コンピュータとプログラミング
第5回（アセンブリ言語 第4回）
ビット演算命令，繰り返し用命令，
実行可能ファイル

大山恵弘

ビット演算命令
（論理演算命令とシフト命令）

論理演算命令

- AND, OR, XOR, NOT命令
- ビットごとの論理演算
  - ビット列を数としてではなくビット列そのものとして操作

```
and src, dst  # dst = dst & src
or  src, dst  # dst = dst | src
xor src, dst  # dst = dst ^ src
not dst       # dst = ~dst
```

論理演算の意味

```
     0 0 1 1
and  0 1 0 1
-------------
     0 0 0 1
```

```
     0 0 1 1
or   0 1 0 1
-------------
     1 1 1 1
```

```
     0 0 1 1
xor  0 1 0 1
-------------
     0 1 1 0
```

```
not  1 0
---------
     0 1
```

- ANDは論理積を求める演算
  - 2つの数が両方1なら1，そうでないなら0
  - 演算例：11001101 AND 10101010 ＝ 10001000
- ORは論理和を求める演算
  - 2つの数の少なくとも片方が1なら1，そうでないなら0
  - 演算例：11001101 OR 10101010 ＝ 11101111
- XORは排他的論理和を求める演算
  - 2つの数が同じなら0，異なるなら1
  - 演算例：11001101 XOR 10101010 ＝ 01100111
- NOTは否定を求める演算

論理演算命令のよくある用途

- AND命令： 特定のビットを切り出す（落とす）

```
     789abcde
and  00ff000f
-------------
     009a000e
```

- OR命令： 特定のビットを立てる

```
     789abcde
or   0000ffff
-------------
     789affff
```

- XOR命令： 特定のビットを反転させる

```
     789abcde
xor  11111100
-------------
     698badde
```

シフト命令

- データをビット単位で左や右にずらす
  - ずらすデータとずらす幅をオペランドで与える
  - SHL命令（SHift logical Left）
  - SHR命令（SHift logical Right）
  - SAL命令（Shift Arithmetic Left）
  - SAR命令（Shift Arithmetic Right）

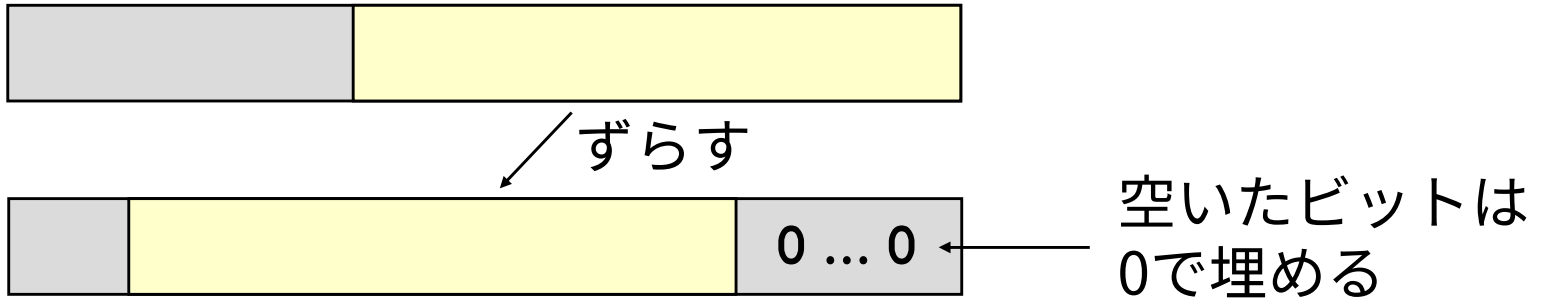
左（more significant bits）へのシフト

- SAL命令，SHL命令
  - この2つは名前が違うだけで，処理は完全に同じ

```
shl src, dst  # dst = dst << src
```

dst を **src** のビット幅だけ左にずらす

src は即値またはclレジスタ（rcxレジスタの下位8ビット部分）でなければならない



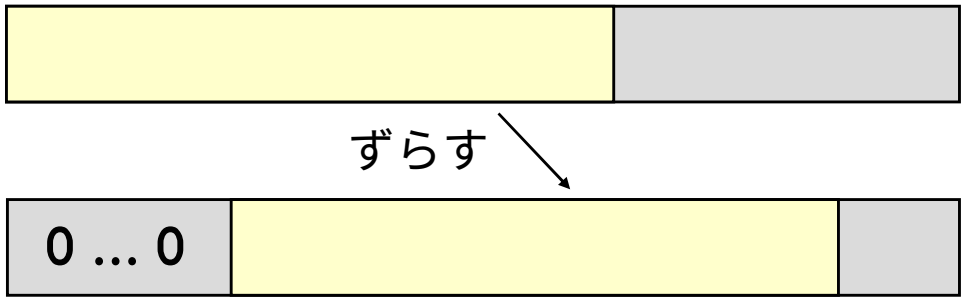
右（less significant bits）へのシフト

- SHR命令，SAR命令

```
shr src, dst  # dst = dst >> src
  上位ビットは0で埋める
sar src, dst  # dst = dst >> src
  上位ビットは最上位ビット（符号）の値で埋める
```

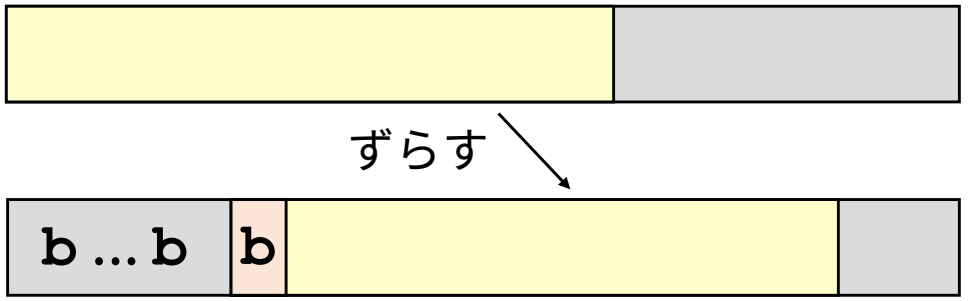
ともに，**src** は即値またはclレジスタでなければならない

SHR命令



空いたビットは0で埋める

SAR命令



空いたビットはbのビットで埋め，元の符号を維持する（負の数を正の数に変えない）

シフト命令とフラグ

- シフトによって最後にはみ出したビットが，RFLAGSレジスタのキャリーフラグ（CF）にセットされる
  - SHLでは上位ビット，SHRとSARでは下位ビット
  - CFはRFLAGS (EFLAGS) レジスタの最下位ビット
  - 例：
    raxレジスタの値が

```
0111 1001 0110 .... 1001 0101 0100 0011
```

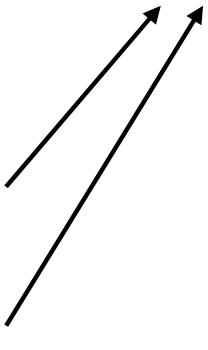
のとき，

```
shr $7, %rax
```

を実行すると，CFに1がセットされる

```
shr $6, %rax
```

を実行すると，CFに0がセットされる



サンプルコードと，実行中のレジスタの値

```
main:
    movq $0x123456789abcdef0, %rbx
 (1)
    shl $1, %rbx
 (2)
    shl $6, %rbx
 (3)
    call finish
```

```
(1) rbx=0x123456789abcdef0, CF=?
(2) rbx=0x2468acf13579bde0, CF=0
(3) rbx=0x1a2b3c4d5e6f7800, CF=1
```

```
main:
    movq $0x9876543210fedcba, %rbx
 (1)
    sar $15, %rbx
 (2)
    sar $1, %rbx
 (3)
    call finish
```

```
(1) rbx=0x9876543210fedcba, CF=?
(2) rbx=0xffff30eca86421fd, CF=0
(3) rbx=0xffff9876543210fe, CF=1
```

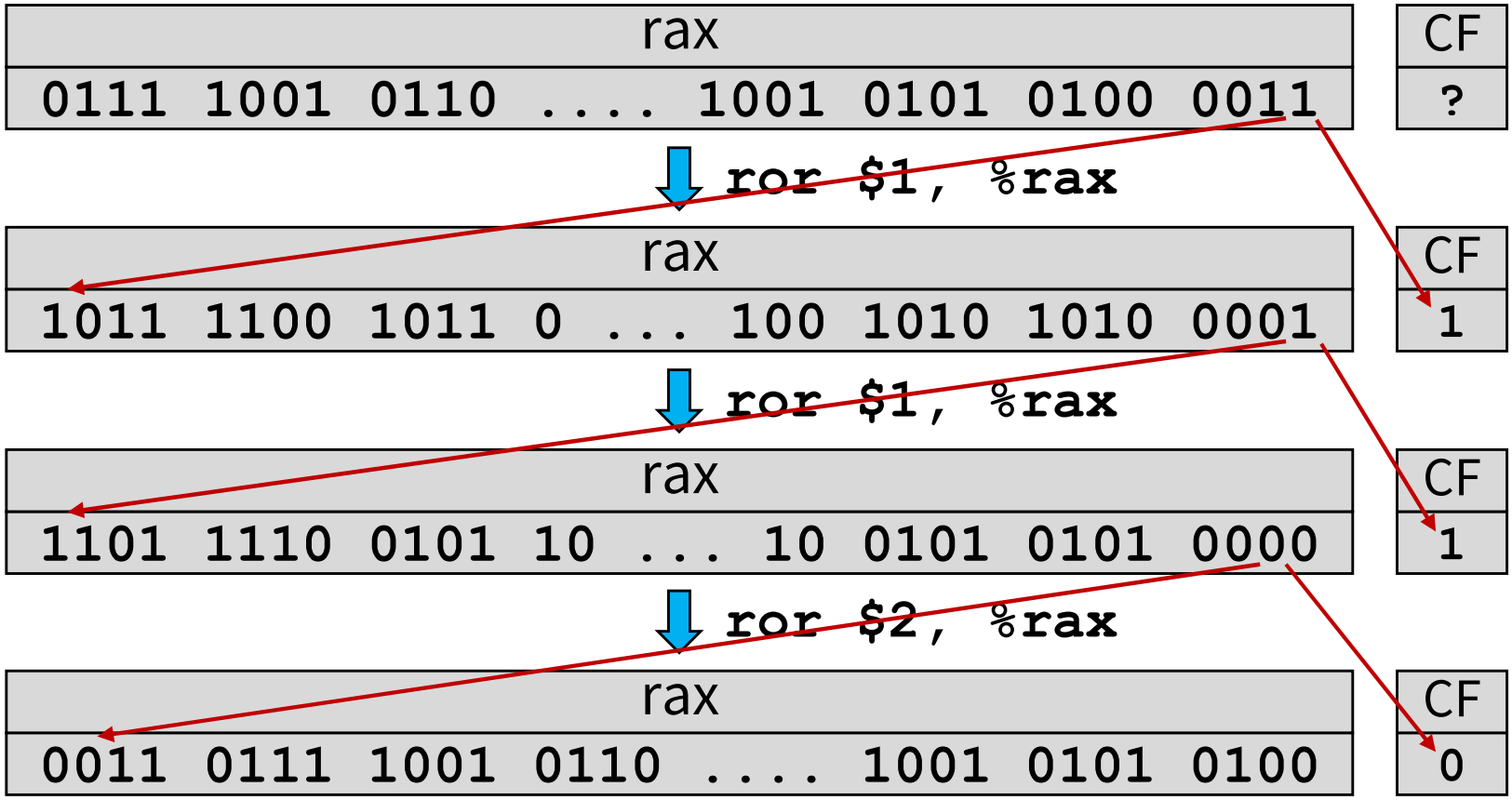
ローテイト命令

- はみ出したビットを反対側の空いたビットに移す
  - ビット列を「回転」させる
  - ROL命令（ROtate Left）： CFを含めずに回転
  - ROR命令（ROtate Right）： CFを含めずに回転
  - RCL命令（Rotate Left through Carry bit）： CFを含めて回転
  - RCR命令（Rotate Right through Carry bit）： CFを含めて回転

    - どの命令も **`src`** と **`dst`** の2オペランドをとる
    - **`src`** は即値またはclレジスタでなければならない

CFが上位に行かないローテイト命令

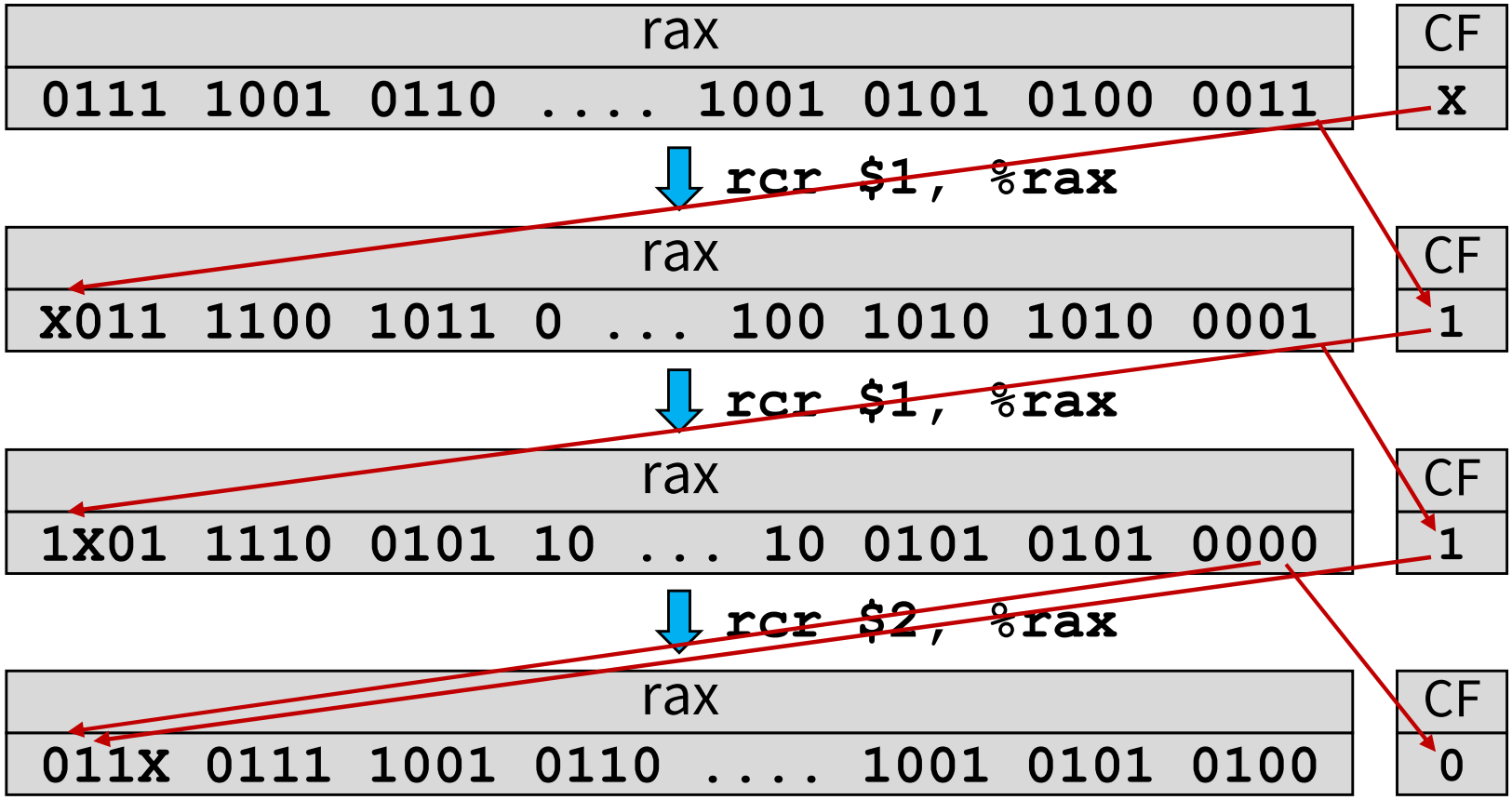
- ROR (rotate right) 命令の例



CFには最後にはみ出したビットが入る

CFが上位に行くローテイト命令

- RCR (rotate right through carry bit) 命令の例



CFには最後にはみ出したビットが入る

CFの値を含めてローテイトを実行する

サンプルコードと，実行中のレジスタの値

```
main:
    movq $0x123456789abcdef0, %rbx
 (1)
    rol $4, %rbx
 (2)
    rol $2, %rbx
 (3)
    call finish
```

```
(1) rbx=0x123456789abcdef0, CF=?
(2) rbx=0x23456789abcdef01, CF=1
(3) rbx=0x8d159e26af37bc04, CF=0
```

```
main:
    sub %rdx, %rdx # clear CF
    movq $0x9876543210fedcba, %rbx
 (1)
    rcr $4, %rbx
 (2)
    rcr $3, %rbx
 (3)
    call finish
```

```
(1) rbx=0x9876543210fedcba, CF=0
(2) rbx=0x49876543210fedcb, CF=1
(3) rbx=0xe930eca86421fdb9, CF=0
```

ビットスキャン命令，バイトスワップ命令

- BSF (Bit Scan Forward) 命令
  - **bsf** src, dst
    srcの1であるビットのうち最も下位のビットの場所をdstにセットする
    ```
    mov $0x124000, %rcx
    bsf %rcx, %rdx # rdxに14がセットされる
    ```
- BSR (Bit Scan Reverse) 命令
  - **bsr** src, dst　　BSF命令とほぼ同じだが，最も下位ではなく最も上位
    ```
    mov $0x0000123456789ab, %r8
    bsr %r8, %r9 # r9に40がセットされる
    ```
- BSWAP (Byte Swap) 命令
  - **bswap** reg
    オペランドに与えられたレジスタの値のバイトの順を逆順にする
    ```
    mov $0x0123456789abcdef, %rax
    bswap %rax # raxに0xefcdab8967452301がセットされる
    ```

繰り返し用命令
（ストリング操作命令）

繰り返し用命令

- movs (move data from string to string)
- cmps (compare string operands)
- scas (scan string)
- stos (store string)
- lods (load string)

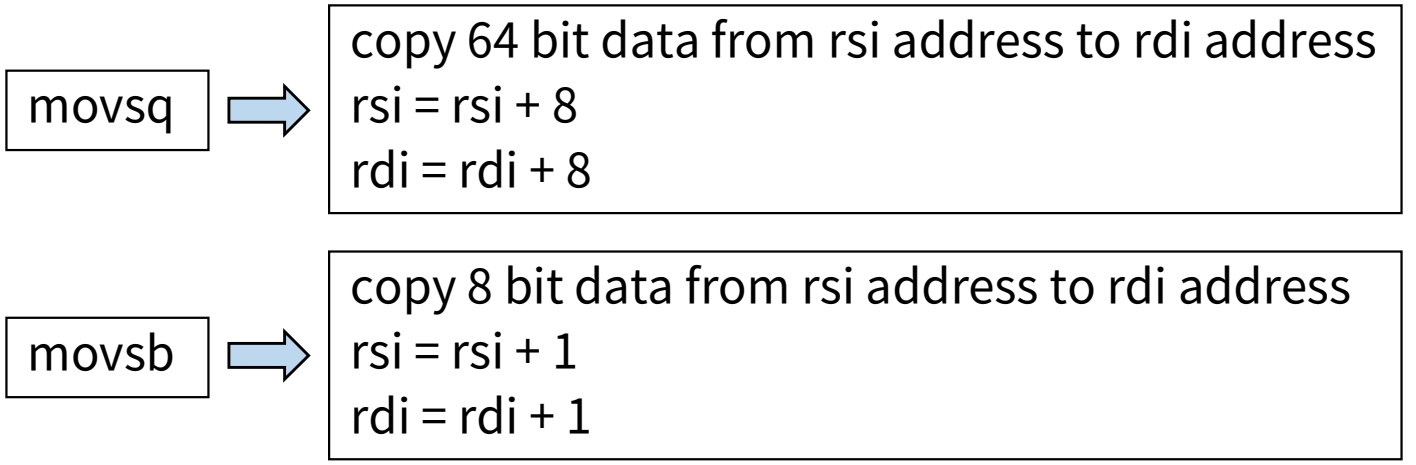
- 繰り返し用命令という呼称は一般的ではないが，この講義では便宜上そう呼ぶ
  - 一般的にはストリング操作命令と呼ぶことが多いはず

この講義での前提

- movs, cmps, scas, stos, lods命令の前には必ずrep系接頭語（後述）を付ける
- rep系接頭語を使う前のどこかでcld命令を実行しておく
  - とりあえずおまじないだと思っておけばよい
  - 正確には，eflafgsレジスタのDFフラグ（データ処理の方向を指示するビット）を0にする命令
    - 0なら，アドレスが増える方向に繰り返しが進んでいく

MOVS命令

- オペランドなし
- rsiレジスタの値のアドレスからデータを読み込み，そのデータをrdiレジスタの値のアドレスに書き込み，両レジスタの値をデータサイズ分増やす
  - cldを実行した後の場合
- データサイズに応じて，movsb, movsw, movsl, movsq命令を（接尾字を）使い分ける



MOVSQ命令の使用例 (少し人工的)

```
        .data
X:      .quad 0x10,0x20,0x30,0x40
Y:      .quad 0,   0,   0,   0
        .text
        .global main
main:
        mov $X, %rsi
        mov $Y, %rdi
        cld  # moving forward
        movsq  # Y[0] = X[0]
        movsq  # Y[1] = X[1]
        movsq  # Y[2] = X[2]
        movsq  # Y[3] = X[3]
        mov -32(%rdi), %rax  # Y[0]
        mov -24(%rdi), %rbx  # Y[1]
        mov -16(%rdi), %rcx  # Y[2]
        mov  -8(%rdi), %rdx  # Y[3]
        call finish
```

```
Finished.
rax=0x0000000000000010 (16)
rbx=0x0000000000000020 (32)
rcx=0x0000000000000030 (48)
rdx=0x0000000000000040 (64)
rsi=...
```

rep系接頭語（1）

- repという接頭語（prefix）が付いた命令は，rcxレジスタを，0になるまで1ずつ減らしながら，繰り返し実行される

```
movq $X, %rsi
movq $Y, %rdi
mov $256, %rcx
rep movsq
```

ラベル x のメモリ領域から
ラベル y のメモリ領域に，
256ワード（256×64ビット）をコピー

```
movq $str1, %rsi
movq $str2, %rdi
mov $4096, %rcx
rep movsb
```

ラベル str1 のメモリ領域から
ラベル str2 のメモリ領域に，
4096バイト（4096×8ビット）をコピー

他の繰り返し用命令

- cmpsb/cmpsw/cmpsl/cmpsq
  - rsiレジスタの値のアドレスの中身と，rdiレジスタの値のアドレスの中身を比較し，フラグレジスタを更新する
- scasb/scasw/scasl/scasq
  - rax レジスタの値と，rdiレジスタの値のアドレスの中身を比較し，フラグレジスタを更新する
- stosb/stosw/stosl/stosq
  - raxレジスタの値を，rdiレジスタの値のアドレスに格納する
    - フラグレジスタへの影響はなし
- lodsb/lodsw/lodsl/lodsq
  - rdiレジスタの値のアドレスからデータを読んでraxレジスタに格納する
    - フラグレジスタへの影響はなし

どの命令も，命令の接尾字に沿って，読み書きする
データのサイズや，rsiとrdiの増分を決定

rep系接頭語（2）

- repz： rcxレジスタが0ではない，かつ，ゼロフラグが立っている限り，rcxを1ずつ減らしながら繰り返す
  - 直感的には，繰り返し回数の上限に至っていない，かつ，演算結果や読み書きしたデータがゼロである限り
- repnz： rcxレジスタが0ではない，かつ，ゼロフラグが落ちている限り，rcxを1ずつ減らしながら繰り返す
  - 直感的には，繰り返し回数の上限に至っていない，かつ，演算結果や読み書きしたデータが非ゼロである限り

SCAS命令の使用例

Xの場所で与えられた数の列を99が出てくるまでスキャンし，
99の次の数をr12レジスタにセットして終了するプログラム

```
          .data
X:        .quad 78,34,90,18,62,71,99,25,33,46
          .text
          .global main
main:     mov $X, %rdi
          mov $99, %rax       # search for 99
          mov $10, %rcx       # at most 10 numbers
          cld
          repnz scasq         # compare repeatedly
          mov (%rdi), %r12    # get the next of 99
          call finish
```

初めて99が
出てくる場所

読んだ値がraxすなわち99と
異なる限り繰り返す

```
Finished.
rax=0x0000000000000063 (99)
rbx=...
rcx=0x0000000000000003 (3)
r12=0x0000000000000019 (25)
...
```

繰り返し用命令とrep系接頭語は何の役に立つのか？

- 大きなメモリ領域を所定の初期値で一気に埋めるのに役立つ
- 大きなメモリ領域の各データに対して，条件が成立するまで同じ操作を繰り返し適用するのに役立つ
- 大きなメモリ領域から所定の値の場所を探すのに役立つ

複数の命令を組み合わせてループの形で同じ処理を実現してもいいが，こちらでは1命令で実現できる

実行可能ファイル

実行可能ファイル (Executable Files)

- 多くの種類の情報を含む
  - プログラムのコード： バイナリ形式での機械語命令の列
  - プログラムのデータ： アドレス，サイズ，データの初期値
  - シンボル情報： main, f, printf, str などの関数や変数の名前と，それらに関する情報（アドレスやサイズ）との間の関連付け
  - ライブラリ情報： 実行時に使用する必要があるライブラリのファイル名と関数名
  - メタデータ： 対象とするOSやCPU，実行開始アドレス
- それらの情報は互いにつながっている
  - 配列を参照する命令のオペランドには，その配列が置かれる予定のアドレスが埋め込まれるなど

実行可能ファイルのフォーマット（ファイル形式）

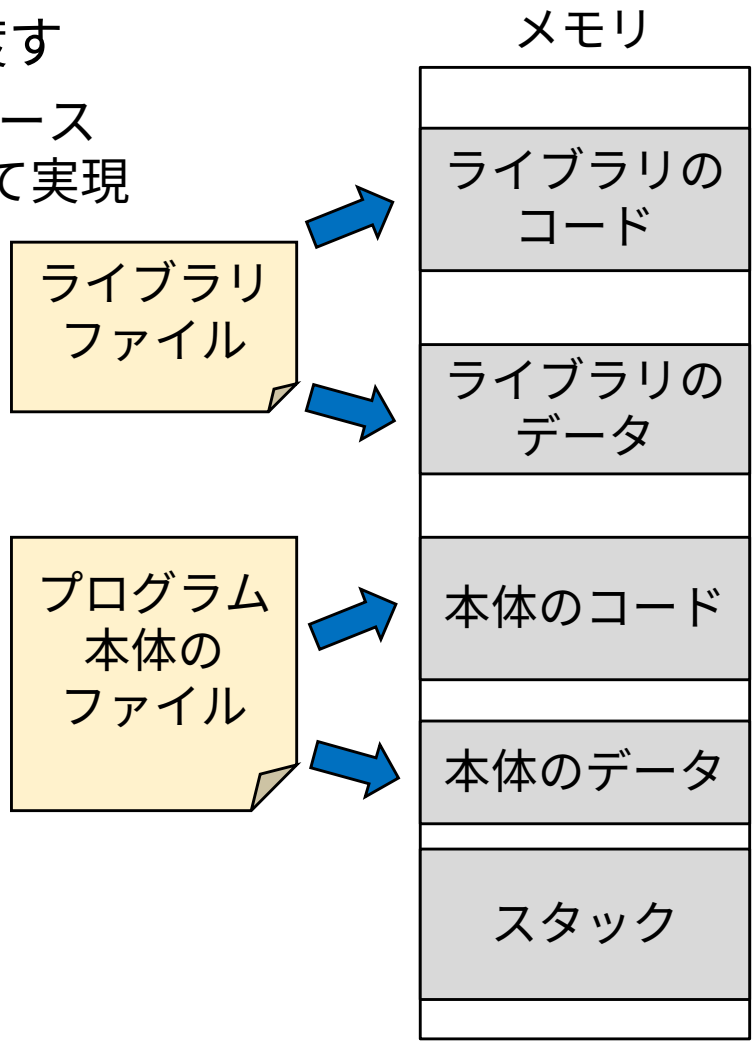
- ELF, PE, Mach-O, a.out, COFFなどのフォーマットがある
  - UNIXではELF，WindowsではPE，macOSではMach-Oが主流
  - 互換性はない
- コード，データ，シンボル情報などの情報をどういう方法でファイル内に並べるかが各フォーマットで決まっている
- ファイルの最初の数バイトには，どのフォーマットであるかを示すマジックナンバーが入っていることが多い
  - ELF： 0x7f 0x45 0x4c 0x46 （すなわち，0x7fの後に “ELF”）
  - PE： 0x4d 0x5a （すなわち，“MZ”）
  - Mach-O (32 bit)：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

実行可能ファイルが含んでいる情報を表示するのに使えるコマンド

- Linux上で，試しに実行してみよう
  （全部実行しても5分以下で終わるはず）
- **`file ./a.out`**
- **`readelf`**
- **`readelf -a ./a.out`**
- **`objdump`**
- **`objdump -x ./a.out`**
- **`objdump -d ./a.out`**
- **`hexdump -C ./a.out`**
- **`strings ./a.out`**
- **`ldd ./a.out`**

シェルからのプログラムの実行では裏で何が行われているか？

- シェルがOSに実行可能ファイルのパスを渡す
  - OSのサービスを呼び出すためのインタフェース（システムコール）を呼び出すことによって実現
- OSは以下の処理を実行する
  - その実行可能ファイルを開いて読む
  - 読んだ情報に従って，プログラムが動くためのメモリを確保する
  - 実行可能ファイルからプログラムのコードやデータを読み，メモリに載せる
  - そのプログラムが外部のライブラリを必要としている場合には，ライブラリファイルも読んで，メモリに載せる
  - メモリに載せたコードの実行開始アドレスから実行を開始する



objdumpコマンド

- 実行可能ファイルから様々な情報を出力できる
  - 実行で使われる埋め込みデータ（定数の整数や文字列など），アセンブリコード，想定CPU，大域関数と大域変数の一覧，…

```
$ objdump -d ./a.out

./a.out:     ファイル形式 elf64-x86-64


セクション .init の逆アセンブル:

0000000000400428 <_init>:
  400428:       f3 0f 1e fa             endbr64
  40042c:       48 83 ec 08             sub    $0x8,%rsp
…
0000000000400536 <calc>:
  400536:       55                      push   %rbp
  400537:       48 89 e5                mov    %rsp,%rbp
  40053a:       89 7d fc                mov    %edi,-0x4(%rbp)
…
```

アセンブリコードと機械語コード

- 機械語コードの各部分が，命令の種類，レジスタ，アドレス，数値などを表現している

```
000000000040057d <main>:
40057d: 48 c7 c0 00 00 00 00     mov $0x0,%rax
400584: 48 c7 c1 00 00 00 00     mov $0x0,%rcx
40058b: 48 83 f9 0a              cmp $0xa,%rcx
40058f: 74 0e                    je  40059f
400591: 48 03 04 cd 3c 10 60 00  add 0x60103c(,%rcx,8),%rax
400599: 48 83 c1 01              add $0x1,%rcx
40059d: eb ec                    jmp 40058b
40059f: e8 00 00 00 00           callq 4005a4 <finish>
```

raxに32ビットの値を入れるmov命令を表現
32ビットの0を表現
rcxに32ビットの値を入れるmov命令を表現
rcxと0xaを比較するcmp命令を表現
0x0eは0x40059fの位置（次の命令の位置＋14）を表現
相対位置に条件ジャンプするje命令を表現
rcxレジスタを表現
0x0060103cを表現
即値0x1を表現
0xecは0x40058bの位置（次の命令の位置－20）を表現
0xe8はcall命令の命令コード

機械語コードの表現について
さらに細かく言うと

- 最初の2バイト（0x48 0xbb）の上位13ビットが，「64ビットの整数をレジスタに入れるmov命令」を表現
- 最初の3バイト（0x48 0x29 0xd6）の上位18ビットが，「レジスタの値からレジスタの値を引いてレジスタにセットする命令」を表現

```
48 bb 90 78 56 34 12 00 00 00
mov $0x0000001234567890,%rbx
```

48 bb = 0100 1000 1011 1011

move imm64 to register　　rbx

```
48 29 d6
sub %rdx,%rsi
```

48 29 d6 = 0100 1000 0010 1001 1101 0110

subtract register value from register value　　rdx　　rsi

レジスタ	機械語コードの表現での数
rax	0 (000)
rcx	1 (001)
rdx	2 (010)
rbx	3 (011)
rsp	4 (100)
rbp	5 (101)
rsi	6 (110)
rdi	7 (111)

閑話休題：なぜレジスタがあるのか？

- こういう疑問が浮かびませんか？
  「データはすべてメモリに置けばいいのに，なぜレジスタなんてものがあるのか？」
  - レジスタがなければ，「プログラム（命令列）が単純になるかもしれない」，「CPUを小さく安くできるかもしれない」，「勉強することが少なくなるかもしれない」，．．．
- レジスタの存在意義（の1つ）：速い
  - 2020年代初頭時点では，レジスタアクセスはメモリアクセスの約100倍速い
    - CPUにとって，メモリというデバイスは亀のように遅い
  - いわば，CPUの中に作った超高速で超小型のメモリを我々はレジスタと呼んでいる

関連情報

- ブログ記事：「各種メモリ／ストレージのアクセス時間，所要クロックサイクル，転送速度，容量の目安」
  https://qiita.com/zacky1972/items/e0faf71aa0469141dede
- ブログ記事：「システムコンポーネント（CPU、メモリ、ディスク、ネットワーク等）のレイテンシとタイムスケールなどなど」
  https://kazuhira-r.hatenablog.com/entry/2021/05/01/150306

演習に関すること

Segmentation fault (コアダンプ)への対処

- プログラムを書いて実行すると "Segmentation fault (コアダンプ)" と表示されて終了することがある
- どう対処する？
  - プログラムのどこで終了したかを突き止めて下さい
  - 終了直前のレジスタの値を取得して下さい
- 具体的にはどうやって？
  - 一番の理想：デバッガ（gdbなど）を使って下さい
  - catchsegvコマンドを使って下さい
  - 色々な場所にcall finishやcall print_regsを挟んで下さい
    - 「少なくともこの行までは意図通りに実行されていた」がわかる

加筆再掲：catchsegvコマンド（やや上級者向け）

- Segmentation faultで強制終了したときのレジスタやスタックの値を表示するコマンド
  - 少なくともLinuxサーバにはインストール済み
- 例えば，memerrorがsegmentation faultで落ちるとする
  → 端末上で ./memerror ではなく catchsegv ./memerror を実行する
  - 落ちる直前のRIPやRAXなどのレジスタの値が見られるのは便利
  - 各命令のアドレスはgdbのdisassemblyコマンドやobjdump -dコマンドで分かるので，それとRIPレジスタの値を突き合わせる

```
$ catchsegv ./memerror
*** Segmentation fault
Register dump:

 RAX: 0000000000000000   RBX: 0000000000000000   RCX: 0000000000400510
 RDX: 00007ffdd3b7d328   RSI: 00007ffdd3b7d318   RDI: 0000000000000001
 RBP: 00007ffdd3b7d230   R8 : 00007f926b10ce80   R9 : 0000000000000000
 R10: 00007ffdd3b7cd60   R11: 00007f926ad66460   R12: 0000000000400400
 R13: 00007ffdd3b7d310   R14: 0000000000000000   R15: 0000000000000000
 RSP: 00007ffdd3b7d230

 RIP: 00000000004004fd   EFLAGS: 00010246
...
```

命令列が置かれるアドレスを調べるためのコマンド

- **objdump -d** *prog*
  - *prog* の逆アセンブル結果（アドレス付き，機械語コード付きのアセンブリ言語命令列）を表示する

```
$ gcc -o minimal minimal.s lib.s
$ objdump -d ./minimal

./minimal:     ファイル形式 elf64-x86-64


セクション .init の逆アセンブル:

0000000000400498 <_init>:
  400498:       f3 0f 1e fa             endbr64
  40049c:       48 83 ec 08             sub    $0x8,%rsp
…
00000000004005d6 <main>:
  4005d6:       48 c7 c0 00 00 00 00    mov    $0x0,%rax
  4005dd:       48 c7 c3 01 00 00 00    mov    $0x1,%rbx
  4005e4:       48 c7 c1 02 00 00 00    mov    $0x2,%rcx
  4005eb:       48 8b 15 3a 0a 20 00    mov    0x200a3a(%rip),%rdx # <x>
  4005f2:       48 c7 c7 ff ff ff ff    mov    $0xffffffffffffffff,%rdi
  4005f9:       48 c7 c6 fe ff ff ff    mov    $0xfffffffffffffffe,%rsi
  400600:       e8 b3 02 00 00          callq  4008b8 <finish>
  400605:       c3                      retq
…
```

再掲：演習用のプログラムlib.sが提供する便利機能

- プログラムの途中でのレジスタの内容，文字列，メモリの内容，改行の表示や，プログラムの終了
  - **`call print_regs`**　主要なレジスタの内容を表示する
  - **`call print_message`**　rdiレジスタに入っているアドレスから始まるメモリ領域に置かれたデータを文字列として表示する
  - **`call print_memory`**　rdiレジスタに入っているアドレスに置かれた64ビットのデータを16進数表記で表示する
  - **`call print_newline`**　改行を表示する
  - **`call silent_finish`**　プログラムを（何も表示せずに）終了する
  - **`call finish`**　プログラムを（メッセージやレジスタの内容を表示して）終了する

testlib.s でこれらの多くを使っているので，
そのコードと実行結果を照合すると，よりよく理解できます

例

「xで与えられる整数に1を足した値をraxレジスタに書き込んで終了するプログラムを書いて下さい」

誤った解答

```
        .data
X:      .quad 100
        .text
        .global main
main:
        mov X, %rbx
        mov (%rbx), %rax
        add $1, %rax
        call finish
```

デバッグ用のプログラム

```
        .data
X:      .quad 100
        .text
        .global main
main:
        mov X, %rbx
        call print_regs
        mov (%rbx), %rax
        call print_regs
        add $1, %rax
        call print_regs
        call finish
```

実行結果

```
$ gcc debug1.s lib.s
$ ./a.out
rax=0x0000000004005d6 (4195798)
rbx=0x0000000000000064 (100)
rcx=0x00007f8a05921758 (140230775674712)
...
rip=0x0000000004005e3
Segmentation fault (コアダンプ)
$
```

print_regsが1回実行された後に
Segmentation fault が出ている
→ どの命令で終了したかがわかる

別のやり方（1/2）

誤った解答

```
        .data
X:      .quad 100
        .text
        .global main
main:
        mov X, %rbx
        mov (%rbx), %rax
        add $1, %rax
        call finish
```

デバッグ用のプログラム

```
        .data
X:      .quad 100
        .text
        .global main
main:
        mov X, %rbx
        mov (%rbx), %rax
        call finish
        add $1, %rax
        call finish
```

```
$ gcc debug2.s lib.s
$ ./a.out
Segmentation fault (コアダンプ)
$
```

別のやり方（2/2）

誤った解答

```
        .data
X:      .quad 100
        .text
        .global main
main:
        mov X, %rbx
        mov (%rbx), %rax
        add $1, %rax
        call finish
```

デバッグ用のプログラム

```
        .data
X:      .quad 100
        .text
        .global main
main:
        mov X, %rbx
        call finish
        mov (%rbx), %rax
        add $1, %rax
        call finish
```

```
$ gcc debug3.s lib.s
$ ./a.out
Finished.
rax=0x00000000004005d6 (4195798)
rbx=0x0000000000000064 (100)
...
rip=0x0000000000400891
$
```