コンピュータとプログラミング
第4回（アセンブリ言語 第3回）
様々な演算命令とエンディアン

大山恵弘

メモリアクセスや加減算についての話の続き

LEA (Load Effective Address) 命令

- アドレスを計算してレジスタにセットする命令
- 第1オペランドで計算されるアドレスを第2オペランドにセット
  - セットされるのはあくまでアドレスそのものであり，そのアドレスのメモリに置かれているデータではない
- 使用例：
  - **`lea -40(%rbp),%rax`**
    rbpレジスタの値から40を引いた値をraxレジスタに入れる
  - **`lea X(%rdx, %rcx, 8),%r12`**
    ラベル **`x`** のアドレスにrdxレジスタの値を足し，さらにrcxレジスタの値と8の積を足した結果をr12レジスタに入れる
- 便利なので，アドレス計算のみならず，広範囲の整数演算にも利用される
  - addやsubを使わなくても加減算ができる！ それも1命令で！

INC命令，DEC命令，NEG命令

- INC (Increment) 命令は1つのオペランドをとり，
  そのオペランドの値を1増やす
- DEC (Decrement) 命令は逆に1減らす
- ADDやSUBとは異なり，フラグの値は変えない

```
inc dst  # dst = dst + 1
dec dst  # dst = dst - 1
```

- NEG (Negate) 命令は1つのオペランドをとり，そのオペランドの符号を反転させた値をそのオペランドにセットする
  - すなわち，0からそのオペランドを引いた結果
  - すなわち，オペランドに与えられた数の2の補数

```
neg dst  # dst = -dst
```

例： raxレジスタの値が2だったら，**neg %rax** の実行により，raxレジスタの値は 0xfffffffffffffffe に変わる

乗算（Multiplication）と除算（Division）

乗算命令

- 乗算と除算では，計算対象を符号無し整数と見るか符号付き整数と見るかに応じて，別の命令を使う必要がある
  - 理由： 2つの解釈の間で，演算結果が異なる（書き込むべき結果のビット列が変わる）から
- 乗算結果の格納に必要なビット長
  - 被乗算数のビット長の和
    - 例： 64ビット整数と32ビット整数の積の格納には96ビットが必要
- 除算結果の格納に必要なビット長
  - 商： 被除数のビット長
  - 剰余： 除数のビット長

IMUL (Signed Multiply) 命令

- 符号付き整数の乗算命令
- オペランド数が1のIMUL命令も，2のIMUL命令も，3のIMUL命令もあるので，好みのオペランド数の命令を使えばよい

```
imul src                # rdx:rax = rax * src
imul src, dst           # dst = dst * src
imul src1, src2, dst    # dst = src1 * src2
```

  - 1オペランド命令： 積の128ビット整数がrdxとraxに格納される
    - 上位64ビットと下位64ビットの両方を取得できる
  - 2, 3オペランド命令：積が，dstが示す場所に格納される
    - 積の上位64ビットは捨てられる
      - 小さい数どうしの乗算には問題なく使える
      - 大きい数が入る乗算では計算結果の一部が消える
    - 上位へのキャリーがあればCFとOFに1が，なければ0がセットされる
  - 3オペランド命令：積が，dstが示す場所に格納される
    - なお，dstはレジスタでなくてはならない（というx86の面倒な仕様あり）

MUL (Unsigned Multiply) 命令

- 符号無し整数の乗算命令
- 1オペランド形式の命令だけしかない

```
mul src  # rdx:rax = rax * src
```

- 乗算結果の上半分が全ビット0ならCFとOFに0がセットされ，そうでなければ1がセットされる
- Nビットの数同士の乗算では，符号付きと見ても符号無しと見ても，下位Nビット部分の結果は同じ
  → その部分だけで十分なら，IMUL命令は
  　符号無し整数の乗算にも使える

MULとIMULによる乗算の例

符号付き

```
main:
   mov $2, %rax
   mov $0, %rdx
   mov $-3, %rbx
   imul %rbx
   call finish
```

```
rdx=0xffffffffffffffff
rax=0xfffffffffffffffa
```

符号無し

```
main:
   mov $2, %rax
   mov $0, %rdx
   mov $-3, %rbx
   mul %rbx
   call finish
```

```
rdx=0x0000000000000001
rax=0xfffffffffffffffa
```

```
imul: 2 * ff...ffd (-3)       = ff...fffff...ffd (-6)
mul:  2 * ff...ffd (巨大な数) = 00...001ff...ffa (さらに巨大な数)
```

別の例

```
mov $0xffffffff, %eax  # eax <- 0xffffffff = -1
mov $0xfffffffe, %ebx  # ebx <- 0xfffffffe = -2
imul %ebx  # (-1) * (-2) = 2 (0x0000000000000002)
           # edx <- high32(eax * ebx) = 0x0 = 0
           # eax <-  low32(eax * ebx) = 0x2 = 2
```

```
mov $0xffffffff, %eax  # eax <- 0xffffffff = 4294967295
mov $0xfffffffe, %ebx  # ebx <- 0xfffffffe = 4294967294
mul %ebx   # 4294967295 * 4294967294
           # = 18446744060824649730 (0xfffffffd00000002)
           # edx <- high32(eax * ebx) = 0xfffffffd
           # eax <-  low32(eax * ebx) = 0x00000002
```

IDIV (Signed Divide) 命令と DIV (Unsigned Divide) 命令

- 128ビット整数を64ビット整数で割り，64ビットの商と余りを得る
- 乗算と同様に，符号付き用命令と符号無し用命令がある
  - 符号付きはIDIV，符号無しはDIV
- 乗算とは異なり，除算には1オペランド命令のみがある
- 被除数の上位64ビットをrdxレジスタ，下位64ビットをraxレジスタ，除数をオペランドに入れた上で実行する

```
idiv src   # rax = rdx:rax / src 商（端数切捨て）
           # rdx = rdx:rax % src 剰余
```

```
div src    # rax = rdx:rax / src 商（端数切捨て）
           # rdx = rdx:rax % src 剰余
```

サンプルコードと，実行中のレジスタの値

```
main:
    mov $0x8888666644442222, %rax
    mov $3, %rcx
 (1)
    mul %rcx
 (2)
    add $5, %rax
    mov $6, %rcx
 (3)
    div %rcx
 (4)
    call finish
```

```
 (1)
rax=0x8888666644442222 (-8608518098300689886)
rcx=0x0000000000000003 (3)
rdx=...
 (2)
rax=0x99993332cccc6666 (-7378810221192518042)
rcx=0x0000000000000003 (3)
rdx=0x0000000000000001 (1) 繰り上がり
 (3)
rax=0x99993332cccc666b (-7378810221192518037)
rcx=0x0000000000000006 (6)
rdx=0x0000000000000001 (1)
 (4)
rax=0x4444333322221111 (4919112987704430865)
rcx=0x0000000000000006 (6)
rdx=0x0000000000000005 (5) 剰余
```

除算において符号付き128ビット整数を表現するときの注意

- 表現する整数が負の場合，rdxレジスタにも0ではなく負の数をセットする必要がある
  - 例： 128ビットの－3を表現したい場合の命令列

```
mov $-3, %rax  # 0xffff...fffd
mov $-1, %rdx  # 0xffff...ffff
```

除算では例外が発生することがある（1）

- 発生すると，通常，プログラムは終了する
- 場合1： ゼロ除算

```
main:
     mov $1234, %rax
     mov $0,    %rdx
     mov $0,    %rbx
     div %rbx
     call finish
```

1234 ÷ 0 = ?

```
$ ./a.out
浮動小数点例外 (コアダンプ)
$
```

整数なのに浮動小数点例外が出るのは，おそらく歴史的理由

除算では例外が発生することがある（2）

- 場合2： 商がraxレジスタすなわち64ビットに収まらない

```
main:
     mov $0x0, %rax
     mov $0x6, %rdx
     mov $0x3, %rbx
     div %rbx
     call finish
```

```
$ ./a.out
浮動小数点例外 (コアダンプ)
$
```

0x6000000000000000 ÷ 3 = 0x2000000000000000

例外を発生させないためには？
→ 危険な入力値を与えないようにする
→ 除算の前に入力値を検査する

繰り上がり（Carry）と
繰り下がり（Borrow）

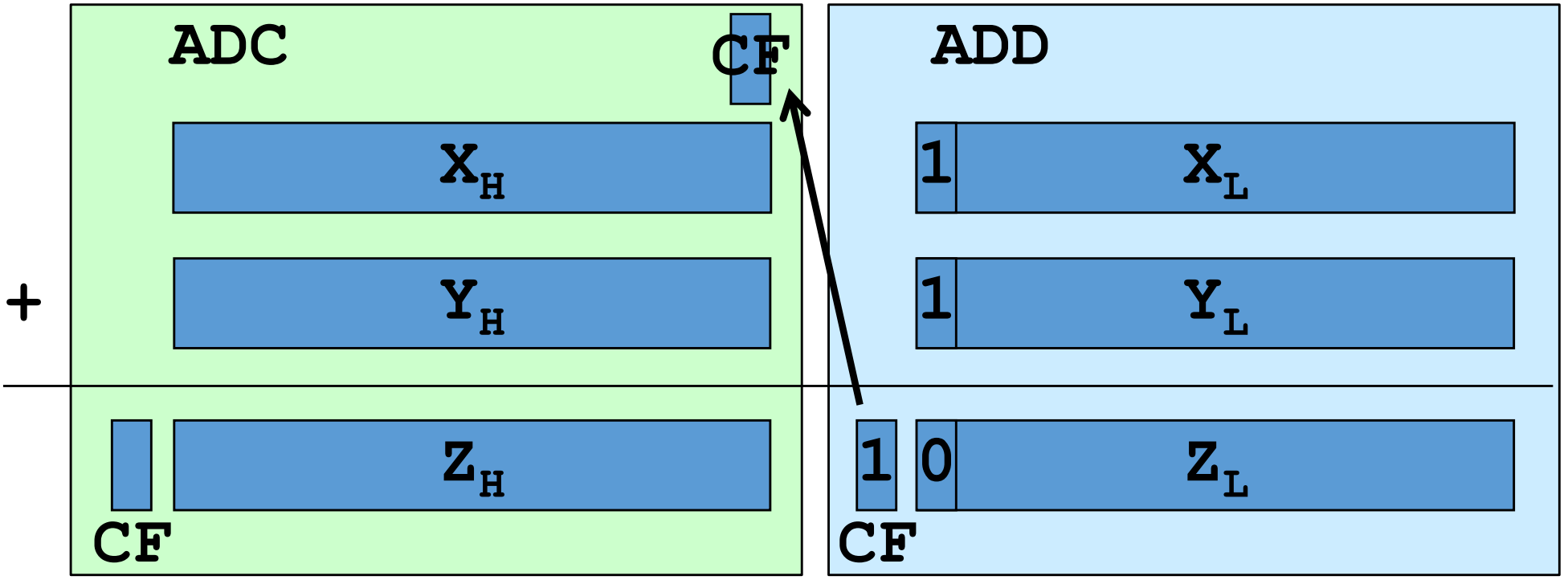
ADC (Add with Carry) 命令

- 繰り上がり処理入りの整数加算命令
- ほぼADD命令と同じだが，演算前にキャリーフラグがセットされていたら，加算結果にさらに1を足す
  - 桁あふれ（繰り上がり）を考慮した加算命令
  - nビットで数を表現しているとき，nより大きいビット長の数の加算で使う
    - 例えば，256ビットの数の加算

筆算を思い出そう

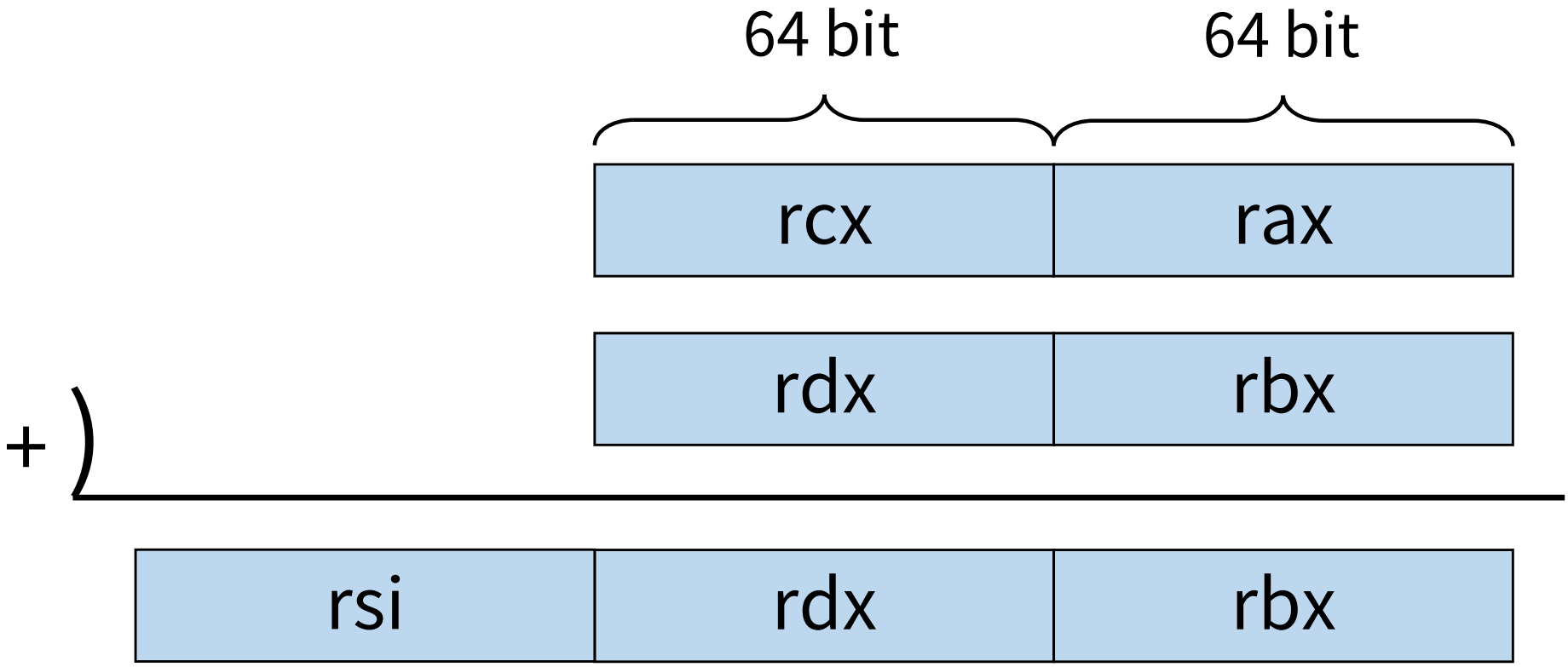
```
  456
+ 789
-----
 1245
```

ADC命令を用いて128ビット整数2つを加算する方法



1. 下位64ビットをADD命令で加算
   - 桁あふれがあれば，CFに1がセットされる
2. 上位64ビットをADC命令で加算
   - 桁あふれ分（すなわちCFの値）を合わせて加算
   - CFを，このADC命令での桁あふれで上書き

例




```
add %rax, %rbx   # rbx <- rax + rbx
adc %rcx, %rdx   # rdx <- rcx + rdx + carry
mov $0, %rsi     # rsi <- 0
adc $0, %rsi     # rsi <- carry
```

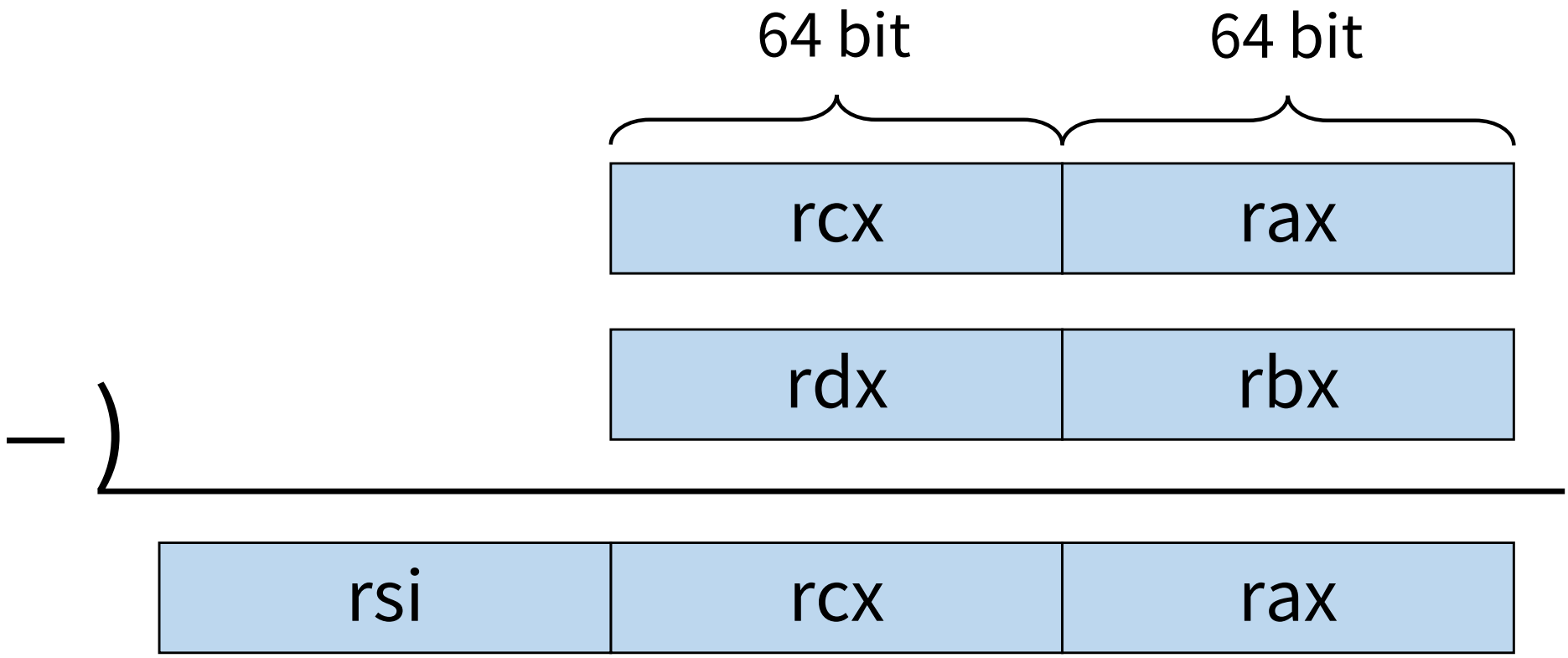
SBB (Subtract with Borrow) 命令

- 繰り下がり処理入りの整数減算命令
- ほぼSUB命令と同じだが，キャリーフラグが立っていたら，減算結果からさらに1を引く
  - ボロー（繰り下がり）を考慮した減算命令
  - SUB命令もADD命令と同じく，対象の数が符号付きでも符号無しでも，減算結果として返される0-1の並びは同じ
  - nビットで数を表現しているとき，nより大きいビット長の数の減算で使う
    - 例えば，32ビットのレジスタを使っての64ビットの数の減算

筆算を思い出そう

```
  1456
- 789
-----
  667
```

例




```
sub %rbx, %rax   # rax <- rax - rbx
sbb %rdx, %rcx   # rcx <- rcx - rdx - borrow
mov $0, %rsi     # rsi <- 0
sbb $0, %rsi     # rsi <- -(borrow)
```

サンプルコードと実行結果（繰り上がり）

```
    .data
X:  .quad 0x4000000000000000
Y:  .quad 0x8000000000000000
Z:  .quad 0x9000000000000000
    .text
    .globl main
main:
# ADC check 1 (no carry)
    mov X(%rip), %rsi
    mov Y(%rip), %rdi
    mov %rsi, %r8
    add %rdi, %r8 # calc X + Y
    movq $0, %r9
    adc $0, %r9 # calc carry

# ADC check 2 (with carry)
    mov Y(%rip), %rsi
    mov Z(%rip), %rdi
    mov %rsi, %r10
    add %rdi, %r10 # calc Y + Z
    movq $0, %r11 # calc carry
    adc $0, %r11

    call print_regs
```

```
rsi=0x8000000000000000 (-9223372036854775808)
rdi=0x9000000000000000 (-8070450532247928832)
r8 =0xc000000000000000 (-4611686018427387904)
r9 =0x0000000000000000 (0)繰り上がり
r10=0x1000000000000000 (1152921504606846976)
r11=0x0000000000000001 (1)繰り上がり
```

サンプルコードと実行結果（繰り下がり）

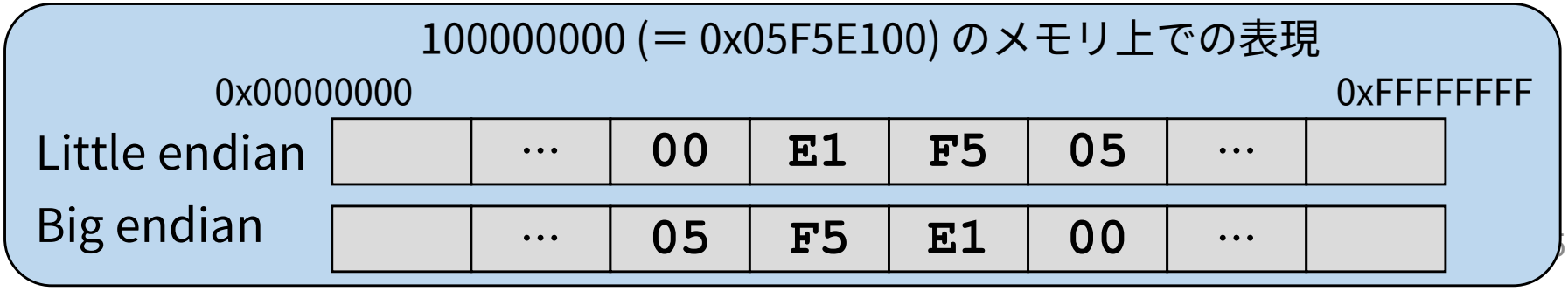
```
...

# SBB check 1 (no borrow)
    mov X(%rip), %rsi
    mov Z(%rip), %rdi
    mov %rdi, %r12
    sub %rsi, %r12 # calc Z - X
    movq $0, %r13
    adc $0, %r13 # r13 <- borrow

# SBB check 2 (with borrow)
    mov X(%rip), %rsi
    mov Z(%rip), %rdi
    mov %rsi, %r14
    sub %rdi, %r14 # calc X - Z
    movq $0, %r15
    adc $0, %r15 # r15 <- borrow

    call finish
```

```
rsi=0x4000000000000000 (4611686018427387904)
rdi=0x9000000000000000 (-8070450532247928832)
...
r12=0x5000000000000000 (5764607523034234880)
r13=0x0000000000000000 (0) 繰り下がり
r14=0xb000000000000000 (-5764607523034234880)
r15=0x0000000000000001 (1) 繰り下がり
```

エンディアン（Endian）

Big EndianとLittle Endian

- 複数バイトデータの各バイトを，どういう順番でメモリに格納するか？
- 低い（小さい）アドレスから高い（大きい）アドレスに向けて，データの最上位ビット（most significant bit, MSB）から順に格納するか，最下位ビット（least significant bit, LSB）から格納するか？
  - Little Endian: 低いビットのバイトから順に格納
  - Big Endian: 高いビットのバイトから順に格納
- 格納方法はCPUアーキテクチャに依存する
  - x86や昔のARM: little endian
  - MC68000や昔のSPARC/POWER: big endian
  - MIPSや最近のARM/SPARC/POWER: 実行時に選択可能（bi-endian）



Endian

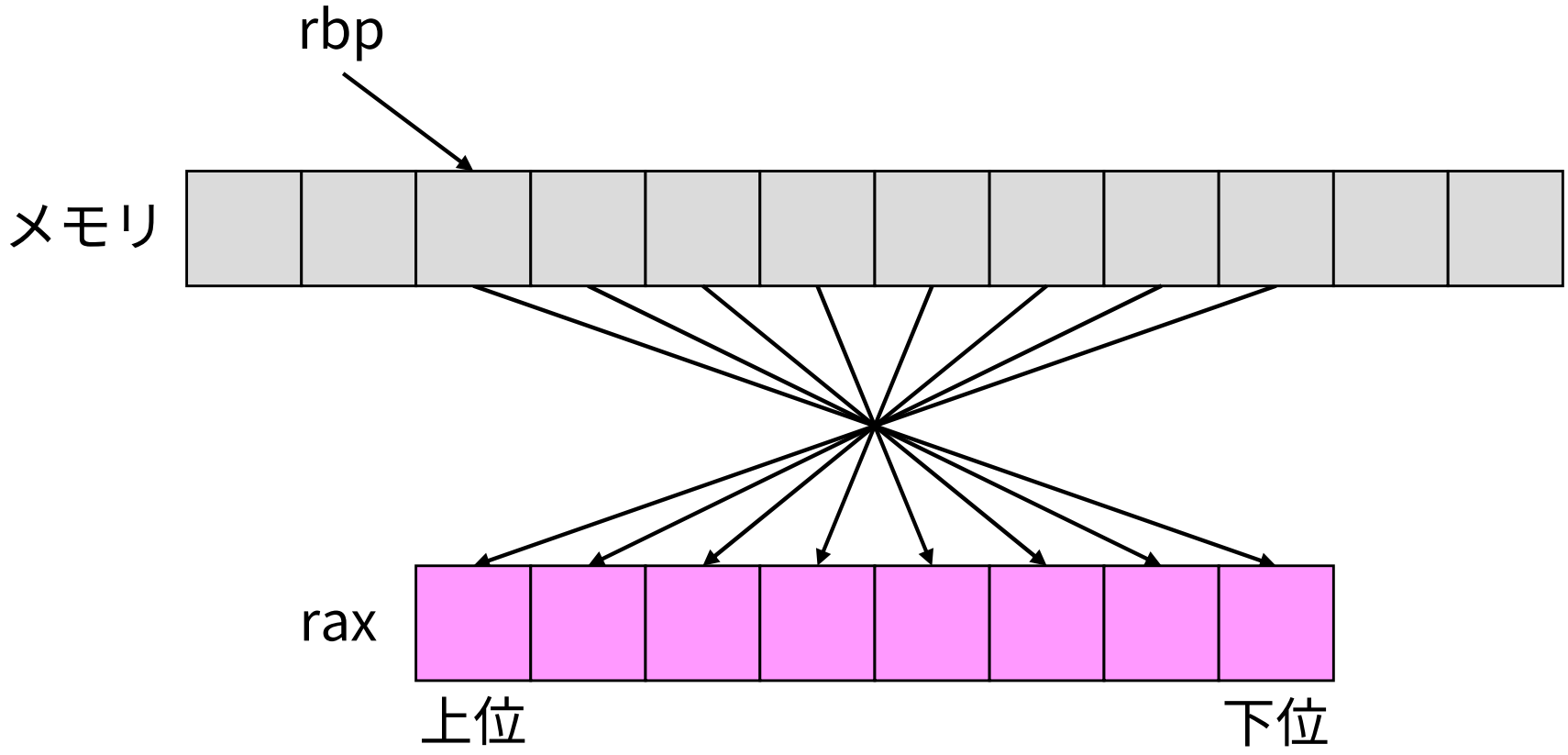
Pictures from:

https://en.wikipedia.org/wiki/Lilliput_and_Blefuscu

https://blog.litespeedtech.com/2017/09/05/developers-corner-quic-v39-added/

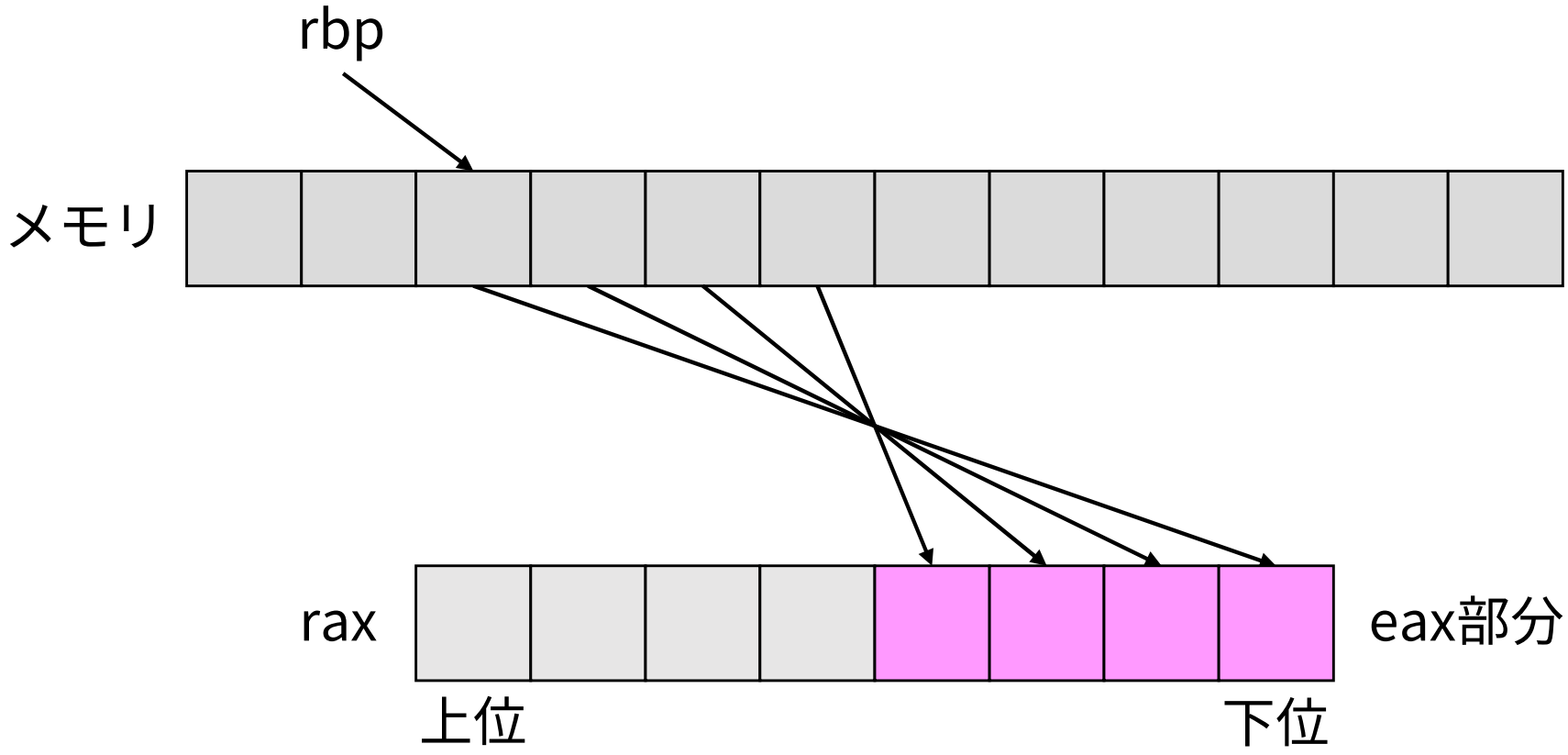
MOVQ命令とEndian

```
movq (%rbp), %rax
# rbpレジスタが指すアドレスから8バイトデータを
# 読んでraxレジスタにセット
```



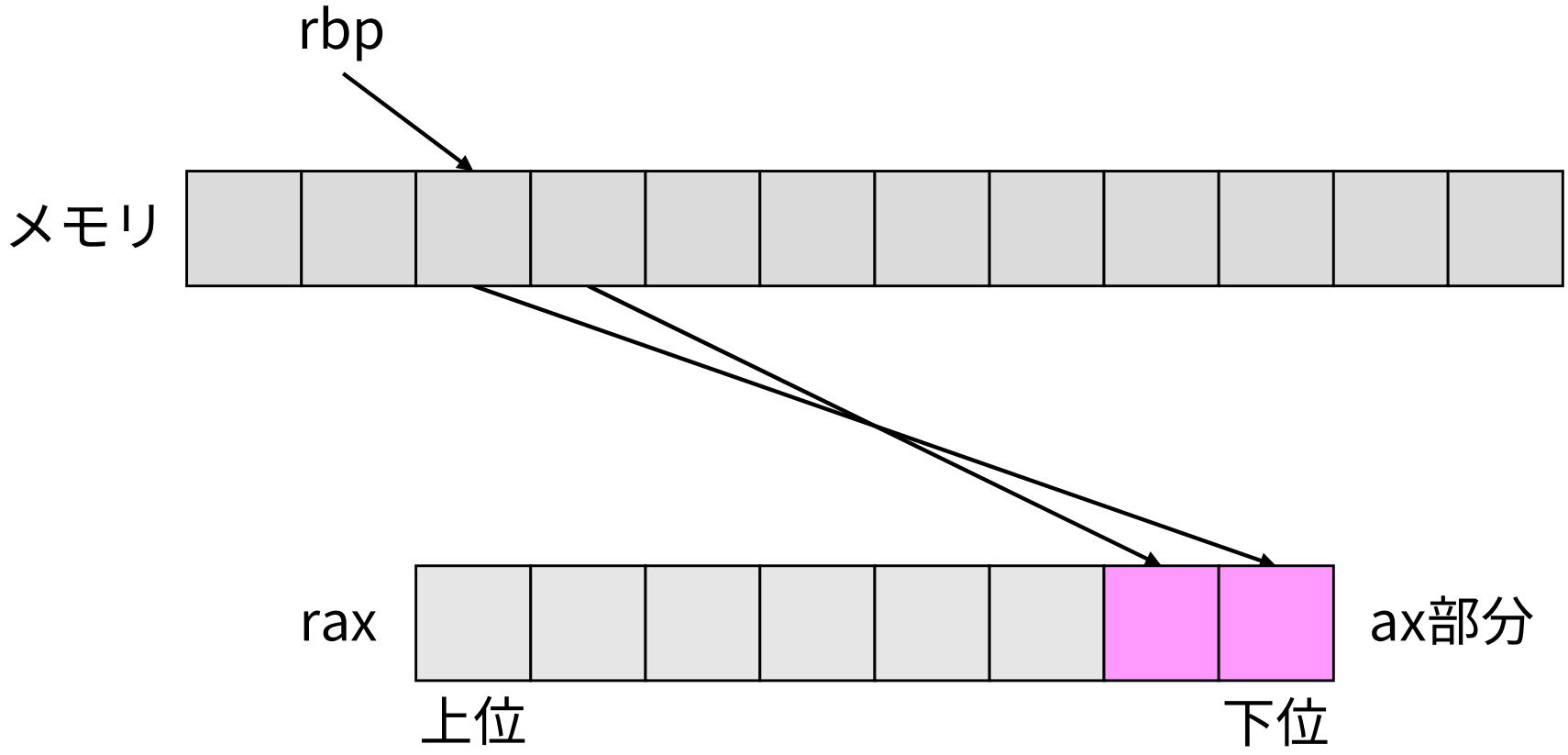
MOVL命令とEndian

```
movl (%rbp), %eax
# rbpレジスタが指すアドレスから4バイトデータを
# 読んでeaxレジスタ（raxの下位32ビット部分）にセット
```



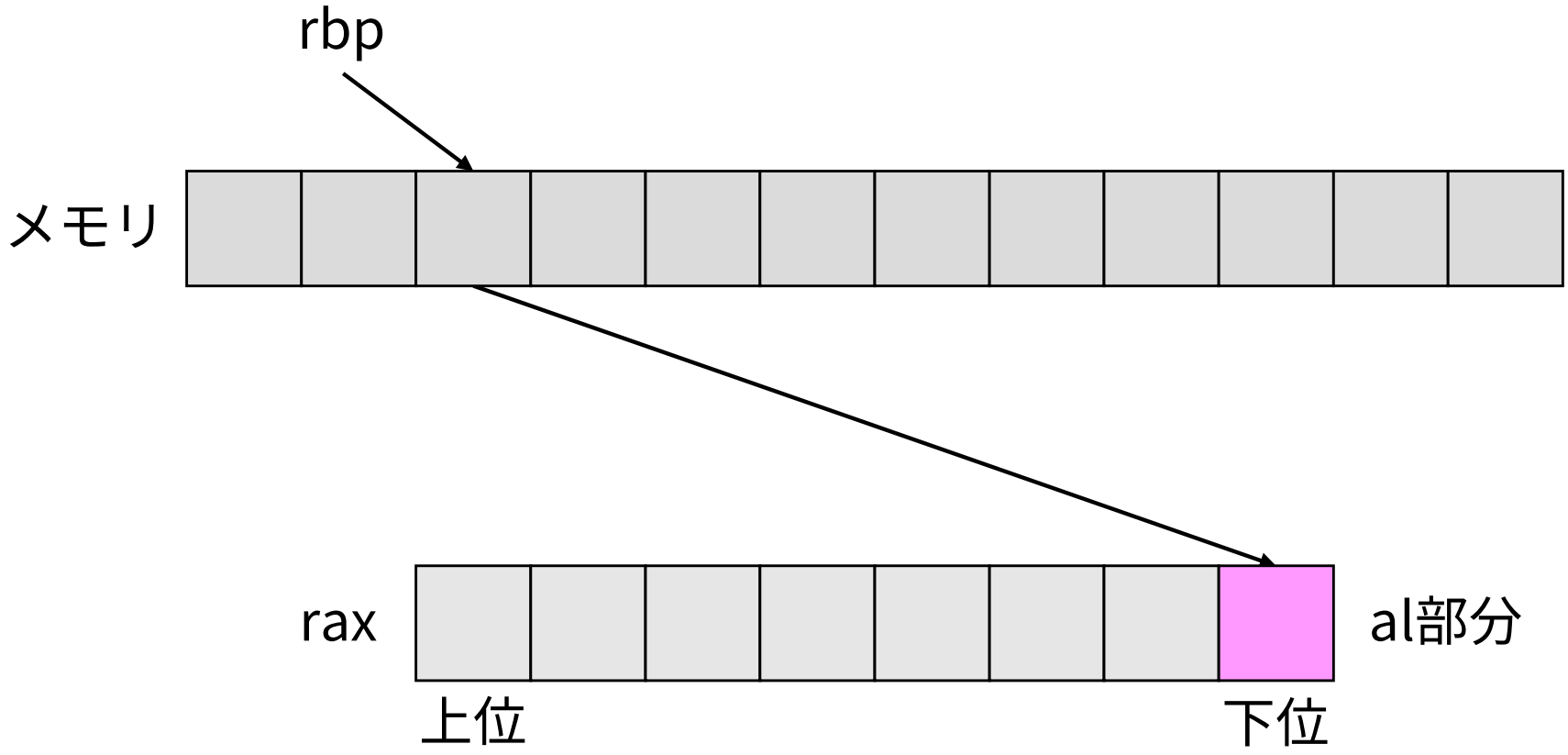
MOVW命令とEndian

```
movw (%rbp), %ax
# rbpレジスタが指すアドレスから2バイトデータを
# 読んでaxレジスタ（raxの下位16ビット部分）にセット
```



MOVB命令とEndian

```
movb (%rbp), %al
# rbpレジスタが指すアドレスから1バイトデータを
# 読んでalレジスタ（raxの下位8ビット部分）にセット
```



複数の1バイトデータの並びを 1バイトより大きい整数として読む

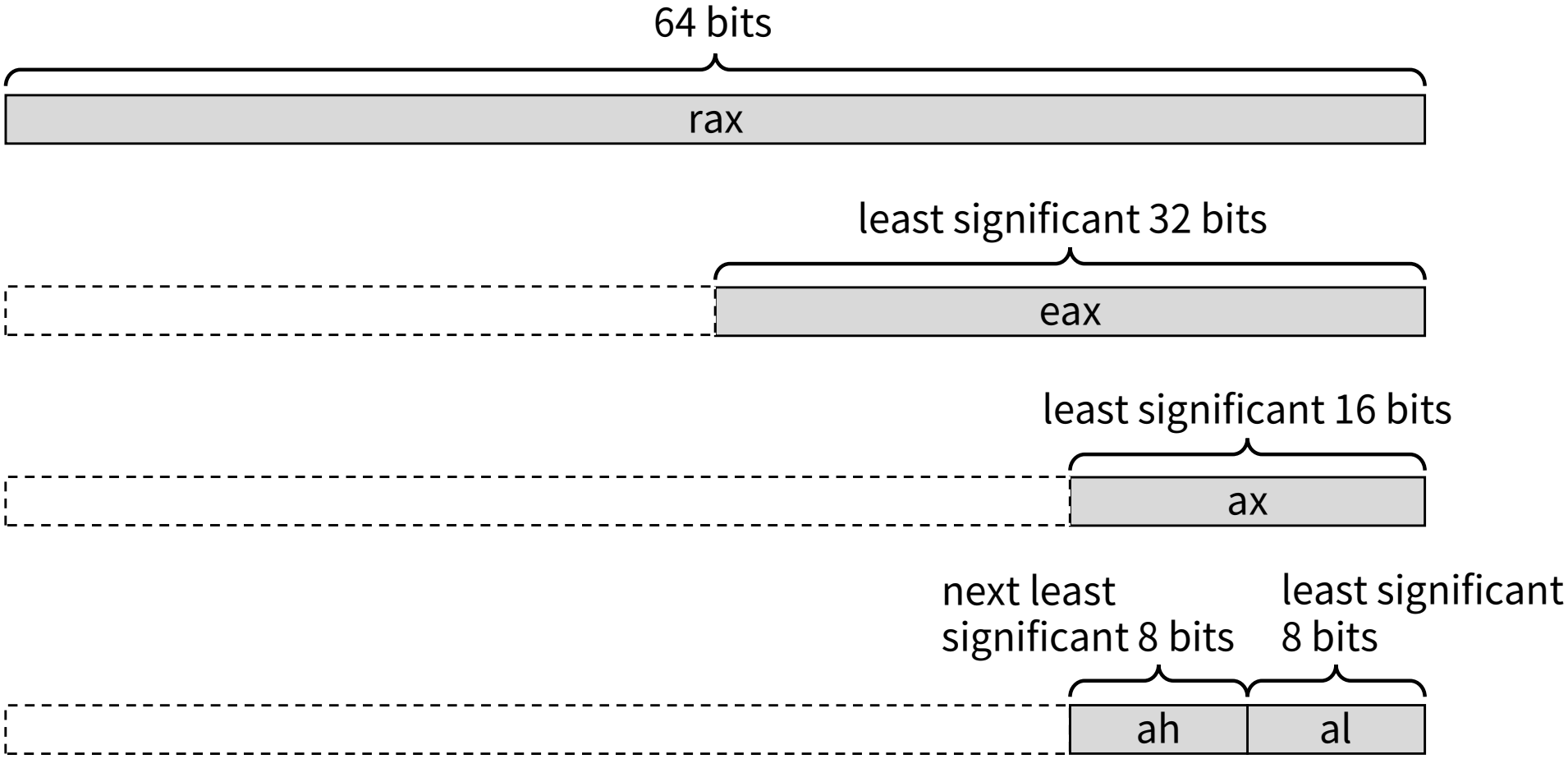
```
        .data
a:  .byte 0x00,0x11,0x22,0x33,0x44,0x55,0x66,0x77
        .text
        .globl main
main:
        mov $a,%rsi
        mov $0xaaaaaaaaaaaaaaaa,%rbx
        movq (%rsi),%rax      ← 64ビットデータとして読む
        mov $0xbbbbbbbbbbbbbbbb,%rbx
        movl (%rsi),%ebx      ← 32ビットデータとして読む
        mov $0xcccccccccccccccc,%rcx
        movw (%rsi),%cx       ← 16ビットデータとして読む
        mov $0xdddddddddddddddd,%rdx
        movb (%rsi),%dl       ← 8ビットデータとして読む
        call finish
```

```
Finished.          ← 0x0011223344556677 ではない．バイト順が逆．
rax=0x7766554433221100 (8603657889541918976)
rbx=0x00000033221100 (857870592)
rcx=0xcccccccccccc1100 (-3689348814741958400)
rdx=0xdddddddddddddd00 (-2459565876494607104)
```

32ビット読み込みの場合だけは，
上位を0で上書き（x86の謎の仕様）

扱うデータのサイズと
各サイズ用の命令やレジスタ

レジスタの一部には
別のレジスタ名がついている



というか，別のレジスタ名をつけながらx86は仕様を拡張していった

レジスタ名とデータサイズ

64ビット幅	32ビット幅	16ビット幅	8ビット幅
rax	eax	ax	al
rcx	ecx	cx	cl
rdx	edx	dx	dl
rbx	ebx	bx	bl
rsp	esp	sp	ah / spl
rbp	ebp	bp	ch / bpl
rsi	esi	si	dh / sil
rdi	edi	di	bh / dil
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

オペランドサイズの指定

- 対象データのサイズ（ビット長）に応じて，レジスタの一部を異なるレジスタ名でアクセスできる
  - 例： eaxはraxレジスタの下位32ビット部分を保持する「レジスタ」
  - 例： blはrbxレジスタの下位8ビット部分を保持する「レジスタ」
  - レジスタ名はデータのサイズを規定する
    - raxは64ビット，eaxは32ビット，axは16ビット，alは8ビット
- オペランドでデータのサイズが決まるなら，命令の接尾字で指定する必要はない
  - 例えば，オペランドに16ビットのレジスタを与えると，命令に接尾字がなくても16ビットを扱う命令としてアセンブルされる

```
mov src, %ax      # src の下位16ビットをaxにセットする命令
                  # movw と書く必要はない
add src, %al      # src の下位8ビットをalに加算する命令
                  # addb と書く必要はない
sub $100, %eax    # 32ビットの100をeaxから引き，結果を
                  # eaxにセットする命令
                  # addl と書く必要はない
```

結局，どのオペランドに何を書けるのか？

- 「アドレシングモードのカッコ内の第1要素には数値は書けるのか？」，
  「第1と第2オペランドの両方をアドレシングモードにはできるのか？」，
  「アドレシングモードのカッコ内を全部省略できるのか？」，...
- x86の仕様書には厳密に書かれているが，全部読むのは大変
- 「gcc (gas) に聞いてみる」が良さそう
  - 調べたいオペランドを入れたプログラムを試しに書いてみて，
    アセンブルできればOK，エラーが出たら修正
- 既存のアセンブリ言語コードを大量に読んで，使われている命令と
  使われていない命令を把握し，頭の中で一般化する方法もあり
  - **`objdump -d`** *program*
    などのコマンドやデバッガにより，実行可能プログラムファイルから
    （逆）生成したアセンブリ言語コードを表示できる

再度の注意：ありそうだが，あってほしいが，無い命令

- 「第1と第2の両オペランドがアドレス（メモリ参照）である命令は使えない（x86には無い）」と過去の回で説明した
  - **`mov (%rax), (%rbx)`** や **`add 80(%rbx, %rcx, 4), 16(%rsp)`** や **`mul 12(, %rsi, 8), (%rdi)`** など
  - そもそも，そのような機械語をx86が提供していない
    - 可のほうがユーザにとってはうれしいが，多様なオペランドを許すと，回路の複雑化や命令長の長大化の問題が生まれる
    - 便利さよりも性能やコストを優先した結果と思われる
- 他にも，x86にありそうだが無い命令はある
  - **`div $2`** や **`shr %rax, %rdx`** など
  - 必要に応じて個別に覚えれば十分
  - もしくは，覚えてなくても，「コンパイラ（アセンブラ）がエラーを出したら対処する」でも十分

中間試験の予告

- 日時と場所： 2025年6月6日（金） 3限 12:15〜13:30　3A204
- 試験時間：60分
  - 12:15から用紙の配布を開始し，数分後に試験開始
- 範囲： UNIXとアセンブリ言語の部分全体
- 形式： 筆記試験
  - 講義で配布した資料（スライド，演習課題，小テストの問題と解答，演習用のプログラムや資料を含む）を印刷した紙と，それらの紙や白紙やノートに手書きで書き込みを加えた紙のみが持ち込み可
  - 印刷などの手書き以外の方法で情報を書き込んだ紙は持ち込み不可
  - 時計以外の全ての電子機器の利用不可
  - 座席自由，学生証を机上に提示