アセンブリ言語

ハードウェア・インタフェース（1）

情報工学系
権藤克彦

2014/10現在の

悲しいお知らせ

- Mac OS XのApple版GNUアセンブラでは,
  16ビットコードを正しくアセンブルできません.

- リアルモード用の16ビットコードのコンパイルは
  LinuxかWindows (Cygwin)上でしましょう.
  - CSCでは仮想マシンソフトVirtualBox で Windows7 起動可.
- i386-elf-gcc などをインストールして，macOS上で
  クロスコンパイルする手もある.
  - CSCでは以下のパスに i386-jos-elf がインストール済み.
    - /opt/local/os/bin （3年生のシステムソフトウェアの授業で使用する）

悲しいお知らせ（続）

foo.s

```
.text
.code16
je foo
nop
foo:
```

Windows XP (Cygwin)

```
% gcc –c foo.s
% objdump –d –mi8086 foo.o
foo.o:     file format pe-i386
Disassembly of section .text:
00000000 <.text>:
 0: 74 01    je     3 <foo>
 2: 90       nop
00000003 <foo>:
```

Linux (Ubuntu 10.4)

```
% gcc –c foo.s
% objdump –d –mi8086 foo.o
foo.o: file format elf32-i386
Disassembly of section .text:
00000000 <foo-0x3>:
 0: 74 01    je  3 <foo>
 2: 90       nop
```

je *rel8* の機械語コードを出力.

Mac OS X 10.8.5, GCC-4.2.1(LLVM)

```
% gcc –c foo.s
% gobjdump –d –mi8086 foo.o
foo.o:     ファイル形式 mach-o-x86-64
セクション .text の逆アセンブル:
00000000 <foo-0x8>:
 0: 0f 84 00 00  je  4 <foo-0x4>
 4: 00 00        add %al,(%bx,%si)
 6: 90           nop
```

je *rel32* の機械語コードを間違って出力.
je *rel8* か je *rel16* を出力するべき.

悲しいお知らせ（続）

CSCの環境，Mac OS X 10.8.5

```
% set path=(/opt/local/os/bin $path)
% i386-jos-elf-gcc –c foo.s
% i386-jos-elf-objdump –d –mi8086 foo.o
foo.o:     file format elf32-i386
Disassembly of section .text:
00000000 <foo-0x3>:
 0: 74 01    je  3 <foo>
 2: 90       nop
```

正しく動作している

参考資料

ハードウェアに関する良い本は少ない.

- OADGテクニカル・リファレンス（ハードウェア），
  - PCオープン・アーキテクチャー推進協議会，1994.
    - http://www.oadg.or.jp/techref/oadghwd.pdf
- OADGテクニカル・リファレンス（DOS/V 技術解説編）
  - PCオープン・アーキテクチャー推進協議会，1994.
    - http://www.oadg.or.jp/techref/oadgdosv.pdf
- パソコンのレガシィI/O活用大全：割り込みとDMAからシリアル/パラレル・ポート，FDD/IDEインターフェースまで，
  - 桑野 雅彦，ISBN: 4789834336，CQ出版，2000（良書，入手困難）
- The Indispensable PC Hardware Book, 4th Ed.,
  - Hans-Peter Messmer, ISBN: 0201596164, Addison Wesley, 2001.

- IA-32 インテルアーキテクチャー・ソフトウェア・デベロッパーズ・マニュアル
  - http://www.intel.com/jp/download/index.htm

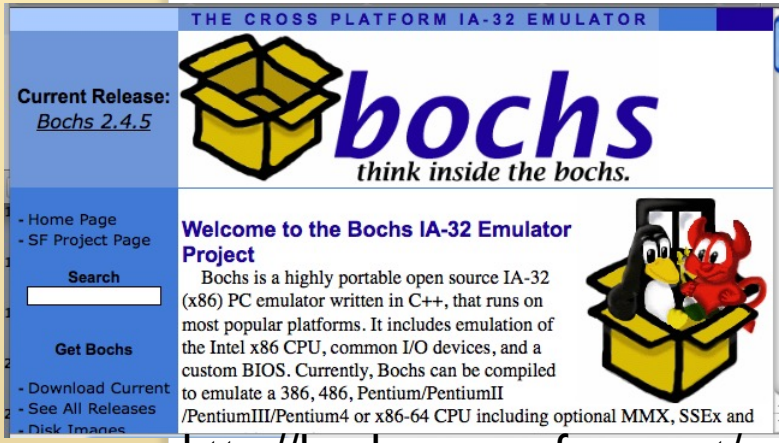
Bochsの使用は興味のある人だけ各自で
インストール方法等は一部古い情報

Bochs：PCエミュレータ

http://bochs.sourceforge.net/

Bochsの概要

- PC/AT互換機エミュレータ（仮想機械）　　類：QEMU
  - WindowsやMac OS Xなど，多くの環境上で動作.
  - 多くのOSをエミュレータ上で動作可能.
  - ただし，エミュレーションは100%完全ではない.
- Bochsはインタプリタ方式.
  - 実行は遅い.
  - Bochs内蔵のデバッガでいろいろ観察可能.
    - 例：レジスタの値，物理メモリや仮想メモリの値，エラーの原因.



http://bochs.sourceforge.net/

Bochsのダウンロード

- Bochsホームページからソースコードをダウンロード.
  - http://bochs.sourceforge.net/
  - http://bochs.sourceforge.net/cgi-bin/topper.pl?name=See+All+Releases&url=http://sourceforge.net/project/showfiles.phpqmrkgroup_ideq12580
- マニュアル等のドキュメント
  - http://bochs.sourceforge.net/doc/docbook/user/index.html

2014/10/2現在で bochs-2.6.6が最新

Bochsのコンパイル（Mac OS X上）

```
% mv bochs-2.4.5.tar.gz コンパイルする場所
% cd コンパイルする場所
% tar xvzf bochs-2.4.5.tar.gz
% cd bochs-2.4.5
% sh .conf.macosx
% make
% sudo make install
```

Apple が Carbon を捨てたので，ちょっと大変

管理者権限が必要.
これ無しでも使用は可能.

デバッガを使用可にするには，
./configure のオプションに
--enable-debugger --enable-disasm
の追加が必要.

Snow Leopard/Mountain Lionでコンパイルするには
g++に -arch i386 -m32 オプションの追加が必要.
CPPFLAGS=""
CXXFLAGS="$CFLAGS"
CXX="g++ -arch i386 –m32"

BochsをMacPortsでインストール

- 自前Macなら MacPorts の方が簡単

```
% sudo port install bochs
```

- Bochs内蔵デバッガを有効にするには上の前に以下が必要

```
% sudo port edit bochs
```

エディタ(vi)が起動するので，configure.args のオプションに
--enable-debugger –enable-disasm を追加

Bochsの使用方法（一般）

- コマンドbochsにPATHを通す.
- 設定ファイルbochsrc を準備する.
  - .bochsrc (bochsrc-sample.txt) をコピーして編集する.
  - 特に，ROMイメージの指定が重要.
    - BIOS-bochs-latest，VGABIOS-lgpl-latest
  - bochsrcの記法はバージョンにより異なるので注意.
- 実行するOSのイメージファイルを準備する.

- bochs -q -f bochs.rc
  - デバッガを使用可にした場合は，起動後に c を入力.
- Mac では X11環境が必要→XQuartzなどを起動しておく

boot-hello（1）

- ブートして画面にHelloを書くプログラム.

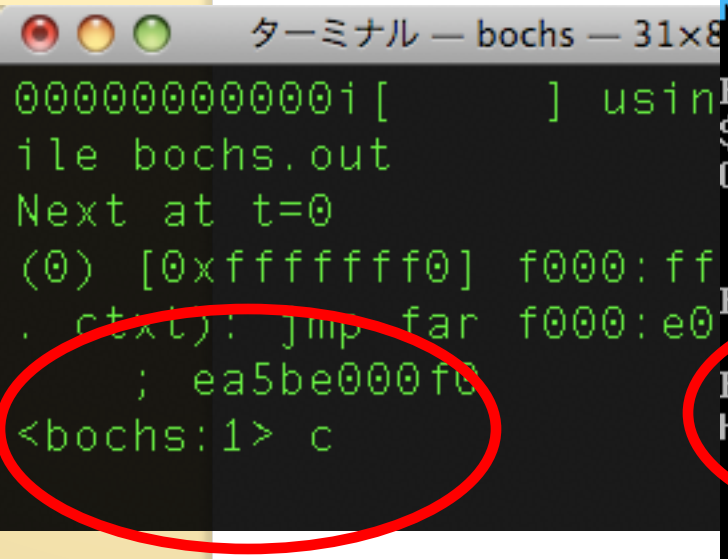
```
.code16
.text
start:
    ljmp $0x07c0, $start2
start2:
    movw %cs,    %ax
    movw %ax,    %ds
    movw %ax,    %ss

    movw $mesg,  %bp
```

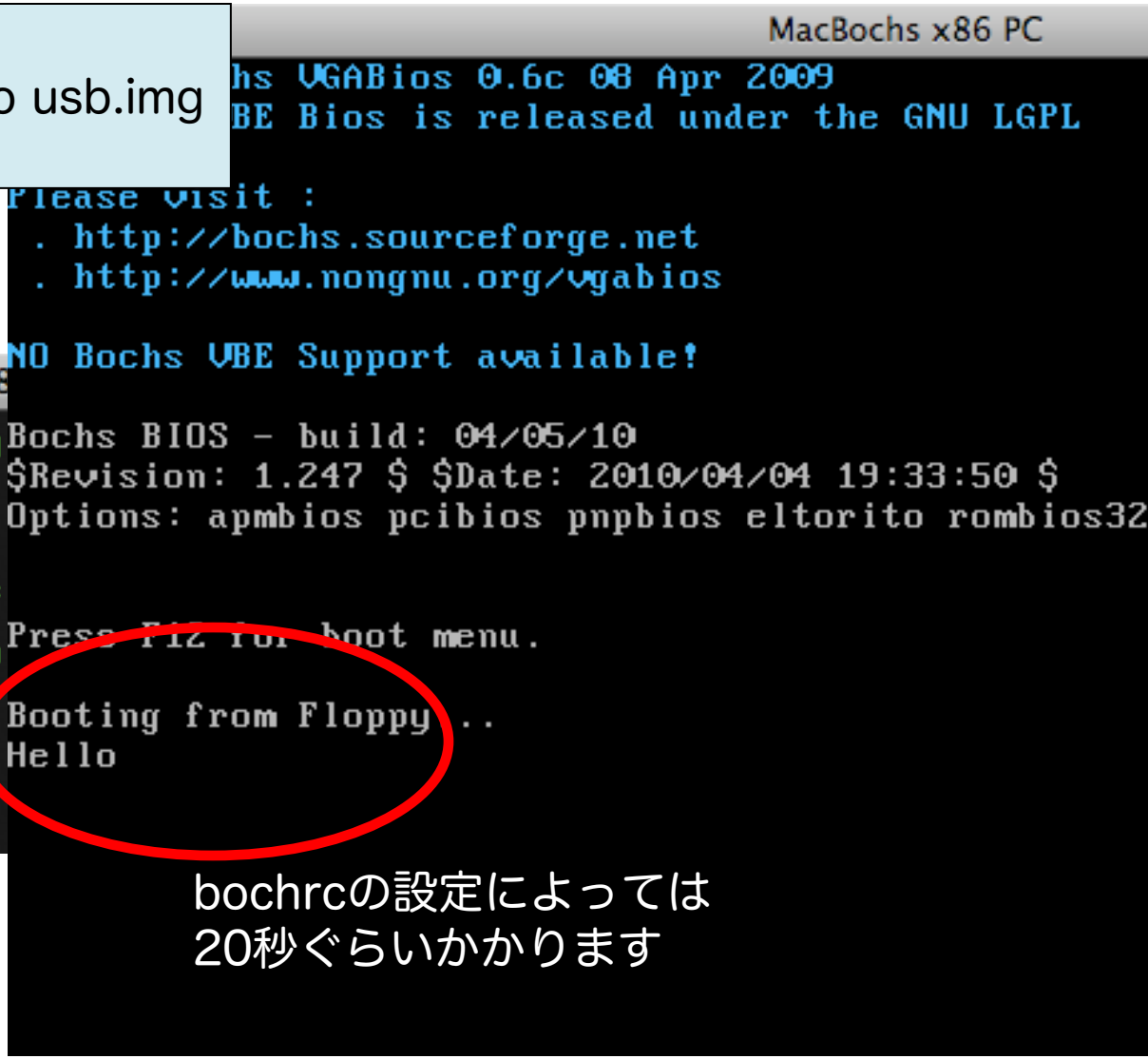
中身の説明は後述.

```
loop:
    movb (%bp),  %al
    cmpb $0,     %al
    je   exit
    movb $0xe,   %ah
    movb $0x12,  %bl
    int  $0x10
    incw %bp
    jmp  loop
exit:
    hlt
    jmp exit
mesg:
    .ascii "Hello¥0"
.org 510
    .word 0xaa55
```

boot-hello（2）

```
% gcc -c boot.s
% objcopy -O binary boot.o usb.img
% bochs -q -f bochs.rc
```





boot-hello　実行

bochrcの設定によっては
20秒ぐらいかかります

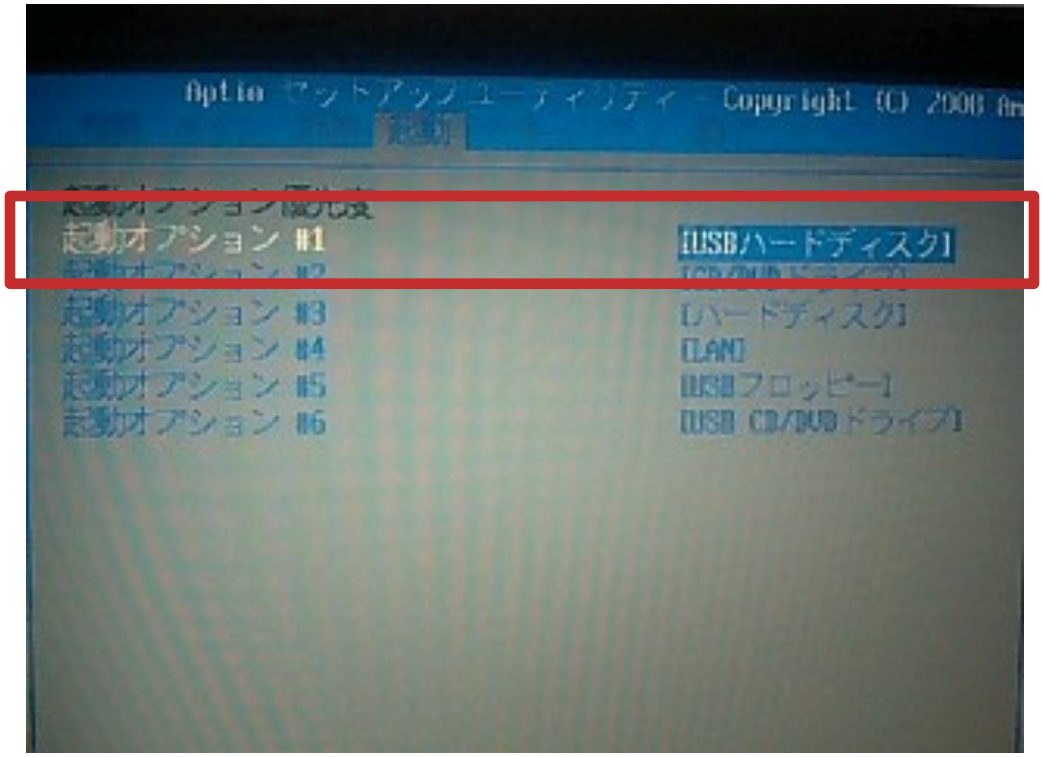
boot-hello（3）

注意：パソコン上のデータ破壊の可能性.
自己責任で実行すること.

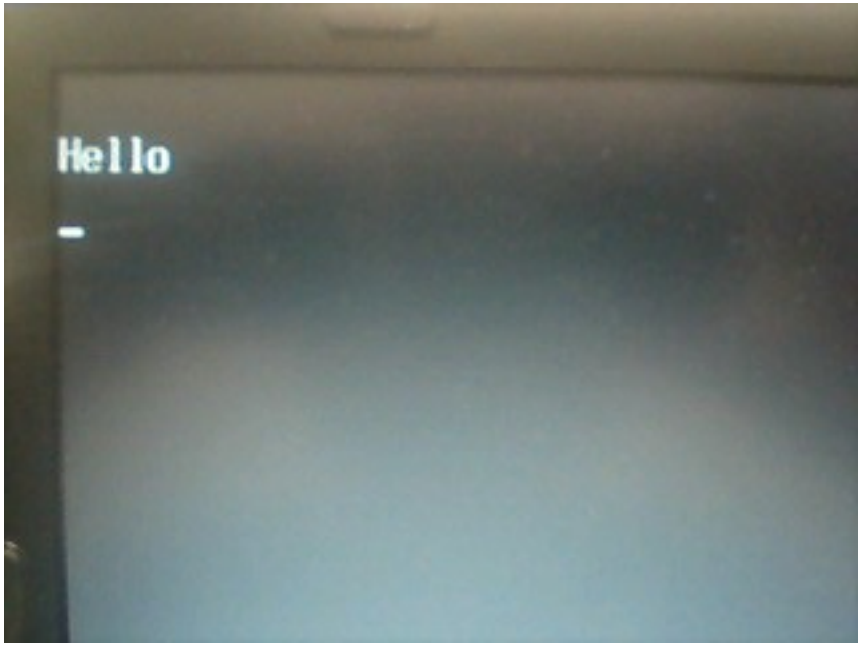
- 実機（PC/AT互換機）上で実行する.
  - ここではUSBブートで.
- BIOS起動画面でUSBブートを選択.
  - 選択できないパソコンもある.
- USBメモリのブートセクタにusb.img を書き込む.
  - dd if=usb.img of=/dev/sdb（Cygwinの場合）
    - 間違って /dev/sda に書き込むとCドライブのデータが壊れる.
  - デバイスファイル名（例：/dev/sdb）はマニュアルや, mountコマンド（取り扱い注意）で調べる.
- USBメモリをさしてパソコンを起動する.

BIOS起動画面

- 起動時に特定のキーを押すとBIOS起動画面になる.
  - 例：Let'snote CF-W8 では「F2」キー.
- ここでは起動オプション#1を「USBハードディスク」に設定.



boot-hello実行画面（実機）



地味です．そんなもんです．

boot-hello.s（前半）

デフォルトのアドレス・データのサイズを16ビットにする.

```
.code16
.text
start:
    ljmp $0x07c0, $start2
start2:
    movw %cs,    %ax
    movw %ax,    %ds
    movw %ax,    %ss

    movw $mesg,  %bp
```

%cs の値を 0x07c0 にするため（後述）

コードセグメント，データセグメント，
スタックセグメントを同一に設定.

%bp が文字列 "Hello¥0" を指す.

boot-hello.s（後半）

```
loop:
    movb (%bp),  %al
    cmpb $0,     %al
    je   exit
    movb $0xe,   %ah
    movb $0x12,  %bl
    int  $0x10
    incw %bp
    jmp  loop
exit:
    hlt
    jmp exit
mesg:
    .ascii "Hello¥0"
.org 510
    .word 0xaa55
```

1文字取り出して，%al にセット．

if %al=='¥0' then ラベルexitにジャンプ．

$0xeはテレタイプ式書き込みコマンド．

%bl には文字の色を指定（後述）．
（bochsだと文字に色がつかない）

BIOSコールのビデオサービスを
呼び出す（後述）．

ブートセクタシグニチャ．
これがセクタの最後の2バイトにないと
ブートセクタとして認識されない．

ご参考

デバッガ

- デバッグするためのツール.
  - プログラムを少しずつ実行しながら，内部動作（変数の値や関数呼び出しなど）をプログラマに表示する.
  - 例：GNUデバッガ(gdb)，Bochs内蔵デバッガ（後述）

主な機能

- プログラムの実行を中断
  - プログラム実行の中断箇所（＝ブレークポイント）を設定.
- ステップ実行
  - 少し（例：1行，1命令）だけ実行した後，実行を中断.
- 内部動作の表示
  - 変数の値の表示，コールスタックの表示など.

Bochs内蔵デバッガ（1）　使用例

```
<bochs:1> break 0x7C00
<bochs:2> cont
(0) Breakpoint 1, 0x0000000000007c00 in ?? ()
Next at t=153228546
(0) [0x00007c00] 0000:7c00 (unk. ctxt): jmp far 07c0:0005 ; ea0500c007
<bochs:3> step
Next at t=153228547
(0) [0x00007c05] 07c0:0005 (unk. ctxt): mov ax, cs ; 8cc8
<bochs:4> xp /xb 0x7C00
[bochs]:
0x0000000000007c00 <bogus+       0>:    0xea
<bochs:5> regs
rax: 0x00000000:0000aa55 rcx: 0x00000000:00000000
 （中略）
rip: 0x00000000:00000005
eflags 0x00000082: id vip vif ac vm rf nt IOPL=0 of df if tf SF zf af pf cf
<bochs:6> quit
```

ご参考

Bochs内蔵デバッガ（2）

主なコマンド一覧

コマンド	別名	説明
continue	c, cont	実行を再開（開始）する
quit	exit, q	デバッガの実行を終了する
help		コマンド一覧の表示
help コマンド名		そのコマンドの説明を表示
^C		実行を中止して，デバッガのプロンプトに戻る
vbreak *seg:offset*	vb	仮想アドレスにブレークポイントを設定
lbreak *linear-addr*	lb	リニアアドレスにブレークポイントを設定
pbreak *physical-addr*	pb, break, b	物理アドレスにブレークポイントを設定
blist	info break	ブレークポイントの一覧を表示
delete *breakpoint-num*	del, d	指定したブレークポイントを削除
step [*count*]	s, stepi	*count* 数（デフォルトは1）だけ命令を実行
next [*count*]	p, n	stepと同じだが，サブルーチンをまたぐ.
xp [/*nuf*] *physical-addr*		物理アドレスの内容を表示（/nufは次ページ）
x [/*nuf*] *linear-addr*		リニアアドレスの内容を表示
registers	regs, reg, r	レジスタの値の一覧を表示
page *linear-addr*		リニアアドレスに対応する物理アドレスを表示

ご参考

Bochs内蔵デバッガ（3）

x, xpコマンドのオプション

オプション	説明	指定可能な値
n	データ数	
u	データサイズ	b (byte), h (halfword), w (word), g (double word)
f	表示する形式	x (hexadecimal), d (decimal), u (unsigned), o (octal), t (binary), c (char), s (string), i (instruction)

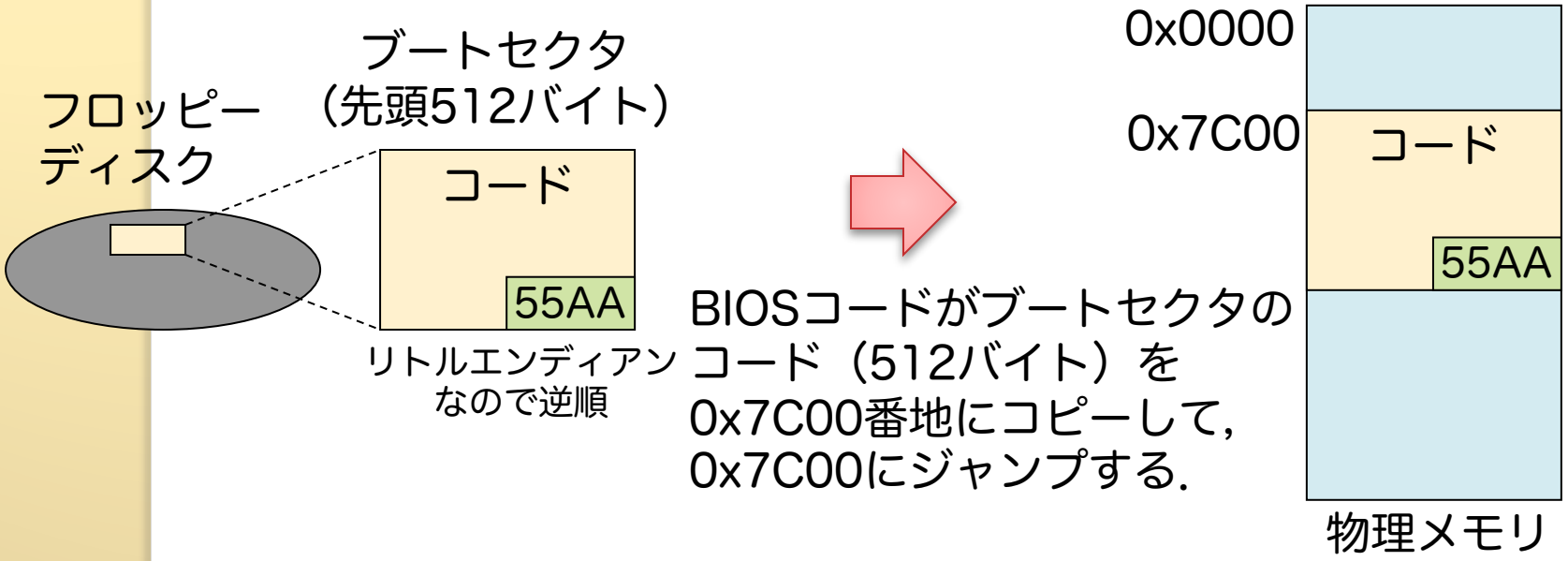
詳細は以下を参照せよ（英語）.
http://bochs.sourceforge.net/doc/docbook/user/internal-debugger.html

他にも多くのコマンドがある．例えば，watchコマンドは「このメモリの値が○○になったらブレーク」できる.

フロッピーディスクからのブート（0x7C00，0xAA55）

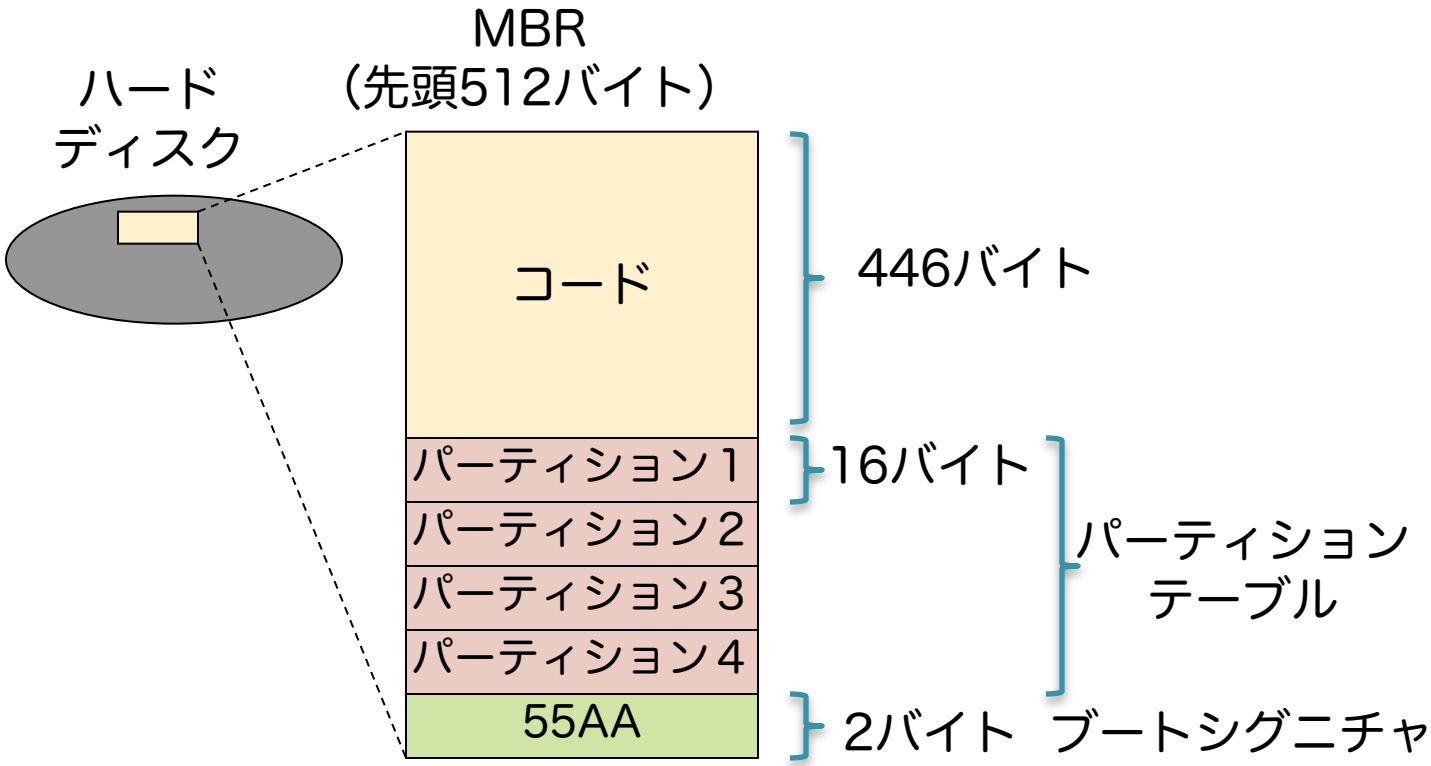
電源投入やリセット後，ROM中のブートコードをCPUが自動的に実行.

- BIOSコードが次のブート手順を自動的に実行.
  1. ブートセクタにブートシグニチャ0xAA55があるか確認.
  2. ブートセクタを0x7C00番地のメモリにコピー.
  3. 0x0000:0x7C00番地にジャンプ.



ハードディスクからのブート

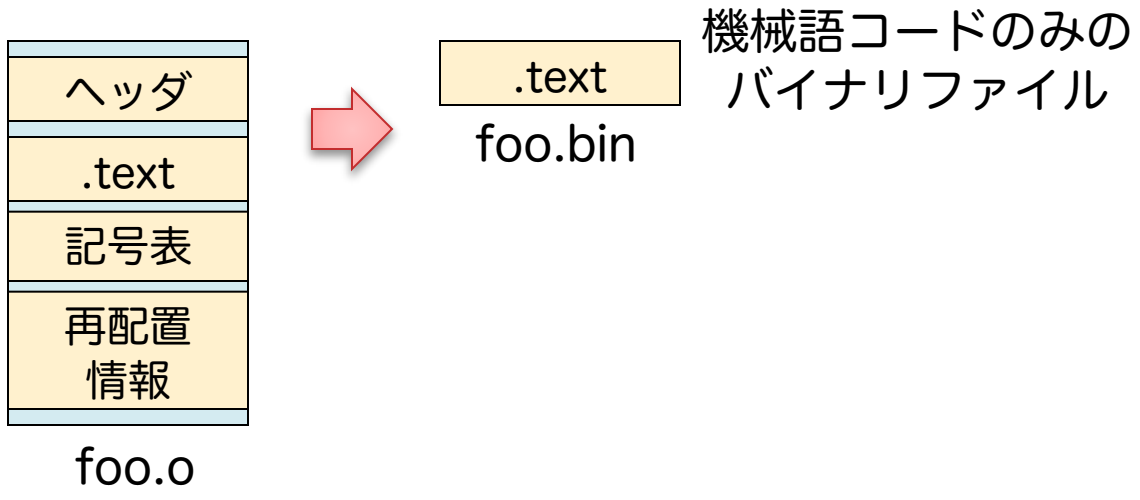
- 先頭セクタはMBR (master boot record).
- 「0x7C00番地にコピーしてジャンプ」という点で同じ.



objcopy -O binary

```
% gcc -c boot.s
% objcopy -O binary boot.o boot.bin
```

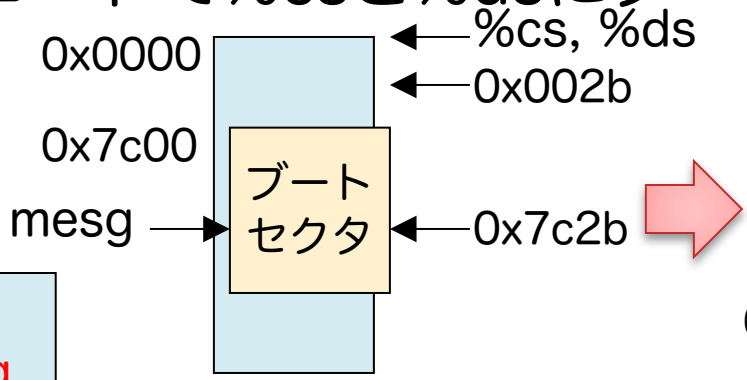
- ヘッダ，記号表，再配置情報などを捨てる．
- .textセクションのみのバイナリファイルを作れる．
  - .text中のコードが先頭ゼロ番地としてコンパイルされてるので（この場合はたまたま）うまくいく．
  - .dataセクションなど他のセクションがある場合はダメ．リンカースクリプト（後述）を使う必要がある．

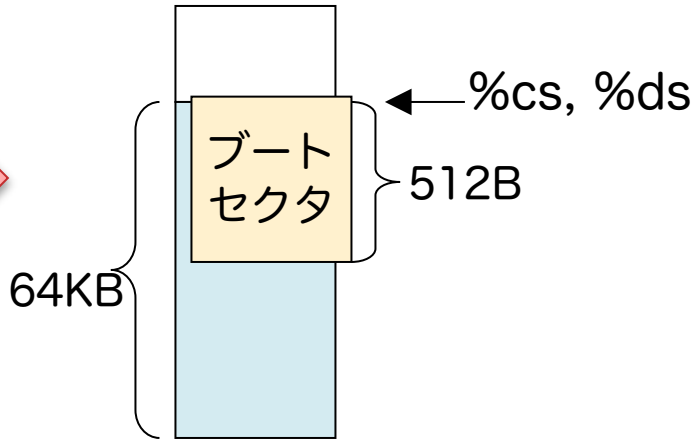


ljmp $0x07C0, $start2

```
start:
    ljmp $0x07c0, $start2
start2:
    movw %cs,    %ax
    movw %ax,    %ds
```

- アドレスはブートセクタ先頭を0番地とした相対アドレス.
  - このオブジェクトファイルでは.
- このコードで%csと%dsにブートセクタ先頭を指させる.






```
%nm boot.o
0000002b t mesg
00000000 t start
00000005 t start2
  (略)
```

%cs =0x0000
%eip=0x7c00
%ds =0x0000
mesg=0x002b

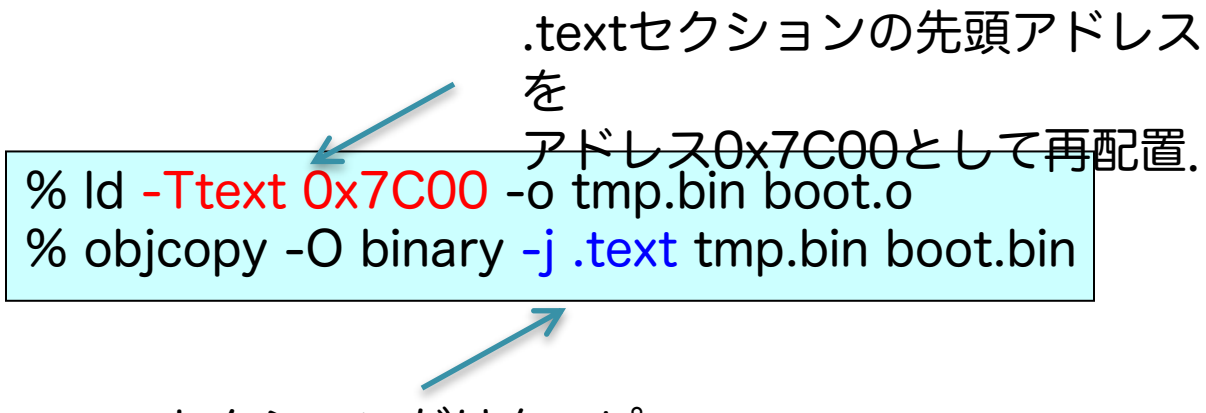
%ds:mesg
(0x002b)
は間違い.

%cs =0x07c0
%eip=0x0000
%ds =0x07c0
mesg=0x002b

%ds:mesg
(0x7c2b)
は正しい.

スタートアドレスの調整（1）

- 次をすれば「ljmp $0x07C0, $start2」は不要.



GNU ld はMach-O形式を未サポートなので,
Mac OS X上では実行できない.

スタートアドレスの調整（2）

- リンカスクリプトでも調整可能.
  - リンカスクリプトはバイナリファイル中やメモリ上でのセクションのレイアウトを細かく調整できる.

small.ls

```
SECTIONS {
    .text    0x7C00: AT (0x7C00) { *(.text) }
}
```

リンカスクリプト

```
% gcc -c boot.s
% ld -Tsmall.ls -o tmp.bin boot.o
% objcopy -O binary -j .text tmp.bin boot.bin
% dd bs=512 count=1 if=boot.bin of=floppya.img
```

GNU ld はMach-O形式を未サポートなので,
Mac OS X上では実行できない.

簡単なリンカスクリプトの読み方

- VMA (virtual memory address)
  - 再配置する（つまり実行時の）アドレスを指定.
- LMA (load memory address)
  - ロードするアドレスを指定．通常，VMAと一致.

```
SECTIONS {
    .text    0x10000: AT (0x10000) { *(.text) }
    .data    0x20000: AT (0x20000) { *(.data) }
}
```

出力するセクション名　VMA　LMA　*(.data) は入力ファイル中のすべての.dataセクションの意味

このアドレスはセグメント内オフセット.

GNU ld はMach-O形式を未サポートなので,
Mac OS X上では実行できない.

16ビットのi386アセンブリコードからC関数を呼び出す（1）

boot.s（前半）

```
.code16
.text
  # load cmain's .text
  movw $0x1000, %ax
  movw %ax,   %es
  movw $0x0000, %bx
  movb $0x00, %dl
  movb $0x00, %dh
  movb $0x00, %ch
  movb $0x02, %cl
  movb $1,    %al
  movb $0x02, %ah
  int  $0x13
```

cmain.cの.textを0x1000:0000にロード.

.textはセクタ#2にある.

BIOSのフロッピーサービスを呼び出す.

boot.s（後半）

```
  # load cmain's .data
  movw $0x1000, %ax
  movw $0x0200, %bx
  movw %ax,     %es
  movb $0x00, %dl
  movb $0x00, %dh
  movb $0x00, %ch
  movb $0x03, %cl
  movb $1,    %al
  movb $0x02, %ah
  int  $0x13

  movw $0x1000, %ax
  movw %ax,     %ds
  movw %ax,     %ss
  movl $0xFFF0, %esp
  movl $0xFFF0, %ebp
  ljmp $0x1000, $0x0000
.org 510
  .word 0xaa55
```

cmain関数を呼び出す.

GNU ld はMach-O形式を未サポートなので,
Mac OS X上では実行できない.

16ビットのi386アセンブリコードからC関数を呼び出す（2）

small.ls

```
SECTIONS {
   .text 0x00000: AT (0x00000) { *(.text) }
   .data 0x00200: AT (0x00200) { *(.data) }
}
```

16ビットモードでもGCCは32ビットでスタック操作するので.code16ではなく，.code16gccが必要.

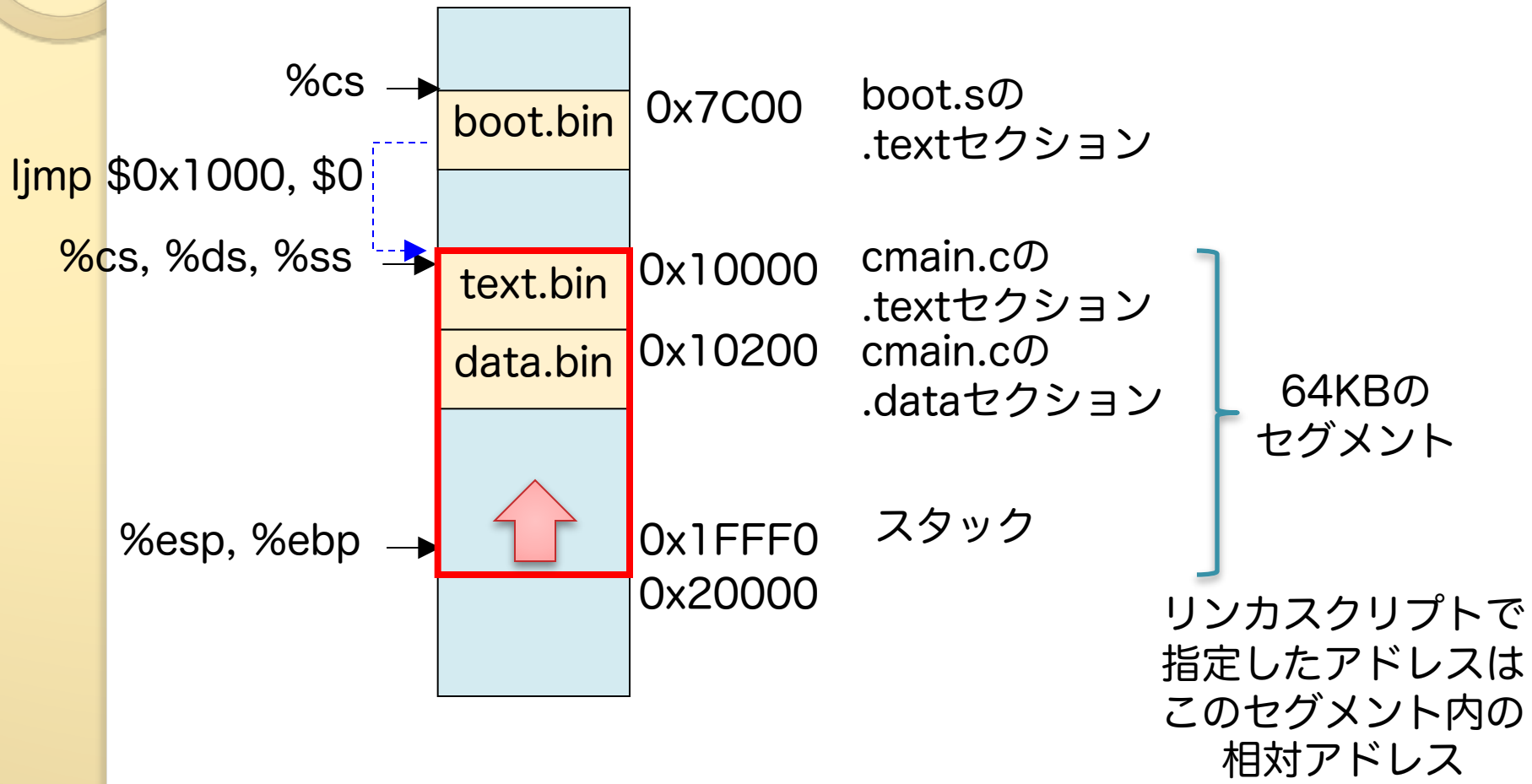
```
asm (".code16gcc");
char xxx = 'Q';
int fact (int n);
int cmain (void)
{
  char tmp;
  asm volatile ("movb %0, %%al"::"m"(xxx));
  asm volatile ("movb $0x0E, %ah; int $0x10;"); /* 'Q' */
  tmp = fact (5);
  asm volatile ("movb %0, %%al"::"m"(tmp));
  asm volatile ("movb $0x0E, %ah; int $0x10;"); /* 'x' == 120 */
  asm volatile ("1: hlt; jmp 1b;");
}
int fact (int n)
{
  if (n <= 0)
    return 1;
  else
    return n * fact (n - 1);
}
```

cmain.c

mainという名前にするとGCCが余計なコードを付けるのでcmain にしてある.

```
% gcc -c boot.s
% ld -Ttext 0x7C00 -o tmp.bin boot.o
% objcopy -O binary -j .text tmp.bin boot.bin
% gcc -c cmain.c
% ld -Tsmall.ls -o cmain.bin cmain.o
% objcopy -O binary -j .text cmain.bin text.bin
% objcopy -O binary -j .data cmain.bin data.bin
% dd bs=512             count=1 if=boot.bin of=floppya.img
% dd bs=512 seek=1 count=1 if=text.bin  of=floppya.img
% dd bs=512 seek=2 count=1 if=data.bin of=floppya.img
```

16ビットのi386アセンブリコードから C関数を呼び出す（3）



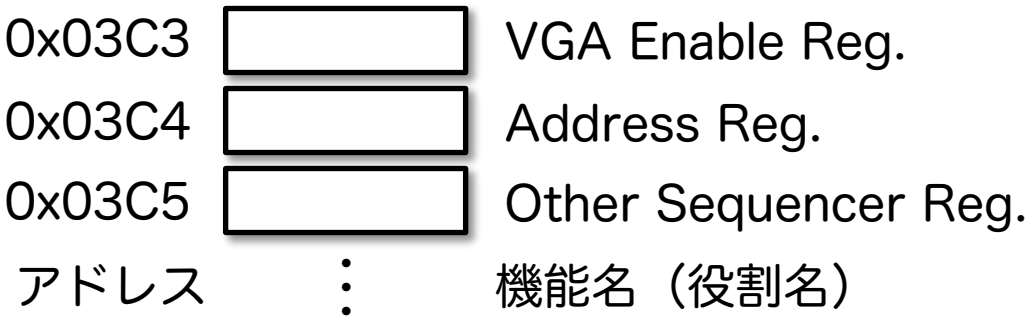
ハードウェア（I/Oデバイス）

プログラマから見たI/Oデバイス

- I/Oデバイス（I/O device，入出力装置）
  - キーボード，ディスプレイ，ハードディスク，プリンタ･･･
- プログラマからはI/Oレジスタ群に見える．
  - I/OレジスタはI/Oデバイス・コントローラ中にある．
  - I/Oレジスタは（名前ではなく）アドレスで区別する．

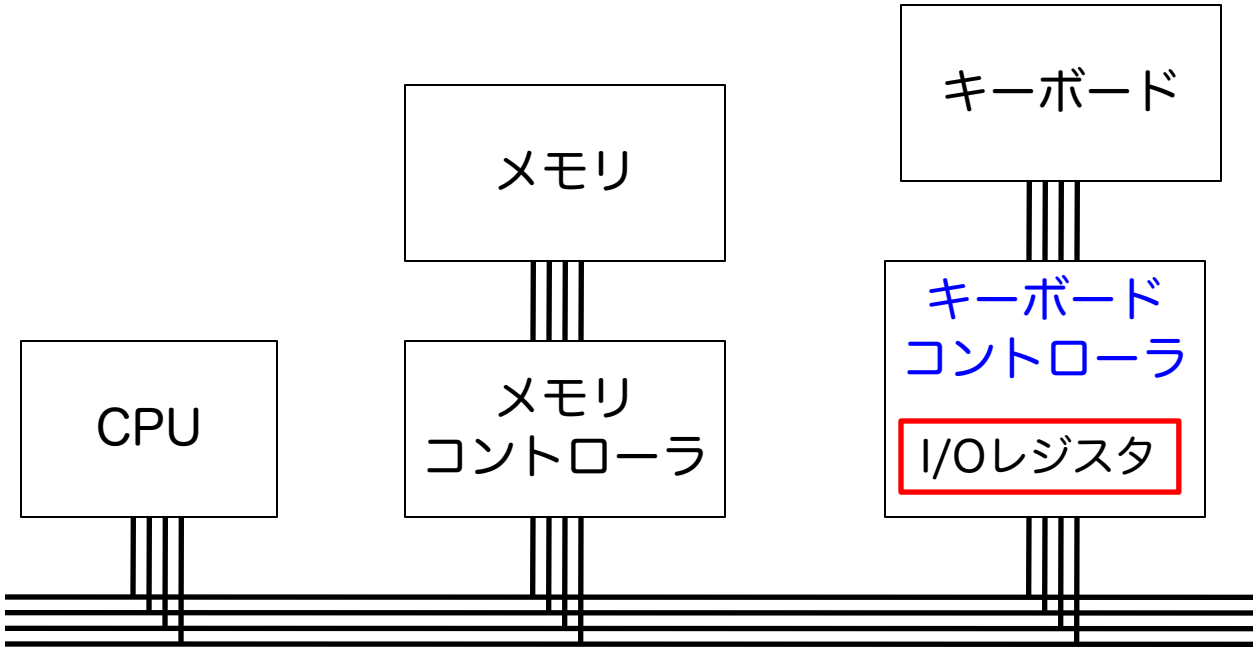
物理的なディスプレイ

ディスプレイの
I/Oレジスタ群



I/Oデバイス・コントローラ

- CPUと周辺機器を橋渡しするチップ.
- 中にI/Oレジスタがある.
  - I/OレジスタはI/Oポートとも呼ぶ.



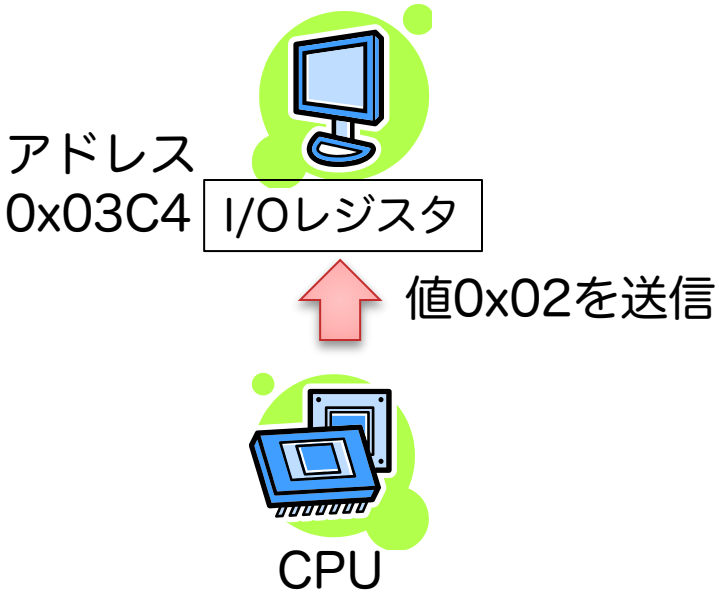
I/Oレジスタの基本

I/Oレジスタの
アドレスを指定して

- データを書く＝I/Oレジスタにデータ送信.
- データを読む＝I/Oレジスタからデータ受信.
  - i386ではmov命令やin/out命令（後述）を使う.

例：ビデオコントローラの
I/Oレジスタにデータ送信

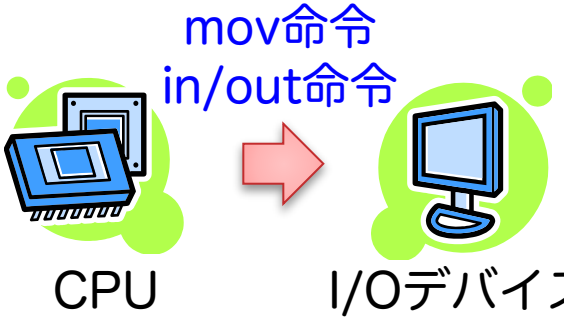
```
movw $0x03C4, %dx
movb $0x02,   %al
outb  %al,    %dx
```

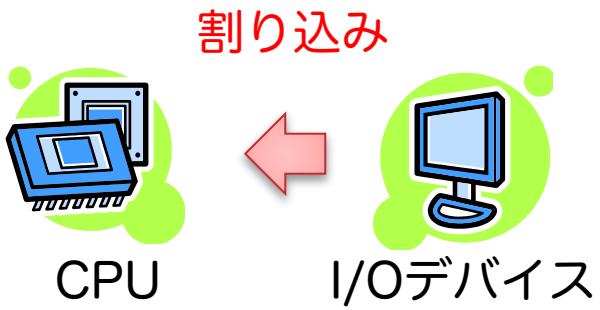


I/Oデバイスの制御の概要

この２つを組み合わせて制御する

- CPU→I/Oデバイス
  - CPUがmov命令やin/out命令で，命令やデータを送受信する．
- I/Oデバイス→CPU
  - I/OデバイスがCPUに割り込んで，I/Oデバイスの状態変化を伝える．それに応じてCPUが割り込みハンドラを実行する．

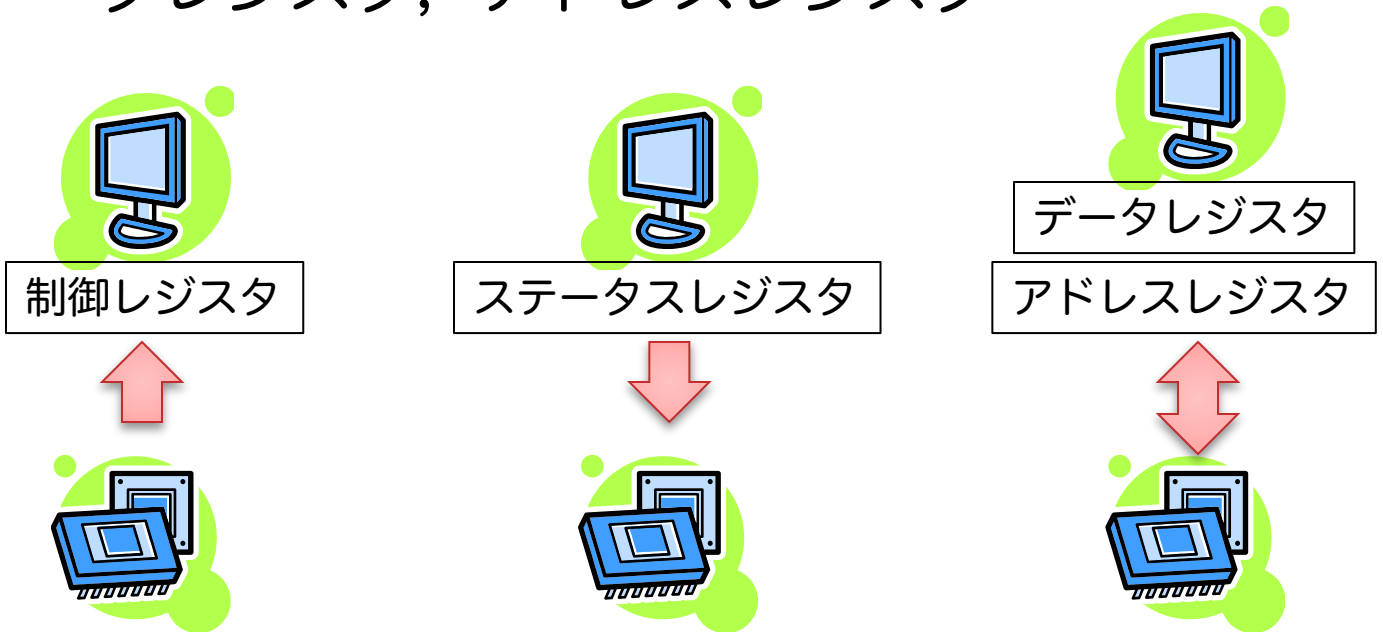




I/Oレジスタの種類

線引きは明確ではない．明確に
分類できないI/Oレジスタも多い．

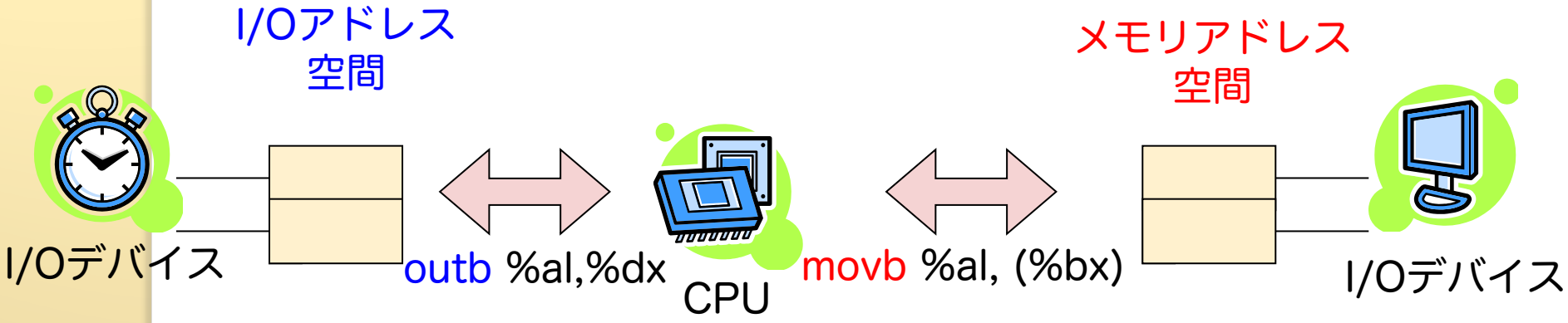
- 制御レジスタ（コマンドレジスタ）
- ステータスレジスタ
- データレジスタ，アドレスレジスタ



たとえ：アメリカの住所と日本の住所では
同じ番地でも違う場所になる

I/O mapped I/O と memory mapped I/O

- I/Oレジスタのアドレスは２種類ある.
  - I/Oポート（I/O用のアドレス）
  - メモリのアドレス
- 前者をI/O-mapped I/O といい,
  後者をmemory-mapped I/O という.

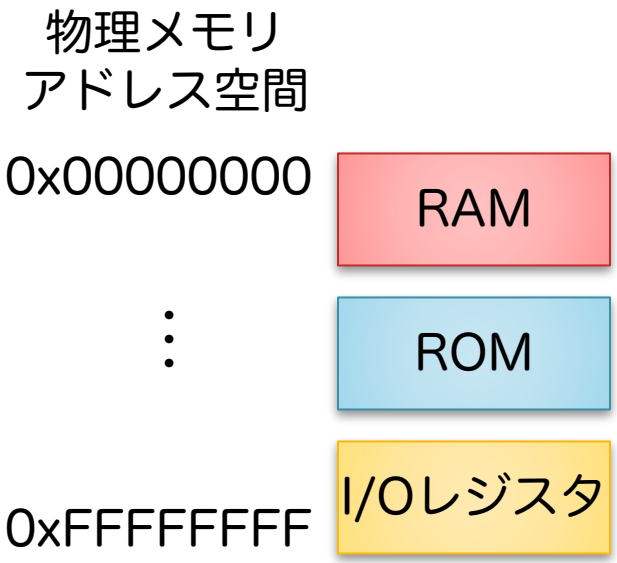


I/O-mapped I/O

memory-mapped I/O

memory-mapped I/O

- 物理メモリアドレスは次の３つを指す（マップする）.
  - RAM
  - ROM
  - I/Oレジスタ （memory-mapped I/O）



注：この図は説明用であり，
実際はこの順序とは限らない.

起動時のデフォルトはモード３（英数字モード）．
グラフィクスモードではない．

memory-mapped I/Oの例：VRAM

- Video RAM (VRAM)にデータを書くことで，VGAディスプレイの座標(10,2)に文字Aを表示．
  - VRAMはmemory-mappedなので，mov命令で書く．

```
.code16
.text
  ljmp $0x07c0, $start2
start2:
  movw %cs,    %ax
  movw $0xB800, %cx
  movw %cx,    %es
  movw $340,   %bx
  movb $'A,    %es:(%bx)
  incw %bx
  movb $0x0C, %es:(%bx)
  exit: hlt; jmp exit
.org 510
  .word 0xaa55
```

モード3のVRAMの先頭アドレスは0xB8000.
座標(10,2)．340=(80*2+10)*2.
ASCII文字 A

文字属性バイト（明るい赤）

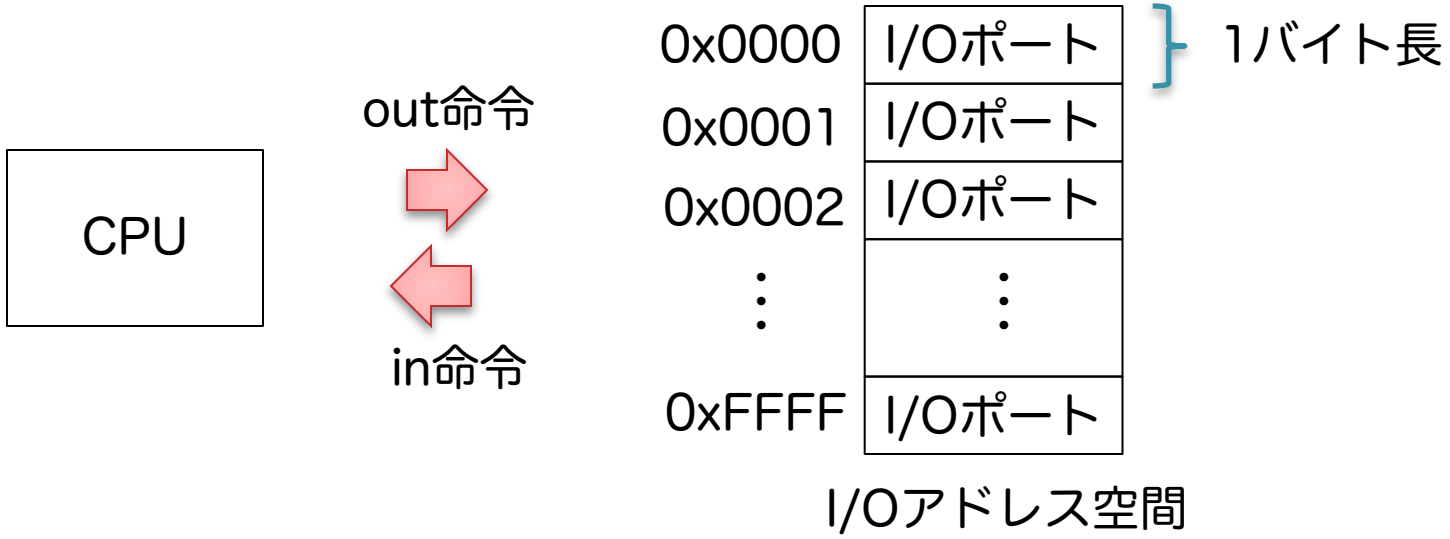
実行

vram

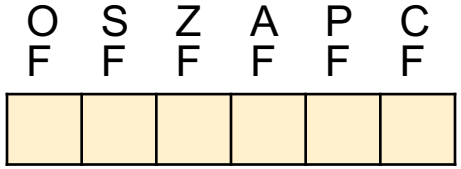
I/O-mapped I/O用のアドレス空間

i386のI/Oポート（I/Oアドレス空間）

- 入出力用のアドレス空間.
  - アドレスは16ビット長．0x0000～0xFFFFの範囲.
  - 各アドレスには1バイトのI/Oポート（I/Oレジスタ）を対応づける（マップ可能）.
- メモリのアドレス空間とは別.
- in/out命令でI/Oアドレスを指定してデータを送受信.



in, out命令



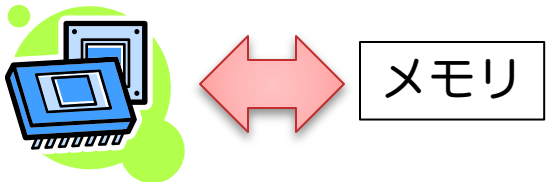
- in命令はI/Oポートからデータを%(e)axや%alに読む.
- out命令は%(e)axや%al中のデータをI/Oポートに書く.
- I/Oポートは *imm8* か %dx で指定する.

文法	例	説明
in *imm8*, %al	inb $10, %al	
in *imm8*, %(e)ax	inw $10, %ax	
in %dx, %al	inb %dx, %al	
in %dx, %(e)ax	inl %dx, %eax	
out %al, *imm8*	outb %al, $10	
out %(e)ax, *imm8*	outw %ax, $10	
out %al, %dx	outb %al, %dx	
out %(e)ax, %dx	outl %eax, %dx	

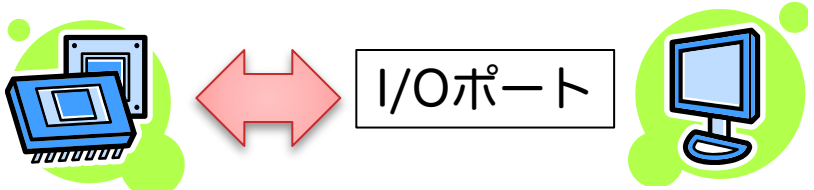
I/Oレジスタの性質（1）

アドレスを指定して（mov命令などで）
読み書きできる点では同じだけど・・・

- I/Oレジスタは，メモリや通常のレジスタと異なる．
  - 書き込み専用や読み込み専用のI/Oレジスタがある．
  - 書いてから読むと，値が一致しないことがある．
  - 読み込むことで値が変化することがある．
    - 例：読むと自動的に値がクリアされる．
  - 同じ値を２度書き込むことに意味がある場合がある．
- I/Oレジスタは記憶装置ではなくデータの受け渡し口（ポート）だから．



書いた値はその後で
何度読んでも同じ．



メモリとは異なる
動作をする．

I/Oレジスタの性質（2）

- 同じアドレスが複数のI/Oレジスタを指すことがある.
- アクセス前の状態で，1つのI/Oレジスタを選択.

例：ビデオコントローラーのI/Oレジスタ（一部）

I/Oアドレス
（I/Oポート番号）

0x03C5にアクセスする前に，0x03C4に
インデックス番号を書き込んで，0x03C5の
I/Oレジスタを選択する.

0x03C4		Sequencer Address Reg.
0x03C5		Reset Reg. (index=0)
0x03C5		Clocking Mode Reg. (index=1)
0x03C5		Map Mask Reg. (index=2)
0x03C5		Character Map Select Reg. (index=3)
0x03C5		Memory Mode Reg. (index=4)

I/O-mapped I/Oの例：キーボード

- キーボードコントローラのI/Oレジスタに書き込むことで，Num Lock LEDをオンにする．
  - キーボードはI/O-mappedなので，in/out命令で読み書きする．

```
.code16
.text
  ljmp $0x07c0, $start2
start2:
  movw %cs,   %ax
  movb $0xED, %al
  movw $0x60, %dx
  outb %al,   %dx
  movw $0x64, %dx
wait:
  inb %dx, %al; test $2, %al; jnz wait
  movw $0x60, %dx
  movb $2,    %al
  outb %al,   %dx
exit: hlt; jmp exit
.org 510
  .word 0xaa55
```

num-lock

LEDセット・リセットコマンドを送信

キーボード側の準備待ち
（IBFが0になるまで待つ）

Num Lockを指定

ここではKBDCからの
Ack確認をサボっている

Cygwin上のBochsでの動作画面

- 確かにNum Lockがオンになっている.