コンピュータとプログラミング
第6回（アセンブリ言語 第5回）
スタック，関数呼び出し

大山恵弘

再掲＆加筆：実行可能ファイルのフォーマット（ファイル形式）

- ELF, PE, Mach-O, a.out, COFFなどのフォーマットがある
  - UNIXではELF，WindowsではPE，macOSではMach-Oが主流
  - 互換性はない
- コード，データ，シンボル情報などの情報をどういう方法でファイル内に並べるかが各フォーマットで決まっている
- ファイルの最初の数バイトには，どのフォーマットであるかを示すマジックナンバーが入っていることが多い
  - ELF： 0x7f 0x45 0x4c 0x46 （すなわち，0x7fの後に “ELF”）
  - PE： 0x4d 0x5a （すなわち，“MZ”）
    - Mark Zbikowskiのイニシャルと言われている
  - Mach-O (32 bit)： 0xce 0xfa 0xed 0xfe （すなわち，0xfeedface）
  - Mach-O (64 bit)： 0xcf 0xfa 0xed 0xfe （すなわち，0xfeedface+1）

命令を大文字で書いてもいいか？大文字小文字混在でもいいか？

- オプコードやレジスタ名の部分は，大文字で書いても小文字で書いてもよい（それらは区別されない）
- ラベルの部分では，大文字と小文字は区別される

```
.text
.global main
main:   mov $1, %rax
        MOV $2, %rbx
        mOv $3, %rcx
        MoV $4, %rdx
        moV $5, %RSI
        Mov $6, %rdI
        cAll finish
```

```
$ ./a.out
Finished.
rax=0x0000000000000001 (1)
rbx=0x0000000000000002 (2)
rcx=0x0000000000000003 (3)
rdx=0x0000000000000004 (4)
rsi=0x0000000000000005 (5)
rdi=0x0000000000000006 (6)
...
```

問題なくアセンブル，実行できる

C言語プログラムからの アセンブリ言語プログラムの生成

- gccに **-S** オプションを付けてC言語プログラムをコンパイルすると，アセンブリ言語プログラムができる
  - C言語プログラムが書ける人は，試してみて下さい
  - 実行するコマンドの例： **`gcc -S prog.c`**
  - 実行可能ファイルの代わりに **`.s`** という拡張子のファイルができるはず
    - 上の例では **`prog.s`**
- どんなプログラムがどんなプログラムに変換されるかを見てみよう

例

```
int calc(int x, int y)
{
  return (x + y) * 7;
}

int main(void)
{
  return calc(3, 5);
}
```

gcc -S

```
        .text
        .globl  calc
calc:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -4(%rbp)
        movl    %esi, -8(%rbp)
        movl    -4(%rbp), %edx
        movl    -8(%rbp), %eax
        addl    %eax, %edx
        movl    %edx, %eax
        sall    $3, %eax
        subl    %edx, %eax
        popq    %rbp
        ret
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $5, %esi
        movl    $3, %edi
        call    calc
        popq    %rbp
        ret
```

(一部の行を省略)

関数呼び出しの基本

関数

- ほぼすべてのプログラミング言語が提供する機能
  - 一連の処理をまとめて手続きとして表現し，プログラム内で呼び出して実行することを可能にする
  - 多くの場合，関数は呼び出し側に返り値を返す

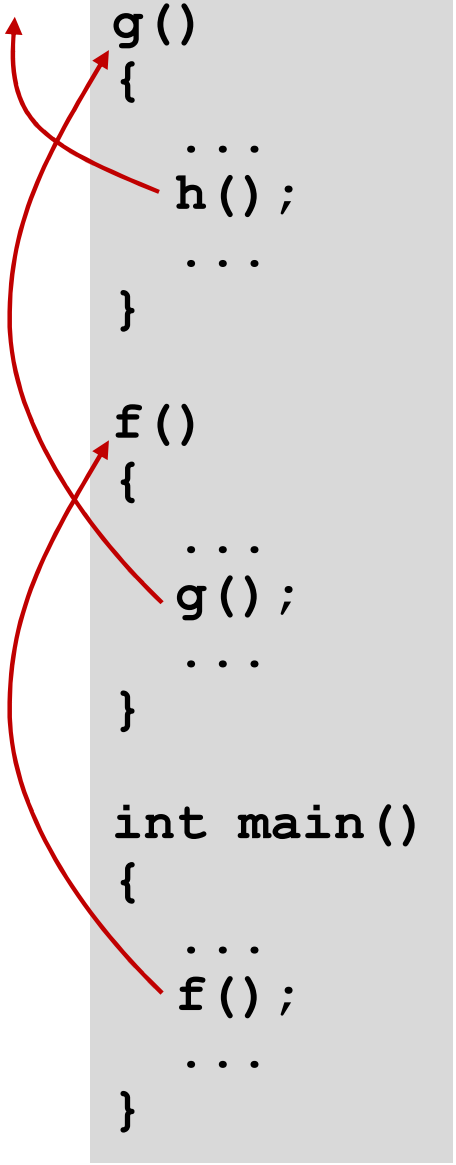
```
int plus(int x, int y)
{
  return x + y;
}

int main()
{
  int sum = plus(3, 5) + 7;
  printf("%d¥n", sum);
  return 0;
}
```

関数の呼び出しと実行で一般的に行われる処理

- 関数の呼び出し
  - 関数に渡す引数を計算し，適切な場所に配置する
  - 関数実行後に「戻るべき場所」の情報をどこかに書いておく
  - 関数のコードにジャンプする
- 関数の実行
  - 関数のコードを実行する
    - 関数に渡された引数を適宜利用する
  - 返り値を計算する
  - 関数からの復帰では，返り値を適切な場所に配置し，「戻るべき場所」にジャンプする

深い関数呼び出し構造



```
g()
{
   ...
   h();
   ...
}

f()
{
   ...
   g();
   ...
}

int main()
{
   ...
   f();
   ...
}
```

```
g:
   ...
   ...
    (h の呼び出し)
   ...
   ...

f:
   ...
   ...
    (g の呼び出し)
   ...
   ...

main:
   ...
   ...
    (f の呼び出し)
   ...
   ...
```

- 自分が順に帰るべき場所を
  どう覚えておくか？
- 10段だろうと100段だろうと，
  戻ってこなければいけない

アセンブリ言語による関数呼び出しの実現

- 単純な処理だけを行うCPU命令の組み合わせで実現する
  - 複雑な処理を行うCPU命令をCPUのハードウェアに回路として入れるのは，性能やコストの面で不利なので
  - よって，たとえば以下の処理を行う命令列を組み合わせて実現する
    - 関数実行後に使う値，関数実行後に帰るべき場所のメモリへの保存
    - 関数の命令列へのジャンプ，関数実行後に帰るべき場所へのジャンプ
    - 保存しておいた値の取り出し
    - 関数の引数と関数の返り値の受け渡し，...
- 呼び出す側（caller）と呼び出される側（callee）のコードが，共通の規約に沿うことが必要
  - 関数呼び出し規約，関数呼び出し慣例などと呼ばれる
  - 規約はCPUごと，OSごと，コンパイラごとに異なる

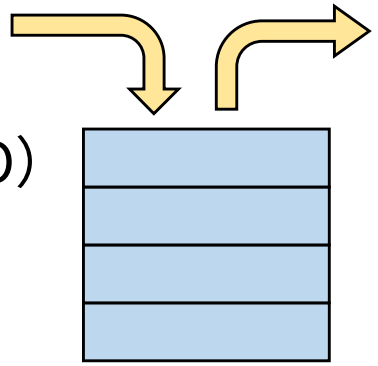
命令列で関数を実現するイメージ

```
plus:
    (引数を取り出すための命令列)
   …
   mov ...
   add ...
   mov ...
    (返り値を渡すための命令列)
    (呼び出した場所に復帰するための命令)

main:
    mov ...
    mov ...
    …
     (引数を渡すための命令列)
     (関数を呼び出すための命令)
     (返り値を取り出すための命令列)
    …
    add ...
    mov ...
```

スタックによる関数呼び出し

- 関数呼び出しを実現するための情報の管理には，しばしば，スタックが用いられる
  - 少なくともgcc + Linux + x86の組み合わせではそう
  - スタック： 先入れ後出し方式（Last-In, First-Out, LIFO）のデータ構造
    - 先に入れた要素が後に取り出される
  - 標準的な使い方：
    - プログラムは，自身の実行状態を記録するための特別なスタックを，プログラム開始時に暗黙に作成する
    - 関数を呼び出す際に，関数の実行後に思い出す情報（しばらく忘れる情報）をスタックに積む
    - 関数から復帰する際に，それらをスタックから取り出す
    - 他の情報（関数の局所変数など）も，スタックで管理する



スタックとPUSH/POP命令

- x86 (64 bit) におけるスタックの処理
  - rspレジスタがスタックポインタを保持
    - スタックポインタ： スタックの一番上の（次に取り出される）要素を指すポインタ
  - push命令でスタックに要素を入れる

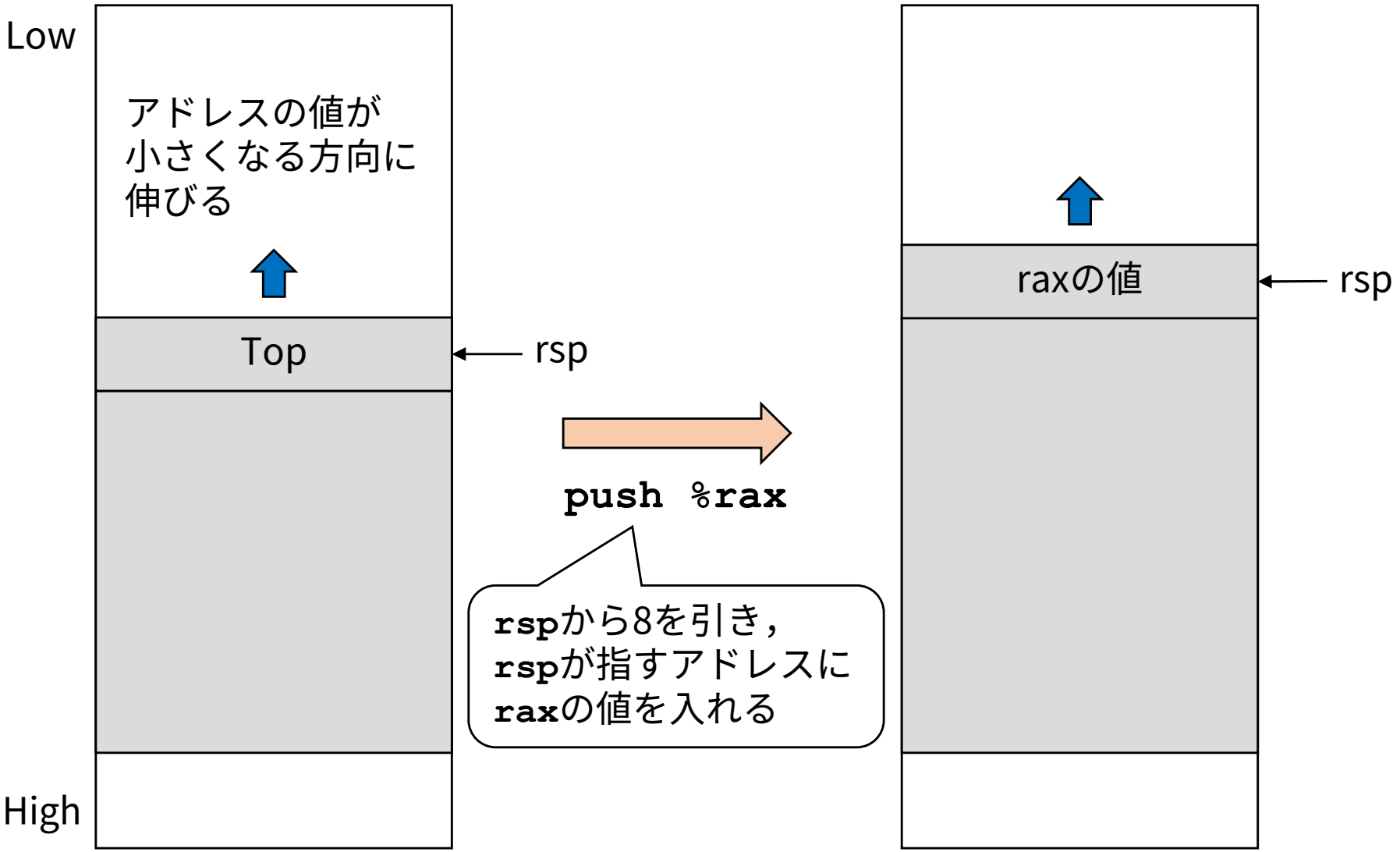
```
push src    # srcをスタックに入れる
```

  - pop命令でスタックから要素を取り出す

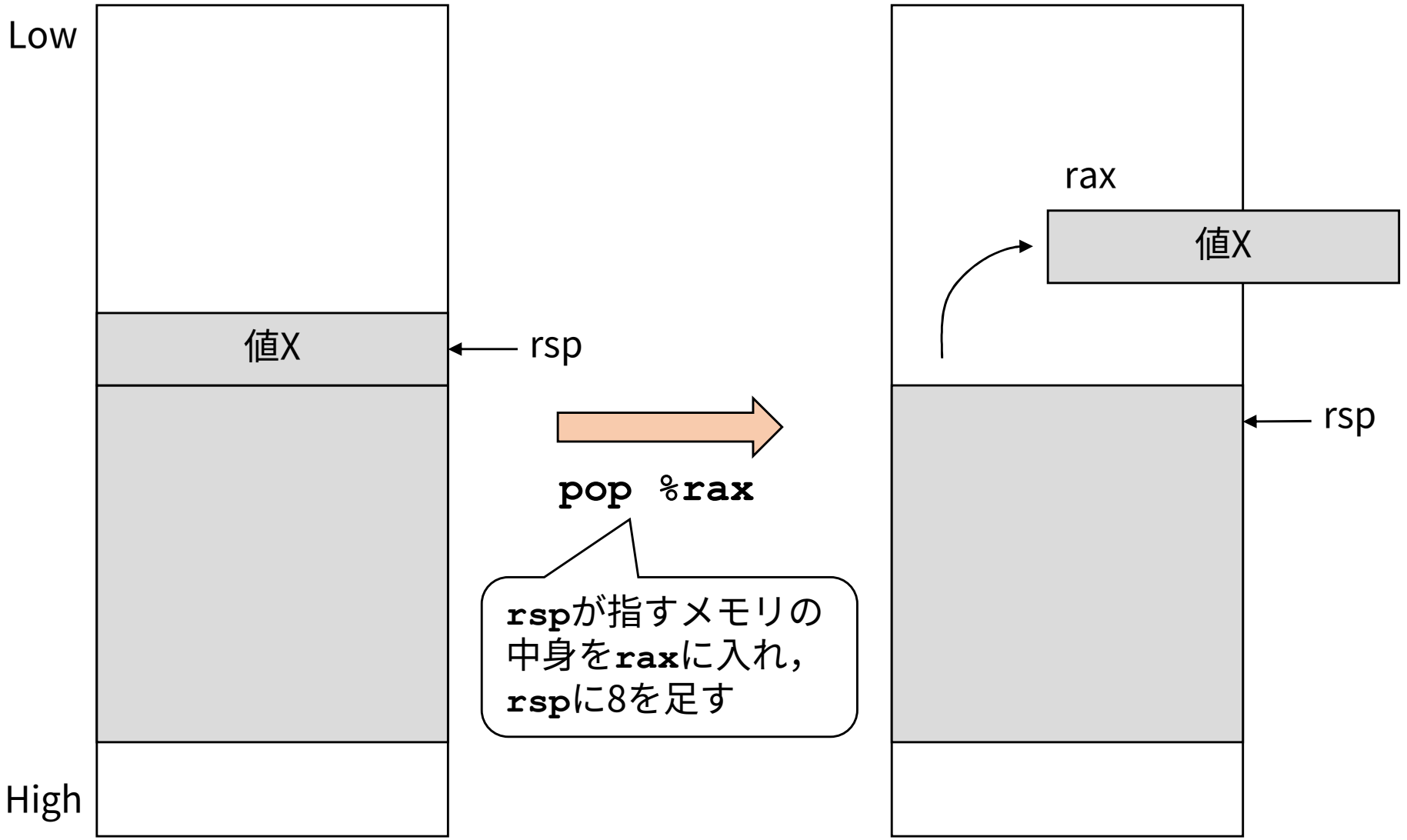
```
pop dst     # スタックから要素を取り出してdstに入れる
            # （dstはレジスタかメモリアドレス）
```

  - push命令，pop命令は，スタックポインタ（rspレジスタ）を，オペランドのデータサイズ分，減らしたり増やしたりしてくれる
  - スタックは，主に関数呼び出しの際に，レジスタの値を一時的にメモリに保存し，後でそれをレジスタに戻すために用いられる
    - 一時的にそのレジスタを「空け」て，関数の側で好きに使ってもらう

スタックとPUSH命令



スタックとPOP命令


Low
値X
rsp
High
pop %rax
rspが指すメモリの
中身をraxに入れ，
rspに8を足す
rax
値X
rsp

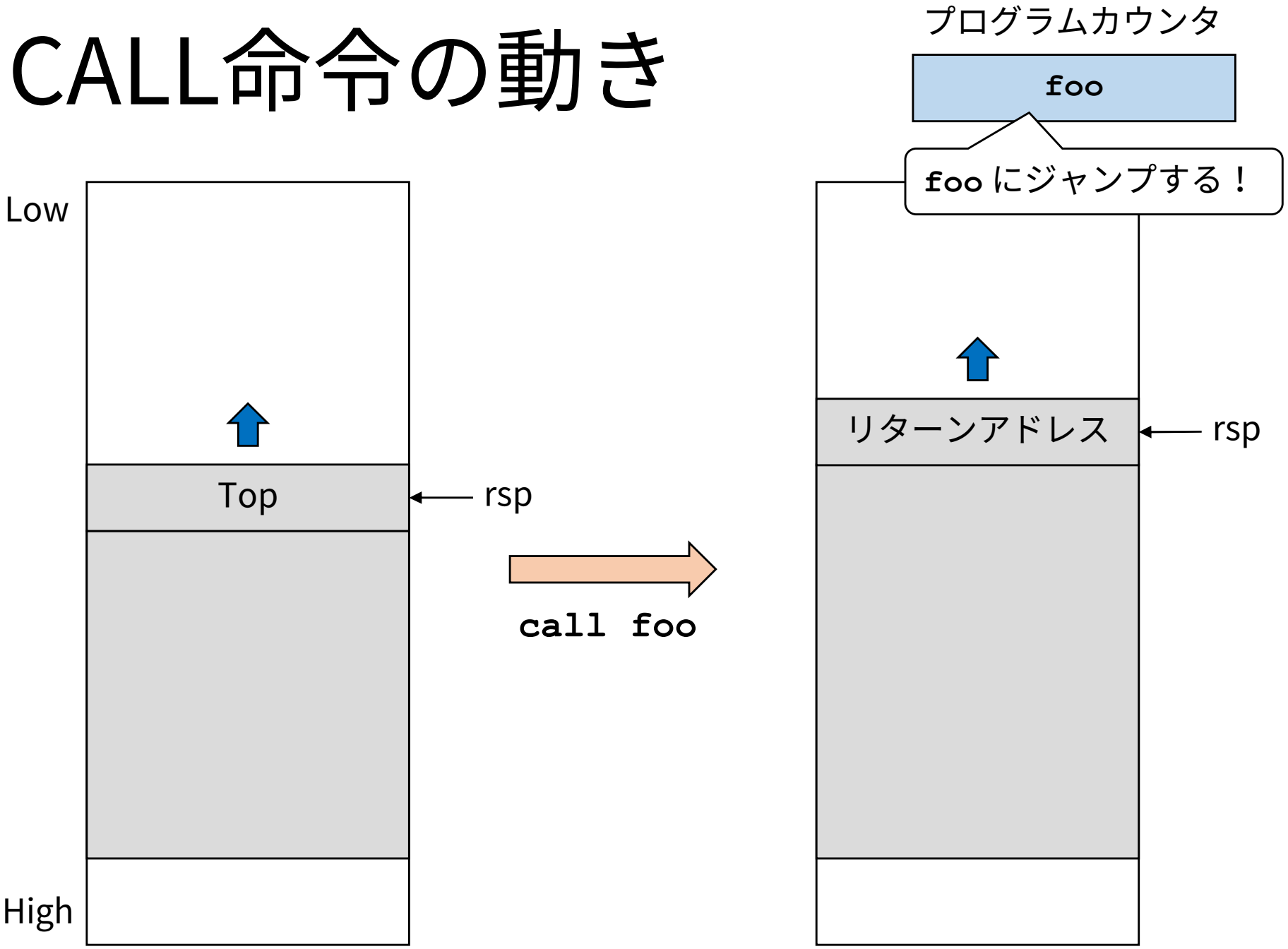
CALL命令

- 処理
  - call命令の次の（1つ下の）命令が置かれているアドレス（リターンアドレス）をスタックにpushする
  - 呼び出す関数の先頭アドレスにジャンプする

```
call label   # 関数（サブルーチン）呼び出し
```

call命令の次の命令が置かれているコードアドレスを
スタックにpushして **label** にジャンプ

CALL命令の動き

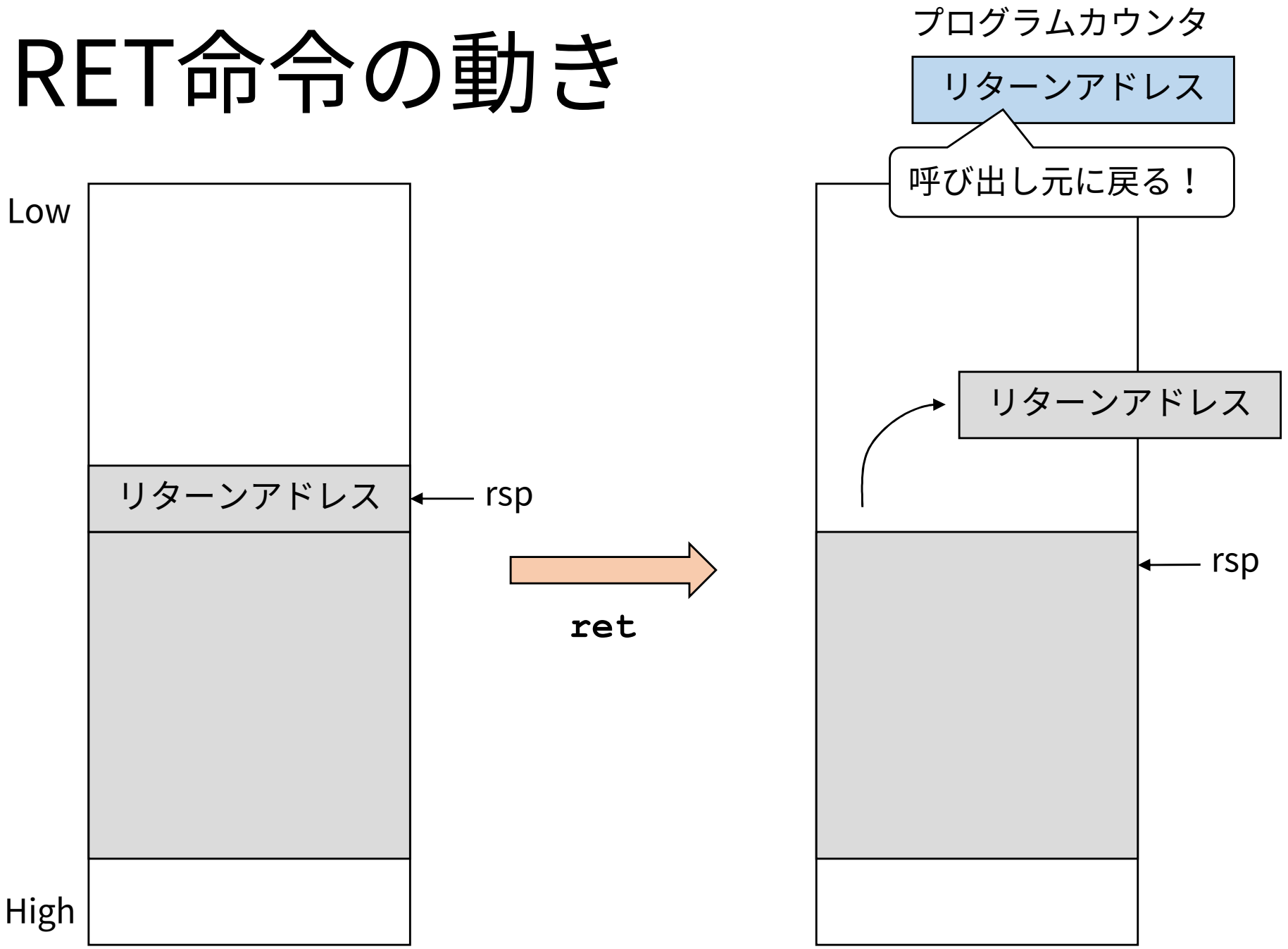

Low
Top
rsp
High
call foo
プログラムカウンタ
foo
foo にジャンプする！
リターンアドレス
rsp

RET命令

- 関数からのリターンを実行する
  - スタックの一番上に積まれている値を取り出し，その値のアドレスにジャンプする
    - ret命令を実行する際には，スタックの一番上にリターンアドレスが入っていることが必要
      - スタックの一番上はrspレジスタが指している（「rspレジスタに入っている値が，スタックの一番上のアドレスである」とみなしてプログラムは動いていく）

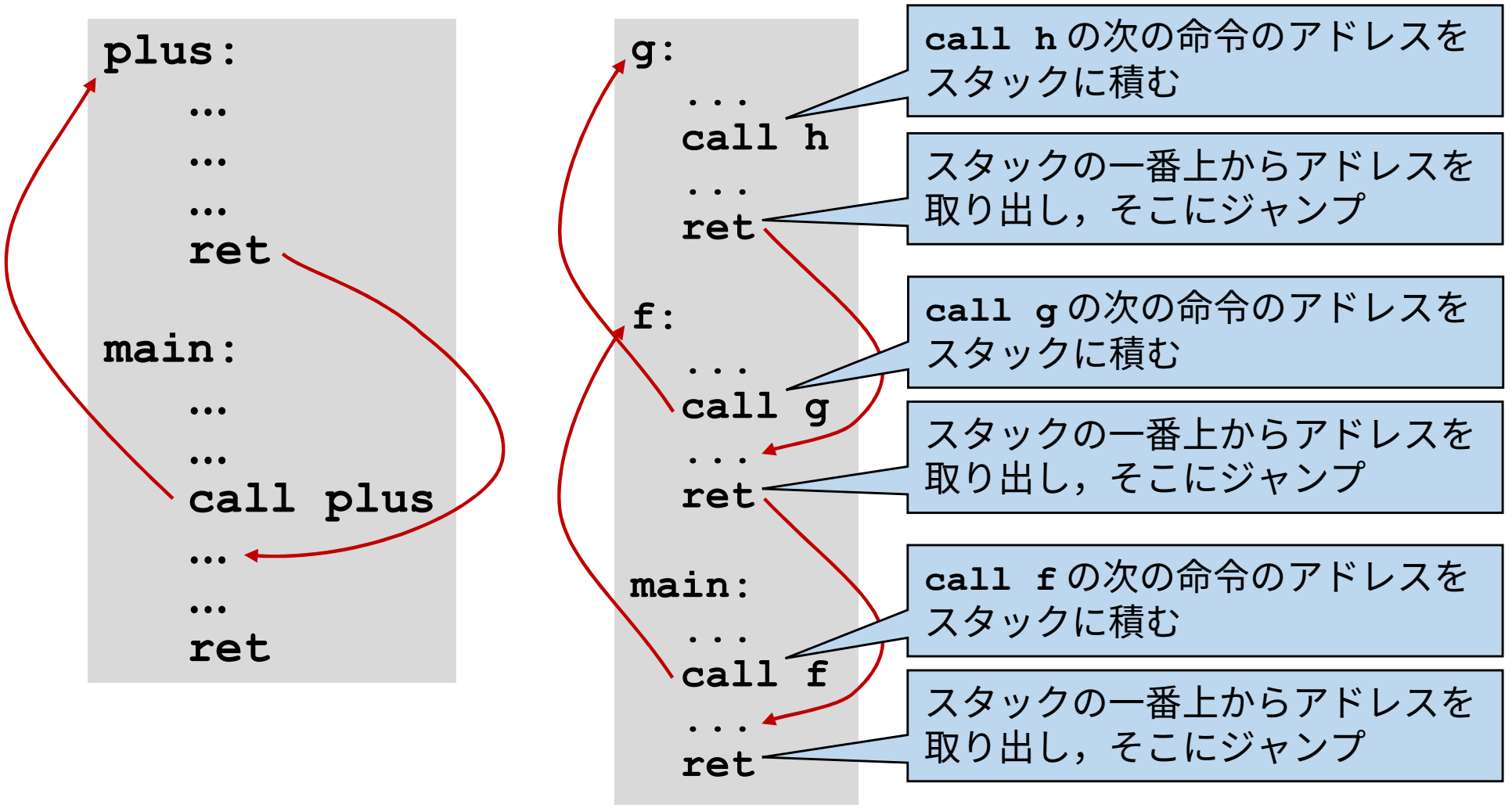
バイナリコードを逆アセンブルするとretqという命令が出てくることがあるが，それはretと同じと考えてよい

RET命令の動き


Low
リターンアドレス
rsp
High
ret
プログラムカウンタ
リターンアドレス
呼び出し元に戻る！
リターンアドレス
rsp

関数呼び出しと関数の骨組み

- まずは引数や返り値や変数は考えないとすると



関数の引数

関数への引数の渡し方

- 一般的には，2種類の方法がある
  - スタック渡し： 引数をスタックの所定の場所に積む方法
  - レジスタ渡し： 引数を所定のレジスタにセットする方法
- 呼び出し側（caller）と被呼び出し側（callee）の間で首尾一貫していれば，どちらの方法を用いてもよい
  - CallerとcalleeのAPIで一致しているかどうかが重要
  - 実際，環境，条件，システム，プログラムなどに応じて使い分けられている
- スタック（メモリ）渡しにもレジスタ渡しにも，それぞれ利点と欠点がある
  - 片方がすべての場面で常に優れているということはない

gcc + Linux + x86 (64 bit) の呼び出し規約

- Callerのやること
  - 引数を所定のレジスタにセットする
  - スタックの一番上にリターンアドレス（関数実行後に戻るべき命令のアドレス）を積む
- Calleeのやること
  - 返り値をraxレジスタにセットする
    - よって，関数呼び出し前にraxレジスタに入っていた値は破壊（上書き）される

第1引数	rdi
第2引数	rsi
第3引数	rdx
第4引数	rcx
第5引数	r8
第6引数	r9

自分一人だけが使うプログラムを書く場合，
規約に沿わなくてもプログラムは動くが，
沿えば互換性，可読性，再利用性が向上する

プログラム例（1）

- 引数を2つ受け取り，それらの和を返す関数plus
  - 第1引数をrdiレジスタ，第2引数をrsiレジスタで受け取る
  - 返り値をraxレジスタで返す

```
        .text
        .global plus
        .global main
plus:
        mov %rdi, %rax
        add %rsi, %rax
        ret
main:
        mov $111, %rdi
        mov $222, %rsi
        call plus
        call finish
```

```
$ gcc -Wall plus.s lib.s
$ ./a.out
Finished.
rax=0x000000000000014d (333)
rbx=...
```

プログラム例（2）

- 引数を2つ受け取り，それらを符号付き整数と解釈したときの大きい方の整数を返す関数max

```
        .text
        .global max
        .global main
max:
        cmp %rsi, %rdi
        jg L1
        mov %rsi, %rax
        ret
L1:
        mov %rdi, %rax
        ret
main:
        mov $8, %rdi
        mov $-3, %rsi
        call max
        call finish
```

```
$ gcc -Wall max.s lib.s
$ ./a.out
Finished.
rax=0x0000000000000008 (8)
rbx=...
```

プログラム例（3）

- アドレスAと整数Nを引数に受け取り，Aに置かれた64ビット符号なし整数の配列の先頭N要素の和を返す関数sumarray

```
        .text
sumarray:
        mov $0, %rax
L1:
        cmp $0, %rsi
        je L2
        add (%rdi), %rax
        add $8, %rdi
        dec %rsi
        jmp L1
L2:
        ret
```

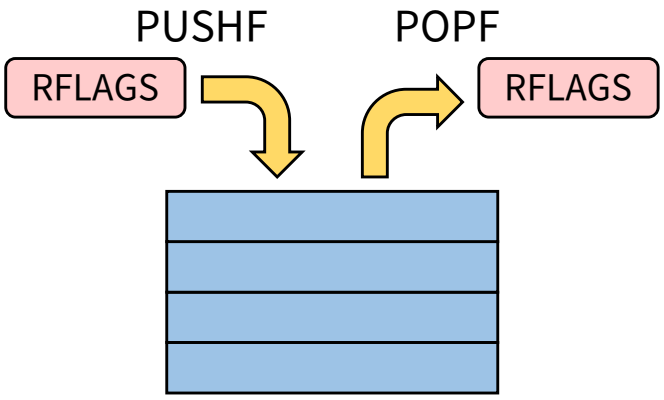
```
        .data
X: .quad 6, 13, 9, 7, 2, 4, 8, 25, 3, 16
        .text
        .global sumarray
        .global main
main:
        mov $X, %rdi # 引数Aに x を与える
        mov $5, %rsi # 引数Nに5を与える
        call sumarray
        call finish
```

```
$ gcc -Wall max.s lib.s
$ ./a.out
Finished.
rax=0x0000000000000025 (37)
rbx=...
```

スタックに関連する他の話題

PUSHF/POPF命令

- PUSHF: RFLAGSレジスタの値をスタックに入れる，すなわち，
  - RSPレジスタの値（アドレス）を8減らし，
  - RFLAGSレジスタの値をRSPレジスタのアドレスに書き込む
- POPF: スタックから値を取り出してRFLAGSレジスタにセットする，すなわち，
  - RSPレジスタのアドレスから値を読み込み，それをRFLAGSレジスタにセットし，
  - RSPレジスタの値（アドレス）を8増やす
  - あまり使わない
- RFLAGSレジスタ（%rflags）はmov命令やadd命令のオペランドに書けないので，これらの命令に存在意義が出てくる



命令の実行時間

命令の実行時間

- 命令の実行時間は命令の種類やオペランドによって激しく異なる
  - 一般に，複雑な処理をする命令や，メモリをアクセスする命令の実行には時間がかかる
  - 100倍以上異なることもよくある
  - 例えば，乗除算には加減算より時間がかかり，関数呼び出しにはレジスタ間の値コピーより時間がかかる（ことが普通）
- プログラムの実行時間をかなり雑に測る手段として，time コマンドがある
  - **`time`** *prog args...*
    - *prog args...* を実行し，実行時間（実際の時間，消費ユーザ時間，消費システム時間）を表示する

add命令とmul命令の実行時間を比較（1）

add100oku.s

```
.data
N: .quad 10000000000
.text
.global main
main:   mov N(%rip), %rbx
        mov $0, %rcx # counter
        mov $0, %rax


L1:     cmp %rcx, %rbx
        jle Lfin
        add $1, %rax # rax += 1
        inc %rcx
        jmp L1
Lfin:   call finish
```

mul100oku.s

```
.data
N: .quad 10000000000
.text
.global main
main:   mov N(%rip), %rbx
        mov $0, %rcx # counter
        mov $1, %rax
        mov $0, %rdx
        mov $1, %rdi
L1:     cmp %rcx, %rbx
        jle Lfin
        mul %rdi # 1 * 1 = 1
        inc %rcx
        jmp L1
Lfin:   call finish
```

add命令とmul命令の実行時間を比較（2）

```
$ gcc -o add100oku add100oku.s lib.s
$ time ./add100oku
Finished.
rax=0x00000002540be400 (10000000000)
…

real    0m3.492s
user    0m3.482s
sys     0m0.000s
$ gcc -o mul100oku mul100oku.s lib.s
$ time ./mul100oku
Finished.
rax=0x0000000000000001 (1)
…

real    0m8.498s   倍以上！
user    0m8.476s
sys     0m0.000s
```

演習関連

なぜazalea系マシンや新しいgccでは，ラベルに(%rip)を付けたり，gccに-no-pieを付けたりするのか？ と思っている人へ

- 「メモリのどこに置いてもいい機械語コードを生成するため」が答だが，やや高度な話
  - 詳しくは以下のWebページなどを見て下さい
    - PIC,PIE,shellcode,ASLR
      https://tanakamura.github.io/pllp/docs/pic_pie.html
- azalea系ではgccのバージョンが新しい
  - azalea系: gcc 11.4.0
  - violet系: gcc 8.5.0
- 付けないでよくするための方法が（少なくとも教員には現在）わからないので，とりあえずvioletを使っておくのが楽

専門的な話題：
スタックアラインメント

- 環境によっては，ライブラリなどのシステムが，関数呼び出し時のスタックポインタ（rsp）が16の倍数であることを求めることがある
  - その結果，call finish や call print_regs を呼び出す際，rsp の値によっては，プログラムがエラーで終了することがある
  - その場合，無駄にpushを呼んだりしてrspの値を調節するとうまく動く
    - 実はlib.sには随所でそのような調節の命令が入っている
  - 詳しくは以下の文書などを参照のこと
    x86-64 モードのプログラミングではスタックのアライメントに気を付けよう
    https://uchan.hateblo.jp/entry/2018/02/16/232029

.include アセンブラ指示

- .include "ファイル名"： この指示がある場所に，指示したファイルの中身を展開する
  - 共通の定義を複数ファイルで使うときに便利

constant.i

```
.data
fibnums: .long 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 0
```

show_fibnums.s

```
.include "constant.i"
.text
.global main
main:
  ...
```

draw_spiral_pattern.s

```
.include "constant.i"
.text
.global main
main:
  ...
```