コンピュータとプログラミング
第2回（アセンブリ言語 第1回）
コンピュータの仕組みと
アセンブリ言語

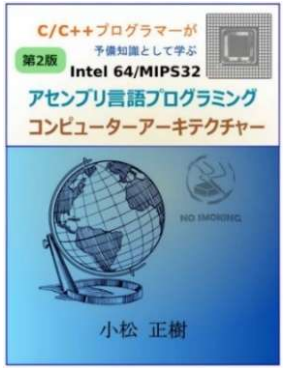
大山恵弘

講義（アセンブリ言語部分）資料

- 講義スライド，講義や演習で使うプログラムなど
- manaba上で電子的に配布
- 再配布は禁止

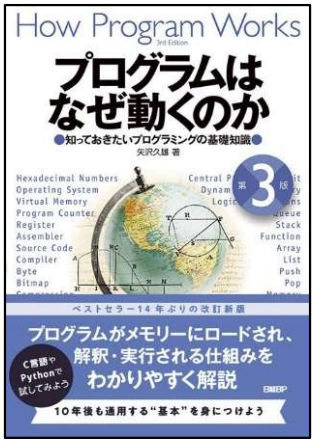
参考図書（≠教科書）

- 必須としている書籍はないが，勉強用の教材がほしい人には下記の書籍を薦める
  - 「独習アセンブラ」，大崎博之
- ざっとしか読んでいないが，下記の本も良いかもしれない
  - 「Intel64/MIPS32アセンブリ言語プログラミング コンピューターアーキテクチャー」，小松正樹
    - 長所も短所もたくさんあるが，情報の量やページ数の多さは◎
  - 「プログラムの不思議を解く」，日向俊二
  - 「大熱血！ アセンブラ入門」，坂井弘亮
    - 「熱血！ アセンブラ入門」の加筆修正版で，1164ページの大著



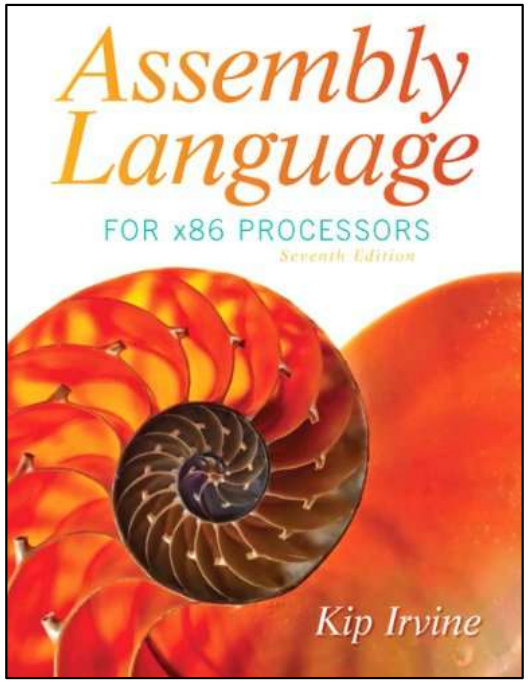
参考図書（≠教科書）

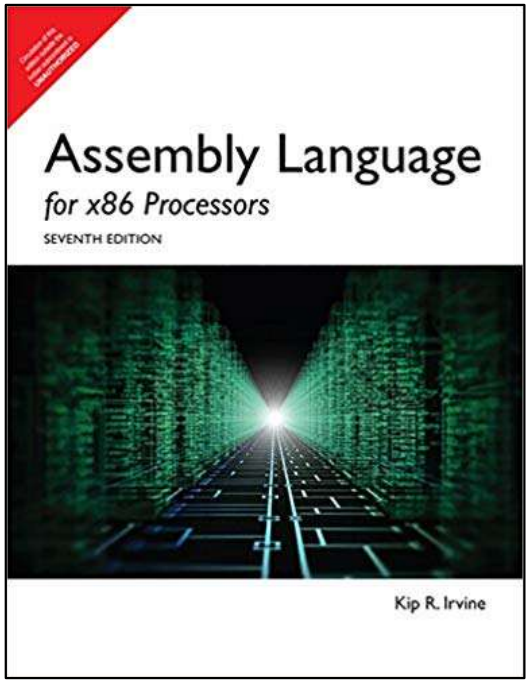
- ざっとしか読んでいないが，下記の本も良いかもしれない（続き）
  - 「作って分かる！ x86_64 機械語入門」，大神祐真
    - 無料のPDF版がWebで公開されている
  - 「コンピュータシステムの理論と実装 ― モダンなコンピュータの作り方」，Noam Nisan, Shimon Schocken 著，斎藤康毅 訳
  - 「プログラムはなぜ動くのか 第3版 ― 知っておきたいプログラミングの基礎知識」，矢沢久雄

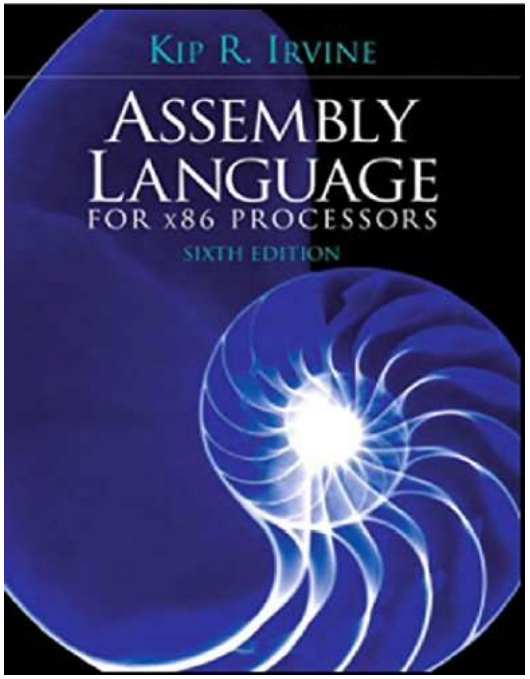


洋書ならこれ一択

- “Assembly Language for x86 Processors,” Kip R. Irvine, Pearson Education, Inc.
  - ハードカバー版，ペーパーバック版，Kindle 版あり
  - ただし Intel 記法（後述）

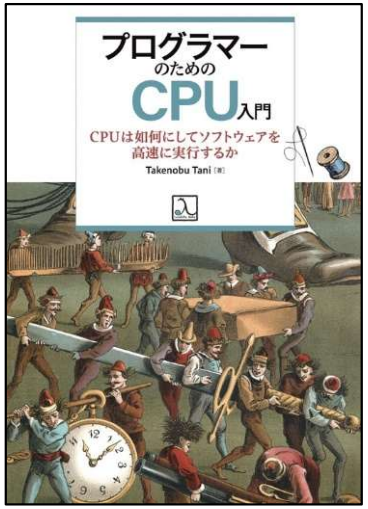






技を極めたい上級者へのお薦め本

- 「低レベルプログラミング」，Igor Zhirkov 著，吉川 邦夫 訳
- 「リバースエンジニアリングバイブル」，姜 秉卓 著，金 輝剛 監修，金 凡峻 訳
- 「プログラマーのためのCPU入門」，Takenobu Tani著
  - CPUそのものに興味を持った人向け
- 「コンピュータの構成と設計 第5版」，David A. Patterson and John L. Hennessy
  - 世界的名著
- レベルが非常に高い
- 全部理解できたら，すぐに開発現場でいい仕事ができる



書籍よりもWeb上の資料がお薦め

- すごい文書が無料で公開されている！
- 推薦する資料のリンク集をmanabaに載せてあります
  - どれもすさまじく情報量が多い珠玉の資料
  - どのサイトも一度は訪れる価値あり
    - どんな情報がどれくらいあるかを1分ざっと見るだけでよい
- 良さそうなサイトが無数にある
  - 書籍とほぼ同じコンテンツが無料で公開されていることもある
  - 使わない手はない
- 自分に合ったサイトを早期に見つけ，折に触れて参照しよう

コンピュータの仕組み

題材

- 対象CPU： Intel x86 (64 bit)
  - 情報科学類教育用計算機の端末やサーバのマシンのCPU
  - 「教育用のおもちゃ」ではなく，「本物」のCPUを体験するのに適している
    - 市場で最も成功したCPUの1つ
    - 現在も，至るところで使える
      - 大半のPCにはx86のCPUが入っている
      - 大半のクラウドサービスはx86のCPU上で稼働している
  - 初心者が学ぶのに適したCPUとは必ずしも言えない
    - 複雑で巨大な仕様
    - 歴史的経緯を引きずった仕様

x86 CPU

- Intel社が設計，開発，販売
- 現在，世界で最も普及しているCPU
- 最初のモデルが発売されたのは1978年
- 商品名としては，Core i9/i7/i5/i3，Celeron，Xeon，Pentium，Atomなど

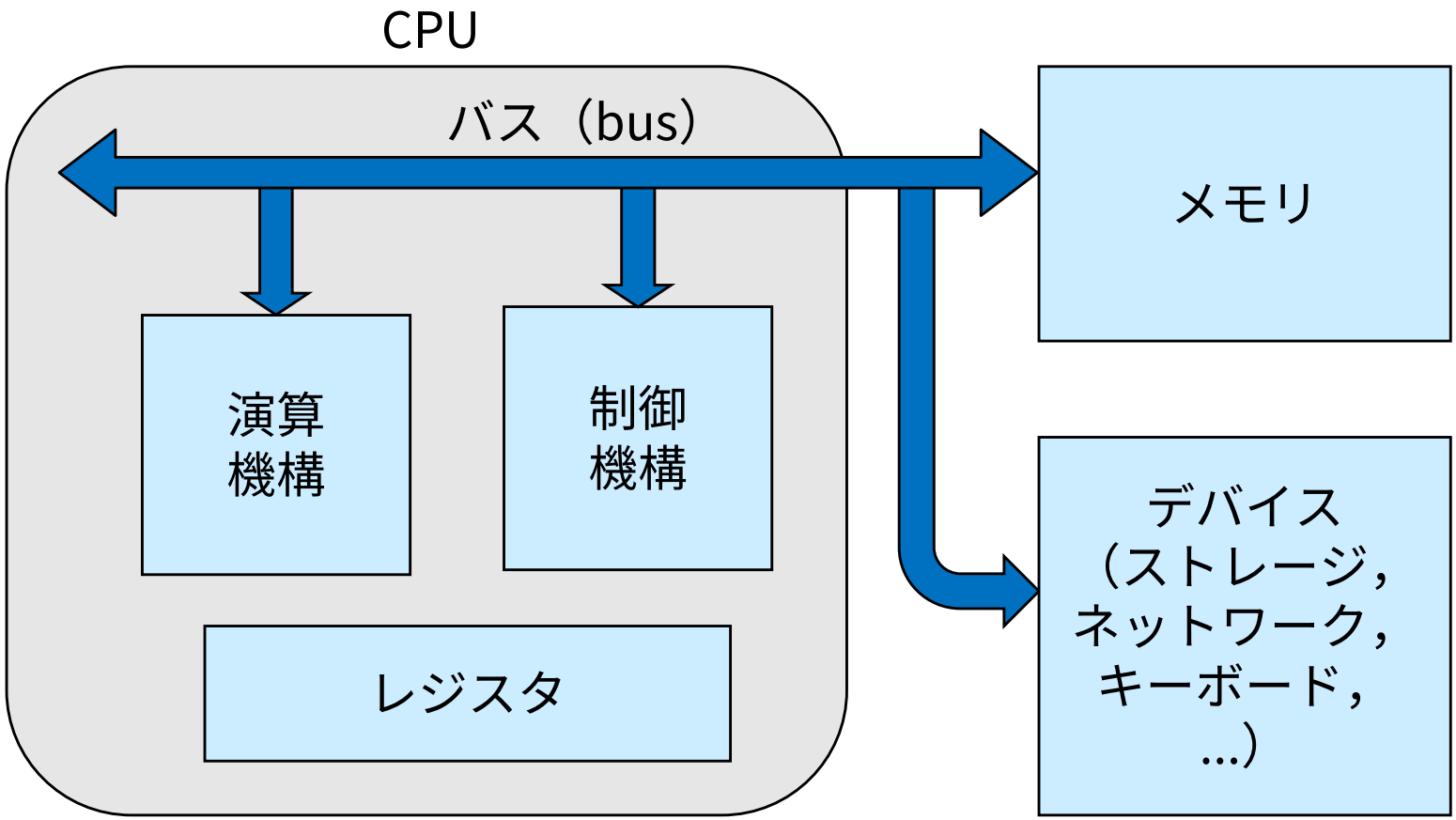
現実のCPUは複雑である

- x86は歴史的な経緯を引きずっており，非常に複雑な命令セットを持っている
  - この講義では，その中のほんの一部を使う
- 詳しく知りたい場合には，Intel® 64 and IA-32 Architectures Software Developer's Manual Vol. 1~3などを参照する
  - PDFファイルをダウンロード可能
  - 長大なので全ページの印刷は非現実的

コンピュータの基本的な構成

- コンピュータの最も基本的な要素はメモリとCPU
  - メモリ： プログラムやデータを格納する場所
  - CPU： メモリからプログラムやデータを読み出して プログラムを実行する
    - 指令する部分： プログラムを解釈する
    - 演算する部分： 足し算や掛け算をする
- プログラムとデータをメモリに入れて，CPUが メモリからプログラムを読み出して実行する
  - 命令の読み出し： フェッチ，命令の解読：デコード
  - 現在のコンピュータでの主流の方式
  - ノイマン型コンピュータなどと呼ばれる
    - フォン・ノイマン・ボトルネック： CPUとメモリとの間の 通信速度の遅さによりシステム全体の性能が抑えられること

コンピュータの基本的な構成



CPUの基本

- 命令やデータの通り道（実体は電線）がバス
- CPUの外にある，データを格納する記憶領域がメモリ
- CPUの中にある，一時的にデータを格納する記憶領域がレジスタ
  - CPUはメモリ上やレジスタ上にある値に対して様々な処理ができる
    - 読み出し，書き込み，算術演算，比較演算，...
  - 格納するデータの種類が固定されているレジスタもあれば，どんなデータを入れてもいいレジスタもある
    - 例： x86ではrbxレジスタには何を入れてもいい
    - 例： 現在実行中のプログラム（命令）の番地を保持しているレジスタがプログラムカウンタ（program counter, PC）
- メモリと比べてのレジスタの重要な特徴：
  - アクセスが速いこと
  - レジスタに対してのみ実行できる処理（命令）があること

整数の表現

整数の表現

- コンピュータの内部では数は0と1の列で表現されている
  - 整数は10進数でも2進数でも16進数でも表現できる
  - 整数を0と1で表現する方法（0と1の列に対応づける方法）は高校の教科「情報」で習ったはず
  - 大学の他の講義でも扱ったかもしれない
  - 正の整数の表現は（負の整数に比べれば）単純
  - 負の整数の表現は？
    → 次回で詳しくやります

2進数，16進数による整数の表現

- 2進数
  - 桁は2になったら繰り上がる
  - 各桁は下から順に$2^0, 2^1, 2^2, 2^3, 2^4, ...$を表す
- 16進数
  - 0〜9はそのまま0〜9に対応づけ，10〜15はa 〜 fに対応づける
  - 桁は16になったら（fを超えたら）繰り上がる
  - 各桁は下から順に$16^0, 16^1, 16^2, 16^3, 16^4, ...$を表す
  - 先頭に0xを付けることが多い

10進数	0	1	2	3	4	5	6	7
2進数	0	1	10	11	100	101	110	111
16進数	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7

10進数	8	9	10	11	12	13	14	15
2進数	1000	1001	1010	1011	1100	1101	1110	1111
16進数	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf

10進数	16	17	18	19	20	21	22	23
2進数	10000	10001	10010	10011	10100	10101	10110	10111
16進数	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17

2進数，16進数から10進数への変換の例

- （2進数の）1001100010101110
  ＝ $2^1+2^2+2^3+2^5+2^7+2^{11}+2^{12}+2^{15}$
  ＝ 2+4+8+32+128+2048+4096+32768
  ＝ 39086
- 0x78cd
  ＝ $13*16^0+12*16^1+8*16^2+7*16^3$
  ＝ 13*1+12*16+8*256+7*4096
  ＝ 30925

2のべき乗：2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, ...
16のべき乗：16, 256, 4096, 65536, 1048576, 16777216, 268435456, 4294967296, ...

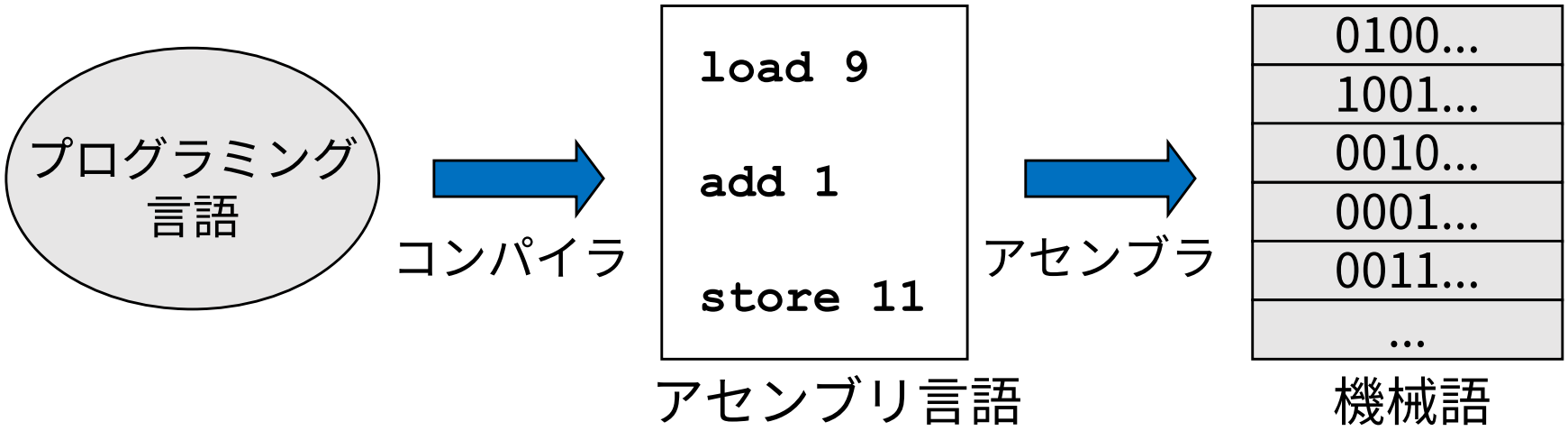
アセンブリ言語

アセンブリ言語

- CPUが理解（実行）できるプログラムは機械語（0と1の列）のもののみ
  - 人間が直接読み書きしてプログラミングするのはほぼ不可能
- 機械語に1対1対応させつつ，人間がなんとか理解できるような単語でプログラムを表すのがアセンブリ言語
  - 人間がアセンブリ言語で直接プログラミングするのは，かなり困難だが不可能ではない
  - アセンブリ言語のプログラムは機械語のプログラムと同一視できる（それらは1対1に対応している）

コンパイラとアセンブラ

- コンパイラは高級なプログラミング言語のプログラムを
  アセンブリ言語のプログラムに翻訳（変換）する
  - この操作が「コンパイル」
  - 例： C言語 → x86アセンブリ言語，Rust言語 → Armアセンブリ言語
- アセンブラはアセンブリ言語のプログラムを機械語のプログラム
  に翻訳（変換）する
  - この操作が「アセンブル」
  - x86での例：“**`cmpl $0x1,-0x4(%rbp)`**” → 83 7d fc 01



変換例

C言語のプログラム

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

x86 (64 bit)
アセンブリ言語の
プログラム

```
000000000040052d <fact>:
40052d: push   %rbp
40052e: mov    %rsp,%rbp
400531: sub    $0x10,%rsp
400535: mov    %edi,-0x4(%rbp)
400538: cmpl   $0x1,-0x4(%rbp)
40053c: jne    400545
40053e: mov    $0x1,%eax
400543: jmp    400556
400545: mov    -0x4(%rbp),%eax
400548: sub    $0x1,%eax
40054b: mov    %eax,%edi
40054d: callq  40052d <fact>
400552: imul   -0x4(%rbp),%eax
400556: leaveq
400557: retq
```

機械語のプログラム

```
000000040052d <fact>:
40052d: 55
40052e: 48 89 e5
400531: 48 83 ec 10
400535: 89 7d fc
400538: 83 7d fc 01
40053c: 75 07
40053e: b8 01 00 00 00
400543: eb 11
400545: 8b 45 fc
400548: 83 e8 01
40054b: 89 c7
40054d: e8 db ff ff ff
400552: 0f af 45 fc
400556: c9
400557: c3
```

x86 (64 bit) の特徴

- 基本的には，整数やアドレス（今後説明する）などのデータは64ビットで表される
- 互換性などのために，データを32ビットで表す32ビット用プログラムも実行できる
- さらに，64ビット用プログラムでも，あえてデータを8, 16, 32ビット単位で操作することもできる
- 様々な種類のレジスタがある
  - 汎用レジスタ
  - プログラムカウンタ（レジスタ）
  - フラグレジスタ
  - スタックポインタ（レジスタ），ベースポインタ（レジスタ）
  - 浮動小数点レジスタ

x86 (64 bit)アーキテクチャ

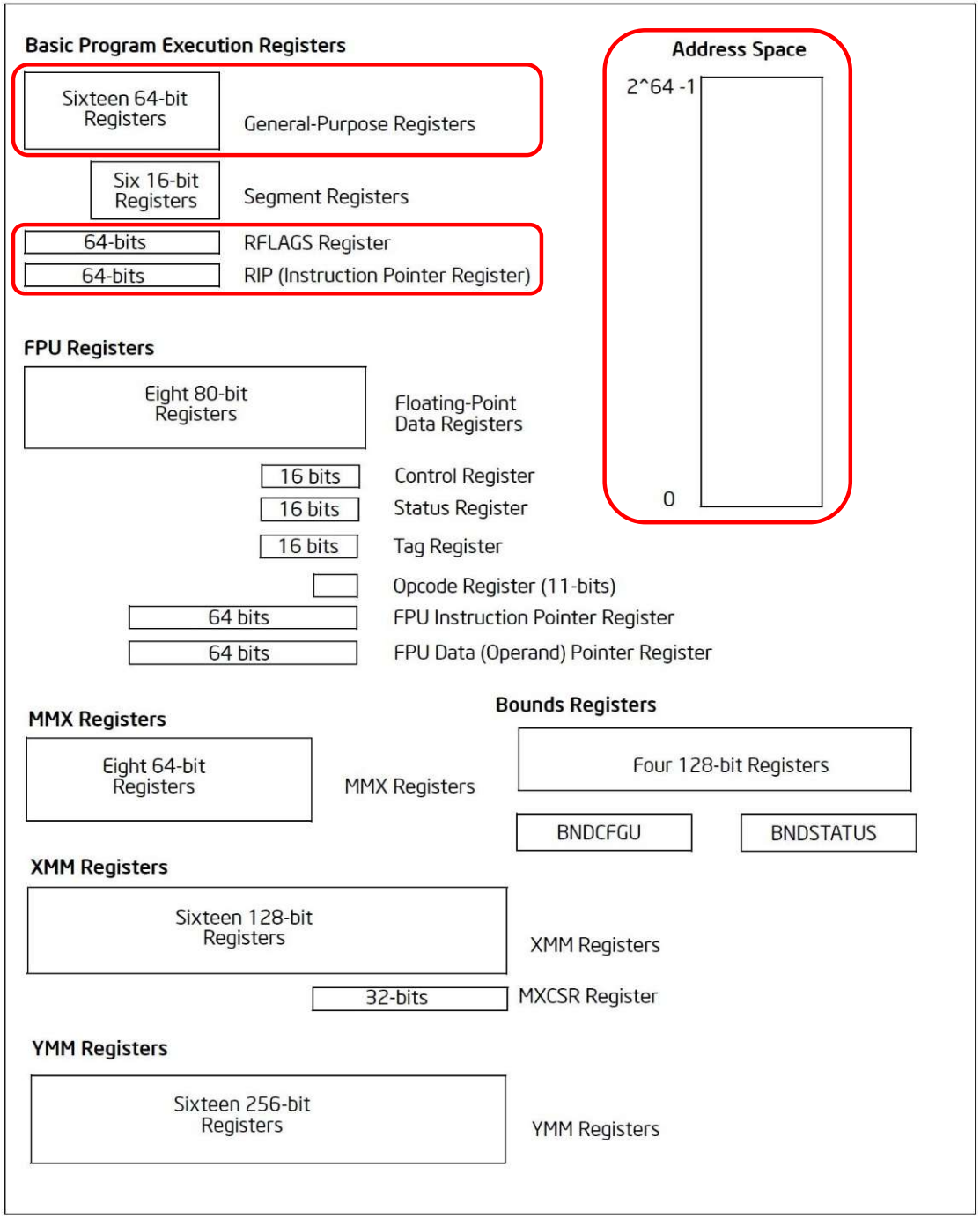


Figure 3-2. 64-Bit Mode Execution Environment

Intel® 64 and IA-32 Architectures Software Developer's Manual

Volume 1: Basic Architecture

レジスタ（1）

- 汎用レジスタ
  - どんなデータを入れるかが決まっていない，どんなデータを入れてもいいレジスタ
    - rax（語源はaccumulator）
    - rbx（語源はbase）
    - rcx（語源はcounter）
    - rdx（語源はdata）
    - rsi（語源はsource index）
    - rdi（語源はdestination index）
    - r8, r9, ..., r15
    - (rsp, rbp)

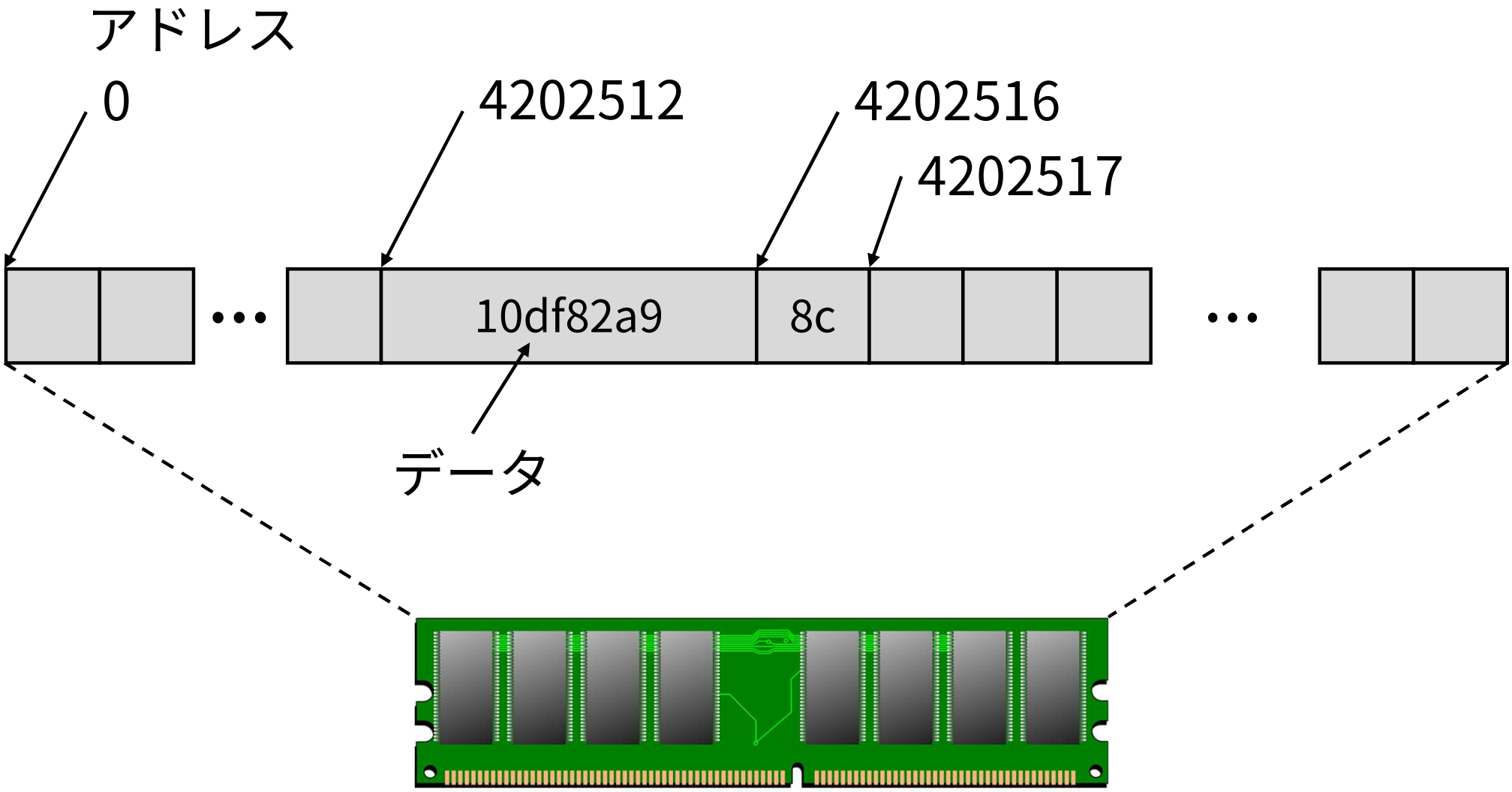
レジスタ（2）

- スタック（今後説明）の場所を示す値を入れるためのレジスタ
  - rsp（語源はstack pointer）
  - rbp（語源はbase pointer）
  - 他のデータを入れてもいいが，普通はそんなことはしない
- フラグ（過去の計算結果の一部など）を示すレジスタ
  - rflags
- 次に実行される命令のメモリアドレスを保持しているレジスタ
  - rip（語源は instruction pointer）

メモリとアドレス

- メモリはデータを記憶するためのデバイス
- アドレスは，CPUなどがメモリをアクセスするときに指定する，メモリ上の場所
  - メモリはデータを入れる箱が並んだものであり，アドレスは箱に付けられた番号
  - メモリは1バイトごとに区切られているので，何バイト目をアクセスするかをアドレスで指定
    - たとえば32 GBのメモリなら，0～34359738367（2の35乗マイナス1）のどこなのかを指定

イメージ



アセンブリ言語の命令

- アセンブリ言語のプログラムは命令の列である
- 各命令は，動作の情報と対象の情報からなる
  - 動作を指定する命令コードの部分がオプコード（opcode）
    - オペコードやニーモニックと呼ぶこともある
  - 対象を指定する引数データの部分がオペランド（operand）
  - オペランドのない命令もある

アセンブリ言語の記法

- 基本的には以下の形式

  オプコード オペランド1, オペランド2, オペランド3, ...

  オペランドの数や種類は命令によって異なる

- x86のアセンブリ記法にはAT&T記法とIntel記法があるが，この科目ではAT&T記法を使う
  - 情報科学類のLinuxではgas (gcc)というアセンブラ，gdbというデバッガ，objdumpという逆アセンブラを使えるが，どのツールも，デフォルトはAT&T記法

オペランドの種類と記法

- オペランドに指定できるもの
  - レジスタ： rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp
    - レジスタであることを示すために頭に % を付ける
    - rspとrbpは用途が決まっている（ので，普通，計算には使わない）
  - 定数（即値とも言う）
    - 定数であることを示すために頭に $ を付ける
    - $ の後ろにラベルが来ることもある
  - メモリ（アドレス）
    - $ を付けない数値
    - $ を付けないラベル
    - d(b, I, s)という形をした式
      - 各要素は省略可
      - 次回詳しくやります

```
#include <stdio.h>

int x = 100;

int main(void)
{
  printf("x + 333 = %d¥n",
         x + 333);
  return 0;
}
```

```
000000000040052d <main>:
40052d: 48 83 ec 08        sub $0x8,%rsp
400531: 8b 05 fd 0a 20 00  mov x,%eax
400537: 8d b0 4d 01 00 00  lea 0x14d(%rax),%esi
40053d: bf f0 05 40 00     mov $0x4005f0,%edi
400542: b8 00 00 00 00     mov $0x0,%eax
400547: e8 c4 fe ff ff     callq 400410 # printf
40054c: b8 00 00 00 00     mov $0x0,%eax
400551: 48 83 c4 08        add $0x8,%rsp
400555: c3                 retq
```

基本的な命令

mov命令

- 最も基本的で，最もよく使う命令

  mov src, dst

  srcからdstに値をコピーする

```
mov $1, %rax    # レジスタraxに1をセット
mov %rax, %rdi  # レジスタraxの内容をrdiにセット
mov 100, %rbx   # アドレス100の内容をレジスタrbxにセット
mov %rdx, 200   # レジスタrdxの内容をアドレス200に書き込み
mov N, %rcx     # ラベルNに置かれたデータをレジスタrcxにセット
mov $N, %rsi    # ラベルNの値（アドレス）をレジスタrsiにセット
```

しばしば，レジスタに値をセットする操作はロード，
メモリへ値を書き込む操作はストアとも呼ばれる

add命令，sub命令

- 整数の加算，減算を行う命令

```
add src, dst # dst = dst + src
sub src, dst # dst = dst - src
```

オペランドの数は3ではなく2であることに注意

```
add $1, %rax    # 1をレジスタraxに加算する
sub 100, %rdx   # アドレス100にある内容をレジスタrdxから引く
add %rdx, %rbx  # レジスタrdxの内容をレジスタrbxに加える
sub %rax, x     # レジスタraxの内容をラベル x に置かれたデータから引く
```

無条件分岐

- **`jmp` 命令**
  - **`jmp L`**
    - ラベル **`L`** にジャンプする
    - あらかじめ，プログラム中のジャンプ先の場所にラベル **`L`** を書いておく
  - **`jmp`** *addr*（**`$`** を伴わない数値）
    - アドレス *addr* にジャンプする
    - あまり使わない
  - **`jmp *%`**レジスタ
    - 指定したレジスタの値をジャンプ先アドレスとしてジャンプする
    - あまり使わない

簡単な分岐

- 他の言語のif文に相当する分岐処理は，**cmp** 命令と条件分岐命令（**je** 命令など）を組み合わせると書ける

```
cmp   opd1, opd2  # 減算数が左，被減算数と減算結果が右
                  #  （AT&T記法では）
je    L           # L はラベル
```

  - **je** 命令（jump if equal命令）：前の **cmp** 命令で比較した結果が「同じ」であれば，オペランドのアドレスにジャンプ

```
cmp   $0, %rax
je    L
```

rax が0だったらラベル **L** にジャンプする

  - **jne** 命令（jump if not equal命令）：前の **cmp** 命令で比較した結果が「違う」であれば，オペランドのアドレスにジャンプ

cmp 命令と条件分岐命令については，次回で詳しくやります

プログラムの書き方

アセンブラ指示 (Assembler Directives)

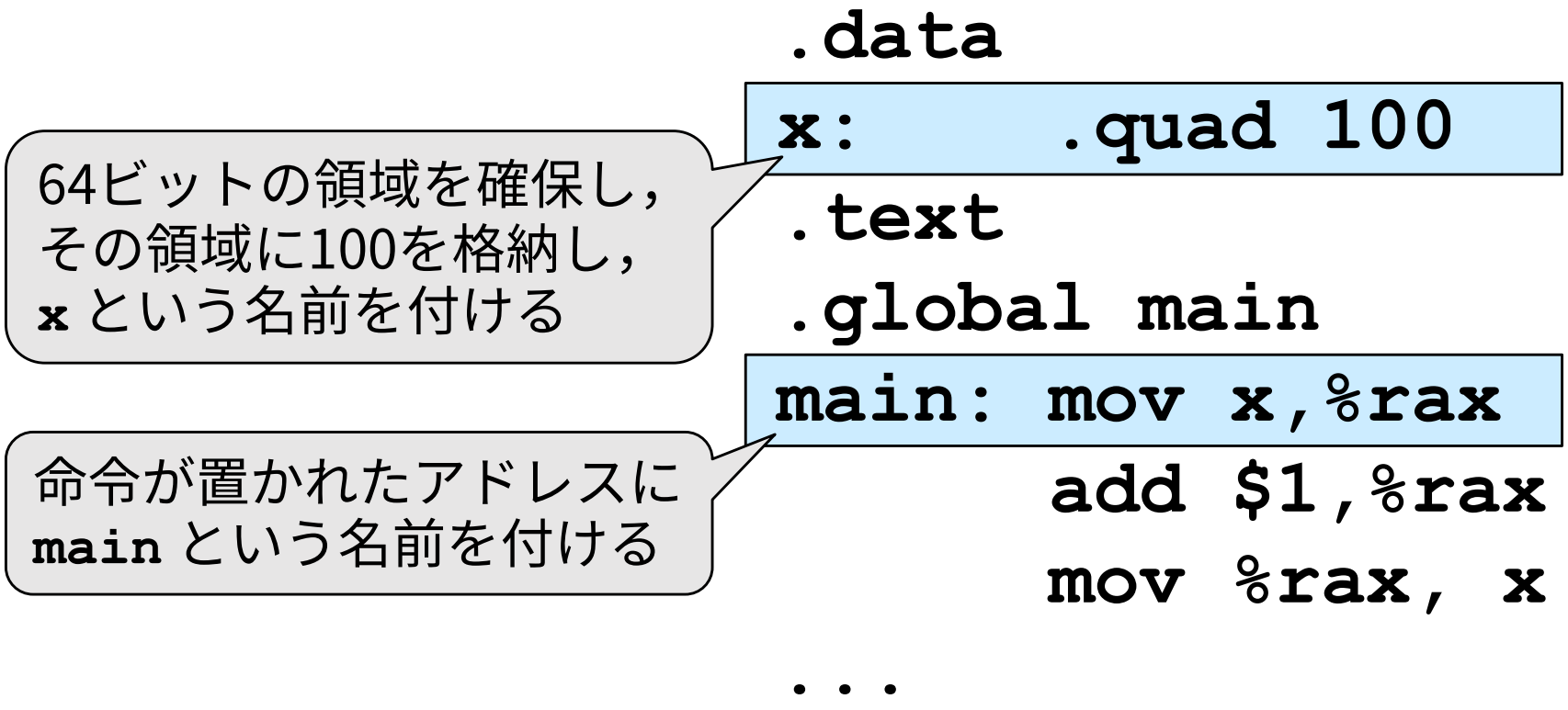
- アセンブラに伝える指示の情報
  - ラベル（名前の後に：をつけたもの）：変数や命令の位置（アドレス）を指定するための名前を伝える
  - **`.text`**：その下の部分がプログラムのコードだと指示する
  - **`.data`**：その下の部分がプログラムのデータだと指示する
  - **`.global`** *label*（または **`.globl`** *label*）：*label* を大域的なラベルであると指示し，外部プログラムから参照可能にする
  - **`.byte`** $n_1, n_2, \ldots$：$n_1, n_2, \ldots$という8ビット整数の並びが格納された領域を確保する
  - **`.quad`** $n_1, n_2, \ldots$：$n_1, n_2, \ldots$という64ビット整数の並びが格納された領域を確保する

これはあくまでGNU Assembler (gas) に関する記述であり，他のアセンブラでは仕様が異なることに注意

他のアセンブラ指示

- **`.short`** *n*, ...: 16ビット整数
- **`.word`** *n*, ...: 16ビット整数（**`.short`** と同じ）
- **`.long`** *n*, ...: 32ビット整数
- **`.int`** *n*, ...: 32ビット整数（**`.long`** と同じ）
- **`.float`** *n*, ...: 32ビット浮動小数点数
- **`.double`** *n*, ...: 64ビット浮動小数点数
- **`.string`** *str*, ...: 文字列
- **`.fill`** *repeat*, *size*, *value*: サイズ *size* バイトである *value* という値の *repeat* 個の並び
- **`.space`** *size*, *fill*: *fill* という1バイト値の *size* バイトの並び
- **`/*`** ... **`*/`**： コメントであり，無視される
- **`#`** から改行まで： コメントであり，無視される

プログラム例


.data
x: .quad 100
.text
.global main
main: mov x,%rax
add $1,%rax
mov %rax, x
...
64ビットの領域を確保し，
その領域に100を格納し，
x という名前を付ける
命令が置かれたアドレスに
main という名前を付ける

プログラム例

データ領域であることを
指示し，これにより，ここは
読み書き可能領域となる

プログラム領域であることを
指示し，これにより，ここは
読み出しと実行ができて
書き込みができない領域となる

```
.data
x:      .quad 100
.text
.global main
main:  mov x,%rax
       add $1,%rax
       mov %rax, x
...
```

よくある間違い

- データに **.data** の指定がない

```
.text
x: .quad 1,1,0,0,0,0,0,0,0,0,0
.global main
main:
    mov x, %rdx
    …
    mov %rax, x
    …
```

  - 実行するとプログラムが強制終了する
    - 理由を細かく説明すると，こう書くとデータ **x** が読み出し専用領域に置かれ，そこへの書き込みでsegmentation faultエラーが発生するため

プログラム例（加減算）

- raxレジスタの初期値を5，rbxレジスタの初期値を3として，それらのレジスタの値の和と差をそれぞれrcx，rdxレジスタにセットする

```
mov $5, %rax
mov $3, %rbx
mov %rax, %rcx
add %rbx, %rcx
mov %rax, %rdx
sub %rbx, %rdx
```

プログラム例（ループ処理）

- raxレジスタの初期値を0として，raxに1ずつ加えながら，rax が10になるまでループする

```
     mov $0, %rax
L1:  cmp $10, %rax   # rax と10を比較
     je  L2          # 等しければ L2 へジャンプ
     …               # ループ本体
     add $1, %rax
     jmp L1          # L1 へ無条件ジャンプ
L2:  …
```

Intel x86アーキテクチャでのメモリオペランドに関する注意

- 第1オペランド（src），第2オペランド（dst）には，基本的には何を書いてもいいが，あまり本質的ではない以下の制限がある

  srcとdstの両方がアドレス（メモリ参照）であってはならない

- 例えば mov 10000, 20000 はエラー
  - 「アドレス10000にあるデータを読み込んでアドレス20000に書き込む」のつもりでこれを書き，アセンブルしようとしても，エラーになる
- オペランドを2つ（以上）取る他の命令でも同様

コマンドライン（ターミナル）でOSとCPUを調べる方法

- **`uname -a`**
- **`cat /proc/cpuinfo`**
- **`lscpu`**
- **`arch`**
- **`dmesg`**
- **`cat /etc/os-release`**
- **`lsb_release -a`**　（Ubuntuの場合）
- **`cat /etc/redhat-release`**　（Red Hat Linuxの場合）

などのコマンドを実行

注：環境によっては，上記コマンドの実行が失敗することもある

演習

まず，アセンブルと実行ができるかどうかを確認

- manabaのコースページにあるサンプルプログラムsample.sをファイルに保存し，中身をテキストエディタなどで見てみる
- サーバ上で起動したターミナル（端末）の中で，そのファイルをgccコマンドでアセンブルする
  - $ gcc sample.s
- 実行型ファイル a.out ができるので，実行する
  - $ ./a.out
  - raxレジスタに3，rbxレジスタに2をセットし，それらの和，差，積をrcx，rdx，rsiレジスタにセットして終了する
  - 右のような表示が出ていればOK
- うまくいかない場合には，状況を教員やTAに連絡する

```
$ gcc sample.s
$ ./a.out
rax=0x0000000000000003 (3)
rbx=0x0000000000000002 (2)
rcx=0x0000000000000005 (5)
rdx=0x0000000000000001 (1)
rsi=0x0000000000000006 (6)
rdi=0x0000000000000001 (1)
…
$
```

課題プログラムの作成 (1)

- 作成するアセンブリプログラムは **.s** で終わるファイル名にする
  - **ex2-1.s** など
- 実行は **main** というラベルが付いた命令から始まるので，それを意識してプログラムを書く
- プログラムの最後で **call finish** という命令を実行する
  - **call** は関数（サブルーチン）を呼び出す命令であり，**finish** は全レジスタの値を表示してプログラムを終了する関数である

ex2-1.s

```
.text
.global main
main: /* ここからプログラムを書く */
      ...
      ...
      call finish # ここでプログラム終了
```

課題プログラムの作成 (2)

- 手順
  - manabaのコースページから **`lib.s`** を入手
    - このファイルには **`finish`** 関数などが入っている
  - **`gcc`**コマンドでアセンブル，リンク
    - **`gcc lib.s ex2-1.s`**
    - リンクとは，1つ以上の機械語プログラムを，必要なライブラリなどと結合させて，実行可能ファイルを作ること
  - できた実行可能ファイルを実行
    - **`./a.out`**
  - 意図通りのレジスタの内容を表示して終了することを確認

再配置（relocation）や位置独立実行形式（PIE）に関するエラー

- コンパイル（アセンブル）しようとしても，gccがrelocationやPIEについてのエラーを出力して失敗することがある
  - 典型的には，最近のバージョンのgcc（azalea系マシン上のgccなど）を使った場合
  - 別のメモリアドレスに再配置できないコードを無くしていくというコンパイラ開発者の方針による
- 解決のために有効かもしれない方策（うまくいく保証はない）
  - gccに -no-pie オプションを与える
  - ラベル（アドレス）そのものを参照するmov命令の書き換え：ラベル前の **`$`** を削り，後ろに **`(%rip)`** を付け，mov命令をlea命令に変える
    - 例：**`mov $N, %rax` → `lea N(%rip), %rax`**
  - ラベルの場所にあるメモリ内容を参照するmov命令の書き換え：ラベル直後に **`(%rip)`** を付ける
    - 例：**`mov N, %rax` → `mov N(%rip), %rax`**

Segmentation Fault

- 自分が作ったプログラムを実行すると，しばしば，

  `Segmentation fault (core dumped)`

  というメッセージが出て，途中で実行が終了するはず
  このメッセージの意味は何か？
  → 「不正なメモリアドレスをアクセスした」
- 次にどうすればいいのか？
  → 与えたアドレスが正しかったかどうかを見直す
  - 例えば，エラーが出ていると予想される行の直前でレジスタの値を表示し，与えているアドレスの正しさを確認する

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というプログラムが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

```
$ 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: 000000000004004fd   EFLAGS: 00010246
...
```

エラーメッセージそのものを検索しよう，AIに相談しよう

- gccがエラーを出力したら，まずそのエラーメッセージをよく読んで，その意味を考えよう
  - プログラムのどこにどういう不備があるかについて，重要な手がかりを伝えてくれていることが多い
  - 英語がわからなければ自動翻訳にかけよう
- 意味がわからなければ，メッセージそのものでググったり，AIに相談したりしよう
  - 同じエラーで悩んだ人は世界におそらく数万人いて，Webの掲示板で似た質問と回答が数百回はやりとりされたはず
  - 生成AIがいい感じで状況や解決法を教えてくれるかもしれない

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

レポート作成における注意事項（1）

- 課題のプログラムの標準動作環境はCOINSシステムのLinuxサーバとします
  - 別の環境で演習，動作確認をしてもいいですが，レポートの評価ではあくまで，標準動作環境上で正しく動くプログラムや，標準動作環境上での動作を正解とします
- レポートに書く実行結果には，アセンブルを行う部分や，できたプログラムを実行する部分も，できる限り含めて下さい
  - 特に，gccにオプションを与えた場合には，必ず書いて下さい
- 指定の書式に沿ったレポートを書いて下さい

レポート作成における注意事項（2）

- 課題で求められている出力を，課題で求められている処理によらずに出力するプログラムは0点です
  - 課題で求められている計算をせずに，最終計算結果をいきなりレジスタに書き込んで終了するプログラムなど
  - 不正行為とみなして，相応の対処をすることがあります
- メ切厳守： manabaの提出用ページが閉じたら終わり
  - 提出に失敗したら，最初から何もしなかったのと全く同じ
  - リスク回避と安心のために，途中状態のレポートでいいのでメ切前に何か提出しておき，後から差し替えるとよい
  - メールでレポートを送ってきても，採点しません