



# アセンブリ言語

## イントロダクション (2)

情報工学系  
権藤克彦



# 機械語命令, アセンブラ命令, ラベル, コメント

- 機械語命令
- アセンブラ命令
- ラベル
- コメント

アセンブリコードは  
この4つで書く.  
(基本的には行単位)

アセンブラ命令

ラベル

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

機械語命令

#リターン命令

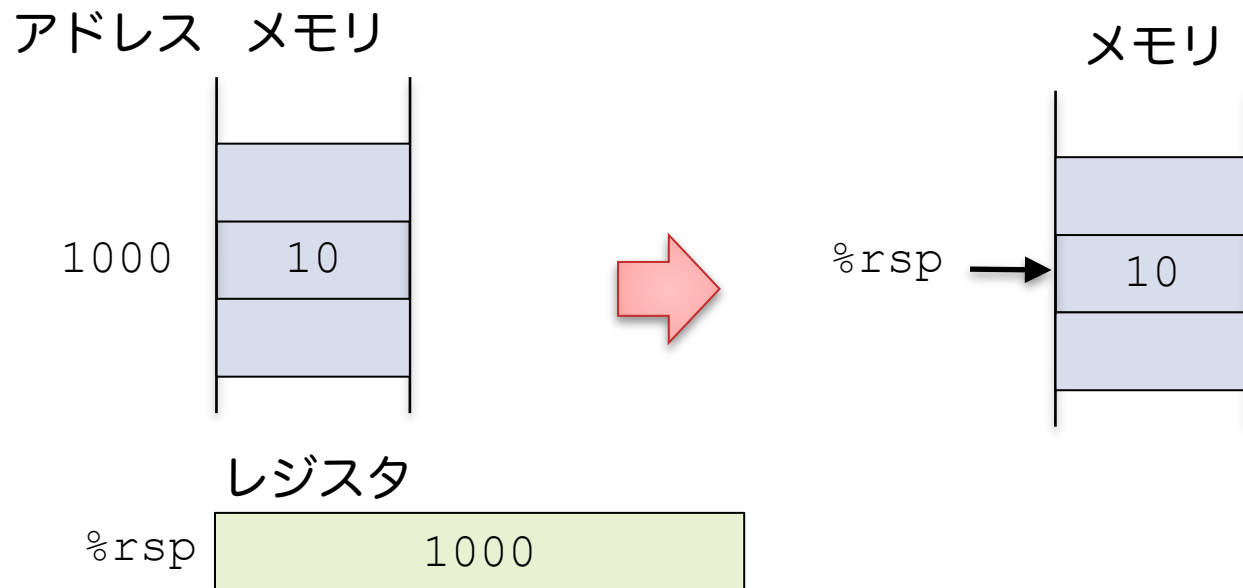
コメント

インデントは見やすさのためだけ



# ポインタの図表現（１）

- レジスタ`%rsp`はメモリのアドレス1000番地を値として保持。
  - つまり`%rsp`は1000番地を「指している」.
  - これを矢印で右図のように図表現する.

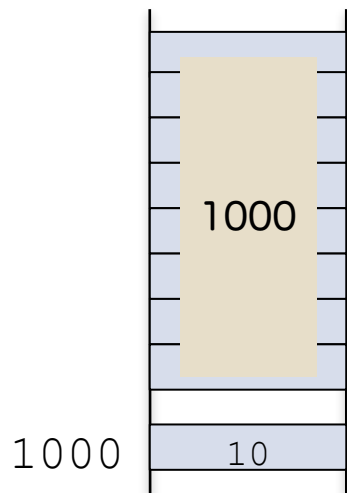




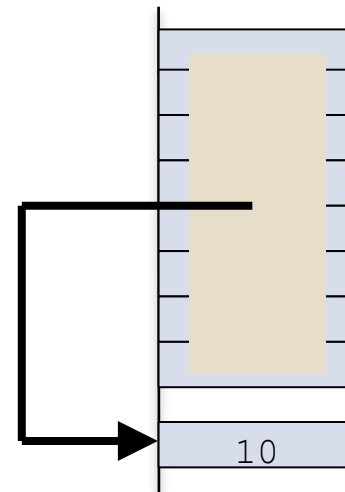
## ポインタの図表現（２）

- メモリがメモリを指す場合も同様に矢印で図表現。
  - x86-64のアドレス長は64ビット（=8バイト）

アドレス メモリ



メモリ

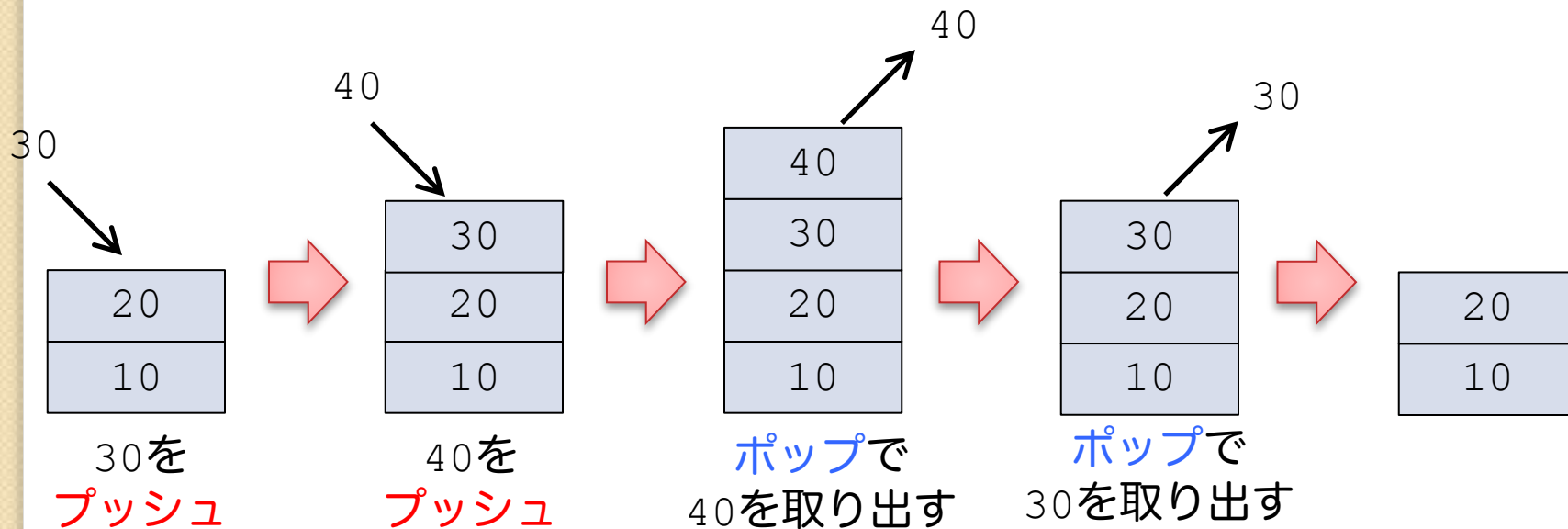


複数バイトのデータは複数のアドレスにまたがる  
→ 先頭アドレス（一番小さいアドレス）を代表アドレスに



# スタック (stack) (1)

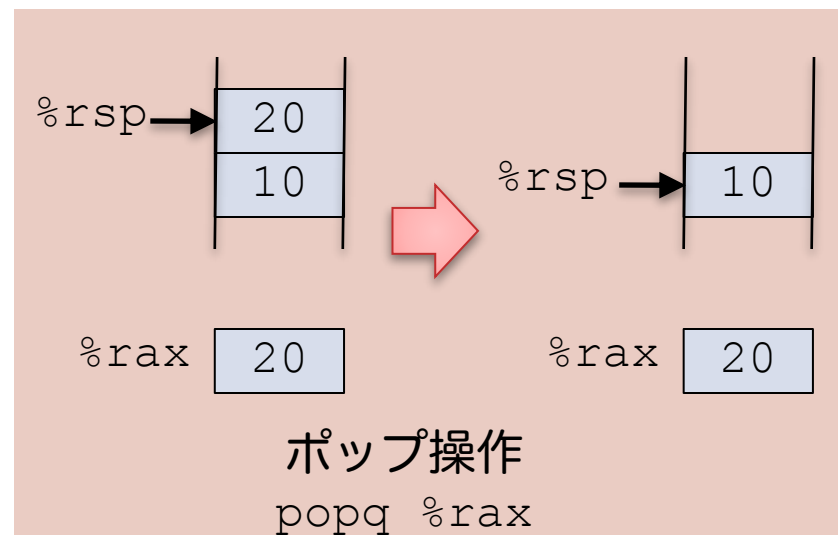
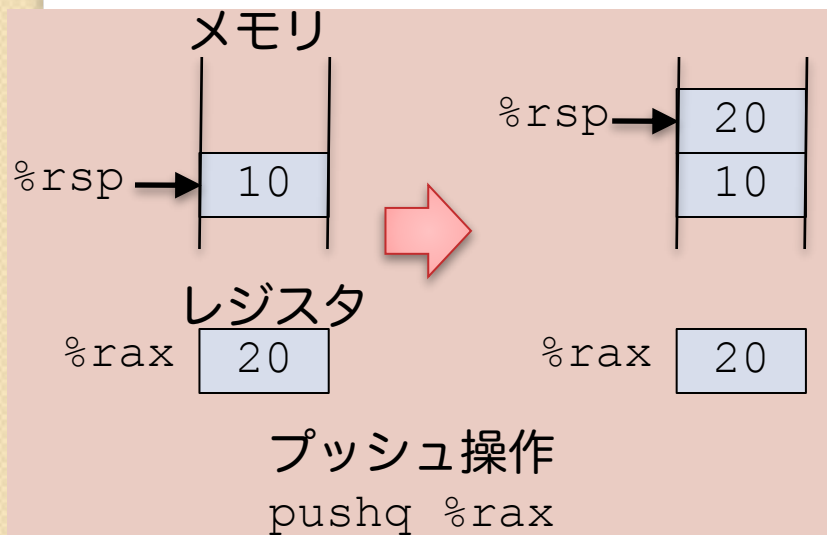
- LIFO (last-in first-out) のデータ構造.
  - 後に入れたデータが, 先に取り出される. cf. キュー(queue)
- プッシュ操作とポップ操作がある.
  - **プッシュ**(push)=スタックトップにデータを積む.
  - **ポップ**(pop)=スタックトップからデータを取り出す.





## スタック (stack) (2)

- 言語処理系では関数（手続き）呼び出しの実装に利用
- スタックフレーム(stack frame)
  - 関数呼び出し1回分のデータ.
  - 局所変数, 引数, 戻り番地, 戻り値などを含む.
- `%rsp`はスタックトップを指す（ようにする）.
  - `%rsp` は「スタックポインタ」と呼ばれる.





# 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
    popq %rbp
    retq
```

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 (アドレスはアセンブラが自動計算) .

%rbp の値をスタック上に退避.

%rbp = %rsp

%rdi += 5

%rax = %rdi

popq %rip (リターン)

} 新しいスタック  
フレームを作成

} スタックフレーム  
を破棄.

この命令は実在しない

- ドット記号 (.) で始まるのはアセンブラ命令.
- 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.

手続き呼び出し規約 (一部)

- 第1引数は %rdi で渡す
- 返り値は %rax で返す



# コメント（１）

- プログラムのメモ書き．アセンブラは単に無視する．
- 2種類のコメントを使える．
  - 行コメント：
    - x86-64の場合はシャープ記号（＃）から行末までがコメント．  
ちなみにSPARCではビックリ（！），H8ではセミコロン（；）．

# これは行コメントです．

- ブロックコメント：
  - C言語のブロックコメントと同じ． /\* から \*/ までがコメント．  
ネスト（入れ子）禁止．

/\* これはブロックコメントです．  
複数行でもOKです．

\*/





## コメント (2)

- 拡張子を.S (大文字) にする→C前処理命令を使える.
  - #if を使って, 入れ子可能なコメントを書ける.

```
#if 0  
これはC前処理命令を使った  
コメントです.  
#endif
```

#define や #include も使用可.



# 機械語命令 (machine instruction)

- CPUが実行する命令. マシン語ともいう.
  - cf. アセンブラ命令はアセンブラが実行する.
  - CPUが直接実行できるのは (2進数の) 機械語だけ.
- 例 : `movq %rsp, %rbp`
  - 「%rspレジスタ中の値を%rbpレジスタにコピーせよ」
  - ただし, これは記号表現 (ニモニック). このままではCPUは実行できない.
  - この命令の2進数表現は `01001000 10001001 11100101`
  - アセンブラがニモニックを2進数 (バイナリ) に変換する.



# ニモニツク (mnemonic)

- 2進数の機械語命令の記号表現.
  - 例：`movq %rsp, %rbp`
- 英語の省略. 人間が覚えやすいように.
  - 例：`movq` = `move quad`
- 機械語命令とニモニツクはほぼ一対一に対応.

mnemonic

- 形：記憶を助ける, 記憶術の
- 名：記憶を助ける工夫 (公式・覚え歌など)

x86-64では

word = 2バイト

double word (long) = 4バイト

quad word = 8バイト

LP64メモリモデルでは  
C言語のlong型は8バイト長



# オペコードとオペランド

- オペコード (opcode)
  - 機械語命令のうち、処理の内容の部分.
    - 例：movq データのコピー
  - operation code の略.
- オペランド (operand)
  - 機械語命令のうち、処理の対象の部分.
    - 例：%rsp レジスタ
  - 便宜上、左から「第1オペランド」「第2オペランド」と呼ぶ.





# AT&T形式とIntel形式

- x86-64用アセンブラは大きく2種類.
  - GNUアセンブラ（のデフォルト）はAT&T形式を使用.
  - 他のアセンブラ（NASM, MASM, TASMなど）やIntelのドキュメントではIntel形式を使用.

## AT&T形式

```
pushq %rbp
movq  %rsp, %rbp
subq  $0x8, %rsp
```

代入は左から右

```
% objdump -d -M att -M suffix add5.o
```

## Intel形式

```
push    rbp
mov     rbp, rsp
sub     rsp, 0x8
```

代入は右から左

```
% objdump -d -M intel add5.o
```

macOSでは, -x86-asm-syntax=intel



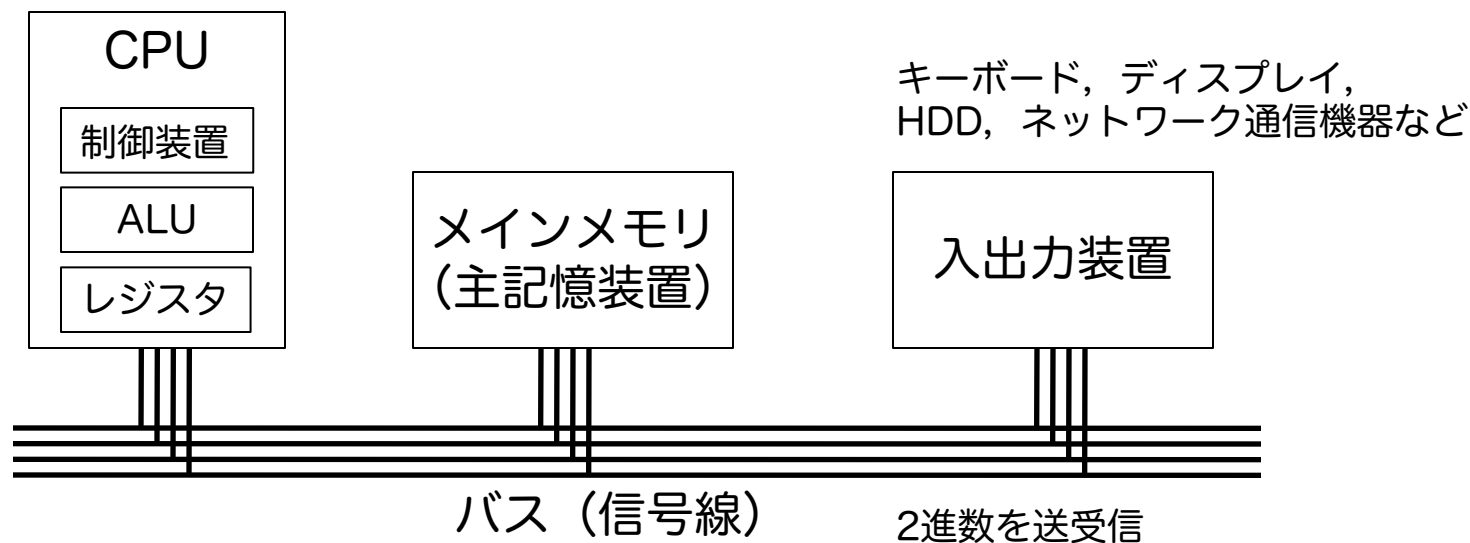
# 機械語命令の主な種類

- 演算命令 = データを計算
  - add, sub, imul, idiv, neg, cmp, cltd, movsbq, movzbq
- 制御命令 = 実行順序の変更
  - jmp, jcc (条件付きジャンプ) , call, ret
- 転送命令 = データのコピー
  - mov, push, pop, lea, setc
- その他
  - 特権命令, 入出力命令 (in, out)



# コンピュータの基本構造

- **CPU**（中央処理装置，プロセッサ）
  - 制御装置，ALU（演算装置），レジスタから構成.
- **メインメモリ**（主記憶） 以後，メモリと略す.
- 入出力装置
- バス(bus)





# CPUの基本構造

- 制御装置

- フェッチ実行サイクルをひたすら繰り返す.

- ALU

- 四則演算や論理演算などを計算（例：addq）

- レジスタ

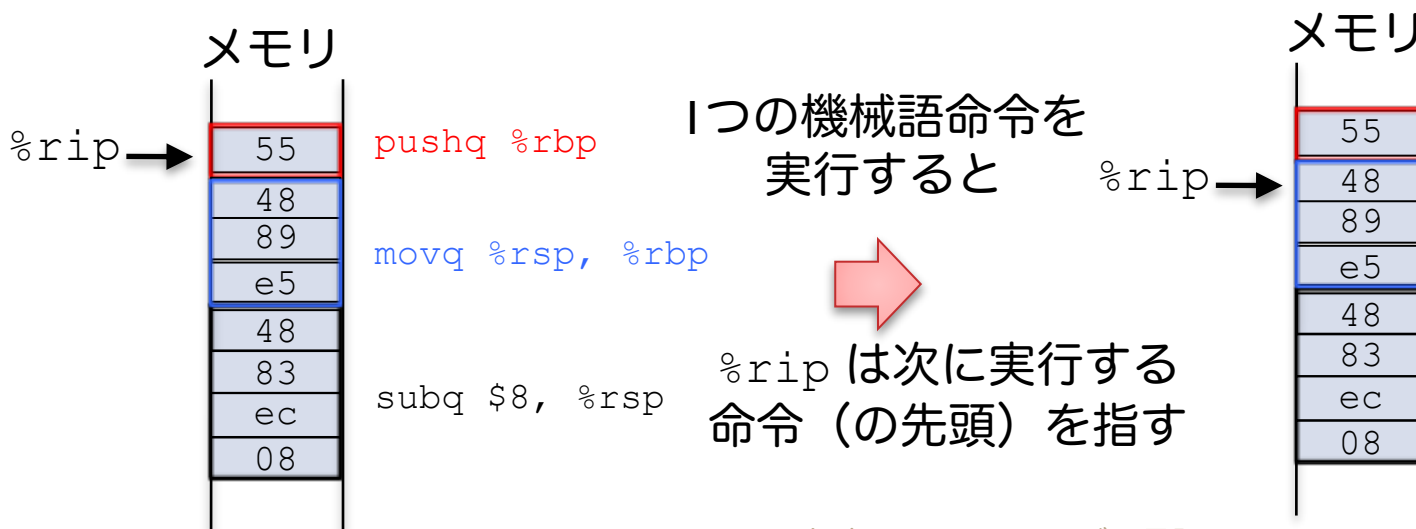
- 高速で小容量・固定長のメモリ.
- 専用レジスタ＝特定の役割を持つレジスタ.
  - 例：プログラムカウンタ（%rip）.
- 汎用レジスタ＝様々な用途に使えるレジスタ.
  - 例：%rax, %rbx





# プログラムカウンタ

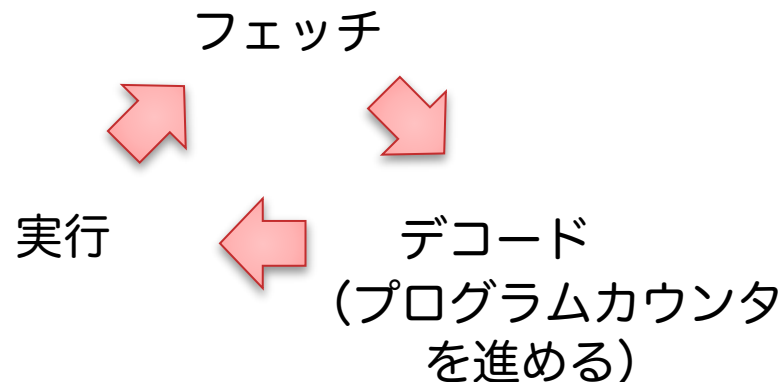
- 「次に実行する機械語命令を格納するメモリのアドレス」を保持.
- x86-64では **%rip** レジスタ.
  - movq命令などは, %ripレジスタにアクセス不可.
    - movq %rip, %rax とは書けない. movq 0(%rip), %rax とは書ける.
  - jmp命令やcall命令などで, 間接的に%rip にアクセス.
    - %rip を書き換える=実行をジャンプする.





# フェッチ実行サイクル (fetch-execute cycle)

- CPUは次の動作をひたすら繰り返す。
  - **フェッチ** (fetch)
    - ・ プログラムカウンタが指す機械語命令をメモリからCPUに読み込む。
  - **デコード** (decode)
    - ・ 読み込んだ命令を解析して、実行の準備をする。
    - ・ 次の機械語命令を指すようにプログラムカウンタの値を増やす。
  - **実行** (execute)
    - ・ 読み込んだ機械語命令を実行する。





# アセンブラ命令 (assembler directive)

- アセンブラ命令（例：`.text`）はアセンブラが実行。
  - cf. 機械語命令（例：`movq`）はCPUが実行。
  - CPUが実行しないので、疑似命令とも呼ばれる。
- すべて**ドット記号（.）**で始まる。
  - GNUアセンブラの場合。

pseudo instruction,  
pseudo opcode

## アセンブラ命令と機械語命令の違い

	例	何が実行	いつ実行	バイナリ ファイル中に
アセンブラ 命令	<code>.text</code>	アセンブラ	アセンブル時	ない
機械語命令	<code>movq %rsp, %rbp</code>	CPU	実行時	ある



# アセンブラの主な仕事（１）

- 機械語命令の**二モニック**を2進数表現に変換.
  - アセンブラにとって機械語命令は処理対象のデータ.

```
movq %rsp, %rbp
```



48
89
e5

- 2進数データを**セクション**毎に順番に出力.
  - ロケーションカウンタ**(LC)で、出力バイト数を管理.

.text pushq %rax
.data .ascii "A¥0"
.text movq %rsp, %rbp
.data .word 0x1234



55
48
89
e5

LC1 ...

.text  
セクション

41
00
34
12

LC2 ...

.data  
セクション



$$0x10000FB8 + 0x48 = 0x100001000$$

## アセンブラの主な仕事 (2) %ripからの 相対アドレス

- 記号表を作り, ラベルをアドレスに変換.
  - 例: ラベル `_x` をアドレス `0x00002014` に変換.

```
.text
movq %rax, _x
```

```
.data
.globl _x
_x:
.long 999
```



```
%objdump -d a.out
__text:
100000fb1:48 89 05 48 00 00 00
               movq %rax, 72(%rip)
100000fb8:
```

```
% nm a.out
00000000100000fb1 T __main
00000000100001000 D _x
```

記号表の  
内容

- 変換したデータをバイナリ形式
- のファイルとして出力.
  - Mac OS Xの場合は Mach-O形式.
  - ヘッダと複数のセクションから構成.

ヘッダ
.text
.data
.rdata
記号表

それぞれが  
セクション



# アセンブラ命令の主な種類

- セクション指定

```
.text
```

以降を `.text` セクションに出力せよ.

- データ配置

```
.long 0x12345678
```

4バイトの整数値 `0x12345678` の2進数表現を出力せよ.

- 出力アドレス調整

```
.align 4
```

4バイト境界にアラインメント調整せよ.  
(4の倍数になるようにロケーションカウンタの値を増やせ. )

- シンボル情報

```
.globl _main
```

シンボル `_main` をグローバルにせよ.  
(記号表のエントリにフラグを立てる)

- その他



# ラベル

ラベル

```
.globl _add5  
add5:  
    pushl    %ebp
```

- ラベルは機械語命令やアセンブラ命令の前に書ける.
  - たいてい, ラベルだけの行を書く.
- アセンブラが自動的にラベルをアドレスに変換する.
- 識別子 (変数名, 関数名) やジャンプ先を表すのにラベルを使う.
- オペランドにラベルを書いて良い.
  - アドレスが書ける場所なら.

```
movq %rax, _x(%rip)
```





# インラインアセンブラ

- インラインアセンブラ (inline assembler)
  - 高級言語中にアセンブリコードの記述を可能にする。
  - 記述したものをインラインアセンブリコードという。
- アセンブリコードの記述量を減らせる。
  - アセンブリコードの生産性・保守性・移植性は低い。
- GCCではasm構文で記述する。

```
int main (void)
{
    asm ("nop");
}
```

nop命令を埋め込む  
単純な例

```
#include <stdio.h>
int main (void)
{
    void *addr;
    asm ("movq %%rsp, %0": "=m" (addr));
    printf ("rsp = %p¥n", addr);
}
```

スタックポインタ (%rsp) の値を変数addrに格納する例