



# アセンブリ言語

## イントロダクション（1）

情報工学系  
権藤克彦





# アセンブリ言語(assembly language) とは何か？

- 機械語を記号で記述するプログラミング言語の一種。
  - 「記号で」は正確には「ニemonic(mnemonic)で」.
  - アドレスも記号（ラベル）で記述して、自動計算させる.
- アセンブリ言語で記述したプログラムの例：

ラベル

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

機械語命令のニemonicの例  
「%rspレジスタの値を%rbpレジスタ  
に転送（コピー）せよ」

今は理解できなくてOK

# コンパイラとアセンブラ（1）

- コンパイラはC言語をアセンブリ言語に変換する。
  - これを「コンパイルする」という。
  - 実行例：gcc -S foo.c で, `foo.s` ができる。
- アセンブラはアセンブリ言語を機械語に変換する。
  - これを「アセンブルする」という。
  - 実行例：gcc -c foo.s で, `foo.o` ができる。



# 機械語

- 機械語（マシン語）(machine language)
- CPUは機械語しか理解・実行できない。
  - 他のプログラミング言語はすべて機械語に変換して実行.
- 2進数で表現.
  - 人間には理解しにくい.

16進数では0x50.

pushq %rax

01010000

x86-64 機械語命令  
のニモニック

x86-64 機械語命令  
(2進数表記)

2進数の機械語命令  
よりは理解しやすい

人間には理解しにくい



# コンパイラとアセンブラー (2)

add5.c

```
long add5 (long n)
{
    return n + 5;
}
```

テキスト

コンパイル

% gcc -S add5.c

add5.s

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

テキスト



# .cfi で始まるアセンブリ命令

- Call Frame Information の略。
  - スタックトレースを得るために必要。特に、-fomit-frame-pointer が有効な場合
  - この授業の範囲では無視してOK
- gcc の以下のオプションで出力抑制
  - fno-asynchronous-unwind-tables
  - fno-verbose-asm

```
_add5:  ## @add5
        .cfi_startproc
## %bb.0:
    pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
    movq   %rsp, %rbp
        .cfi_def_cfa_register %rbp
```



# コンパイラとアセンブラー (3)

- gcc -c でアセンブルすると, add5.o ができる.
    - % gcc -c add5.s
  - add5.o はバイナリファイルなので, lessで表示不可.
    - 中身を見るには, odコマンドで16進ダンプする.

```
% less add5.o
<CE><FA><ED><FE>^G^@^@^@^C^@^@^A^@^@^@^C^@^@^@<E4>^@^@
^@^@^@^@^A^@^@^@ | ^@^@^@
(中略)
^@^@^O^A^@^@^@^@^@^@^@ add5^@^@
```

読めない

```
% od -A d -t x1 add5.o
0000000 cf fa ed fe 07 00 00 01 03 00 00 00 00 01 00 00 00
0000016 04 00 00 00 10 01 00 00 00 00 00 00 00 00 00 00 00 00
0000032 19 00 00 00 98 00 00 00 00 00 00 00 00 00 00 00 00 00
(中略)
0000336 64 35 00 00
```

## 先頭からのバイト数 (10進表記)

add5.oの内容の16進ダンプ (1バイトごとに16進表記)  
2023年度・3O アセンブリ言語



他にも多くのオプションがある。  
詳しくはマニュアル(info gcc)を参照。

# GCCの主なオプション

オプション	説明
-E	C前処理系の処理結果を標準出力に出力.
-S	アセンブリコードを.sファイルに出力.
-c	オブジェクトファイル (.o) を出力.
-o filename	出力ファイル名をa.outではなく, filenameにする.
-v	コンパイルの内部処理コマンドを表示. GCCのバージョンも表示.
-Wall	ほぼすべての警告を表示.
-g	デバッグ情報をa.outに付加. lldbコマンドでデバッグするのに必要.
-pg	プロファイル情報 (gmon.out) を生成するa.outを出力. gmon.outはgprofコマンドで処理して読む.
-l name	(小文字のエル) . リンクするライブラリ名nameを指定.
-L dir	ライブラリの検索ディレクトリにdirを追加.
-static	静的リンクを行う. (最近のGCCはデフォルトで動的リンクする)
-I dir	(大文字のアイ) . ヘッダファイルの検索ディレクトリにdirを追加.
-H	取り込んだヘッダファイルの絶対パスを表示.



# コンピュータは0と1だけの世界

- コンピュータ中のデータは、すべて0と1から成る。
  - つまり、2進数（バイナリ）で表現。
  - プログラム（機械語命令）も2進数で表現。



すべて  
2進数で表現



01100011
10000100
10100110
11000100
....

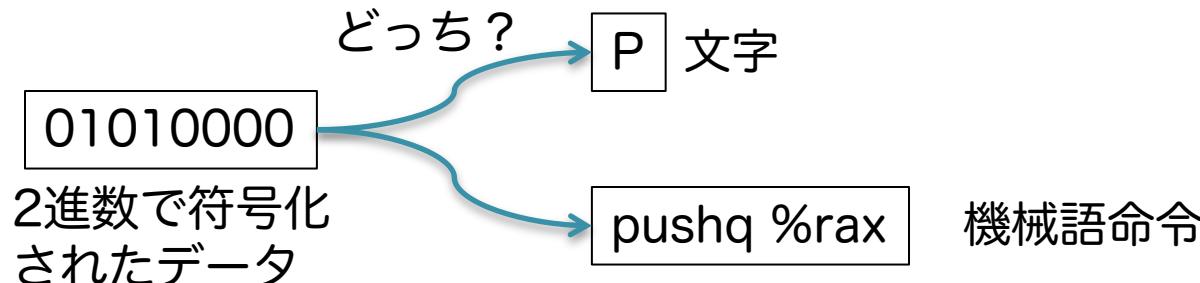
0と1からなる  
2進数データ  
(バイナリ)

- 2進数表現は長い→人間の読み書きには16進数を使う。
  - 例：2進数の01010000は、16進数では50。  
(10進数と区別するため 0x50, 50<sub>16</sub>などとも表現する)



# 2進数と符号化

- 符号化 (encoding)
  - ある規則に従って、文字や機械語命令などを符号（2進数）に変換すること。
  - 例：文字PをASCII文字として符号化すると 01010000. 同じ
  - 例：pushq %raxをx86-64機械語命令として符号化すると 01010000.
- ある2進数が何を表すかは解釈によって異なる。
  - 解釈の方法が分からなければ、01010000が何を表すのか分からない。





# テキストとバイナリ

- テキスト(text)とバイナリ(binary)
  - テキスト=文字として表示可能な2進数だけを含むデータ.
  - バイナリ=文字以外の2進数も含んだデータ.
- テキストはバイナリでもある.
  - テキスト中の文字も2進数で表現されているから.



# テキスト(text)の2つの意味

- テキストとバイナリ
  - テキスト=文字データ. 例：テキストファイル.
  - バイナリ=2進数データ（文字として表示不可能）.
- テキストとデータ
  - テキスト=機械語コード. 例：.textセクション.
  - データ=（機械語命令に処理される）データ.

# odコマンドと16進ダンプ

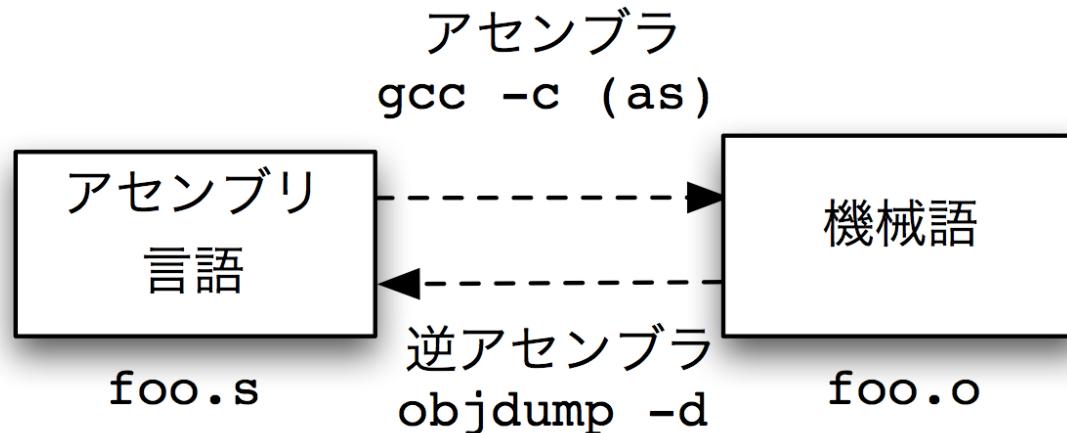
- odコマンドはバイナリファイルの内容を、指定した形式でダンプ（表示）する。
  - 指定した形式の例=1バイト毎に16進数で表示。
  - 名前の由来は「octal dump」（8進ダンプ）。
- 16進ダンプ=バイナリデータを1バイト毎に16進数で表示すること。
  - odコマンドで16進ダンプするには `-t x1` オプションを指定。

オプション	意味
<code>-t x1</code>	1バイト毎に16進数で表示
<code>-A d</code>	アドレスを10進数で表示
<code>-c</code>	1バイト毎に文字として表示
<code>-j 数値</code>	最初の「数値」バイトをスキップして表示

他のオプションは「man od」で確認。

# 逆アセンブラー (disassembler) (1)

- 逆アセンブラーは機械語からアセンブリ言語を復元する。
  - これを「逆アセンブルする」という。
  - 実行例 : objdump -d foo.o
  - プログラマの意図通りにアセンブルできたかの確認に使う。
- ただし、アセンブラー命令（例 : .text），ラベル，コメントは復元されない。



# 逆アセンブラー(disassembler) (2)

- 実行例

```
% objdump -d add5.o (出力を一部省略)
add5.o: file format Mach-O 64-bit x86-64
Disassembly of section __TEXT,__text:
_add5:
 0: 55                      pushq  %rbp
 1: 48 89 e5                movq   %rsp, %rbp
 4: 48 83 c7 05             addq   $5, %rdi
 8: 48 89 f8                movq   %rdi, %rax
 b: 5d                      popq   %rbp
 c: c3                      retq
```

先頭からの  
バイト数  
(16進表記)

機械語命令の  
16進表記

機械語命令の  
ニモンイック表記

注意：gobjdumpという名前でインストールされる場合がある。

# なぜアセンブリ言語を学ぶのか？（1）

- アセンブリ言語でしか書けない（書くと嬉しい）プログラムがあるから。
  - 組み込みシステム, デバイスドライバ, OSなど.
  - CPUの機能を最大限に引き出すため.
    - 特権命令, 特殊な命令 (例 : x86-64のSSE命令やAVX命令)
  - 実行速度を高速にするため.
    - ただし, コンパイラの最適化 (gcc -O) による高速化に勝つには, 適切にアセンブリ言語を使う必要あり.

# なぜアセンブリ言語を学ぶのか？（2）

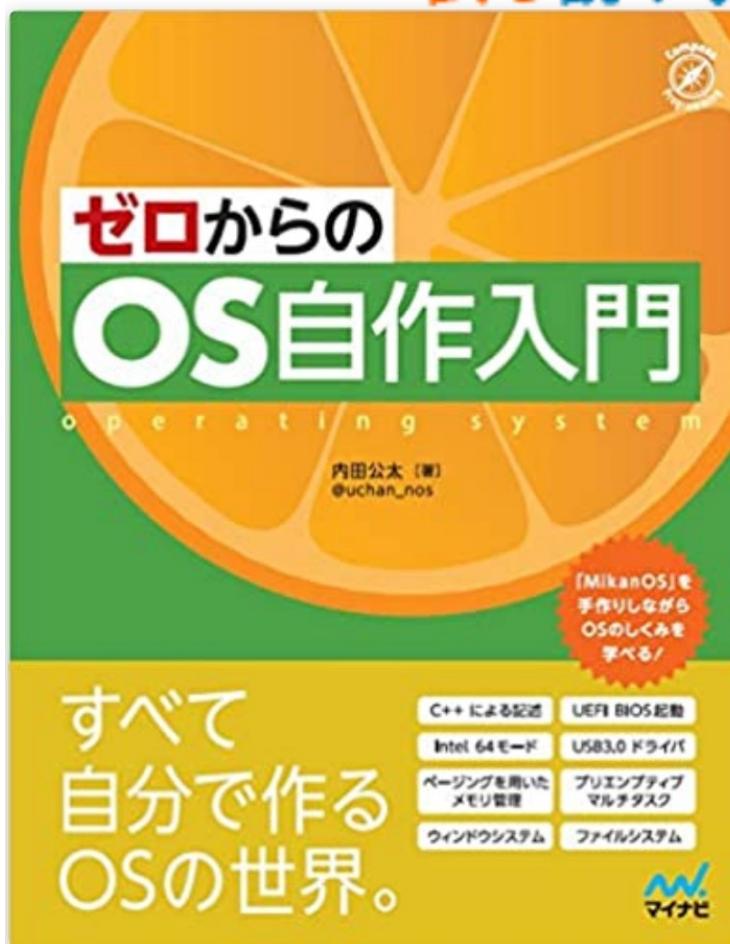
- コンピュータシステムの仕組みの学習に役立つ。
  - コンパイラ, OS, 計算機アーキテクチャなどの講義で.
- コードの解析（例：マルウェア）にも必須
- トラブルシューティング（問題解決）能力がアップ。
  - gcc -S の出力結果を読めるようになる.
- アセンブリ言語を使ったプログラミングは面白い.
  - 例：マイOSの作成.



# 宣传

情報工学系の旧名

- 著者の内田公太さんは計算工学専攻出身・権藤研OB.



通称「みかん本」  
2021/3 発売



# 参考：内田公太さんの講義動画

- 2020年度まで大学院講義を特任助教として担当
  - Youtube動画として閲覧可能
    - 例：Gitによるバージョン管理 ← 超オススメ  
<https://www.youtube.com/watch?v=WMIiPcqGC4Q>

# ワークツリーとインデックス

---

## ワークツリー

- 現在作業中のディレクトリ

## インデックス

- コミットのステージング領域
- git commitのコミット対象
- git addは変更をインデックスに登録する

```
graph TD; A[ワークツリー] -- "git add" --> B[インデックス]; B -- "git commit" --> C[リポジトリ(HEAD)]
```

The diagram illustrates the workflow of version control. It consists of three rounded rectangular boxes arranged vertically. The bottom box is labeled 'ワークツリー' (Working Directory). An upward-pointing arrow labeled 'git add' points from it to the middle box, which is labeled 'インデックス' (Index). Another upward-pointing arrow labeled 'git commit' points from the 'インデックス' box to the top box, which is labeled 'リポジトリ (HEAD)' (Repository).



# なぜアセンブリ言語は難しいのか？

- 本質はあまり難しくない。
- 難しく見える原因：
  - CPUやアセンブラにより、アセンブリ言語の記法が異なる。
    - このため、一般にアセンブリコードの移植性は悪い。
  - CPU（特にx86-64）は非常に多くの機械語命令を持つ。
    - この授業の理解に必要な機械語命令は少ない（20～30個程度）。
  - 様々な細かい知識が必要なため（総合格闘技）。
    - CPUの機械語命令
    - アセンブラ命令
    - ツール（gcc -S, odコマンド, objdumpコマンドなど）
    - バイナリ形式、バイトオーダ、スタックレイアウトなどのABI
      - ABI=application binary interface
    - ハードウェアの入出力インターフェース



# ABI と API

- **ABI** (application binary interface)
  - バイナリコードのためのインターフェース規格.
  - 同じABIをサポートするシステム上では再コンパイル無しで同じバイナリコードを実行できる.
  - コーリングコンベンション, バイトオーダ, アラインメント, バイナリ形式などを定める.
- **API** (application programming interface)
  - ソースコードのためのインターフェース規格.
  - 同じAPIをサポートするシステム上では再コンパイルすれば同じソースコードを実行できる.
  - 例 : POSIX や SUSv3 は UNIX の API . システムコール, ライブラリ関数, マクロなどの形式や意味を定めている.

UNIXシステムコールは3年1Qの  
システムプログラミングで学ぶ.

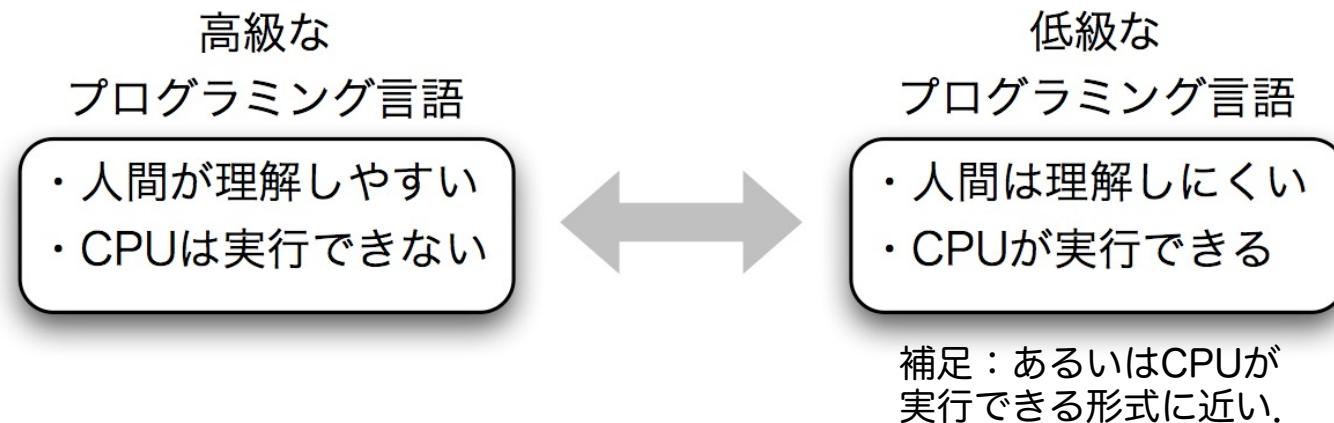
# アセンブリ言語の特徴

- 低レベル（機械語命令に近い）
  - アセンブリ言語の機械語命令（ニモニック）と2進数表現の機械語命令は、通常は一対一に対応。
  - 異なる高級言語（C, C++, Objective-C…）が同じアセンブリ言語にコンパイルされる。（ターゲットが同じなら）
- ○CPUの機能を最大限に引き出せる。
- ×生産性・移植性・保守性がとても低い。
  - プラットフォームが異なると、アセンブリコードを大幅に書き換える必要がある。cf. C言語, Java言語。
  - アセンブリコードは人間にとって理解しにくい。高級言語に比べてコードの記述量が増える。



# 高級言語, 低級言語

- C言語やJava言語などを**高級言語**,  
アセンブリ言語や機械語を**低級言語**という.
  - 記述がハードウェア(CPU)に近いことを低級（低レベル）,  
人間に近いことを高級（高レベル）と呼ぶことから.
  - 低級は「劣っている」という意味ではない.



# 一対一（1）：2つの意味

- 処理結果の命令数

- 1つの文や式をコンパイルすると複数の機械語命令（ニモニック）になる。
- 1つの機械語命令（ニモニック）は（通常は）1つの機械語命令（2進数コード）になる。

- 処理結果の唯一性 (uniqueness)

- （同じ動作をする）コンパイル結果は複数ある。解は複数。
- アセンブル結果は（通常は）ただ1つに決まる。他に解はない。

要注意な他の英単語：naive



## 一対一（2）：命令数

- 高級言語の命令（文や式）と機械語命令のニモニックは**一対多の関係**。
- 機械語命令のニモニックと機械語コードは（通常は）**一対一の関係**。



# 一対一（3）：唯一性 uniqueness

```
if (n > 0)
    x = -1;
```

同じ動作をする  
コンパイル結果は  
何通りもある。

```
cmpq $0, -8(%rbp)
jle L1
movq $-1, _x(%rip)
L1:
```

```
cmpq $0, 8(%ebp)
jg L1
jmp L2
L1:
    movq $-1, _x
L2:
```

$n \leq 0$ ならジャンプ

$n > 0$ ならジャンプ  
無条件ジャンプ

```
movq %rdi, %rax
```

48 89 f8

通常、機械語命令の  
ニモニックは、ただ1つの  
機械語コードに対応する。



# 移植性と保守性

- 移植性 (portability)
  - 移植=他のプラットフォーム上で動作するようにプログラムを修正すること.
  - 移植性=移植のしやすさ
- 保守性 (maintainability)
  - 保守=機能追加・変更, バグ修正, 移植などのプログラム修正.
  - 保守性=保守（プログラム修正）のしやすさ.



環境(environment)とも言う.

# プラットフォーム (platform)

- プログラムのコンパイルや実行に必要となるソフトウェアやハードウェアのこと.
- 通常, OS, コンパイラ (バイナリ形式) , CPU の組み合わせで決まる.

OS	バイナリ形式	CPU
Windows	PE	x86-64
macOS	Mach-O	x86-64
Linux	ELF	x86-64

この授業で私達が使うプラットフォーム



# バイナリ形式 (binary format)

- オブジェクトファイルや実行可能ファイルの形式.
- いろいろある.

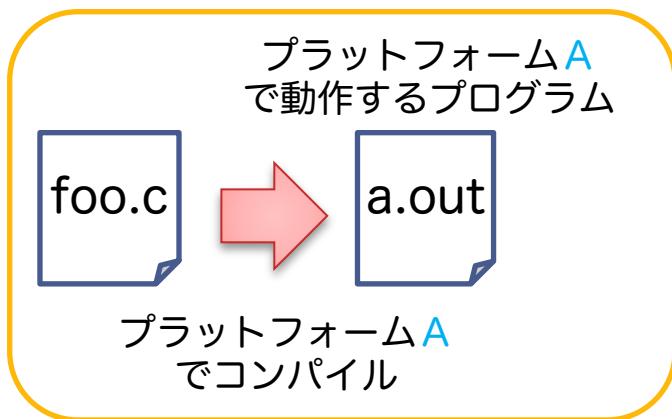
バイナリ形式	説明
Mach-O	macOS
a.out (assembler output)	古いUNIX
ELF (executable and linking format)	Linuxなど
PE (portable executable)	Windows

- GNU Binutils中のツール (例 : objdump)
  - バイナリを処理するコマンド群.
  - 異なるバイナリ形式に対して使って便利.

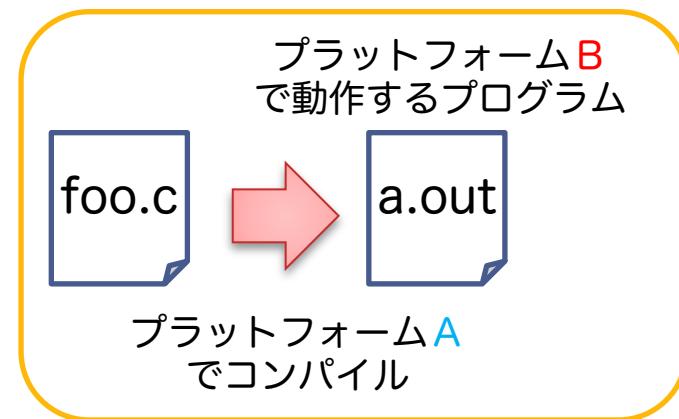


# ターゲット(target)

- ・ プログラムを実行するプラットフォーム.
- ・ 通常はプログラムをコンパイルするプラットフォームと同じ.
  - 異なる場合をクロスコンパイル(cross compile)という.



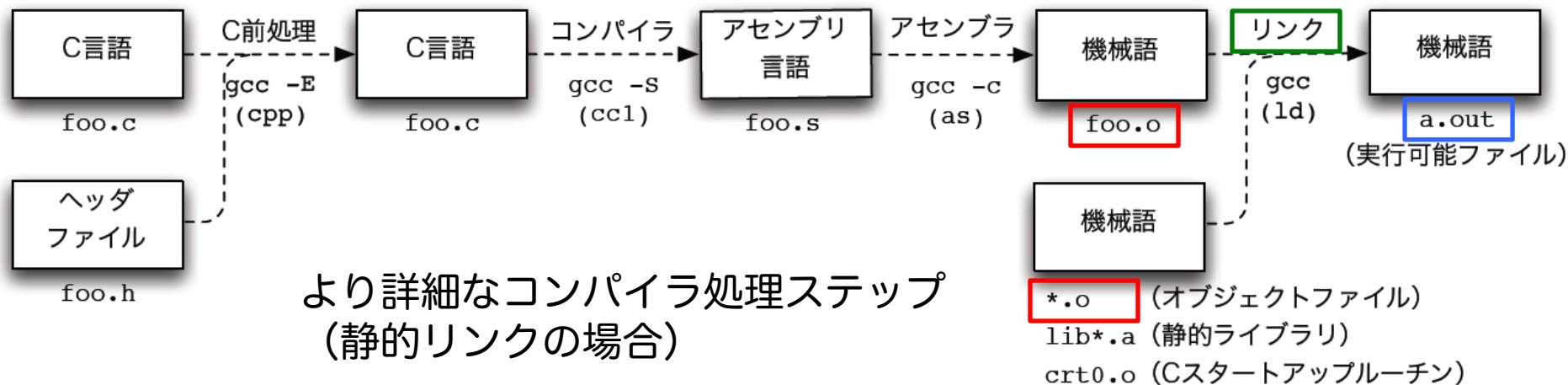
通常のコンパイル



クロスコンパイル

# オブジェクトファイル, 実行可能ファイル

- オブジェクトファイル = 拡張子が .o のファイル.
- 実行可能ファイル = a.out
  - Windowsでは拡張子が .exe のファイル.
- 複数のオブジェクトファイルを1つの実行可能ファイルに合体させることをリンクという.



より詳細なコンパイラ処理ステップ  
(静的リンクの場合)

\*.o (オブジェクトファイル)  
lib\*.a (静的ライブラリ)  
crt0.o (Cスタートアップルーチン)

# 静的リンクと動的リンク

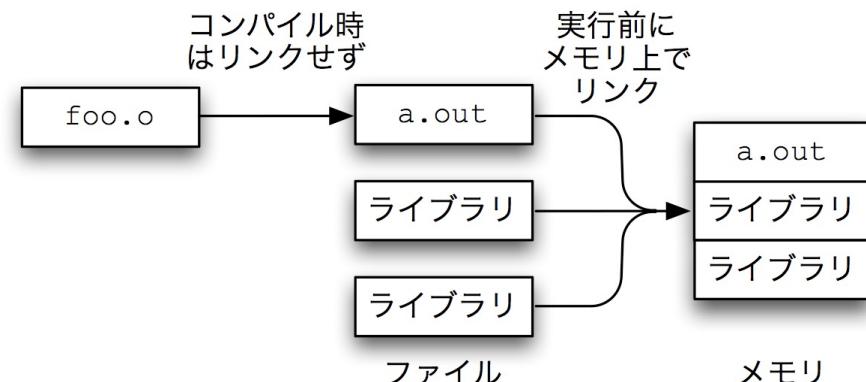
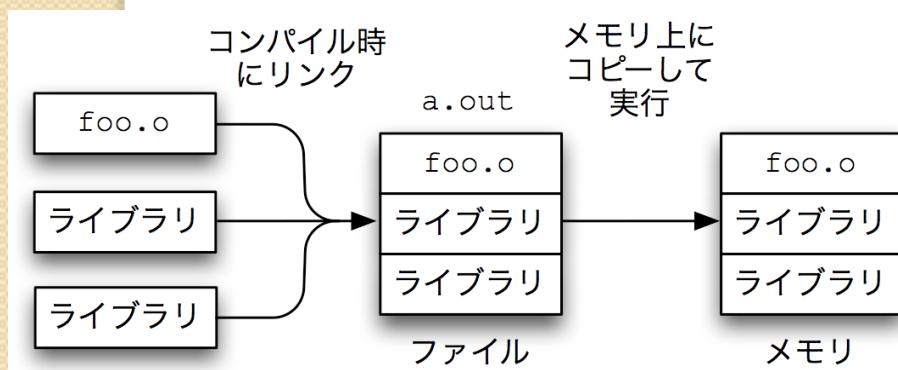
- 静的リンク

- コンパイル時にリンク
- 仕組みが単純
- a.outのサイズが大きい
- 実行時のメモリ消費大

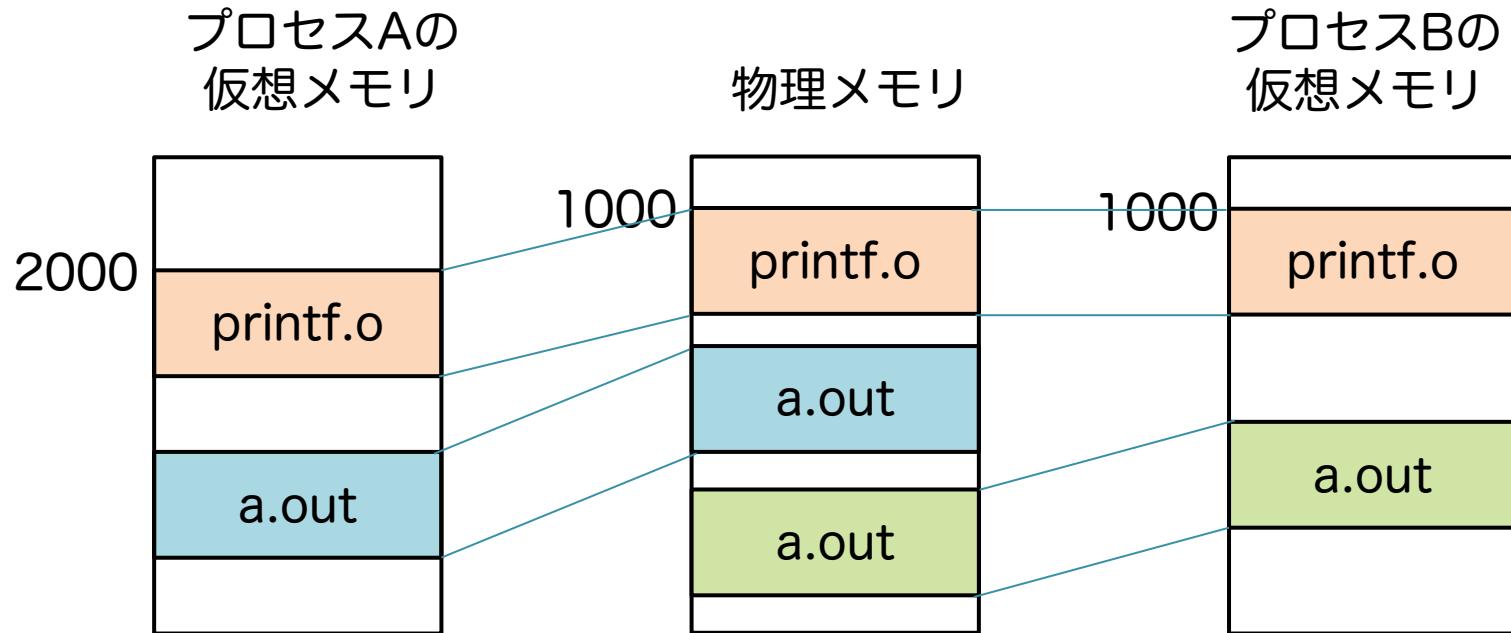
- 動的リンク

呼び出し時のリンクも

- 実行時にリンク
- 仕組みは複雑
- a.outのサイズは小さい
- 実行時のメモリ消費小
- 同じライブラリを使用する他のプロセスと共有するから



# メモリ上での動的ライブラリの共有



- 動的ライブラリは物理メモリ上で共有される
- 仮想メモリ上のアドレスはプロセスごとに異なる  
→ [位置独立コード](#) (PIC: position independent code)  
任意のアドレスで実行可能なコード



# gcc -static (gccで静的リンク)

- 最近のコンパイラのデフォルト動作は動的リンク。
  - 出力アセンブリコードが複雑。
- gcc に **-static** オプションをつけると静的リンク。
- ただし、macOS では静的リンクで a.out 作成不可。
  - foo.s や foo.o は作成可。

```
% gcc -static foo.c
ld: library not found for -lcrt0.o
clang: error: linker command failed with exit
code 1 (use -v to see invocation)
```

静的リンク用のcrt0.o (Cスタートアップルーチン) が無い、  
と文句を言われている。

# Cスタートアップルーチン

- 初期化とmain関数呼び出しを行うアセンブリコード  
擬似コード

```
//レジスタなどの初期化
initialize ();
// main実行後に終了
exit (main (argc, argv));
```

exit はプロセスを終了させる  
ライブラリ関数.  
引数は終了コード.

- アセンブリコードでmain関数を書くとき,  
returnもexitもしないとどうなる?  
→ main関数の次の命令を実行しようとする  
→ 暴走したり, segmentation faultが発生したり

C言語で書いたmain関数からは明示的にreturnしなくても  
関数終了時に return したことになる.  
コンパイラが ret 命令を最後につけてくれるから.



# Segmentation fault, Bus errorって何？

- 不適切なメモリアクセスによる実行時エラー
  - 発生時のデフォルト動作はプロセスの終了。
- 発生条件はプラットフォーム依存
- 発生したらラッキーと思え  
→ たまたま書き込めたらデバッグが大変だから

```
int main ()  
{  
    int *p = (int *)0x1;  
    return *p;  
}
```

```
% ./a.out  
Segmentation fault  
%
```

# x86-64：この授業で仮定するCPU

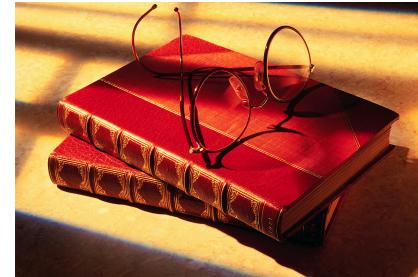
- Intel社のCPUのx86アーキテクチャ。
  - 別名 : x64, x86\_64, AMD64, Intel 64
  - 具体例 : Intel Core i7, Intel Core 2, Pentium D (2005)
- 64ビット, リトルエンディアン, CISC型.





# 参考資料（一次資料）

すぐURL切れするので、URL未掲載



- x86-64 機械語命令のマニュアル
  - インテル64およびIA-32アーキテクチャのソフトウェア開発者向けマニュアル
- GNUアセンブラー・逆アセンブラーのマニュアル
  - info as, info binutils, info bfd, info ld
  - Mac OS X Assembler Reference (英語) なぜか32ビットの情報のみ
- GCCオンラインアセンブラーのマニュアル
  - info gcc
- ABI for Mac OS X
  - System V Application Binary Interface AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models), Ver 1.0 (英語)
  - OS X ABI Mach-O File Format Reference (英語)



## 関連書籍

- プログラミングの力を生み出す本—インテルCPUのGNUユーザへ（改訂2版）
  - ISBN-13: 978-4274132070
- はじめて読む8086—16ビット・コンピュータをやさしく語る
  - ISBN-13: 978-4871482455
- はじめて読む486—32ビットコンピュータをやさしく語る
  - ISBN-13: 978-4756102133
- x86アセンブラー入門—PC/ATなどで使われている80x86のアセンブラーを習得
  - ISBN-13: 978-4789833424