



# アセンブリ言語

## x86-64 機械語命令（1）

情報工学系  
権藤克彦



命令の種類は多いが、  
よく使う命令はごく一部。

# x86-64機械語命令の特徴

- **CISC** (complex instruction set computer)
  - 1つの命令で複雑な（複数の）処理. cf. MIPSはRISC
  - 直交性が低い（レジスタやアドレッシングモードの）.
  - 汎用レジスタの数が少ない.
  - アドレッシングモードが複雑.
    - cf. RISCのload/storeアーキテクチャ.
- **2アドレスコード** （オペランドは2つが基本）
  - 例：addq %rax, %rbx # %rbx+=%raxの意味
  - cf. MIPSは3アドレスコード.
    - 例：add \$s0,\$s1,\$s2 # \$s0=\$s1+\$s2の意味
- **可変長の命令** : 1~15バイト
  - cf. MIPSは固定長で4バイト. (RISCの多くは固定長)

# この資料で

- 学ぶこと
  - アドレッシングモードと主な非特権命令（64ビットモード）
  - 関数呼び出し（スタックフレームの構造）
  - コンパイル例で見る、機械語命令の使用例
  - Intel形式とAT&T形式の違い
- 学ばないこと
  - 特権命令、浮動小数点命令、入出力命令、拡張命令

- アドレッシングモードと  
主な非特権命令  
(64ビットモード)

# アドレッシングモード

- オペランドの記法を指す。
  - 元々はメモリのアドレスの記法を指す言葉。

ラベル=アドレス

アドレッシングモード	オペランドの値	例
即値	定数の値	addq \$0x100, %rax addq \$_foo, %rax
レジスタ	レジスタ中の値	addq %rbx, %rax
メモリ (直接アドレス指定)	定数アドレスで指定された メモリ中の値	addq 0x100, %rax addq (0x100), %rax addq _foo, %rax
メモリ (間接アドレス指定)	レジスタ中の値 (アドレス) で指定されたメモリ中の値	addq (%rbp), %rax addq -4(%rbp), %rax addq _foo(%rbp), %rax

# メモリ参照（1）

info as では section,  
Intelマニュアルでは segment.  
セグメントレジスタで指定する.  
セグメントレジスタはほぼ不使用

メモリ参照の形式	
AT&T形式	section: disp (base, index, scale)
Intel形式	section: [base + index * scale + disp]

実効アドレス（アクセスするメモリアドレス）は  
section: (base + index \* scale + disp) で計算.

## メモリ参照の例

AT&T形式	Intel形式	説明
-4(%rbp)	[rbp - 4]	dispとbase
foo(%rax,4)	[foo+rax*4]	dispとindexとscale
foo(,1)	[foo]	dispとscale
%gs:foo	gs:foo	sectionとdisp

GNU アセンブラー	C言語
\$foo	foo
foo	*foo

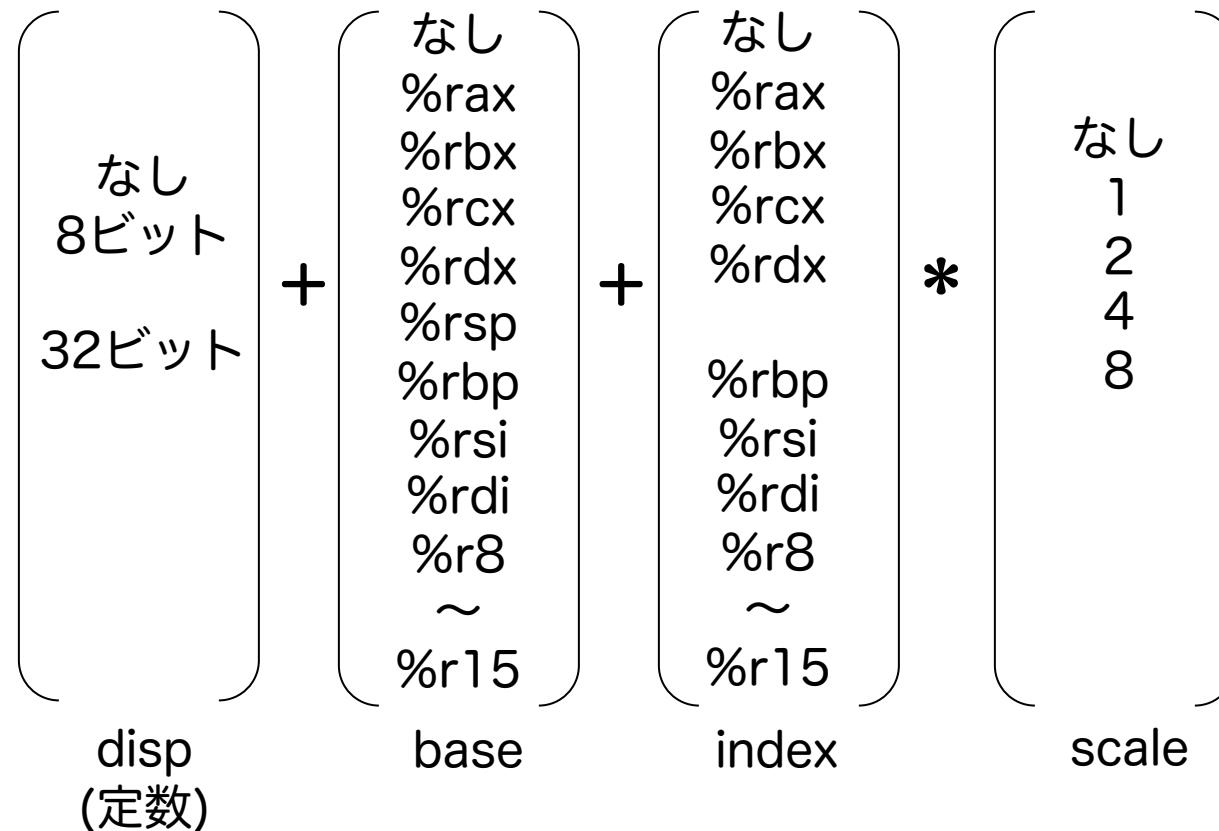
displacement (変位) = 相対アドレスを表す符号あり整数定数値



# メモリ参照 (2)

disp (%rip) という形式も可.  
dispは32ビット。  
アドレス計算時に64ビットに符号拡張

- 指定可能なものの (64ビットモードの場合)



64ビットが無い  
ことに注意！

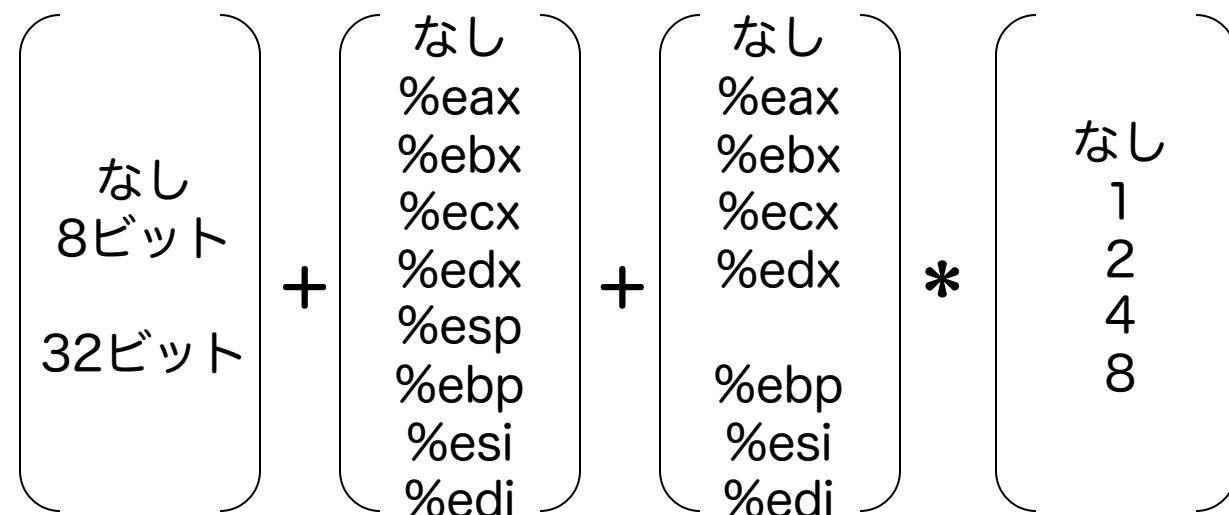
%rspは  
指定不可

なしの場合  
scale=1



# メモリ参照 (3)

- 指定可能なものの (32ビットモードの場合)



disp  
(定数)

base

index

scale

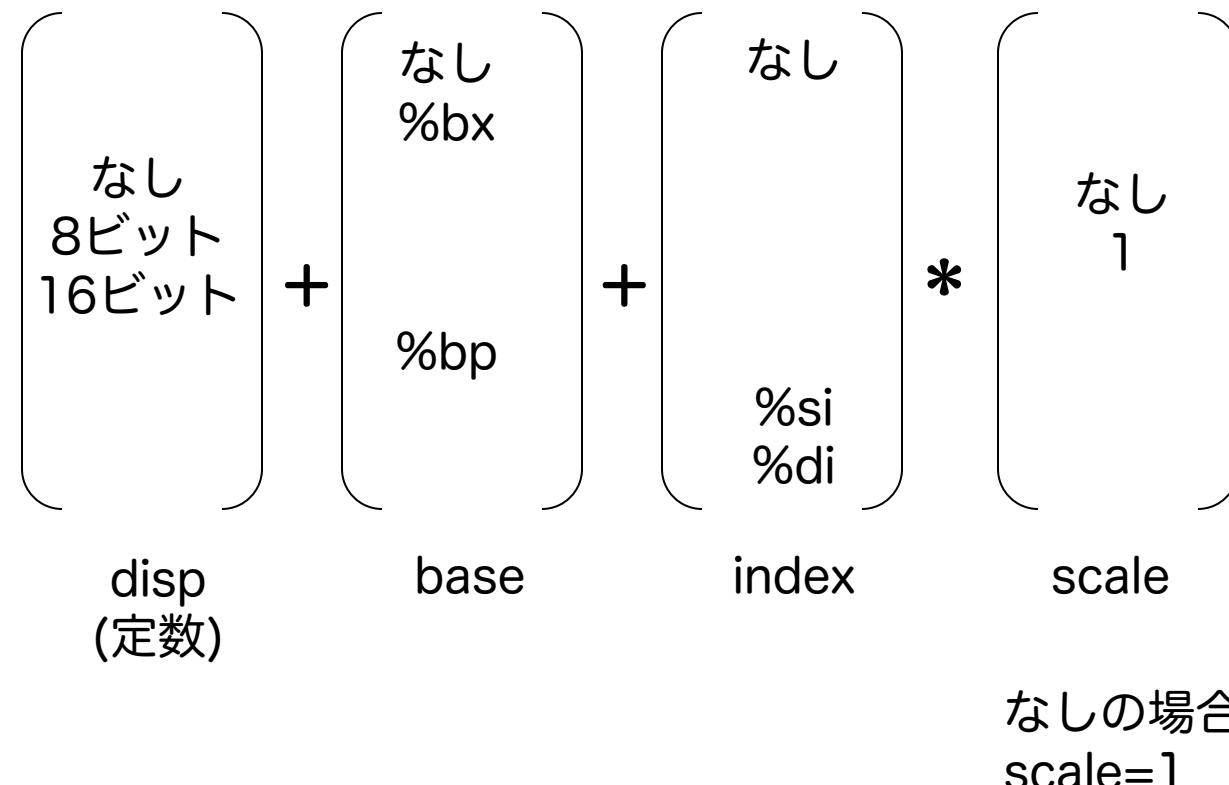
%espは  
指定不可

なしの場合  
scale=1

# メモリ参照 (4)

かなり制限が厳しい

- 指定可能なものの (16ビット保護モード, リアルモード)





このスライドで勝手に

# 命令表で使うオペランド省略記法

省略記法	Intelマニュアルの記法	例	説明
imm	imm8, imm16, imm32, imm64	\$1234 \$_foo	8, 16, 32, 64ビットの 即値 ( <b>immediate value</b> ).
r	r8, r16, r32, r64	%rax	8, 16, 32, 64ビットの 汎用レジスタ.
r/m	r/m8, r/m16, r/m32, r/m64	%rax 1234 (reax) 4(%rbp)	8, 16, 32, 64ビットの 汎用レジスタか メモリ参照.
Sreg	Sreg	%ds	セグメントレジスタ

op1	第1オペランド (AT&T形式で)
op2	第2オペランド (AT&T形式で)

# nop命令

%rflagsに影響無し。

- **nop**は何もしない命令（%ripだけが進む）。
  - 使用例：コード中のパディングとして使う。
    - RISCでは遅延分岐スロットに使う。MIPSでの例：j Label; nop
- nop命令は**1バイト長(0x90)**と多バイトのものがある。

文法	例	説明
nop	nop	何もしない（1バイト長）
nop r/m	nopw 4(%rax)	何もしない（多バイト長）

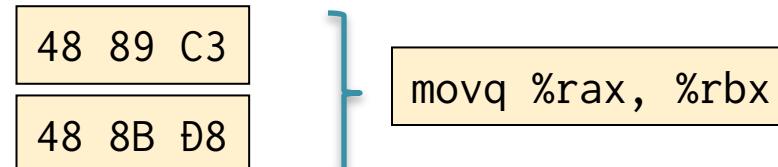
# mov命令：データ転送（コピー）（1）

- **mov**は第1オペランドの値を第2オペランドにコピー。
  - メモリからメモリへの直接コピーはできない。
  - %rflags のフラグは変更しない。

	文法	例	説明
89	<b>mov</b> r, r/m	movq %rax, %rdx movq %rax, 4(%rbp)	%rdx=%rax *(%rbp+4)=%rax
8B	<b>mov</b> r/m, r	movq 4(%rbp), %rax	%rax=*(%rbp+4)
b8	<b>mov</b> imm, r	movq \$999, %rdx	%rdx=999
c7	<b>mov</b> imm, r/m	movq \$999, 4(%rbp)	*(%rbp+4)=999

注意：b8はimm64あり、c7はimm64なし（imm32まで）

注意：異なるオペコードで同じ動作のmov命令あり。



# 確認：movq命令を実行

```
.globl _main
.text
_main:
    movq $999, %rax
    retq
```

foo.s という  
ファイルで保存

出力を大幅に省略

```
% gcc -g foo.s
% lldb ./a.out
(lldb) b main      main関数の最初にブレーク（実行一時停止）設定
(lldb) run         実行開始
a.out`main:
-> 0x100003fb1 <+0>: movq $0x3e7, %rax  movq前で停止
    0x100003fb8: addl %eax, (%rax)  これはゴミ
(lldb) si          1命令だけステップ実行
● Process 9847 stopped
* thread #1, queue = 'com.apple.main-thread', stop
(lldb) p $rax      %raxの値を表示
(unsigned long) $1 = 999    999が入っていた
(lldb) quit
%
```

# 確認：機械語命令 2進数直書きでも同じ

```
.text  
movq %rax, %rbx  
.byte 0x48, 0x89, 0xC3  
.byte 0x48, 0x8B, 0xD8
```

違う機械語命令だけど  
実行効果は同じ

```
% gcc -c foo.s  
objdump -d foo.o  
Disassembly of section __TEXT,__text:  
0: 48 89 c3    movq %rax, %rbx  
3: 48 89 c3    movq %rax, %rbx  
6: 48 8b d8    movq %rax, %rbx
```

- コンパイラは、命令が短い方（または実行速度が速い方）を自動的に選択。



# xchg : オペランドの値を交換

- **xchg**命令は2つのオペランドの値を交換する.
- lock xchgはアトミック→同期機構の実装に使用可.
  - mov命令はアトミック性を保証していない.
    - 「アライメント制約を満たせば保証される」とよく言われるが…
    - ページ境界をまたぐmov命令は明らかに保証されない
  - アトミックな命令は実行途中で割り込まれない.
  - 実はlock つけなくても特別にアトミックになる

文法	例	説明
xchg r, r/m	xchg %eax, %edx xchg %eax, 4(%rbp)	値を交換
xchg r/m, r	xchg 4(%rbp), %eax	

32ビットでは、nop命令は xchg %eax, %eax だった。

64ビットでは、xchg %eax, %eax は %raxの上位32ビットをゼロクリアするので、もはや nopではない

# lea : 実効アドレスをロード

- lea = load effective address
- lea命令は第1オペランドの実効アドレスを第2オペランドに格納.
  - 実効アドレス=実際にアクセスすべきメモリアドレス.
  - lea命令はメモリを読まない. アドレスを計算するだけ.

文法	例	説明
lea m, r	leaq 4(%rbx, %rsi), %rax leaq 4(%rip), %rax	%rax=4+%rbx+%rax %rax=4+%rip

- lea命令を使うと、加算・乗算を高速にできることがある.

leaq 4(%rbx, %rsi, 4), %rax

=

movq \$4, %rax  
addq %rbx, %rax  
shlq \$2, %rsi  
addq %rsi, %rax

論理左シフト

同じ計算だが、leaを使った方が高速.



32ビットのpush/popは不可

%rflagsに影響無し.

# pushとpop：スタック操作（1）

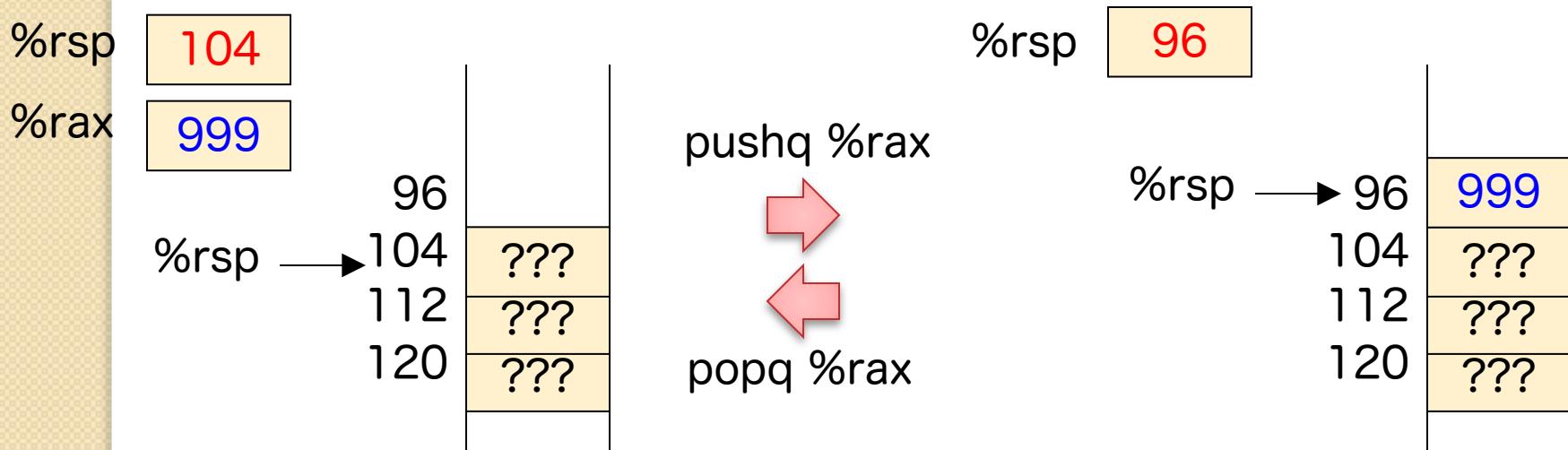
- push命令はスタックポインタ（%rsp, %sp）を減らしてから、スタックトップにオペランド値を格納。
- pop命令はスタックトップの値をオペランドに格納してから、スタックポインタを増やす。

文法	例	説明
push imm	pushq \$999	%rsp -=8; *(%rsp)=999
push r/m16	pushw %ax	%sp -= 2; *(%sp)=%ax
push r/m64	pushq 4(%rbp)	%rsp -= 8; *(%rsp)=*(%rbp+4)

文法	例	説明
pop r/m16	popw %ax	%ax=*(%sp); %sp += 2;
pop r/m64	popq 4(%rbp)	*(%rbp+4)=*(%rsp); %rsp += 8;

# pushとpop：スタック操作（2）

- x86-64のスタックは（通常）低アドレスに成長。



抽象データ型のスタックとは異なり、`movq`命令等を使って、スタックトップ以外にもアクセス可能。

# 算術論理演算

演算の種類	主な命令
算術	add, sub, mul, div, inc, dec, neg
論理	and, or, xor, not,
シフト	sal, sar, shl, shr, rol, ror, rcl, rcr
比較	cmp, test
変換	movs, movz, cbtw, cwtd

- ほぼどれも%rflagsのフラグを変更する.  
次の表記で表す.

O	S	Z	A	P	C
F	F	F	F	F	F
x	?	0	1		

(空白)	変更無し
x	変更あり
?	未定義
0	クリアされる
1	セットされる



# 算術演算（1）

O	S	Z	A	P	C
F	F	F	F	F	F
x	x	x	x	x	x

文法	例	説明
add imm32, r/m	addq \$123, %rax	%rax += 123;
add r, r/m	addq %rax, (%rbx)	*(%rbx) += %rax;
add r/m, r	addq 4(%rbp), %rax	%rax += *(%rbp+4);
adc op1, op2 (addと同じ)	adcq \$123, %rax adcq %rax, (%rbx)	キャリー付きの加算. op2 += op1 + CF;
sub op1, op2 (addと同じ)	subq \$123, %rax subq %rax, (%rbx)	%rax -= 123; *(%rbx) -= %rax;
sbb op1, op2 (addと同じ)	sbbq \$123, %rax sbbq %rax, (%rbx)	ボロー付きの減算. op2 -= op1 + CF;

imm64 は不可。 movabsq で転送して加算する。  
addq \$0x1122334455667788, %rax とは書けない

# 算術演算（2）

- **sub**も符号なし・ありを区別しない。
  - 両方の結果を生成する。
  - addと同様に, **OF, CF, SF**で演算結果をチェックする。

movb \$0xFF, %al  
subb \$0x01, %al

$$\begin{array}{r} 0xFF \\ - \quad - 1 \\ \hline 0x01 \end{array}$$

0xFE  
OF=0  
CF=0  
SF=1

movb \$0x80, %al  
subb \$0x7F, %al

$$\begin{array}{r} 0x80 \\ -0x7F \\ \hline 0x01 \end{array}$$

OF=1  
CF=0  
SF=0

movb \$0x01, %al  
subb \$0x02, %al

$$\begin{array}{r} 0x01 \\ -0x02 \\ \hline 0xFF \end{array}$$

OF=0  
CF=1  
SF=1



# 算術演算（3）

%rdx:%rax

乗算結果の上位64ビットが%rdxに入り  
下位64ビットが%raxに入る

文法	例	説明
<code>mul r/m</code>	<code>mulq %rbx</code>	符号なしの乗算. %rdx:%rax=%rax*%rbx;
<code>imul r/m</code> <code>imul r/m, r</code> <code>imul imm32, r/m, r</code> <code>imul imm32, r</code>	<code>imulq %rbx</code> <code>imulq %rbx, %rax</code> <code>imulq \$4, %rbx, %rax</code> <code>imulq \$4, %rax</code>	符号ありの乗算. %rdx:%rax=%rax*%rbx %rax *= %rbx; %rax = %rbx * 4; %rax *= 4;
<code>div r/m</code>	<code>divq %rbx</code>	符号なしの除算. %rax = (%rdx:%rax) / %rbx; %rdx = (%rdx:%rax) % %rbx;
<code>idiv r/m</code>	<code>idivq %rbx</code>	符号ありの除算. 同上

mul	O	S	Z	A	P	C
	F	F	F	F	F	F
imul	x	?	?	?	?	x

div	O	S	Z	A	P	C
	F	F	F	F	F	F
idiv	?	?	?	?	?	?



# idiv 使用例

```
.globl _main
.text
_main:
    movl $10, %eax
    movl $0, %edx
    movl $3, %ecx
    idiv %ecx
    retq
```

$10/3 = 3$  が %eax に入る

%edxの値設定を忘れないように  
cltd命令を使うと%edxの設定が楽  
%rax なら, cqto

```
.globl _main
.text
_main:
    movl $-10, %eax
    movl $0xFFFFFFFF, %edx
    movl $3, %ecx
    idiv %ecx
    retq
```

$-10/3 = -3$  が %eax に入る

デバッガで確認せよ

商が大きすぎると浮動小数点エラーになる



# 算術演算 (4)

文法	例	説明
inc r/m	incq %rax	1だけインクリメント. %rax++;
dec r/m	decq %rax	1だけデクリメント. %rax--;
neg r/m	neg %rax	2の補数による符号逆転. %rax = -%rax;

CFが変化しない  
点がポイント.

O S Z A P C  
F F F F F F

X X X X X

O S Z A P C  
F F F F F F

X X X X X X



# ビット論理演算

文法	例	説明
not r/m	notq %rax	%rax = $\sim$ %rax;
and imm32, r/m	andq \$0xFFFF, %rax	%rax $\&=$ 0xFFFF;
and r, r/m	andq %rbx, 4(%rbp)	$(%rbp+4) \&= %rax;$
and r/m, r	andq %rax, %rbx	%rbx $\&= %rax;$
or op1, op2 (andと同じ)	orq %rax, %rbx	%rbx $ = %rax;$
xor op1, op2 (andと同じ)	xorq %rax, %rbx	%rbx $\wedge= %rax;$

not	O	S	Z	A	P	C
	F	F	F	F	F	F

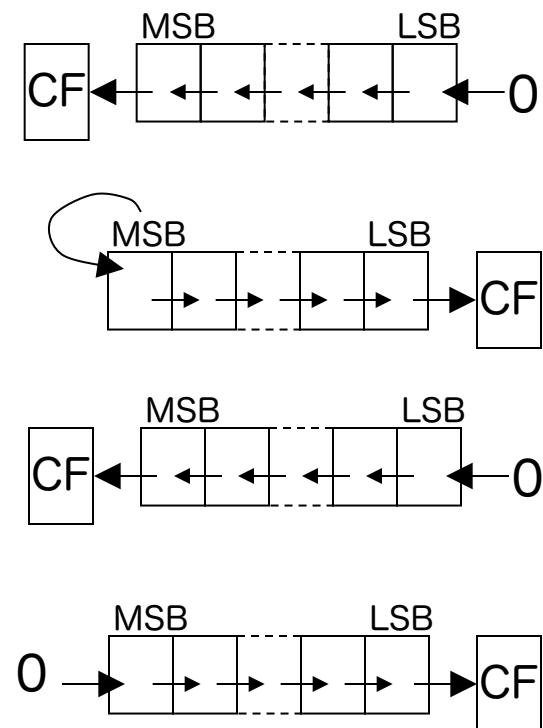
and	O	S	Z	A	P	C
or	F	F	F	F	F	F
xor	0	x	x	?	x	0



# シフト演算

O	S	Z	A	P	C
F	F	F	F	F	F
x	x	x	?	x	x

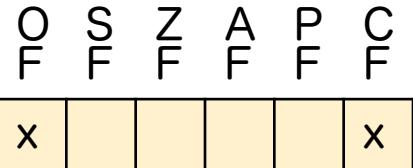
文法	例	説明
sal r/m	salq %rax	算術左シフト
sal imm8, r/m	salq \$2, %rax	imm8回シフト
sal %cl, r/m	salq %cl, %rax	%cl回シフト
sar op1 [, op2] (salと同じ)	sarq \$2, %rax sarq %cl, %rax	算術右シフト
shl op1 [, op2] (salと同じ)	shlq \$2, %rax shlq %cl, %rax	論理左シフト
shr op1 [, op2] (salと同じ)	shrq \$2, %rax shrq %cl, %rax	論理右シフト



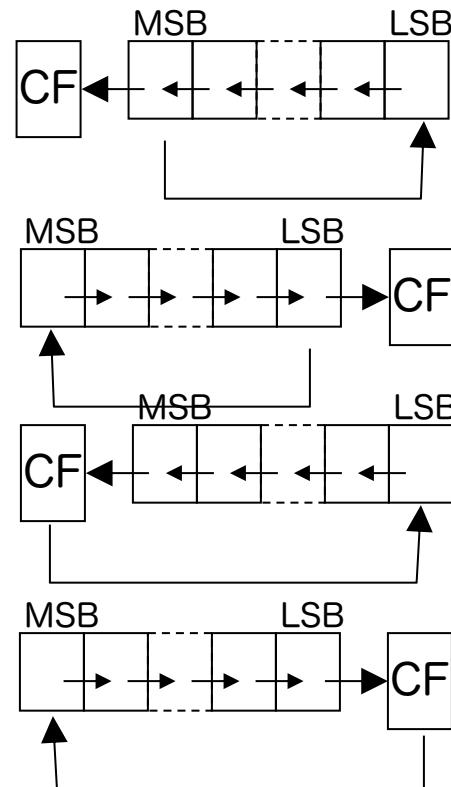
salとshl は同じ動作



# ローテート命令



文法	例	説明
<b>rol</b> r/m	rolq %rax	左にローテート.
<b>rol</b> imm8, r/m	rolq \$2, %rax	
<b>rol</b> %cl, r/m	rolq %cl, %rax	
<b>ror</b> op1 [, op2] ( <b>rol</b> と同じ)	rorq \$2, %rax rorq %cl, %rax	右にローテート.
<b>rcl</b> op1 [, op2] ( <b>rol</b> と同じ)	rclq \$2, %rax rclq %cl, %rax	CFを含めて左に ローテート.
<b>rcr</b> op1 [, op2] ( <b>rol</b> と同じ)	rcrq \$2, %rax rcrq %cl, %rax	CFを含めて右に ローテート.



rotate=回転する。循環する。

# 比較命令

フラグ計算だけを行う命令  
通常、次に条件付きジャンプ命令が続く

文法	例	説明
cmp imm32, r/m	cmpq \$123, %rbx	
cmp r, r/m	cmpq %rcx, %rbx	
cmp r/m, r	cmpq 4(%rbp), %rbx	
test imm32, r/m	testq \$123, %rbx	
test r, r/m	testq %rcx, %rbx	
test r/m, r	testq 4(%rbp), %rcx	

O	S	Z	A	P	C
F	F	F	F	F	F

x	x	x	x	x	x
---	---	---	---	---	---

O	S	Z	A	P	C
F	F	F	F	F	F

0	x	x	?	x	0
---	---	---	---	---	---

- cmp命令はsub命令と同じ。  
ただし、cmpはオペランドを変更せず、%rflagsだけを変更。
- test命令はand命令と同じ。  
ただし、testはオペランドを変更せず、%rflagsだけを変更。

cmp ≈ sub  
test ≈ and

# ジャンプ（1）：文法

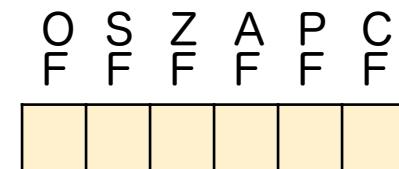
- jmp命令は無条件ジャンプを実行。

rel32は32ビットの  
符号付き定数整数値。  
相対アドレスを表す。

文法	例	説明
jmp rel8	jmp 0x10 jmp _foo	short, 相対, 直接ジャンプ (\$はつけない)
jmp rel32	jmp 0x1000 jmp _foo	near, 相対, 直接ジャンプ (\$はつけない)
jmp r/m	jmp *%rax	near, 絶対, 間接ジャンプ
ljmp m16:64	ljmp *4(%rbp)	far, 絶対, 間接ジャンプ

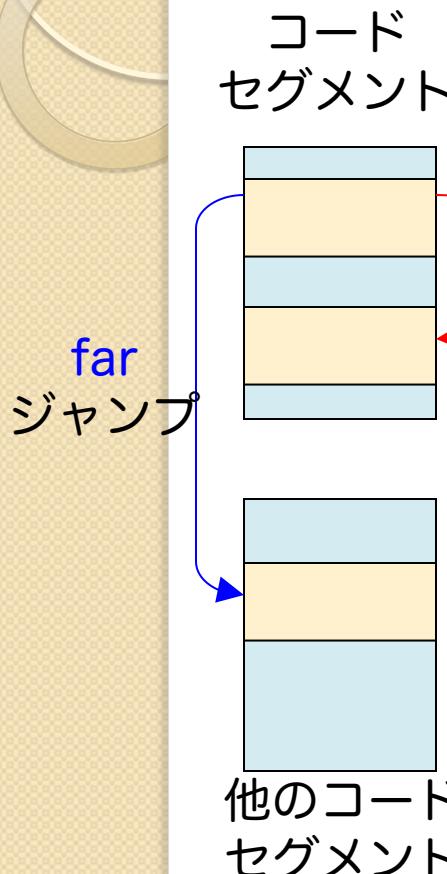
相対ジャンプは32ビットのみで、64ビットは無い。  
32ビットの範囲を超えるジャンプは絶対間接ジャンプを使う。

保護モードのfarジャンプ（タスクスイッチにも使用）  
はここでは簡単のため説明していない。

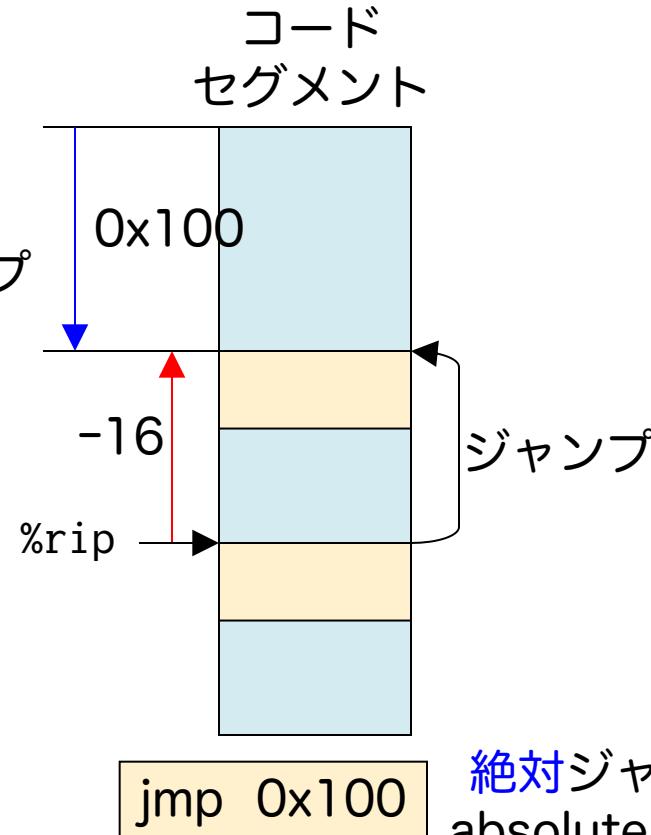




# ジャンプ (2) : 用語



-128~127の範囲の  
nearジャンプを特に  
**shortジャンプ**と呼ぶ。



x86-64では、  
near直接ジャンプは  
相対ジャンプのみ。

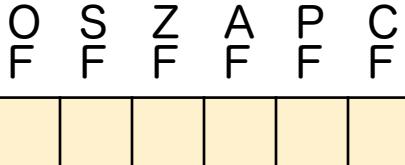
# ジャンプ（3）：文法のおかしな点

- ドル記号(\$)とアスタリスク記号(\*)
  - 記法が一貫していない。

jmp _foo	\$はつけない。
jmp *%eax	*をつける。

- ジャンプ命令には命令の接尾語（b, w, l, q）をつけてない。

jmp _foo	OK
jmp <b>b</b> _foo	NG
jmp <b>l</b> _foo	NG



# 条件付きジャンプ（1）

条件付き  
ジャンプ命令の  
総称

- **jcc** は%rflags (CF, OF, PF, SF, ZF) か%rcxをチェックして、条件が成り立てば、ジャンプする。

文法	例	説明
jcc rel	jmp _foo	条件が成り立てばジャンプ

- **jcc** は **cmp** と組み合わせてよく使う。

cmpq \$0, 8(%rbp)	jump if 8(%rbp) > 0
jg L2	

- **jcc** は **far** ジャンプをサポートしていない。
  - farジャンプをするには、次のようにjccとjmpを組み合わせる。

NG **jz \*4(%rbp)**



**jnz 1f**  
**ljmp \*4(%rbp)**  
**1:**

OK



使用パターン

```
cmp op1, op2
jcc rel
```

# 条件付きジャンプ（2）

- 条件付きジャンプ命令（符号あり整数用）

命令	条件	フラグ	説明
jg jnle	op2 > op1 !(op2 <= op1)	ZF==0 && SF==OF	greater not less nor equal
jge jnl	op2 >= op1 !(op2 < op1)	SF==OF	greater or equal not less
jle jng	op2 <= op1 !(op2 > op1)	ZF==1    SF!=OF	less or equal not greater
jl jnge	op2 < op1 !(op2 >= op1)	SF!=OF	less not greater nor equal

- less と greater は符号あり整数に使い,  
above と below は符号なし整数を使う.

cmp op1, op2  
jcc rel

# 条件付きジャンプ（3）

- 条件付きジャンプ命令（符号なし整数用）

命令	条件	フラグ	説明
ja	$op2 > op1$	$CF == 0 \ \&\& ZF == 0$	above
jnbe	$!(op2 \leq op1)$		not below nor equal
jae	$op2 \geq op1$	$CF == 0$	above or equal
jnb	$!(op2 < op1)$		not below
jbe	$op2 \leq op1$	$CF == 1 \    ZF == 1$	below or equal
jna	$!(op2 > op1)$		not above
jb	$op2 < op1$	$CF == 1$	below
jnae	$!(op2 \geq op1)$		not above nor equal

- less と greater は符号あり整数に使い,  
above と below は符号なし整数を使う.

# 条件付きジャンプ（4）

- 条件付きジャンプ（カウンタ用）

文法	条件	説明
<code>jcxz rel8</code>	<code>%cx == 0</code>	<code>%cx</code> が0ならジャンプ
<code>jecxz rel8</code>	<code>%ecx == 0</code>	<code>%ecx</code> が0ならジャンプ

- jcxz と jecxz はshortジャンプだけサポート.
- つまり、ジャンプ範囲は-128～127に限定. 使いにくい.
  - 同じ理由で loop命令も使いにくい. 警告が出ない場合も！

NG

```
jecxz _foo
.space 256
_foo:
```

```
% gcc -c foo.s
foo.s:unknown:Fixup of 256 too large
for field width of 1
%
```

# 条件付きジャンプ（5）

- 条件付きジャンプ（フラグ用）

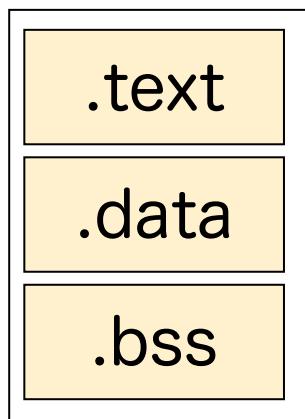
命令	フラグ	説明
jc	CF==1	carry
jnc	CF==0	not carry
jo	OF==1	overflow
jno	OF==0	not overflow
js	SF==1	sign
jns	SF==0	not sign

命令	フラグ	説明
je	ZF==1	equal
jne	ZF==0	not equal
jz	ZF==1	zero
jnz	ZF==0	not zero
jpe	PF==1	parity even
jpo	PF==0	parity odd
jp	PF==1	parity
jnp	PF==0	not parity



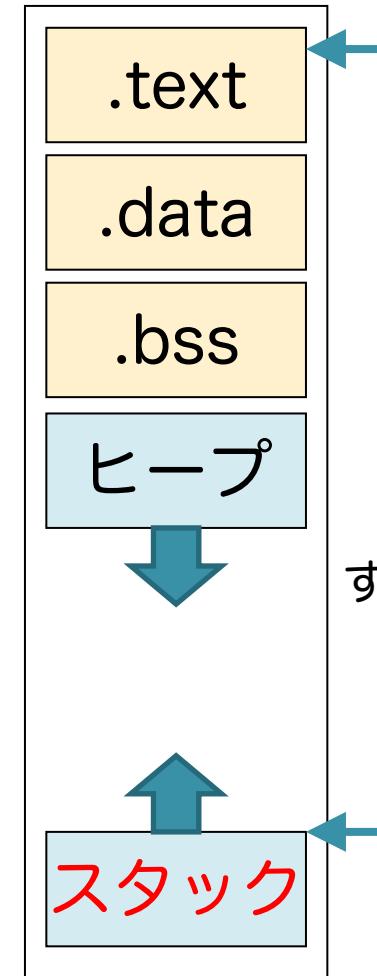
## ◦ 関数呼び出し (スタックフレームの構造)

# プロセスのメモリレイアウト



実行可能ファイル  
a.out

機械語命令列  
静的変数  
(初期化済み)  
静的変数  
(未初期化)



プログラム  
カウンタ %rip  
malloc  
すると成長  
スタック  
ポインタ%rsp

ユーザプロセスの  
メモリ



near  
の場合

O	S	Z	A	P	C
F	F	F	F	F	F

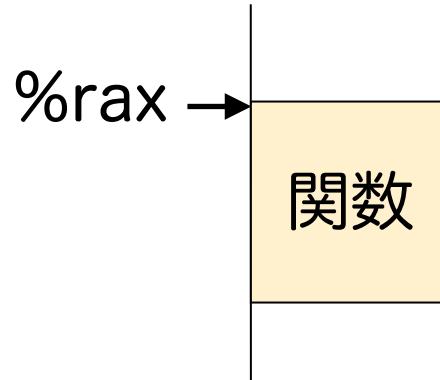
# call, ret, enter, leave (1)

- `call _foo`は次の2動作を実行.
    1. `pushq %rip` (スタック上に%ripの値を退避)
    2. `movq $_foo, %rip` (\_fooにジャンプ)
  - `ret` は次の動作を実行.
    1. `popq %rip`
- call命令の  
 次の命令を指す

文法	例	説明
<code>call rel32</code>	<code>callq _foo</code>	near・相対・直接の関数呼び出し
<code>call r/m64</code>	<code>callq *%rax</code>	near・絶対・間接の関数呼び出し
<code>call m16:16/32</code>	<code>lcallq *4(%rbp)</code>	far・絶対・間接の関数呼び出し
<code>ret</code>	<code>retq</code>	リターン (near用)
	<code>lretq</code>	リターン (far用)
<code>ret imm16</code>	<code>retq \$10</code>	リターン, %rsp += imm16

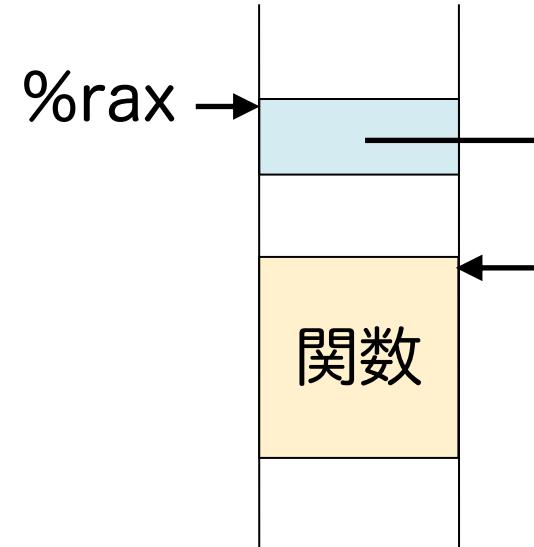
# call命令による間接呼び出し（1）

- `call *%rax`



関数のアドレスを  
レジスタが保持

- `call *(%rax)`



関数のアドレスを  
メモリが保持

# call命令による間接呼び出し（2）

```
int add5 (int n)
{
    return n + 5;
}

int main (void)
{
    int (*fp)(int n);
    fp = add5;
    return fp (10);
}
```

変数fpの型は関数ポインタ。  
その関数の引数はint型であり,  
返り値の型もint型。

```
.text
.globl _main
.p2align 4, 0x90
_main:
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $10, %edi
leaq _add5(%rip), %rax
movl $0, -4(%rbp)
movq %rax, -16(%rbp)
callq *-16(%rbp)
addq $16, %rsp
popq %rbp
retq
```

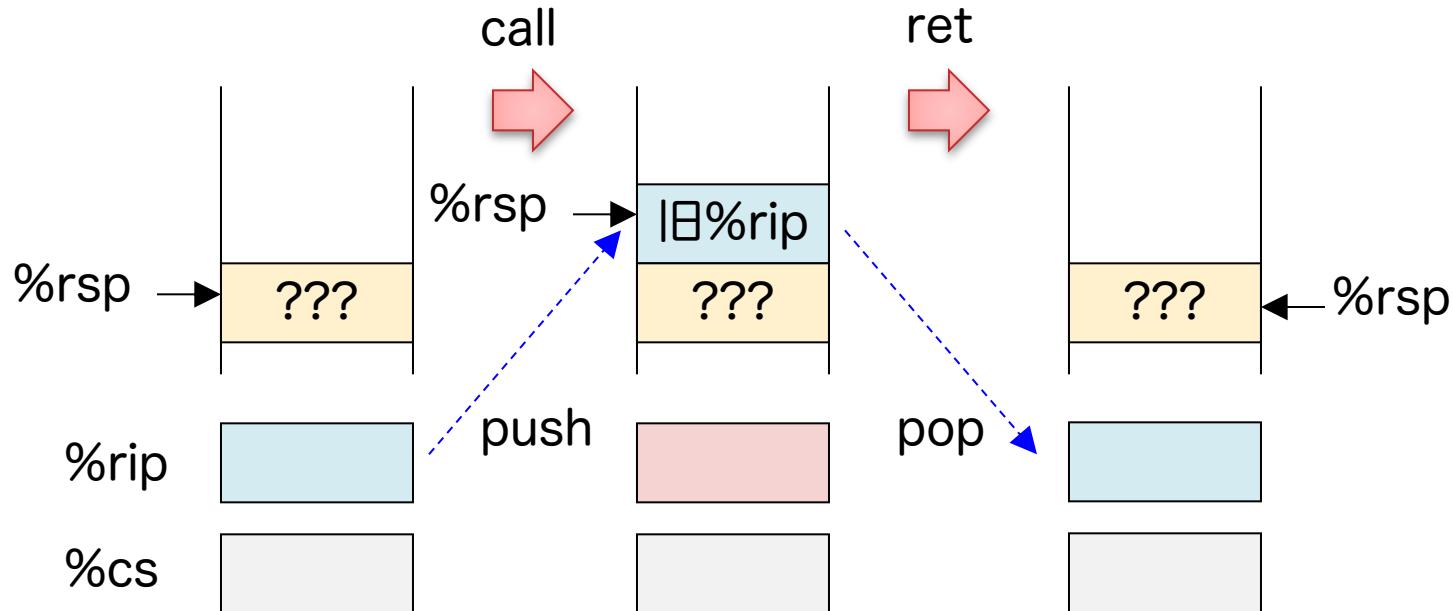
関数ポインタを使うと, call命令は  
間接呼び出しになる。



# call, ret, enter, leave (2)

farの場合は説明しない

- call/ret のスタックの使い方 (nearの場合)



- スタックに積んだ%ripの値 (旧%rip) を  
**戻り番地** (return address)とも呼ぶ。
- スタックを使うことで、呼び出した順序と逆順にリターンできる。
- nearの場合は、%csの値は変化しない。

# call, ret, enter, leave (4)

O	S	Z	A	P	C
F	F	F	F	F	F

- `enter`はスタックフレームを作る.
  - *imm8*はネストレベル. *imm16*はスタックフレームのサイズ.
  - GCCは（遅いので）`enter`命令を使わない.
- `leave`はスタックフレームを解放する.

文法	例	説明
<code>enter imm16, imm8</code>	<code>enter \$32, \$0</code>	<code>pushq %rbp;</code> <code>movq %rsp, %rbp;</code> <code>subq \$32, %rsp</code>
<code>leave</code>	<code>leave</code>	<code>%rsp=%rbp; popq %rbp</code>

C言語だと*imm8*は常にゼロ（入子関数が無いから）



# enterは遅い

```
#include <stdio.h>
#include <stdint.h>
#define MAX (10000)
uint64_t rdtsc (void) {
    uint64_t hi, lo;
    asm volatile ("rdtsc": "=a"(lo), "=d"(hi));
    return ((hi << 32) | lo);
}
int main (void) {
    uint64_t t1, t2, t3; int i;
    t1 = rdtsc ();
    for (i = 0; i < MAX; i++) {
        asm volatile (
            "enter $32, $0; leave");
    }
    t2 = rdtsc ();
    for (i = 0; i < MAX; i++) {
        asm volatile (
            "pushq %rbp; movq %rsp, %rbp;" "subq $32, %rsp; leave");
    }
    t3 = rdtsc ();
    printf ("%llu\n%llu\n", t2-t1, t3-t2);
}
```

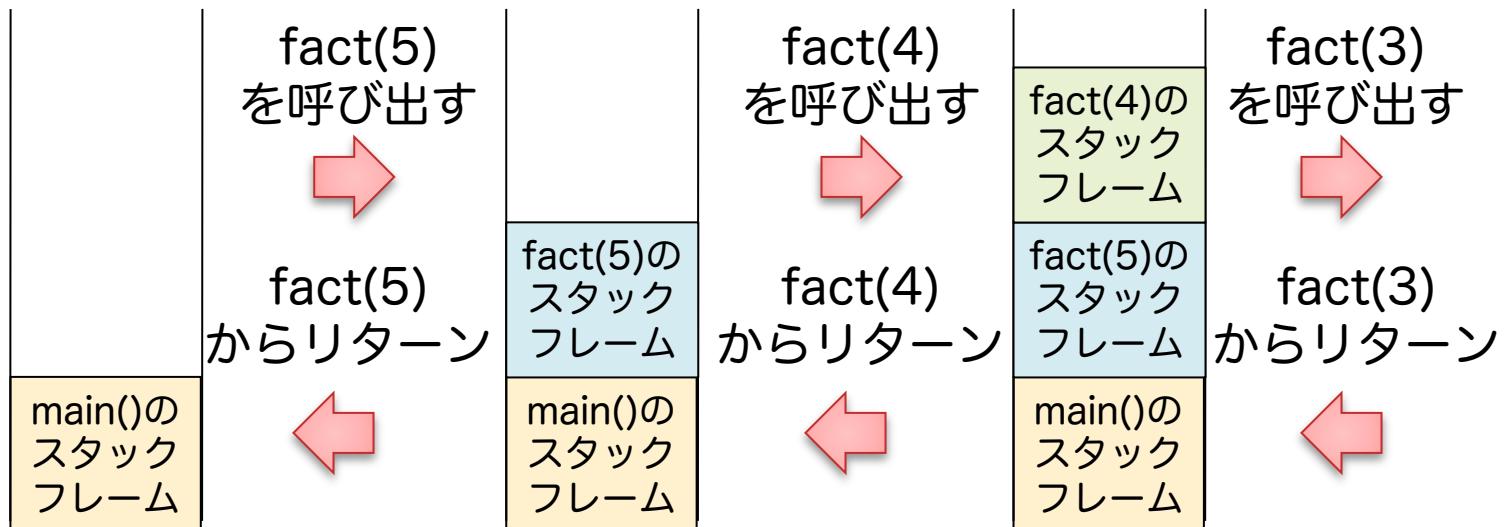
rdtscは64ビットのタイムスタンプカウンタを読む命令。  
CPU起動時からの全サイクル数をカウント。Pentium以降。

```
% gcc -O0 enter.c
% time ./a.out
118582
52842
0.000u 0.000s 0:00.00
```

約2倍遅い  
(以前は約15倍だった)

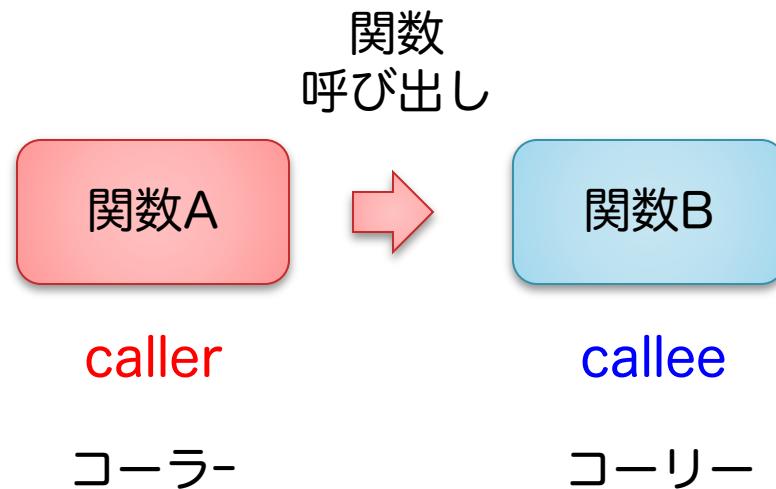
# スタックフレーム (stack frame) (1)

- 関数呼び出し1回分のデータ。スタック上に配置。
  - 局所変数, 引数, 返り値, 戻り番地, 退避したレジスタの値などを含む。
- 関数を呼び出すとスタックフレームをスタックに積み, リターンするとスタックから取り除く。



# caller と callee

- 関数Aが関数Bを呼び出すとき、
  - 関数Aを **caller** (呼び出す側) ,
  - 関数Bを **callee** (呼び出される側) , という.



# レジスタの退避と回復

- 関数呼び出しではスタックへのレジスタ退避が必要。
  - レジスタの個数は有限でごく少ないから。
- **caller**側で退避・**callee**側で退避の2つの方法あり。

caller側で退避

```
pushq %reg  
call _foo  
popq %reg
```

%regを破壊

callee側で退避

```
call _foo
```

```
pushq %reg  
%regを破壊  
popq %reg
```



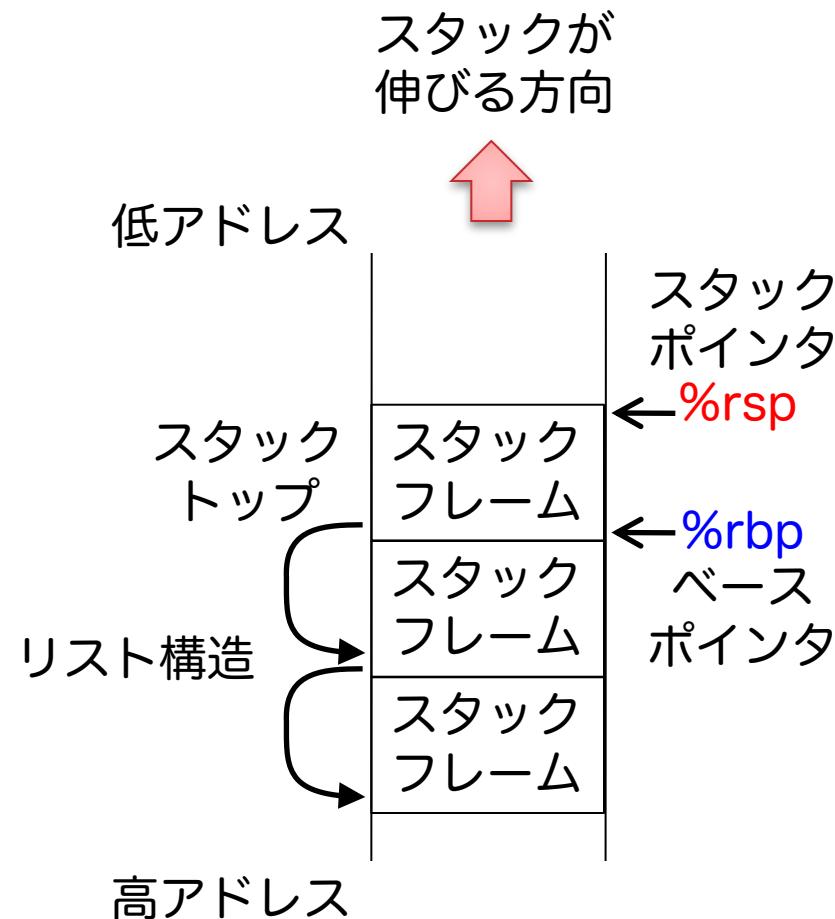
# caller-saveレジスタと callee-saveレジスタ

- caller-saveレジスタ = caller側が退避するレジスタ。
  - 通常, x86-64では %rax, %rcx, %rdx, %rdi, %rsi, %r8 ~ %r11
- callee-save レジスタ = callee側が退避するレジスタ。
  - 通常, x86-64では %rbp, %rsp, %rbx, %r12 ~ %r15

ABIが定める事項

# スタックフレーム (2)

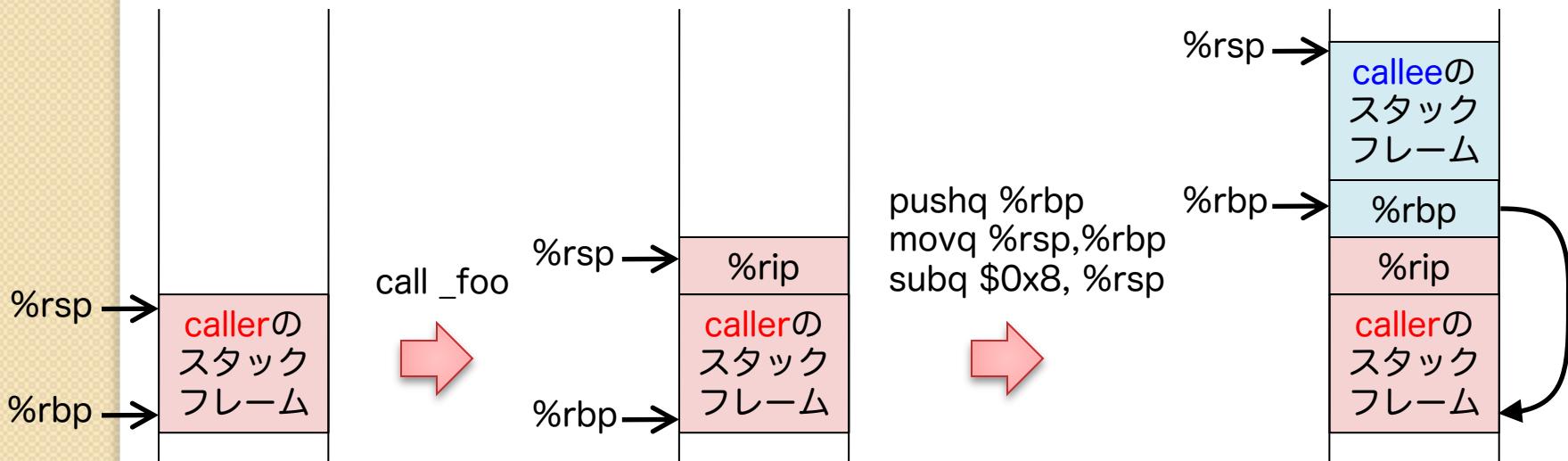
- **%rsp**と**%rbp**
  - x86-64では、**%rsp**と**%rbp**が、  
スタックトップのスタック  
フレームの両端を指す。  
 • そうなるように、関数呼び出し・  
リターン時に調整する。
- 連結リスト (linked list)
  - 各スタックフレームは  
**連結リスト**として接続。



注：-fomit-frame-pointer (Linuxでは -fno-frame-pointer) オプションを使うと、%rbp はベースポインタではなく汎用レジスタとして使われる。ライブラリ関数であるある。

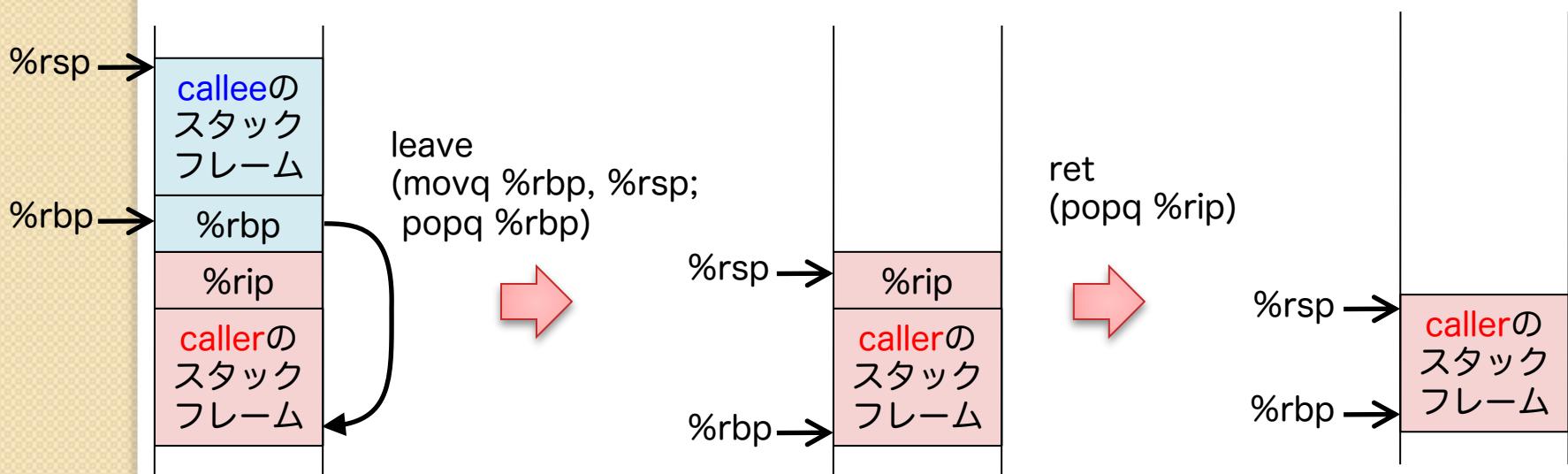
# スタックフレーム（3）

- ・ スタックフレームの作成



# スタックフレーム (4)

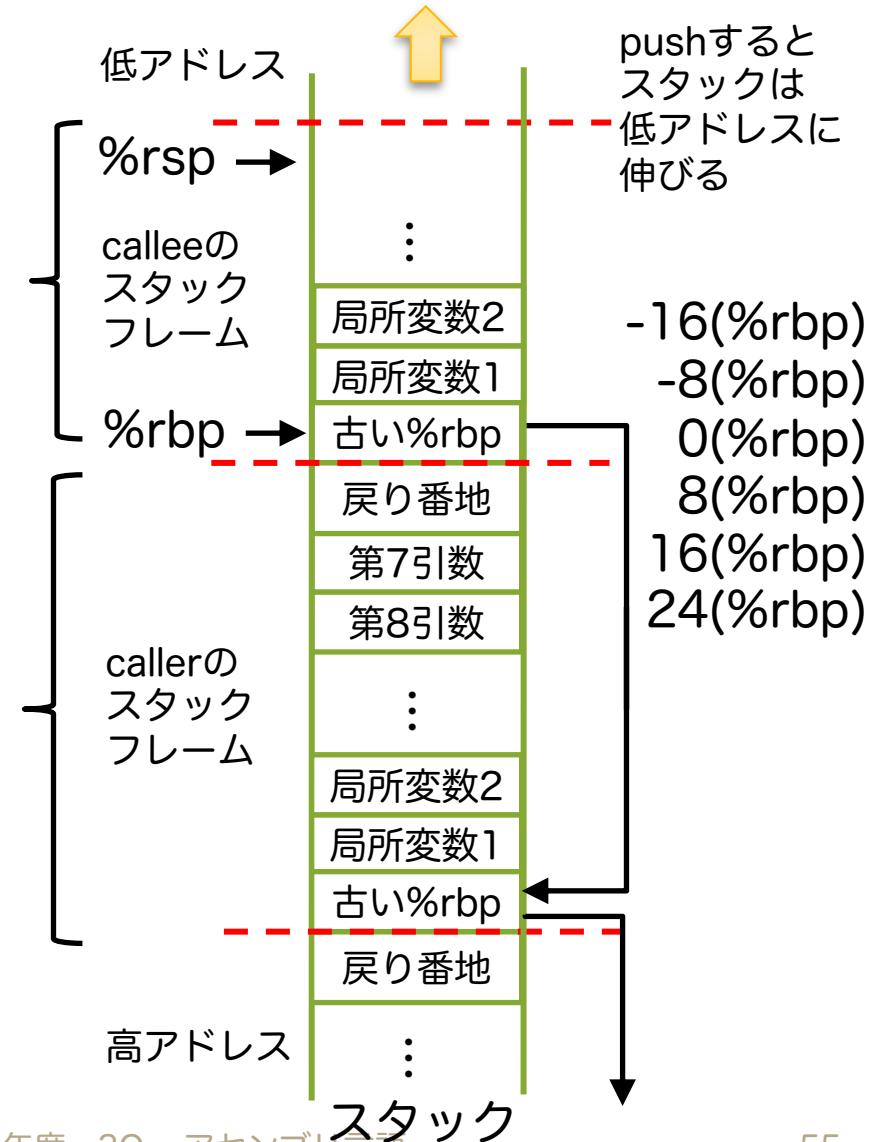
- ・ スタックフレームの破棄





# スタックレイアウト (stack layout)

- ABIや状況により異なる
  - 例：スタックにレジスタを退避
- 引数は逆順にスタックに積む。
  - 必要ならパディングを入れる。
- 自動変数と第7引数以降は %rbpを使ってアクセス。
  - 例：8(%rbp), -8(%rbp)
- レジスタの役割
  - %rsp: スタックポインタ
  - %rbp: ベースポインタ
  - %rax: 返り値 (スタック併用も)



# 関数呼び出し規約 (calling convention)

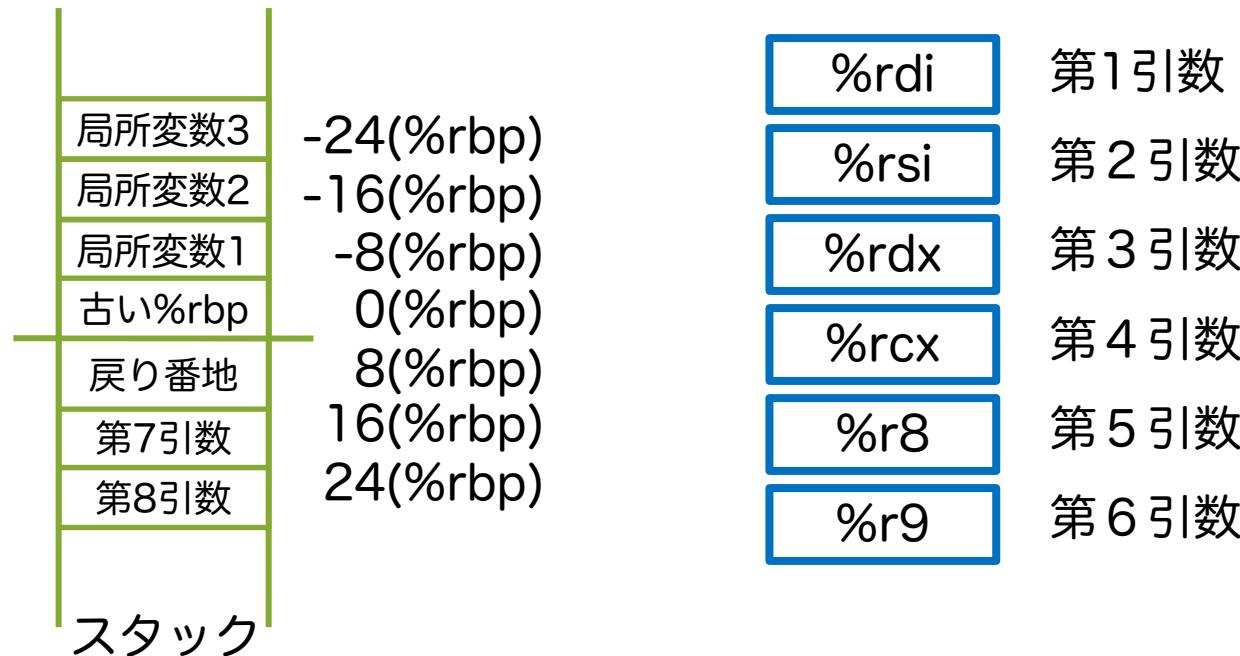
- ABIが定める、 callerとcallee間のお約束。
  - 引数の渡し方,
  - スタックフレームのレイアウト,
  - レジスタの役割,
  - caller-saveレジスタとcallee-saveレジスタ, など.
- プラットフォーム毎に異なる。
  - 間違うとプログラムは正しく動作しない（クラッシュする）。
  - GCCでは`_attribute_`構文で関数呼び出し規約を指定可能。

```
void foo1 () __attribute__ ((stdcall));  
void foo2 () __attribute__ ((cdecl));  
void foo3 () __attribute__ ((fastcall));
```

cdeclがデフォルト



# 引数の渡し方 (AMD64 ABI)





# add5.s 早わかり

add5.c

```
long add5 (long n){  
    return n + 5; }
```

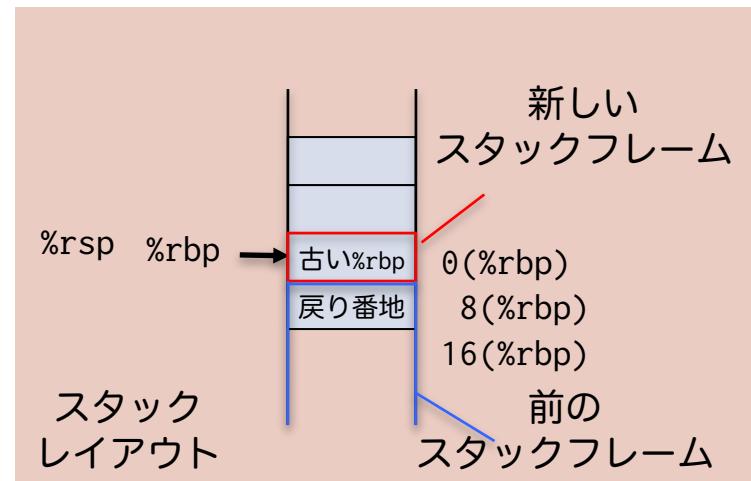
add5.s

```
.text  
.globl _add5  
_add5:  
    pushq %rbp  
    movq %rsp, %rbp  
    addq $5, %rdi  
    movq %rdi, %rax  

```

以降を .text セクションに出力.  
このラベルをグローバル（大域）にせよ.  
ラベル（アドレスは自動的に計算）.  
%rbp の値をスタック上に退避. } [スタックフレームの作成]  
%rbp = %rsp  
%rdi += 5 %rdi は第1引数n  
%rax = %rdi  
popl %rip (リターン) [スタックフレームを破棄.]

- ドット記号 (.) で始まるのはアセンブリ命令.
- ドル記号 (\$) は定数 (即値).
- パーセント記号 (%) はレジスタ.
- コロン記号 (:) で終わるのはラベル.
- 8(%rbp) は C 言語では \*(%rbp+8) の意味.



prologue=序幕, 前口上  
epilogue=終幕, 納め口上

# 典型的な関数プロローグと関数エピローグ

## 関数プロローグ

```
pushq    %rbp  
movq    %rsp, %rbp  
pushq    %rbx  
pushq    %r12  
pushq    %r13  
subq    $28, %rsp
```

新しいスタックフレームを作成

callee-saveレジスタを  
スタック上に退避

必要な  
場合だけ

スタックフレーム上に領域を確保  
(自動変数や引数のための領域など)

必要なサイズは  
関数ごとに異なる

## 関数エピローグ

```
addq    $28, %rsp  
popq    %r13  
popq    %r12  
popq    %rbx  
leave  
retq
```

自動変数や引数などのための領域を解放

スタック上に退避していた  
callee-saveレジスタを回復

必要な  
場合だけ

スタックフレームを破棄

leave=mvq %rbp, %rsp; popq %rbp



今の macOS環境では実行できない

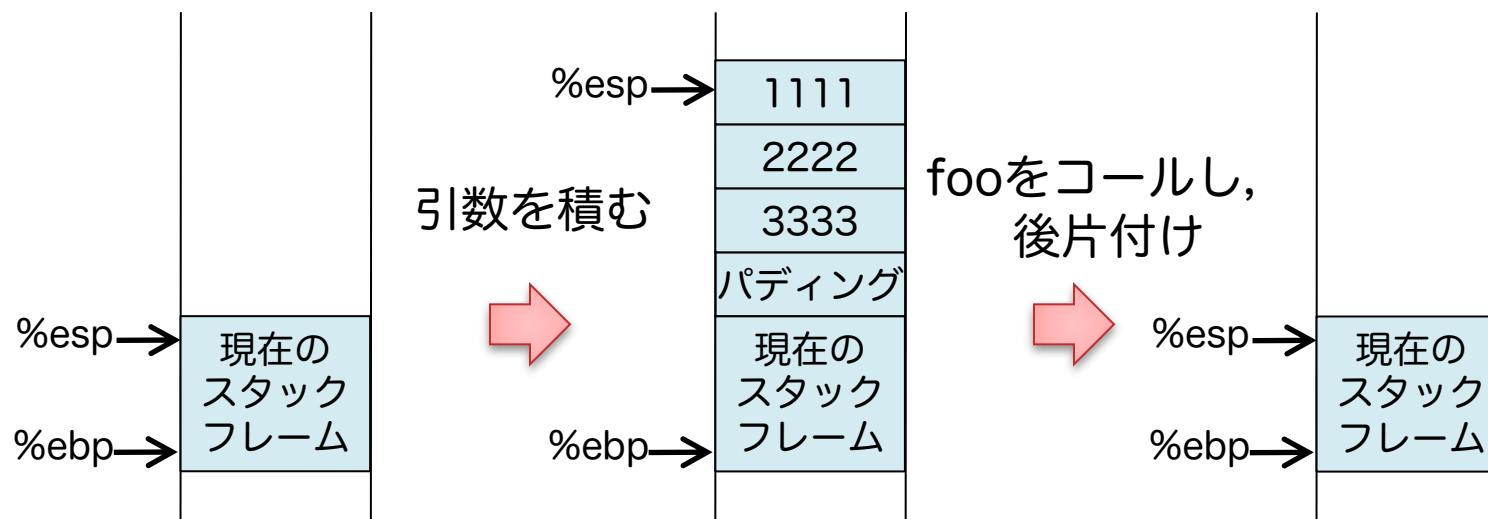
一昔前のGCCが吐くコード.  
引数をpushするので自然.

# 典型的な関数呼び出し（1）

```
void foo (int a, int b, int c);
void foo2 (void)
{
    foo (1111, 2222, 3333);
}
```

```
% gcc --mtune=i386 --static -S foo.c
```

```
subl    $4, %esp
pushl   $3333
pushl   $2222
pushl   $1111
call    _foo
addl    $16, %esp
```



Mac OS XのABIが、**関数呼び出し時にスタック  
ポインタが16バイト境界**を指すことを要求するので  
パディングが入っている。

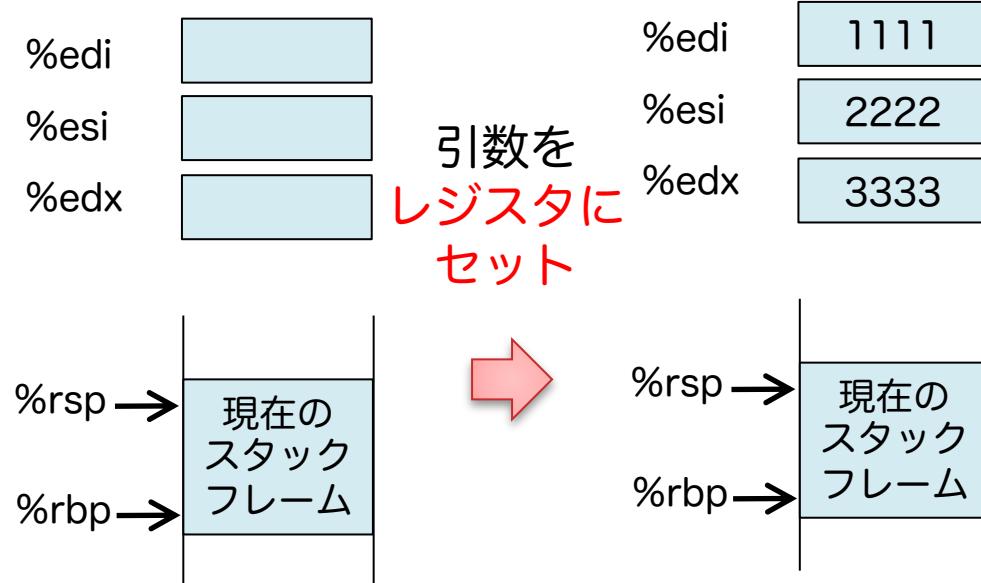


最近のGCCが吐くコード.  
引数をレジスタにセット.

## 典型的な関数呼び出し（2）

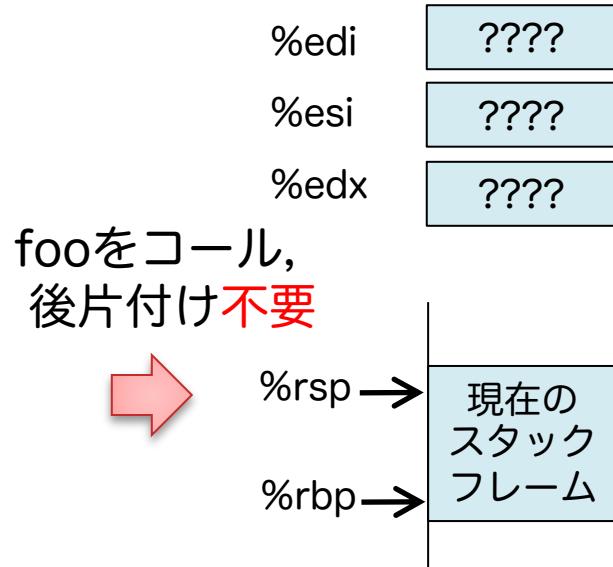
```
void foo (int a, int b, int c);
void foo2 (void)
{
    foo (1111, 2222, 3333);
}
```

```
% gcc -S foo.c
```



```
movl $3333, %edi
movl $2222, %esi
movl $1111, %edx
callq _foo
```

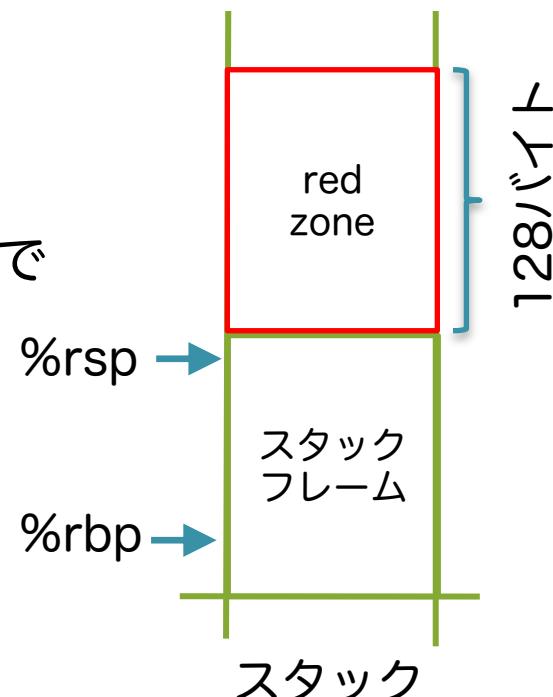
caller-saveレジスタ  
なので必要なら退避と  
復旧は必要



第7引数以降は (pushではなく)  
movでstack上にセット

# red zone

- 通常は%rspを動かして局所変数等の場所を確保。
  - 確保 : subq \$16, %rsp
  - 解放 : movq %rbp, %rsp
- red zone は右図の場所
  - %rsp-1 ~ %rsp-128 の領域
  - 局所変数や一時データ置き場に使用可
  - シグナルハンドラや割り込みハンドラで破壊されない
- red zoneを使うと上記2命令を使わずに済む→高速化





# Cからアセンブリコードを呼び出す

sub.s

```
.text
.globl _sub
_sub:
    pushq %rbp
    movq %rsp, %rbp
    subq %rsi, %rdi
    movq %rdi, %rax
    leave
    ret
```

main.c

```
#include <stdio.h>
int sub (int, int);
int main (void)
{
    printf ("%d\n",
            sub (23, 7));
}
```

```
% gcc main.c sub.s
% ./a.out
16
%
```

ラベルの先頭にアンダースコア(\_)がつくかどうかはプラットフォーム依存.



# アセンブリコードからCを呼び出す

main.s

```
.text
.globl _main
_main:
    pushq %rbp
    movq %rsp, %rbp
    movq $23, %rdi
    movq $7, %rsi
    call _sub
    leave
    ret
```

sub.c

```
int sub (int a, int b)
{
    return a - b;
}
```

```
% gcc main.s sub.c
% ./a.out
% echo $status
16
%
```

main関数の返値は終了コード（8ビット）として参照可能。  
csh系では \$status, sh系では \$? というシェル変数に格納。

# アセンブリコードからprintfを呼び出す

```

.data
L_fmt:
.ascii "%d\n\0"
.text
.globl _main
_main:
    pushq %rbp
    movq %rsp, %rbp
    leaq L_fmt(%rip), %rdi
    movq $999, %rsi
# pushq $888
    movb $0, %al
    call _printf
    leave
    ret

```

```
% gcc foo.s
% ./a.out
999
%
```

printf ("%d\n", 999); を呼ぶ。  
第1引数は文字列定数の先頭番地。

この文字列は .textセクションに  
置いても大丈夫

このコメントを外すと  
Segmentation fault になる。  
%rspの16バイト境界制約に  
違反するため

- ABIをちゃんと満たせば、呼び出せる。
  - 謎なこともあるので、GCCの出力を真似するのが吉。
- %alは隠し引数で、使用するベクタレジスタの数。



## おなじみの階乗計算

# fact.s を読む（1）

```
.text
.globl _fact
.p2align 4, 0x90
_fact:
    pushq %rbp
    movq %rsp, %rbp
    subq $16, %rsp
    movl %edi, -8(%rbp)
    cmpl $0, -8(%rbp)
    jg LBB0_2
    movl $1, -4(%rbp)
    jmp LBB0_3
LBB0_2:
    movl -8(%rbp), %eax
    movl -8(%rbp), %ecx
    subl $1, %ecx
    movl %ecx, %edi
```

```
int fact (int n)
{
    if (n <= 0) {
        return 1;
    } else {
        return n * fact (n - 1);
    }
}
```

第1引数 n を -8(%rbp)にコピー

n <= 0 の比較

n > 0 なら else節 (LBB0\_2)へジャンプ

返り値1を-4(%ebp)に格納

%eax = n

%ecx = n

%ecx -= 1 (n-1になる)

第1引数 %edi = n - 1



## おなじみの階乗計算

# fact.s を読む (2)

```
int fact (int n)
{
    if (n <= 0) {
        return 1;
    } else {
        return n * fact (n - 1);
    }
}
```

```
movl    %eax, -12(%rbp)
callq   _fact
movl    -12(%rbp), %ecx
imull   %eax, %ecx
movl    %ecx, -4(%rbp)
LBB0_3:
    movl    -4(%rbp), %eax
    addq    $16, %rsp
    popq    %rbp
    retq
```

%eaxを -12(%rbp)に退避  
返り値は %eaxに格納されてるはず  
退避した-12(%rbp)の値を%ecxに回復  
n \* fact(n-1)を計算  
計算結果（返り値）を-4(%rbp)に格納

計算結果（返り値）を%eaxに格納

# 補足 (1)

- 原則、即値・変位は32ビットまで、64ビットは無い。
  - 例 : addq \$0x1122334455667788, %rax NG
  - 例外 : movq \$0x1122334455667788, %rax OK
  - 32ビットの即値は64ビットの演算前に64ビットに**符号拡張**
    - 例 : 0xFFFFFFFF は、0xFFFFFFFFFFFFFFFFF になる

## 補足 (2)

- 32ビット演算の場合、32ビットの計算結果をゼロ拡張して64ビットレジスタに格納。

```
.text
.globl _main
_main:
    movq $0x1122334455667788, %rax
    addl $1, %eax
    ret
```

```
% lldb ./a.out
(lldb) b main
(lldb) run
(lldb) si
-> 0x100000fb4 <+10>: addl    $0x1, %eax
(lldb) si
-> 0x100000fb7 <+13>: retq
(lldb) register read $rax
      rax = 0x0000000055667789
```

%raxの上位32ビットが0に！！  
←