



アセンブリ言語

x86-64 機械語命令（2）



情報工学系
権藤克彦

● コンパイル例で見る, 機械語命令の使用例

GCCなどのコンパイラが出力する
アセンブリコードを読めることは重要です.
GCCのバージョンにより出力は異なります



if (式) 文

単純な実装パターン

式のコード

```
popq %rax
cmpq $0, %rax
je    label
```

文のコード

label :

式の計算結果が
スタックトップに
あるという仮定.

コンパイル方法の正解は
1つだけではない.

```
long foo (long n)
{
    if (n > 0)
        return -1;
}
```

```
.text
.globl _foo
_foo:
    pushq %rbp
    movq  %rsp, %rbp
    movq  %rdi, -16(%rbp)
    cmpq  $0, -16(%rbp)
    jle   LBB0_2
    movl  $-1, -8(%rbp)
LBB0_2:
    movl  -8(%rbp), %rax
    popq  %rbp
    retq
```

第 1 引数

} ifの条件判断

} thenの部分

GCCのコンパイル例



if (式) 文1 else 文2

GCCのコンパイル例

単純な実装パターン

式のコード

```
popq %rax
cmpq $0, %rax
je L1
```

文1のコード

```
jmp L2
```

L1 :

文2のコード

L2 :

```
long foo (long n)
{
    if (n > 0)
        return -1;
    else
        return 1;
}
```

```
.text
.globl _foo
_foo:
pushq %rbp
movq %rsp, %rbp
movq %rdi, -16(%rbp)
cmpq $0, -16(%rbp)
jle LBB0_2
movq $-1, -8(%rbp)
jmp LBB0_3
LBB0_2:
movq $1, -8(%rbp)
LBB0_3:
movq -8(%rbp), %rax
popq %rbp
retq
```

thenの部分

elseの部分



while (式) 文

単純な実装パターン

L1:

式のコード

```
popq %rax
cmpq $0, %rax
je L2
```

文のコード

```
jmp L1
```

L2:

```
long foo (long n)
{
    long i = 10;
    while (i != 0)
        i--;
}
```

GCCのコンパイル例

```
.text
.globl _foo
_foo:
    pushq %rbp
    movq %rsp, %rbp
    movq %rdi, -16(%rbp)
    movq $10, -24(%rbp)
LBB0_1:
    cmpq $0, -24(%rbp)
    je LBB0_3
    movq -24(%rbp), %rax
    addq $-1, %rax
    movq %rax, -24(%rbp)
    jmp LBB0_1
LBB0_3:
    movq -8(%rbp), %rax
    popq %rbp
    retq
```

引数 n
変数 i

} while
の条件
文の部分



式のコード生成（概要）

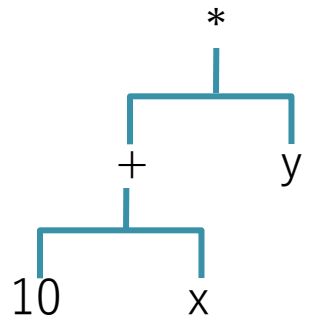
- スタック機械
 - レジスタの無いコンピュータ
- 中置記法と後置記法
- 後順序での木の訪問
- スタック使用のお約束（規約）
- 式をアセンブリコードに変換



式のコード生成（概要の続き）

- $(10+x)*y$ のコード生成のイメージ.

式の計算結果は
スタックに積む



抽象構文木

$10\ x + y\ *$

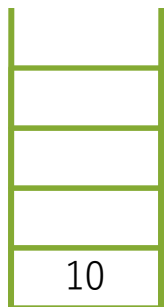
後置記法に変換

push 10
push x
add
push y
mul

これは実在しない
仮想コード

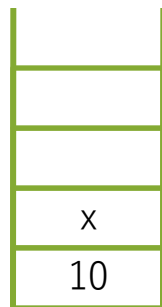
定数や変数はpush命令,
演算子は演算子命令に.

push 10



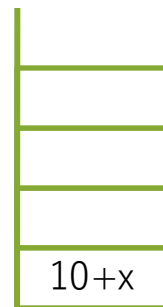
スタックに
10を積む

push x



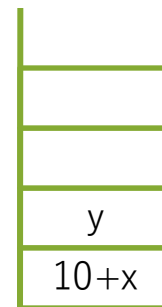
スタックに
xの値を積む

add



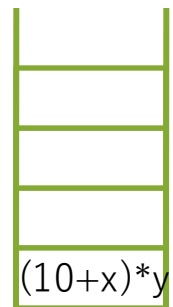
スタック上の
2つの値を足す

push y



スタックに
yの値を積む

mul

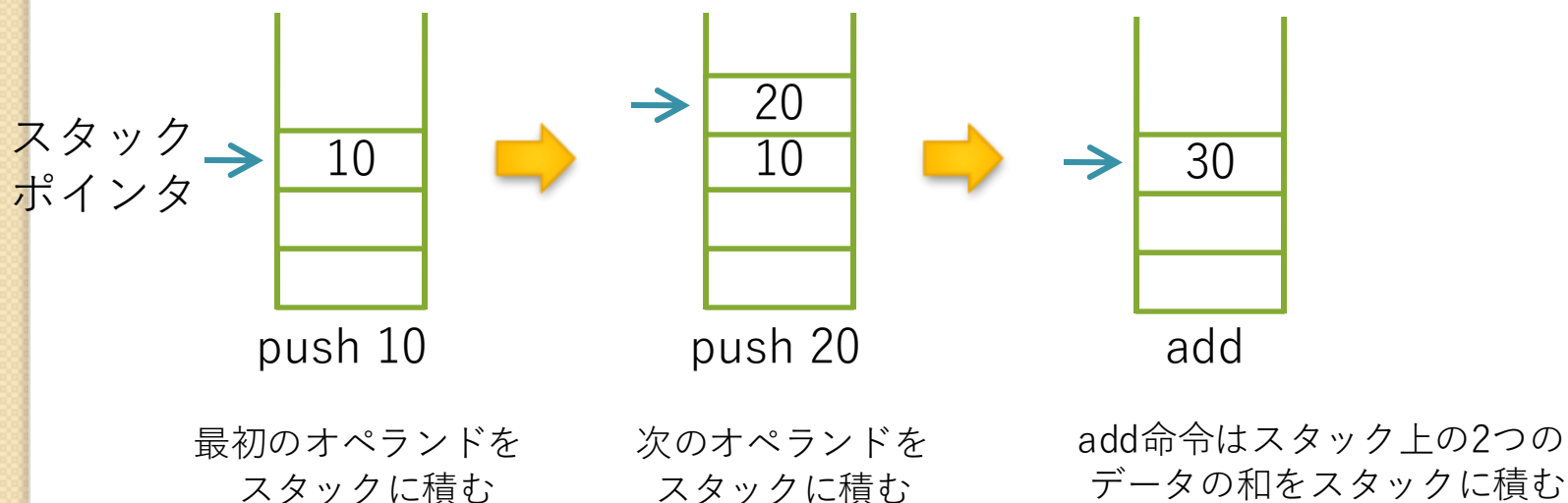


スタック上の
2つの値をかける



スタック機械

- スタック機械
 - 汎用レジスタが無い。→ 機械語命令にオペランドが無い。
 - 演算の対象や結果はすべて、スタック上に置く。
- 例：add は、 $\text{push}(\text{pop}() + \text{pop}())$ を計算する。
- 例：10+20の計算





後置記法(postfix notation)

- 後置記法 = 演算子をオペランドの後に書く.
 - 例: $10+20$ の後置記法は $10\ 20\ +$.
 - cf. $10+20$ は中置記法, $+ 10\ 20$ は前置記法.
 - 例: $(10+x)*y$ の後置記法は $10\ x\ +\ y\ *$.
- 別名, 逆ポーランド記法.
- 後置記法の利点:
 - スタック機械の演算順序を直接表現する.
 - 式を左から右に読む.
 - オペランドならスタックにプッシュする.
 - 演算子 (+や*) ならば, その演算子を実行する.

$10\ x\ +\ y\ *$



push 10
push x
add
push y
mul

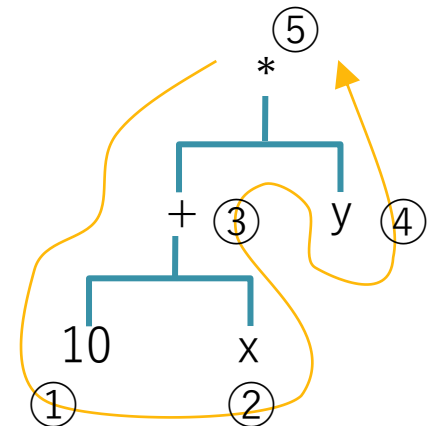


後順序(postorder)の木の訪問

- 抽象構文木から後置記法を得る方法.
 - 抽象構文木を後順序で訪問すれば良い.
- 後順序の訪問アルゴリズム（再帰的定義）
 1. まず子ノードに後順序で（左から右に）訪問する.
 2. 次の自分のノードを処理する.

C言語での記述例

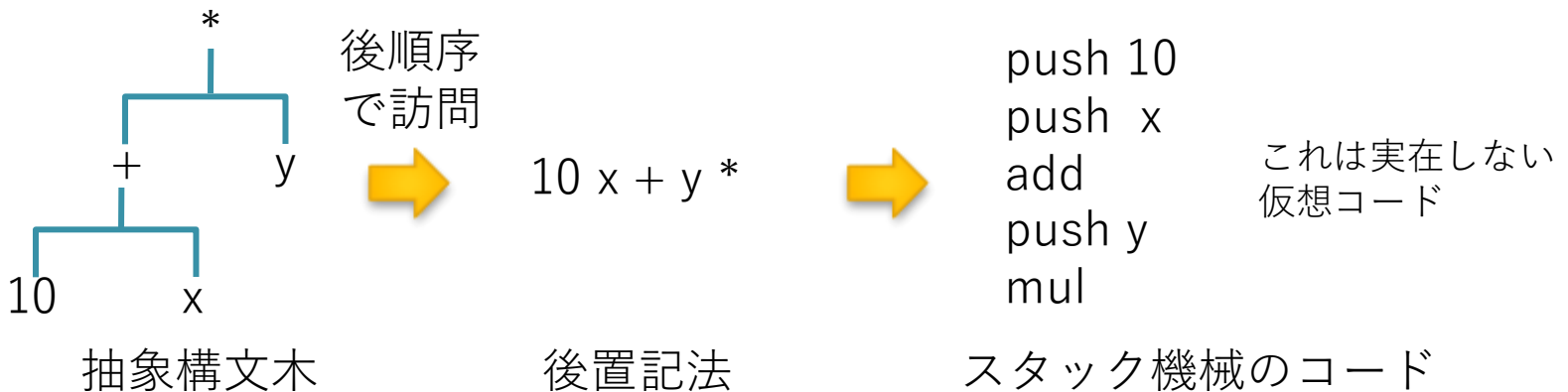
```
void postorder (struct AST *ast)
{
    int i;
    for (i = 0; i < ast->num_child; i++) {
        postorder (ast->child [i]);
    }
    printf ("%d¥n", ast->ast_type);
}
```





スタック機械：式のコード生成（１）

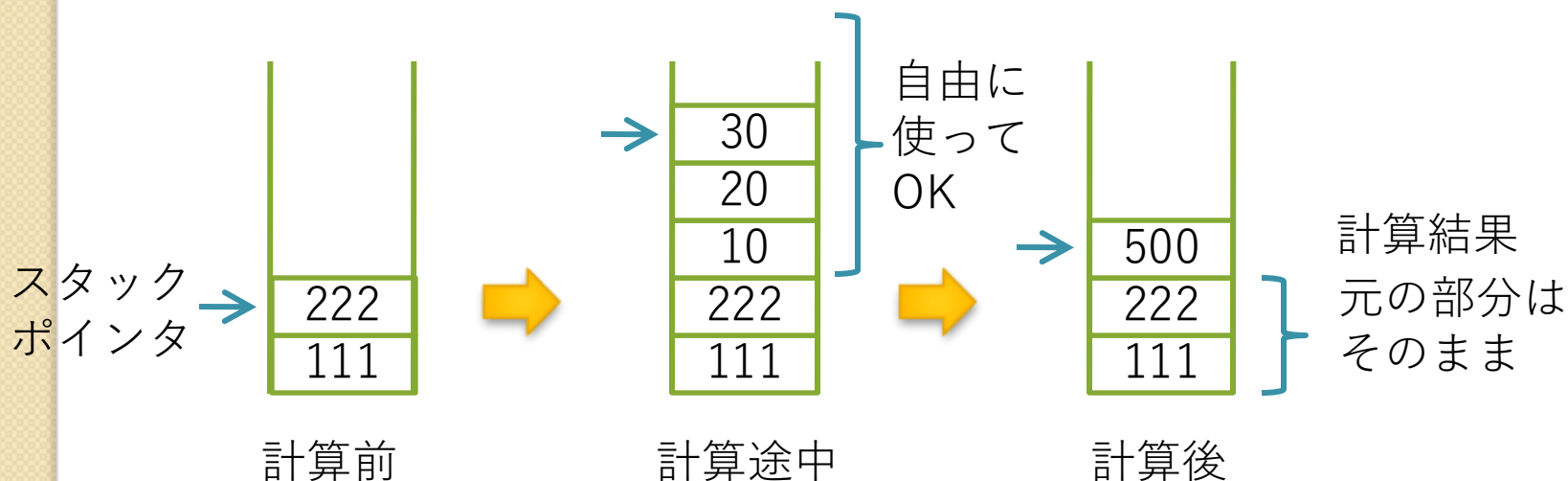
- スタック機械での式のコード生成：
 - 式の後置記法を左から右に読み、次の書き換えを行う。
 - 定数や変数→push命令. （例：push 10）
 - 演算子→演算子命令. （例：add）
- 例： $(10+x)*y$ のコード生成.





お約束：スタックの使い方

- 計算途中はいくらでもスタックを使ってよい。
 - ただし、最初のスタックポインタより上の部分に限る。
 - 計算終了後は元のスタック + 計算結果に必ずする。
- 例： $10*(20+30)$ の計算





スタック機械：式のコード生成（２）

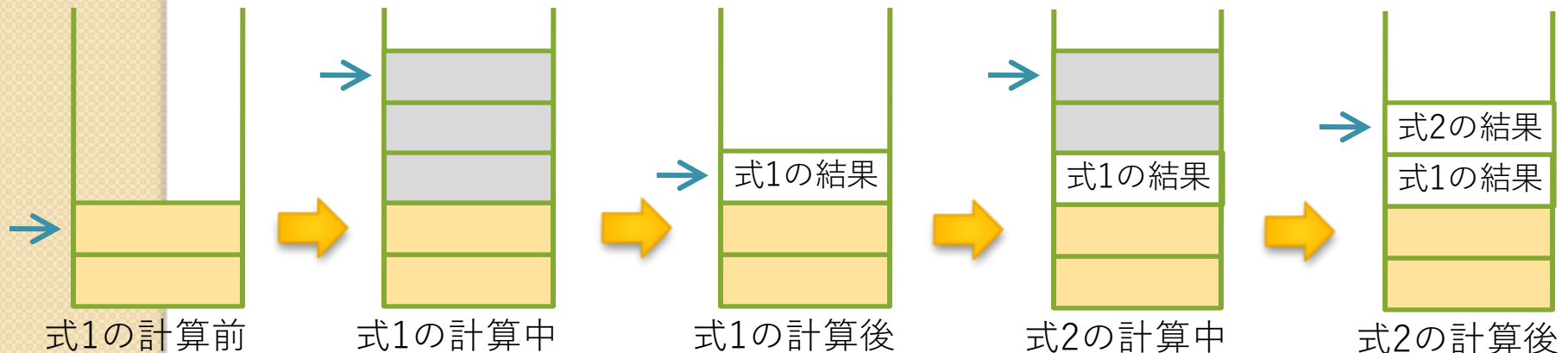
- 「式1+式2」のコード生成

式1 のコンパイル結果

式2 のコンパイル結果

add

- 式1や式2がどんなに複雑でも，これでうまくいく.
 - 理由：式2の実行前には，元のスタックに式1の計算結果をプッシュした状態になる（する）から．式2も同じ．





x86-64での式のコード生成

- 演算子の実行直前に、スタックからレジスタに値を転送する。
 - この方法なら、決してレジスタ不足にはならない。
- 「式1+式2」のコード生成（long型同士の場合）

式1 のコンパイル結果

式1の計算結果をスタックに積む

式2 のコンパイル結果

式2の計算結果をスタックに積む

```
popq   %rcx
popq   %rax
addq   %rcx, %rax
pushq  %rax
```

式2の計算結果をレジスタ%rcxにポップ
式1の計算結果をレジスタ%raxにポップ
%rcx と %rax の和を%rax に格納
%rax中の値をスタックにプッシュ

これは単純な方法。他にも方法あり。
GCCは最適化でもっと短いコードを生成。
その代わり、レジスタ割り付けが面倒。

x86-64では2つのオペランドが
同時にメモリ参照できない。
→レジスタへの転送必要。



x86-64での式のコード生成

- GCCのコード生成例

```
.data
.globl _x
.p2align 3
_x:
.quad 10

.text
.globl _main
.p2align 4, 0x90
_main:
pushq %rbp
movq %rsp, %rbp
movq _x(%rip), %rax
addq $999, %rax
popq %rbp
retq
```

```
long x = 10;
int main (void)
{
    return x + 999;
}
```

前ページの方法での生成例

```
movl _x(%rip), %rax
pushq %rax
pushq $999
popq %rcx
popq %rax
addq %rcx, %rax
pushq %rax
```

← 変数xや定数999をスタックに積まずに直接、レジスタに格納して演算。このため命令数が少なくて済む。



式のコード生成：変数（1）

- long型の「変数x」のコード生成. _xは大域変数xの格納場所のラベル

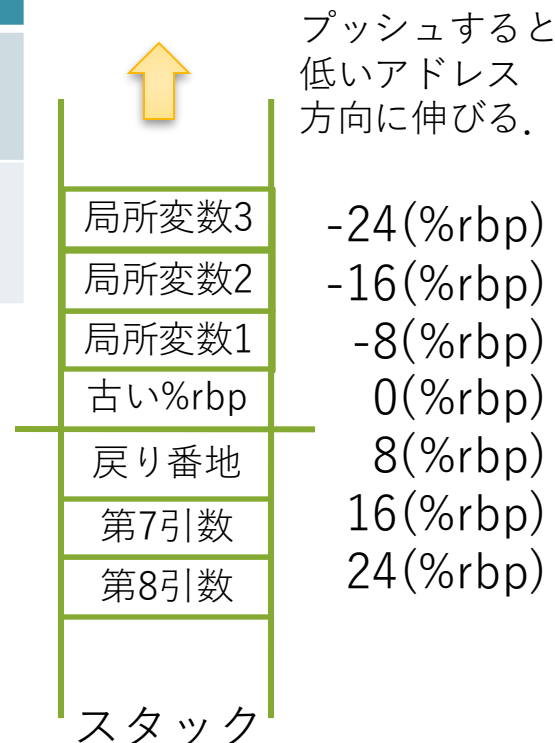
```
_x:  
    .quad 10
```

	左辺値	右辺値
大域変数	<code>leaq _x(%rip), %rax;</code> <code>pushq %rax</code>	<code>pushq _x(%rip)</code>
引数 局所変数	<code>leaq offset(%rbp), %rax;</code> <code>pushq %rax</code>	<code>pushq offset(%rbp)</code>

左辺値は
メモリアドレスを
スタックに積む

右辺値は
メモリの中身を
スタックに積む

offset はベースポインタからの
相対オフセット（何バイト離れているか）





代入演算子(=)のコード生成

- 「式1 = 式2」のコード生成（long型の場合）.

式2 の右辺値のコード

代入する値

式1 の左辺値のコード

代入先のアドレス

popq %rax

アドレスを%raxに代入

popq %rcx

代入する値を%rcx に代入

movq %rcx, 0(%rax)

%rcx の値を%raxが指すメモリ中に代入

pushq %rcx

代入する値をスタック上にも残す

- 代入する値をスタックトップに残してすることに注意.
 - 代入式の値は, 代入した値そのもの.
 - 例: `printf ("%d¥n", x = 999); /* 999 を表示 */`
- `x = y = z = 999` という式を評価するために必要.



x86-64での代入文のコード生成

- GCCのコード生成例

```
.data
.globl _x
.p2align 3
_x:
.quad 10

.text
.globl _main
.p2align 4, 0x90
_main:
pushq %rbp
movq %rsp, %rbp
xorl %eax, %eax
movq $999, _x(%rip)
popq %rbp
retq
```

```
long x = 10;
int main (void)
{
    x = 999;
}
```

前ページの方法での生成例

```
pushq $999
leaq _x(%rip), %rax
pushq %rax
popq %rax
popq %rcx
movq %rcx, 0(%rax)
pushq %rcx
```

生成方法は単純だが、
無駄な命令が多くなってしまう



位置独立コード (1)

- 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
- 絶対アドレスは使えない
 - 絶対番地を使うと要リロケーション（番地調整）→共有不可に
 - 相対アドレスと間接アクセスを使う
- 例：ラベル `_x` は相対アドレス（PIC中では）
 - `%rip` を起点とした相対アドレス

```
static long x = 999;  
int main (void)  
{ x = 888; }
```

.textセクション

```
movq $888, _x(%rip)
```

.dataセクション

```
.p2align 3  
_x:  
.quad 999
```



位置独立コード（２）：lldbで確認

```
% objdump -D a.out | more (出力略)
```

```
__text:
```

```
100000fa6: movq $888, 0x4f(%rip)
```

```
100000fb1: popq %rbp
```

```
% nm ./a.out
```

```
0000000100001000 d _x
```

$0x1000 - 0xfb1 = 0x4f$

0x100001000 と 0x100000fb1 は
実行時には違うアドレスかも。

（どのアドレスにロードされるかは実行時に決まるから）
でも、その差（0x4f）は常に一定

```
% lldb ./a.out
```

```
(lldb) b main
```

```
(lldb) r
```

```
1 static long x = 999;
```

```
2 int main (void)
```

```
-> 3 { x = 888; }
```

```
(lldb) reg read $rip
```

```
rip = 0x0000000100000fa6 a.out`main + 6 at foo.c:3:7
```

```
(lldb) memory read --format d 0x100001000
```

```
0x100001000: 999
```

```
(lldb) quit
```



位置独立実行可能ファイル

- 位置独立コードのみをリンクした実行可能ファイル
 - 共有ライブラリだけでなく，main関数のアドレスも変わる
 - 最近，PIEコンパイルがデフォルトのシステムが増えてきた
↑ セキュリティ向上のため
 - 実行するたびにコードのアドレスが変化→デバッグしにくい！
 - デバッガでは通常，ASLRは無効になってる．デバッグのしやすさのため
- ASLR (Address Space Layout Randomization)
 - テキスト，ヒープ，スタック，共有ライブラリなどのアドレスをランダム化
 - セキュリティ向上のため，こちらも良く使われている



ASLRとPIEの無効化：macOS

- lldb だと自動で無効化される
 - (lldb) settings set target.disable-aslr true がデフォルト
- -fno-PIE オプションでも無効化

ld: warning: -no_pie is deprecated when targeting new OS versions
という警告がでるが, a.out は作成可能

```
$ gcc -g -fno-PIE foo.c
$ ./a.out
0x100003f60, 0x100004010, 0x600000f4c030
$ ./a.out
0x100003f60, 0x100004010, 0x600002af0030
$ ./a.out
0x100003f60, 0x100004010, 0x600001b94030
```

アドレスが固定に

でもmallocの返り値は変わっちゃう



ASLRとPIEの無効化：Linux

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

再起動するたび

```
$ gcc -g -no-pie foo.c
```

```
$ ./a.out
```

```
0x401146, 0x7ffff7e1dc90, 0x4052a0
```

```
$ ./a.out
```

```
0x401146, 0x7ffff7e1dc90, 0x4052a0
```

```
$ ./a.out
```

```
0x401146, 0x7ffff7e1dc90, 0x4052a0
```

コンパイルするたび

mallocが返したアドレスも固定に

↓これはシステム全体のASLRを無効化し、セキュリティ上危険なのでデバッグが終わったら、元に戻すこと（1をセットする）

```
sudo sysctl -kernel.randomize_va_space=0
```

- Intel形式とAT&T形式の違い



ツールのオプション

- GCCのオプションで出力形式を選択可.
 - `-masm=intel`, `-masm=att` (デフォルト)
 - ただし, macOS では `-masm=intel` は未サポート.
- GNUアセンブラのアセンブラ命令で記法を選択可.
 - `.intel_syntax`, `.att_syntax` (デフォルト)
- objdump のオプションで出力形式を選択可.
 - `-M intel`, `-M att` (デフォルト)
 - AT&Tモードでは, `-M suffix` でオペランドサイズを示す接尾語 (b, w, l, q)を付加. (過剰につくのが難点)



ツールのオプション (使用例)

```
% gcc -S -masm=intel sub.c
```

sub.c

```
int sub (int a, int b) {  
    return a - b;  
}
```

sub.s

```
.intel_syntax noprefix  
.globl _sub  
.p2align 4, 0x90  
_sub:  
push rbp  
mov rbp, rsp  
mov dword ptr [rbp-4], edi  
mov dword ptr [rbp-8], esi  
mov esi, dword ptr [rbp-4]  
sub esi, dword ptr [rbp-8]  
mov eax, esi  
pop rbp  
ret
```


```
% objdump -d -M suffix sub.o  
0000000000000000 <_sub>:  
0: 55      pushq %rbp  
1: 48 89 e5 movq %rsp,%rbp  
4: 89 7d fc movl %edi,-0x4(%rbp)  
7: 89 75 f8 movl %esi,-0x8(%rbp)  
a: 8b 75 fc movl -0x4(%rbp),%esi  
d: 2b 75 f8 subl -0x8(%rbp),%esi  
10: 89 f0    movl %esi,%eax  
12: 5d      popq %rbp  
13: c3      retq
```



AT&T形式とIntel形式の違い（１）

- オペランドの順序が逆

	AT&T形式	Intel形式
オペランドの順序	addq \$4, %rax	add rax, 4



- 接頭語 (prefix) の記号の有無

	AT&T形式	Intel形式
即値オペランド	pushq \$ 4	push 4
レジスタオペランド	pushq % rbp	push rbp
絶対ジャンプ（コール）のオペランド	jmp * 0x100	jmp [0x100]



AT&T形式とIntel形式の違い（2）

- オペランドサイズの指定方法

サイズ	型	AT&T形式	Intel形式
1バイト	byte	mov b \$2, (%rbx)	mov BYTE PTR [rbx], 2
2バイト	word	mov w \$2, (%rbx)	mov WORD PTR [rbx], 2
4バイト	long	mov l \$2, (%rbx)	mov DWORD PTR [rbx], 2
8バイト	quad	mov q \$2, (%rbx)	mov QWORD PTR [rbx], 2

- 即値形式のfar jmp/call/ret の表記

	AT&T形式	Intel形式
jump	ljmp \$segment, \$offset	jmp far segment:offset
call	lcall \$segment, \$offset	call far segment:offset
return	lret \$stack-adjust	ret far stack-adjust



AT&T形式とIntel形式の違い（3）

- 機械語命令の二モニック
 - 基本的に同じ。ただし例外あり。
 - 例外1：AT&T形式にはオペランドサイズを示す接尾語あり。
 - 例：mov**l**, mov**w**, mov**b**
 - 例外2：AT&T形式のlcall/ljmpはIntel形式ではcall far/jmp far
 - 例外3：変換命令の二モニック（以下参照）

AT&T形式	Intel形式	説明
cblw	cbw	%al (byte)→%ax (word)の符号拡張
cwtl	cwde	%ax (word)→%eax (long)の符号拡張
cltq	cdqe	%eax (long)→%rax (quad)の符号拡張
cwtd	cwd	%ax (word)→%dx:%ax (long)の符号拡張
cltd	cdq	%eax (long)→%edx:%eax (quad)の符号拡張
cqto	cqo	%rax (quad)→%rdx:%rax (octet)の符号拡張



AT&T形式とIntel形式の違い（４）

- 例外4：符号拡張・ゼロ拡張を伴うmov命令のニモニック。
 - AT&T形式では、2つのサイズを指定する接尾語（左下の表）を、それぞれ **movs..** と **movz..** に埋め込む。
 - Intel形式では、**movsx** と **movzx** というニモニックを使う。

AT&T形式 の接尾語	説明
bl	byte → long
bw	byte → word
wl	word → long
bq	byte → quad
wq	word → quad
lq	long → quad

例

AT&T形式	Intel形式
movsbq %al, %rdx	movsx rdx, al
movslq %eax, %rdx	movsxd rdx, eax

movzrq は存在しない。
%eaxに値を入れれば
%raxの上位32ビットは
0クリアされるから