アセンブリ言語

アセンブラ命令

情報工学系
権藤克彦

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.

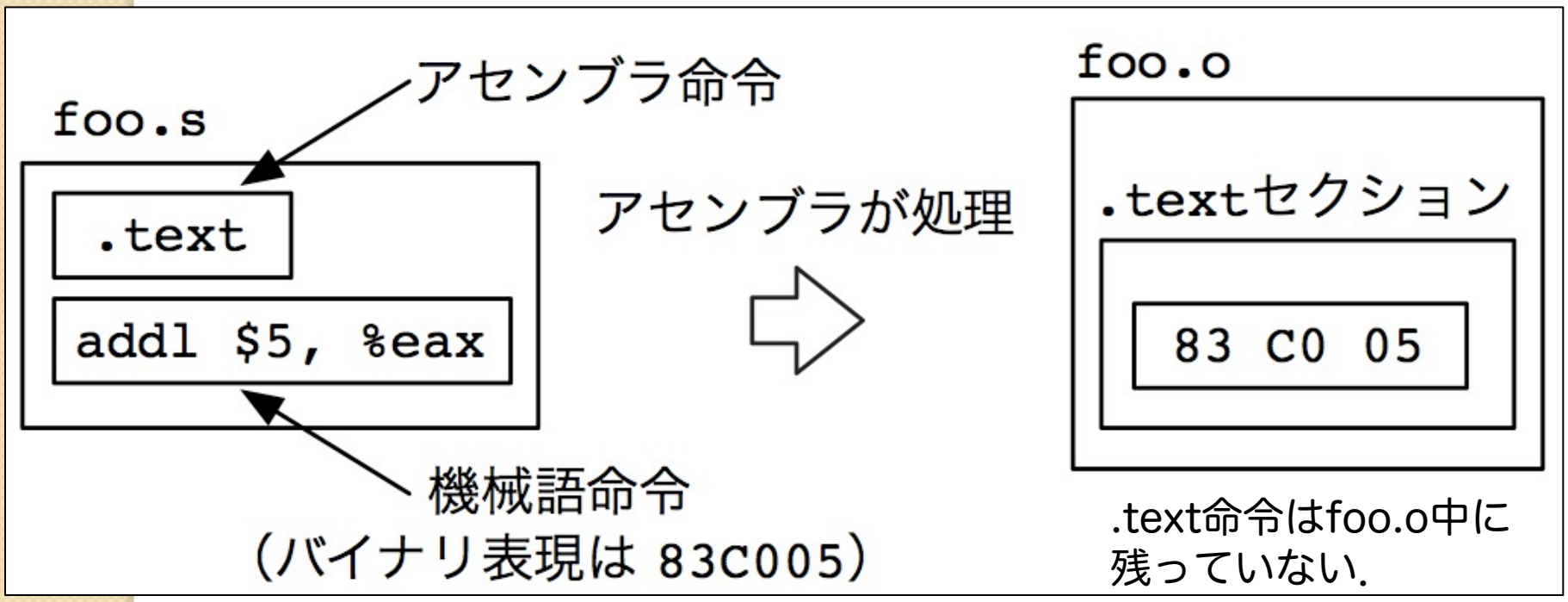
pseudo instruction,
pseudo opcode

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	`.text`	アセンブラ	アセンブル時	ない
機械語命令	`movq %rsp, %rbp`	CPU	実行時	ある

アセンブラ命令（例：.text）

- .textは次をアセンブラに指示.
  - 「これ以降の機械語命令やデータを.textセクションに格納せよ」



アセンブラの主な仕事（1）

- 機械語命令のニモニックを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.

```
pushq %rax
```

```
50
```

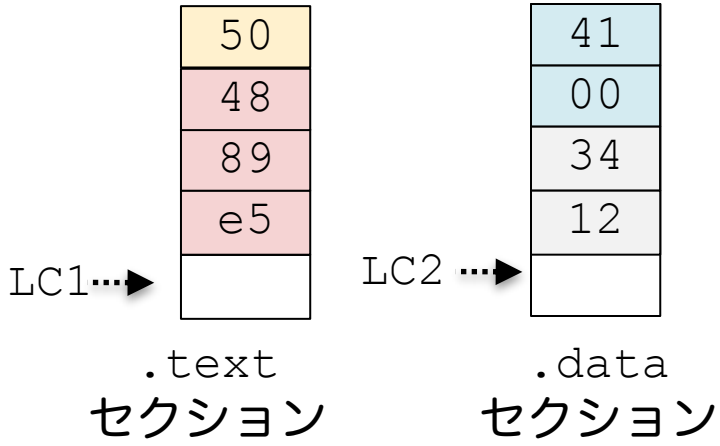
- 2進数データをセクション毎に順番に出力.
  - ロケーションカウンタ(LC)で，出力バイト数を管理.

```
.text
pushq %rax
```

```
.data
.ascii "A¥0"
```

```
.text
movq %rsp, %rbp
```

```
.data
.word  0x1234
```



アセンブラの主な仕事（2）

- 記号表を作り，ラベルをアドレスに変換.
  - 例：ラベル _x を相対アドレス 0x00000048 に変換.

```
.text
movl %eax, _x(%rip)
```

89
05
48
00
00
00

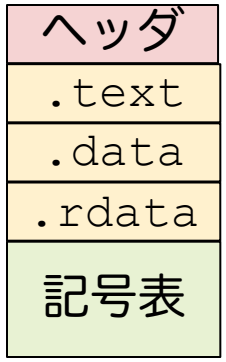
```
.data
.globl _x
_x:
.long  10
```

```
% nm a.out
00000fb2 T _main
00001000 D _x
```

記号表

0xfb2+6+0x48=0x1000

- 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.



それぞれがセクション

GNUアセンブラの文法

- 文
- コメント
- 定数
- 演算子
- シンボル

文(statement)

- 文はアセンブリ言語の構文単位.

ラベル: アセンブラ命令

ラベル: 機械語命令

ラベル:

アセンブラ命令

機械語命令

空の文

どれも1つの文

- 文は改行(¥n)かセミコロン記号(;)で区切る.
  - セミコロンで区切れば，複数の文を1行に書いて良い.

```
pushq %rbp
movq  %rsp, %rbp
subq  $8, %rsp
```

```
pushq %rbp;
movq  %rsp, %rbp;
subq  $8, %rsp;
```

どれもOK

```
pushq %rbp; movq %rsp, %rbp; subq $8, %rsp
```

コメント（1）

- プログラマのメモ書き．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.

定数(constant)

- C言語とほぼ同じ.
- 数値
  - 例：74, 0112, 0x4a, 0b01001010 （すべて同じ値）
- 文字
  - 例：'a, '¥¥, '¥b, '¥n, '¥x4a （クオートで囲まない）
  - 'a' という形式を許す場合もあるが，'a が正しい.
  - なぜか，macOSでは 'a' ，'a はエラーになる💢
- 文字列
  - 例："Hello¥n", "Hello¥n¥0"
  - アセンブラ命令 .ascii はヌル文字(¥0)終端しない.
    必要な場合は明示的に ¥0をつける.
    - cf. アセンブラ命令 .asciz はヌル文字終端する.

式と演算子

- 定数やラベルを書ける場所には式を書ける.
- 式は静的に（アセンブル時に）計算できるものだけ.
  - レジスタやメモリ中の値は参照できない.
- 式では演算子（例：+）を使える.

```
.text
movq L1+10(%rip), %rax
L1:
```

演算子の意味と優先順位

2項演算子の+と-を除いて，演算子のオペランドは絶対値（リンクで変化しない値）でなければいけない．

単項演算子

演算子	意味
–	（2の補数による）負数
~	ビット毎の反転（1の補数）

２項演算子

演算子	意味
*	乗算
/	除算
%	剰余
<<	左シフト
>>	右シフト

優先順位が最も高い

>

演算子	意味
|	ビットOR
&	ビットAND
^	ビットXOR
!	ビットOR NOT

a!b=a|~b
macOSでは使えない

>

演算子	意味
+	加算
–	減算
==	等しい
!= <>	等しくない
<	小さい
>	大きい
<=	以下
>=	以上

>

演算子	意味
&&	論理AND
||	論理OR

優先順位が最も低い

述語演算子は，真の時は-1を，偽の時は0を返す．ただし，なぜかmacOSでは真の時に1を返す．

シンボル(symbol)

- シンボルに使える文字
  - [$_.a-zA-Z][$_.a-zA-Z0-9]*
    - アルファベットか「_」「$」「.」のどれかで始まり，その後に，アルファベットか数字か「_」「$」「.」の0個以上の任意の組み合わせが続いたもの.
  - 大文字と小文字は区別する (case-sensitive).
  - 「.」で始めるとアセンブラ命令と紛らわしい（使えるけど）.
  - 「$」は（x86-64では使えるが）使えない環境もある.
- 主にラベルに使う.
  - シンボルにコロン記号(:)が続くのがラベル.
  - 環境によっては，_foo が，C言語上では foo に対応.
- シンボルは値を保持する.
  - シンボル≒（アセンブラが使用する）変数
  - 例：ラベルのアドレス，アセンブラ命令.setで与えた数値

普通は値を変えない.

ラベル

- ジャンプ命令のジャンプ先
- 変数名
- 関数名

ラベル(label)

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)
```

```
movq %rax, _x
```

PIC (position independent code)ではこれはNG.
PICでは絶対アドレスは使えないから（相対アドレスのみ）

position independent code

PICと絶対・相対アドレス

- 近年，静的リンクではなく動的リンクを通常使う．
- PIC
  - 仮想メモリ中のどこに配置しても実行可能なコード
  - 共有ライブラリ（動的ライブラリ）に使用
  - 絶対アドレス （例：_x）は使えない
  - 相対アドレスと間接アクセスを使う
    - コンパイル時に相対アドレスが決まる（ことがある）
    - 「実行時にはこのアドレスに絶対アドレスを入れる」とコンパイル時に約束

```
movq %rax, _x
```

```
%  gcc -c foo.s
foo.s:2:1: error: 32-bit absolute addressing is
not supported in 64-bit mode
movq %rax, _x
^          macOSの64ビットモードでは絶対アドレス使えない
```

ラベルのスコープ

弱い(weak)シンボルは同名でも
大丈夫だが，ここでは説明しない．

- 同一ファイル内に同じラベル名があってはダメ．

```
.text
Label_foo:
Label_foo:
```

```
% gcc -c foo.s
foo.s:3:FATAL:Symbol Label_foo
already defined.
```

- グローバルなラベルは他のファイルでもダメ．

foo1.s

```
.text
.globl _foo
_foo:
```

foo2.s

```
.text
.globl _foo
_foo:
```

```
% gcc foo1.s foo2.s
ld: duplicate symbol _foo in ...
collect2: ld returned 1 exit status
```

ローカルなラベル

- 2つの異なる「ローカルなラベル」がある.
  A. .globl がないグローバルではないラベル（以下のfoo2）.
  B. 大文字Lで始まるラベル（以下のLfoo3）.
- 大文字Lで始まるラベルはバイナリファイル中の記号表には格納されない.
  - ジャンプ命令のジャンプ先のラベルに主に使用.

```
.text
.globl foo1
foo1:
foo2:
Lfoo3:
```

foo1: グローバルなラベル.
　　記号表に入る.
foo2: ローカルなラベル（A）.
　　記号表に入る.
foo3: ローカルなラベル（B）.
　　記号表に入らない.

```
% gcc -c foo.s
% nm foo.o
00000000 T foo1
00000000 t foo2
```

記号表中にLfoo3は無い

ラベル：変数名

- 初期化済みの静的変数は .dataセクションに静的に（コンパイル時に）確保される.
  - cf. 未初期化の静的変数は .bssセクションに確保（後述）.
  - cf. 自動変数や引数は動的に（実行時に）スタック上に確保.
- 静的変数名＝静的変数の先頭を表すラベル（アドレス）

foo.c

```
        int x1 = 111;
static int x2 = 222;
int main (void) {
    return x2;
}
```

foo.s

```
   .globl _x1
   .data
   .p2align 2
_x1:
   .long 111
   .p2align 2
_x2:
   .long 222
```

```
% gcc -c foo.c
% nm a.out
00001004 D _x1
00001008 d _x2
```

アセンブラが大域変数に
アドレスを割り振っている.

静的変数

- 静的変数＝大域変数かstaticのついた局所変数.
- staticのついた局所変数は
  - 所属するブロック内からしか参照できないが,
    実行開始前から終了まで，ずっと存在している.

```
        int x1 = 111;
static int x2 = 222;
int main (void)
{
static int x3 = 333;
        int x4 = 444;
}
```

グローバルスコープを持つ静的変数.
ファイルスコープを持つ静的変数.
} 外部変数
(大域変数)

ブロックスコープを持つ静的変数.
ブロックスコープを持つ自動変数.
} 局所変数

ラベル：関数名

- 関数は .textセクションに静的に確保.
- 関数名＝関数の先頭を表すラベル（アドレス）

foo.c

```
int inc (int n)
{
    return n + 1;
}

int main ()
{
}
```

foo.s

```
  .text
.globl _inc
_inc:
pushq   %rbp
movq    %rsp, %rbp
addl    $1, %edi
movl    %edi, %eax
popq    %rbp
retq
```

```
% gcc foo.c
% nm a.out
00001fea T _inc
```

ラベル：ジャンプ命令のジャンプ先

- ローカルなラベルはジャンプ命令のジャンプ先として使用する.

```
int abs (int n)
{
    if (n < 0)
        n = -n;
    return n;
}
```

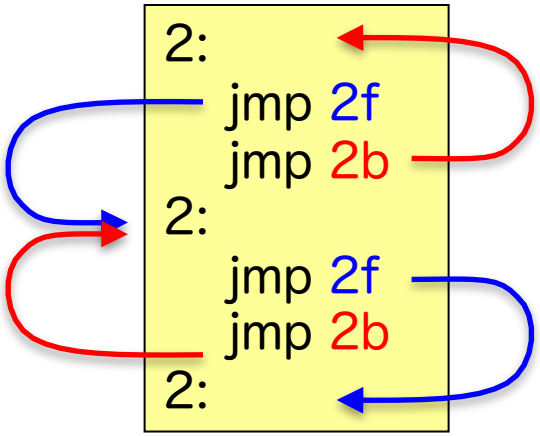
```
.text
.globl  _abs
.p2align        4, 0x90
_abs:
  pushq   %rbp
  movq    %rsp, %rbp
  movl    %edi, -4(%rbp)
  cmpl    $0, -4(%rbp)
  jge     LBB0_2
  xorl    %eax, %eax
  subl    -4(%rbp), %eax
  movl    %eax, -4(%rbp)
LBB0_2:
  movl    -4(%rbp), %eax
  popq    %rbp
  retq
```

ラベル：数値ラベル

インラインアセンブリコード中で使うと便利.

- 数値ラベル
  - 正の整数にコロン(:)がついたもの.
- 何度でも同じ数値ラベルを再定義可能.
  - ラベル名の衝突を気にせずに済む.
- 参照時には接尾語としてbかfを付ける.
  - b (backward)は後方で最初の同名の数値ラベルを参照.
  - f (forward)は前方で最初の同名の数値ラベルを参照.

```
__text:
0: eb 02 jmp 2  <__text+0x4>
2: eb fc jmp -4 <__text>
4: eb 02 jmp 2  <__text+0x8>
6: eb fc jmp -4 <__text+0x4>
```



ラベル：特別なドットラベル(.)

- ドット記号(.)のラベルは特別扱い.
  - アセンブラによる現在の出力アドレスを意味する.
  - つまり，現在のロケーションカウンタの値.

```
_foo:
   .quad   .
```

_fooが指す値は
_fooのアドレス自身.

```
_foo:
   .quad _foo
```

これと同じ

Linux (Ubuntu 18.04), GCC-7.3.0の出力

```
add5:
   pushq    %rbp
   movq     %rsp, %rbp
   movq     %rdi, -8(%rbp)
   movq     -8(%rbp), %rax
   addq     $5, %rax
   popq     %rbp
   ret
   .size    add5, .-add5
```

「.-add5」は関数add5の
サイズ（バイト単位）になる.

アセンブラ命令

- セクション指定
- データ配置
- 出力アドレス調整
- シンボル情報

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
```

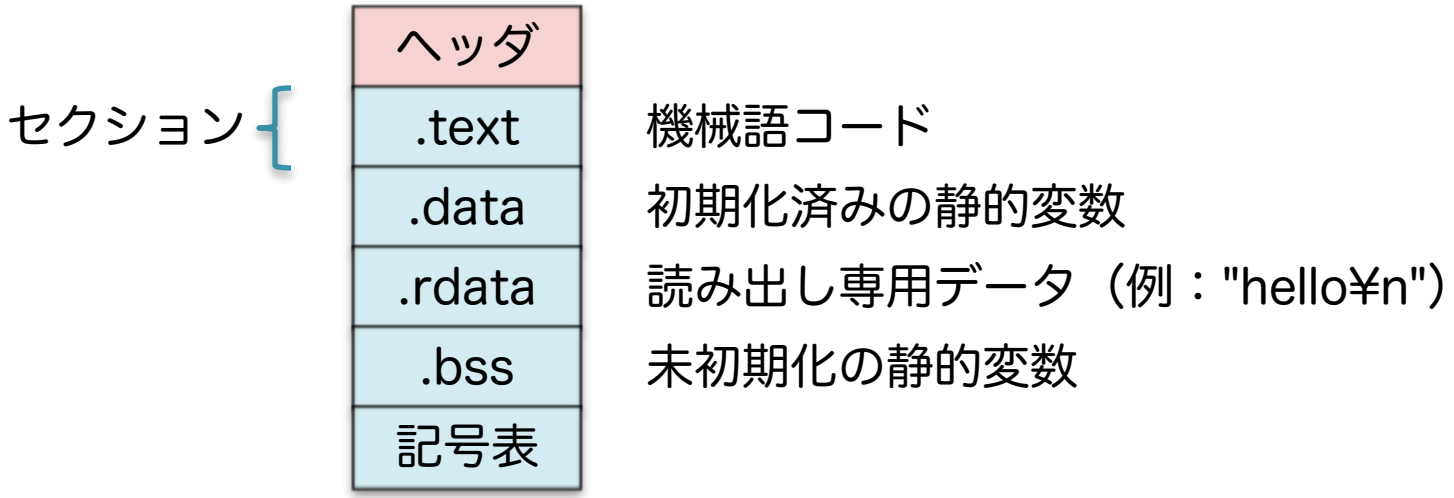
シンボル_mainをグローバルにせよ.
（記号表のエントリにフラグを立てる）

- その他

セクション

- セクションはバイナリコードの分割単位.
- 主なセクション
  - .text，.data，.rdata，.bss の４つ.
  - 他には記号表や再配置情報などのセクションがある.
- ヘッダ
  - 「どんなセクションがあるか」という情報を保持.

名前が異なる場合もあるが，
役割は大きくこの4つ.



セクション：objdump –h （1）

読み方は
次スライド参照.

- objdump –h はセクション情報を表示.

```
% objdump -h foo.o
foo.o:     file format mach-o-x86-64
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000033  0000000000000000  0000000000000000  00000270  2**4
                  CONTENTS, ALLOC, LOAD, RELOC, CODE
  1 .data         00000004  0000000000000034  0000000000000034  000002a4  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .cstring      00000008  0000000000000038  0000000000000038  000002a8  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 __LD.__compact_unwind 00000020  0000000000000040  0000000000000040  000002b0  2**3
                  CONTENTS, RELOC, DEBUGGING
  4 .eh_frame     00000040  0000000000000060  0000000000000060  000002d0  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```

.rdata は，macOS では .cstring というセクション名に.

記号表の中身を見るには nm コマンドや objdump --symsを使う.
文字列を見るには strings を使う.

セクション：objdump –h （2）

- セクション＝中身＋様々な属性値.

項目	説明
Size	セクションのバイト数
VMA	Virtual Memory Addressの略．実行時の先頭メモリアドレス．再配置に使用.
LMA	Load Memory Addressの略．ロード時の先頭メモリアドレス．通常はVMAと一致.
File off	ファイルオフセット．ファイル先頭からのバイト数.
Align	アラインメント制約．例えば，2**2は2^2を意味し，4バイト境界に要配置.

セクションフラグ	説明
CONTENTS	このセクションには内容がある.
ALLOC	ロード時にメモリ割り当てが必要.
LOAD	ロード時にこのセクションをファイルからロードする必要がある.
RELOC	このセクションは再配置が必要である．再配置情報を含む.
READONLY	このセクション中のデータは読み出し専用である.
CODE	このセクションは機械語コードを含む.
DATA	このセクションはデータ（静的変数）を含む.

アセンブラ命令：セクション指定

- .textや.dataなどは出力先のセクションを指定する.
  - 指定しなくてもアセンブラが勝手に作るセクションもある.
    - 例：記号表，.bss（後述），再配置情報，デバッグ情報.

出力先セクションを指定する主なアセンブラ命令

アセンブラ命令	説明
.text	以降のデータを.textセクションに出力.
.data	以降のデータを.dataセクションに出力.
.rdata	以降のデータを.rdataセクションに出力.
.section セクション名	以降のデータを指定したセクションに出力.

例：.text と .data

アセンブリコード

```
    .text
    pushl %ebp
    movl  %esp, %ebp
    movl  $5, %eax
    .data
_x:
    .long 10
_y:
    .long 20
```

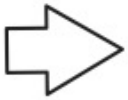
対応する16進バイナリ表現

```
55
89 e5
b8 05 00 00 00

0a 00 00 00

14 00 00 00
```

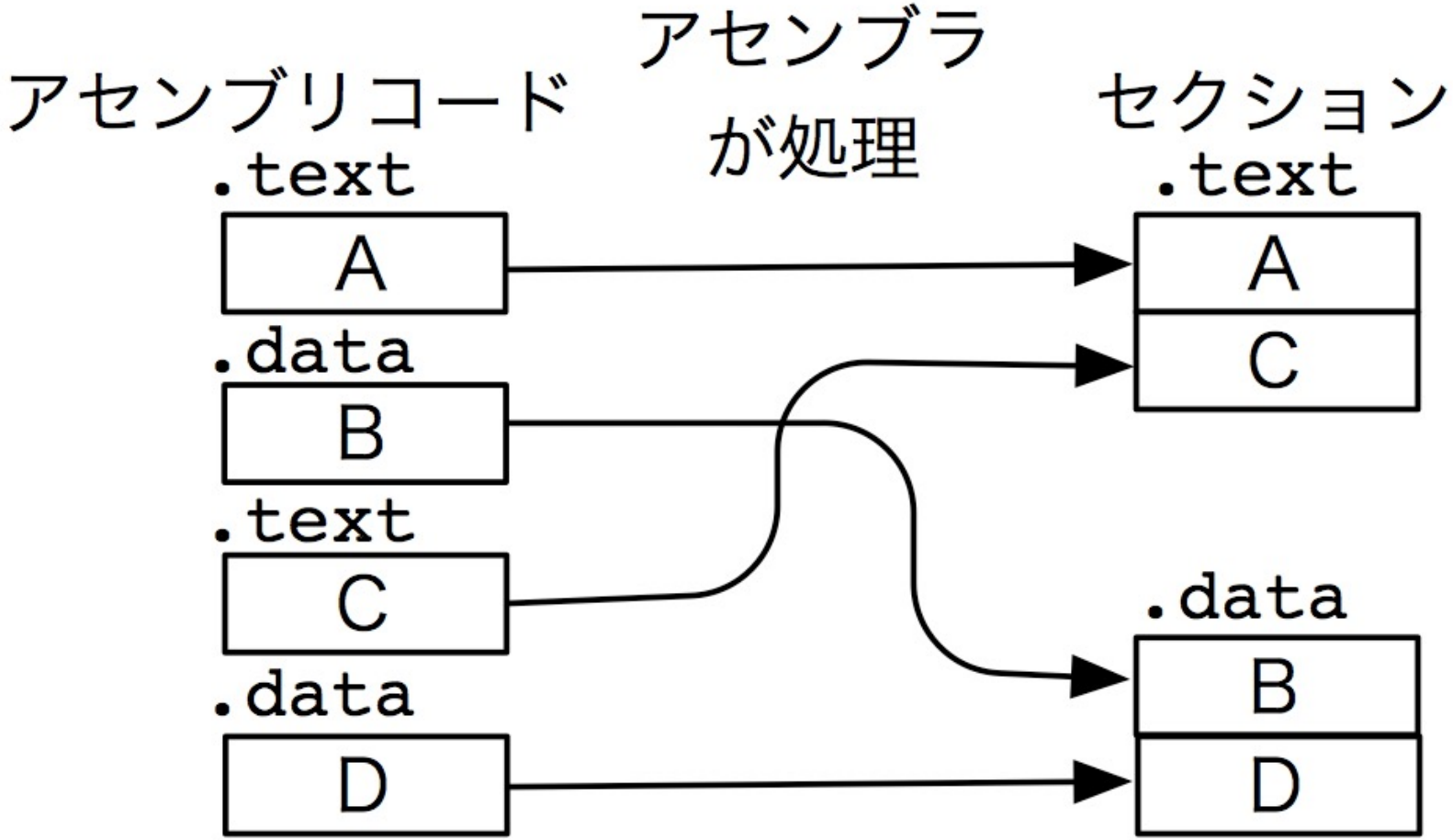
アセンブラで処理



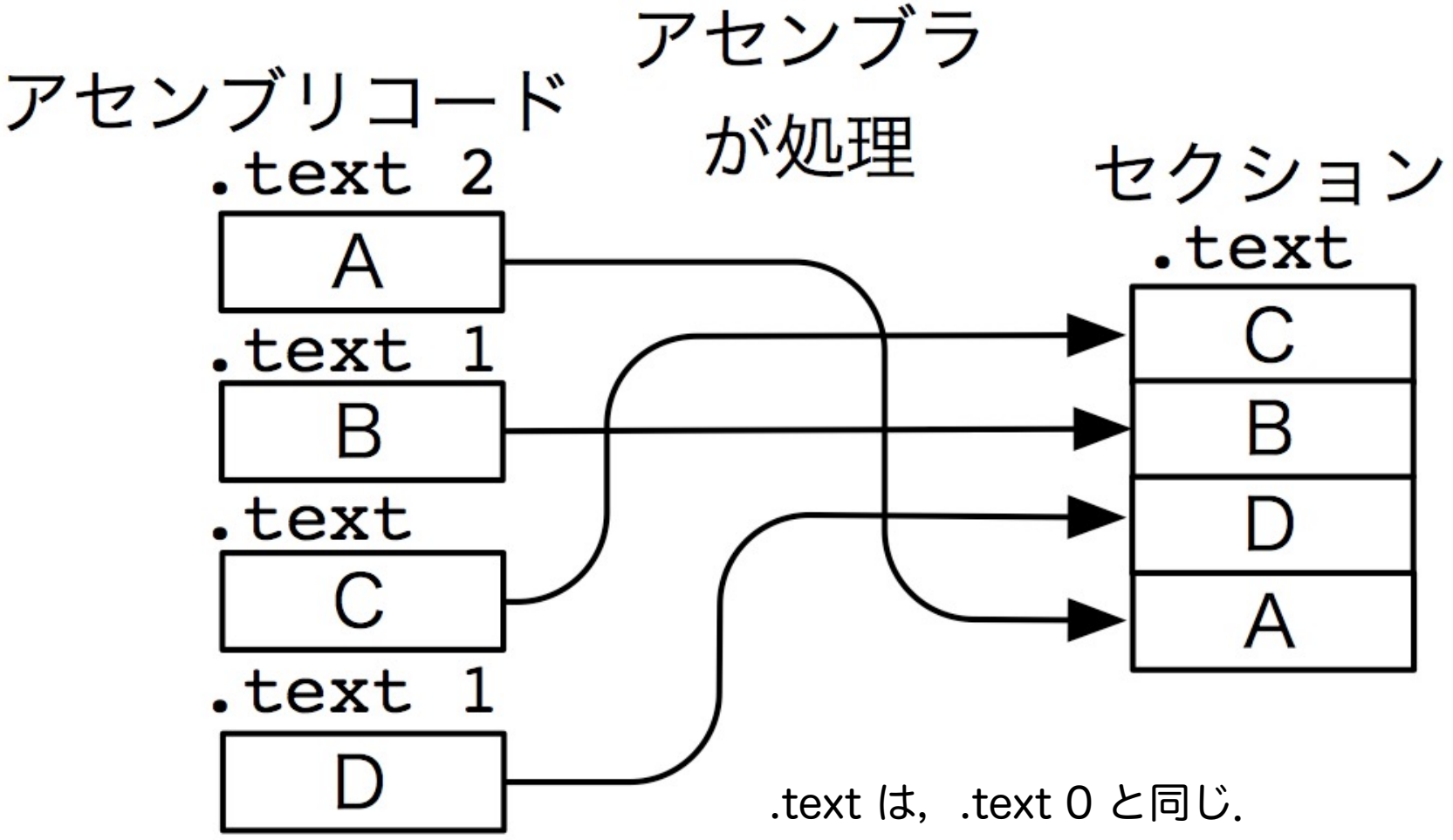
セクション

.text	.data
55	0a
89	00
e5	00
b8	00
05	14
00	00
00	00
00	00

.textや.dataは何度でも指定できる



.textや.dataは順序も指定できる



.text は，.text 0 と同じ.

.text中のデータは機械語と解釈される

- アセンブラにとっては「pushq %rbp」と「0x55」は同じデータ.

```
.text
.byte 0x55
.byte 0x48, 0x89, 0xe5
.byte 0x48, 0xc7, 0xc0, 0x05, 0x00, 0x00, 0x00
```

```
% gcc -c asm-bin.s
% objdump -d asm-bin.o
0: 55                          pushq %rbp
1: 48 89 e5                    movq  %rsp, %rbp
4: 48 c7 c0 05 00 00 00        movq  $5, %rax
```

.bssセクション

- .bssセクションは「未初期化の静的変数」を格納.
- .bssセクションは，メモリにそのままコピーしない.
  - cf. .textや.dataはそのままメモリにコピーする.

- .bssセクションはバイナリファイルのサイズを小さくする効果がある.
- .bss という名前は歴史的な経緯でついた.
  - block started by symbol の略．でも覚える必要はない.
- Mac OS Xでは2つに分かれる.
  - __DATA.__bss　　static付きの静的変数
  - __DATA.__common　グローバルスコープを持つ大域変数
- 共通シンボル(common symbol)
  - グローバルスコープを持つ未初期化の静的変数の別名.
  - 記号Cやアセンブラ命令 .comm は，この common に由来.

変数の分類（1）

bss.c

```
        int x1 = 10;
        int x2;
static int x3 = 10;
static int x4;
extern int x5;

int main (void)
{
    int y1 = 10;
    int y2;
    static int y3 = 10;
    static int y4;
    extern int y5;
}
```

コンパイル時に.dataセクションに確保.
コンパイル時に.bssセクションに確保.
コンパイル時に.dataセクションに確保.
コンパイル時に.bssセクションに確保.
何も確保しない.

実行時にスタック上に確保.
実行時にスタック上に確保.
コンパイル時に.dataセクションに確保.
コンパイル時に.bssセクションに確保.
何も確保しない.

変数の分類（2）

```
% gcc -c bss.c
% nm bss.o
00000000 T _main
00000044 d _main.y3
000000b0 b _main.y4
00000040 D _x1
00000004 C _x2
00000048 d _x3
000000b4 b _x4
```

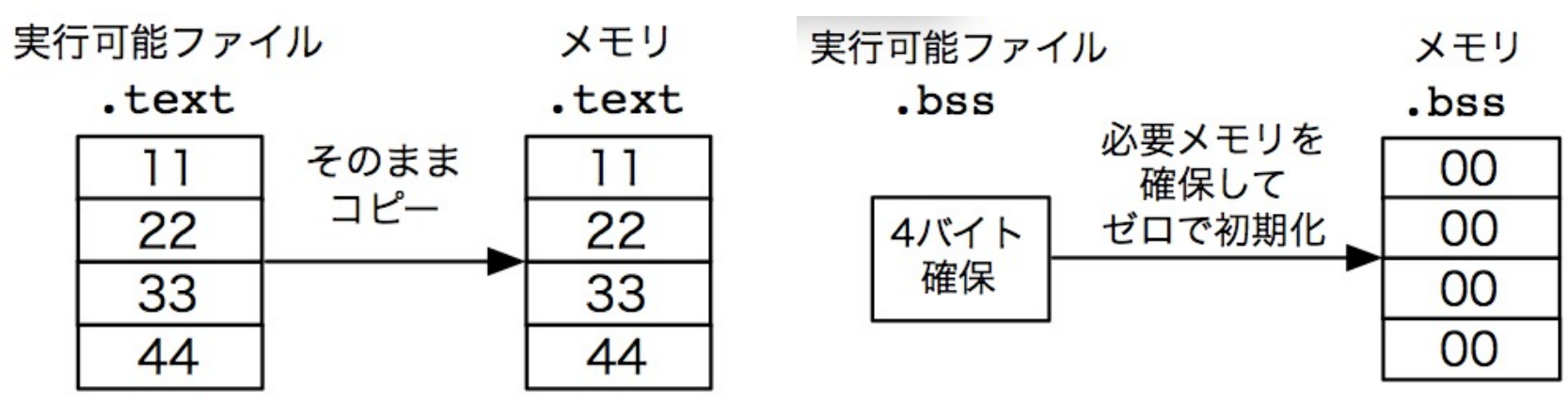
nmコマンドが出力する記号の意味

記号	説明
T	.textセクション中のシンボル（関数名）
B	.bssセクション中のシンボル
C	.bssセクション中のグローバルスコープを持つシンボル
D	.dataセクション中のシンボル
U	参照されているが未定義のシンボル

大文字の記号はグローバルスコープ，
小文字の記号はファイルスコープであることを示す．

.bss がファイルサイズを小さくする理由

- .bss中の変数は「ゼロで初期化する」と決まっている.
  - つまり，変数の初期値の格納が不要.
  - .bssセクションはロード時に必要メモリを確保して
    ゼロで初期化する.
- 例：char a[4096];
  - .dataセクションに格納するには 4KBの領域が必要.
  - .bssセクションに格納するには「4KB確保せよ」という
    情報のみ必要→ファイルサイズは小さくて済む.



出力サイズはx86-64の場合.

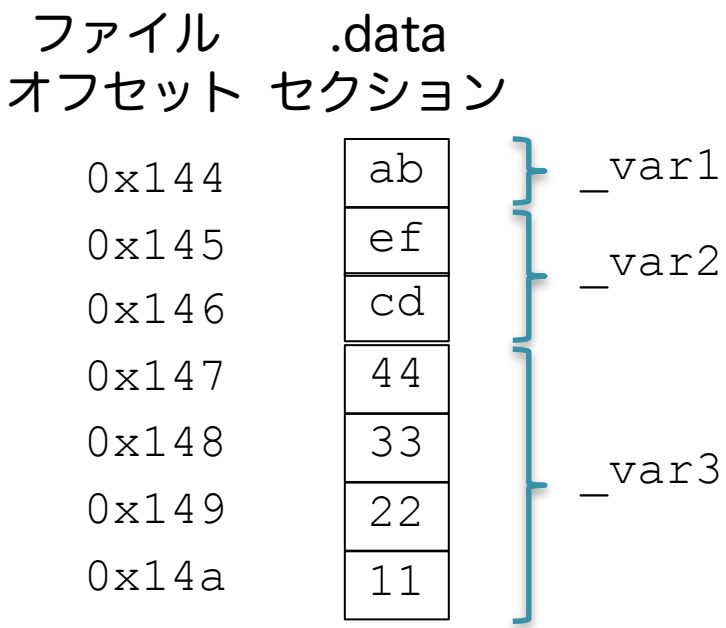
アセンブラ命令：データ配置

- .long や .ascii はアセンブル結果としてデータを出力.
- .ascii はヌル文字(¥0)を付加しないが,
  .asciz はヌル文字(¥0)を付加する.
- カンマ記号(,)で区切って複数個のデータを指定可.

アセンブラ命令	説明
.byte　式	1バイトデータを出力.
.word　式	2バイトデータを出力.　.shortも可.
.long　式	4バイトデータを出力.　.int も可.
.quad　式	8バイトデータを出力.
.ascii　文字列	文字列を出力（ヌル文字 ¥0を最後に付加しない）
.asciz　文字列	文字列を出力（ヌル文字¥0を最後に付加する）. .string も可.
.fill　データ数, サイズ，値	指定した「データ数，サイズ，値」のデータを出力. サイズと値は省略可.

.byte，.word，.longは数値を出力（1）

```
.data
_var1:
.byte 0xAB
_var2:
.word 0xCDEF
_var3:
.long 0x11223344
```



odコマンドで.dataセクションの中身を表示.

```
% od -Ax -t x1 data.o | less    （出力を大幅に省略）
0000144   ab ef cd 44 33 22 11 00 01 00 00 00 0e 02 00 00
0000154   00 00 00 00 08 00 00 00 0e 02 00 00 01 00 00 00
```

この実行例では出現箇所が0x144バイト目だが，これは環境依存.
x86-64はリトルエンディアンなのでデータが逆順に見える.

.byte，.word，.longは数値を出力（2）

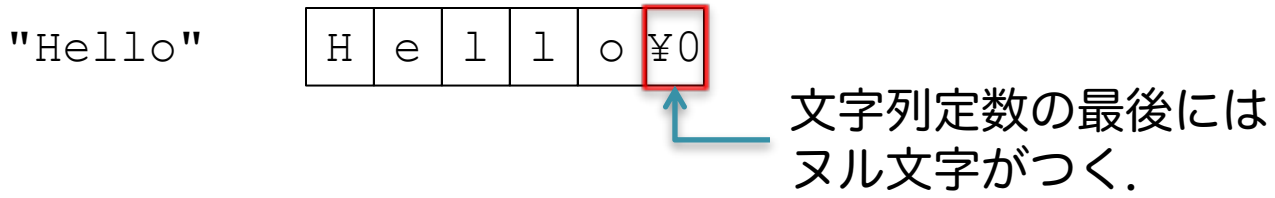
```
% objdump -D data.o
Disassembly of section LC_SEGMENT.__DATA.__data:
00000000 <_var1>:
   0:   ab              stos   %eax,%es:(%edi)
00000001 <_var2>:
   1:   ef              out    %eax,(%dx)
   2:   cd 44           int    $0x44
00000003 <_var3>:
   3:   44              inc    %esp
   4:   33 22           xor    (%edx),%esp
   6:   11              .byte 0x11
```

objdumpコマンドで.dataセクションの中身を表示.
（-d ではなく -D を指定すると全セクションを逆アセンブルする．）

ヌル文字 (null character)

- C言語で値0を持つ文字．文字定数 '¥0'．
- ヌルポインタ(NULL)ではない．
  - ヌルポインタは値0のポインタ．「どこも指してない」を表現．
- C言語では文字列の最後をヌル文字で表現する．

サイズが異なる．
値は一緒．



- アセンブリコードではヌル文字が邪魔なことも．
- ヌル文字の有無をプログラマが意識する必要あり．

.asciiと.asciz は文字列を出力

- .asciiは最後にヌル文字(¥0)を付加しない.
- .ascizは最後にヌル文字(¥0)を付加する.

```
.data
.ascii "abc"
.asciz "def"
.byte  'X
```

```
% gcc -c ascii.s
% objdump -h ascii.o  (出力を大幅に省略)
1 .data 00000008 0000000000000000 0000000000000000 00000118  2**0
% od -Ax -t c -j0x118 ascii.o
0000118   a  b  c  d  e  f ¥0  X
0000120
```

確かに.ascizはヌル文字を付加.

objdump –h で.dataセクションがファイル先頭から0x118バイト目と判明.
odコマンドの-jオプション（jump)で0x118バイト目以降を表示.

.fillは指定した数値で領域を埋める

- 形式は「.fill　データ数，バイトサイズ，値」
  - サイズと値は省略可．省略するとサイズは1，値は0.
- 特定パターンでメモリ領域を埋めたいときに使用.
  - 例：0で初期化された40バイトの領域を確保したい時.

```
.data
.fill 3, 2, 0xA
.byte 0xF
```

2バイトのデータ0xAを
3個出力せよ.

```
% gcc -c fill.s
% objdump -h fill.o
000000dc（出力を大幅に省略）
% od -Ax -tx1 -j0xdc fill.o
00000dc 0a 00 0a 00 0a 00 0f 00
00000e4
```

確かに2バイトの0xAが3個出力されている.

アセンブラ命令：出力アドレス調整

- .align はパディングを入れてアラインメント調整する.
- .skip や .org はロケーションカウンタを増加.
  - .skip は増加量を（現在のLCの値に対する）相対値で指定.
  - .org は増加量を（セクションの先頭からの）絶対値で指定.

アセンブラ命令	説明
.align 式	式の値（Mac OS Xでは$2^{式の値}$）の倍数になるようにロケーションカウンタを増やす（アラインメント調整する）.
.p2align 式	$2^{式の値}$ の倍数になるようにロケーションカウンタを増やす（アラインメント調整する）.
.space 式	式の値をロケーションカウンタに加える.（.skip)
.org 式	式の値をロケーションカウンタに代入する.（.origin)

すき間には0が埋められる. .space 4, 0xFF などとすれば,
別の値（この場合は0xFF）で埋めることが可能.

例：.alignでアラインメント調整

```
.data
.byte 0x11
.align 4
.byte 0xaa
```

Linux (Ubuntu 18.04), GCC-7.3.0
4バイト境界に配置するため，3バイトのパディングが入っている．

```
% gcc -c align.s
% objdump -h align.o
0000008c （出力を大幅に省略）
% od -Ax -tx1 -j0x8c align.o
00008c 11 00 00 00 aa 00 00 00 2e 66 69 6c 65 00 00 00
```

macOS (10.13.6), GCC (Apple LLVM version 10.0.0)
2^4=16バイト境界に配置するため，15バイトのパディングが入っている．

```
% gcc -c align.s
% objdump -h align.o
000000dc （出力を大幅に省略）
% od -Ax -tx1 -j0xdc align.o
00000dc 11 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000ec aa 00 00 00
```

混乱を防ぐため，macOS では .p2align を使おう

セクション全体のアラインメント

- セクションもアラインメントされる.
- objdump –h の Algn の項を見れば確認できる.

```
% gcc -c align.s
% objdump -h align.o（出力を大幅に省略）
Sections:
Idx Name Size      VMA       LMA       File off Algn
.data    00000011 00000000 00000000 000000dc 2**4
```

このデータセクションは
16バイト境界にアラインメント
調整される.

例：.space

```
.data
.byte 0x11
.space 4
.byte 0xaa
```

確かに4バイト，スキップできている．

```
% gcc -c space.s
% objdump -h space.o
000000dc  2**0
%  od -Ax -tx1 -j0xdc space.o
00000dc 11 00 00 00 00 aa 00 00
00000e4
```

.org

- .org はセクション先頭からのオフセット値を指定して，ロケーションカウンタを変更．
  - ただし，後戻りする指定は不可．後戻りを指定すると，単に無視されるかエラーになる．

確かに4バイト目に0xaaを出力している．

```
.data
.byte 0x11
.byte 0x22
.byte 0x33
.org  4
.byte 0xaa
```

```
% gcc -c org.s
% objdump -h org.o
000000dc （出力を大幅に省略）
%  od -Ax -tx1 -j0xdc org.o
00000dc 11 22 33 00 aa 00 00 00
00000e4
```

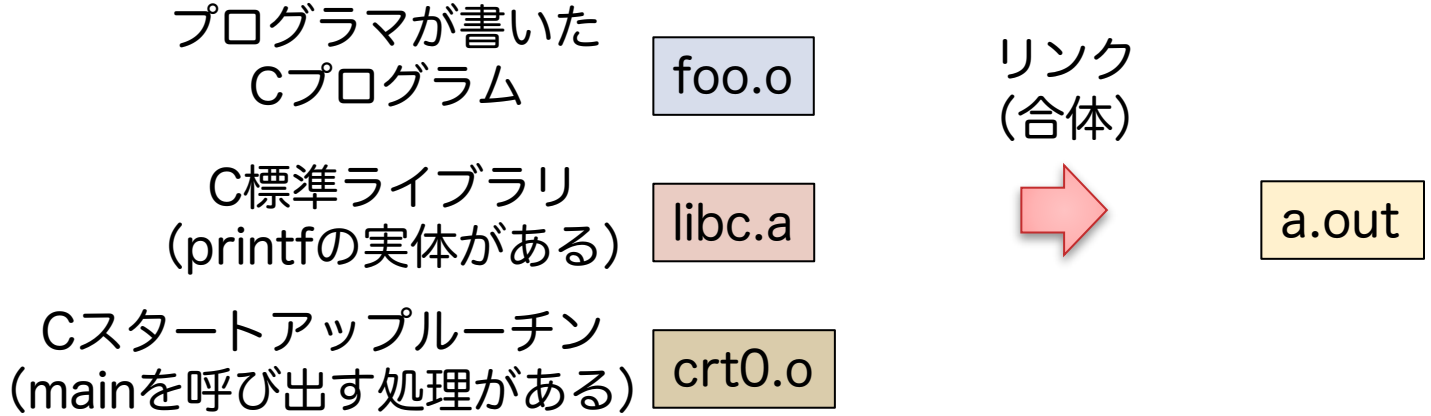
アセンブラ命令：シンボル情報など

- グローバルスコープを持つ大域変数や関数は，.globl (.global)で宣言する.
- 未初期化の静的変数は .comm や .lcommで宣言する.
  - .comm や .lcomm の宣言は，環境で多少異なる.

アセンブラ命令	説明
.globl　シンボル	シンボルをグローバル（リンク可能）にする. (.global)
.local　シンボル	シンボルをローカル（リンク不可能）にする.（ELFのみ）
.comm シンボル，サイズ，アラインメント	.bssセクションにグローバルスコープの未初期化変数を登録する.
.lcomm シンボル，サイズ，アラインメント	.bssセクションにファイルスコープの未初期化変数を登録する.

リンク

- リンク(link)
  - オブジェクトファイル(*.o)やライブラリファイルを結合して，1つの実行可能ファイル(a.out)にすること．
  - リンカ（コンパイラの一部，ldコマンド）が実行処理．
- アドレスの調整が必要．
  - 外部シンボルの解決や再配置などを行って調整する．



外部シンボルの解決

```
% gcc -c main.c
% gcc -c sub.c
% gcc main.o sub.o ← リンク
% ./a.out
999
%
```

- 外部シンボル (external symbol)
  - 自分のファイル中で未定義の変数や関数．例：以下の変数x.
- 外部シンボルの解決
  - ファイルをまたいで，変数や関数の対応関係を調べて，変数名や関数名の参照にメモリアドレスを割り当てること．
  - つまり，未定義の変数や関数を無くす．

main.c

```
#include <stdio.h>
extern int x;
int main (void)
{
    printf ("%d¥n", x);
}
```

sub.c

```
int x = 999;
```

対応付けて，未定義な状態を解決．

.globl （1）

- グローバルスコープを持つ大域変数や関数は，.globl （または .global）で宣言する．
  - .globlで宣言したシンボルは他ファイルから参照可能になる．

g1.c

```
int foo1 = 1;
```

g2.c

```
static int foo2 = 1;
```

g1.s

```
.globl _foo1
  .data
  .p2align 2
_foo1:
  .long  1
```

g2.s

```
  .data
  .p2align 2
_foo2:
  .long  1
```

.globl （2）

- 関数 foo3 はグローバルスコープ.
- 関数 foo4 はファイルスコープ（ローカル）.

g3.c

```
int foo3 () {}
```

g4.c

```
static int foo4 () {}
```

g3.s

```
   .text
.globl _foo3
_foo3:
   pushq %rbp
    (中略)
   ret
```

g4.s

```
   .text

_foo4:
   pushq %rbp
    (中略)
   ret
```

.comm と .lcomm（1）

- 形式は「.comm シンボル，サイズ，アラインメント」

c1.c

```
int foo5;
```

c2.c

```
static int foo6;
```

c1.s

```
.comm _foo5,4,2
```

macOS

c2.s

```
.lcomm _foo6,4,2
```

macOS

.zerofill も使用.
セクションを指定可能

=

c3.s

```
.globl _foo5
  .data
  .p2align 2
_foo5:
  .long  0
```

c1.sはc3.sと同じ意味.
ただし，c1.sは.bssセクションに_foo5を置き，
c3.sは.dataセクションに_foo5を置く点のみ異なる.

.comm と .lcomm（2）

- .commや.lcommの宣言は環境で微妙に変化.
  - あまり気にせず，コンパイラ出力をまねる（この授業では）.

c2.c

```
static int foo6;
```

c2.s

```
.local _foo6
.comm _foo6,4,4
```

Linux/ELF

c2.s

```
.lcomm _foo6,4,2
```

macOS/Mach-O

c2.s

```
.lcomm _foo6,16
```

Windows(Cygwin)/PE

アラインメント制約の指定を省略し，代わりに必要なバイト数より多く確保する作戦らしい.
最新のWindowsは知らんw

アセンブラ命令：マクロ

- マクロ用のアセンブラ命令が用意されている.
- でも，C前処理系を使えば，ほぼ使わずに済む.
  - そのためには拡張子を .S （大文字）にする.

マクロ関連の主なアセンブラ命令

アセンブラ命令	説明	C前処理系
.set シンボル，式	式にシンボル名を与える.	#define
.macro	.endmなどと組み合わせてマクロを定義する.	#define
.ifdef	.endifなどと組み合わせて条件付きアセンブルを行う.	#ifdef
.include ファイル名	指定したファイルを取り込む.	#include

例：マクロ関連のアセンブリ命令

- .set と #define の例．どちらも同等のマクロ．

macro1.s

```
.set X, 0x999
.text
movl $X, %eax
```

=

macro2.S ← 大文字

```
#define X $0x999
.text
movl X, %eax
```

.setでは数値しかセットできないので，
.set X, $0x999 とはできない．

環境依存なアセンブラ命令（1）

- （この授業では）あまり気にしなくて良い.
  - コンパイラ出力の真似をして，しのぐ.
  - よく見るものを次のスライドで例示します.
  - 詳しくはアセンブラのマニュアルを熟読する.

環境依存なアセンブラ命令（2）

- Windows (Cygwin)/PE

コンパイルしたソースファイル名

```
.file   "add5.c"
  .text
.globl _add5
  .def _add5;.scl 2;.type 32;.endef
_add5:
  pushl   %ebp
  movl    %esp, %ebp
  movl    8(%ebp), %eax
  addl    $5, %eax
  popl    %ebp
  ret
```

scl=storage classの略

シンボル_add5の記憶クラスは外部(2)で，型は（返り値情報のない）関数（0x20）であることを示す．

環境依存なアセンブラ命令（3）

- Linux/ELF

```
   .file   "add5.c"
   .text
   .globl  add5
   .type   add5, @function
add5:
   pushq   %rbp
   movq    %rsp, %rbp
   movq    %rdi, -8(%rbp)
   movq    -8(%rbp), %rax
   addq    $5, %rax
   popq    %rbp
   ret
   .size   add5, .-add5
   .ident  "GCC: (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0"
   .section  .note.GNU-stack,"",@progbits
```

シンボルadd5の型は関数.
（@objectだと変数）

シンボルadd5のサイズをアセンブラに伝える.

コンパイラ情報等のコメント

スタック上のコード実行を禁止.

環境依存なアセンブラ命令（4）

- macOS/Mach-O

```
   .section __TEXT,__text,regular,pure_instructions
   .macosx_version_min 10, 13
   .globl      _add5
   .p2align    4, 0x90
_add5:
   pushq       %rbp
   movq %rsp, %rbp
   movq %rdi, -8(%rbp)
   movq -8(%rbp), %rdi
   addq $5, %rdi
   movq %rdi, %rax
   popq %rbp
   retq
   .subsections_via_symbols
```

デッドコード＝どんな入力に対しても決して実行されないコード.

最適化のためのアセンブラ命令.
デッドコードの削除が目的.
セクションが個別のブロックに分割可で,
そのブロック単位で削除可能と伝える.
各ブロックの先頭にシンボルが必要.

結果としてMach-Oのヘッダファイル中の該当フラグがオンになる．otoolで見られる.

```
%  otool -hv add5.o （一部省略）
Mach header
      magic cputype caps filetype ncmds sizeofcmds      flags
MH_MAGIC_64   X86_6  0x00    OBJECT     4        432 SUBSECTIONS_VIA_SYMBOLS
```