

アセンブリ言語

インラインアセンブラ

情報工学系
権藤克彦



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



インラインアセンブラの構文は コンパイラ依存

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

GCC

```
int main (void)
{
    __asm {
        nop
    }
}
```

VC, ICC

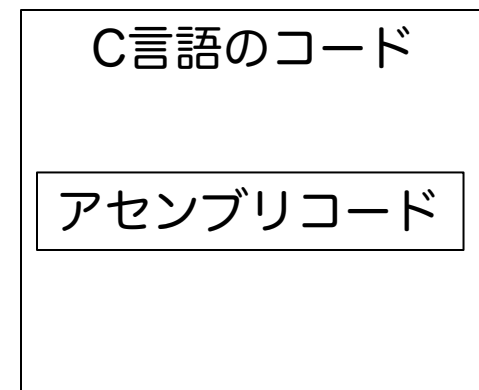
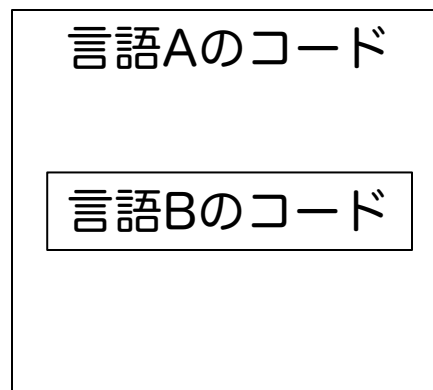
インラインアセンブラの構文の詳細は
コンパイラのマニュアル（例：gcc info）を参照する。



埋め込み型言語

- 埋め込み型言語 (embedded language)
 - ある言語のプログラム中に記述できる, プログラミング言語.
- 例
 - C言語中のアセンブリコード. ← インラインアセンブリコード
 - HTML中のPHPやJavascript.
 - COBOL中のSQL.
 - Yacc/Lex中のC.

言語Bが
埋め込み型言語





インラインアセンブラは機械語命令をチェックしない

- 間違った機械語命令でも、そのまま出力。
 - 正しい機械語命令でもプログラマの意図に反した出力かも。
- ↓
- 人間がチェックすべき
 - アセンブル結果（や逆アセンブル結果）を目視でチェック。

```
int main (void)
{
    asm ("foo bar");
}
```

存在しない
機械語命令

gcc -S



```
.text
.globl _main
_main:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $8, %esp
    foo bar
    leave
    ret
```

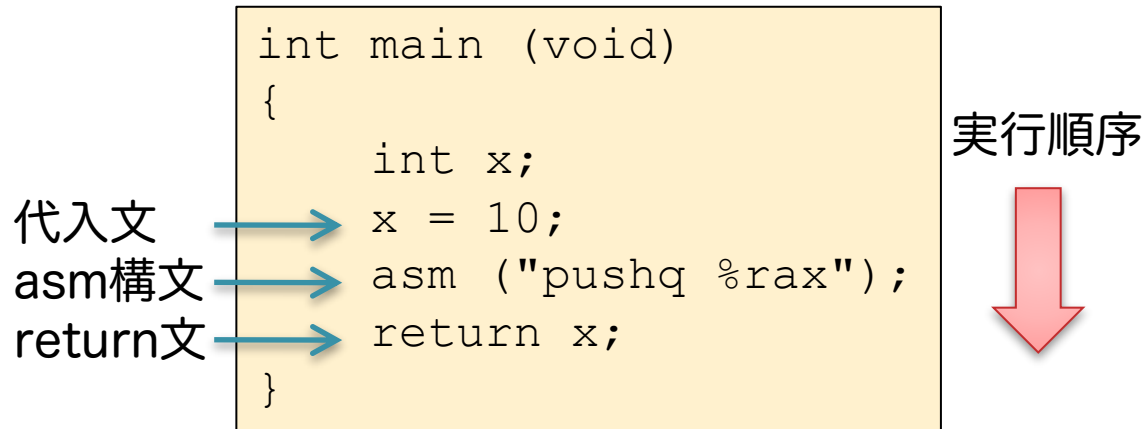
そのまま
埋め込まれてる

最近のmacOSのgcc (LLVM)では
エラーを出力してくれる



asm構文は「文」か「外部宣言」（１）

- 文や外部宣言として，asm構文を記述可能.
- 文の実行順序に従って，asm構文も実行される.



※ただし，最適化器の影響（後述）を
考えない場合.



asm構文は「文」か「外部宣言」 (2)

- 文や外部宣言として, asm構文を記述可能.
- asm構文で関数や外部変数を定義できる.

```
asm (".globl _x\n"  
     ".data\n"  
     ".p2align 2\n"  
     "_x:\n"  
     ".long 111");  
extern int x;  
int main (void)  
{  
    return x;  
}
```

}
int x = 111;
という変数定義に相当

※この使い方はあまりしない
(別ファイル foo.s に書けば良いので)



asm構文中の改行

- OK: 文字列定数に分割して改行.
- NG: 文字列定数中で改行.

OK : セミコロンで区切る.

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

OK : ¥nで区切る.

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

OK : 文字列定数に分割して改行.

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

NG : 文字列定数中で改行.

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



コンパイルエラー
になる



複数のasm構文は好ましくない

- なるべく1つのasm構文にまとめて書く。
 - volatile (次スライド参照) も指定する.
- 最適化の影響を防ぐため。
 - 最適化の結果, asm構文の順番が入れ替わったり, asm構文の間に別のコードが入ることがある.

複数のasm構文は好ましくない

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

1つのasm構文にまとめて, volatileもつける

```
int main (void)
{
    asm volatile ("nop¥n¥t"
                  "nop¥n¥t"
                  "nop");
}
```



asm volatile構文

- asm volatile構文
 - 最適化による, asm構文の移動・変更・消去を防ぐ.
 - ただし完全ではない. → 逆アセンブルで要チェック.
- __asm__, __volatile__
 - 名前の衝突を防ぐため, asmの代わりに__asm__を使用可. 同様に volatileの代わりに __volatile__を使用可.
 - __ はアンダースコア記号(_)が2つ並んでいる.

予約識別子名 (予約語ではない)

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

```
int main (void)
{
    __asm__ __volatile__ ("nop");
}
```

どちらもOK



最適化

- 最適化(optimization)
 - コンパイラ最適化の略.
 - より高速に, より少ないメモリで動作するようにコードを改善すること. (外部から見た動作は変更せずに)
 - より最適に近くするだけ. 本当の「最適」は難しい.
- 最適化はコードを移動・変更・消去する.



最適化（例）

-Os はバイナリサイズを小さくする

- GCCでは -Oオプションで最適化.
 - -O (-O1), -O2, -O3, -O0, -Os
 - 他にも数多くのオプションあり.

最適化なし : gcc -S

```
.text
.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
```

最適化あり : gcc -O -S

```
.text
.globl _add5
.p2align 4, 0x90
_add5:
pushq   %rbp
movq    %rsp, %rbp
leaq    5(%rdi), %rax
popq    %rbp
retq
```



拡張asm構文（１）

- 拡張asm構文を使うと，機械語命令のオペランドにC言語の式を指定できる．

拡張asm構文の例

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

addrはC言語の変数（式）



レジスタ%rspの値を変数addrに代入

拡張asm構文ではコロン(:)
以降にオペランドを記述する



拡張asm構文（2）

- 単純なasm構文との主な違い.
 - レジスタ名は%rspではなく, %%rspと書く.
 - %0はaddrに対応. 一般に%nはn番目のオペランドに対応.
 - コロン(:)以降のオペランドは制約とC言語の式からなる.
 - "=m" は制約. =が出力, mがメモリを意味する.
 - addr はC言語の式. %0 に対応.

レジスタ%esp
を意味する.

出力先が変数addrに
なるように%0を展開.

```
asm volatile ("movq %%rsp, %0" : "=m" (addr));
```



インライン
アセンブラが処理

```
movq %rsp, -8(%rbp)
```

addrのアドレスを示す
メモリオペランド

=は出力, mはメモリを
意味する制約.



拡張asm構文の使用例：rdtsc命令

- タイムスタンプカウンタ
 - Pentium以降のx86-64が持つ64ビットのカウンタ.
 - CPU起動時からの全サイクル数をカウントして保持.
- **rdtsc命令**
 - タイムスタンプカウンタを読み, `%edx:%eax`に格納.

```
#include <stdio.h>
#include <stdint.h>
int main (void)
{
    uint64_t hi, lo;
    asm volatile ("rdtsc":"=a" (lo), "=d" (hi));
    printf ("%llx¥n", (hi << 32) | lo);
}
```

rdtsc

```
movq %rax, -16(%rbp)
movq %rdx, -8(%rbp)
```

変数 hi と lo

64ビット符号なし整数
なので%dではダメ.

補足：コンパイラの制約の使い方.
(1) %0などの展開.
(2) 上のmovq命令等の追加.
(グルーコード, つなぎコード)



拡張asm構文の形式

- コロン(:)で区切って、入出力オペランドなどを指定.
- 制約が機械語命令オペランドのアドレッシングモードを決める.
- インラインアセンブラは必要に応じてデータ転送コードを追加.

拡張asm構文の形式

```
asm [volatile] (命令テンプレート  
                [: 出力オペランド列  
                [: 入力オペランド列  
                [: 破壊レジスタ列 ]]  
                );
```

[] は「省略可能」の意味.



入出力オペランドの形式

- 「制約とC言語の式」をカンマ(,)で区切って並べる.
- 制約で指定できる文字は後述.

入力（出力）オペランドの形式

`["制約" (C言語の式) [, "制約" (C言語の式)]*]`

`[]` は「省略可能」の意味.

`[]*` は「0回以上の繰り返し」の意味.



拡張asm構文の形式（例）

xに対応

yに対応

zに対応

```
asm ("movl %1, %%eax; addl %2, %%eax; movl %%eax, %0"  
    : "=r" (z)  
    : "m" (x), "m" (y)  
    : "%eax");
```

← 出力オペランド列

← 入力オペランド列

← 破壊レジスタ列

命令テンプレート

コロン (:) で区切る
カンマ (,) で区切る
制約
C言語の式

- 制約rはレジスタオペランドの要求を意味する。
- 出力オペランドのCの式は左辺値でなくてはならない。
 - 左辺値=代入文の左辺に出現できる式。（例：x, *p, p->a）
- 破壊レジスタの指定は，例えば%eaxレジスタの値が変更されることをコンパイラに伝える。



制約（１）

入出力を指定する制約（１文字目に指定）

制約	説明
=	オペランドは出力専用.
+	オペランドは入出力.
(指定無し)	オペランドは入力専用.

プラットフォーム非依存の主な制約

制約	説明
r	オペランドはレジスタ
m	オペランドはメモリ
i	オペランドは整数即値
g	オペランドの制約無し
&	オペランドは早期破壊レジスタ
0	マッチング制約. 1～9も同じ.

後述



制約m（メモリ）とr（レジスタ）の違い

メモリ制約(m)を指定

オペランドはメモリ参照に展開

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

```
movq %rsp, -8(%rbp)
```

レジスタ制約(r)を指定

オペランドはレジスタ参照に展開

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

```
movq %rsp, %rax
movl %rax, -8(%rbp)
```



制約（２）

x86-64用の制約（レジスタ）

制約	説明
a	%rax, %eax, %ax, または %al
b	%rbx, %ebx, %bx, または %bl
c	%rcx, %ecx, %cx, または %cl
d	%rdx, %edx, %dx, または %dl
A	制約 d と a
q	制約 a, b, c, または d
D	%rdi, %edi, または %di
S	%rsi, %esi, または %si



制約 (3)

x86-64用の制約 (定数)

制約	説明
I	範囲0～31 (32ビットシフト用)
J	範囲0～63 (64ビットシフト用)
O	範囲0～127
N	範囲0～255 (in/out命令用)
K	範囲-128～127 (符号ありimm8用)
L	0xFF, または 0xFFFF
M	0, 1, 2, または 3 (lea命令のシフト用)



入出力の制約

- 読み書きするオペランドは（**=**ではなく）**+**を指定.
- マッチング制約**を使って指定しても良い.

出力専用 汎用レジスタ

```
asm ("movq %1, %0": "=r" (b): "r" (a));
```

b = a;

入出力

```
asm ("addq %1, %0": "+r" (b): "r" (a));
```

b += a;

b += a;

```
asm ("addq %1, %0": "=r" (b): "r" (a), "0" (b));
```

マッチング制約.


%0と同じオペランドであることを示す



早期破壊オペランド制約(&) (1)

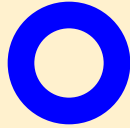
- GCCの仮定
 - 「命令テンプレートでは、入力オペランドをすべて参照してから、出力オペランドに代入している。」
- この仮定が崩れる場合は、**制約&の指定が必要**.
 - &は「同じレジスタに割り当てるな」
 - cf. マッチング制約は「同じレジスタに割り当てろ」

```
#include <stdio.h>
int main (void)
{
    int a = 20, b;
    asm ("movl $10, %0¥n"
        "addl %1, %0"
        : "=r" (b) : "r" (a));
    printf ("b = %d¥n", b);
}
```



%1の参照より先に%0を破壊

```
#include <stdio.h>
int main (void)
{
    int a = 20, b;
    asm ("movl $10, %0¥n"
        "addl %1, %0"
        : "=&r" (b) : "r" (a));
    printf ("b = %d¥n", b);
}
```



b=10;
b+=a;



早期破壊オペランド制約(&) (2)

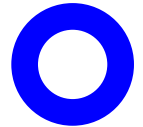
```
#include <stdio.h>
int main (void)
{
    int a = 20, b;
    asm ("movl $10, %0\n"
        "addl %1, %0"
        : "=r" (b) : "r" (a));
    printf ("b = %d\n", b);
}
```



```
movl -16(%ebp), %eax
movl $10, %eax
addl %eax, %eax
```

%eaxの値(20)が破壊されている。

```
#include <stdio.h>
int main (void)
{
    int a = 20, b;
    asm ("movl $10, %0\n"
        "addl %1, %0"
        : "&r" (b) : "r" (a));
    printf ("b = %d\n", b);
}
```



```
movl -16(%ebp), %eax
movl $10, %edx
addl %eax, %edx
```



破壊レジスタ宣言 (1)

- 入出力オペランドとは関係ないレジスタを命令テンプレート中で破壊するなら、破壊レジスタ列でそのレジスタを宣言する.
- 破壊レジスタ宣言すると、GCCはそのレジスタを入出力オペランドに割り当てない.
- 特殊な破壊レジスタ宣言：
 - "cc" (condition code) フラグレジスタを破壊.
 - "memory" メモリを破壊.

```
#include <stdio.h>
int main (void)
{
    int a = 10, b;
    asm ("movl $20, %%eax\n"
        "movl %1, %0"
        : "=r" (b) : "r" (a) : "%%eax");
    printf ("b = %d\n", b);
}
```

破壊レジスタ
宣言



破壊レジスタ（２）

- 破壊レジスタ宣言すると，GCCは（必要なら）そのレジスタの退避・回復コードも生成する．

```
int main (void)
{
    int a = 10;
    asm ("movl %0, %%ebx"
        :: "r" (a) : "%ebx");
}
```

pushl %rbx

movl \$10, -12(%ebp)

movl -12(%ebp), %eax

movl %eax, %ebx

popl %rbx

%rbxの退避

%rbxの回復

アセンブリコードは大幅に省略

%ebxはcallee-saveレジスタなので，
%ebxを破壊レジスタと宣言すると，
GCCは%rbxの退避・回復コードも生成．
上位32ビットも変更の可能性があるから．



練習問題（１）

- 拡張asm構文を使って，レジスタ%rbpの値を調べるプログラムを書こう．
- また，デバッガを使ってプログラムの動作を確認しよう．



解答例 (1)

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

```
% gcc -g asm-rbp.c
% lldb ./a.out
(lldb) b main
(lldb) run
-> 5    asm volatile ("movq %%rbp, %0": "=m"(rbp));
(lldb) s
-> 6    printf ("rbp = %p¥n", rbp);
(lldb) register read rbp
      rbp = 0x00007ffefbfff750
(gdb) c
rbp = 0x7ffefbfff750
(lldb) quit
```

デバッガ出力の結果と一致



練習問題（２）

- 拡張asm構文を使って、レジスタ%rflagsの値を調べるプログラムを書こう.
- ヒント：movq命令では%rflagsの値を読めないので、pushfq命令とpopq命令を使おう.



解答例（２）

```
#include <stdio.h>
int main (void)
{
    void *rflags;
    asm volatile ("pushfq; popq %0"
                  : "=m"(rflags)::"memory");
    printf ("rflags = %p¥n", rflags);
}
```

pushfqが%rflagsの値をスタックに積む（＝メモリを破壊する）ので、"memory" が必要.



練習問題（3）

- 拡張asm構文を使って、レジスタ%ripの値を調べるプログラムを書こう.
- ヒント：movq命令では%ripの値を読めないので、leaq命令を使おう. 以前はcall命令が戻り番地をスタックに積むことを利用していた.



解答例（3）

```
#include <stdio.h>
int main (void)
{
    void *rip;
    asm volatile ("leaq 0(%%rip), %0": "=r"(rip));
    printf ("rip = %p¥n", rip);
}
```

```
#include <stdio.h>
int main (void)
{
    void *rip;
    asm volatile ("call 1f; 1: popq %0": "=m"(rip)::"memory");
    printf ("rip = %p¥n", rip);
}
```

call命令が戻り番地をスタックに積む（＝メモリを破壊する）
ので、"memory" が必要。