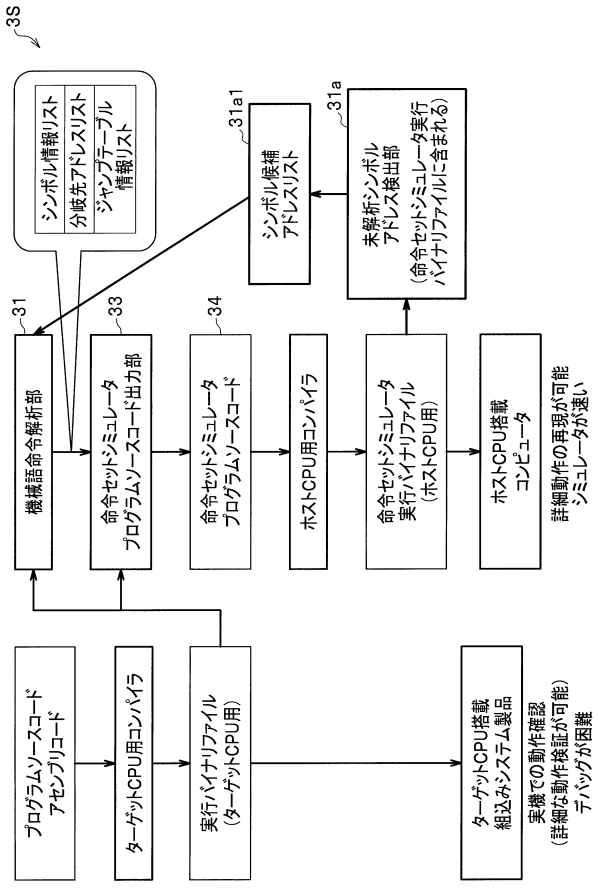
```
typedef U32 (* _FP_TYPE_)(U32 sp, U32 r0); /// 関数ポインタデータ型
```

【図５０】

データ依存サブルーチン呼出し命令のプログラム記述（第２実施形態）

```
_COND_(AL){ _FP_INFO_ * fpi = _GET_FPI_(_r_(3)); r0 = fpi->func(sp, r0);}
```

【図５１】




【図５２】

_FP_INFO_構造体のポインタを返す_GET_FPI_関数の定義（第３実施形態）

```
_FP_INFO_  _GET_FPI_(unsigned int addr)
{
   int i;
   _FP_INFO_ * fpi;
   for(i = 0; i < _FP_SIZE_; i ++){
         if(fp_info_array[i].addr == addr){
            return &fp_info_array[i]; /// addrに対応する_FP_INFO_構造体のポインタ
         }
   }
   printf("Unregistered symbol at address (0x%08x) called via _GET_FPI!!¥n", addr);
   /// addrを「シンボル候補アドレスリスト」ファイルに「追加で書き出す」
   /// (append)するソースコードをこの部分に埋め込む
   exit(-1); /// シミュレータを強制終了する
   return 0; /// addrに対応するサブルーチンは存在しない場合、0を返す
}
```

【図５３】

```
#define FETCH(a) case a:
#define SAVESTATE(a) PC = a;SP = S - M;FP = F - M; AC = A;
#define ADD(a) FETCH(a); A += *S++;
#define ADI(a, n) FETCH(a); A += n;
#define CBLS(a, n) FETCH(a); if (*S++ <= A){P = n; break;}
#define HALT(a) FETCH(a); SAVESTATE(a); return 1;
#define JSR(a, n) FETCH(a); *--S = a + 2; P = n; break;
#define LAI(a, n) FETCH(a); A = n;
#define LAR(a, n) FETCH(a); A = F[n];
#define LDI(a, n) FETCH(a); *--S = A; A = n;
#define LDR(a, n) FETCH(a); *--S = A; A = F[n];
#define LDS(a, n) FETCH(a); S = M + (n);
#define LINK(a, n) FETCH(a); *--S = F - M; F = S;S += n;
#define PA(a) FETCH(a); *--S = A;
#define PI(a, n) FETCH(a); *--S = n;
#define SAR(a, n) FETCH(a); F[n] = A;
#define UNLK(a, n) FETCH(a); S = F; F = M + *S++; P = *S++; S += n; break;
```

【図５４】

```
int execute(M)
register UL * M;
{
 register UL * S = M, *F = H,
             A = 0, P = START;
 for (;;) /* ever */
  switch (P) {
  LDS (START, START);
  JSR (START + 2, START + 5);
  HALT(START + 4);
  LINK(START + 5, -2);
  LAI (START + 7, 1);
  SAR (START+ 9, -1);
  PI (START + 11, 24);
  JSR (START + 13, START + 31);
  SAR (START + 15, -2);
  LAR (START + 17, -1);
  ADI (START + 19, 1);
  SAR (START + 21, -1);
  LDI (START + 23, 100);
  CBLS(START + 25, START + 11)
  LAR (START + 27, -2);
  UNLK(START + 29, 0);
  LINK(START+ 31, 0);
  LAR (START + 33, 2);
  LDI (START + 35, 2);
  CBLS(START + 37, START + 56);
  LAR (START + 39, 2);
  ADI (START + 41, -1);
  PA (START + 43);
  JSR (START + 44, START + 31);
  LDR (START+ 46, 2);
  ADI (START + 48, -2);
  PA (START + 50);
  JSR (START + 51, START + 31);
  ADD (START + 53);
  UNLK(START + 54, 1);
  LAI (START + 56, 1);
  UNLK(START + 58, 1);
  default: SAVESTATE(P);
    return O;
  }
}
```

【図５５Ａ】

```
void Region3() {

  while (1) {
    switch(ac_pc) {

    ...

    case 0x1954: // be PC-348
      ac_behavior_instruction(4);
      ac_behavior_Type_F2B(0, 0, 1, 2, -348);
      ac_behavior_be(0, 0, 1, 2, -348);
      ac_instr_counter++;
      break;

    case 0x1958: // andcc %28, 4, 0
      ac_behavior_instruction(4);
      ac_behavior_Type_F3B(2, 0, 17, 28, 1, 4);
      ac_behavior_andcc_imm(2, 0, 17, 28, 1, 4);
      ac_instr_counter++;
      break;

    ...

    default:
      if ((ac_pc >= 4096) && (ac_pc < 6144)) {
        AC_ERROR(... non-decoded memory location ...);
        ac_stop(EXIT_FAILURE);
      }
      return;
    }
  }
}
```

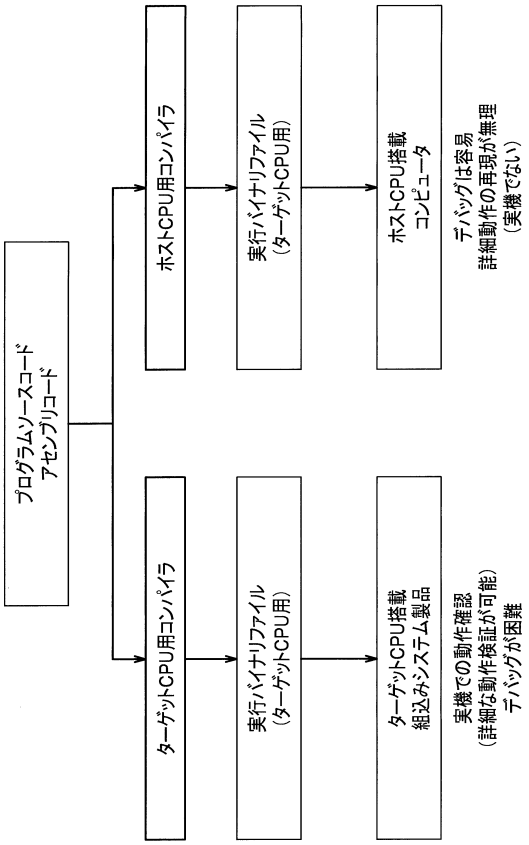
One of the region functions automatically generated for SPARC V8 architecture

【図５５Ｂ】

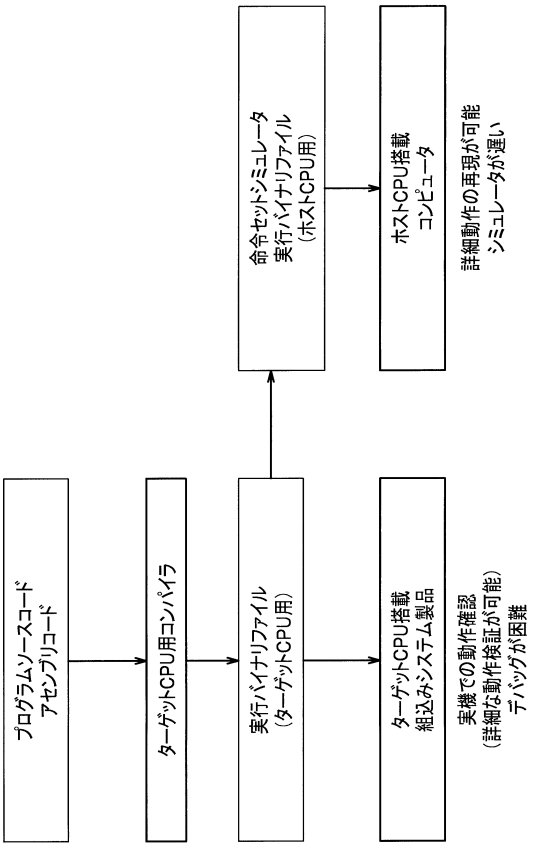
```
void Execute(int argc, char *argv[]) {

  model_syscall.set_prog_args(argc, argv);

  while (!ac_stop_flag) {
    switch(ac_pc >> 11) {

    case 0: Region0(); break;

    case 1: Region1(); break;

    ...

    case 14: Region14(); break;

    default:
      AC_ERROR(... print error and debug info ...);
      ac_stop(EXIT_FAILURE);
      break;
    }
  }
}
```

The main simulation routine

【図５６】


プログラムソースコード
アセンブリコード
ホストCPU用コンパイラ
実行バイナリファイル
（ターゲットCPU用）
ホストCPU搭載
コンピュータ
デバッグは容易
詳細動作の再現が無理
（実機でない）
ターゲットCPU用コンパイラ
実行バイナリファイル
（ターゲットCPU用）
ターゲットCPU搭載
組込みシステム製品
実機での動作確認
（詳細な動作検証が可能）
デバッグが困難

【図５７】


プログラムソースコード
アセンブリコード
ターゲットCPU用コンパイラ
実行バイナリファイル
（ターゲットCPU用）
命令セットシミュレータ
実行バイナリファイル
（ホストCPU用）
ホストCPU搭載
コンピュータ
詳細動作の再現が可能
シミュレータが遅い
ターゲットCPU搭載
組込みシステム製品
実機での動作確認
（詳細な動作検証が可能）
デバッグが困難

フロントページの続き

(56)参考文献 特開平０９－００６６４６（ＪＰ，Ａ）
特表２００１－５１５２４０（ＪＰ，Ａ）
特開平１０－０８３３１１（ＪＰ，Ａ）
特開平１１－０２４９４０（ＪＰ，Ａ）

(58)調査した分野(Int.Cl.，ＤＢ名)
Ｇ０６Ｆ　　８／４１
Ｇ０６Ｆ　　９／４５５
Ｇ０６Ｆ　１１／２８－１１／３６