2015
春号

百萬石
Hyakumangoku Vol.43

Published by

部長挨拶

f_taka6

電気通信大学 MMA の部誌『百萬石』をお手に取って頂き、ありがとうございます。

MMA では百萬石を、新歓時期 (春号) と本学の学園祭の調布祭時期 (秋号) の年 2 回発行しており、部員有志が MMA の活動内容に関係することやしないことを記事に書き纏めています。また、この春号は、新入生の方々に対して「MMA はこのような感じのサークルです」と紹介する役割も兼ねています。

MMA については、電気通信大学内でも「名前を聞いても何をやっているのか分からない」「あるのそんなサークル？」「総合格闘技[*1]関連のサークル？」なとど言われることもありますが、この名称は "Microcomputer Making Association (マイクロコンピュータを作る会)" の頭文字に由来し、現在は「計算機に関わることならなんでもやってみよう」というスタンスで活動している、1975 年度に結成された技術系の大学公認サークルです。今年度で結成 40 周年を迎えます。

昨年度は、部員が個々に行ったものの他に、MMA として以下のようなことなどが行われました。

- CTF[*2]大会やプログラミングコンテストなどへの部員有志の参加
- MMA のサーバやネットワーク、wiki(MoinMoin)、部室電子錠 "kagisys" 等の管理運営
- サークル会館のネットワークの委託を受けての管理
- MMA 主催の LT[*3]大会 Dentoo.LT の部員有志による運営
- 講習会・ワークショップの部員有志による開催
- 技術系サークル合同新入生歓迎会を幹事サークルとして他の技術系サークル[*4]とともに開催
- 調布祭でのジャンク品の販売
- 部誌『百萬石』の発行
- 合宿

昨年度は部員有志が、内閣の IT 総合戦略本部や情報処理推進機構 (IPA)、情報通信研究機構 (NICT) などが後援する CTF 大会である SECCON 2014 の全国大会・決勝戦に進出し、24 チーム (日本国外からの 11 チーム含む) 中 6 位となり健闘賞 (NEC 賞) を受賞することができました。

Dentoo.LT については、昨年度は 4 回開催し、その内の Dentoo.LT #7.5 は電気通信大学の 7 月のオープンキャンパスにおいて、Dentoo.LT #8.5 は調布祭において、開催させていただきました。

また、部誌・百萬石のバックナンバーを次の URL にて公開しております。是非ご覧ください。

`https://wiki.mma.club.uec.ac.jp/Booklet`

最後に、部室がサークル会館 2 階奥に在るので、興味を持って頂けた方は気兼ねなくお訪ねください。

2015 年 4 月吉日

[*1] 英語で "Mixed Martial Arts" 略称が "MMA"

[*2] Capture The Flag 情報セキュリティの分野での CTF は 情報セキュリティ技術についての競うもの

[*3] Lightning Talks 短い時間でのプレゼンテーションのこと 制限時間を 5 分とすることが多い Dentoo.LT では 10 分

[*4] U.E.C.wings 無線部 工学研究部 TeRes X680x0 同好会

目次

第 1 章

冴えない Emacs パッケージの育てかた kakakaya

概要
「自作の謎ライブラリを公的な場所で配布したい」という欲求に駆られて Emacs のパッケージを作成して MELPA リポジトリに登録される所までやったので、感想を述べたり、今後パッケージ作りたい人のために知見を共有したりします。
Emacs Lisp の解説は主題ではないので、具体的なソースコードは掲載しておりません。気になる方は成果物の節の github のページに公開されているのを参照してください。

1.1 導入

聡明なる読者の皆様方におかれましては、当然日々手足の如く Emacs を愛用しているものと思われますので、基本的な説明は割愛し、起動時ロゴ関連と package.el 関連の説明のみとさせて頂きます。

1.1.1 Emacs の起動時ロゴについて

通常 (何も設定していない場合)、以下のようなロゴが出ます。

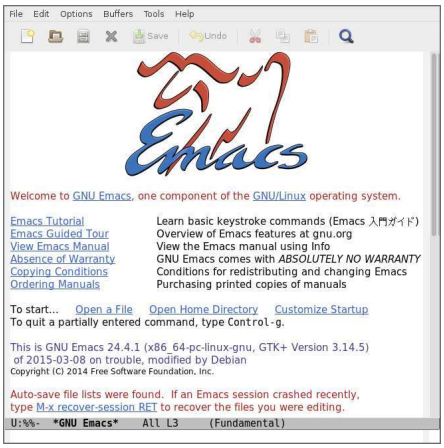


図 1.1 設定を読み込まないで起動した時

味気ない画像で、表示する理由も特に無いので当初は非表示にしていましたが、変数 `fancy-splash-image`

に画像のパスを指定するとその画像が表示されるというのを知り、適当な画像を表示して遊んでいた……ものの、やはり毎回同じ画像が表示されるのも少々虚しいという感じでした。

1.1.2　Emacs のパッケージ管理システムについて

バージョン 24 以降[*1]の Emacs には package.el と呼ばれるパッケージ管理システムが標準で付属しており、各種リポジトリからパッケージを簡単にインストールしたり、更新できるようになりました (今までは Emacs Wiki や github を参照する必要がありました)。また、パッケージを開発して公開する手段としても、更新の処理などの観点から格段に便利になっています。

1.1.3　動機の形成

これらの理由により、「ランダムに起動時のロゴを設定しするパッケージを作って公開すれば楽しいのではないか」という考えを持ち、実行に至りました。

1.2　開発の流れ

初期設定状態の Emacs では*scratch*というバッファが起動時に開くようになっており、ここに Emacs Lisp の式を記述、C-x C-e(eval-last-sexp、直前の S 式を評価する) などを使って Emacs で書いた式をそのまま Emacs に適用して動作を調べる、というのが基本的な開発になります。しかし、「任意のパスからファイルのリストを取得、その中からランダムに選択して起動時のロゴを表示する」という基本的な部分は問題無く書けたものの、外部ファイルとして保存して init.el から読み込ませた所、いくつかの部分でハマってしまいました。
一応、GNU がドキュメント公開していたり[*2]、EmacsWiki で解説記事が載っていたりする[*3] ものの、日本語の解説記事が少ないという印象があり、ググっても解決に役立つ有益な情報が見つけられなかったりしました。

1.2.1　詰まった場所

1. ユーザに「ランダムに表示したい画像を入れるディレクトリ」を指定させたいが、「ユーザが定義する値」というのはどのように記述すれば良いのか？
2. パッケージを `(require 'random-splash-image)`して読み込んだが、`M-x(execute-extended-command)`から実行できなかった。S 式として評価することはできるので読み込みに失敗している訳ではないらしい。なぜか？
3. 配布されているパッケージには、必ず-autoload.el と-pkg.el という機械的に生成されたらしいファイルが付属しているが、これらはどのように作成しているのか？

[*1] バージョン 23 でも package.el に対応はしている
[*2] `https://www.gnu.org/software/emacs/manual/html_node/eintr`
[*3] `http://www.emacswiki.org/emacs/EmacsLisp`

1.2.2　解決

色々探したものの、結局良い情報が見つからなかったので、他のパッケージのソースを読んだりして解決しました。

1. `defcustom`という関数を使い、デフォルト値や変数の説明と共に変数を定義する。
2. `defun`の後に`(interactive)`を挟んだら参照できるようになり、実行できた。また、`(interactive)`ではミニバッファで数値入力などさせることも可能とのこと。[2]
3. 無くても動くので無いままリポジトリに取り込んでもらったら勝手に生成されていた。そういうものらしい。

参考の為にも Emacs Lisp の本が欲しくなったのですが、今後も Emacs Lisp を書き続ける訳ではない[*4]のでやめておきます。

1.3　そして公開へ……

package.el の標準リポジトリは ELPA(`http://elpa.gnu.org`) ですが、代替リポジトリとして MELPA(`http://melpa.org`) と marmalade(`https://marmalade-repo.org`) の 2 つが有名です。ELPA は登録までの時間が長いらしいこと、marmalade は登録しようとしたらユーザ登録が必要と言われたが登録ページで「you can't register for marmalade-repo」とか言われた[*5]ので、とりあえず MELPA だけに公開することにしました。

1.3.1　MELPA へのパッケージ登録

MELPA へのパッケージ登録は以下のような流れで行いました。[1]

1. MELPA のリポジトリ[*6]をフォークする
2. パッケージの名前や説明、取得する方法などを recipe に記述する
3. recipe を利用して make を実行し、意図通りに動いているか確認を行う
4. コミットして元のリポジトリに PR[*7]を送るとメンテナの方が取り込んでくれます。この時、プログラムの修正や、既存のパッケージと似ていたりする場合に教えてもらえたりします。

最後の部分にて、`defcustom`の使い方などについての修正を頂きました。

[*4] とか考えていましたが、この記事の初稿を書いた 10 日後にまた「ニコニコ動画におけるごちうさの再生数を取得する」という Emacs Lisp を書いていた (しかも URL 関連の処理や interactive 関連で再び詰まってしまった) ので、そのうち購入しようと思います

[*5] 泣いた

[*6] `https://github.com/milkypostman/melpa`

[*7] Pull Request

そして……

> @melpa_emacs
> random-splash-image (20150313.859) — Randomly sets splash image to *GNU Emacs* buffer on startup. http://dlvr.it/8y11Rp
> https://twitter.com/melpa_emacs/status/576448290700468224

無事公開されました!
最初に PR を飛ばしたのが 0 時 48 分頃、メンテナの方とのやりとりを終えマージされたのが 1 時 12 分頃で、それからリポジトリの更新は数回あったものの、なかなか反映されずドキドキしていたのですが、3 時 22 分になってようやく無事公開されるという流れと相成りました。
サクっと作って PR 飛ばせば 1 時間半で自分の作ったものが有名な所で公開される、というのはなかなか面白い経験でした。

1.4　成果物

github のページ:
https://github.com/kakakaya/random-splash-image
公開されたリポジトリのページ:
http://melpa.org/#/random-splash-image

1.5　インストール

この記事の主旨通り、package.el を利用して MELPA から取得してインストールすることも、もちろん直接 github[*8]から取得することも可能です。

[*8] https://github.com/kakakaya/random-splash-image

1.6　動作スクリーンショット

以下のように、設定を変更せずに、起動する度にランダムに表示される画像が変わります。

図 1.2　ランダムに画像が表示される様子

1.7　まとめ

以上、知見でした。
この記事を読んだ皆さんは Emacs のパッケージを公開するための手順を完全に理解したも同然なので、何か面白いパッケージとか作成して適当なリポジトリに公開してみるとよいのではと思います。

参考文献

[1] http://d.hatena.ne.jp/syohex/20121021/1350823391
[2] http://mukaer.com/archives/2012/03/24/emacsinteractiv

第 2 章

逆アセンブルを読んでみる (gcc 版)

hiro1357

概要
＊＊この記事は 2013 年の夏号 (夏コミで頒布) に掲載したものですが、今年度の春号を手にする皆さんにも読んでもらいたいと思い、再掲しました！＊＊
最近、プロコン[*1]と並んで、CTF[*2]という競技が人気ですね！[*3]
CTF というのは、セキュリティの知識を競う競技です。サーバーへの侵入・保護、パケット解析、バイナリ解析、暗号、トリビアなど、幅広い問題が出題され、それを解くことで点数を稼ぎます。
問題に対してある程度の分類はされていますが、横断的な知識を必要とする複合問題も多数出題されます。
もちろん、この記事で取り扱うようなリバースエンジニアリングの知識[*4]も例外なく必要とされます。
そこで、とりあえず逆アセンブルの読み方を自分用のメモも兼ねてまとめてみることにしました。

2.1　前提知識

逆アセンブルを読むには、やはりある程度のアセンブリ言語に関する知識と CPU に関する知識、それから少しだけ OS に関する知識[*5]が必要になってきます。

2.1.1　アセンブリ言語の文法

CPU が解釈する命令は電気的な ON-OFF の組み合わせで成り立っているため、人間が直接理解しづらいものになっています。それらの命令に人間が分かる名前をつけることで、人間にも理解できるようにしたものが、アセンブリ言語です[*6]。そのため、アセンブリ言語は CPU が解釈する命令とほぼ 1 対 1 で対応しています。

一般的なアセンブリ言語では、次のような文法になっています

[*1] プログラミングコンテストのこと
[*2] Capture-The-Flag
[*3] 無理やりそういうことにしてみる
[*4] の一部
[*5] 呼び出し規約など
[*6] ニーモニックとも呼ばれます

ソースコード 2.1　一般的なアセンブリ言語の文法

```
ラベル:
        命令　演算対象1，演算対象2
        命令　演算対象1，演算対象2
        命令　演算対象1
        命令
```

命令は CPU によって上から順番に解釈されます。ラベルは命令の存在するメモリ番地を表しており、実行位置の移動をする際に目印として使われます。

命令には、いくつかの演算対象が与えられます。演算結果を返す必要がある場合には、演算対象または、命令内に指定されている領域に上書きされます。

演算対象には、次に説明するレジスタ、数値、メモリ番地などが与えられます。

一般的に、命令のことを「オペコード」、演算対象のことを「オペランド」と呼び、特に、計算に使用されるオペランドを「ソースオペランド」、計算結果が転送されるオペランドを「デスティネーションオペランド」と呼びます。

説明から想像がつくと思いますが、アセンブリ言語は CPU の種類の数だけ存在します。ですから、一般的な PC に採用されている x86, x86_64 系のアセンブリ言語と、スマートフォンやゲーム機などに採用されている ARM 系、SH 系、MIPS 系などのアセンブリ言語ではそれぞれ文法が異なります。

今回は、最近の一般的な PC について学んでいこうと思うので、x86_64 の文法を見てみましょう。

x86_64 のアセンブリ言語には主に Intel 形式と AT&T 形式の 2 種類の文法があります。

ソースコード 2.2　Intel 形式

```
mov rax, 1
mov rbx, qword ptr [rbp-0x8]
add rax, rbx
```

Intel 形式では、演算結果が第一オペランドに転送されます。上の例では、ADD という命令の結果は rax に転送されることになります。

また、メモリの位置を示すポインタは、上の例のように byte, word, dword, qword といったデータを扱う単位を表す修飾語をつけて与えます。

ソースコード 2.3　AT&T 形式

```
mov 1, %rax
movq -0x8(%rbp), %rbx
add %rbx, %rax
```

AT&T 形式では、演算結果が第二オペランドに転送されます。上の例では、ADD という命令の結果は rax に転送されることになります。

また、メモリの位置を示すポインタは、上の例のように与えますが、データを扱う単位については、命令を修飾することで与えます。

ソースコード 2.4　mov の例

```
mov = move （オペランドから自動的に判断できる場合はmovが使われます）
movb = move byte
movw = move word
movl = move long word
movq = move quad word
```

今回は、筆者の趣味から、Intel 形式で話を進めていきたいと思います。

2.1.2　レジスタ

アセンブリ言語には、変数というものがありません。というのは、CPU だけでは、変数のような仕組みを作ることが困難だからです。

CPU はハードウェアなので、ソフトウェアほどの柔軟性がありません。変数のように、一時記憶のための領域を必要に応じて作ったり消したりといったことができないのです。[*7]

その代わりに、CPU には固定の一時記憶領域があります。これを、「レジスタ」と言います。

今回逆アセンブルしてみる C 言語を始め、その他の高級なプログラミング言語では、レジスタとメモリをうまく組み合わせて変数の仕組みを作っています。

今回扱う x86_64 のレジスタでよく使われるものを一部紹介します。

x86_64 の主なレジスタ (汎用レジスタ)

Register Type	**Without REX**	**With REX**
Byte Registers	AL, BL, CL, DL, AH, BH, CH, DH	AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8L - R15L
Word Registers	AX, BX, CX, DX, DI, SI, BP, SP	AX, BX, CX, DX, DI, SI, BP, SP, R8W - R15W
Doubleword Registers	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D
Quadword Registers	N.A.	RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8 - R15

(REX とは、64 ビット動作時の命令の拡張のことです。)

Intel 64 and IA-32 Architectures Software Developer's Manual (69 ページ) より引用

http://download.intel.com/products/processor/manual/325462.pdf

CPU には表にあるように、扱うサイズごとに、名前をつけられた一時記憶領域があります。これらの領域は変数のように増やすことができないため、上書きして使いまわされます。

64 ビットレジスタ R**の下位 32 ビットが 32 ビットレジスタ E**、32 ビットレジスタ E**の下位 16 ビットが 16 ビットレジスタ**、16 ビットレジスタ**の上位 8 ビット・下位 8 ビットがそれぞれ 8 ビットレジスタ*H・*L となるようになっています。

[*7] FPGA のように自己書き換え可能なハードウェアであれば、多少違ってくるとは思いますが

rax の例

<table>
<tr><td colspan="4">64bit レジスタ rax</td></tr>
<tr><td></td><td colspan="3">32bit レジスタ eax</td></tr>
<tr><td colspan="2"></td><td colspan="2">16bit レジスタ ax</td></tr>
<tr><td colspan="2"></td><td>8bit ah</td><td>8bit al</td></tr>
</table>

また、表から分かるとおり、x86_64 では、32 ビット動作時と、64 ビット動作時で扱えるレジスタが変わります。違いとしては、64 ビット動作時には 64 ビットレジスタ R**が使えるようになること、それから、16 ビットレジスタの上位 8 ビットを単体のレジスタとして扱えなくなることです。

x86_64 の主なレジスタ (フラグレジスタの一部)

CF	**Carry flag** — Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic.
PF	**Parity flag** — Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.
AF	**Adjust flag** — Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic.
ZF	**Zero flag** — Set if the result is zero; cleared otherwise.
SF	**Sign flag** — Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)
OF	**Overflow flag** — Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two’s complement) arithmetic.

(ステータスレジスタとも呼ばれます)

Intel 64 and IA-32 Architectures Software Developer’s Manual (72 ページ) より引用

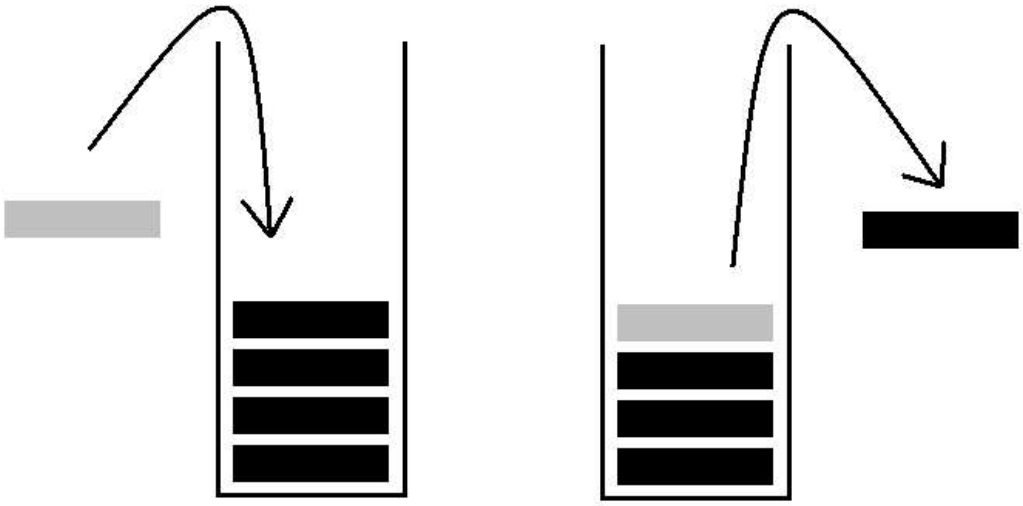
http://download.intel.com/products/processor/manual/325462.pdf

フラグレジスタは、演算結果の状態を表すレジスタです。主に条件付ジャンプ命令の条件などに使われています。

2.1.3　スタック

CPU には、スタックと呼ばれる簡単なデータ構造を扱う機構がついています。

スタック構造のイメージ



スタックは、物を積み上げたような構造をしています。最初に積んだデータは、その後に積んだデータをすべて取り出さなければ、取り出すことができません。

スタックにデータを積むことを push、スタックからデータを取り出すことを pop といいます。

CPU がハードウェア的に扱えるスタックの数はひとつです。複数の処理でひとつのスタックを共有しているため、スタックを pop しすぎたり、push したデータを pop せずに次の処理を実行すると、問題になることがあります。

2.1.4　命令セット

CPU には、種類にもよりますが、足し算や引き算をするだけの単純なものから、CPU がもつ高度な機能を扱うための命令まで、さまざまな命令が用意されています。これらの命令の集まりを「命令セット」と呼びます。もちろん、命令セットも CPU によって異なります。

今回扱う x86_64 の命令セットでよく使われるものを一部紹介します。

アセンブリ言語の主な命令 (x86_64)

命令	例	説明
転送命令	mov rax, rbx	rbx の値を rax に転送します。ソースオペランドには数値も指定可。
交換命令	xchg rax, rbx	rax の値と rbx の値を交換します。
AND 命令	and rax, rbx	rax と rbx の AND 演算を行い、結果を rax に転送します。
OR 命令	or rax, rbx	rax と rbx の OR 演算を行い、結果を rax に転送します。
XOR 命令	xor rax, rbx	rax と rbx の XOR 演算を行い、結果を rax に転送します。
NOT 命令	not rax	rax の NOT 演算を行い、結果を rax に転送します。
NEG 命令	neg rax	rax の符号 (プラスかマイナスか) を反転します。
TEST 命令	test rax, rbx	AND 演算を行い、符合のみを変化させます。(演算結果は捨てる)
加算命令	add rax, rbx	rax と rbx の足し算を行い、結果を rax に転送します。演算結果に応じて CF、OF、SF がセットします。
減算命令	sub rax, rbx	rax から rbx を引き算し、結果を rax に転送します。演算結果に応じて CF、OF、SF がセットします。
INC 命令	inc rax	rax の値をインクリメント (1 加算) します。
DEC 命令	dec rax	rax の値をデクリメント (1 減算) します。
乗算命令	mul rbx	rax と rbx を乗算し、結果の上位 64 ビットを rbx、下位 64 ビットを rax に転送します。
符号付乗算命令	imul rbx	符号付の乗算を行います。
除算命令	div rbx	上位 64 ビットを rdx 下位 64 ビットを rax とした 64 ビットのデータを、rbx で除算して商を rax、余りを rdx へ転送します。
符号付除算命令	idiv rbx	符号付の除算を行います。
比較命令	cmp rax, rbx	rax から rbx を引き算し、減算命令と同様に演算結果に応じて CF、OF、SF をセットしますが、演算結果は転送されません。
ジャンプ命令	jmp LABEL	指定したラベルの位置へジャンプします。
JA 命令	ja LABEL	CF と ZF0 のときにラベルの位置へジャンプします。
JNA 命令	jna LABEL	CF か ZF が 1 のときにラベルの位置へジャンプします。
JC 命令	jc LABEL	CF が 1 のときにラベルの位置へジャンプします。
JNC 命令	jnc LABEL	CF が 0 のときにラベルの位置へジャンプします。
JZ 命令	jz LABEL	ZF が 1 のときにラベルの位置へジャンプします。
JNZ 命令	jnz LABEL	ZF が 0 のときにラベルの位置へジャンプします。
JCXZ 命令	jcxz LABEL	rcx が 0 のときにラベルの位置へジャンプします。

命令	例	説明
ループ命令	loop LABEL	rcx をデクリメントし、rcx が 0 でないならラベルの位置へジャンプします。
LOOPZ 命令	loopz LABEL	rcx をデクリメントし、rcx が 0 でないかつ ZF が 1 ならばラベルの位置にジャンプします。
LOOPNZ 命令	loopnz LABEL	rcx をデクリメントし、rcx が 0 でないかつ ZF が 0 ならばラベルの位置にジャンプします。
PUSH 命令	push rax	rax の値をスタックへ入れます。
POP 命令	pop rax	スタックから値を取り出し、rax へ転送します。
CALL 命令	call LABEL	現在の位置をスタックへ入れてラベルの位置にジャンプします。
RETURN 命令	ret (2h)	スタックから CALL 命令を実行した位置を取り出して、その位置に戻ります。オペランドを指定した場合は、さらに指定したバイト数のスタックを捨てます。
NOP 命令	nop	何もせず、次の命令へ進みます。

そのほかにもたくさんの命令がありますが、[*8]使われるのはこのような基本的な命令が大半ですので、分からない命令を見つけたら、そのつど辞書を引くという感じで大丈夫です。

2.1.5　呼び出し規約と関数の構造

アセンブリ言語に関してはざっと説明したとおりの内容である程度把握できるようになると思いますが、逆アセンブルを解析するにはプログラムの仕様について知っておかなければなりません。

先ほども説明したように、アセンブリ言語には変数というものがありません。レジスタと呼ばれる有限の領域を使いまわすということでしたが、同様に、関数を定義する場合にも演算から引数や戻り値の受け渡しまで、すべてのデータの受け渡し演算で、有限個のレジスタを使いまわして実現しなければなりません。

レジスタを使いまわして、プログラムを書くところまではよいのですが、それぞれの関数が適当なレジスタを適当に使っていたら、どのレジスタに何が入っているのか分からなくなってしまいます。もしかしたら、他の関数が使っていたレジスタに上書きしてしまい、エラーになってしまうかもしれません。そこで、レジスタの用途や、関数呼び出し時の事前・事後処理などについて定義した、呼び出し規約というものが存在します。

[*8] それだけ高機能ってことですね！

今回は x86_64 で動作する Linux の実行ファイル (ELF) の逆アセンブルを読みたいと思っているので、Linux が採用している System V と呼ばれる仕様で定義されている呼び出し規約について確認してみたいと思います。

詳しいところは、直接仕様書を確認してもらいたいのですが、「System V Application Binary Interface AMD64 Architecture Processor Supplement Draft Version 0.98」の「3.2 Function Calling Sequence」によると、次のように定義されています。

スタックフレームの構造とベースポインタ・スタックポインタ

Position	Contents	Frame
8n+16(%rbp)	memory argument eightbyte *n*	Previous
	...	
16(%rbp)	memory argument eightbyte 0	
8(%rbp)	return address	Current
0(%rbp)	previous %rbp value	
−8(%rbp)	unspecified	
	...	
0(%rsp)	variable size	
−128(%rsp)	red zone	

System V Application Binary Interface AMD64 Architecture Processor Supplement Draft Version 0.98(16 ページ) より引用
http://refspecs.linuxfoundation.org/elf/x86_64-abi-0.98.pdf

レジスタの用途

Register	Usage	Preserved across function calls
%rax	temporary register; with variable arguments passes information about the number of SSE registers used; 1st return register	No
%rbx	callee-saved register; optionally used as base pointer	Yes
%rcx	used to pass 4th integer argument to functions	No
%rdx	used to pass 3rd argument to functions; 2nd return register	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No
%r10	temporary register, used for passing a function’s static chain pointer	No
%r11	temporary register	No
%r12-r15	callee-saved registers	Yes
%xmm0—%xmm1	used to pass and return floating point arguments	No
%xmm2—%xmm7	used to pass floating point arguments	No
%xmm8—%xmm15	temporary registers	No
%mmx0—%mmx7	temporary registers	No
%st0,%st1	temporary registers; used to return long double arguments	No
%st2—%st7	temporary registers	No
%fs	Reserved for system (as thread specific data register)	No
mxcsr	SSE2 control and status word partial x87 SW x87 status word No x87 CW x87 control word	Yes

System V Application Binary Interface AMD64 Architecture Processor Supplement Draft Version 0.98(21 ページ) より引用
http://refspecs.linuxfoundation.org/elf/x86_64-abi-0.98.pdf

これらの仕様書の内容と、実際の逆アセンブル結果を比較してわかりやすくまとめてみると、次のようになりました。

関数の構造
(セミコロン「;」で始まるテキストは、コメントです。)

1. ラベル

関数名はラベルとして逆アセンブルされます。
main:

2. スタックフレームのベースポインタ rbp を push

呼び出す側の関数のスタックフレームのベースポインタをスタックに退避します。
push rbp

3. 他に退避すべきレジスタがあれば push

規約により退避すべきとされているレジスタを使用するのであれば、スタックに退避します。
push rbx ; 例

4. スタックのベースポインタを現在のスタックの先頭に設定

呼び出される側の関数のスタックフレームを作成します。
mov rbp, rsp

5. 必要であれば、変数用領域を確保

引き算をしているのは、スタックはメモリを後ろから順に使用するためです。必要なバイト数分 rsp を引き算します。
sub rsp, 0x10

6. 変数への読み書き

スタックフレームに確保したメモリ領域へのアクセスはポインタを使って行います。
mov DWORD PTR [rbp-0x4], 0xa ; 変数へ書き込み
mov eax, DWORD PTR [rbp-0x4] ; 変数から読み出し

7. 関数の呼び出し

mov r9, 0x6 ; 第六引数
mov r8, 0x5 ; 第五引数
mov rcx, 0x4 ; 第四引数
mov rdx, 0x3 ; 第三引数
mov rsi, 0x2 ; 第二引数
mov rdi, 0x1 ; 第一引数
mov eax, 0x0 ; 戻り値を正しい値で受け取るために、eax の初期化がされる場合があります。
call 000000 ; 関数を呼び出します。現在の実行位置を push し、呼び出し先の関数の先頭にジャンプしま

す。
mov edx, eax ;eax=戻り値

8. 戻り値のセット
mov eax, 0x0 ;return 0

9. 呼びだし前の環境の復元
pop rdi ;pop の順番には注意します。
pop rsi
mov rsp, rbp ; スタックポインタの位置を戻し、関数のスタックフレームを開放します。
pop rbp ; 呼び出し前の状態にベースポインタを戻します。
もしくは
leave ;mov rsp, rbp と pop rbp を行ってくれる便利な命令です。

10. 呼び出し元に戻る
呼び出し前の実行位置をポップし、そこへジャンプします。
ret

2.2 hello, world

さて、やっと前提知識の説明が終わりました。これから逆アセンブルを読んでみたいと思います。
今回は Linux と gcc を使って確認していきます。[*9]
まずは、みなさんご存知の hello, world から見てみましょう。

ソースコード 2.5　C のソース

```
#include <stdio.h>

int main() {
      printf("hello, world\n");
      return 0;
}
```

これを gcc でコンパイルします。

```
gcc hello.c -o hello
```

コンパイルされたプログラムが実行できることを確認したら、逆アセンブルしてみましょう。
逆アセンブルには objdump というコマンドを使います。

[*9] Windows では結果が異なります

```
objdump -d -M intel hello
```

-d が逆アセンブルを行うオプション、-M はアセンブリ言語の形式を指定するオプションです。(今回は筆者の趣味により、intel 形式を指定しました)

逆アセンブルを実行すると、次のような結果が得られます。

ソースコード 2.6 objdump の結果の一部

```
00000000004004fc <main>:
  4004fc:       55                      push   rbp
  4004fd:       48 89 e5                mov    rbp,rsp
  400500:       bf c4 05 40 00          mov    edi,0x4005c4
  400505:       b8 00 00 00 00          mov    eax,0x0
  40050a:       e8 d1 fe ff ff          call   4003e0 <printf@plt>
  40050f:       b8 00 00 00 00          mov    eax,0x0
  400514:       5d                      pop    rbp
  400515:       c3                      ret
```

長い逆アセンブルが出力されるので、main というラベルのついた部分を探してください。この部分が main 関数となります。

main 関数の最初の 2 行を見てみましょう。

```
  4004fc:       55                      push   rbp
  4004fd:       48 89 e5                mov    rbp,rsp
```

この部分は、関数内の処理を始めるための前処理になります。Linux の呼び出し規約どおり、ベースポインタ (rbp) をスタックに push し、スタックの先頭に移動してスタックフレームを作成しています。

今回は変数を使っていないため、スタックフレームを拡張して変数領域を確保することは行っていません。

次の 3 行を見てみます。

edi は関数の第一引数でした。第一引数に文字列リテラルへのポインタをわたしています。また、戻り値が入れられる eax を初期化しています。

引数のセット後に、printf 関数を呼び出しています。

```
  400500:       bf c4 05 40 00          mov    edi,0x4005c4
  400505:       b8 00 00 00 00          mov    eax,0x0
  40050a:       e8 d1 fe ff ff          call   4003e0 <printf@plt>
```

最後の 3 行を見てみます。

eax には戻り値 0 をセットしています。スタックに push したベースポインタを pop して、元に戻しています。スタックポインタは、変数領域の確保をしなかったため、変更する必要がありません。

最後に ret 命令で呼び出し元の関数に戻ります。

```
  40050f:       b8 00 00 00 00          mov    eax,0x0
  400514:       5d                      pop    rbp
  400515:       c3                      ret
```

2.3　変数

次は変数を使ってみることにします。

次のようなプログラムをコンパイルして、逆アセンブルしてみます。

```
#include <stdio.h>

int main() {
        int i=10;
        printf("hello, world\ni = %d", i);
        return 0;
}
```

```
objdump -d -M intel hello2
```

ソースコード 2.7　逆アセンブル結果の一部

```
000000000040050c <main>:
  40050c:       55                      push   rbp
  40050d:       48 89 e5                mov    rbp,rsp
  400510:       48 83 ec 10             sub    rsp,0x10
  400514:       c7 45 fc 0a 00 00 00    mov    DWORD PTR [rbp-0x4],0xa
  40051b:       8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
  40051e:       89 c6                   mov    esi,eax
  400520:       bf ec 05 40 00          mov    edi,0x4005ec
  400525:       b8 00 00 00 00          mov    eax,0x0
  40052a:       e8 b1 fe ff ff          call   4003e0 <printf@plt>
  40052f:       b8 00 00 00 00          mov    eax,0x0 ;return 0
  400534:       c9                      leave
  400535:       c3                      ret
```

main 関数の最初の 3 行を見てみます。

ベースポインタをスタックに push し、スタックの先頭に移動しています。次に、変数領域を確保するために、スタックフレームを拡張しています。

スタックはメモリを後ろから順に使うため、スタックフレームを拡張するためにはスタックポインタの値を減らします。

```
  40050c:       55                      push   rbp
  40050d:       48 89 e5                mov    rbp,rsp
  400510:       48 83 ec 10             sub    rsp,0x10
```

次の 1 行を見てみます。

mov 命令で、確保した変数領域へ代入処理を行っています。

```
  400514:       c7 45 fc 0a 00 00 00    mov    DWORD PTR [rbp-0x4],0xa
```

次の 5 行を見てみます。

変数の値を読み出し、第二引数に使用する esi にわたしています。第一引数に使用する edi には文字列リテラルへのポインタをわたしています。

戻り値に使用する eax を初期化してから、printf 関数を呼び出しています。

```
40051b:        8b 45 fc                    mov     eax,DWORD PTR [rbp-0x4]
40051e:        89 c6                       mov     esi,eax
400520:        bf ec 05 40 00              mov     edi,0x4005ec
400525:        b8 00 00 00 00              mov     eax,0x0
40052a:        e8 b1 fe ff ff              call    4003e0 <printf@plt>
```

最後の 3 行を見てみます。

戻り値に使用する eax に 0 をセットし、leave 命令でスタックポインタとベースポインタを呼び出し前の状態に戻しています。

最後に ret 命令で呼び出しもとの関数に戻っています。

```
40052f:        b8 00 00 00 00              mov     eax,0x0 ;return 0
400534:        c9                          leave
400535:        c3                          ret
```

2.4　さいごに

逆アセンブルを読むための知識をひたすら説明してきましたが、説明しなければならない内容が多く、だいぶ疲れました。しかし、この記事を書いたことで、自分の足りない知識を得ることができました。この記事には、もっと書かなければならないことはたくさんあったのですが、執筆が超期限ぎりぎりになってしまったので、ここまで書いて時間が来てしまいました。

部誌担当の方には最後まで待っていただき、感謝しています。

眠くなるような内容だったと思いますが、最後まで読んでいただきありがとうございました。

hiro1357 / eserver_dip_jp@yahoo.co.jp

参考文献

[1]『アセンブリ言語の教科書』 愛甲健二 データハウス ISBN4-88718-829-3

[2] Intel(R) 64 and IA-32 Architectures Software Developer Manuals `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`

[3] System V Application Binary Interface AMD64 Architecture Processor Supplement Draft Version 0.98 `http://refspecs.linuxfoundation.org/elf/x86\_64-abi-0.98.pdf`

百萬石 2015 -春- ©電気通信大学 MMA

2015 年 4 月 6 日　初版第一刷発行

著　者　kakakaya,hiro1357

表　紙　g_nootoko
編集者　masn19
発行者　電気通信大学 MMA
発行所　電気通信大学 MMA 部室
〒182−8585 東京都調布市調布ヶ丘 1−5−1 サークル会館 208 号室
http://www.mma.club.uec.ac.jp/
印刷所　電気通信大学 MMA 部室
製本所　電気通信大学 MMA 部室