作って分かる！
x86_64 機械語入門

— OS レスで CPU と直接話す! —

大神祐真　著

技術書典 7（2019 年秋）新刊
2019 年 9 月 22 日　ver 1.0

■免責

本書は情報の提供のみを目的としています。
本書の内容を実行・適用・運用したことで何が起きようとも、それは実行・適用・運用した人自身の責任であり、著者や関係者はいかなる責任も負いません。

■商標

本書に登場するシステム名や製品名は、関係各社の商標または登録商標です。
また本書では、™、®、© などのマークは省略しています。

はじめに

本書をお手に取っていただきありがとうございます。

本書では、一つ一つ構文を理解していくことで機械語を書いてみます。機械語は、ソフトウェアエンジニアの領域の言語としては低レイヤー中の低レイヤーで、これより下は無いってくらいですが、バイナリ列とはいえちゃんとパターンがあり、構文として捉えることができ、人の手で書くこともできます。

本書の構成

本書は以下の構成です。

- 第 1 章 環境構築とはじめてのプログラム
  - ループだけの簡単なプログラムを通して、環境構築をしながら、本書で機械語を理解していく流れを紹介します
  - 開発はテキストベースで行えるように簡単なスクリプトを使います
  - 実行環境は QEMU 上です。OS レスで直接動作させます。
  - また、機械語を理解するための補助器具として、シェル上で手軽にアセンブル・逆アセンブルするツールも用意しておきます
    - 以降の章ではこのツールも活用しながら、機械語表現が分からないときは適宜アセンブラからアセンブルしてみます[*1]
- 第 2 章 基本的な演算
  - 基本的な演算命令 (四則演算、論理演算等) の機械語における構文を紹介します
- 第 3 章 シリアルドライバを作る
  - シリアルドライバを作り、QEMU のシリアル出力画面に文字を表示してみます
- 第 4 章 フレームバッファで画面描画
  - フレームバッファと呼ばれる領域へピクセル値を書き込むことで画面描画します
- 第 5 章 キーボードドライバを作る
  - キーボードドライバを作り、キーボードからの入力を取得します
- 付録 A 登場した機械語命令と構文の一覧

[*1] 「それなら最初からアセンブラで書けば良いじゃん」は禁句です (笑)。まあ、この本の目的は機械語を勉強して書いてみることであって、何らかの別の言語から変換してきては意味がないので。

- 登場した構文と命令を一覧にまとめています

開発環境・動作確認環境

筆者が開発や動作確認を行っている環境を以下に記載します。

- OS: Debian GNU/Linux 9.9 (stretch)
- binutils: 2.28 (Debian stretch のバージョン)
- QEMU: 2.8.1 (Debian stretch のバージョン)

本書の PDF 版/HTML 版やソースコードの公開場所について

これまでの当サークルの同人誌同様、本書も PDF 版/HTML 版は以下のページで無料で公開しています。

- `http://yuma.ohgami.jp`

サンプルコードは以下の GitHub リポジトリで公開しています。

- `https://github.com/cupnes/introduction_to_x86_64_machine_language_samples`

サンプルコードのリポジトリ内はディレクトリで分かれています。各ディレクトリが以降の各節あるいは各項で作るものです。節あるいは項の冒頭で「この節/項のサンプルディレクトリは XX です。」と紹介しますので適宜参照してみてください。

目次

はじめに **i**
本書の構成 i
開発環境・動作確認環境 ii
本書の PDF 版/HTML 版やソースコードの公開場所について ii

第 1 章 環境構築とはじめてのプログラム **1**
1.1 本書で機械語を理解していく流れ 1
1.2 機械語構文の理解を補助するツール (objdump, as_dump, das_dump) 1
binutils をインストール 1
簡単なプログラムを逆アセンブルしてみる 2
小さなアセンブラコードを通して jmp 命令を理解 4
その場で逆アセンブル 6
構文まとめ (命令 1「jmp」) 6
1.3 テキストベースでバイナリを書く (t2b) 7
16 進数が書かれたテキストをバイナリへ変換するコマンド (t2b) 7
テキストで機械語コードを書いてみる 7
バイナリへ変換 8
1.4 VM 上で OS レスで実行する (QEMU, OVMF, poiboot) 9
バイナリフォーマットについて 9
QEMU と OVMF をインストール 10
poiboot を取得 11
実行できるようにファイルを配置 11
QEMU で動作確認 12
QEMU モニターで実行状況を確認する 13
1.5 何もしない命令「nop」(命令 2) 15
1.6 CPU を休ませる「hlt」(命令 3) 16

第 2 章 基本的な演算 **19**
2.1 データ格納命令 (mov)(命令 4) 19
汎用レジスタ 19
即値をレジスタへ格納する機械語コードを見てみる 20
他のレジスタへ格納する命令も見てみる 21
構文: mov: 即値 (8bit) をレジスタ (8bit) へ格納 22
構文: mov: 即値 (16bit) をレジスタ (16bit) へ格納 22
2.2 四則演算 24

和 (add)(命令 5) 24
差 (sub)(命令 6) 26
積 (imul)(命令 7) 27
商と剰余 (idiv)(命令 8) 30
2.3 論理演算 31
AND(命令 9) 31
OR(命令 10) 32
XOR(命令 11) 33
NOT(命令 12) 34
2.4 シフト演算 (命令 13-15) 34
2.5 インクリメント/デクリメント (命令 16-17) 36

第 3 章 シリアルドライバを作る 37
3.1 文字を送信して画面表示 37
シリアル通信を行うには 37
IO アドレス空間のレジスタへデータを書き込む「out」(命令 18) 38
動作確認 39
3.2 コントローラの状態を確認して送信する 39
Line Status Register(LSR) 40
IO アドレス空間のレジスタ値取得「in」(命令 19) 40
フラグの待ち方「and」/「je」(命令 20) 41
フラグレジスタの動きを見てみる 42
LSR のビット 5 を待つ処理を実装 43
3.3 受信も追加しエコーバックプログラムを作る 44
受信バッファにデータが来るのを待つ (命令 21「jne」/命令 22「pause」) . . . 44
受信バッファのデータを取得する 48
mov 命令の新構文 (8 ビットレジスタから 8 ビットレジスタへコピー) 49
受信バッファから取得したデータを送信する 50
動作確認 50
CR を受け取ったら LF を追加で送信するようにする (命令 23「cmp」) . . . 50
動作確認 52

第 4 章 フレームバッファで画面描画 53
4.1 フレームバッファのアドレスを取得する 53
フレームバッファに関する情報の在処 53
レジスタ間接アドレッシング 54
実装 55
動作確認 56
4.2 画面の幅と高さを取得する 56

ディスプレースメント付きレジスタ間接アドレッシング 56
備考: DSP の有無による機械語の変化 58
実装 59
動作確認 59
4.3 画面へ 1 ピクセル描画してみる 60
即値をレジスタ間接で書き込む 60
実装 61
動作確認 61
4.4 画面を塗りつぶしてみる 62
実装の流れ 62
等しいか小さければジャンプ (命令 24「jbe」) 62
実装 63
初出の構文について 64
動作確認 67
4.5 ランダム色で塗りつぶしてみる 67
rdrand 命令で乱数取得 (命令 25) 68
実装 68
動作確認 70

第 5 章 キーボードドライバを作る 71
5.1 キー入力を受け取るまでの流れ 71
5.2 ステータスレジスタ値を取得 72
5.3 キー入力値が無ければ先頭へ戻る 72
5.4 データレジスタからキー入力値を取得 73
5.5 キー入力値が Break ならスキャンコードを抽出し、Make なら先頭へ戻る 74
実装方針 74
比較結果が「小さい」ならジャンプ (命令 26「jb」) 75
実装 75
5.6 スキャンコードをレジスタ CL へコピーし、先頭へ戻る 76
5.7 動作確認 77

付録 A 登場した機械語命令と構文の一覧 79
A.1 構文表記のキーワードについて 79
rel_addr 79
reg_ofs 79
reg_src_dst 79
reg_dst_src 80

A.2 四則演算 82
add: 加算 82
sub: 減算 82
imul: 乗算 83
idiv: 除算 83
A.3 インクリメント/デクリメント 83
inc: インクリメント 83
dec: デクリメント 84
A.4 比較 84
cmp: 2 つのオペランドの差を評価 84
A.5 論理演算 84
and: 論理積 84
or: 論理和 85
xor: 排他的論理和 85
not: 否定 85
A.6 ビットシフト 85
sal: 算術左シフト 86
sar: 算術右シフト 86
shr: 論理右シフト 86
A.7 読み出し/書き込み 86
mov: 第 1 から第 2 オペランドへコピー 86
in: IO アドレス空間から読み出し 87
out: IO アドレス空間へ書き込み 88
A.8 CPU 状態制御 88
hlt: CPU をスリープ 88
pause: ビジーループのヒント 88
A.9 分岐 88
jmp: 無条件ジャンプ 88
jb: 小さい (Below) 場合にジャンプ 88
jbe: 小さいか等しい (Below or Equal) 場合にジャンプ 88
je: 等しい (Equal) 場合にジャンプ 89
jne: 等しくない (Not Equal) 場合にジャンプ 89
A.10 その他 89
nop: 何もしない (No Operation) 89
rdrand: 乱数取得 89

おわりに **91**

参考情報 **93**
参考にさせてもらった情報 93
poiboot について 93

第1章 環境構築とはじめてのプログラム

1.1 本書で機械語を理解していく流れ

本書では、C 言語やアセンブラをコンパイル/アセンブルした実行バイナリを調べることで、機械語における各命令の書き方 (構文) を一つ一つ理解します。そして、理解した構文を使って小さな機械語プログラム (=実行バイナリ) を書いて実行してみます。

流れをまとめると以下の通りです。

1. 構文を理解する
2. 実行バイナリを書く
3. 実行する

この章では以降、上記の流れを、環境構築を行いながら簡単なプログラムを使って紹介します。

1.2 機械語構文の理解を補助するツール (objdump, as_dump, das_dump)

♣ binutils をインストール

本書では、実行バイナリから機械語構文を確認する際、「objdump」というコマンドを使って逆アセンブルを行います。逆アセンブル結果を機械語とアセンブリ言語で並べて表示してくれるため、機械語で書いた場合のバイナリ列 (すなわち構文) はどのような表現になるのかを確認するのに役立ちます。

objdump コマンドは「binutils」パッケージに入っています。まずはこのパッケージをインストールします。

```
$ sudo apt install binutils
```

以下のようにバージョンが表示できれば OK です。(バージョンは違っていても構い

ません。)

```
$ objdump -v
GNU objdump (GNU Binutils for Debian) 2.28
Copyright (C) 2017 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License version 3 or (at your option) any later version
.
This program has absolutely no warranty.
```

♣ 簡単なプログラムを逆アセンブルしてみる

objdump を使ってみます。試しに C 言語で「無限ループだけのプログラム」を書いて、逆アセンブルしてみることにします。

C 言語のプログラムはリスト 1.1 の通りです。お好きなエディタで書いてみてください。

▼リスト 1.1: 011_jmp/loop.c

```
int main(void)
{
        while (1);

        return 0;
}
```

なお、このサンプルプログラムは、本書のサンプルプログラムをまとめているリポジトリ[*1]内の「011_jmp」ディレクトリ内に置いています。

ここでは「loop.c」というファイル名で作成したとします。

これをコンパイルします。本書ではコンパイラには GCC を使用しますので、まだインストールされていなければインストールしてください。

```
$ sudo apt install gcc
```

以下のようにバージョンが表示できれば OK です。(本書で扱う範囲では、バージョンは違っていても構いません。)

```
$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/6/lto-wrapper
```

[*1] サンプルリポジトリの URL は「はじめに」をご覧ください。

```
Target: x86_64-linux-gnu
Configured with: ../src/configure -v ・・・省略・・・
Thread model: posix
gcc version 6.3.0 20170516 (Debian 6.3.0-18+deb9u1)
```

それでは、コンパイルします。「-g」を付けて、実行バイナリにデバッグ情報も付加しています。

```
$ gcc -g -o loop loop.c
```

これを逆アセンブルしてみます。

```
$ objdump -S loop | less
・・・省略・・・

0000000000000660 <main>:
int main(void)
{
 660:   55                      push   %rbp
 661:   48 89 e5                mov    %rsp,%rbp
        while (1);
 664:   eb fe                   jmp    664 <main+0x4>
 666:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 66d:   00 00 00

・・・省略・・・
```

「-S」は「デバッグ情報があれば C のソースコードも併せて表示」するオプションです。

また、行数が多いので、スクロールできるようにパイプでページャ (less) に繋いでいます。

出力をスクロールしていくと、「int main(void)」があり、そこより下の行に main 関数の実装があります。

少し下の行に行くと「while (1);」があり、その直後の行が、無限ループを実現している機械語、およびアセンブラのコードです。該当の行のみ抜粋するとリスト 1.2 の通りです。

▼リスト 1.2: 無限ループの機械語およびアセンブラの抜粋

```
 664:   eb fe                   jmp    664 <main+0x4>
```

objdump の出力は、各行で機械語とアセンブラが対応しています。この場合、アセンブラが「`jmp 664 <main+0x4>`」で、その機械語が「`eb fe`」です。たった 2 バイトで無限ループを実現できるのですね。

一番左にコロン区切りで書かれている「**664:**」は実行バイナリ先頭からのバイト数です。「`jmp 664`」の「664」はこの値を示しており、jmp 命令自身と同じ場所にジャンプしてくるため、無限ループとなります。

そんなわけで、1 つ目の命令は「jmp」命令です。もう少し詳しく見てみます。

♣ 小さなアセンブラコードを通して jmp 命令を理解

それでは、たった 2 バイトの命令の中に「先頭から何バイト目」という絶対アドレスが埋め込まれているのでしょうか？ このような無限ループを、例えば先頭から 0 バイト目に配置すると機械語コードも変わるのでしょうか？

実験してみましょう。ループを jmp 命令で実現できることはわかっているので、ここでは 1 行〜数行のアセンブラコードを機械語に変換してみることで、jmp 命令のアセンブラの表現と機械語の対応を理解します。

アセンブリ言語で書かれたソースコードを機械語 (実行バイナリ) へ変換するにはアセンブラを使用します。ですが、たった 1 行〜数行のコードで実験して見るためだけにわざわざコードを書いて、アセンブルして、実行バイナリを 16 進数でダンプする、というのは面倒です。本書では、それらを行うコマンドを、「`as_dump`」というシェルスクリプトで用意しています。

as_dump はサンプルリポジトリの「`tools`」ディレクトリに入っています。

as_dump はアセンブルに「as コマンド (GNU アセンブラ)」を使用します。as は binutils に入っているので、objdump を使用する際にインストールしていれば問題ありません。

それでは、as_dump を使って簡単なアセンブラを機械語コードに変換してみます。

jmp 命令のオペランド[*2]にはラベルが使えます。そして、「今この場所」を示すラベルには「.(ドット)」があり、「`jmp .`」で無限ループを実現できます。

as_dump を使って「`jmp .`」の 1 行だけのアセンブラコードを機械語に変換してみます。

```
$ as_dump jmp .

/tmp/tmp.hdumfEdnIQ/a.out:     ファイル形式 elf64-x86-64


セクション .text の逆アセンブル:

0000000000000000 <.text>:
   0:   eb fe                   jmp    0x0
```

[*2] 命令へ与えるパラメータのこと。jmp 命令の場合、先程の「4」がオペランドに相当します。

このように、as_dump コマンドは引数にアセンブラを与えることで、1 行のアセンブラをその場で機械語に変換し、結果を objdump による逆アセンブル結果として表示します。

結果に関して、jmp 命令以外には何もないアセンブラコードなので、jmp のオペランドが「0x0(先頭から 0 バイト目)」なのは意図通りです。ただ、今回の場合も機械語コードは「`eb fe`」となりました。どうやら、機械語コードにはジャンプ先は相対アドレスで埋め込まれていそうです。それを逆アセンブラのツール (objdump) が逆アセンブルの際、よしなに絶対アドレスで表示してくれていたのでしょう。

それでは、「`eb fe`」のどこに相対アドレスが埋め込まれていたのでしょうか。今度はラベルを使って少し先へジャンプさせてみましょう。

as_dump を引数なしで実行すると入力を受け付ける状態になりますので、その場で 2 行のアセンブラコードを書きます。「`Ctrl-D`」で抜ければ、書いたアセンブラコードを機械語へ変換します。

```
$ as_dump
        jmp     loop
loop:   jmp     .
<Ctrl-Dで抜ける>

/tmp/tmp.V4twRVogBv/a.out:     ファイル形式 elf64-x86-64


セクション .text の逆アセンブル:

0000000000000000 <loop-0x2>:
   0:   eb 00                   jmp    2 <loop>

0000000000000002 <loop>:
   2:   eb fe                   jmp    2 <loop>
```

2 つ目の jmp 命令の地点に「loop」というラベルを付けました。

1 行目の jmp 命令は自身の命令の場所から自身の命令長である 2 バイト先にジャンプすることになります。その場合、機械語コードは「`eb 00`」になりました。自身の命令と同じ場所へジャンプする場合は、これまで確認した通り「`eb fe`」です。

どうやら、機械語コードの「`eb`」が「`jmp`」命令であることを示し、続く 2 バイト目がジャンプ先の相対アドレスを示しているようです。

また、2 バイト目に関して、「今この場所」へジャンプする際は「`fe`」で、「2 バイト目」へジャンプする際は「`00`」でした。「0xfe + 2 = 0x100」で、1 バイトからオーバーフローした分を捨てると「0x00」になることから、eb 命令 (jmp 命令) のオペランドは 0xfe を基準とした相対アドレスであることがわかります。

♣ その場で逆アセンブル

as_dump の逆に、その場で逆アセンブルする「das_dump」コマンドもサンプルディレクトリの tools ディレクトリに入れてあります。

少し追加で実験してみます。

先ほど、jmp 命令の機械語である「eb XX」命令のオペランドは、「fe」を基準として、それに足し合わせた数値の分がジャンプ先の相対的なアドレスになることを確認しました。

それでは、「fe」から値を引いた場合はどうなるのでしょうか？

例として、「fe」から 1 を減算した値である「fd」をオペランドとした「eb fd」を逆アセンブルしてみます。

```
$ das_dump eb fd
/tmp/tmp.qQjOC52hIN/a.bin:     ファイル形式 binary


セクション .data の逆アセンブル:

0000000000000000 <.data>:
   0:   eb fd                   jmp    0xffffffffffffffff
```

das_dump コマンドも、使い方は as_dump コマンドと同様です。コマンドライン引数に逆アセンブルしたい機械語を 1 バイトずつ半角スペース区切りで 16 進数で指定すると、それを objdump で逆アセンブルした結果を表示します。

逆アセンブルの結果、jmp のオペランドは「0xffffffffffffff」となりました。この値は 2 の補数で「-1」を示します。「基準 fe から引いた値を指定するとその分だけマイナスの方向へジャンプする」ということだと分かります[*3]。

♣ 構文まとめ (命令 1「jmp」)

相対アドレス指定でジャンプする jmp 命令の機械語コードとアセンブラの構文を表 1.1 にまとめます。

表 1.1 の「アセンブラ」の表記について、objdump では jmp のオペランドを実行バイナリ先頭からのバイト数あるいは相対アドレスで表示していましたが、一般的にアセンブラを書く際はジャンプ先にラベルを置いて、jmp のオペランドにはラベルを指定します。

[*3] プラス方向へのジャンプについて fe からの増分が 2 以上の場合、00 へラップ (1 周) するので、プラス方向へのジャンプの最大値とマイナス方向へのジャンプの最大値はどこかでぶつかります。それがどこなのかも das_dump で適当にオペランドを変えて逆アセンブルしてみることで確認できます。興味があれば実験してみてください。

▼表 1.1: 構文 1: jmp: 相対アドレス指定でジャンプ

アセンブラ	jmp <ラベルあるいは".">
機械語	eb fe+rel_addr

また「機械語」の欄の「+rel_addr」は相対的にジャンプさせたいバイト数を足し合わせる事を示しています。

以降では、新たな構文を見つける/あるいは紹介する度に、このような表にまとめます。

1.3　テキストベースでバイナリを書く (t2b)

ここまでで、jmp 命令の相対アドレス指定での構文を理解しました。

次は、理解した構文を使って、実際に動作する実行バイナリを作ってみます。

♣ 16 進数が書かれたテキストをバイナリへ変換するコマンド (t2b)

バイナリを書く際、一般的にはバイナリエディタを使うかと思うのですが、そもそも「バイナリを書く」事が一般的ではないためか、0 から書いていくような場合に使い勝手のよいバイナリエディタがなかなか無い印象です。

そこで、本書では、16 進数が羅列されたテキストファイルをバイナリへ変換するスクリプト「t2b」を用意して、機械語でのコーディング作業はテキストエディタで行うことにします。

t2b もシェルスクリプトで、サンプルリポジトリの tools ディレクトリにあります。

標準の UNIX コマンドしか使っていないため、動作のために新たに必要なパッケージは特にありません。

♣ テキストで機械語コードを書いてみる

それでは、コードを書いてみます。

前述の実験を少し変更し、2 つの jmp 命令を並べて、交互にジャンプするようにしてみました (リスト 1.3)。

内容は「sample.txt」というファイルで保存することにします。また、サンプルリポジトリの中の、「011_jmp」ディレクトリの中にも置いてあります。

▼リスト 1.3: 011_jmp/sample.txt(機械語コードのみ)

```
eb 00
eb fc
```

なお、t2b は行頭・行中どちらでも、「#」以降の記述をコメントとして扱います。そのため、リスト 1.4 の様にコメントを付けることが可能です。機械語と「#」の間はタブ文字でもスペースでもどちらでも構いません。

▼リスト 1.4: 011_jmp/sample.txt

```
# 2つのjmp命令を交互に行き来する
eb 00   # jmp +2
eb fc   # jmp -2
```

また、「:(コロン)」より左側の記述も無視するため、リスト 1.5 の様に先頭からのバイト数などを付け加えても構いません。

▼リスト 1.5: 011_jmp/sample.txt

```
# 2つのjmp命令を交互に行き来する
0: eb 00        # jmp +2
2: eb fc        # jmp -2
```

♣ バイナリへ変換

t2b を使用して以下のようにバイナリへの変換が行えます。

```
$ t2b sample.txt > code.bin
```

od コマンドでバイナリをダンプしてみると、意図したとおりに書かれていることを確認できます。

```
$ od -t x1 code.bin
0000000 eb 00 eb fc
0000004
```

「-t x1」は 1 バイト区切りで表示するオプションです。

アドレスの大きい方へ 2 バイト先にジャンプする「eb 00」と、アドレスの小さい方へ 2 バイト前へジャンプする「eb fc」が書かれており、意図通りのバイナリであることを確認できました。

1.4　VM 上で OS レスで実行する (QEMU, OVMF, poiboot)

♣ バイナリフォーマットについて

ここまでで機械語命令を格納したバイナリは生成できました。ただ、「実行バイナリ」とするにはヘッダが足りません。

「はじめに」でも説明した通り、本書では手で書いた実行バイナリを、QEMU 上で OS レスで実行します。なお、QEMU にしろ実機にしろ、電源を入れてから (QEMU の場合はエミュレータを立ち上げてから) 最初に実行されるのは BIOS[*4]です。ただ、本書で作るバイナリは BIOS のお作法に従ったものではないため、作ったバイナリをロードし実行を開始してくれる「ブートローダー」が必要です。本書では「poiboot」という独自のブートローダーを使用します。ブートローダーが関わるのはバイナリの実行開始のみで、その後は OS レスで直接ハードウェアを制御します。

ブートローダー「poiboot」が実行できるバイナリフォーマットはシンプルで表 1.2 の通りです。

▼表 1.2: poiboot が実行できるバイナリフォーマット

ヘッダ	bss 領域の先頭アドレス (8 バイト)
(先頭 16 バイト)	bss 領域のサイズ (8 バイト)
本体	実行バイナリ本体 (機械語コード)
(17 バイト目以降)	※ poiboot はロード後、本体の先頭 (ヘッダの直後) へジャンプする

bss とは、C 言語での初期値なし (あるいは初期値 0) のグローバル変数や static 変数等で、プログラム実行開始時にゼロで初期化しておくべき領域です。poiboot は実行バイナリ先頭の 16 バイトの内容から bss 領域のアドレスとサイズを確認し、本体の実行開始前にその領域をゼロで初期化します。

poiboot が事前にやることはこれだけです。初期化が完了したら、実行バイナリ本体の先頭へジャンプします。

bss については、ヘッダの「bss 領域のサイズ」が 0 の場合、poiboot は何もしないです。本書では特に bss 領域は使わないので、全て 0 の 16 バイトのバイナリを、機械語コードの先頭にくっつければ良いだけです。

dd コマンドで全て 0 の 16 バイトのデータ「`head.bin`」を生成し、前項で作成した

[*4] 現代は UEFI BIOS

「code.bin」と結合します。

```
$ dd if=/dev/zero of=head.bin bs=1 count=16
16+0 レコード入力
16+0 レコード出力
16 bytes copied, 0.000514814 s, 31.1 kB/s
$ cat head.bin code.bin > kernel.bin
```

poiboot は「kernel.bin」という名前の実行ファイルを認識するので、この名前で生成しました。

♣ QEMU と OVMF をインストール

以下のように QEMU をインストールします。

```
$ sudo apt install qemu-system-x86 ovmf
```

poiboot は、旧来の BIOS(レガシー BIOS) ではなく、「UEFI」という近年一般的に使われている BIOS を前提としているのですが、QEMU には UEFI のファームウェアが付属しないため、オープンソースの UEFI ファームウェアである「OVMF」を併せてインストールしています。

OVMF については、ホームディレクトリ直下に「OVMF」というディレクトリを作成し、そこへファームウェアバイナリを配置しておきます。

「ovmf」パッケージによってインストールされたファイルの一覧は以下のコマンドで確認できます。

```
$ dpkg -L ovmf
/.
/usr
/usr/share
/usr/share/OVMF
/usr/share/OVMF/OVMF_CODE.fd
/usr/share/OVMF/OVMF_VARS.fd
/usr/share/doc
/usr/share/doc/ovmf
/usr/share/doc/ovmf/changelog.Debian.gz
/usr/share/doc/ovmf/copyright
/usr/share/ovmf
/usr/share/ovmf/OVMF.fd
/usr/share/qemu
/usr/share/qemu/OVMF.fd
```

「OVMF_CODE.fd」と「OVMF_VARS.fd」が本書で使用するファームウェアバイナリです。

以下のようにホームディレクトリ直下にディレクトリを作成し、ファイルをコピーしておいてください。

```
$ mkdir ~/OVMF
$ cp /usr/share/OVMF/OVMF_CODE.fd ~/OVMF/
$ cp /usr/share/OVMF/OVMF_VARS.fd ~/OVMF/
```

♣ poiboot を取得

poiboot は GitHub 上で公開しています。

- poiboot - GitHub
  - `https://github.com/cupnes/poiboot/`

本書のブートローダーとして動作確認した時点は「`x86_64_ml_20190922`」でリリースしており、リリースページの URL は以下の通りです。

- Release x86_64_ml_20190922 - cupnes/poiboot
  - `https://github.com/cupnes/poiboot/releases/tag/x86_64_ml_20190922`

poiboot のコンパイル済み実行バイナリは、上記のリリースページにて「poiboot_x86_64_ml_20190922.zip」という zip アーカイブで取得できます。
ダウンロードし、展開しておいてください。以下の 2 つのファイルを使用します。

- poiboot.efi: poiboot 本体
- poiboot.conf: 設定ファイル

♣ 実行できるようにファイルを配置

必要なものは全て用意できました。作成した実行バイナリを QEMU 上で実行するために、リスト 1.6 のようにファイルを配置してください。

▼リスト 1.6: QEMU で実行するためのファイル配置

```
fs/
├ efi/
│   └ boot/
│       └ bootx64.efi  ← poiboot.efi をリネームして配置
├ kernel.bin
└ poiboot.conf
```

ここでは、「`fs`」という名前の作業ディレクトリへファイルを配置したとします。
この階層構造について簡単に説明すると、これは UEFI ファームウェア (QEMU 上で実行する場合 OVMF) が実行ファイルを見つけるためのファイル配置です。UEFI は

レガシー BIOS とは違い、FAT ファイルシステムを認識できます。そして、起動ディスク上の FAT ファイルシステムに「efi/boot/bootx64.efi」という名前で実行ファイルが配置されていた場合、それを実行してくれます。

bootx64.efi は「poiboot」です。poiboot は FAT ファイルシステム上の「kernel.bin」を実行するため、作成した実行ファイルが実行される、という流れです。

作成した実行バイナリ「kernel.bin」が実行されるまでの流れをまとめると以下の通りです。

1. QEMU を起動する
2. UEFI ファームウェア (OVMF) が起動し efi/boot/bootx64.efi(poiboot) 実行
3. poiboot は kernel.bin(作成した実行バイナリ) を実行

♣ QEMU で動作確認

ファイルを配置できたので、QEMU で実行してみます。

fs が存在するディレクトリと同じ階層で以下のコマンドを実行してください。

```
$ ls -d fs
fs  ← 「fs」が見える階層(fsと同じ階層)で実行
$ qemu-system-x86_64 \
        -m 4G -enable-kvm \
        -drive if=pflash,format=raw,readonly,file=$HOME/OVMF/OVMF_CODE.fd \
        -drive if=pflash,format=raw,file=$HOME/OVMF/OVMF_VARS.fd \
        -drive dir=fs,driver=vvfat,rw=on
```

KVM が無い環境の場合、「-enable-kvm」のオプションは外してください。

「-drive dir=fs,driver=vvfat,rw=on」のオプションで、前項で作成した「fs」ディレクトリを FAT ファイルシステムとして扱うように指定しています。QEMU を実行する際のカレントディレクトリが fs が存在するディレクトリとは異なる場合、「dir=fs」の「fs」を良しなに変更してください。(相対パスあるいは絶対パスで指定すれば良いです。)

実行すると、GUI で QEMU の画面が表示されたかと思います。ただ、今回は何も画面に表示するプログラムではないので、見た目上は何も変化はありません。

この QEMU コマンドは今後手で打つにはしんどいくらいに長いので、「run_qemu」という名前でシェルスクリプト化してあります。他のツールと同様にサンプルリポジトリの tools ディレクトリに入れてありますので、今後はこちらを使います。

先ほどのコマンドを「run_qemu」を使って実行する場合、以下の通りです。

```
$ run_qemu fs
```

run_qemu は、引数に指定したディレクトリを、ブートローダーや実行バイナリが配置されたディレクトリとして使用します。

なお、QEMU は CUI で実行することもできます。QEMU に「-nographic」オプションを追加すると CUI で QEMU を起動できます。run_qemu は 2 つ目以降のコマンドライン引数を QEMU への追加オプションとして扱いますので、以下のようにオプションを追加すると、CUI モードで起動できます。

```
$ run_qemu fs -nographic
```

♣ QEMU モニターで実行状況を確認する

QEMU には QEMU の実行状態を確認できる「QEMU モニター」という機能があります。

QEMU モニターの画面へ切り替えてみましょう。切り替え方は GUI/CUI のそれぞれ、表 1.3 の通りです。

▼表 1.3: QEMU モニターへの切り替え方法

GUI の場合	
QEMU モニターへ切り替え	**Ctrl-Alt-2**
元の画面に戻る	**Ctrl-Alt-1**
CUI の場合	
QEMU モニターへ切り替え	**Ctrl-a 押下のあと c キー押下**
元の画面に戻る	**Ctrl-a 押下のあと c キー押下**
ヘルプ表示	**Ctrl-a 押下のあと?キー押下**

CPU の演算やその結果は基本的に「レジスタ」という CPU 専用の高速な記憶領域に格納されます。「`info registers`」コマンドで現在の CPU のレジスタ値を確認できます。

```
QEMU 2.8.1 monitor - type 'help' for more information
(qemu) info registers
RAX=0000000000110000 RBX=00000000bef67f98 RCX=0000000000000000 RDX=00000000000›
›00000
RSI=0000000000407180 RDI=00000000bfe9b018 RBP=00000000bff0eb10 RSP=00000000002›
›10000
```

```
R8 =00000000bff0e9dc R9 =0000000000000078 R10=00000000bfe6e080 R11=00000000e7d›
›c4657
R12=00000000bef69540 R13=00000000bef69548 R14=00000000bff24b60 R15=00000000bef›
›6a018
RIP=0000000000110002 RFL=00000002 [-------] CPL=0 II=0 A20=1 SMM=0 HLT=0
...
```

GUI の際、出力が 1 画面に収まらず流れてしまった際は「Ctrl を押しながら上下キー」でスクロールできます。

ここで注目すべきは「RIP=」の値です。これは、今 CPU が実行している命令のアドレスです。poiboot は「kernel.bin」内の実行バイナリ本体 (ヘッダより後の 17 バイト目以降) を、メモリアドレス「0x110000」から配置するので、RIP が「0x110000」から大きく離れていると異常です。今回の場合、1 つ目の jmp 命令のアドレスは「0x110000」で、2 つ目の jmp 命令のアドレスは「0x110002」となりますので、RIP がこのいずれかの値になっていれば正常です。

1 つ目と 2 つ目の jmp 命令を繰り返しているはずなので、何度か info registers コマンドを繰り返すと「RIP」が「0x110000」になったり「0x110002」になったりします。

また、メモリの内容を表示させることもできます。例えば、実行バイナリ本体が配置されている 0x110000 以降の内容を逆アセンブルして命令 2 個分を表示させるには以下の通りです。

```
(qemu) x/2i 0x110000
0x0000000000110000:  jmp    0x110002
0x0000000000110002:  jmp    0x110000
```

「x/2i」の「x」が「引数に指定されたメモリアドレスの内容を表示する」というコマンドで、「/」以降の「2i」がフォーマット指定です。「i」が「逆アセンブルしてアセンブラ表示」という表示形式を示しており、「2」はそれを「2 個分」表示することを示しています。なお、もちろん 16 進数等で表示させることもできます。詳しくは QEMU のドキュメントをご覧ください[*5]。

このようなビジーループのプログラムを実行しているとホスト PC 側の CPU 使用率が上昇していきます。レジスタ値やメモリの内容を確認する際は一旦 QEMU の実行を「stop」コマンドで止めておくと良いです。「cont(短縮表記 c)」コマンドで再開できます。

[*5] https://qemu.weilnetz.de/doc/qemu-doc.html#Commands

```
<一時停止>
(qemu) stop

<再開>
(qemu) cont

<再開(短縮表記)>
(qemu) c
```

確認後、QEMU を終了させる際は、QEMU モニター上では「quit」コマンドで終了できます。GUI の場合はウィンドウの [X] ボタンでも構いません。また、CUI の場合は「Ctrl-a のあと x」でも終了できます。

```
(qemu) quit
```

以上で、「機械語の構文理解」から「実行バイナリ作成」、「QEMU での動作確認」までのフルセットの流れが完了です。今後は、命令の簡単さや章ごとのテーマによって、フルセットで都度確認したりはしないですが、気になる場合はここで確認したように色々と実験してみてください。

この章では最後にあと 2 つ、無限ループプログラムに関連した命令を紹介します。

1.5　何もしない命令「nop」(命令 2)

今後のためにこの章ではあと 2 つ、簡単な命令をさくっと紹介します。

まずは、「nop」命令です。サンプルディレクトリは「012_nop」です。

nop 命令は「何もしない命令」です。前節では jmp 命令のジャンプ先の相対アドレスを色々と変えてみるために jmp 命令自体を間に入れてみたりしていましたが、そのような場合はこの命令を使うと良いです。

as_dump を使ってこの命令の機械語を確認します。

```
$ as_dump nop

/tmp/tmp.MrEmEUlxOC/a.out:     ファイル形式 elf64-x86-64


セクション .text の逆アセンブル:

0000000000000000 <.text>:
   0:   90                      nop
```

オペランドを取らない nop 命令のアセンブラ/機械語構文は表 1.4 の通りであると分かりました。

▼表 1.4: 構文 2: nop: 何もしない

アセンブラ	nop
機械語	90

例えば、「3 バイト分戻るジャンプ」を実験として試してみる際は、nop を使ってリスト 1.7 のように書けます。

▼リスト 1.7: 012_nop/sample.txt

```
0: 90           # nop
1: 90           # nop
2: 90           # nop
3: eb fb        # jmp -3
```

1.6　CPU を休ませる「hlt」(命令 3)

この章で紹介する最後の命令は「hlt」命令です。サンプルディレクトリは「013_hlt」です。

hlt 命令はシャットダウン等の意味合いで使われる「halt」の略です。この命令自体はシャットダウンという程ではなく、割り込みがあるまで CPU をスリープさせます。

本書で行うような OS レスのプログラミングの場合、戻る先が無いので、プログラムの最後では必ず無限ループを設置して処理を止めておく必要があります。その際、ビジーループだと CPU 負荷が上がってしまうので、hlt 命令で CPU を休ませるようにすると良いです。

hlt 命令も as_dump で機械語を確認します。

```
$ as_dump hlt

/tmp/tmp.5dlph8ymMx/a.out:     ファイル形式 elf64-x86-64


セクション .text の逆アセンブル:

0000000000000000 <.text>:
   0:   f4                      hlt
```

オペランドを取らない hlt 命令のアセンブラ/機械語構文をまとめると表 1.5 の通りです。

▼表 1.5: 構文 3: hlt: CPU をスリープさせる

アセンブラ	hlt
機械語	f4

例えばリスト 1.8 のような感じで使います。

▼リスト 1.8: 013_hlt/sample.txt

```
# 何らかの処理
90      # nop
90      # nop

# 無限Halt
f4      # hlt
eb fd   # jmp -1
```

hlt 命令による CPU のスリープ状態からは、割り込み等で起床するので、完全にやることがなくなった際は hlt 命令を実行し続けるように jmp 命令でループを入れておきます。

第2章 基本的な演算

この章では、基本的な演算命令の機械語における構文を紹介します。以降の章で使うことになる構文の紹介が主で、この章では章を通して作り上げるサンプルはありません。

CPU は、基本的にレジスタ上の値で演算を行うため、まず、任意のレジスタへ値を格納する方法を紹介し、その後、四則演算・論理演算等の基本的な演算の方法を紹介します。

なお、機械語コードは、オペランドのパターン (オペランドの数や種類) が変わると機械語のバイト列も変化します。この章では、以降の章でも使用するような代表的な構文のみ紹介します。アセンブラの構文も併せて紹介しますので、演算対象が異なる場合など、ここで紹介しない構文については、as_dump 等で確認してみてください。

2.1 データ格納命令 (mov)(命令 4)

♣ 汎用レジスタ

まず、一般的に CPU 命令で使用するレジスタを紹介します。

CPU が汎用的にデータ格納先として利用できるレジスタが「汎用レジスタ」です。QEMU モニターの `info registers` コマンドで表示されるレジスタの内、リスト 2.1 のレジスタが汎用レジスタです。

▼リスト 2.1: QEMU モニターの info registers で表示されるレジスタ

```
 RAX=0000000000110101 RBX=00000000bef67f02 RCX=0000000000000000 RDX=00000
›00000000000
 RSI=0000000000407180 RDI=00000000bfe9b018 RBP=00000000bff0eb10 RSP=00000
›00000210000
 R8 =00000000bff0e9dc R9 =0000000000000078 R10=00000000bfe6e080 R11=00000
›000e7dc4657
 R12=00000000bef69540 R13=00000000bef69548 R14=00000000bff24b60 R15=00000
›000bef6a018
```

RAX から RDI、R8 から R15 までの計 16 個のレジスタが並んでいます。これらはそれぞれ 64 ビットの大きさがあります。なお、一部のレジスタは、64 ビット幅の中の

一部分にのみアクセスすることが可能です。レジスタの下位 32 ビットへのアクセスや、さらにその中の下位 16 ビットへのアクセス、その 16 ビットの内の上位/下位それぞれ 8 ビットずつへのアクセスといったことができます。アクセスできるビット幅には名前が付いており、一覧にすると表 2.1 の通りです。

▼表 2.1: 汎用レジスター一覧

64bit 全体	下位 32bit	下位 16bit	下位 16bit の上位 8bit	下位 16bit の下位 8bit
RAX	EAX	AX	AH	AL
RCX	ECX	CX	CH	CL
RDX	EDX	DX	DH	DL
RBX	EBX	BX	BH	BL
RSP	ESP	SP	-	-
RBP	EBP	BP	-	-
RSI	ESI	SI	-	-
RDI	EDI	DI	-	-
R8	R8D	-	-	-
R9	R9D	-	-	-
R10	R10D	-	-	-
R11	R11D	-	-	-
R12	R12D	-	-	-
R13	R13D	-	-	-
R14	R14D	-	-	-
R15	R15D	-	-	-

8 ビットの場合のみ、16 ビットアクセス時の上位側 (H) と下位側 (L) とで別々にアクセスできます。16 ビット/32 ビットのアクセスの際は、「32 ビット幅の上位 16 ビット」や「64 ビット幅の上位 32 ビット」というアクセスはできません。

なお、「R8」から「R15」のレジスタ (32 ビットにおける「R8D」から「R15D」) は本書では使用しないため特に紹介しません。使い方はその他のレジスタ同様なので、使ってみたい場合は各自で as_dump 等から機械語を確認してみてください。

♣ 即値をレジスタへ格納する機械語コードを見てみる

レジスタへの値の格納は「`mov`」命令で行えます。ここでは 8 ビットの即値[*1]を 8 ビットのレジスタへ格納する命令を見てみます。mov 命令はバリエーションが多いの

[*1] 値そのものを示すオペランドです。アセンブラでは先頭に「$」を付けると即値を表すオペランドになります。

で、その他の構文は、必要になったらその都度確認することにします。

例えば「0x12」という 8 ビット即値を「AL」レジスタへ格納する場合、アセンブラとしては以下のコードになります。

▼リスト 2.2:

```
mov     $0x12, %al
```

as_dump を使って機械語コードを確認してみます。

```
$ as_dump mov \$0x12,%al

/tmp/tmp.zHlktVD4V2/a.out:     ファイル形式 elf64-x86-64


セクション .text の逆アセンブル:

0000000000000000 <.text>:
   0:   b0 12                   mov    $0x12,%al
```

「mov $0x12,%al」は機械語では「b0 12」であることが分かりました。2 バイト目の「12」がオペランドで指定した即値「0x12」です。この値を変えれば、AL へ格納する値を変更できます。

また、機械語の 1 バイト目の「b0」が「AL へ格納する」事を示しており、ここを変更することで別のレジスタへ値を格納できます。

♣ 他のレジスタへ格納する命令も見てみる

例えば、CL レジスタ・DL レジスタへ格納する命令のアセンブラと機械語は表 2.2 の通りです。

▼表 2.2: mov: 即値 0x12 をレジスタ CL・DL へ格納

CL レジスタ	
アセンブラ	mov $0x12, **%cl**
機械語	**b1** 12
DL レジスタ	
アセンブラ	mov $0x12, **%dl**
機械語	**b2** 12

機械語の 1 バイト目に着目すると、AL の時は「b0」、CL の時は「b1」、DL の時は「b2」と、1 ずつ増えています。

このように、レジスタをオペランドに取る機械語は、レジスタを示すバイトを 1 ずつ変化させることで別のレジスタを指定することができ、レジスタには並び順があります。

8 ビットレジスタの並び順は他の命令でも共通で以下の通りです。

▼表 2.3: 機械語における 8 ビットレジスタ並び順

reg_ofs	レジスタ
0	AL
1	CL
2	DL
3	BL
4	AH
5	CH
6	DH
7	BH

「reg_ofs」は、レジスタを示すオペランド部分のベース値に対するオフセットを示します。今回の場合ベースは「b0」で、そこにオフセット値を足すことで任意の 8 ビットレジスタを選択できます。今後、レジスタをオペランドにとる構文の表現に使用します。

♣ 構文: mov: 即値 (8bit) をレジスタ (8bit) へ格納

以上をまとめると、構文としては表 2.4 の通りです。

▼表 2.4: 構文 4: mov: 即値 (8bit) をレジスタ (8bit) へ格納

アセンブラ	mov 即値 (8bit), レジスタ (8bit)
機械語	b0+reg_ofs 即値 (8bit)

♣ 構文: mov: 即値 (16bit) をレジスタ (16bit) へ格納

続いて、16 ビット即値を 16 ビットレジスタへ格納する場合も見てみます。

```
$ as_dump mov \$0x1234,%ax

/tmp/tmp.eTaoyVyqMf/a.out:     ファイル形式 elf64-x86-64
```

```
セクション .text の逆アセンブル:

0000000000000000 <.text>:
   0:   66 b8 34 12             mov    $0x1234,%ax
```

機械語「66 b8 34 12」の最後の 2 バイト「34 12」が即値 0x1234 を示しています。「0x12」と「0x34」がひっくり返っているのは、Intel 命令のバイトオーダーがリトルエンディアン[*2]であるためです。そして、対象とするレジスタは 2 バイト目の「b8」が示しています。

なお、16 ビットレジスタの並び順は以下の通りです。

▼表 2.5: 16 ビットレジスタ並び順

reg_ofs	レジスタ
0	AX
1	CX
2	DX
3	BX
4	SP
5	BP
6	SI
7	DI

以上をまとめると構文としては以下の通りです。

▼表 2.6: 構文 5: mov: 即値 (16bit) をレジスタ (16bit) へ格納

アセンブラ	mov 即値 (16bit), レジスタ (16bit)
機械語	66 b9+reg_ofs 即値 (16bit)

32 ビットと 64 ビットについては、即値をレジスタへ格納する機会が本書では無く、確認方法等もこれまで同様なので得に紹介しません。

ただ、それぞれのレジスタは他の命令で使うことがあるので、並び順だけ他のビットも合わせて紹介しておきます。

[*2] バイトの並び順 (バイトオーダー) を、下の位から順に 1 バイトずつ並べること。

▼表 2.7: レジスタの並び順まとめ

reg_ofs	64bit	32bit	16bit	8bit
0	RAX	EAX	AX	AL
1	RCX	ECX	CX	CL
2	RDX	EDX	DX	DL
3	RBX	EBX	BX	BL
4	RSP	ESP	SP	AH
5	RBP	EBP	BP	CH
6	RSI	ESI	SI	DH
7	RDI	EDI	DI	BH

2.2　四則演算

それでは、続いて四則演算を機械語で記述する方法を紹介します。

四則演算の命令は、一部を除いて基本的に 2 つのオペランドを取り、第 1 オペランドと第 2 オペランドで演算した結果を第 2 オペランドのレジスタ (あるいはメモリアドレス先) へ格納するという挙動になります。

分かりやすい代表的な例として、8 ビット即値と 8 ビットレジスタで計算する場合を中心に紹介します。その他のパターンも同様の方法で構文を確認できますので、興味があれば適宜確認してみてください。

♣ 和 (add)(命令 5)

和を計算するのが add 命令です。2 つのオペランドを取り、第 1 オペランドと第 2 オペランドの和を第 2 オペランドへ格納します。

例えば、8 ビットレジスタ AL へ即値 0x12 を加えた結果を AL へ格納する場合、アセンブラでは「add $0x12,%al」となり、その機械語コードは以下の通りです。

```
$ as_dump add \$0x12,%al

/tmp/tmp.Ebxte7x4HM/a.out:     ファイル形式 elf64-x86-64


セクション .text の逆アセンブル:

0000000000000000 <.text>:
   0:   04 12                    add    $0x12,%al
```

「12」は即値で、「04」が対象レジスタを決めるベース値に見えます。
CL レジスタの場合はどうなるのか見てみましょう。

```
$ as_dump add \$0x12,%cl

/tmp/tmp.t62F0yZWGi/a.out:     ファイル形式 elf64-x86-64


セクション .text の逆アセンブル:

0000000000000000 <.text>:
   0:   80 c1 12                add    $0x12,%cl
```

機械語コードの構文ががらっと変わりました。
実は、「04 XX」という表現は、8 ビット即値を AL に加算する場合にだけ使えるもので、8 ビット即値と 8 ビットレジスタでの加算命令の「シンタックスシュガー」のようなものです。

▼表 2.8: 構文 6: add: 即値 (8bit) を AL へ加算

アセンブラ	add 即値 (8bit), %al
機械語	04 即値 (8bit)

AL 以外の 8 ビットレジスタへ 8 ビット即値を加算する場合の構文は以下の通りです。

▼表 2.9: 構文 7: add: 即値 (8bit) をレジスタ (8bit) へ加算

アセンブラ	add 即値 (8bit), レジスタ (8bit)
機械語	80 c0+reg_ofs 即値 (8bit)

das_dump 逆アセンブルしてみると分かりますが、構文 7 で AL レジスタを使うこともできるようです。

```
$ das_dump 80 c0 12

/tmp/tmp.NLv4saxnSr/a.bin:     ファイル形式 binary


セクション .data の逆アセンブル:
```

```
0000000000000000 <.data>:
   0:   80 c0 12                add    $0x12,%al
```

ただ、せっかく 1 バイト減らせる表現があるので、AL レジスタを使う際は構文 6 の表現を使うほうが良いです。このように、命令によっては特定のパターンのみ命令のバイト数を減らす構文があります。

♣ 差 (sub)(命令 6)

差は「sub」命令です。
和 (add) と同様なので、さくっと構文をまとめます。
sub も add と同様に AL レジスタでのみ使える構文があります。以下の通りです。

▼表 2.10: 構文 8: sub: 即値 (8bit) を AL レジスタから減算

アセンブラ	sub 即値 (8bit), %al
機械語	2c 即値 (8bit)

例えば、8 ビット即値 0x12 を 8 ビットレジスタ AL から減算する場合、以下のようになります。

▼表 2.11: 例: sub: 0x12 を AL から減算

アセンブラ	sub $0x12, %al
機械語	2c 12

次に、AL 以外でも使える構文です。

▼表 2.12: 構文 9: sub: 即値 (8bit) をレジスタ (8bit) から減算

アセンブラ	sub 即値 (8bit), レジスタ (8bit)
機械語	80 e8+reg_ofs 即値 (8bit)

例えば、8 ビット即値 0x12 を 8 ビットレジスタ CL から減算する場合、以下のようになります。

▼表 2.13: 例: sub: 0x12 を CL から減算

アセンブラ	sub $0x12, %cl
機械語	80 e9 12

♣ 積 (imul)(命令 7)

オペランドが 1 つの場合

オペランドが 1 つの指定ができます。オペランドが 1 つのパターンで指定できるのはレジスタ名で、例えば 8 ビットレジスタを指定した場合、「AL * 指定されたレジスタ → AX」という命令になります。暗黙の内に AL レジスタとの掛け算を行い、結果を一回り大きい 16 ビットの AX レジスタへ格納します。単一オペランド時のレジスタ幅に対する挙動を表 2.14 にまとめます。

▼表 2.14: imul: 単一オペランドの挙動まとめ

レジスタ幅	挙動
8 ビット	AL * オペランド → AX
16 ビット	AX * オペランド → DX:AX
32 ビット	EAX * オペランド → EDX:EAX
64 ビット	RAX * オペランド → RDX:RAX

オペランドが 16 ビット以上のレジスタの場合、積は上位ビットと下位ビットが D のレジスタと A のレジスタに分けて格納されます。例えば、16 ビットの演算で積が「0x1234 5678」であった場合、「0x1234」が DX へ、「0x5678」が AX へ格納されます。

オペランドに 8 ビットレジスタを指定した場合、構文としては以下の通りです。

▼表 2.15: 構文 10: imul: AL * レジスタ (8bit) を AX へ格納

アセンブラ	imul レジスタ (8bit)
機械語	f6 e8+reg_ofs

例えば、AL の二乗を計算するような場合、以下のようになります。

▼表 2.16: 例: imul: AL * AL を AX へ格納

アセンブラ	imul %al
機械語	f6 e8

オペランドが 2 つの場合

オペランドが 2 つの場合の挙動は、和や差と同じで「第 1 オペランドと第 2 オペランドの演算結果を第 2 オペランドへ格納」です。

ただし、第 1 オペランドに即値を使う構文はありません。

例えば、第 1 オペランドに「AX」を、第 2 オペランドに「CX」を指定した場合は、「AX * CX」の結果が「CX」へ格納されます。(単一オペランドの時のように「一回り大きなレジスタへ格納」とはなりません。)

構文としては以下の通りです。

▼表 2.17: 構文 11: imul: 第 1 * 第 2 オペランド (16bit レジスタ) を第 2 オペランドへ格納

アセンブラ	imul レジスタ (16bit,src), レジスタ (16bit,dst)
機械語	66 0f af c0+reg_dst_src

「reg_dst_src」は「第 1 オペランド (src)」と「第 2 オペランド (dst)」のレジスタの組み合わせを示すオフセット値で、以下の通りです。

▼表 2.18: 16 ビットレジスタの reg_dst_src 一覧

dst/src	AX	CX	DX	BX	SP	BP	SI	DI
AX	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
CX	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
DX	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
BX	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
SP	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
BP	0x28	0x29	0x2a	0x2b	0x2c	0x2d	0x2e	0x2f
SI	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
DI	0x38	0x39	0x3a	0x3b	0x3c	0x3d	0x3e	0x3f

各列が「第 1 オペランド (src)」で、各行が「第 2 オペランド (dst)」を示す表です。

例に挙げた「AX * CX を CX へ格納する」場合、以下のようになります。

▼表 2.19: 例: AX * CX を CX へ格納

アセンブラ	imul %ax, %cx
機械語	66 0f af c8

なお、imul 命令のオペランド 2 つの場合では、第 1・第 2 いずれにも 8 ビットレジスタを使用する構文はありません。

オペランドが 3 つの場合

imul はオペランドが 3 つの指定もできます。その場合、第 1 オペランドは「即値」、第 2・第 3 オペランドが「レジスタ」での指定となり、「第 1 オペランドと第 2 オペランドの演算結果を第 3 オペランドへ格納」という挙動になります。

なお、第 2・第 3 オペランドのレジスタ幅に応じて第 1 オペランドで指定できる即値のビット数が異なります。まとめると表 2.20 の通りです。

▼表 2.20: オペランド 3 つの場合の即値・レジスタ幅の組み合わせまとめ

第 1 オペランド (即値)	第 2・第 3 オペランド (レジスタ)
8 ビット	16 ビット
8 ビット	32 ビット
8 ビット	64 ビット
16 ビット	16 ビット
32 ビット	32 ビット
32 ビット	64 ビット

例えば、第 1 オペランドに 8 ビット即値「0x12」を、第 2 オペランドに 16 ビットレジスタ「AX」を、第 3 オペランドに 16 ビットレジスタ「CX」を指定した場合、「0x12 * AX」の結果が「CX」へ格納されます。

構文としては以下の通りです。

▼表 2.21: 構文 12: imul: 第 1 オペランド * 第 2 オペランドを第 3 オペランドへ格納

アセンブラ	imul 即値 (8bit), レジスタ (16bit,src), レジスタ (16bit,dst)
機械語	66 6b c0+reg_dst_src 即値 (8bit)

例に挙げた「0x12 * AX を CX へ格納する」場合、以下のようになります。

▼表 2.22: 例: 0x12 * AX を CX へ格納

アセンブラ	imul $0x12, %ax, %cx
機械語	66 6b c8 12

♣ 商と剰余 (idiv)(命令 8)

商と剰余を求める「idiv」命令の場合、「オペランドが 1 つ」のパターンの命令しかありません。単一オペランドでレジスタを指定することができ、指定したレジスタのビット幅に応じて表 2.23 の挙動となります。

▼表 2.23: idiv: 指定したレジスタのビット幅に対する挙動まとめ

オペランドレジスタ幅	挙動
8 ビット	「AX/オペランド」の商を AL へ剰余を AH へ格納
16 ビット	「DX:AX/オペランド」の商を AX へ剰余を DX へ格納
32 ビット	「EDX:EAX/オペランド」の商を EAX へ剰余を EDX へ格納
64 ビット	「RDX:RAX/オペランド」の商を RAX へ剰余を RDX へ格納

そして、オペランドが 8 ビットレジスタの場合の構文は以下の通りです。

▼表 2.24: 構文 13: idiv: AX/レジスタ (8bit) の商を AL へ剰余を AH へ格納

アセンブラ	idiv レジスタ
機械語	f6 f8+reg_ofs

例えば、AX/AL の商を AL へ剰余を AH へ格納する場合、以下のようになります。

▼表 2.25: 例: AX/AX の商 (1) を AL へ剰余 (0) を AH へ格納

アセンブラ	idiv %al
機械語	f6 f8

なお、0 で割る演算を行った場合、CPU で「0 除算」の例外が発生します。本書では

例外時のジャンプ先 (ハンドラ) の設定を行っていないので、例外発生時の挙動は未定義となりますので、ご注意ください[*3]。

2.3 論理演算

続いて、論理演算の方法を AND・OR・XOR・NOT の 4 つについて紹介します。

♣ AND(命令 9)

オペランドの数は 2 つのパターンのみで、第 1 オペランドと第 2 オペランドの演算結果を第 2 オペランドへ格納します。

8 ビット即値と 8 ビットレジスタの AND の結果を 8 ビットレジスタへ格納する場合の構文を紹介します。

第 2 オペランドに指定するレジスタが AL の場合、1 バイト少ない機械語表現があり、構文は以下の通りです。

▼表 2.26: 構文 14: and: 即値 (8bit) AND AL を AL へ格納

アセンブラ	and 即値 (8bit), %al
機械語	24 即値 (8bit)

例えば、8 ビット即値 0x12 と AL の AND を AL へ格納する場合、以下のようになります。

▼表 2.27: 例: 0x12 と AL の AND を AL へ格納

アセンブラ	and $0x12, %al
機械語	24 12

そして、第 2 オペランドのレジスタが AL 以外でも使える構文は表 2.28 の通りです。

例えば、即値 0x12 とレジスタ CL の AND をレジスタ CL へ格納する場合の機械語コードは表 2.29 の通りです。

[*3] QEMU 上なので試してみても何も被害は無いとは思います。何が書かれているか分からないところへジャンプして CPU がそこに書かれたバイト列を命令と解釈して突き進んで行きます。

▼表 2.28: 構文 15: and: 即値 (8bit) AND レジスタ (8bit) をレジスタ (8bit) へ格納

アセンブラ	and 即値 (8bit), レジスタ (8bit)
機械語	80 e0+reg_ofs 即値 (8bit)

▼表 2.29: 例: 0x12 と CL の AND を CL へ格納

アセンブラ	and $0x12, %cl
機械語	80 e1 12

♣ OR(命令 10)

or 命令も and 命令同様、オペランドの数は 2 つのパターンのみです。

「8 ビット即値 (第 1 オペランド) OR 8 ビットレジスタ (第 2 オペランド)」の結果を第 2 オペランドへ格納する構文を紹介します。

and 命令同様に、第 2 オペランドのレジスタが AL の場合には 1 バイト短い構文があります。

▼表 2.30: 構文 16: or: 即値 (8bit) OR AL を AL へ格納

アセンブラ	or 即値 (8bit), %al
機械語	0c 即値 (8bit)

例えば、8 ビット即値 0x12 とレジスタ AL の OR をレジスタ AL へ格納する場合は以下のようになります。

▼表 2.31: 例: 0x12 と AL の OR を AL へ格納

アセンブラ	or $0x12, %al
機械語	0c 12

そして、第 2 オペランドのレジスタが AL 以外でも使える構文は表 2.32 の通りです。

例えば、即値 0x12 とレジスタ CL の OR をレジスタ CL へ格納する場合の機械語コードは表 2.33 の通りです。

▼表 2.32: 構文 17: or: 即値 (8bit) OR レジスタ (8bit) をレジスタ (8bit) へ格納

アセンブラ	or 即値 (8bit), レジスタ (8bit)
機械語	80 c8+reg_ofs 即値 (8bit)

▼表 2.33: 例: 0x12 と CL の OR を CL へ格納

アセンブラ	or $0x12, %cl
機械語	80 c9 12

♣ XOR(命令 11)

XOR もオペランドの数は 2 つのパターンのみです。

「8 ビット即値 (第 1 オペランド) XOR 8 ビットレジスタ (第 2 オペランド)」の結果を第 2 オペランドへ格納する構文を紹介します。

and 命令・or 命令同様に、第 2 オペランドのレジスタが AL の場合には 1 バイト短い構文があります。

▼表 2.34: 構文 18: xor: 即値 (8bit) XOR AL を AL へ格納

アセンブラ	xor 即値 (8bit), %al
機械語	34 即値 (8bit)

例えば、8 ビット即値 0x12 とレジスタ AL の XOR をレジスタ AL へ格納する場合は以下のようになります。

▼表 2.35: 例: 0x12 と AL の XOR を AL へ格納

アセンブラ	xor $0x12, %al
機械語	34 12

そして、第 2 オペランドのレジスタが AL 以外でも使える構文は表 2.36 の通りです。

例えば、即値 0x12 とレジスタ CL の XOR をレジスタ CL へ格納する場合の機械語コードは表 2.37 の通りです。

▼表 2.36: 構文 19: xor: 即値 (8bit) XOR レジスタ (8bit) をレジスタ (8bit) へ格納

アセンブラ	xor 即値 (8bit), レジスタ (8bit)
機械語	80 f0+reg_ofs 即値 (8bit)

▼表 2.37: 例: 0x12 と CL の XOR を CL へ格納

アセンブラ	xor $0x12, %cl
機械語	80 f1 12

♣ NOT(命令 12)

NOT は、オペランドの数は 1 つのみです。
「オペランドで指定された 8 ビットレジスタの NOT をオペランドのレジスタへ格納する」場合、構文は以下の通りです。

▼表 2.38: 構文 20: not: レジスタ (8bit) の NOT をレジスタ (8bit) へ格納

アセンブラ	not レジスタ (8bit)
機械語	f6 d0+reg_ofs

例えば、レジスタ AL の NOT をレジスタ AL へ格納する場合は以下のようになります。

▼表 2.39: 例: AL の NOT を AL へ格納

アセンブラ	not %al
機械語	f6 d0

2.4　シフト演算 (命令 13-15)

シフト演算の命令は以下の 4 つです。

- SAL: 算術左シフト
- SAR: 算術右シフト

- SHL: 論理左シフト
- SHR: 論理右シフト

全てオペランドの数は 2 つです。
各命令について、オペランドを使って実際に行われる演算は以下の通りです。

▼表 2.40: 各シフト演算の挙動

sal	第 1 オペランドの回数分、第 2 オペランドへ 2 を掛ける
sar	第 1 オペランドの回数分、第 2 オペランドを 2 で割る (符号有)
shl	第 1 オペランドの回数分、第 2 オペランドへ 2 を掛ける
shr	第 1 オペランドの回数分、第 2 オペランドを 2 で割る (符号無)

sal 命令と shl 命令は挙動が同じで、どちらでアセンブラを書いても同じ機械語が生成されます。そのため、機械語として実質存在するシフト命令は 3 つです。
なお、シフト命令は、第 1 オペランドに即値あるいは CL レジスタしか指定できません。
「第 1 オペランドを即値 (8bit)、第 2 オペランドをレジスタ (8bit)」の場合、構文は以下の通りです。(shl は sal と同一なので省略)

▼表 2.41: 構文 21-23: sal/sar/shr: 第 1 オペランド=即値 (8bit)、第 2 オペランド=レジスタ (8bit)

アセンブラ	sal 即値 (8bit), レジスタ (8bit)
機械語	c0 e0+reg_ofs 即値 (8bit)
アセンブラ	sar 即値 (8bit), レジスタ (8bit)
機械語	c0 f8+reg_ofs 即値 (8bit)
アセンブラ	shr 即値 (8bit), レジスタ (8bit)
機械語	c0 e8+reg_ofs 即値 (8bit)

例えば、AL レジスタに対して即値 3 で SAL/SAR/SHR を実施する場合の機械語コードは表 2.42 の通りです。
前述の通り、算術左シフトと論理左シフトは挙動は同じなので、機械語としては同じバイナリ列になっています。

▼表 2.42: 例: AL に対して即値 3 で SAL/SAR/SHR を実施

アセンブラ	sal $3,%al
機械語	c0 e0 03
アセンブラ	sar $3,%al
機械語	c0 f8 03
アセンブラ	shl $3,%al
機械語	c0 e0 03
アセンブラ	shr $3,%al
機械語	c0 e8 03

2.5　インクリメント/デクリメント (命令 16-17)

インクリメント (inc) とデクリメント (dec) は共に取れるオペランドの数は一つのみで、構文も似たような感じです。

8 ビットレジスタをインクリメント/デクリメントする場合の構文は以下の通りです。

▼表 2.43: 構文 24-25: inc/dec: レジスタ (8bit) をインクリメント/デクリメント

アセンブラ	inc レジスタ (8bit)
機械語	fe c0+reg_ofs
アセンブラ	dec レジスタ (8bit)
機械語	fe c8+reg_ofs

例えば、AL レジスタをインクリメント/デクリメントする場合は以下のようになります。

▼表 2.44: 例: AL をインクリメント/デクリメント

アセンブラ	inc %al
機械語	fe c0
アセンブラ	dec %al
機械語	fe c8

第3章 シリアルドライバを作る

デバッグ用にテキストの表示などが行えると便利です。ここではシリアル通信[*1]のドライバを作成し、シリアル通信により文字の送受信を行うことで、QEMU のシリアル通信用の画面で文字の表示と入力された文字の取得を行ってみます[*2]。

3.1 文字を送信して画面表示

それでは早速、シリアル通信による文字の送信を試してみましょう。この節のサンプルディレクトリは「031_ser_tx」です。

♣ シリアル通信を行うには

シリアル通信には専用のコントロール IC(コントローラ) があります。コントローラにも CPU 同様にレジスタがあり、そのレジスタへ値を書き込んだり、逆に値を読み出したりすることで、シリアル通信でデータを送信したり受信したりできます。

外部の IC が持つレジスタは、基本的に「IO アドレス空間」というアドレス空間からアクセスします。これは、ポインタ等でアクセスするような通常のメモリのアドレス空間とは別で、「`in`」命令と「`out`」命令という専用の命令を使用して、そのアドレス空間へアクセスします。

シリアルでのデータ送信は「`THR(Transmitter Holding Register)`」というレジスタで行います。IO アドレスなどについては表 3.1 の通りです[*3]。

レジスタサイズが 1 バイトなので、1 文字ずつ書き込み、1 文字ずつ送信します。

[*1] 送信 (Tx) と受信 (Rx) をそれぞれ 1 本の信号線でやり取りする通信方式です。Windows では「COM ポート」と呼ばれていたり、Linux では「シリアルポート (ttySx)」等と呼ばれているやつです。昔の PC には「D-sub9 ピン」と呼ばれるコネクタが搭載されていて、「RS-232C」規格のケーブルを使ってシリアル通信を行うことができました。

[*2] ここはあくまでもシリアル通信による文字の送受信なので、キーボードドライバ自体は後の章で作成します。

[*3] マシンには複数のシリアルポートが搭載されていたりするのですが、これは 1 つ目のシリアルポートのアドレスです。複数のシリアルポートを使う予定も無いので本書では常に 1 つ目のシリアルポートを使います。

▼表 3.1: THR について

IO アドレス	0x03f8
レジスタサイズ	1 バイト

♣ IO アドレス空間のレジスタへデータを書き込む「out」(命令 18)

それでは、out 命令を使って、文字をシリアル通信で送ってみます。
1 バイトのデータ送信を行う際の out 命令の構文は以下の通りです。

▼表 3.2: 構文 26: out: AL を DX が指すアドレス先へ書き込む

アセンブラ	out %al, %dx
機械語	ee

使用できるレジスタは固定です。今回の場合、「AL」に送信したい文字の ASCII コードを格納し、「DX」へ THR の IO アドレス (0x03f8) を格納した状態で out 命令を実施することで、データ送信を行えます。
8 ビットレジスタと 16 ビットレジスタへそれぞれのビット幅の即値を格納する構文は 2 章で紹介しました。
再掲すると以下の通りです。

▼表 3.3: 構文 4: mov: 即値 (8bit) をレジスタ (8bit) へ格納 (再掲)

アセンブラ	mov 即値 (8bit), レジスタ (8bit)
機械語	b0+reg_ofs 即値 (8bit)

▼表 3.4: 構文 5: mov: 即値 (16bit) をレジスタ (16bit) へ格納 (再掲)

アセンブラ	mov 即値 (16bit), レジスタ (16bit)
機械語	66 b9+reg_ofs 即値 (16bit)

AL レジスタの reg_ofs は「0」で、DX レジスタの reg_ofs は「2」なので、AL レジスタと DX レジスタへの値格納までを sample.txt へ記述するとリスト 3.1 の通りです。

▼リスト 3.1: 031_ser_tx/sample.txt

```
# ASCII文字'A'(0x41)をシリアルポートへ送信
b0 41           # mov $0x41,%al
66 ba f8 03     # mov $0x03f8,%dx
```

ここでは文字 A(0x41) を送信してみることにしました。

そして、out 命令を使い、AL に格納した文字を DX が指す THR へ書き込んでみます (リスト 3.2)。

▼リスト 3.2: 031_ser_tx/sample.txt

```
# ASCII文字'A'(0x41)をシリアルポートへ送信
b0 41           # mov $0x41,%al
66 ba f8 03     # mov $0x03f8,%dx
ee              # out %al,%dx   (追加)

# 無限Halt                        (追加)
f4              # hlt           (追加)
eb fd           # jmp -1        (追加)
```

out 命令の後に hlt を使った無限ループを設置しました。

♣ 動作確認

kernel.bin を生成し、run_qemu スクリプトで実行してみてください。

シリアル送信した文字は、QEMU を GUI で起動した場合「Ctrl-Alt-3」キーでシリアル通信用の画面へ切り替えることで確認できます。CUI で起動した場合、標準でシリアル通信画面になっているのでそのまま表示されます。

文字「A」が表示されれば成功です。

```
$ run_qemu fs -nographic
A
```

3.2　コントローラの状態を確認して送信する

前節では、コントローラの状態確認無しで THR へ書き込みを行っていました。THR はコントローラが持つ送信バッファです。コントローラの側で送信バッファにデータを受け入れる準備ができていない時にこのバッファへ書き込みを行ってしまうと、最悪の場合、未送信のデータを上書きして消してしまう恐れがあります。

シリアル通信のコントローラには「送信バッファが空になった」事を示すフラグがあるため、ここではこのフラグを確認してから THR へ書き込むように変更してみます。

この節のサンプルディレクトリは「032_ser_tx_chk_flg」です。

♣ Line Status Register(LSR)

コントローラのステータスは「Line Status Register(LSR)」というレジスタで確認できます。

▼表 3.5: LSR について

IO アドレス	0x03fd
レジスタサイズ	1 バイト

LSR は各ビットがコントローラのステータスを示しており、各ビットの意味は以下の通りです。

▼表 3.6: LSR の各ビットについて

ビット	意味
7	受信バッファに何らかのエラーが発生
6	送信バッファが空で内部の送信処理も全て完了
5	送信バッファが空 (データ受け入れ可能)
4	ブレークシグナルを受信
3	フレームエラー
2	パリティエラー
1	オーバーランエラー
0	受信バッファに 1 つ以上のデータがある

使うのはビット 5 です。このビットがセットされていれば送信バッファへデータ書き込みを行って問題ありません。

♣ IO アドレス空間のレジスタ値取得「in」(命令 19)

それでは、LSR の値を読み出す処理を追加してみます。

IO アドレス空間上のレジスタの値を取得する命令は「in」命令で、構文は表 3.7 の通りです。

試しに、単体のソースコード「read_lsr.txt」を作り、LSR の値を読み出してみます (リスト 3.3)。

▼表 3.7: 構文 27: in: DX が指すレジスタ値を AL へ格納

アセンブラ	in %dx, %al
機械語	ec

▼リスト 3.3: 032_ser_tx_chk_flg/read_lsr.txt

```
# シリアルポートのステータス取得
66 ba fd 03     # mov $0x03fd,%dx
ec              # in %dx,%al

# 無限Halt
f4              # hlt
eb fd           # jmp -1
```

リスト 3.3 を実行し、QEMU モニターでレジスタ値を確認すると以下の通りでした。

```
(qemu) info registers
RAX=0000000000110060 RBX=00000000bef67f98 RCX=0000000000000000 RDX=00000000000›
›003fd
RSI=0000000000407180 RDI=00000000bfe9b018 RBP=00000000bff0eb10 RSP=00000000002›
›10000
R8 =00000000bff0e9dc R9 =0000000000000078 R10=00000000bfe6e080 R11=00000000e7d›
›c4657
```

RAX レジスタの最下位 1 バイト (AL レジスタ) が 0x60 になっています。ビット 6 とビット 5 がセットされているので、このステータス値なら、送信して問題無さそうです。

♣ フラグの待ち方「and」/「je」(命令 20)

新たに 2 つの命令を使ってビットがセットされるまで待つ処理を実装します。流れとしては以下の 2 ステップです。

1. and 命令: AL(LSR の値) と 0x20 で AND を取る (AL のビット 5 以外を全て 0 にする)
2. je 命令: 1. の演算結果が 0 の場合、in 命令まで戻る (再度 LSR 取得)

新たに登場した「`je`」命令は、直前の演算結果に応じてジャンプするか否かが決まる「条件付きジャンプ命令」の 1 つです。CPU では、数値の演算を行うと、その結果に応じて CPU 内の「フラグレジスタ」と呼ばれるレジスタの各ビットが変化します。条件付きジャンプ命令は、このフラグレジスタの状態に応じてジャンプするか否かが決まる命令です。

je は「Jump if Equal」の略です。関係するのはフラグレジスタの中の「ゼロフラグ」のビットで、je 命令の実行時、このビットがセットされていればオペランドで示した相対アドレスへジャンプし、ビットがセットされていなければ、ジャンプはせずに次の命令へ進みます。

上記の 2 ステップの場合、ステップ 1 の AND によって、LSR のビット 5 がセットされていた場合のみステップ 2 で次の命令へ進むようになり、LSR のビット 5 がセットされていなければ LSR 値を取得する in 命令まで戻ることになります。

je 命令の構文は jmp 命令と同じで以下の通りです。

▼表 3.8: 構文 28: je: ゼロフラグがセットされていればジャンプ

アセンブラ	je <ラベルあるいは".">
機械語	74 fe+rel_addr

♣ フラグレジスタの動きを見てみる

ここで、フラグレジスタの動きを見てみます。

例えば、al に 0 を格納し、0x20 と and を取るだけのコードを作ってみます (リスト 3.4)。

▼リスト 3.4: 032_ser_tx_chk_flg/zf.txt

```
# 0x00と0x20のANDでゼロフラグを立てる
b0 00   # mov $0x00,%al
24 20   # and $0x20,%al

# 無限Halt
f4      # hlt
eb fd   # jmp -1
```

実行し、QEMU モニターでレジスタ値を表示させてみます。

```
(qemu) info registers
RAX=0000000000110000 RBX=00000000bef67f98 RCX=0000000000000000 RDX=00000000000
00000
RSI=0000000000407180 RDI=00000000bfe9b018 RBP=00000000bff0eb10 RSP=00000000002
10000
R8 =00000000bff0e9dc R9 =0000000000000078 R10=00000000bfe6e080 R11=00000000e7d
c4657
R12=00000000bef69540 R13=00000000bef69548 R14=00000000bff24b60 R15=00000000bef
6a018
RIP=0000000000110005 RFL=00000046 [---Z-P-] CPL=0 II=0 A20=1 SMM=0 HLT=1
```

「RFL=」に書かれているのがフラグレジスタの値です。すぐ右側の"[]"の中にどんなフラグが設定されているかが現れていて、"Z"が「ゼロフラグ」の設定を示しています。ゼロフラグは演算結果がゼロだった場合に設定されるフラグです。今回の場合、0x00と 0x20 の AND は 0x00 になるため、このフラグが設定されました。

この状態で「je」命令を実行するとオペランドで指定した相対アドレスへジャンプします。

フラグレジスタの算術演算に関わるビット (下位 8 ビット) の一覧を紹介しておきます (表 3.9)。

▼表 3.9: フラグレジスター覧

ビット	意味
7	符号フラグ。演算結果がマイナスの場合にセットされる
6	ゼロフラグ。演算結果がゼロの場合セットされる
5	予約
4	調整フラグ。2 進化 10 進 (BCD) でのキャリー・ボローでセットされる
3	予約
2	パリティフラグ。演算結果最下位 8 ビットの 1 の数が偶数の場合セット
1	予約
0	キャリーフラグ。キャリー・ボローでセットされる

フラグレジスタ自体は 64 ビット CPU の場合、64 ビットの大きさがあります。ここでは算術演算に関わるもののみ紹介しました。今後使用するビットは登場した際に改めて紹介します[*4]。

♣ LSR のビット 5 を待つ処理を実装

以上をまとめると、前節の機械語コードにフラグ確認の処理を追加するとリスト 3.5 のようになります。

▼リスト 3.5: 032_ser_tx_chk_flg/sample.txt

```
# 追加(ここから)
# シリアルポートのステータス取得
00: 66 ba fd 03 # mov $0x03fd,%dx
04: ec          # in %dx,%al
```

[*4] フラグレジスタについて詳しくは SDM の Volume 1 か、Wikibooks の記事が分かりやすいです。共に本書末尾の「参考情報」で紹介しています。

```
# 送信バッファが空でなければステータス取得まで戻る
05: 24 20       # and $0x20,%al
07: 74 fb       # je -3
# 追加(ここまで)

# ASCII文字'A'(0x41)をシリアルポートへ送信
09: b0 41       # mov $0x41,%al
0b: 66 ba f8 03 # mov $0x03f8,%dx
0f: ee          # out %al,%dx

# 無限Halt
10: f4          # hlt
11: eb fd       # jmp -1
```

実行結果は前節と変わりませんが、これでポーリングによるシリアル送信をちゃんと実装できました。

3.3　受信も追加しエコーバックプログラムを作る

この章の最後に、シリアルポートからの受信データの取得を行ってみます。そして、受信した文字をそのままシリアルポートへ送信することで、エコーバック処理[*5]を実装してみます。

この節のサンプルディレクトリは「033_echoback」です。

♣ 受信バッファにデータが来るのを待つ (命令 21「jne」/命令 22「pause」)

まずは、受信バッファにデータが来るのを待ちます。

シリアルポートの受信バッファにデータがあるか否かもステータスレジスタ LSR で確認できます。受信バッファに 1 つ以上のデータがあれば、LSR のビット 0 がセットされます (表 3.10)。

sample.txt に受信バッファに 1 つ以上のデータが来るまで待つ処理を追加するとリスト 3.6 の通りです。

[*5] 入力したキーに対応する文字がそのまま画面へ表示される処理。シリアルドライバの送受信の動作確認としてよく使われます。

▼表 3.10: LSR の各ビットについて (再掲)

ビット番号	意味
ビット 7	受信バッファに何らかのエラーが発生
ビット 6	送信バッファが空で内部の送信処理も全て完了
ビット 5	送信バッファが空 (データ受け入れ可能)
ビット 4	ブレークシグナルを受信
ビット 3	フレームエラー
ビット 2	パリティエラー
ビット 1	オーバーランエラー
ビット 0	受信バッファに 1 つ以上のデータがある

▼リスト 3.6: 033_echoback/sample.txt

```
# 追加(ここから)
# シリアルポートのステータス取得
00: 66 ba fd 03 # mov $0x03fd,%dx
04: ec          # in %dx,%al

# 受信バッファが空であればpauseしステータス取得まで戻る
05: 24 01       # and $0x01,%al
07: 75 04       # jne +6
09: f3 90       # pause
0b: eb f7       # jmp -7
# 追加(ここまで)

# シリアルポートのステータス取得
14: 66 ba fd 03 # mov $0x03fd,%dx
18: ec          # in %dx,%al

# 送信バッファが空でなければステータス取得まで戻る
# ・・・ 省略 ・・・
```

「シリアルポートのステータス取得」のコードブロックはシリアル送信の際と同じものです。

「受信バッファが空であれば pause しステータス取得まで戻る」のコードブロックは、これまでと少し異なります。

まず、新たに登場した「jne」命令は、「Jump if Not Equal」の略です。je 命令の逆で、「ゼロフラグがセットされていなければジャンプ」という挙動になります。

構文も je 命令等と同様で表 3.11 の通りです。

▼表 3.11: 構文 29: jne: ゼロフラグがセットされていなければジャンプ

アセンブラ	jne <ラベルあるいは".">
機械語	75 fe+rel_addr

そして、AL へ格納したステータス値と 0x01 で AND の結果に対して、以下の振る舞いを実装しています。

- AND の結果が 0x01 (ゼロフラグが 0)
  - jne 命令で 6 バイト先 (このコードブロックの直後) へジャンプ
  - 意図: 受信バッファにデータ来たのでループを抜ける
- AND の結果が 0x00 (ゼロフラグが 1)
  - jne 命令でジャンプせず、次の命令へ進む
  - 「pause」命令を実行後、jmp 命令で 7 バイト前 (LSR 取得) へ戻る
  - 意図: 受信バッファにデータ無いので、pause して再度ステータス取得

ここでは、「ビジーループで待つの際は pause 命令を挟む」という事をするためにこのようなロジックにしています。

「pause」命令は、「ビジーループしている」事を CPU へヒントとして伝える命令で、ビジーループにこの命令を挟んでおくことで CPU の実行効率を改善する効果があります。(詳しくは後述のコラムにまとめています。)

pause 命令はオペランドが不要で、構文としては表 3.12 の通りです。

▼表 3.12: 構文 30: pause: ビジーループのヒントを CPU へ伝える

アセンブラ	pause
機械語	f3 90

併せて、送信バッファが空になるのを待つ処理にも、pause を挟んでおくことにします (リスト 3.7)。

▼リスト 3.7: 033_echoback/sample.txt

```
# ・・・省略・・・

# 受信バッファのデータを取得
0d: 66 ba f8 03 # mov $0x03f8,%dx
11: ec          # in %dx,%al
12: 88 c3       # mov %al,%bl
```

```
# シリアルポートのステータス取得
14: 66 ba fd 03 # mov $0x03fd,%dx
18: ec          # in %dx,%al

# 変更(ここから)
# 送信バッファが空でなければpauseしステータス取得まで戻る
19: 24 20       # and $0x20,%al
1b: 75 04       # jne +6
1d: f3 90       # pause
1f: eb f7       # jmp -7
# 変更(ここまで)

# ・・・省略・・・
```

【コラム】メモリ順序違反と pause 命令

フラグ設定待ちのようなビジーループに pause 命令を入れておくと CPU の実行効率低下を防ぐことができるのは、「メモリ順序違反 (memory order violation)」を防ぐことができるためです。

インテル等の CPU には「投機的実行」と呼ばれる機能があり、現在実行している命令よりも先の命令を CPU 内のバッファへ予め読み込み、場合によっては先に実行しておくことで、CPU の性能最適化を図っています。

ただ、条件付きジャンプ命令等、命令を評価し終えるまでどこへ分岐するか分からない命令があると、その先をどのように先読みするかが問題になります。

この時、インテル等の CPU では「過去にどちらのコードパスを多く通ったか」等の統計値を元に、分岐先を推測して先読みを続行します。

しかし、今回のように CPU としてはほとんどの時間を「フラグ設定待ち」で過ごしていると、命令先読みのバッファは「フラグ設定待ちの処理」の先読みで埋め尽くされ、それらの処理が予め実行される形で最適化が行われます。これは、「次もきっとフラグ待ちだろう」という推測で最適化されている状況です。

そのため、いざフラグが設定された時に、推測が外れ、先読みしたバッファの内容や予め実行した処理もムダになり、実行効率が低下します。これを「メモリ順序違反」と呼びます。

pause 命令があると CPU がこのような誤った先読みをしなくなり、メモリ順序違反を防ぐことができます。

詳しくは、インテルの CPU マニュアルである「Intel 64 and IA-32 Architectures Software Developer's Manual(SDM)」の「Volume 2」の「4.3 INSTRUCTIONS

(M-U)」にある pause 命令の解説をご覧ください。(本書末尾の「参考情報」に PDF が公開されているページの URL を載せています。)

♣ 受信バッファのデータを取得する

次に受信バッファのデータを取得します。受信バッファのレジスタは「RBR(Receiver Buffer Register)」です。

▼表 3.13: RBR について

IO アドレス	0x03f8
レジスタサイズ	1 バイト

RBR の IO アドレスは THR と同じです。同一のアドレスに対して、in 命令による読み出す時は「RBR」へ、out 命令による書き込み時は「THR」へ、となるように内部の回路が組まれています。

RBR からデータを読み出す処理を追加するとリスト 3.8 の通りです。

▼リスト 3.8: 033_echoback/sample.txt

```
# シリアルポートのステータス取得
00: 66 ba fd 03 # mov $0x03fd,%dx
04: ec          # in %dx,%al

# 受信バッファが空であればpauseしステータス取得まで戻る
05: 24 01       # and $0x01,%al
07: 75 04       # jne +6
09: f3 90       # pause
0b: eb f7       # jmp -7

# 追加(ここから)
# 受信バッファのデータを取得
0d: 66 ba f8 03 # mov $0x03f8,%dx
11: ec          # in %dx,%al
12: 88 c3       # mov %al,%bl
# 追加(ここまで)

# シリアルポートのステータス取得
14: 66 ba fd 03 # mov $0x03fd,%dx
18: ec          # in %dx,%al
```

```
# 送信バッファが空でなければpauseしステータス取得まで戻る
# ・・・ 省略 ・・・
```

in 命令の箇所では、受信バッファから AL レジスタへ受信データを格納しています。AL レジスタはその他の in/out 命令の箇所でも使用するため、AL へ格納したデータは mov 命令で BL レジスタへコピーしています。

♣ mov 命令の新構文 (8 ビットレジスタから 8 ビットレジスタへコピー)

ここで、8 ビットレジスタから 8 ビットレジスタへ値をコピーする mov 命令の構文を紹介します。

▼表 3.14: 構文 31: mov: 第 1 から第 2 オペランドへコピー (共に 8 ビットレジスタ)

アセンブラ	mov レジスタ (8bit,src), レジスタ (8bit,dst)
機械語	88 c0+reg_src_dst

「reg_src_dst」については表 3.15 の通りです。

▼表 3.15: 8 ビットレジスタの reg_src_dst 一覧

src/dst	AL	CL	DL	BL	AH	CH	DH	BH
AL	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
CL	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
DL	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
BL	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
AH	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
CH	0x28	0x29	0x2a	0x2b	0x2c	0x2d	0x2e	0x2f
DH	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
BH	0x38	0x39	0x3a	0x3b	0x3c	0x3d	0x3e	0x3f

2 章の imul 命令で紹介した表とは異なり、各行が「第 1 オペランド (src)」で、各列が「第 2 オペランド (dst)」となっている点に注意してください。

♣ 受信バッファから取得したデータを送信する

受信バッファからデータを取得できたので、今度はそれを送信してみます (リスト 3.9)。

▼リスト 3.9: 033_echoback/sample.txt

```
# ・・・ 省略 ・・・

# 送信バッファが空でなければpauseしステータス取得まで戻る
19: 24 20       # and $0x20,%al
1b: 75 04       # jne +6
1d: f3 90       # pause
1f: eb f7       # jmp -7

# 追加・変更(ここから)
# 受信バッファから取得したデータを送信
21: 88 d8       # mov %bl,%al
23: 66 ba f8 03 # mov $0x03f8,%dx
27: ee          # out %al,%dx

# 先頭へ戻る
28: eb d6       # jmp -0x28
# 追加・変更(ここまで)
```

特に新たな要素はありません。受信したデータは BL レジスタにあるため、それを AL へコピーし、DX レジスタへ送信バッファアドレスを設定の上、out 命令でシリアルポートへ送信しています。

♣ 動作確認

これまで通り、ビルドし、動作確認してみてください。

入力したキーに対応する文字がシリアルの画面へ表示されます。

ただ、改行を入力すると、次の行へ移らずに行頭へ戻ってしまいます。

```
$ ../tools/run_qemu ../fs -nographic
...
hogeo world
↑"hello world"と入力後、改行の後、
  "hoge"と入力したが、次の行へ移らない。
```

♣ CR を受け取ったら LF を追加で送信するようにする (命令 23「cmp」)

これは、シリアルポートで受信する改行文字が CR(Carriage Return、復帰コード) のみであるためです。次の行への改行を行う LF(Line Feed) を送信していないため、改

行キーを入力した際に、行頭に移動するのみで次の行へ移動しなかったのでした。

受信バッファから取得したデータを送信後、受信したデータが CR(0x0d) であった場合、LF(0x0a) を追加で送信するようにします (リスト 3.10)。

▼リスト 3.10: 033_echoback/sample.txt

```
# ・・・ 省略 ・・・

# シリアルポートのステータス取得(*1)
14: 66 ba fd 03 # mov $0x03fd,%dx
18: ec          # in %dx,%al

# 送信バッファが空でなければpauseしステータス取得まで戻る
19: 24 20       # and $0x20,%al
1b: 75 04       # jne +6
1d: f3 90       # pause
1f: eb f7       # jmp -7

# 受信バッファから取得したデータを送信
21: 88 d8       # mov %bl,%al
23: 66 ba f8 03 # mov $0x03f8,%dx
27: ee          # out %al,%dx

# 追加・変更(ここから)
# 送信したデータがCR(0x0d)でなければ先頭まで戻る
28: 3c 0d       # cmp $0x0d,%al
2a: 75 d4       # jne -0x2a

# LF(0x0a)を受信データとして(*1)まで戻る
2c: b3 0a       # mov $0x0a,%bl
2e: eb e4       # jmp -0x1a
# 追加・変更(ここまで)
```

新たに「cmp」命令が登場しました。これはオペランドで指定した数値の比較を行う命令です。挙動としては「第 2 オペランド (dst) を書き換えない sub 命令」です。sub 命令は第 2 オペランドから第 1 オペランド減算した結果を第 2 オペランドへ格納しますが、cmp 命令は減算まで行ってフラグレジスタのみ変化させた後、減算結果は捨てます。

構文としては、cmp 命令も sub 命令同様、レジスタとして AL が対象の場合は 1 バイト少ない構文があります (表 3.16)。

AL を含むその他のレジスタについては表 3.17 の通りです。

▼表 3.16: 構文 32: cmp: 即値 (8bit) と AL を比較

アセンブラ	cmp 即値 (8bit), %al
機械語	3c 即値 (8bit)

▼表 3.17: 構文 33: cmp: 即値 (8bit) とレジスタ (8bit) を比較

アセンブラ	cmp 即値 (8bit), レジスタ (8bit)
機械語	80 f8+reg_ofs 即値 (8bit)

追加した処理でやっていることとしては、送信処理後、cmp 命令を使用して、AL レジスタに残っている送信済みの文字と 0x0d(CR) を比較し、等しくなかった場合は jne 命令で先頭まで戻り、等しかった場合は、以降の LF 送信処理へ進みます。

LF 送信処理は、BL へ 0x0a(LF) を格納し、送信のためのシリアルポートステータス確認処理 (*1) までジャンプするだけです。

♣ 動作確認

この状態で動作確認してみると、改行まで反映されているはずです。

```
$ ../tools/run_qemu ../fs -nographic
...
hello world
hoge
```

第4章 フレームバッファで画面描画

「フレームバッファ」を使用してディスプレイへの描画を試してみます。ディスプレイの各ピクセルには、メモリ空間上に対応する領域があり、それが「フレームバッファ」と呼ばれる領域です。

poiboot では、kernel.bin へジャンプする際、UEFI で取得したフレームバッファ情報を渡すようにしています。この章では poiboot から渡されたフレームバッファ情報を取得し、取得したフレームバッファ情報を使用して、画面へドットを描画したり、塗りつぶしたりしてみます。

フレームバッファの各ピクセルは「B(青)」・「G(緑)」・「R(赤)」・「Reserved(予約領域)」の順に各 1 バイトずつ計 4 バイトで構成されています。フレームバッファ領域の最も若いアドレスが画面左上のピクセルに対応します。

4.1 フレームバッファのアドレスを取得する

まずは、フレームバッファ領域の先頭アドレスを取得してみます。

この節のサンプルディレクトリは「041_get_fb_addr」です。

♣ フレームバッファに関する情報の在処

フレームバッファに関する情報は、ブートローダー (poiboot) から渡されます。

poiboot では、フレームバッファに関する情報を「struct fb(リスト 4.1)」という構造でメモリ上に配置しています。そして、kernel.bin を実行する際、その先頭アドレスを RSI レジスタ[*1]へ格納した上で、kernel.bin へジャンプします。

[*1] RSI レジスタは、関数の第 2 引数として使われるレジスタです。そのため、普通に C 言語などでカーネルを書く際は、kernel.bin の一番先頭の関数の第 2 引数で struct fb の先頭アドレスを取得できます。

▼リスト 4.1: struct fb の構造

```
struct fb {
        /* フレームバッファの先頭アドレス */
        unsigned long long base;
        /* フレームバッファサイズ[バイト] */
        unsigned long long size;
        /* 画面の幅[ピクセル] */
        unsigned int hr;
        /* 画面の高さ[ピクセル] */
        unsigned int vr;
};
```

リスト 4.1 のコメントに記載の通り、フレームバッファの先頭アドレスは、先頭のメンバー変数「base」に格納されています。「先頭のメンバー変数のアドレス」は「構造体の先頭アドレス」と等しいので、RSI に格納されているアドレス (struct fb の先頭アドレス) は、メンバー変数 base のアドレスでもあります。そのため、RSI が指す先に「フレームバッファの先頭アドレス」があります。

♣ レジスタ間接アドレッシング

「あるアドレスが指す先」のメモリを参照したい場合、例えば C 言語では「ポインタ」がありますが、機械語 (およびアセンブラ) では「レジスタ間接アドレッシング」(あるいは単に「レジスタ間接」) というオペランド指定方法があります。

これは、命令のオペランドとして「レジスタに格納されたアドレスの指す先」を指定するものです。アセンブラでは「**(% レジスタ名)**」の様に書きます。ここでは、RSI に格納されたアドレス先のメモリ上のデータ (フレームバッファ先頭アドレス) を、64 ビットレジスタ RAX へコピーしてみます。対応する mov 命令は表 4.1 の通りです。

▼表 4.1: mov 命令: RSI に格納されたアドレス先の内容を RAX へコピー

アセンブラ	mov (%rsi),%rax
機械語	48 8b 06

レジスタ間接で取得した値を 64 ビットレジスタへ格納する構文は表 4.2 の通りで、構文内の「`reg_dst_src`」は表 4.3 の通りです。

表 4.2 の備考について、レジスタ間接に使用するレジスタが RSP あるいは RBP の場合、機械語のパターンが少し変わります。例としては表 4.4 と表 4.5 の通りです。

なお、一部の命令では、レジスタ間接に 64 ビット幅以外のレジスタを使う命令もあったりしますが、本書で使わないため省略します。

▼表 4.2: 構文 34: mov: 第 1 オペランドの指す先を第 2 オペランドへ格納

アセンブラ	mov (レジスタ (64bit)), レジスタ (64bit)
機械語	48 8b 00+reg_dst_src
備考	レジスタ間接が RSP の場合、末尾に 0x24 追加
	レジスタ間接が RBP の場合、3 バイト目に 0x40 加算、末尾に 0x00 追加

▼表 4.3: 64 ビットレジスタの reg_dst_src

dst/(src)	(RAX)	(RCX)	(RDX)	(RBX)	(RSP)	(RBP)	(RSI)	(RDI)
RAX	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
RCX	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
RDX	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
RBX	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
RSP	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
RBP	0x28	0x29	0x2a	0x2b	0x2c	0x2d	0x2e	0x2f
RSI	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
RDI	0x38	0x39	0x3a	0x3b	0x3c	0x3d	0x3e	0x3f

▼表 4.4: 例: レジスタ間接のレジスタが RSP の場合

アセンブラ	mov (%rsp), %rax
機械語	48 8b 04 24

▼表 4.5: 例: レジスタ間接のレジスタが RBP の場合

アセンブラ	mov (%rbp), %rax
機械語	48 8b 45 00

♣ 実装

サンプルコードはリスト 4.2 の通りです。

▼リスト 4.2: 041_get_fb_addr/sample.txt

```
# フレームバッファ先頭アドレスをRAXへ格納
00: 48 8b 06    # mov (%rsi),%rax
```

```
# 無限Halt
03: f4          # hlt
04: eb fd       # jmp -1
```

フレームバッファの先頭アドレスを RAX へコピーした後、halt の無限ループで待機するようにしました。

♣ 動作確認

QEMU で動作させて、QEMU モニターで RAX を見てみると、フレームバッファの先頭アドレスが格納されています。

```
(qemu) info registers
RAX=00000000c0000000 RBX=00000000bef67f98 RCX=0000000000000000 RDX=00000000000›
›00000
RSI=0000000000407180 RDI=00000000bfe9b018 RBP=00000000bff0eb10 RSP=00000000002›
›10000
R8 =00000000bff0e9dc R9 =0000000000000078 R10=00000000bfe6e080 R11=00000000e7d›
›c4657
R12=00000000bef69540 R13=00000000bef69548 R14=00000000bff24b60 R15=00000000bef›
›6a018
RIP=0000000000110004 RFL=00000002 [-------] CPL=0 II=0 A20=1 SMM=0 HLT=1
```

この場合、フレームバッファの先頭アドレスは「0xc0000000」であったと分かります。

4.2　画面の幅と高さを取得する

続いて、画面の幅と高さを取得します。

この節のサンプルディレクトリは「042_get_width_height」です。

♣ ディスプレースメント付きレジスタ間接アドレッシング

画面の幅と高さも struct fb の構造の中にあります。それぞれ、幅 (hr) は struct fb の先頭から 16(0x10) バイト目、高さ (vr) は 20(0x14) バイト目です。

これらのデータを参照する際、アドレスを格納したレジスタへオフセット分の値を都度足したり引いたりしても参照できるのですが、少し面倒です。

レジスタ間接アドレッシングには「ディスプレースメント (dsp)」と呼ばれるオフセット値を指定できるため、これを使うと「あるアドレスから N バイト先のデータ」へ 1 命令でアクセスできます。

ディスプレースメント付きの間接アドレスは、アセンブラでは「dsp(レジスタ

名)」の様に書きます。「RSI に格納されたアドレスの 16(0x10) バイト先」であれば「`0x10(%rsi)`」、「RSI に格納されたアドレスの 20(0x14) バイト先」であれば「`0x14(%rsi)`」です。

幅 (hr) と高さ (vr) はそれぞれ 32 ビットです。ここでは、幅の値を 32 ビットレジスタ EBX へ、高さの値を 32 ビットレジスタ ECX へコピーすることにします。それぞれのアセンブラと機械語コードは表 4.6 と表 4.7 の通りです。

▼表 4.6: mov 命令: 幅 (RSI 格納アドレスの 0x10 バイト先) を EBX へコピー

アセンブラ	mov **0x10**(%rsi),%ebx
機械語コード	8b 5e **10**

▼表 4.7: mov 命令: 高さ (RSI 格納アドレスの 0x14 バイト先) を ECX へコピー

アセンブラ	mov **0x14**(%rsi),%ecx
機械語コード	8b 4e **14**

それぞれ、末尾の 1 バイトがディスプレースメントです。

8 ビットのディスプレースメント付きレジスタ間接の値を 32 ビットレジスタへ格納する構文は表 4.8 の通りで、構文内の「`reg_dst_src`」は表 4.9 の通りです。

▼表 4.8: 構文 35: mov: DSP(8bit) 付きレジスタ間接の値をレジスタ (32bit) へ格納

アセンブラ	mov dsp(レジスタ (64bit)), レジスタ (32bit)
機械語	8b 40+reg_dst_src dsp(8bit)
備考	レジスタ間接が RSP の場合、3 バイト目に 0x24 追加

表 4.8 の備考に記載の通り、レジスタ間接のレジスタが RSP の場合、機械語のパターンが変わります。例としては表 4.10 の通りです。

▼表 4.9: レジスタ間接とレジスタ (32bit) のオフセット値

dst/(src)	(RAX)	(RCX)	(RDX)	(RBX)	(RSP)	(RBP)	(RSI)	(RDI)
EAX	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
ECX	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
EDX	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
EBX	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
ESP	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
EBP	0x28	0x29	0x2a	0x2b	0x2c	0x2d	0x2e	0x2f
ESI	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
EDI	0x38	0x39	0x3a	0x3b	0x3c	0x3d	0x3e	0x3f

▼表 4.10: 例: レジスタ間接のレジスタが RSP の場合

アセンブラ	mov 0x12(%rsp), %rax
機械語	8b 44 24 12

♣ 備考: DSP の有無による機械語の変化

ここで、構文 33 で紹介した DSP 無しの構文では、レジスタを示すバイトは「00+reg_dst_src」という構文でした。今回の場合と比較すると、レジスタを示すバイトに 0x40 を足すと「DSP 付き」の意味になり、末尾に追加した 1 バイトが DSP として扱われるようです。

構文 34 より、「0x12」の DSP 付きで RAX のレジスタ間接で取得した値を EAX へ格納する機械語は「`8b 40 12`」です。

試しに、2 バイト目から 0x40 を引き、3 バイト目の DSP を除いた「`8b 00`」を das_dump で逆アセンブルしてみます。

```
$ das_dump 8b 00

/tmp/tmp.pYcO1hU5Ag/a.bin:     ファイル形式 binary


セクション .data の逆アセンブル:

0000000000000000 <.data>:
   0:   8b 00                   mov    (%rax),%eax
```

「`8b 00`」は DSP 無しに対応する命令だと分かりました。これより、「レジスタ間接

の対象レジスタを示すバイトに 0x40 を足すと DSP 付きを示し、末尾の 1 バイトが DSP になる」と考えて良さそうです。

DSP 無しの構文 33 の備考を改めて見てみると、『レジスタ間接のレジスタ (src) が RBP の場合、3 バイト目に 0x40 を足し、末尾に「00」を追加』とありました。これは、RBP には DSP 無しの機械語が無いので、DSP 無しのアセンブラからアセンブルすると、DSP を 0 とした機械語が生成されていたのですね。

♣ 実装

以上を踏まえて、前節の sample.txt に幅と高さの取得を追加してみます (リスト 4.3)。

▼リスト 4.3: 042_get_width_height/sample.txt

```
# フレームバッファ先頭アドレスをRAXへ格納
00: 48 8b 06    # mov (%rsi),%rax

# 追加(ここから)
# 画面の幅をEBXへ、高さをECXへ格納
03: 8b 5e 10    # mov 0x10(%rsi),%ebx
06: 8b 4e 14    # mov 0x14(%rsi),%ecx
# 追加(ここまで)

# 無限Halt
09: f4          # hlt
0a: eb fd       # jmp -1
```

♣ 動作確認

QEMU 上で実行し、QEMU モニターでレジスタ値を確認すると、EBX と ECX にそれぞれ、フレームバッファの幅と高さが格納されています。

```
(qemu) info registers
RAX=00000000c0000000 RBX=0000000000000320 RCX=0000000000000258 RDX=00000000000›
›00000
RSI=0000000000407180 RDI=00000000bfe9b018 RBP=00000000bff0eb10 RSP=00000000002›
›10000
R8 =00000000bff0e9dc R9 =0000000000000078 R10=00000000bfe6e080 R11=00000000e7d›
›c4657
R12=00000000bef69540 R13=00000000bef69548 R14=00000000bff24b60 R15=00000000bef›
›6a018
RIP=000000000011000a RFL=00000002 [-------] CPL=0 II=0 A20=1 SMM=0 HLT=1
```

EBX に格納されている「0x320」は 10 進数で「800」、ECX に格納されている「0x258」

は 10 進数で「600」です。QEMU のフレームバッファ解像度はデフォルトで 800x600 なので、正常に取得できているとわかります。

4.3　画面へ 1 ピクセル描画してみる

前節までで、フレームバッファで画面描画を行うために必要な情報は揃いました。この節からはそれらを使って画面描画を行ってみます。まずは 1 ピクセルだけ画面に描画してみます。

この節のサンプルディレクトリは「043_draw_pixel」です。

♣ 即値をレジスタ間接で書き込む

前節までで確認したレジスタ間接による方法で、メモリ上にマップされたフレームバッファの領域へ書き込みを行います。

フレームバッファのピクセルフォーマットは「青 (B), 緑 (G), 赤 (R), 予約 (RSV)」が各 1 バイトずつで、1 ピクセル当たり 4 バイトです。フレームバッファには画面左上から順にピクセルが並んでいるため、フレームバッファ先頭の 4 バイトに書き込みを行うと画面左上の 1 ピクセルを操作できます。

ここではフレームバッファ先頭の 4 バイトに即値で「B=0x00, G=0xff, R=0x00, RSV=0x00」を書き込んで、画面左上の 1 ピクセルを緑で塗ってみます。対応する機械語コードは表 4.11 の通りです。

▼表 4.11: mov 命令: 画面左上 (0,0) を緑 (0x00,0xff,0x00,0x00) で塗る

アセンブラ	movl $**0x0000ff00**,(%rax)
機械語コード	c7 00 **00 ff 00 00**

即値をレジスタ間接で書き込むの場合、アセンブラ上では何ビットの書き込みかが分からない[*2]ため、書き込みのビット幅を明示する必要があります。

ここでは、32 ビットアクセスであることを明示するため `mov` に接尾辞として"l"[*3]を付けています。

[*2] アセンブラでは即値で「0x0000ff00」と書いていても操作するビット幅に応じて上の位のビットを良しなに 0(あるいは符号ビット) で補完します。そのため、「0x0000ff00(計 32 ビット)」と書いているからといって、「32 ビットの即値書き込み」の意思がアセンブラへ伝わるわけではないです。「mov $0x0000ff00,(%rax)」でアセンブルしようとすると「サイズが分からない」といったエラーが出ます。

[*3] 他にも、8 ビットアクセスは"b"、16 ビットアクセスは"w"、64 ビットアクセスは"q"といった接尾辞があります。

機械語コードの末尾 4 バイトが書き込む即値データです。
構文をまとめると以下の通りです。

▼表 4.12: 構文 36: mov: 即値 (32bit) をレジスタ間接で書き込む

アセンブラ	movl 即値 (32bit), (レジスタ (64bit))
機械語	c7 00+reg_ofs 即値 (32bit)
備考	レジスタ間接が RSP の場合、3 バイト目に 0x24 追加
	レジスタ間接が RBP の場合、2 バイト目に 0x40 加算、3 バイト目に 0x00 追加

♣ 実装

前節までの sample.txt へ、画面左上 (0,0) のピクセルを緑で塗りつぶす処理を追加してみます (リスト 4.4)。

▼リスト 4.4: 043_draw_pixel/sample.txt

```
# フレームバッファ先頭アドレスをRAXへ
# 画面の幅をEBXへ、高さをECXへ格納
00: 48 8b 06            # mov (%rsi),%rax
03: 8b 5e 10            # mov 0x10(%rsi),%ebx
06: 8b 4e 14            # mov 0x14(%rsi),%ecx

# 追加(ここから)
# フレームバッファ先頭ピクセル(画面左上)を緑で塗る
09: c7 00 00 ff 00 00   # movl $0x0000ff00,(%rax)
# 追加(ここまで)

# 無限Halt
0f: f4                  # hlt
10: eb fd               # jmp -1
```

♣ 動作確認

GUI モードで実行すると、たった 1 ピクセルなので見難いですが、一番左上の 1 ピクセルが緑で塗られていることを確認できます。

また、QEMU モニターでフレームバッファのデータをダンプすることでも確認できます。

```
(qemu) x/4bx 0xc0000000
00000000c0000000: 0x00 0xff 0x00 0x00
```

筆者の場合、フレームバッファのアドレスは 0xc0000000 だったので、そこから 1 バイトずつ、4 個分のデータを 16 進数でダンプしてみました。

「0x00(B) 0xff(G) 0x00(R) 0x00(RSV)」と、意図通りにフレームバッファへ書き込みができていることを確認できます。

4.4　画面を塗りつぶしてみる

前節で画面の左上 (0,0) の 1 ピクセルを塗ることができました。フレームバッファ上の書き込み先を 1 ピクセル (4 バイト) ずつずらしながら、全ピクセル (幅 * 高さ) 分を書き込むことで、画面全体を単色で塗りつぶしてみます。

この節のサンプルディレクトリは「044_fill」です。

♣ 実装の流れ

jmp 命令を使うことでループを作ることはできるので「書き込み先を 4 バイトずつずらしながら、即値を書き込む」事はできそうです。

ただ、フレームバッファの領域を超えて書き込みを行ってしまうとフレームバッファ外の領域のメモリ破壊につながります。そのため、レジスタ EDX を「書き込んだピクセル数を管理するカウンタ」、レジスタ ESI を「画面の総ピクセル数 (幅 * 高さ)」とし、「EDX >= ESI」のときループを脱出するようにします。

ここで主に使用する命令が、数値比較を行う「cmp」命令と、「jbe」命令です。

♣ 等しいか小さければジャンプ (命令 24「jbe」)

jbe は「Jump if Below or Equal」の略です。既に登場した「je(Jump if Equal) 命令」等と同じく条件付きジャンプ命令の一種です。

対応するフラグレジスタはキャリーフラグとゼロフラグで、いずれかがセットされていた場合にジャンプします。

キャリーフラグは、算術演算の結果、レジスタサイズを超える桁上がり (キャリー) あるいはボロー (桁借り) が発生した場合にセットされます。今回の場合、キャリーフラグがセットされるのは「ESI - EDX が負となる場合」で、ゼロフラグがセットされるのは「ESI と EDX が等しい場合」なので、「EDX >= ESI」のときジャンプすることになります。

jbe 命令の構文は表 4.13 の通りです。

▼表 4.13: 構文 37: jbe: 等しいか小さければジャンプ

アセンブラ	jbe <ラベルあるいは".">
機械語コード	76 fe+rel_addr

末尾の 1 バイトがジャンプ先を決める相対アドレスで、jmp 命令等と同様、基準は 0xfe です。

♣ 実装

前節までの sample.txt へ画面塗りつぶし処理を追加してみました (リスト 4.5)。

▼リスト 4.5: 044_fill/sample.txt

```
# フレームバッファ先頭アドレスをRAXへ
# 画面の幅をEBXへ、高さをECXへ格納
00: 48 8b 06            # mov (%rsi),%rax
03: 8b 5e 10            # mov 0x10(%rsi),%ebx
06: 8b 4e 14            # mov 0x14(%rsi),%ecx

# 追加・変更(ここから)
# EDXを「カウンタ」、ESIを「画面ピクセル数(幅*高さ)」で初期化
09: 31 d2               # xor %edx,%edx
0b: 89 ce               # mov %ecx,%esi
0d: 0f af f3            # imul %ebx,%esi

# カウンタ(EDX)が画面ピクセル数(ESI)へ達したら「無限Halt」へジャンプ(*1)
10: 39 d6               # cmp %edx,%esi
12: 76 0e               # jbe +0x10

# 現在のピクセルを緑で描画し次のピクセルへ移動
14: c7 00 00 ff 00 00   # movl $0x0000ff00,(%rax)
1a: 48 83 c0 04         # add $4,%rax

# カウンタ(EDX)をインクリメントし(*1)へジャンプ
1e: ff c2               # inc %edx
20: eb ee               # jmp -0x10
# 追加・変更(ここまで)

# 無限Halt
22: f4                  # hlt
23: eb fd               # jmp -1
```

初出の構文については後述しますが、やっていることとしてはコメントに記載のとおりです。

気をつける点としては、1 ピクセルの描画を行っている以下の命令です。

▼リスト 4.6: 即値のバイトオーダーに注意

```
# 現在のピクセルを緑で描画し次のピクセルへ移動
14: c7 00 00 ff 00 00   # movl $0x0000ff00,(%rax)
```

機械語の末尾 4 バイトには、描画したいピクセル情報が B(0x00)・G(0xff)・R(0x00)・RSV(0x00) の順に並んでいますが、この命令自体は「32 ビット即値をレジスタ間接で書き込む」です。そのため、アセンブラでは、バイトオーダーをリトルエンディアンで並べると「B・G・R・RSV」の順になるように、「0x0000ff00」という即値を指定しています。

♣ 初出の構文について

今回、「xor」・「mov」・「imul」・「cmp」・「add」・「inc」で新しい構文がでてきましたので、紹介しておきます。

xor

32 ビットレジスタを使う構文が新しく登場しました。

▼表 4.14: 構文 38: xor: レジスタ (32bit) 同士の XOR を第 2 オペランドへ格納

アセンブラ	xor レジスタ (32bit), レジスタ (32bit)
機械語	31 c0+reg_src_dst

「reg_src_dst」は以下の通りです。

▼表 4.15: 32 ビットレジスタの reg_src_dst

src/dst	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
EAX	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
ECX	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
EDX	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
EBX	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
ESP	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
EBP	0x28	0x29	0x2a	0x2b	0x2c	0x2d	0x2e	0x2f
ESI	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
EDI	0x38	0x39	0x3a	0x3b	0x3c	0x3d	0x3e	0x3f

今回、xor は EDX レジスタをゼロクリアするために使用しています。mov 命令で即値 0 を格納しても良いのですが、xor を使用しているのは機械語のバイト数が 2 バイトと小さいからです。

試しに as_dump で確認してみます。

```
$ as_dump mov \$0,%edx

/tmp/tmp.0Sw3Al6M9z/a.out:     ファイル形式 elf64-x86-64


セクション .text の逆アセンブル:

0000000000000000 <.text>:
   0:   ba 00 00 00 00          mov    $0x0,%edx
```

mov 命令の場合、即値部分で 4 バイト使ってしまう 5 バイトの命令でした。

mov

mov も 32 ビットレジスタに関する構文が初出です。

▼表 4.16: 構文 39: mov: 第 1 から第 2 オペランドへコピー (共に 32bit レジスタ)

アセンブラ	mov レジスタ (32bit), レジスタ (32bit)
機械語	89 c0+reg_src_dst

「reg_src_dst」は XOR と同じです。

imul

imul も 32 ビットレジスタに関する構文です。

▼表 4.17: 構文 40: imul: レジスタ (32bit) 同士の積を第 2 オペランドへ格納

アセンブラ	imul レジスタ (32bit), レジスタ (32bit)
機械語	0f af c0+reg_dst_src

「reg_dst_src」は以下の通りです。

先に紹介した xor・mov とは行・列が異なるのでご注意ください。

▼表 4.18: 32 ビットレジスタの reg_dst_src

dst/src	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
EAX	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
ECX	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
EDX	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
EBX	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
ESP	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
EBP	0x28	0x29	0x2a	0x2b	0x2c	0x2d	0x2e	0x2f
ESI	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
EDI	0x38	0x39	0x3a	0x3b	0x3c	0x3d	0x3e	0x3f

cmp

cmp も 32 ビットレジスタに関する構文です。

▼表 4.19: 構文 41: cmp: レジスタ (32bit) 同士の比較

アセンブラ	cmp レジスタ (32bit), レジスタ (32bit)
機械語	39 c0+reg_src_dst

「reg_src_dst」は、xor・mov と同じです。

add

add は 8 ビット即値を 64 ビットレジスタへ加算する構文が初出でした。

▼表 4.20: 構文 42: add: 即値 (8bit) をレジスタ (64bit) へ加算

アセンブラ	add 即値 (8bit), レジスタ (64bit)
機械語	48 83 c0+reg_ofs 即値 (8bit)

inc

inc も 32 ビットレジスタについてです。

▼表 4.21: 構文 43: inc: レジスタ (32bit) をインクリメント

アセンブラ	inc レジスタ (32bit)
機械語	ff c0+reg_ofs

♣ 動作確認

GUI で実行すると、画面が緑一色で塗りつぶされることが確認できます (図 4.1)。(白黒印刷なので印刷上はわからないですが。。)

▲図 4.1: 044_fill/sample.txt の実行結果

4.5　ランダム色で塗りつぶしてみる

各ピクセルを塗る色をランダムに決めることでカラーノイズを描画してみます。
この節のサンプルディレクトリは「045_color_noise」です。

♣ rdrand 命令で乱数取得 (命令 25)

実は x86 CPU には乱数を取得する命令があります。それが「rdrand 命令」です。rdrand はオペランドにレジスタを指定することで、そのレジスタへ乱数を格納してくれます。

ここでは、レジスタ EDI へ乱数を格納し、この 4 バイトレジスタの内容をピクセルデータとしてフレームバッファへ書き込んでみます。

rdrand 命令の今回使用する構文は以下の通りです。

▼表 4.22: 構文 44: rdrand: 乱数をレジスタ (32bit) へ格納

アセンブラ	rdrand レジスタ (32bit)
機械語	0f c7 f0+reg_ofs

♣ 実装

早速実装してみます (リスト 4.7)。

▼リスト 4.7: 045_color_noise/sample.txt

```
# ・・・ 省略 ・・・

# カウンタ(EDX)が画面ピクセル数(ESI)へ達したら「無限Halt」へジャンプ(*1)
10: 39 d6       # cmp %edx,%esi
12: 76 0d       # jbe +0x0f     (変更)

# 変更(ここから)
# 現在のピクセルをランダム色で描画し次のピクセルへ移動
14: 0f c7 f7    # rdrand %edi
17: 89 38       # mov %edi,(%rax)
# 変更(ここまで)
19: 48 83 c0 04 # add $4,%rax

# カウンタ(EDX)をインクリメントし(*1)へジャンプ
1d: ff c2       # inc %edx
1f: eb ef       # jmp -0x0f     (変更)

# ・・・ 省略 ・・・
```

rdrand 命令で 32 ビットレジスタ EDI へ乱数を格納し、それを mov 命令でフレームバッファへ書き込むように変更しました。

なお、32 ビットレジスタをレジスタ間接で書き込む mov 命令は新しい構文なので、紹介しておきます。

▼表 4.23: 構文 45: mov: レジスタ (32bit) をレジスタ間接で書き込む

アセンブラ	mov レジスタ (32bit), (レジスタ (64bit))
機械語	89 00+reg_src_dst
備考	レジスタ間接が RSP の場合、末尾に 0x24 追加
	レジスタ間接が RBP の場合、2 バイト目に 0x40 加算、末尾に 0x00 追加

「reg_src_dst」は以下の通りです。

▼表 4.24: 32 ビットレジスタ間接での reg_src_dst

src/(dst)	(RAX)	(RCX)	(RDX)	(RBX)	(RSP)	(RBP)	(RSI)	(RDI)
EAX	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
ECX	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
EDX	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
EBX	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
ESP	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
EBP	0x28	0x29	0x2a	0x2b	0x2c	0x2d	0x2e	0x2f
ESI	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
EDI	0x38	0x39	0x3a	0x3b	0x3c	0x3d	0x3e	0x3f

備考について、例えば EAX から RSP のレジスタ間接で書き込む場合、以下のようになります。

▼表 4.25: 例: EAX から RSP のレジスタ間接で書き込む

アセンブラ	mov %eax, (%rsp)
機械語	89 04 24

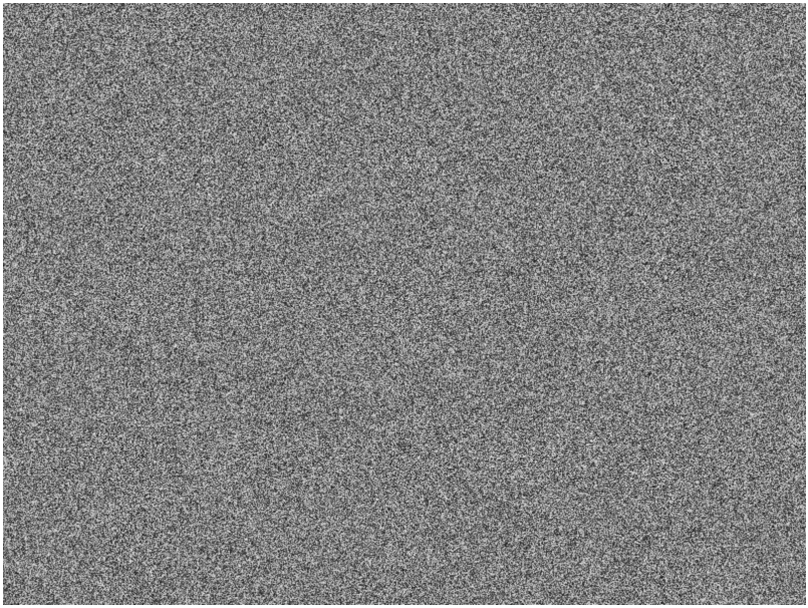
また、EAX から RBP のレジスタ間接で書き込む場合は以下のようになります。

▼表 4.26: 例: EAX から RBP のレジスタ間接で書き込む

アセンブラ	mov %eax, (%rbp)
機械語	89 45 00

♣ 動作確認

実行すると図 4.2 の様に、ランダム色の描画により、ノイズ画像のようなものが表示されます。(これも白黒印刷なのでわかりにくいですが)



▲図 4.2: 045_color_noise/sample.txt の実行結果

第5章 キーボードドライバを作る

前章で画面描画が行えるようになりました。

この章ではキーボードドライバを作成することで、シリアル通信ではない方法でキー入力を取得できるようにします。

この章では 1 章を通して「051_get_scancode」のサンプルを作成します。

5.1 キー入力を受け取るまでの流れ

キーボードのキー入力は、キーボードコントローラ (KBC) という IC とやり取りすることで取得できます。KBC もレジスタを持っており、KBC のレジスタを参照することで、入力されたキーの情報や、キーボードに関する現在のステータスを確認できます。

KBC のレジスタは IO アドレス空間に対応付けられています。そのため、in 命令/out 命令でレジスタを読み書きします。

キー入力 (スキャンコード) を取得する流れは、シリアル通信でデータ受信を行ったときと同様で、以下の通りです。

1. ステータスレジスタ値を取得
2. キー入力値が無ければ 1.(プログラム先頭) へ戻る
3. データレジスタからキー入力値を取得
4. キー入力値からキー押した (Make) かキーを離した (Break) かを判別 (希望の状態でなければ 1. へ戻る)
5. キー入力値からスキャンコードを取得
6. 1. へ戻る

今回は、キーから指を離した (Break) 時のスキャンコードを取得してみることにします。

5.2　ステータスレジスタ値を取得

それでは、実装してみます。

KBC のステータスレジスタは IO アドレス空間の 0x0064 に割り当てられている 8 ビットのレジスタです。

例によって in 命令でレジスタ値を取得します。なお、今回のように IO アドレスが 8 ビット (1 バイト) に収まる場合、2 バイトで書ける構文があります。

▼表 5.1: 構文 46: in: IO アドレス (1 バイト) 先から AL へ値を読み出す

アセンブラ	in 即値 (8bit), %al
機械語	e4 即値 (8bit)

今回の場合、以下のようになります。

▼表 5.2: 例: in: IO アドレス (0x64) から AL へ読み出す

アセンブラ	in $0x64,%al
機械語	e4 64

なので、「1. ステータスレジスタ値を取得」の部分の機械語コードはリスト 5.1 の通りです。

▼リスト 5.1: 051_get_scancode/sample.txt

```
# 1. ステータスレジスタ値を取得
00: e4 64       # in $0x64,%al
```

5.3　キー入力値が無ければ先頭へ戻る

ステータスレジスタの最下位ビット (ビット 0) で KBC にキー入力値がある (=1) か無い (=0) かが分かります。

「フラグが立つまで待つ」系の処理で、コードブロックの構造としては「シリアル通信」の「受信待ち」の時と使う命令も同じです (リスト 5.2)。

▼リスト 5.2: 051_get_scancode/sample.txt

```
# 1. ステータスレジスタ値を取得
00: e4 64       # in $0x64,%al

# 追加(ここから)
# 2. キー入力値が無ければ先頭へ戻る
# (pauseしながら待つ)
02: 24 01       # and $0x01,%al
04: 75 04       # jne +6 (入力が有れば進む)
06: f3 90       # pause
08: eb f6       # jmp -0x08 (入力が無ければ戻る)
# 追加(ここまで)
```

and 命令によるマスクで AL レジスタの最下位ビットだけ抽出し、その結果に応じて以下の挙動となります。

- and の結果が 0 (ビット 0 がセットされていない)
  - ゼロフラグがセットされる
  - jne はスルーし、pause した後、jmp で in 命令まで戻る
- and の結果が 1 (ビット 0 がセットされている)
  - ゼロフラグがクリアされる
  - jne でこの命令から (この命令含む)6 バイト先へジャンプ (このコードブロック直後)

5.4　データレジスタからキー入力値を取得

データレジスタの IO アドレスは 0x60 です。in 命令で取得し AL レジスタへ格納します。

sample.txt へ追記するとリスト 5.3 の通りです。

▼リスト 5.3: 051_get_scancode/sample.txt

```
# 1. ステータスレジスタ値を取得
00: e4 64       # in $0x64,%al

# 2. キー入力値が無ければ先頭へ戻る
# (pauseしながら待つ)
02: 24 01       # and $0x01,%al
04: 75 04       # jne +6 (入力が有れば進む)
06: f3 90       # pause
08: eb f6       # jmp -0x08 (入力が無ければ戻る)
```

```
# 追加(ここから)
# 3. データレジスタからキー入力値を取得
0a: e4 60       # in $0x60,%al
# 追加(ここまで)
```

5.5　キー入力値が Break ならスキャンコードを抽出し、Make なら先頭へ戻る

データレジスタの 8 ビットの内容は以下の通りです。

▼表 5.3: データレジスタの内容

ビット	内容
7	Make(=0) か Break(=1) か
0 - 6	スキャンコード

Make 時のスキャンコードなら、ビット 7 は 0 なので、単にデータレジスタから取得した 8 ビットをそのままスキャンコードとして使えば良いのですが、今回は Break 時のスキャンコードを取得したいので、ビット 7 がスキャンコードに含まれないように何らかの対処が必要です。

♣ 実装方針

今回は以下の 2 命令で実装してみます。

1. sub 命令: AL - 0x80 → AL
2. jb 命令：演算結果がマイナスなら先頭へジャンプ

1 つ目の sub 命令がミソです。AL のビット 7 がセットされていたか否かに応じて、以下の挙動になります。

- AL のビット 7 が 0 (Make)
  - 0x80 の減算結果はマイナスの値になる
  - AL には負値の何らかの値が格納される (使わないので気にしない)
- AL のビット 7 が 1 (Break)
  - 0x80 の減算結果はプラスの値になる
  - AL にはビット 7 のみ 0 にした値が格納される (スキャンコードのみ抽出)

こうすることで、「スキャンコード部分の抽出」と「最上位ビットの状態確認」を同時に行うことができます。

♣ 比較結果が「小さい」ならジャンプ (命令 26「jb」)

なお、「jb」命令は初出なのでここで紹介します。

条件付きジャンプ命令の一種で、「Jump if Below」の略です。フラグレジスタとしてはキャリーフラグがセットされているとジャンプします。

構文は以下の通りです。

▼表 5.4: 構文 47: jb: 比較結果が小さいならジャンプ

アセンブラ	jb <ラベルあるいは".">
機械語	72 fe+rel_addr

キャリーフラグに応じてジャンプする命令でありながら「Jump if Below」という名前なのは、cmp 命令が内部的には減算によって比較を行っていることに由来していると思いますが、キャリーフラグはオーバーフローした場合にもセットされるので、add 命令等で桁溢れが起きてキャリーフラグがセットされた場合も、jb 命令はもちろんジャンプします。

♣ 実装

sample.txt へ追記するとリスト 5.4 の通りです。

▼リスト 5.4: 051_get_scancode/sample.txt

```
# 1. ステータスレジスタ値を取得
00: e4 64       # in $0x64,%al

# 2. キー入力値が無ければ先頭へ戻る
# (pauseしながら待つ)
02: 24 01       # and $0x01,%al
04: 75 04       # jne +6 (入力が有れば進む)
06: f3 90       # pause
08: eb f6       # jmp -0x08 (入力が無ければ戻る)

# 3. データレジスタからキー入力値を取得
0a: e4 60       # in $0x60,%al

# 追加(ここから)
# 4. キー入力値がBreakならスキャンコードを抽出し、Makeなら先頭へ戻る
0c: 2c 80       # sub $0x80,%al
```

```
0e: 72 f0       # jb -0x0e (Makeなら先頭へ戻る)
# 追加(ここまで)
```

jb 命令 2 バイト目の相対アドレス表記はその他の jmp 命令と同じで、自分自身の命令位置が 0xfe です。今回の場合、sub 命令含め、jb より上に 14 バイトあるので、相対アドレス表記としては 0xf0 になります。

5.6　スキャンコードをレジスタ CL へコピーし、先頭へ戻る

特に意味はないのですが、一応、アウトプット代わりとして、取得したスキャンコードを CL レジスタへコピーした上で、先頭へ戻ります (リスト 5.5)。

▼リスト 5.5: 051_get_scancode/sample.txt

```
# 1. ステータスレジスタ値を取得
00: e4 64       # in $0x64,%al

# 2. キー入力値が無ければ先頭へ戻る
# (pauseしながら待つ)
02: 24 01       # and $0x01,%al
04: 75 04       # jne +6 (入力が有れば進む)
06: f3 90       # pause
08: eb f6       # jmp -0x08 (入力が無ければ戻る)

# 3. データレジスタからキー入力値を取得
0a: e4 60       # in $0x60,%al

# 4. キー入力値がBreakならスキャンコードを抽出し、Makeなら先頭へ戻る
0c: 2c 80       # sub $0x80,%al
0e: 72 f0       # jb -0x0e (Makeなら先頭へ戻る)

# 追加(ここから)
# 5. スキャンコードをレジスタCLへコピー
10: 88 c1       # mov %al,%cl

# 6. 先頭へ戻る
12: eb ec       # jmp -0x12
# 追加(ここまで)
```

5.7 動作確認

QEMU の GUI 画面でキー入力をしながら QEMU モニターで CL レジスタの値を確認したいので、動作確認の際は、以下のように QEMU モニターだけ、実行したシェル上で起動するようにするとやりやすいです。そして、別途立ち上がった GUI のウィンドウの方でキー入力 (Make&Break) を行います。

```
$ ../tools/run_qemu ../fs -monitor stdio
vvfat ../fs chs 1024,16,63
QEMU 2.8.1 monitor - type 'help' for more information
(qemu) info registers
RAX=000000000011001c RBX=00000000bef6ab18 RCX=0000000000000000 RDX=00000000000›
›00000
# ↑起動直後、CLは0x00
...
# '1'キーをMake&Break
(qemu) info registers
RAX=000000000011001c RBX=00000000bef6ab18 RCX=0000000000000002 RDX=00000000000›
›00000
# ↑CLにはスキャンコード0x02が格納された
...
# '2'キーをMake&Break
(qemu) info registers
RAX=000000000011001c RBX=00000000bef6ab18 RCX=0000000000000003 RDX=00000000000›
›00000
# ↑CLにはスキャンコード0x03が格納された
...
# '3'キーをMake&Break
(qemu) info registers
RAX=000000000011001c RBX=00000000bef6ab18 RCX=0000000000000004 RDX=00000000000›
›00000
# ↑CLにはスキャンコード0x04が格納された
...
# 'a'キーをMake&Break
(qemu) info registers
RAX=000000000011001c RBX=00000000bef6ab18 RCX=000000000000001e RDX=00000000000›
›00000
# ↑CLにはスキャンコード0x1eが格納された
...
```

スキャンコードは ASCII コードとは別で、スキャンコードセットには大きく 3 つの種類があります。スキャンコードについては例えば以下のページ等が詳しいです。

- 5. Scan Code Sets - PS/2 Keyboard - OSDev Wiki
  - https://wiki.osdev.org/PS/2_Keyboard#Scan_Code_Sets

付録A 登場した機械語命令と構文の一覧

A.1 構文表記のキーワードについて

本文でも説明した通り、機械語の構文には、オフセットを足すことで対象のレジスタやアドレス等、細かい意味が変わる構文があります。

本書では、そのようなオフセットはキーワードを対応付けて、構文を表現しています。

ここでは、本書で登場したキーワードをまとめます。

♣ rel_addr

ジャンプ命令で使用するキーワードで、自身の命令の地点からのジャンプ先の相対的なバイト数を示します。

♣ reg_ofs

対象のレジスタを示すオフセット値です。

reg_ofs	64bit	32bit	16bit	8bit
0	RAX	EAX	AX	AL
1	RCX	ECX	CX	CL
2	RDX	EDX	DX	DL
3	RBX	EBX	BX	BL
4	RSP	ESP	SP	AH
5	RBP	EBP	BP	CH
6	RSI	ESI	SI	DH
7	RDI	EDI	DI	BH

♣ reg_src_dst

オペランドにレジスタが複数登場する場合、元 (src) と先 (dst) のレジスタの組み合わせを示すオフセット値です。

8 ビットレジスタ同士の場合

src/dst	AL	CL	DL	BL	AH	CH	DH	BH
AL	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
CL	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
DL	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
BL	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
AH	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
CH	0x28	0x29	0x2a	0x2b	0x2c	0x2d	0x2e	0x2f
DH	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
BH	0x38	0x39	0x3a	0x3b	0x3c	0x3d	0x3e	0x3f

32 ビットレジスタ同士の場合

src/dst	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
EAX	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
ECX	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
EDX	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
EBX	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
ESP	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
EBP	0x28	0x29	0x2a	0x2b	0x2c	0x2d	0x2e	0x2f
ESI	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
EDI	0x38	0x39	0x3a	0x3b	0x3c	0x3d	0x3e	0x3f

32 ビットレジスタと 64 ビットレジスタ間接の場合

src/(dst)	(RAX)	(RCX)	(RDX)	(RBX)	(RSP)	(RBP)	(RSI)	(RDI)
EAX	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
ECX	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
EDX	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
EBX	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
ESP	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
EBP	0x28	0x29	0x2a	0x2b	0x2c	0x2d	0x2e	0x2f
ESI	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
EDI	0x38	0x39	0x3a	0x3b	0x3c	0x3d	0x3e	0x3f

♣ reg_dst_src

こちらも、元 (src) と先 (dst) のレジスタの組み合わせを示すオフセット値で、オフセット値を増やしていった際の src と dst の関係が「reg_dst_src」とは異なるパターンです。

端的には、オフセット値を行方向に増やしていく表記をする際、「reg_dst_src」は、

列が dst、行が src になります。

16 ビットレジスタ同士の場合

dst/src	AX	CX	DX	BX	SP	BP	SI	DI
AX	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
CX	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
DX	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
BX	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
SP	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
BP	0x28	0x29	0x2a	0x2b	0x2c	0x2d	0x2e	0x2f
SI	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
DI	0x38	0x39	0x3a	0x3b	0x3c	0x3d	0x3e	0x3f

32 ビットレジスタ同士の場合

dst/src	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
EAX	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
ECX	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
EDX	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
EBX	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
ESP	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
EBP	0x28	0x29	0x2a	0x2b	0x2c	0x2d	0x2e	0x2f
ESI	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
EDI	0x38	0x39	0x3a	0x3b	0x3c	0x3d	0x3e	0x3f

32 ビットレジスタと 64 ビットレジスタ間接の場合

dst/(src)	(RAX)	(RCX)	(RDX)	(RBX)	(RSP)	(RBP)	(RSI)	(RDI)
EAX	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
ECX	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
EDX	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
EBX	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
ESP	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
EBP	0x28	0x29	0x2a	0x2b	0x2c	0x2d	0x2e	0x2f
ESI	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
EDI	0x38	0x39	0x3a	0x3b	0x3c	0x3d	0x3e	0x3f

64 ビットレジスタと 64 ビットレジスタ間接の場合

dst/(src)	(RAX)	(RCX)	(RDX)	(RBX)	(RSP)	(RBP)	(RSI)	(RDI)
RAX	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
RCX	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
RDX	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
RBX	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
RSP	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
RBP	0x28	0x29	0x2a	0x2b	0x2c	0x2d	0x2e	0x2f
RSI	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
RDI	0x38	0x39	0x3a	0x3b	0x3c	0x3d	0x3e	0x3f

A.2　四則演算

♣ add: 加算

構文 6	**add: 即値 (8bit) を AL へ加算**
アセンブラ	add 即値 (8bit), %al
機械語	04 即値 (8bit)
構文 7	**add: 即値 (8bit) をレジスタ (8bit) へ加算**
アセンブラ	add 即値 (8bit), レジスタ (8bit)
機械語	80 c0+reg_ofs 即値 (8bit)
構文 42	**add: 即値 (8bit) をレジスタ (64bit) へ加算**
アセンブラ	add 即値 (8bit), レジスタ (64bit)
機械語	48 83 c0+reg_ofs 即値 (8bit)

♣ sub: 減算

構文 8	**sub: 即値 (8bit) を AL レジスタから減算**
アセンブラ	sub 即値 (8bit), %al
機械語	2c 即値 (8bit)
構文 9	**sub: 即値 (8bit) をレジスタ (8bit) から減算**
アセンブラ	sub 即値 (8bit), レジスタ (8bit)
機械語	80 e8+reg_ofs 即値 (8bit)

♣ imul: 乗算

構文 10	**imul: AL * レジスタ (8bit) を AX へ格納**
アセンブラ	imul レジスタ (8bit)
機械語	f6 e8+reg_ofs
構文 11	**imul: 第 1 * 第 2 オペランド (16bit レジスタ) を第 2 オペランドへ格納**
アセンブラ	imul レジスタ (16bit,src), レジスタ (16bit,dst)
機械語	66 0f af c0+reg_dst_src
構文 12	**imul: 第 1 オペランド * 第 2 オペランドを第 3 オペランドへ格納**
アセンブラ	imul 即値 (8bit), レジスタ (16bit,src), レジスタ (16bit,dst)
機械語	66 6b c0+reg_dst_src 即値 (8bit)
構文 40	**imul: レジスタ (32bit) 同士の積を第 2 オペランドへ格納**
アセンブラ	imul レジスタ (32bit), レジスタ (32bit)
機械語	0f af c0+reg_dst_src

♣ idiv: 除算

構文 13	**idiv: AX/レジスタ (8bit) の商を AL へ剰余を AH へ格納**
アセンブラ	idiv レジスタ
機械語	f6 f8+reg_ofs

A.3　インクリメント/デクリメント

♣ inc: インクリメント

構文 24	**inc: レジスタ (8bit) をインクリメント**
アセンブラ	inc レジスタ (8bit)
機械語	fe c0+reg_ofs
構文 43	**inc: レジスタ (32bit) をインクリメント**
アセンブラ	inc レジスタ (32bit)
機械語	ff c0+reg_ofs

♣ dec: デクリメント

構文 25	**dec: レジスタ (8bit) をデクリメント**
アセンブラ	inc レジスタ (32bit)
機械語	ff c0+reg_ofs

A.4　比較

♣ cmp: 2 つのオペランドの差を評価

構文 32	**cmp: 即値 (8bit) と AL を比較**
アセンブラ	cmp 即値 (8bit), %al
機械語	3c 即値 (8bit)
構文 33	**cmp: 即値 (8bit) とレジスタ (8bit) を比較**
アセンブラ	cmp 即値 (8bit), レジスタ (8bit)
機械語	80 f8+reg_ofs 即値 (8bit)
構文 41	**cmp: レジスタ (32bit) 同士の比較**
アセンブラ	cmp レジスタ (32bit), レジスタ (32bit)
機械語	39 c0+reg_src_dst

A.5　論理演算

♣ and: 論理積

構文 14	**and: 即値 (8bit) AND AL を AL へ格納**
アセンブラ	and 即値 (8bit), %al
機械語	24 即値 (8bit)
構文 15	**and: 即値 (8bit) AND レジスタ (8bit) をレジスタ (8bit) へ格納**
アセンブラ	and 即値 (8bit), レジスタ (8bit)
機械語	80 e0+reg_ofs 即値 (8bit)

♣ or: 論理和

構文 16	**or: 即値 (8bit) OR AL を AL へ格納**
アセンブラ	or 即値 (8bit), %al
機械語	0c 即値 (8bit)
構文 17	**or: 即値 (8bit) OR レジスタ (8bit) をレジスタ (8bit) へ格納**
アセンブラ	or 即値 (8bit), レジスタ (8bit)
機械語	80 c8+reg_ofs 即値 (8bit)

♣ xor: 排他的論理和

構文 18	**xor: 即値 (8bit) XOR AL を AL へ格納**
アセンブラ	xor 即値 (8bit), %al
機械語	34 即値 (8bit)
構文 19	**xor: 即値 (8bit) XOR レジスタ (8bit) をレジスタ (8bit) へ格納**
アセンブラ	xor 即値 (8bit), レジスタ (8bit)
機械語	80 f0+reg_ofs 即値 (8bit)
構文 38	**xor: レジスタ (32bit) 同士の XOR を第 2 オペランドへ格納**
アセンブラ	xor レジスタ (32bit), レジスタ (32bit)
機械語	31 c0+reg_src_dst

♣ not: 否定

構文 20	**not: レジスタ (8bit) の NOT をレジスタ (8bit) へ格納**
アセンブラ	not レジスタ (8bit)
機械語	f6 d0+reg_ofs

A.6 ビットシフト

ビットシフトは挙動としては表 A.20 の通りです。
sal と shl は挙動が同じなので、機械語も同じです。そのため、sal のみ紹介します。

▼表 A.1: 各シフト演算の挙動

sal	第 1 オペランドの回数分、第 2 オペランドへ 2 を掛ける
sar	第 1 オペランドの回数分、第 2 オペランドを 2 で割る (符号有)
shl	第 1 オペランドの回数分、第 2 オペランドへ 2 を掛ける
shr	第 1 オペランドの回数分、第 2 オペランドを 2 で割る (符号無)

♣ sal: 算術左シフト

構文 21	**sal: 第 1 オペランド＝即値 (8bit)、第 2 オペランド＝レジスタ (8bit)**
アセンブラ	sal 即値 (8bit), レジスタ (8bit)
機械語	c0 e0+reg_ofs 即値 (8bit)

♣ sar: 算術右シフト

構文 22	**sar: 第 1 オペランド＝即値 (8bit)、第 2 オペランド＝レジスタ (8bit)**
アセンブラ	sar 即値 (8bit), レジスタ (8bit)
機械語	c0 f8+reg_ofs 即値 (8bit)

♣ shr: 論理右シフト

構文 23	**shr: 第 1 オペランド＝即値 (8bit)、第 2 オペランド＝レジスタ (8bit)**
アセンブラ	shr 即値 (8bit), レジスタ (8bit)
機械語	c0 e8+reg_ofs 即値 (8bit)

A.7　読み出し/書き込み

♣ mov: 第 1 から第 2 オペランドへコピー

構文 4	**mov: 即値 (8bit) をレジスタ (8bit) へ格納**
アセンブラ	mov 即値 (8bit), レジスタ (8bit)
機械語	b0+reg_ofs 即値 (8bit)
構文 31	**mov: 第 1 から第 2 オペランドへコピー (共に 8 ビットレジスタ)**
アセンブラ	mov レジスタ (8bit,src), レジスタ (8bit,dst)
機械語	88 c0+reg_src_dst

構文 5	**mov: 即値 (16bit) をレジスタ (16bit) へ格納**
アセンブラ	mov 即値 (16bit), レジスタ (16bit)
機械語	66 b9+reg_ofs 即値 (16bit)

構文 39	**mov: 第 1 から第 2 オペランドへコピー (共に 32bit レジスタ)**
アセンブラ	mov レジスタ (32bit), レジスタ (32bit)
機械語	89 c0+reg_src_dst
構文 36	**mov: 即値 (32bit) をレジスタ間接で書き込む**
アセンブラ	movl 即値 (32bit), (レジスタ (64bit))
機械語	c7 00+reg_ofs 即値 (32bit)
備考	レジスタ間接が RSP の場合、3 バイト目に 0x24 追加
	レジスタ間接が RBP の場合、2 バイト目に 0x40 加算、3 バイト目に 0x00 追加
構文 45	**mov: レジスタ (32bit) をレジスタ間接で書き込む**
アセンブラ	mov レジスタ (32bit), (レジスタ (64bit))
機械語	89 00+reg_src_dst
備考	レジスタ間接が RSP の場合、3 バイト目に 0x24 追加
	レジスタ間接が RBP の場合、2 バイト目に 0x40 加算、末尾に 0x00 追加
構文 35	**mov: DSP(8bit) 付きレジスタ間接の値をレジスタ (32bit) へ格納**
アセンブラ	mov dsp(レジスタ (64bit)), レジスタ (32bit)
機械語	8b 40+reg_dst_src dsp(8bit)
備考	レジスタ間接が RSP の場合、3 バイト目に 0x24 追加

構文 34	**mov: 第 1 オペランドの指す先を第 2 オペランド (64 ビットレジスタ) へ格納**
アセンブラ	mov (レジスタ (64bit)), レジスタ (64bit)
機械語	48 8b 00+reg_dst_src
備考	レジスタ間接が RSP の場合、末尾に 0x24 追加
	レジスタ間接が RBP の場合、3 バイト目に 0x40 加算、末尾に 0x00 追加

♣ in: IO アドレス空間から読み出し

構文 46	**in: IO アドレス (1 バイト) 先から AL へ値を読み出す**
アセンブラ	in 即値 (8bit), %al
機械語	e4 即値 (8bit)
構文 27	**in: DX が指すレジスタ値を AL へ格納**
アセンブラ	in %dx, %al
機械語	ec

♣ out: IO アドレス空間へ書き込み

構文 26	out: AL を DX が指すアドレス先へ書き込む
アセンブラ	out %al, %dx
機械語	ee

A.8　CPU 状態制御

♣ hlt: CPU をスリープ

構文 3	hlt: CPU をスリープさせる
アセンブラ	hlt
機械語	f4

♣ pause: ビジーループのヒント

構文 30	pause: ビジーループのヒントを CPU へ伝える
アセンブラ	pause
機械語	f3 90

A.9　分岐

♣ jmp: 無条件ジャンプ

構文 1	jmp: 相対アドレス指定でジャンプ
アセンブラ	jmp <ラベルあるいは".">
機械語	eb fe+rel_addr

♣ jb: 小さい (Below) 場合にジャンプ

構文 47	jb: 比較結果が小さいならジャンプ
アセンブラ	jb <ラベルあるいは".">
機械語	72 fe+rel_addr

♣ jbe: 小さいか等しい (Below or Equal) 場合にジャンプ

構文 37	jbe: 等しいか小さければジャンプ
アセンブラ	jbe <ラベルあるいは".">
機械語コード	76 fe+rel_addr

♣ je: 等しい (Equal) 場合にジャンプ

構文 28	je: ゼロフラグがセットされていればジャンプ
アセンブラ	je <ラベルあるいは".">
機械語	74 fe+rel_addr

♣ jne: 等しくない (Not Equal) 場合にジャンプ

構文 29	jne: ゼロフラグがセットされていなければジャンプ
アセンブラ	jne <ラベルあるいは".">
機械語	75 fe+rel_addr

A.10　その他

♣ nop: 何もしない (No Operation)

構文 2	nop: 何もしない
アセンブラ	nop
機械語	90

♣ rdrand: 乱数取得

構文 44	rdrand: 乱数をレジスタ (32bit) へ格納
アセンブラ	rdrand レジスタ (32bit)
機械語	0f c7 f0+reg_ofs

おわりに

本書をお読みいただきありがとうございました！

機械語を手で書く、しかもそれをコンパイラ等も十分に在る x86_64 アーキテクチャで、という内容でしたが、いかがだったでしょうか。「意外と書こうと思えば書けるな」と感じてもらえれば幸いです。

完全にネタ本なのですが、何か役に立つとすれば、高級な言語でも処理を書きながら「機械語に落とし込まれたら何バイトくらいかな」という機械語におけるサイズ感が身につく、とかでしょうか。ただ、機械語の 1 命令は小さいものでは 1 バイトから、大きいものでは 10 数バイトもあったりするし、同じ命令でもオペランドの違いだけでサイズが大きく変わってきたりすることもあるので、高級言語で処理を書きながら機械語の実装をエスパーするのは相当難易度高いかも。。

やっぱり「何かの役に立つものでは無い」のですが。。機械語を書いて実行することで「通訳を介さずに CPU と直接話している感覚」とか、面白がってもらえれば嬉しいです。

あと、本書のサンプルはどれも、「フレームバッファへの書き込み」や「命令フェッチ」を除き、一切メモリアクセスをしないのもポイントです。ただ、これに関してはアセンブラで書いても同じことはできるのですが、少なくとも C 言語とか、スタックを使うバイナリを自動的に生成してしまう高級言語ではできないことです。ただ、スタックの仕組みを使わないことには、今後大きなプログラムを書いていくのは難しそうですが。。

参考情報

参考にさせてもらった情報

- Intel 64 and IA-32 Architectures Software Developer Manuals
  - https://software.intel.com/en-us/articles/intel-sdm
  - Intel 公式の x86 CPU のソフトウェア開発マニュアルです
  - CPU の基本的なアーキテクチャやレジスタなどについては Volume 1 に書かれています
  - 各命令のリファレンスは Volume 2 に書かれており、全命令と全構文が網羅されています
- WikiBooks「X86 アセンブラ/x86 アーキテクチャ」
  - https://ja.wikibooks.org/wiki/X86 アセンブラ/x86 アーキテクチャ
  - x86 CPU の基本的なアーキテクチャについてはこちらの記事が分かりやすいです。
  - ただ、64 ビットに関する記載はありませんが。。
- 書籍「ハードウェアデザインシリーズ 12 パソコンのレガシィ I/O 活用大全」
  - 桑野 雅彦 著、CQ 出版社、2000 年
  - 本書で扱ったシリアルポートやキーボードコントローラなどは、今やレガシーなアーキテクチャで、レジスタの仕様等の詳細はネット上よりもこの書籍が詳しいです
  - ただ、中古で買うしかない状況ではありますが。。

poiboot について

本書でブートローダーとして使用している「poiboot」は、筆者の過去の著作である以下の同人誌の成果物に少し手を加えたものです。

- フルスクラッチで作る!UEFI ベアメタルプログラミング! パート 1、パート 2
  - ブートローダーを作る上で必要な UEFI を直接叩く方法を紹介

当サークルの同人誌は、すべて以下の URL から PDF 版/HTML 版は無料でダウンロード/閲覧できますので興味があればぜひ見てみてもらえると嬉しいです。

- http://yuma.ohgami.jp

作って分かる！ x86_64 機械語入門

OS レスで CPU と直接話す!

2019 年 9 月 22 日　ver 1.0

著　者　大神祐真
印刷所　日光企画

(powered by Re:VIEW Starter)