

CPUのセキュリティ対策を可能にする Reduced Assembly Set Compiler の提案と実装

坂井弘亮

Global Fujitsu Distinguished Engineer



自己紹介

- Global Fujitsu Distinguished Engineer
- セキュリティ・キャンプ協議会 ステアリングコミッティ GM
- SecHack365トレーナー
- SECCON実行委員
- アセンブラ短歌・六歌仙のひとり (白樺派)
- バイナリかるた発案者(エバンジェリスト)
- バイナリ駄洒落発案者(エバンジェリスト)
- 雑誌記事・執筆書籍多数
- 技術士(情報工学部門)

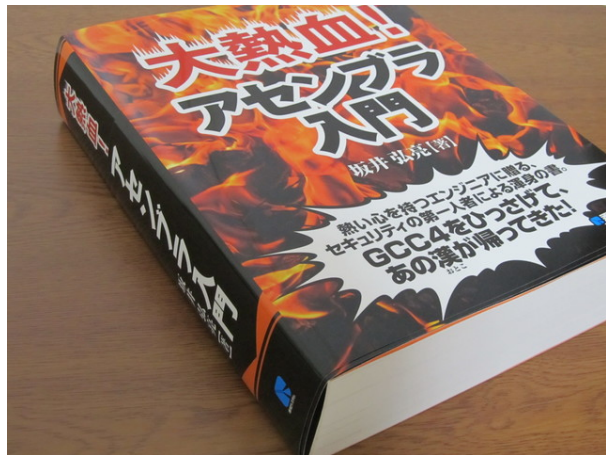


<https://kozozos.jp/>



自己紹介

CPUのドキュメントをほとんど参照せずに、コンパイラの出力からCPUのISA(Instruction Set Architecture)を推測してアセンブリを読解する手法(熱血アセンブラ・メソッド)を確立し、様々なCPUのアセンブリを読解する。



読んだことのあるCPUアセンブリ

AArch64 Alpha ARC ARM ARM(Thumb) ARM(Thumb2) AVR AVR(8bit)
Blackfin CR16 CRIS Epiphany FR30 FR-V FT32 H8/300 H8/300H
H8/300S PA-RISC i386 i960 IA64 IQ2000 LM32 M32C M32R
68HC11 68000 M-CORE MicroBlaze MIPS MIPS16 MIPS64 MMIX
MN10300 Moxie MSP430 NiosII OpenRISC PDP-11 PowerPC
PowerPC64 PRU RISC-V RISC-V(32bit) RISC-V(64bit) RL78 RX S/390
SH SH64 SPARC SPARC(64bit) SPU StrongARM TIC6x V850 VAX x86-
64 XScale Xstormy16 Xtensa MIST32 Z80 6502

動かしたことのあるCPUアセンブリ

AArch64 ARM ARM(Thumb) ARM(Thumb2) AVR Blackfin CR16 CRIS
FR-V FT32 H8/300 H8/300H H8/300S i386 M32C M32R 68HC11
M-CORE MicroBlaze MIPS MIPS16 MIPS64 MN10300 Moxie
MSP430 PowerPC PRU RISC-V RISC-V(32bit) RISC-V(64bit) RL78 RX
SH SH64 SPARC V850 x86-64

本日の内容

こういうことをやっている人間
が**コンパイラ**を作ったらこうなっ
た, という話です

Reduced Assembly Set Compiler の提案と実装

組込みシステムにおいては様々なCPUアーキテクチャが用いられる。一方、ソフトウェアの脆弱性対策のひとつとして、CPUによるセキュリティ対策がある。これにはセキュリティのための専用命令の追加やセキュリティに配慮した新たなCPUアーキテクチャの設計・開発が考えられるが、それらに対応したコンパイラの開発が障壁となる。またスタック・プロテクタに代表されるコンパイラのセキュリティ機構はソースコードを修正せずに適用できる点で有効であるが、新たなセキュリティ機構を検証・導入する際にはコンパイラの改造が前提となる。さらに脆弱性解析を新たなCPUに対して行う場合にはそのCPUの動作理解が必要であるが、そのためにはそのCPUの実行コードを生成できるコンパイラの存在がやはり重要となる。

これらのためには新たなCPUアーキテクチャに迅速に対応できるコンパイラが求められる。従来のオープンソースのコンパイラにはGCC等のように複数のCPUアーキテクチャに対応しているものもあるが、それらの多くは高度な最適化のため、新たなCPUアーキテクチャへの対応は非常に難解になっており現実的でない。

Reduced Assembly Set Compiler の提案と実装

そこで生成するアセンブリのパターンを限定することで新たなCPUアーキテクチャへの対応を容易にした Reduced Assembly Set Compiler (RASC) を提案し, RASCの実装としてCコンパイラNLCCを開発した.

NLCCは64種類のアセンブリパターンを登録するだけで新たなCPUアーキテクチャに対応可能であり, さらに28種類のパターンはビルトインにより省略可能となっている. x86やARMを含む12種類のCPUアーキテクチャに対応させた結果, 平均8時間程度で対応できたため, 1日で新たなCPUアーキテクチャに対応できる 1day-compiler としての利用が可能である. RASCの可能性とNLCCでのセキュリティ機能について検討・考察する.

Reduced Assembly Set Compiler (RASC) とは何か

RASCとは何か

出力するアセンブリのパターンを必要最低限のものに削減することで、アセンブリの出力部を単純化したコンパイラ

そもそも「コンパイラ」
とは何か

コンパイラとは何か

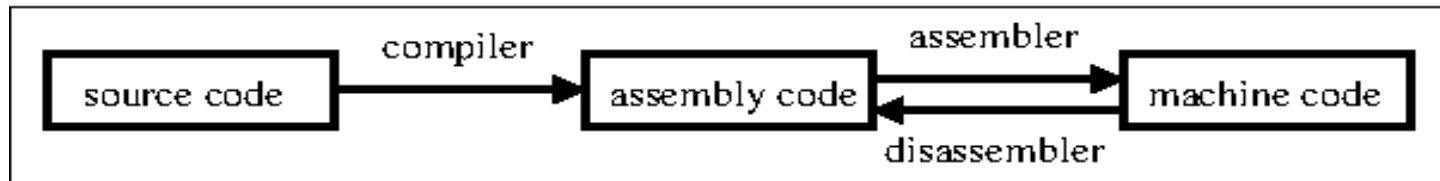
C言語などのプログラミング言語を、アセンブリ言語に変換するツール

CPUが理解するのは機械語コード

機械語コードと1対1に対応する命令語がアセンブリ言語

C言語などのソースコードをアセンブリ言語に変換するのが「コンパイラ」

アセンブリ言語を機械語コードに変換するのは「アセンブラ」



コンパイラとは何か

複数のアセンブリ言語を組み合わせないで実現できないような**高度な表現を解釈**し、アセンブリ言語に変換してくれるツール

例えば数式の演算

$$E = A * B + C * D$$

CPUは「加算命令」や「乗算命令」のような、「2値の演算」の命令しか(一般には)持たない

このためこのような数式は、以下のように複数のアセンブリ言語による命令を組み合わせないで実現できない。

上のような式の構文を解析して、下のようにアセンブリ言語の組み合わせで実現してくれるのがコンパイラ

$$\begin{aligned} E1 &= A * B \\ E2 &= C * D \\ E &= E1 + E2 \end{aligned}$$

構文解析をしてくれるのが、コンパイラ

コンパイラの代表的な機能

- **数式**: 複雑な数式を複数のアセンブリ言語の命令で実現
- プロシージャ(C言語では「**関数**」): 先頭と終端で必要な処理を生成

```
int func(void)
{
  ...
}
```

- **ブロック**: 条件分岐やループをブロック単位で行うような処理を生成

```
if (...) {
  ...
}
```

「コンパイラ」を「自作」してみても

- **再帰**をとにかくよく使う(言語処理系に共通して言えることだが)
※ 組込み技術者は再帰を嫌う:スタック使用量の見積りが必要のため
- (C99準拠とかを気にしないなら)純粋な「ロジック・プログラミング」なので、**頭の中ですべてが完結**する
※ OSやシェルを作るには「仕様」を調べなければならなかったりする
- システムコールをほとんど使わない(入出力とメモリ獲得)ので、実は移植が簡単
- 基本的に一度の実行で終わるので、メモリ解放に神経質にならなくていい(どうせカーネルが解放する)

セキュリティの視点から見るコンパイラ

コンパイラで実現すべきセキュリティ機能は何か

- 命令に対して, 特定の処理を行う
→CPUで実現すべき
- システムコールに対して, 特定の処理を行う
→OSカーネルで実現すべき
- すべての関数の先頭と終端に, 特定の処理を埋め込む
→コンパイラで実現すべき(スタック・プロテクタ)

アプリケーション・プログラマはライブラリで実現しようとするが, コンパイラを知っているとそこに「コンパイラで実現する」という選択肢が新たに生まれる (コンパイラを知っていることで, 問題解決の幅が広がる)

セキュリティの視点から見るコンパイラ

- 複数の階層をまたがって、**協調的に動作**させることで実現できるセキュリティ機能がある
(CPU, OSカーネル, ライブラリなどの各階層)
- コンパイラの技術があれば、そこに「コンパイラ」という階層を入れて考えることができる

各層の連携の例

例えば

「カナリア値を定期的に変えるスタック・プロテクタ」を
「有効に」「適切に」実現するならば

各層の連携の例

- CPUで行うべき対応
 - 乱数の種やアルゴリズムの設定の命令を提供する
 - 乱数値を返す命令を提供する
- OSカーネルで行うべき対応
 - 乱数値を返すサービスを提供する
- **コンパイラで行うべき対応**
 - 各関数の先頭に, CPUの乱数でカナリア値を再生成しスタックに埋め込む処理を入れる
 - 各関数の終端に, スタック上のカナリア値をチェックする処理を入れる
 - チェックエラーの場合に, エラー関数を呼び出す処理を入れる
- ライブラリで行うべき対応
 - プログラムのスタートアップで, カーネルから得た乱数でCPUの乱数の種を設定する
 - カナリア値の保存先(グローバル変数)とエラー関数を用意する

機能は「提供するだけ」では意味が無く、
それを「使う」側も必要(連携が必要)

「CPU」や「OSカーネル」や「コンパイラ」の各層が
単体でなく連携することで可能になる
(もしくは、より有効になる)
セキュリティ対策は多く考えられる

既存のコンパイラは

既存の多くのコンパイラは、高速な実行コードを生成することを主眼として、高度な最適化を行っている

(最適化競争)

このためコード生成ロジックが複雑になり、新たなCPUアーキテクチャに対応させることが困難になっている

Reduced Assembly Set Compiler (RASC) の提案

RASCとは何か

出力するアセンブリのパターンを必要最低限のものに削減することで、アセンブリの出力部を単純化したコンパイラ

定型化した単純なアセンブリ・パターン(アセンブリ・セット)の組み合わせによりアセンブリを生成

アセンブリ・セットは定型化されており、1～数行程度のアセンブリの記述で容易に作成可能

65種類のCPUアセンブリを読み、37種類のCPUでHello Worldを書いた経験と知見から、コンパイラのアセンブリ出力のロジックをいかに共通化できるかを考案

一言で説明すると

「新たなCPUへの対応のしやすさを最優先に考えて設計したコンパイラ」です

1 day-compiler そのための設計

1 day-compilerのための設計

- 演算用レジスタの限定
- 代入処理の限定
- 関数のプロローグ・エピローグの限定
- ビルトインの活用
- 最適化を限定

演算用レジスタの**限定**

演算に用いるレジスタを**限定し定型化**する。

これにより演算処理のアセンブリの生成を定型化・簡略化することが可能になる。

現代的なCPUアーキテクチャの命令セットでは、レジスタは汎用化されており、演算に対して多くの汎用レジスタが等価に利用できる。これは生成されるアセンブリの実行効率には寄与するが、レジスタの扱いが繁雑となりコンパイラのアセンブリ出力ロジックを複雑なものとする。

代入処理の**限定**

値の代入は**レジスタ・ベースに限定**する。

一部のCPUアーキテクチャでは、メモリからメモリへの値の代入が可能になっている。しかし現代的なRISCアーキテクチャではロードストア・アーキテクチャにより必ずレジスタを経由する設計となっており、コンパイラもそれに準じた設計とすることで、値の代入処理をアーキテクチャ共通にする。

関数のプロローグ・エピローグの**限定**

関数のプロローグとエピローグの処理を**限定**することで、必要とするスタックフレームのサイズ計算を簡略化し、関数内部の処理への依存を不要にする。これにより関数のプロローグ・エピローグの処理を定型化する。

関数のプロローグ(先頭の処理)とエピローグ(末尾の処理)は、スタックフレームの調整処理が行われるため関数内部でのレジスタの利用に左右される。とくに不揮発性のレジスタにおいてはプロローグでの保存とエピローグでの復旧が必要となるため、その利用数はスタックフレームのサイズに影響する。よって関数内部の処理で利用される不揮発性レジスタの見積もりが必要になり、プロローグ・エピローグの生成処理が煩雑になる。

ビルトインの活用

CPUが持つ命令には、ソフトウェア処理により代替可能な処理が多くある。

それらの処理はソフトウェアによる代替処理の**ビルトイン**を持たせることで、CPUへの対応時に作成が必要なアセンブリ・セットの必要数を削減する。

例えばビット反転は、 -1 との排他的論理和により代替できる。また符号拡張は、ビット判定と加算により代替できる。他にも掛け算命令は、加算命令をループで繰り返すことで同等の処理が可能であり、割算も同様である。

最適化を限定

共通的に行える最適化に限定する。

RASCではアセンブリ・セットによるアセンブリ・パターンの固定化により、生成されるアセンブリの全体サイズは増加してしまうが、これは最適化によりある程度は軽減できる。

コンパイラによる最適化は、アーキテクチャ共通で行えるものと、アーキテクチャ固有のものに分類される。アーキテクチャ固有の最適化はそのアーキテクチャの高速化のための専用命令を使うことなどで高速化に大きく寄与するが、CPUアーキテクチャの特有の処理が増加する原因ともなるため廃止する。

NLCCによる実装

NLCC (No Look C Compiler)



- **NLUXプロジェクト**のコンパイラ(<https://kozoz.jp/nlux/>)
- セルフホストが可能(NLCCでNLCCをコンパイルできる)
- nllibc(NLUXの標準Cライブラリ)と連携し, NLUXに完全に閉じてのビルドが可能
- Debian/GNU LinuxやFreeBSDの環境で, システムのヘッダファイルをインクルードしてのコンパイルが可能
- 構造体・共用体・可変長引数・ビットフィールド・構造体の引数渡しに対応

RISC設計により, 各種CPUアーキテクチャへの対応(クロスコンパイラの作成)が非常に低いコストで可能

デモ

1. gccでnlccをビルドする
2. そのnlccでnllibcをビルドする
3. そのnllibcとnlccでnlccをビルドする
4. そのnlccでテストプログラムをすべて通す

NLCCのRASC設計 の実際

演算用レジスタを2個に**限定**

演算に用いるレジスタは2個(R0/R1)に**限定**している。

演算処理のアセンブリ・パターンはR0/R1に**限定**して記述される。これにより記述するアセンブリを**固定化**し、アセンブリ・セットとして大幅に簡略化できている。

代入処理をレジスタ・ベースに**限定**

値の代入処理はレジスタ・ベースに**限定**している。

このため多くのCPUアーキテクチャにおいてアセンブリ・セットの共通化が可能になっている。

レジスタをR0/R1に**限定**し, さらに代入処理をレジスタベースに**限定**することで, アセンブリ・セットを64種類に**限定**できている。

アセンブリ・セットの例

```
static void get_value(FILE *out, long value)
{
    ASM_CODE_OUT(out, "%tmov%t$0x%lx, %%eax%n", value);
}

static void get_value_r1(FILE *out, long value)
{
    ASM_CODE_OUT(out, "%tmov%t$0x%lx, %%edx%n", value);
}

static void get_address_stack(FILE *out, int offset)
{
    ASM_CODE_OUT(out, "%tlea%t0x%x(%%esp), %%eax%n", offset);
}

static int get_address_stack_r1(FILE *out, int offset)
{
    ASM_CODE_OUT(out, "%tlea%t0x%x(%%esp), %%edx%n", offset);
    return 0;
}
```

関数のプロローグ・エピローグの**固定化**

スタックフレームの各種サイズを固定値とすることで、スタックフレームの構造を**固定**にする。

これにより関数のプロローグ・エピローグの処理が**定型化**され、関数の内部の処理に応じた調整が不要となっている。

不揮発性のレジスタなど、各種レジスタ数などはスタックフレームのサイズに影響するが、これらのパラメータを固定値とすることで(処理やスタックフレームのサイズは無駄にはなるが)スタックフレームのサイズを固定とし、関数のプロローグ・エピローグ処理のアセンブリ・セットとしての**定型化**を可能としている。

関数のプロローグ・エピローグの例

```
static void function_start(FILE *out)
{
    ASM_CODE_OUT(out, "¥tpush¥t%%ebp¥n");
    ASM_CODE_OUT(out, "¥tmov¥t%%esp, %%ebp¥n");
}
```

```
static void function_end(FILE *out)
{
    ASM_CODE_OUT(out, "¥tleave¥n");
    ASM_CODE_OUT(out, "¥tret¥n¥n");
}
```

```
static void function_register_save(FILE *out)
{
    ASM_CODE_OUT(out, "¥tpush¥t%%edi¥n");
    ASM_CODE_OUT(out, "¥tpush¥t%%esi¥n");
    ASM_CODE_OUT(out, "¥tpush¥t%%ebx¥n");
}
```

設定パラメーター一覧

名称	意味
word_size	整数値のバイトサイズ
pointer_size	ポインタ値のバイトサイズ
funcall_args_reg_number	関数の引数として用いるレジスタの個数
funcall_args_stack_number	関数の引数として用いるスタックのフィールド数
tmp_reg_number	一時利用するレジスタの個数
tmp_stack_number	一時利用するスタックのフィールド数
function_register_number	関数のプロローグで保存する, 不揮発性(callee-saved)のレジスタの個数
function_saveparam_number	関数のプロローグで保存する, 各種の値(戻り先アドレスなど)のフィールド数
stack_align_size	スタックのアラインサイズ
stack_correct_size	スタックの調整サイズ

ビルトイン

ビルトインにより一部のアセンブリ・セットは記述が不要となる。

記述が不要なアセンブリ・セットの個数は**28**である。ビルトインを活用することで、動作に必要なアセンブリ・セットを最少で**36**個に削減できる。

ビルトインはソフトウェア処理による代替であるため、生成される実行コードのサイズと速度は一般に悪化する。しかしビルトインにより、一部のアセンブリ・セットを作成しなくとも動作・検証が可能となるため、CPUアーキテクチャ対応の初期段階での一時対応では有用である。

ビルトインの例

```
”__nlcc_attr_funcnoinit /* used in the processing of assembly output */¥n”
```

```
”static long __builtin_inv(long val)¥n”
```

```
”{¥n”
```

```
”    return val ^ -1L;¥n”
```

```
”}¥n”
```

```
”¥n”
```

```
”__nlcc_attr_funcnoinit /* used in the processing of assembly output */¥n”
```

```
”static long __builtin_minus(long val)¥n”
```

```
”{¥n”
```

```
”    return 0L - val;¥n”
```

```
”}¥n”
```

```
”¥n”
```

```
”__nlcc_attr_funcnoinit /* used in the processing of assembly output */¥n”
```

```
”static unsigned long __builtin_xor_ulong(unsigned long val, unsigned long xor, int bits)¥n”
```

```
”{¥n”
```

```
”    return (val | xor) - (val & xor);¥n”
```

```
”}¥n”
```

ビルトイン一覧

名称	意味	内容
BUILTIN_EXTENSION	符号拡張	最上位ビットを参照し符号拡張ぶんのビットを加算する
BUILTIN_INV	ビット反転	オール1の値との排他的論理和
BUILTIN_MINUS	符号反転	0からの減算
BUILTIN_MUL	かけ算	ループによる加算
BUILTIN_DIV	除算	ループによる減算
BUILTIN_MOD	剰余算	ループによる減算時の余り
BUILTIN_LSHIFT	左シフト	配列化したマスク値による生成
BUILTIN_RSHIFT	右シフト	配列化したマスク値による生成
BUILTIN_BRANCH	分岐(一部)	非ゼロの判定を, ゼロ判定により行う
BUILTIN_CMP	比較(一部)	比較の判定を, 他の比較命令により行う
BUILTIN_CALL	関数呼び出し	関数呼び出しを関数へのポインタにより行う
BUILTIN_R1	R1に対する処理	R1によるアセンブリ・パターンを利用しない

最適化

最適化はアセンブリ・セットをいったんキューイングし、不要なロジックを削除することで行う。これによりCPUアーキテクチャに依存しない共通処理で最適化し、最適化がCPUアーキテクチャの対応に影響しないようにしてある。

NLCCでは最適化をすることで機械語コードサイズを約52%に削減できた。またNLCCでは最適化時には `gcc -O1` に対して約6倍のコードサイズ増加となっていた。

なお狭い範囲での最適化でも、複数を組み合わせることで連鎖的に最適化がされ、実行コードの効果的なサイズ削減がされていた。

最適化の効果

コンパイル方法	サイズ(バイト)	gcc -O1 に対する割合(%)
gcc -O0	19966	137
gcc -O1	14546	100
gcc -Os	12498	85
nlcc (最適化無)	162917	1120
nlcc (最適化有)	85553	588

※amd64における, gcc-13.3.0とNLCCでの機械語コードのサイズ比較
※サンプル・プログラムにはnlccのsyntax.cを利用

多種アーキテクチャへの対応

新たなCPUアーキテクチャへの対応は、アセンブリ・セットを記述することで可能となる。

実際に**12種類**のCPUアーキテクチャに対応させたところ、1アーキテクチャあたりの対応は概ね、基本対応が2時間、デバッグが6時間、合計**8時間程度**で可能であった。

また対応行数の平均値は115行であり、概ね**100行程度の対応**で新たなアーキテクチャに対応可能になっている。

CPUアーキテクチャの対応状況

アーキテクチャ名	ファイル行数	対応行数	対応状況
x86	539	110	セルフビルド・全テストを確認完了
amd64	548	119	セルフビルド・全テストを確認完了
ARM	532	103	エミュレータ環境での動作を確認
MIPS	541	112	エミュレータ環境での動作を確認
PowerPC	524	95	エミュレータ環境での動作を確認
AArch64	556	127	エミュレータ環境での動作を確認
Thumb	619	190	エミュレータ環境での動作を確認
Thumb2	583	154	エミュレータ環境での動作を確認
MIPS16	564	135	エミュレータ環境での動作を確認
OSECPU	494	65	実験的対応
RISC-V	521	92	エミュレータ環境での動作を確認
RX	517	88	エミュレータ環境での動作を確認
テンプレート	429	0	
平均	544	115	

NLCCCによる
セキュリティ

NLCCによるセキュリティ

RASCはアセンブリ出力が簡易なため、アセンブリ・レベルでの機能を追加しやすい

ここに追加すれば、対応しているすべてのCPUアーキテクチャで有効になる

当り前のことではあるのだが、これこそがコンパイラでセキュリティ対策を行うことの大きなアドバンテージではある

簡易スタック・プロテクタの例

- 関数の先頭で, スタックに積まれた戻り先アドレスを暗号化しておく
(戻り先アドレスの上書き防止)
- 関数の末尾で, 復号化してからリターンする

デモ

ライブコーディング: 簡易スタック・プロテクタ

(検証)

NLCCは、本当に
CPU対応させやすい
のか

デモ

ライブコーディング: V850対応

RASCとは何か

出力するアセンブリのパターンを必要最低限のものに削減することで、アセンブリの出力部を単純化したコンパイラ

NLCCとは何か

RASCの実装であるコンパイラ. 12種類のCPUアーキテクチャに対応.



付録

アセンブリ・セット一覧(1/7)

名称	省略の可否	処理	内容
get_value	不可	R0 = val	定数値をR0に代入
get_value_r1	不可	R1 = val	定数値をR1に代入
get_address_stack	不可	R0 = SP + offset	スタックポインタをR0に代入
get_address_stack_r1	可	R1 = SP + offset	スタックポインタをR1に代入
get_address	不可	R0 = &label	ラベルのアドレスをR0に代入
add_address	不可	R0 += offset	R0をアドレス値として加算
get_r1	不可	R0 = R1	R1をR0に代入
set_r1	不可	R1 = R0	R0をR1に代入

アセンブリ・セット一覧(2/7)

名称	省略の可否	処理	内容
memory_load	不可	$R0 = *(R1 + \text{offset})$	メモリからR0にロード
memory_store	不可	$*(R1 + \text{offset}) = R0$	メモリにR0からストア
stack_load	不可	$R0 = *(SP + \text{offset})$	スタックからR0にロード
stack_store	不可	$*(SP + \text{offset}) = R0$	スタックにR0からストア
stack_load_r1	不可	$R1 = *(SP + \text{offset})$	スタックからR1にロード
stack_store_r1	不可	$*(SP + \text{offset}) = R1$	スタックにR1からストア
stack_expand	不可	$SP -= \text{size}$	スタックの獲得
stack_reduce	不可	$SP += \text{size}$	スタックの解放

アセンブリ・セット一覧(3/7)

名称	省略の可否	処理	内容
funcall_reg_load	不可	R0 = arg	関数の引数をR0にコピー
funcall_reg_store	不可	arg = R0	関数の引数にR0をコピー
tmp_reg_load	不可	R0 = tmp	テンポラリレジスタをR0にコピー
tmp_reg_save	不可	tmp = R0	テンポラリレジスタにR0をコピー
tmp_reg_load_r1	不可	R1 = tmp	テンポラリレジスタをR1にコピー
tmp_reg_save_r1	不可	tmp = R1	テンポラリレジスタにR1をコピー

アセンブリ・セット一覧(4/7)

名称	省略の可否	処理	内容
sign_extension_char	可	R0 = (char)R0	符号拡張(signed char)
sign_extension_uchar	可	R0 = (unsigned char)R0	符号拡張(unsigned char)
sign_extension_short	可	R0 = (short)R0	符号拡張(signed short)
sign_extension_ushort	可	R0 = (unsigned short)R0	符号拡張(unsigned short)
sign_extension_int	可	R0 = (int)R0	符号拡張(signed int)
sign_extension_uint	可	R0 = (unsigned int)R0	符号拡張(unsigned int)

アセンブリ・セット一覧(5/7)

名称	省略の可否	処理	内容
calc_inv	可	$R0 = R0 \wedge -1$	ビット反転(R0)
calc_minus	可	$R0 = -R0$	符号反転
calc_op1	可		単項演算子一般
calc_add	不可	$R0 = R0 + R1$	加算
calc_sub	不可	$R0 = R0 - R1$	減算
calc_and	不可	$R0 = R0 \& R1$	論理積
calc_or	不可	$R0 = R0 R1$	論理和
calc_xor	可	$R0 = R0 \wedge R1$	排他的論理和
calc_mul	可	$R0 = R0 * R1$	掛け算
calc_div	可	$R0 = R0 / R1$	割り算
calc_mod	可	$R0 = R0 \% R1$	剰余算
calc_llshift	可	$R0 = R0 \ll R1$	左シフト
calc_rashift	可	$R0 = R0 \gg R1$	算術右シフト
calc_rlshift	可	$R0 = R0 \gg R1$	論理右シフト
calc_op2	可		2項演算子一般

アセンブリ・セット一覧(6/7)

名称	省略の可否	処理	内容
branch	不可	goto label	無条件分岐
branch_zero	不可	if (!R0) goto label	R0がゼロの場合に分岐
branch_nzero	可	if (R0) goto label	R0が非ゼロの場合に分岐
branch_cmp_eq	不可	if (R0 == R1) goto label	分岐(equal)
branch_cmp_ne	可	if (R0 != R1) goto label	分岐(not equal)
branch_cmp_lt	不可	if (R0 < R1) goto label	分岐(less than)
branch_cmp_gt	可	if (R0 > R1) goto label	分岐(greater than)
branch_cmp_le	可	if (R0 <= R1) goto label	分岐(less equal)
branch_cmp_ge	可	if (R0 >= R1) goto label	分岐(greater equal)
branch_cmp_ult	不可	if (R0 < R1) goto label	分岐(unsigned less than)
branch_cmp_ugt	可	if (R0 > R1) goto label	分岐(unsigned greater than)
branch_cmp_ule	可	if (R0 <= R1) goto label	分岐(unsigned less equal)
branch_cmp_uge	可	if (R0 >= R1) goto label	分岐(unsigned greater equal)
branch_cmp	可		分岐一般

アセンブリ・セット一覧(7/7)

名称	省略の可否	処理	内容
function_call	可	function()	関数呼び出し
function_call_set	不可	fp = R0	関数呼び出しのポインタをR0に設定
function_call_pointer	不可	(*fp)()	ポインタ経由での関数呼び出し
function_start	不可		関数の先頭での処理
function_end	不可		関数の末尾での処理
function_register_save	不可		関数の先頭でのレジスタ保存
function_register_load	不可		関数の末尾でのレジスタ復旧