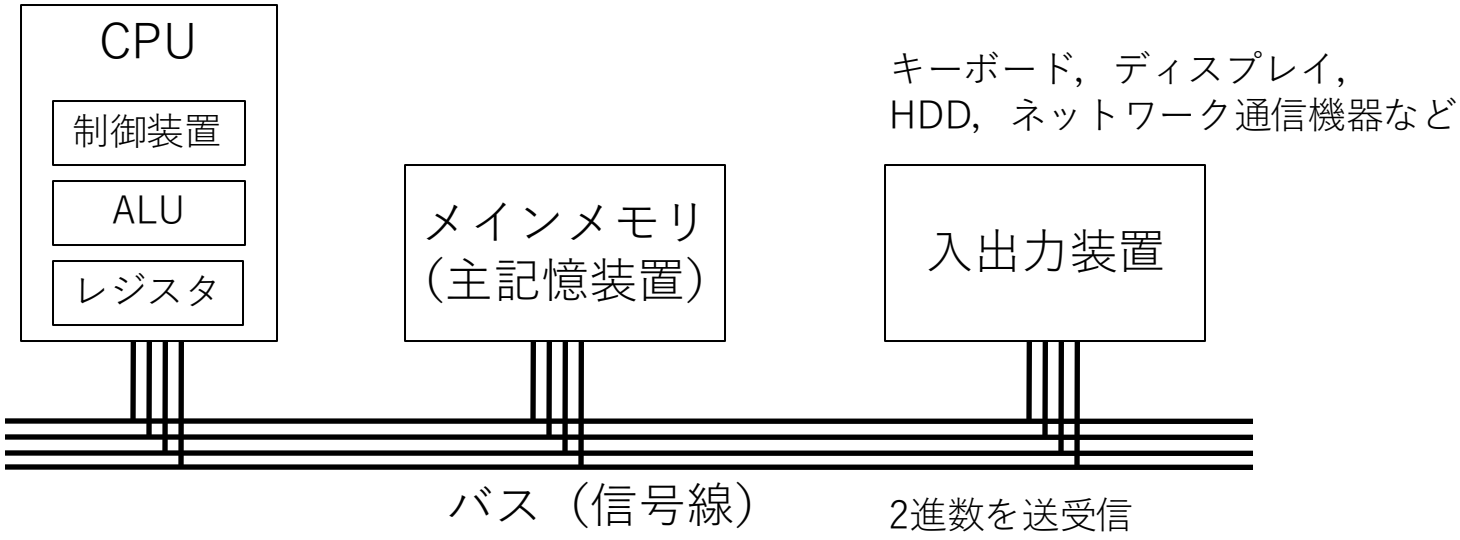
アセンブリ言語

x86-64 の基本構造（プログラマの視点）

情報工学系
権藤克彦

コンピュータの基本構造

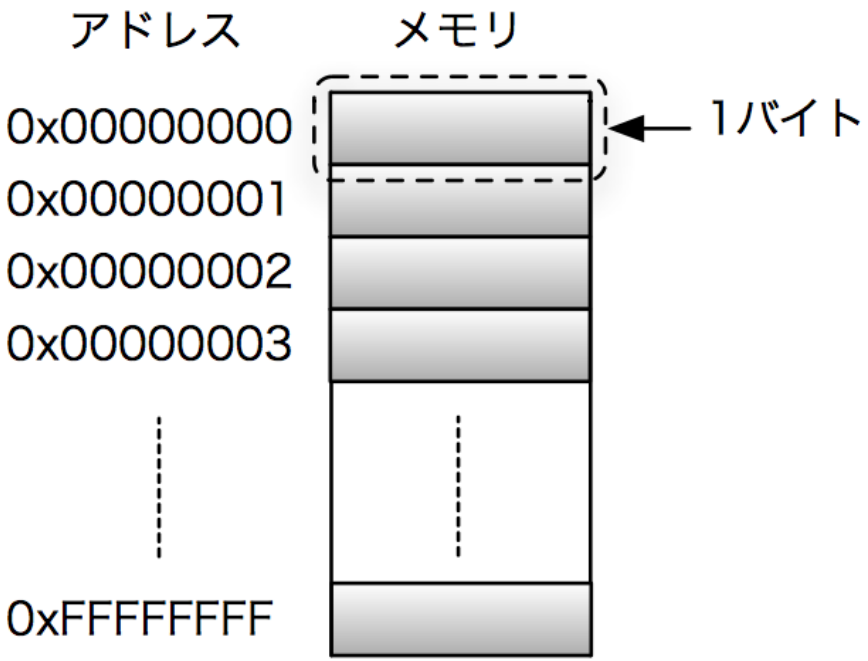
- CPU（中央処理装置，プロセッサ）
  - 制御装置，ALU（演算装置），レジスタから構成.
- メインメモリ（主記憶）　以後，メモリと略す.
- 入出力装置
- バス(bus)



メモリ（memory）

一般論（つまり例外あり）

- RAMを使用.
  - 揮発性(volatile)，読み書き可能，ランダムアクセス可能.
- メモリはバイトの（巨大な）配列.
  - メモリの各バイトはアドレスで指定して，読み書きする.
- アドレス
  - アドレスは各バイトを一意に指定する通し番号.
  - バイトアドレッシング方式(byte-addressing).

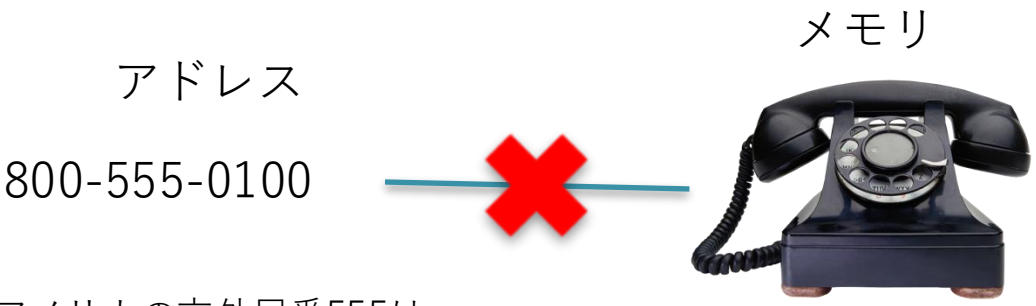


「1バイト毎にアドレスがある」こと

アドレス空間

- アドレス空間 (address space)
  - メモリのすべてのアドレスの集合.
- 例：32ビットアドレス空間.
  - アドレスは0番地から $(2^{32} - 1)$番地（16進数で 0xFFFFFFFF）.
  - アドレス長は4バイト．4GB分のメモリを扱える.

- 注意：メモリのないアドレスもある．アクセス不可.
  - 電話がつながってない電話番号と同じ.



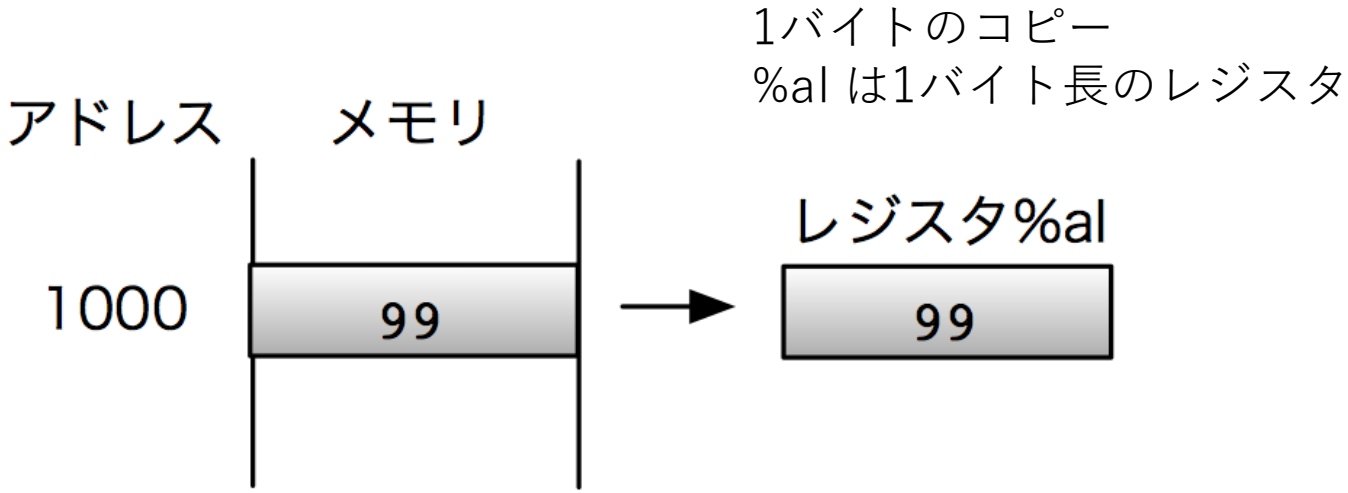
アメリカの市外局番555は
フィクション用の架空番号

メモリアクセスの例

- 1000番地のメモリに99が入っている状態で,

  movb 1000, %al

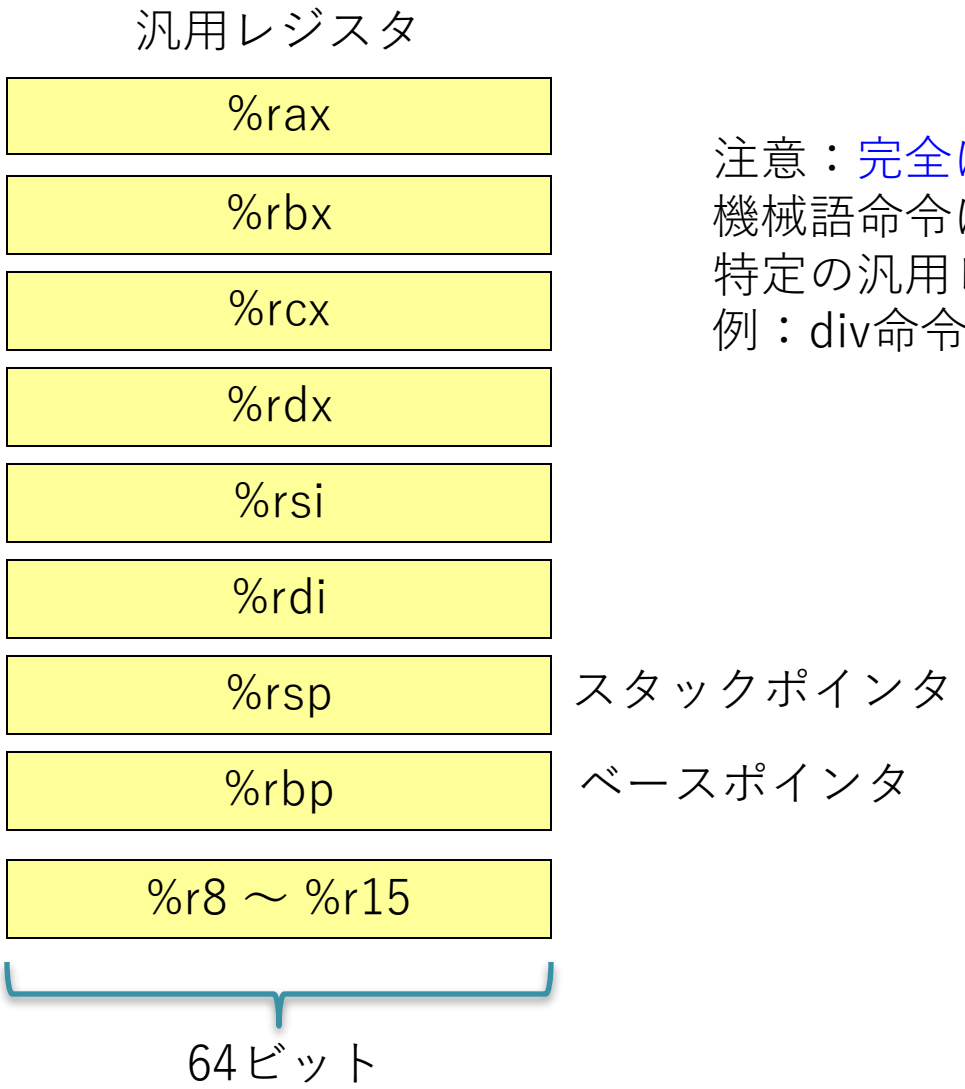
  を実行すると，レジスタ%alに値99がコピーされる.



レジスタ (register)

- CPU内の少量で高速なメモリ．
- レジスタにアドレスはない．名前で指定する．
  - 例：%rax
- いろいろ種類がある．
  - 専用レジスタ，汎用レジスタ，システムレジスタ

x86-64のレジスタ：アプリケーション用（1）



注意：完全に汎用ではない.
機械語命令によっては
特定の汎用レジスタを要求する.
例：div命令は%raxと%rdxを使う.

AT&Tスタイルの表記に従って,
レジスタ名にパーセント記号（%）を付記.

x86-64のレジスタ：アプリケーション用（2

ステータスレジスタ
（フラグレジスタ）

%rflags

プログラムカウンタ

%rip

64ビット

セグメントレジスタ

%cs コードセグメント

%ds データセグメント

%ss スタックセグメント

%es

%fs

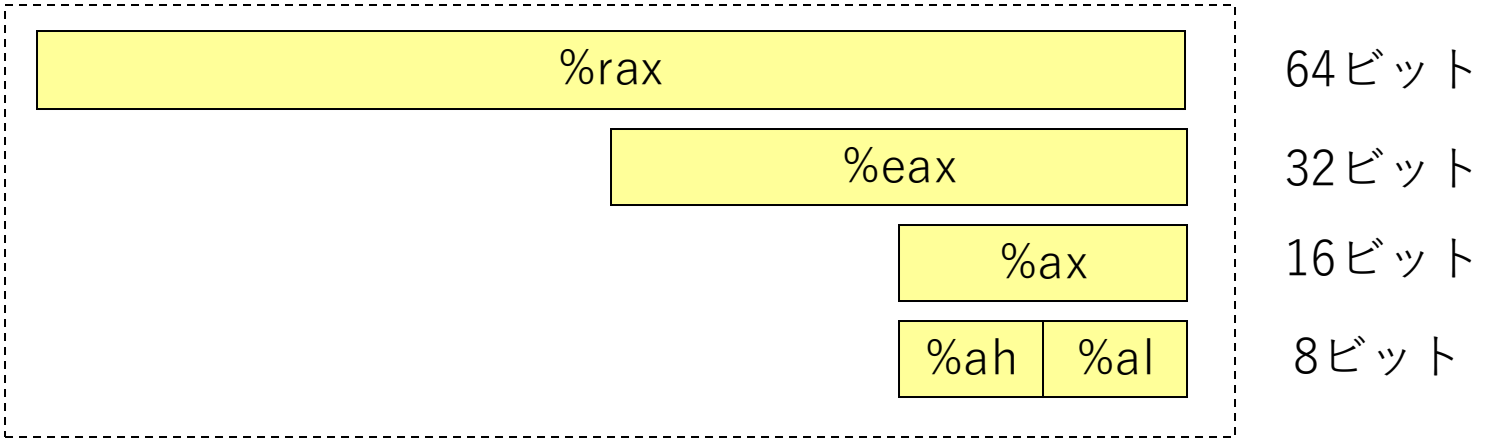
%gs

16ビット

64ビットモードではセグメントレジスタの役割は非常に小さい

汎用レジスタの別名（1）

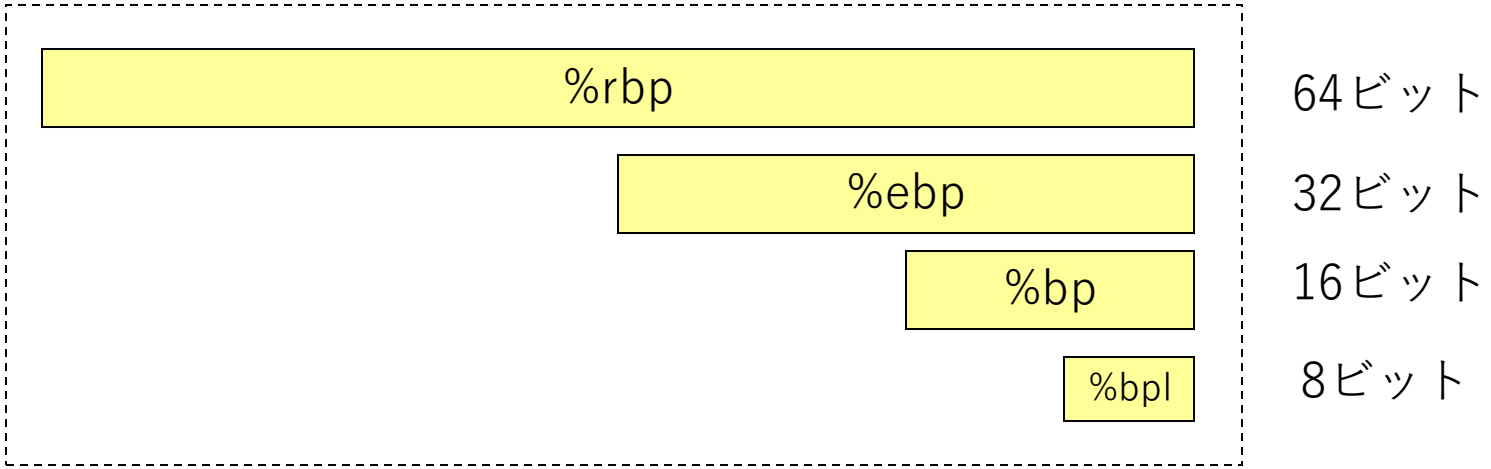
- %raxの別名：
  - %raxの下位32ビットは%eaxとしてアクセス可.
  - %eaxの下位16ビットは%axとしてアクセス可.
  - %axの上位8ビットは%ahとしてアクセス可.
  - %axの下位8ビットは%alとしてアクセス可.



%rbx, %rcx, %rdxも同様.

汎用レジスタの別名（2）

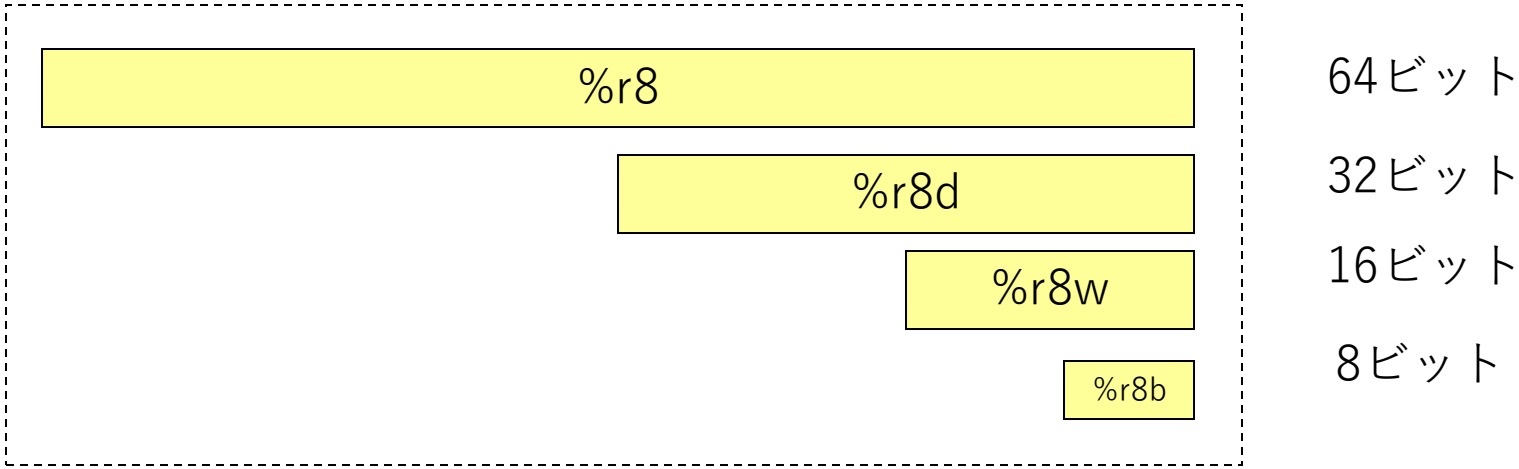
- %rbpの別名：
  - %rbpの下位32ビットは%ebpとしてアクセス可.
  - %rbpの下位16ビットは%bpとしてアクセス可.
  - %rbpの下位8ビットは%bplとしてアクセス可.



%rsp, %rsi, %rdiも同様.

汎用レジスタの別名（3）

- %r8の別名：
  - %r8の下位32ビットは%r8dとしてアクセス可.
  - %r8の下位16ビットは%r8wとしてアクセス可.
  - %r8の下位8ビットは%r8bとしてアクセス可.

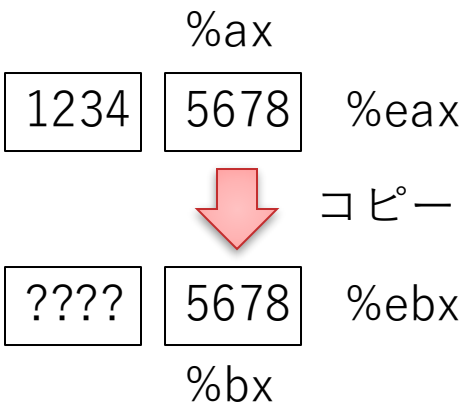


%r9 ～ %r15も同様.

汎用レジスタの別名（例）

デバッグ情報を付加するために，
-gオプションをつける．

```
.text
.globl _main
_main:
movl $0x12345678, %eax
movw %ax, %bx
ret
```



%eaxに0x12345678を代入した後，
%axを読むと0x5678 が得られる．

```
% gcc –g reg-alias.s
% gdb ./a.out
(gdb) break main
(gdb) run
Breakpoint 1, _main () at alias.s:4
4           movl $0x12345678, %eax
(gdb) stepi
5           movw %ax, %bx
(gdb) stepi
6           ret
(gdb) print /x $ebx
$1 = 0xbfff5678
(gdb) quit
The program is running.
Exit anyway? (y or n) y
%
```

ブレークポイントの設定

実行開始

1命令ずつステップ実行

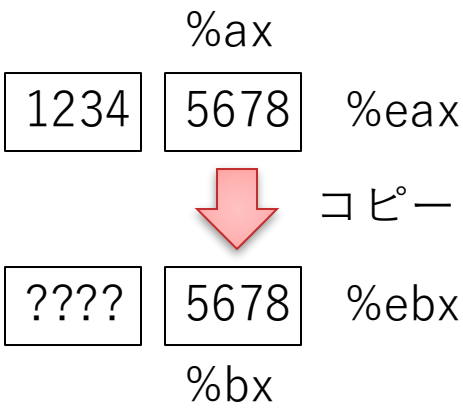
レジスタ%ebxの値を表示．
/x は16進表示を指定．
レジスタ名にはドル記号
（$）をつける．

デバッガgdbを使ってレジスタ%ebxの値を確認．

汎用レジスタの別名（例）

デバッグ情報を付加するために，
-gオプションをつける．

```
.text
.globl _main
_main:
movl $0x12345678, %eax
movw %ax, %bx
ret
```



%eaxに0x12345678を代入した後，
%axを読むと0x5678 が得られる．

```
% gcc -g reg-alias.s
%  lldb ./a.out
(lldb) b main
Breakpoint 1: where = a.out`main, address =
0x00001faf
(lldb) run
a.out`main:
->  0x1faf <+0>: movl   $0x12345678, %eax
(lldb) si
->  0x1fb4 <+5>: movw   %ax, %bx
(lldb) si
->  0x1fb7 <+8>: retl
(lldb) register read bx
      bx = 0x5678
(lldb) quit
Do you really want to proceed: [Y/n] y
%
```

ブレークポイントの設定

実行開始

1命令ずつステップ実行

レジスタ%bxの値を表示

プログラムカウンタ %rip

- 次に実行する機械語命令のアドレスを保持.
- 通常の実行では，次の命令を指すように値が増加.
  - フェッチ実行サイクルの中で，制御装置が自動的に加算.

- 直接，movq命令でのアクセスは不可.
  - 例：movq $0x12345678, %rip とは書けない.
- 間接的に，制御命令でアクセス.
  - jmp命令，jcc命令，call命令，ret命令など.

プログラムカウンタ %rip （例）（１）

```
  .text
.globl _main
_main:
  pushq %rbp
  movq  %rsp, %rbp
  jmp _main
  leave
  ret
```

```
% gcc  -g rip.s
% lldb ./a.out
(lldb) d -n main
a.out`main:
a.out[0x100000fb0] <+0>: pushq  %rbp
a.out[0x100000fb1] <+1>: movq   %rsp, %rbp
a.out[0x100000fb4] <+4>: jmp    0x100000fb0
a.out[0x100000fb6] <+6>: leave
a.out[0x100000fb7] <+7>: retq
```

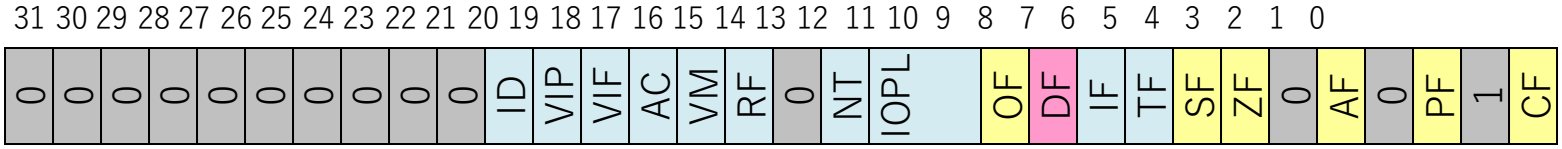
逆アセンブル

プログラムカウンタ %rip （例）（２）

```
(lldb) b main
Breakpoint 1: where = a.out`main, address = 0x0000000100000fb0
(lldb) run
->  0x100000fb0 <+0>: pushq  %rbp
(lldb) si
->  0x100000fb1 <+1>: movq   %rsp, %rbp
(lldb) si
->  0x100000fb4 <+4>: jmp    0x100000fb0
Target 0: (a.out) stopped.
(lldb) register read rip
    rip = 0x0000000100000fb4  a.out`main + 4
(lldb) si
->  0x100000fb0 <+0>: pushq  %rbp
(lldb) register read rip
    rip = 0x0000000100000fb0  a.out`main
(lldb)
```

ステータスレジスタ %rflags（1）

- 演算結果の状態を自動的に（ビット毎に）保持する.
  - 例：オーバーフロー，キャリー，演算結果の正負や真偽.
- プログラマは読むだけ．通常は書き込まない.
- 主に条件付きジャンプで使う.



ここではステータスフラグだけを扱う.

ステータスフラグ

制御フラグ

システムフラグ

予約（使用禁止．読んだ値と同じ値を必ずセットすること）

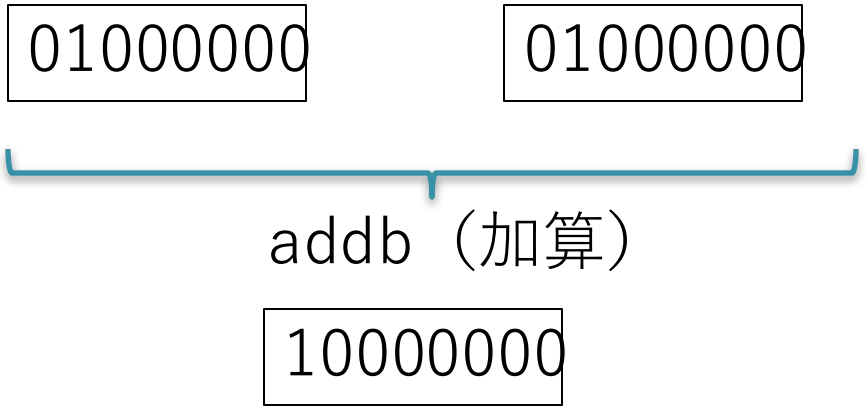
上位32ビットは不使用（予約）

ステータスレジスタ %rflags（2）

フラグ	名前	説明
CF	キャリー フラグ	算術演算で結果の最上位ビットにキャリーかボローが生じるとセット．それ以外はクリア． 符号なし整数演算でのオーバーフロー状態を表す．
OF	オーバー フロー フラグ	符号ビットを除いて，整数の演算結果が大きすぎるか小さすぎるとセット．それ以外はクリア． 2の補数の符号あり整数演算でのオーバーフロー状態を表す．
ZF	ゼロ フラグ	結果がゼロの時にセット．それ以外はクリア．
SF	符号 フラグ	符号あり整数の符号ビット(MSB)と同じ値をセット． （0は正の数，1は負の数を表す．）
PF	パリティ フラグ	結果の最下位バイトに値1のビットが偶数個あればセット． 奇数個であればクリア．
AF	調整 フラグ	算術演算で，結果のビット3にキャリーかボローが生じるとセット．それ以外はクリア．BCD演算で使用する．

ステータスレジスタ %rflags（３）

- 例：movb $64, %al; addb $64, %al



- CF=0 キャリーもボローも発生して無い.
- OF=1　128は8ビット符号あり整数でオーバーフロー.
- SF=1　MSB=1.　128は符号あり整数では負の数.
- ZF=0　128はゼロではない.

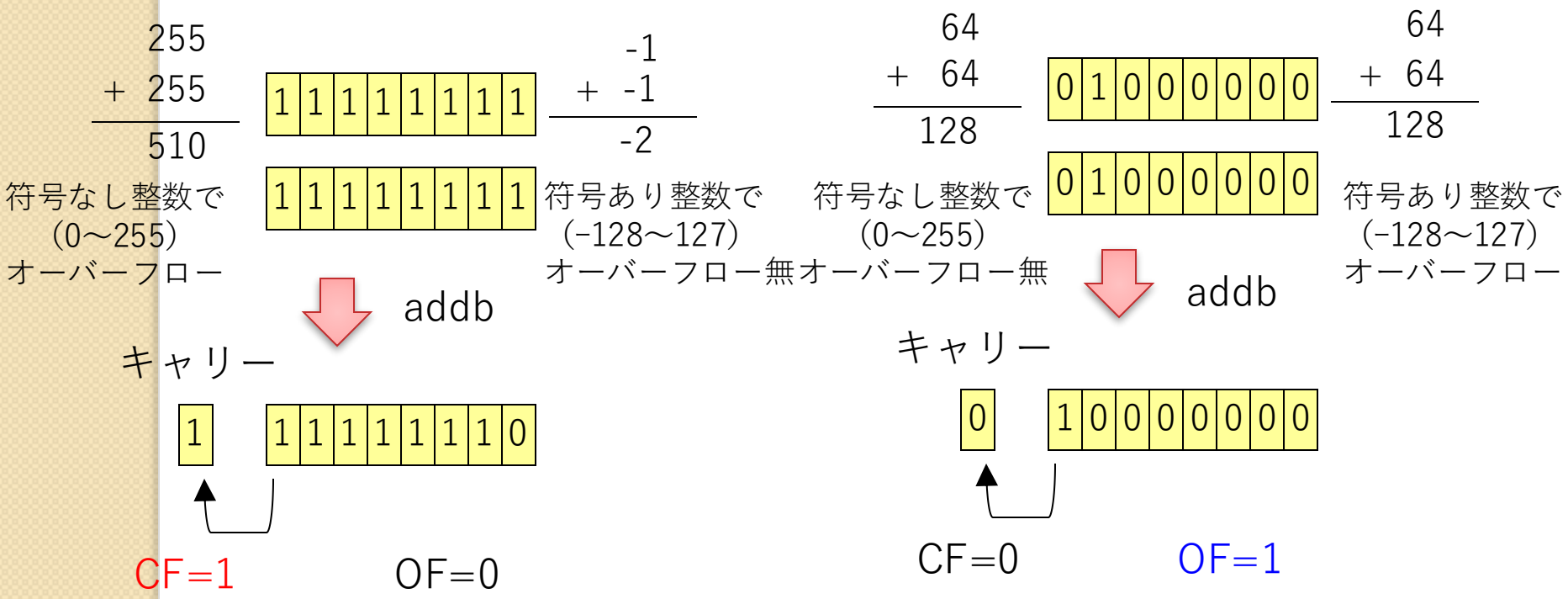
ステータスレジスタ %rflags（４）

```
% gcc -g rflags.s
% lldb ./a.out
(lldb) b main
Breakpoint 1: where = a.out`main, address = 0x00001fad
(lldb) run
->  0x1fad <+0>: movb   $0x40, %al
(lldb) si
->  0x1faf <+2>: addb   $0x40, %al
(lldb) register read --format binary rflags
  rflags = 0b00000000000000000000001001000110
(lldb) si
->  0x1fb1 <+4>:  subb   $-0x80, %al
(lldb) register read --format binary rflags
  rflags = 0b00000000000000000000101010000010
```

OF=1, SF=1, ZF=0, CF=0

ステータスレジスタ %rflags（5）

- キャリーフラグとオーバーフローフラグの違い
  - CF=1 は符号なし整数のオーバーフローを表す.
  - OF=1 は符号あり整数のオーバーフローを表す.



ステータスレジスタ %rflags（6）

- %rflags の各フラグは，最後の命令の実行結果に従って，セット・クリアされる．
- フラグの値は条件付きジャンプ命令で使う．
  - 特に，cmp命令とtest命令．

```
cmpl   $0, 8(%rbp)
jg     L2
```

if 8(%rbp)>0 then
ラベルL2にジャンプ

- cmpl op1, op2 は，引き算（op2-op1）のフラグ変化のみ計算．
- jg命令は条件「より大きい」が成り立てばジャンプ．
  - 符号あり整数に対して使う．
  - 条件：ZF==0 && SF==OF

なぜ SF==OF？

- OF=0 （オーバーフローなし）
  - SF=0（結果が正）ならば，op2 – op1 >= 0.
- OF=1 （オーバーフローあり）
  - 結果の正負が逆になる.
    つまり，SF=1（結果が負）の時，op2 – op1 >= 0.

```
  (+64)
- (-64)
  128        =-128
```

オーバーフロー（OF=1）.
結果は負（SF=1）.
(+64) > (-64).

```
  (-64)
- (+65)
  -129       =127
```

オーバーフロー（OF=1）.
結果は正（SF=0）.
(-64) < (+65).

これ以降は小難しい話

最初は読み飛ばしてＯＫ.

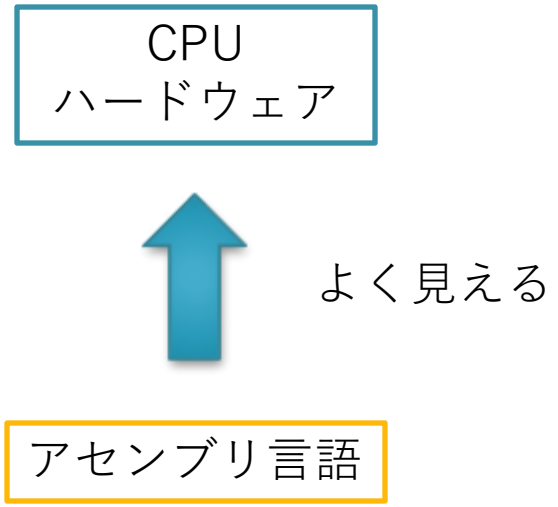
ユーザプロセス

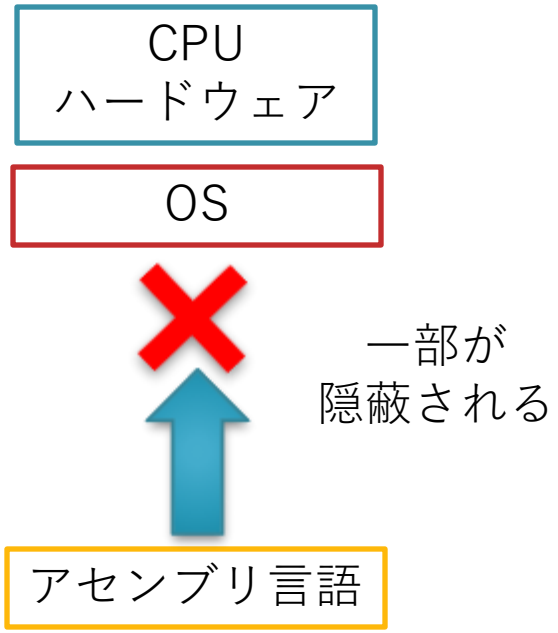
- OSはユーザプロセスを管理し，いろいろ制限.
  - アセンブリプログラミングにも影響あり.

- 制限や影響の例：
  - ユーザプロセスは特権命令や入出力命令を実行できない.
    - システムコールを使って，間接的にOSに実行を依頼するしかない.
    - 制御レジスタなどのシステムレジスタ，メモリ管理ユニット(MMU), キャッシュなども制御できない．（制御するのはOSの役目）
  - 仮想記憶（ページング）により，OSや他プロセス のメモリ領域にアクセスできない.
  - メモリ領域はフラットモデルであり，セグメントレジスタの値を変更できない.
  - CPUは（リアルモードではなく）保護モードで動作し，ユーザプロセスはその動作モードを変更できない.

OSは邪魔！？

- アセンブリ言語の利点はCPUやハードウェアがよく見えること.
- OSはCPUやハードウェアを見せない働きをする.

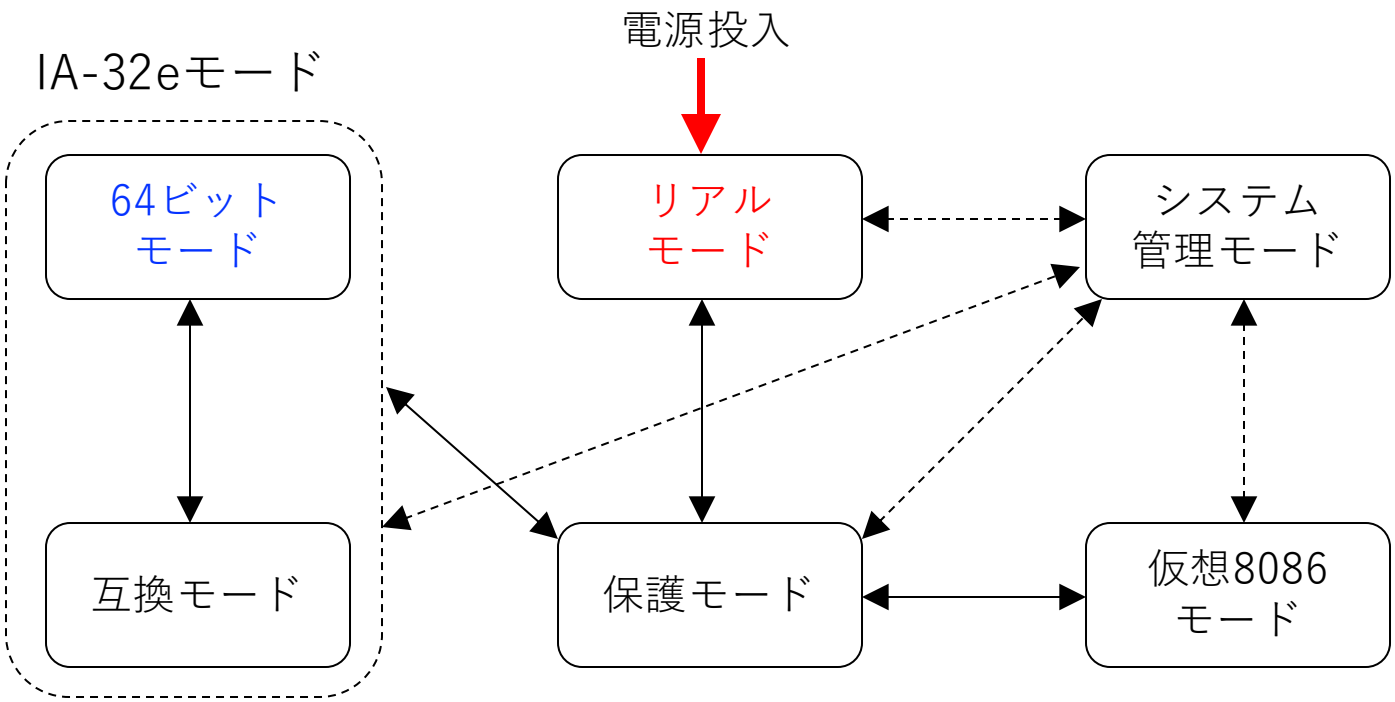




正式には実アドレスモード

リアルモードと64ビットモード

- x86-64は次の6つの動作モードを持つ.
  - 電源投入直後はリアルモード.
  - OS管理下では64ビットモード.



リアルモード

HDDの内容も破壊可能.
つまり，とても危険.

- 保護がないので何でもできる.
  - 特権命令や入出力命令を実行できる.
  - BIOSコールも呼び出せる.
  - ページングはオフ．生の物理メモリにアクセス可能.
- 8086プロセッサ互換のため，機能は貧弱.
  - 物理アドレス空間は 20ビット（1MB）.
  - 最大セグメントサイズは 64KB.
  - デフォルトのアドレス・データサイズは16ビット.
    - 32ビットのサイズでのアクセスは可能.
  - 保護モード・ページングに移行するのは（可能だが）面倒.
    - LILOやGRUBなどのブートローダを使うと比較的簡単.

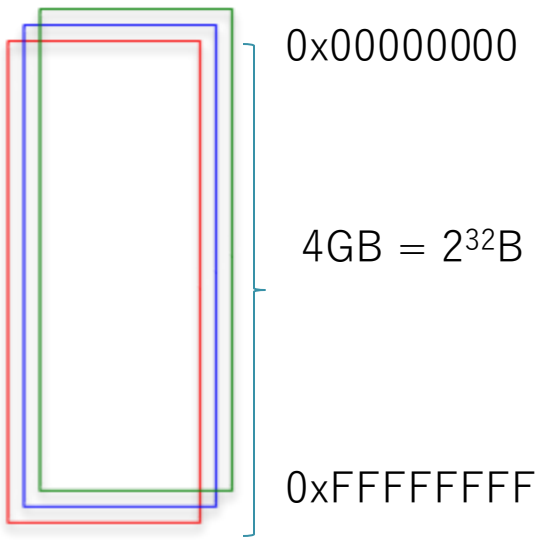
ユーザプロセス

セグメントレジスタの値は
ユーザプロセスからは通常は
変更しない.

- ユーザプロセス
  - 保護モード，仮想記憶，フラットモデル.
- フラットモデル
  - %cs, %ds, %ssがアドレス空間全体を指すメモリ設定.

%cs %ds %ss

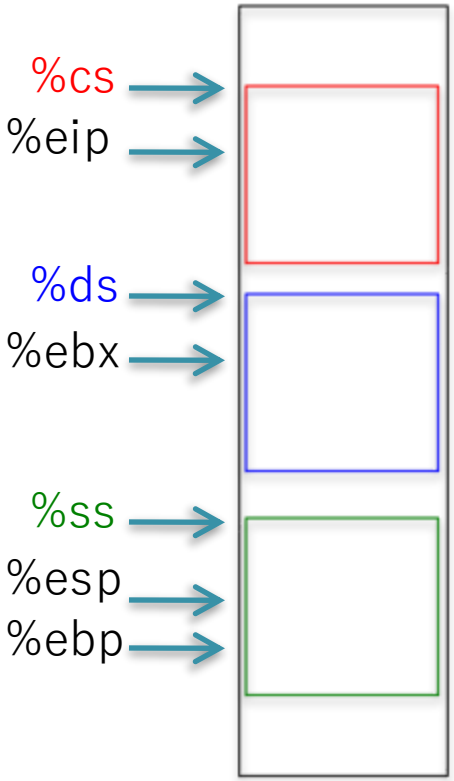
コードセグメント,
データセグメント,
スタックセグメントは
完全に重なっていて,
意識しなくて済む設定.



64ビットモードでは
セグメント機構は無効化

フラットモデルではない例（32ビットモード）

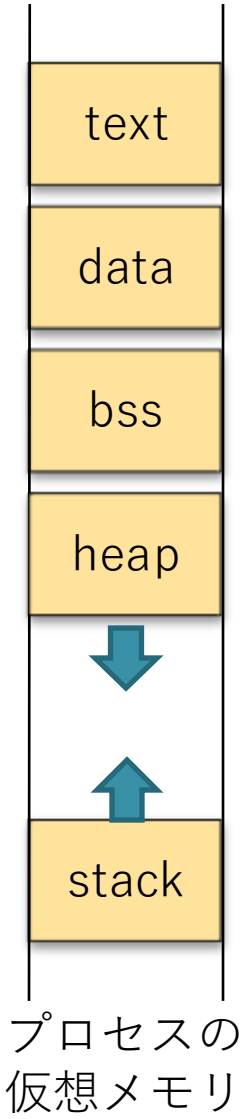
- 各セグメントは重ならない.
- セグメントの大きさはバラバラ.
- %eip, %ebx, %esp, %ebp などは
  セグメントの先頭からのオフセット.
  - %eip=%ebxという可能性あり.
  - 別物が同じポインタ値を持つ可能性あり.
- リアルモードではセグメントサイズ
  が最大64KBしかないので，つらい.
  - 64KBの壁



アドレス空間（再）

- すべてのアドレスにメモリがあるとは限らない.
- メモリがあってもアクセスできるとは限らない.
  - 例：その物理アドレスに対応する物理メモリがない.
  - 例：OSがアクセスを許可してない.
  - 例：その物理アドレスが入出力装置にマップされている.
  - 例：その仮想アドレスが物理メモリにマップされていない.
  - だから，勝手なアドレスでメモリにアクセスしてはいけない.
- 使って良いメモリ領域.
  - テキスト領域，データ領域，BSS領域，ヒープ領域.
  - 要するに「OSからもらったメモリ」だけを使う.

プロセスのメモリ領域（1）



- 便宜上，使用目的ごとにメモリ領域を区別

- テキスト：機械語命令列を配置
- データ：初期化済みデータ（例：大域変数）
- bss：未初期化のデータ
  - テキスト・データ・bssは実行中，サイズ不変
- ヒープ
  - malloc/free で確保・解放するメモリ
  - 実行時に下向きに成長
- スタック
  - 手続き呼び出しのために使用するメモリ
  - 関数コールで上向きに成長

Linux だと –no-pie
sudo sysctl -w kernel.randomize_va_space=0 も必要

プロセスのメモリ領域（２）

```
#include <stdio.h>
#include <stdlib.h>
int v1 = 999; // data
int v2; // bss
int main (void) { // text
    int *v3 = malloc (8); // heap
    int v4; // stack
    printf ("main=%p¥n &v1=%p¥n &v2=%p¥n"
            "  v3=%p¥n &v4=%p¥n",
            main, &v1, &v2, v3, &v4);
}
```

```
% gcc -fno-pie -g foo.c
% ./a.out
main=0x100003f10    テキスト領域
 &v1=0x100004020    データ領域
 &v2=0x100004024    BSS領域
  v3=0x600000004030 ヒープ領域
 &v4=0x7ff7bfeff804 スタック領域
%
```

プロセスのメモリ領域（3）

- macOS では vmmap コマンドでメモリ領域を確認
  - 9666 はプロセス番号

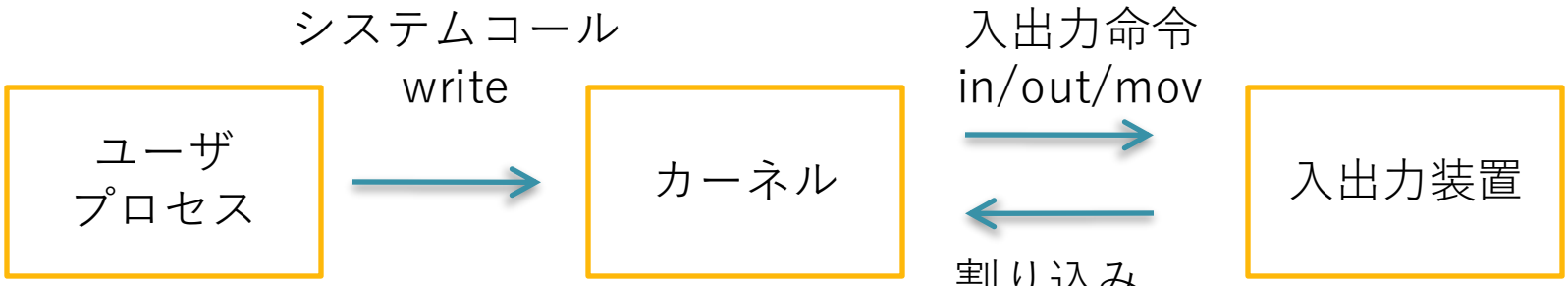
```
% vmmap 9666 （出力を大幅に省略）
REGION TYPE        START - END
__TEXT          100000000-100004000
__DATA          100004000-100008000
MALLOC_NANO    600000000000-600008000000
Stack         7ff7bf700000-7ff7bff00000
```

- Linux では pmap コマンド

```
% pmap 9666 （出力を大幅に省略）
0000000000400000      4K r---- a.out
0000000000401000      4K r-x-- a.out
0000000000402000      4K r---- a.out
0000000000403000      4K r---- a.out
0000000000404000      4K rw--- a.out
00007ffffffde000    132K rw---[ stack ]
```

ユーザ空間とカーネル空間

- ユーザ空間＝ユーザプロセスが動作するアドレス空間.
  - カーネル空間＝カーネル（OS本体）が動作するアドレス空間.
- OSは（CPU機能で）ユーザプロセスをいろいろ制限.
  - ユーザプロセスは，特権命令を実行できない.
  - 入出力装置に直接アクセスできない.
    - OSが提供するシステムコール経由で間接的にアクセスする.
  - カーネルや他のプロセスのメモリ領域にアクセスできない.



printfを呼び出すと
内部でwriteシステム
コールを呼び出す.

BCD

「BCDって何や？」という人に，ご参考.
ただし，BCD用の命令（例：daa）はx86-64では削除.

- 2進化10進数 (binary-coded decimal)
- パック形式BCD (packed BCD)
  - 10進数の1桁を4桁の2進数（4ビット）で表した数.
- アンパック形式BCD (unpacked BCD)
  - 10進数の1桁を8桁の2進数（8ビット）で表した数.
    値は下位4ビットに格納し，上位4ビットは他の目的に使う.
- 例：パック形式BCD
  - 10進数の123をパック形式BCDで表すと 0001 0010 0011.