



アセンブリ言語

プログラム課題：電卓コンパイラ



情報工学系
権藤克彦



概要

- 電卓コンパイラを作り，ソースコードとレポートを提出。
 - C言語を使用.
 - 情報工学科電子計算機室（CSC）での動作が基本.
 - 次のファイル中のテストケースを使う.
 - testcase*.txt （テストケースのデータ）
 - test*.csh （テスト実行用のシェルスクリプト）
- 〆切は11月22日（金） 17:00.
 - プログラムは未完成でも提出可（レポートをきちんと書く）
 - 締切厳守. 遅刻レポートは一切受け取らない.
 - 提出締切時刻ぎりぎりを狙わないこと. 1秒でも遅れたら不受理.
 - T2SCHOLA上で提出し，レポート受理もT2SCHOLA上で確認.
 - トラブル時はメールかSlackで連絡



入力と同じ計算をするアセンブリコードを出力.

電卓コンパイラ

- 内容：電卓コンパイラ（Cプログラム）を作成。
 - 入力＝電卓の入力（の文字列表現）.
 - 出力＝計算を実行するアセンブリコード.

電卓コンパイラ

1+2x3=

入力



```
.data
L_fmt:
    .ascii "%d¥n¥0"
.text
.globl _main
_main:
    pushq %rbp
    movq %rsp, %rbp
    movl $0, %eax
    movl $0, %ecx
    movl $0, %ebx
    imull $10, %ebx
(略)
```



電卓

```
% gcc foo.s
% ./a.out
9
```

コンパイル&実行

出力.
1+2x3を計算する
アセンブリコード.



電卓コンパイラと電卓の仕様（ポイント）

- カッコは使えない． 演算の優先順位無し．
 - 例： $1+2*3=$ の答えは，7ではなく，9が正しい．
- 電卓はアセンブリコード中でprintfを呼び出して計算結果を標準出力に（最後の結果だけ）出力．
- 電卓コンパイラは複数の数字キーの入力をまとめてアセンブリコードに出力してはダメ．
 - 例：「1」「2」「3」という入力を123と出力してはダメ．
 - Cコード中で123を計算するのもダメ．
 - 例：「-」「3」という入力を-3と出力してはダメ．
- 電卓はオーバーフローしたら「E」を表示して終了．
 - 32ビット符号あり整数で．（浮動小数点数は扱わない）
- 電卓はメモリー機能をサポートする．
 - M+, M-, CM, RM.



電卓の仕様

- 電卓の動作を細かく精密に説明するのは面倒 & 退屈.
- この課題では **テストケース式** を電卓の仕様とする.
- テストケース
 - プログラムの入力と, その入力に対する正しい出力のペア.

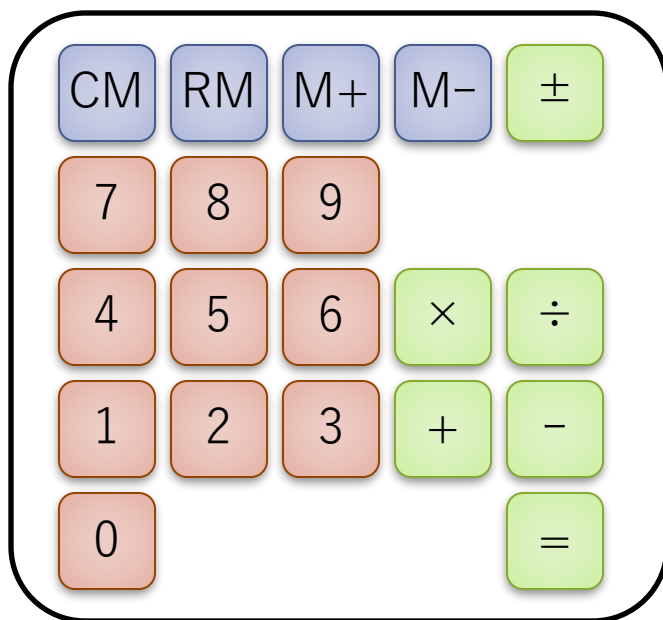
testcase1.txt

```
'1+2=',3  
'1+2*3=',9  
'12+34*56=',2576  
'12S+34=',22  
'2147483648=',E  
'2147483648S=',E  
'2147483647+1=',E
```

「1+2=」という入力に対して,
「3」と出力するのが正しいことを示す.
(この書式はこの課題が勝手に定めたもの)



想定する電卓のキー



電卓キーの文字表現

CM	C (clear memory)
RM	R (recall memory)
M+	P (memory plus)
M-	M (memory minus)
±	S (sign inversion)
×	*
÷	/

それ以外はそのまま



計算例（１）：普通の計算

$$1 + 2 =$$

3

$$1 + 2 \times 3 =$$

9

$$1 + 2 \times 3 + 4 \times 5 + 6 =$$

2576



計算例（２）：符号キー

$$1 \ 2 \ \pm \ + \ 3 \ 4 \ =$$

22

$$1 \ 2 \ \pm \ \pm \ + \ 3 \ 4 \ =$$

46

$$1 \ \pm \ 2 \ \pm \ + \ 3 \ 4 \ =$$

46

$$1 \ 2 \ + \ \pm \ 3 \ 4 \ =$$

46



計算例（３）：演算キー

連続する演算キーは後のものが有効.

1 ÷ + 2 =

3

1 ÷ × + 2 =

3

無効な演算キーはアセンブリコードに出力しなくて良い.



計算例（３）：オーバーフロー

2 1 4 7 4 8 3 6 4 7 + 1 = E

32ビット符号あり整数として、
オーバーフローは計算結果を E と表示.



計算例（４）：メモリ機能

1 0 × 2 M+ 4 0 ÷ 4 M+ RM 30

1 0 × 2 + 4 0 ÷ 4 = 15

上は、 $(10 \times 2) + (40 / 4)$ を計算.

下は、 $((10 \times 2) + 40) / 4$ を計算.

CM メモリの値を0にセット.

RM メモリの値を現在の計算結果にセット.

M+ 現在の計算結果をメモリの値に加算.

M- 現在の計算結果をメモリの値から減算.



計算例（５）：変な入力

1	0	×	M+	2	=	2
1	0	×	M+	2	RM	10

変なキー入力でもエラーとはしない.

（どう動作するかは電卓の実装による.）

このような入力に対する動作は（エラー以外の）任意とする.
テストケースには含めない.



ポイント（１）：４つの記憶領域（変数）

変数名	説明
num	現在，入力中の数値を保持.
acc	現在の計算結果を保持.
mem	メモリ機能で記憶する値を保持.
last_op	最後に見えた演算子を保持.

こう初期化する.
最初の入力数値の
扱いを楽にするため.

最初の入力

入力	num	acc	last_op
1	1	0	+
2	12	0	+
*	12	12	*
3	3	12	*
=	3	36	=

最後の入力

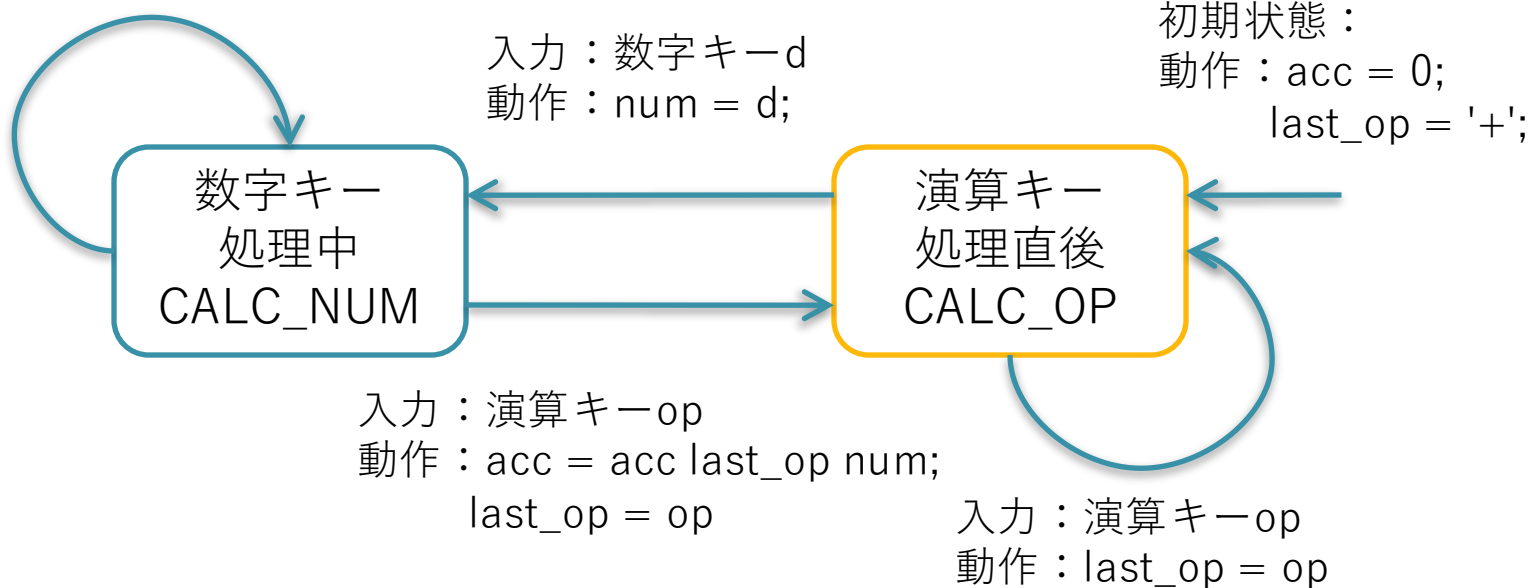


ポイント（２）：状態遷移図

1 2 + 3 4 * 5 6 =

入力：数字キーd

動作：num = num * 10 + d;



簡単のため，符号キーと
メモリキーは無しでの図。



電卓コンパイラ レベル1：calc1.c

- ここまで説明した電卓を実装する.
- ただし
 - オーバーフローは扱わなくて良い.
 - 乗算命令(imul)や除算命令(idiv)を使って良い.
- 先に calc0.c を作ることをお勧め.
 - インタプリタ版の電卓. アセンブリコードを出力せず, その場で入力に対する計算を行って結果を出力する.

```
% gcc calc1.c
% ./a.out '1+2*3=' > foo.s
% gcc -o b.out foo.s
% ./b.out
9
%
```

```
% gcc calc0.c
% ./a.out '1+2*3='
9
```



電卓コンパイラ レベル2：calc2.c

- オーバーフローが発生したら、Eを表示して、実行を終了させる。
 - Eを表示するには、アセンブリコードからprintf() を呼び出す。
 - 実行終了にはライブラリ関数 exit() を呼び出す。
 - 終了ステータスは何でも良い。（通常はゼロ以外に設定）
- ゼロ割の場合も同じエラー処理をする。
 - 除算前に除数が0か否かをチェックする。
- オーバーフローの検査コードが、オーバーフローのフラグを変更しないように注意。



注意：idiv は浮動小数点例外を出す

- idivが浮動小数点例外を出す場合
 - 0割りをした時 （こっちは扱う）
 - 商 $> 2^{31}-1$ or 商 $< -2^{31}$ の時
 - つまり、商が32ビット符号あり整数の範囲を超えた場合
 - きちんと%eaxを符号拡張して、%edxを初期化して下さい



電卓コンパイラ レベル3：calc3.c

- 乗算命令（mul, imul）や除算命令（div, idiv）を使わずに，乗算と除算を行うように変更する.
 - 注：数値入力を計算する x10 でも，乗算命令禁止
- 加減算やシフト命令を組み合わせて実現する.
- （簡単のため）オーバーフローやゼロ割のチェックは不要.

課題コード	行数
calc0.c	148
calc1.c	176
calc2.c	213
calc3.c	327

参考：権藤が作った
コード量



加算とシフトで，乗算を計算

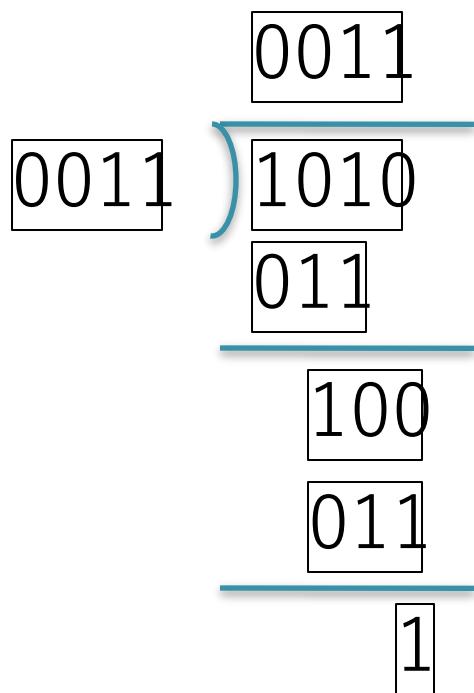
$$\begin{array}{r} 0101 \\ 1011 \\ \hline 0101 \\ 0101 \\ 0000 \\ + 0101 \\ \hline 0110111 \end{array}$$

5*11=55の計算

- ヒント
 - ロータート命令 (rcr) を使って，かける数を下位ビットから1ビットずつ処理.
 - かけられる数は左に1ビットずつシフトする.
 - 負の数を扱うには，まず絶対値同士の乗算を求めておき，最後に符号を調整.
 - 注：この方法だと，表現できる最小の負数を扱えない．これは考えなくて良い．（符号なし整数を使えば扱える）
 - $m*n$ を「 m を n 回足す」で実現すると実行速度が遅すぎるので禁止



減算とシフトで，除算を計算



• ヒント

- 左シフト命令 (shl) を使って，割られる数を上位ビットから1ビットずつ別レジスタに格納.
- その別レジスタ \geq 割る数ならば，商に1をたてて引き算する.
- 負の数の処理は前ページと同じ.
 - 最小の負数の扱いも同じ.

10/3=3...1の計算



%rsp の 16バイト境界制約に注意

- call命令の実行時に
「%rspの値は16の倍数」を満たすことが必要.
- %rsp の値を適宜, 調整する.
 - 分かっているけど, うっかり間違えやすい.

```
pushq %rbp
movq %rsp, %rbp
# ここで16の倍数
...
popq %rbp
retq
```

return address (%rip)と
旧%rbp の合計で 16バイト

```
# ここで16の倍数
leaq L_fmt(%rip), %rdi
movslq %eax, %rsi
call _printf
```

たまたまこの例では調整が不要だった.
第6引数まではレジスタ渡しのため.
%rdi = 第1引数
%rsi = 第2引数



提出方法

- T2SCHOLAの課題から提出.
- 提出物一式（後スライド参照）をtarで1つのファイルに固めて，アップロードすること.
 - 学籍番号と同じディレクトリを作り，そこに送るファイルをすべてコピーする.
% mkdir 23B12345
% cp ファイル名 23B12345
 - tar で1つのファイルにまとめる.
% tar cvzf 23B12345.tgz 23B12345

このファイルを
アップロード.



提出物一式の確認

- 固めたファイルが正しく解凍できるかチェックする

```
% cp 23B12345.tgz /tmp      別ディレクトリにコピー
% cd /tmp
% tar xvzf 23B12345.tgz     解凍・展開
% cd 23B12345
% ls
calc1.c calc2.c calc3.c report.pdf
%
```



提出物

ファイル分割しない

- プログラム（ソースコードのみ，各ファイル1つで）
 - calc1.c：電卓コンパイラ レベル1
 - calc2.c：電卓コンパイラ レベル2
 - calc3.c：電卓コンパイラ レベル3
- レポート
 - report.txt, report.md または report.pdf
（図が必要な場合は PDF形式の report.pdf中に含めること）

コンパイルにオプションが必要な場合は，
Makefile を書いて提出することを推奨．



提出してはいけないもの

- バイナリファイル (a.out, *.o)
- 昔のソースコード, バックアップファイル (*~)
- winmail.dat (Outlook 使いの人は要注意)
- その他, レポートに関係ないファイル.



レポートに書く内容

無駄にダラダラ長い文章は減点.
ページ稼ぎ禁止.
ソースコード引用は最低限に.
内容が少ないものも減点.

- 次のことをアピールする文章を（簡潔に）書く.
 - 「私はこの課題を深く理解して、しっかり取り組んだ」
 - 「私は様々な試行錯誤をし、様々な工夫もした」

情報少ないと「分かってないな」とこちらは判断

- 書くことの例
 - 設計上の取捨選択とその理由・結果.
 - より良いテストケースの提案と説明
 - 何をどこまで作ったか. 既知のバグ（もしあれば）.
 - 改良や拡張すべき点とその方法.
 - 議論（考察）.
 - 感想（採点外）.

単に「プログラムをきれいに直したい」と書くだけでは、議論として意味がない.



レポートはラブレターと同じ💖

- 短いと
 - 読んでも全然面白くない
 - 本当にこの課題，ちゃんと（自分で）考えてやったの？
- 長いと
 - 読むのが大変，でも中身が薄いし面白くない
 - ページ稼ぎだけで，課題への愛が感じられない
- 面白くするには
 - 読みやすく簡潔な説明，後で詳細を書く
 - 新聞の「見出し」と「本文」
 - 図表（手書きでもOK），例を使う
 - 卒論の執筆・発表や就活のESで非常に大事なテクニック



入力と出力のお約束

- こちらで自動テストするために必要.
- 守らなかった場合は減点します.
- 電卓コンパイラは入力を `argv[1]` から受け取る.
- 電卓コンパイラはアセンブリコードを標準出力に出力する.
- 電卓は標準出力に計算結果（と改行）だけを出力する.

```
% gcc calc1.c
% ./a.out '1+2*3=' > foo.s
% gcc -o b.out foo.s
% ./b.out
9
%
```



簡易自動テスト test1.csh, test2.csh (1)

- 提出プログラムを自動的にテストする簡易ツール.
- もし良ければ, 提出前に使ってください.
- 使い方:
 - 電卓コンパイラをコンパイルする. 実行可能ファイル名は a.out にする.
 - おなじディレクトリに, test*.csh, testcase*.txt をコピー.
 - `chmod +x test*.csh`
 - `./test.csh`

```
% ./test1.csh
'1+2=', 3, bad result!! 4
num = 6, bad = 1
%
```

テストケース6個中,
1つエラーがあったことを報告.



簡易自動テスト test1.csh, test2.csh (2)

- 配布のtest.tgzの解凍方法
 - tar xvzf test.tgz
- testcase1.txt は通常の計算のテストケース.
 - test1.csh はtestcase1.txt を使ってテスト.
 - calc1.c と calc3.c テスト用.
- testcase2.txt はオーバースpillのテストケース.
 - test2.csh はtestcase2.txt を使ってテスト.
 - calc2.c テスト用.

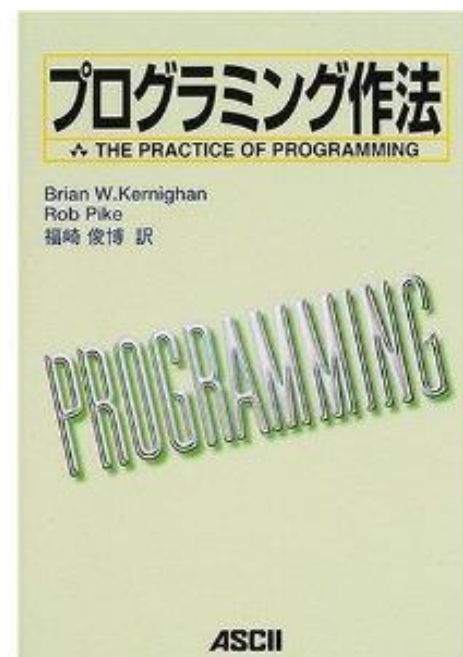


プログラムの書き方

- 他人が読みやすく理解しやすいプログラムを書く.
- 定石：
 - きれいにインデント（字下げ）する.
 - 分かりやすい変数名をつける.
 - 良いコメントを書く ← 超大事
 - モジュール化する.

参考書：プログラミング作法

<http://www.amazon.co.jp/dp/4756136494>





研究者としての倫理
社会的立場の自覚
つまらないことで人生を棒に振らない

カンニングはダメ

- アイデアレベルでの議論はOK（推奨）.
- でもソースコードを見るのはカンニングと見なす.
- 警告：剽窃チェッカーを使います.
見せた方も同罪.

権藤への質問は大歓迎

- 見せない努力も必要.
 - ソースコードを不用意に印刷して放置しない.
 - ソースコードを表示したまま，席を離れない.
 - ファイルやディレクトリの他人の読み取りを不許可にする.
 - 議論する際にソースコード（疑似コード）を使わない.
 - デバッグを助けてもらう時もソースコードは見せない.
 - Github等でパブリックに公開しない.



その他の注意（１）

- 情報工学科計算機室（CSC）の使用
 - 他の授業や演習を行っていない時間だけ使用すること.
 - その他，CSCの利用規則を遵守すること.
 - 違反した場合は厳罰を科す.
- リスク管理をすること.
 - **×切りギリギリを狙わない.**
 - ・ 根拠無く「×切一週間前からやろう」などと思わない.
 - 風邪，停電，学会発表，CSCの保守など，課題に取り組めない事態はいくらでもありうる.
 - パソコンが壊れても遅刻提出は認めない←バックアップ必須
 - 自己責任で，早めに課題に取り組み提出すること.
 - リスク管理は社会人に必要な重要なスキル.



その他の注意（２）

- つまずいたら質問すること
 - 一人で悩むと，１～２週間すぐ過ぎてしまう（時間の無駄）
 - 質問上手になろう，他人の質問に答えてあげよう
- レポートをちゃんと書く
 - レポートが貧弱だと大減点。
せっかくプログラム頑張ったのが無駄に。
- コンパイラの警告は**全て潰す**
 - むしろ，gcc -Wall オプションでより厳しくチェック
 - コンパイラ警告が出るコードは減点

```
calc1.c:156:9: warning: expression result unused [-Wunused-value]
    *p++;
```



ヒント(1/4)

- 一桁の加減算のみの実装のインタプリタ版
 - 例：5 + 3
 - 例：5 - 2
- num1, num2 は最終的には一般化して num だけにする.
- argv の使い方が分からない人はC言語を要復習.

*p++ は *(p++)と同じ

```
#include <stdio.h>
int main (int argc, char *argv [])
{
    char last_op, *p = argv [1];
    int num1, num2, acc;

    num1  = *p++ - '0';
    last_op = *p++;
    num2  = *p++ - '0';
    switch (last_op) {
    case '+':
        acc = num1 + num2;
        break;
    case '-':
        acc = num1 - num2;
        break;
    }

    printf ("%d¥n", acc);
}
```

```
% gcc -o sample0 sample0.c
% ./sample0 "5+3"
8
% ./sample0 "5-2"
3
%
```



sample1.c

ヒント(2/4)

- 一桁の加減算のみの実装のコンパイラ版
- エスケープ文字に注意.
 - `¥¥` は ¥ を印字.
 - `¥"` は " を印字.
 - `¥t` はタブ（字下げ）を印字.
 - `%%` は % を印字.
- 文字列定数中では改行禁止.
 - 悪い例：
".data¥n
L_fmt:¥n"
 - 複数の連続する文字列定数はコンパイル時に1つに合体.

```
#include <stdio.h>
int main (int argc, char *argv []) {
    char last_op, *p = argv [1];
    printf (".data¥n"
           "L_fmt:¥n"
           "¥t.ascii ¥"%%d¥¥n¥¥0¥"¥n"
           ".text¥n"
           ".globl _main¥n"
           "_main:¥n"
           "¥tpushq %%rbp¥n"
           "¥tmovq %%rsp, %%rbp¥n");

    printf ("¥tmovl $%c, %%ecx¥n", *p++);
    last_op = *p++;
    printf ("¥tmovl $%c, %%edx¥n", *p++);
    switch (last_op) {
    case '+':
        printf ("¥taddl %%edx, %%ecx¥n");
        break;
    case '-':
        printf ("¥tsubl %%edx, %%ecx¥n");
        break;
    }
    printf ("¥tmovb $0, %%al¥n"
           "¥tleaq L_fmt(%%rip), %%rdi¥n"
           "¥tmovslq %%ecx, %%rsi¥n"
           "¥tmovb $0, %%al¥n"
           "¥tcall _printf¥n"
           "¥tleave¥n"
           "¥tret¥n");
}
```



ヒント (3/4)

```
% gcc -o sample1 sample1.c
% ./sample1 "5+3" > foo.s
% gcc foo.s
% ./a.out
8
% ./sample1 "5-2" > foo2.s
% gcc foo2.s
% ./a.out
3
%
```

%ebxはcallee-saveなので
%ebxを使うなら %rbxを要退避

foo.s

```
.data
L_fmt:
    .ascii "%d¥n¥0"
.text
.globl _main
_main:
    pushq %rbp
    movq %rsp, %rbp
    movl $5, %ecx
    movl $3, %edx
    addl %edx, %ecx
    movb $0, %al
    leaq L_fmt(%rip), %rdi
    movslq %ecx, %rsi
    movb $0, %al
    call _printf
    leave
    ret
```

%al は隠しパラメタ (ベクトルレジスタの使用個数)
なので, 0を入れておく.



ヒント(4/4)

- オススメ(1)：callee-saveレジスタは使わない
 - main関数でcallee-saveレジスタを退避せずに上書きすると、main関数からリターン後にクラッシュすることがある（クラッシュしないこともある！）
- オススメ(2)：main関数からリターンせずにリターン直前に exit関数をcallして実行終了する
 - 正しく結果を出力できても、その後でクラッシュするともったいないから。
- オススメ(3)：コメントも出力する
 - 例：printf ("4: # loop exit¥n");
- calc1～3ではCコードでnumやaccを計算してはダメ
 - オーバーフローやゼロ割の検知もCコードでやってはダメ



Linux版 calc を作る注意点（１）

- 関数エピローグとプロローグをLinux用に変更
 - gcc -S の出力を真似する
 - でも endbr64 は使わなくてOK
- 関数名の頭のアンダースコア（_）は不要
- レポート中に「Linux環境を使用」と明示すること
 - 例：Linux Ubuntu 20.04 LTS (64bit)



Linux版 calc を作る注意点（2）

- Linux版 foo.s
- PLTは procedure linkage tableの略

```
.data
L_fmt:
    .ascii "%d¥n¥0"
.text
.globl main
.type main, @function
main:
    pushq %rbp
    movq %rsp, %rbp
    movl $5, %ecx
    movl $3, %edx
    addl %edx, %ecx
    movb $0, %al
    leaq L_fmt(%rip), %rdi
    movslq %ecx, %rsi
    movb $0, %al
    call printf@PLT
    leave
    ret
.size main, .-main
```