



2019: x86-64用に変更



コンパイラ構成

コード生成
(アセンブリ言語の復習 + α)

情報工学系
権藤克彦



概要

- `xcc-bison-dist-64/codegen.c` 中の関数 `codegen()` にコード生成部を実装せよ。
 - `codegen()` は構文解析・意味解析後に `xcc.y` から呼ばれる。
- コード生成のポイント
 - 制御文：（条件付）ジャンプ命令を使う
 - 式：後順序に構文木を訪問。式の計算結果をスタックに積む
 - 関数呼び出し：引数をスタックに積んでから `call` 命令実行
 - 引数・局所変数：ベースポインタのオフセットでアクセス
(例：`8(%rbp)`)
 - 大域変数：ラベルと `.skip` で確保 (例：`_x: .skip 8`)
ラベル名でアクセス (例：`_x(%rip)`)



準備



XCCの記号表：記号表の修正

- 関数やブロックの出入り毎に記号表 (sym_table) を要修正→以下の関数を使う

```
static void codegen_begin_block    (struct AST *ast);
static void codegen_end_block      (void);
static void codegen_begin_function (struct AST *ast);
static void codegen_end_function   (void);
```

codegen.c

- それぞれ、以下のタイミングで使う
 - ブロックに入る時
 - ブロックから出る時
 - 関数に入る時
 - 関数から出る時



使用例

ブロック中の宣言はAST中に入れていない。
意味解析で解析済みだから。
文のリストは左再帰なので下から先に処理。

```
static void codegen_block (struct AST *ast_block)
{
    struct AST *ast, *ast_stmt_list;
    assert (!strcmp (ast_block->ast_type, "AST_compound_statement"));
    codegen_begin_block (ast_block);
    ast_stmt_list = ast_block->child [1];
    ast = search_AST_bottom (ast_stmt_list,
                             "AST_statement_list_single", NULL);
    while (1) {
        if (!strcmp (ast->ast_type, "AST_statement_list_single"))
            codegen_stmt (ast->child [0]);
        else if (!strcmp (ast->ast_type, "AST_statement_list_pair"))
            codegen_stmt (ast->child [1]);
        else
            assert (0);
        if (ast == ast_stmt_list) break;
        ast = ast->parent;
    }
    codegen_end_block ();
}
```



XCCの記号表：記号表の検索

- 記号は以下の関数で検索する

```
struct Symbol *sym_lookup      (char *name);
struct Symbol *sym_lookup_label (char *name);
struct String *string_lookup   (char *data);
```

symbol.h

- それぞれ、以下で使う
 - (ラベル以外の) 記号の検索
 - ラベル記号の検索
 - 文字列定数の検索



使用例

```
else if (!strcmp (ast_stmt->ast_type, "AST_statement_goto"))
{
    label1 = ast_stmt->child [0]->u.id;
    sym = sym_lookup_label (label1);
    if (sym == NULL) {
        fprintf (stderr, "No such label: %s\n", label1);
        exit (1);
    }
    emit_code (ast_stmt, "\tjmp      %s.%s.%s\n",
               LABEL_PREFIX, func_name, label1);
}
```



ラッパ関数 emit_code

- 使用例 : emit_code (ast, "\t jmp %s\n", label);
- 可視化のため（後述）の、 fprintf のラッパ関数
 - 第2引数以降はそのまま fprintf に渡される
- ast は関連するASTノードへのポインタ
 - 「このアセンブリコードは、このASTノードの出力だよ」と紐付けるために必要

```
static void emit_code (struct AST *ast, char *fmt, ...)  
{  
    va_list argp; va_start (argp, fmt);  
    vfprintf (xcc_out, fmt, argp);  
    va_end (argp);  
}  
// 可視化モードがオンの時、以下でemit_codeを書き替える  
#ifdef XCC_VIS  
#include "vis/emit_code.h"  
#endif
```

MieruCompiler: XCCの可視化ツール

- <http://www.sde.cs.titech.ac.jp/mc64/>
- XCのコード断片と対応するコンパイル結果のアセンブリコードをハイライトして可視化

The screenshot shows the MieruCompiler interface with three main panes:

- SRC**: Shows the C code input. A portion of the loop body (lines 6-10) is highlighted in pink.
- ASM**: Shows the generated assembly code. The label `L.XCC.CL0:` and the loop body assembly are highlighted in pink.
- STACK**: Shows the stack layout. The stack grows downwards. Local variables are at -8 to -32, the old base pointer is at 0 (labeled "%rbp"), and the return address is at +8 (labeled "%rsp"). The assembly code `pushq %rax` and `popq %rax` are annotated with arrows pointing to the stack slots at addresses -8 and 0 respectively, indicating they access the local variable `i`.

もし良ければ使って下さい（使わなくても全然OK）



コード生成：制御文



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)  

```

第1引数
ifの条件判断
thenの部分

GCCのコンパイル例



例：if文のコード生成

```
} else if (!strcmp (ast_stmt->ast_type, "AST_statement_if")) {  
    codegen_exp (ast_stmt->child [0]);  
    label1 = create_ctrl_label ();  
    emit_code (ast_stmt, "\tpopq    %%rax\n");  
    emit_code (ast_stmt, "\tcmpq    $0, %%rax\n");  
    emit_code (ast_stmt, "\tje     %s\n", label1);  
    codegen_stmt (ast_stmt->child [1]);  
    emit_code (ast_stmt, "%s:\n", label1);  
}
```

if (式) 文1 else 文2

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;
}
```

GCCのコンパイル例

```
.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)  

```

引数 n
変数 i

} 文の部分

式文のコード生成

- 式文「式;」のコード生成

式 のコード

addq \$8, %rsp

スタック上の式の値を捨てる



ブロック文のコード生成

- 「{宣言リスト 文1 文2 … 文n }」のコード生成

文1 のコード

文2 のコード

・

・

・

文n のコード



goto文とラベル文のコード生成

- 「ラベル：」のコード生成

label :

label は他のラベルと
重複しないラベル

- 「goto ラベル;」のコード生成

jmp label

return文のコード生成

- 「return 式;」 のコード生成

式 のコード

```
popq %rax      戻り値は %rax に入る  
popq %rbp  
retq
```

- 別の方法 (こちらを推奨)

式 のコード

```
popq %rax  
jmp label  
.  
.  
label :  
    .  
    popq %rbp  
    retq
```

関数の最後の部分 (関数エピローグ)



コード生成：変数宣言

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

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

foo.s

```
.data
.globl _x1
.p2align 3
_x1:
.quad    111
.p2align 3
_x2:
.quad    222
```

```
% gcc -c foo.c
% nm foo.o
00000000 T _main
00000018 D _x1
00000020 d _x2
```

アセンブラーが大域変数に
アドレスを割り振っている。

静的変数のコード生成

- 未初期化の静的変数には、簡単のため、
.bss は使わず、.data セクションに割り当てる
 - ファイルやメモリの無駄遣いだが、動作には問題無い

```
long x;  
int main ()  
{;}
```

```
.section __DATA,__data  
.globl _x  
.p2align 3  
  
_x:  
.quad 0
```

セクション名は gcc の出力例を真似る
.data を使っても（たぶん）大丈夫

```
.comm _x,8,3
```

.bss セクションを使う場合の
アセンブリコード出力

文字列定数のコード生成

- .rdata セクションや.text セクションに割り当てる
 - .data セクションでも良い（書き換え可能になるけど）
- ラベルはカウンタを使って重複しないように自動生成
 - 重複さえしなければ、どんなラベル名でもOK

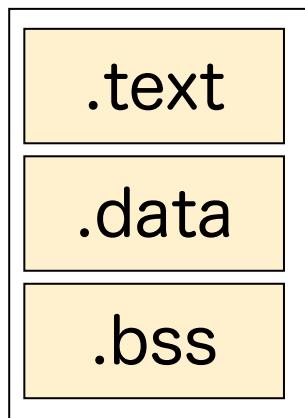
```
int main () {
    char *s;
    s = "Hello\n";
}
```

```
.section __TEXT,__cstring,cstring_literals
L_.str:
    .ascii  "Hello\n\0"
```

セクション名は gcc の出力例を真似る

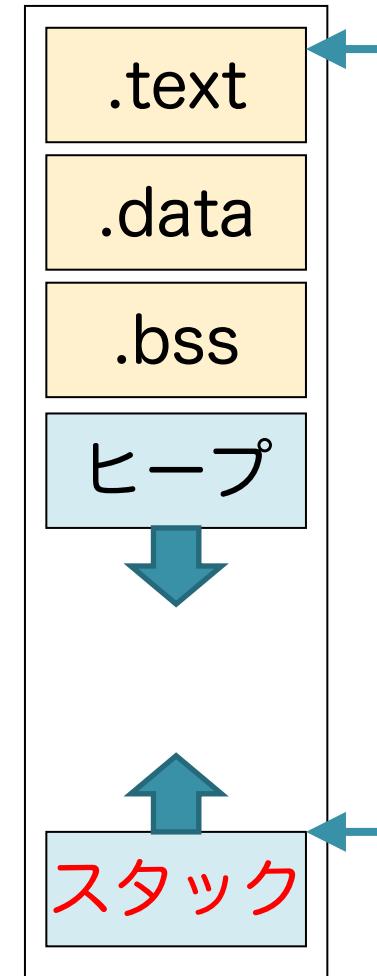


プロセスのメモリレイアウト



実行可能ファイル
a.out

機械語命令列
静的変数
(初期化済み)
静的変数
(未初期化)



プログラム
カウンタ %rip

malloc
すると成長

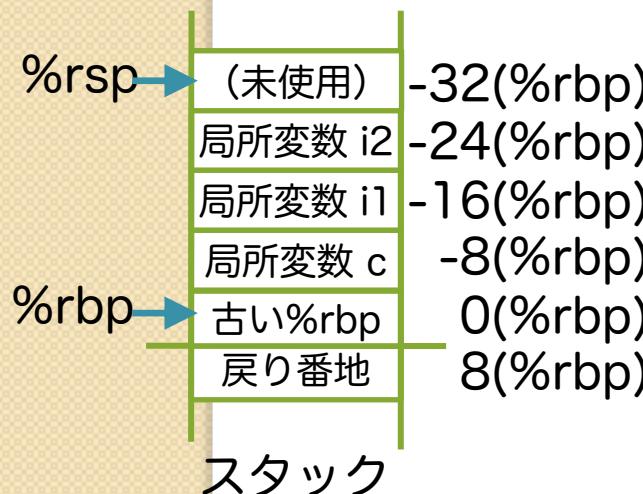
スタック
ポインタ%rsp

ユーザプロセスの
メモリ

局所変数：方法1

```
int main ()
{
    char c;
    int i1;
    int i2;
    return 0;
}
```

- スタックフレーム上に割り当てる
 - 局所変数のサイズ合計を16バイトの倍数に
 - call命令呼び出し時に $\%rsp \bmod 16 == 0$ が必要だから



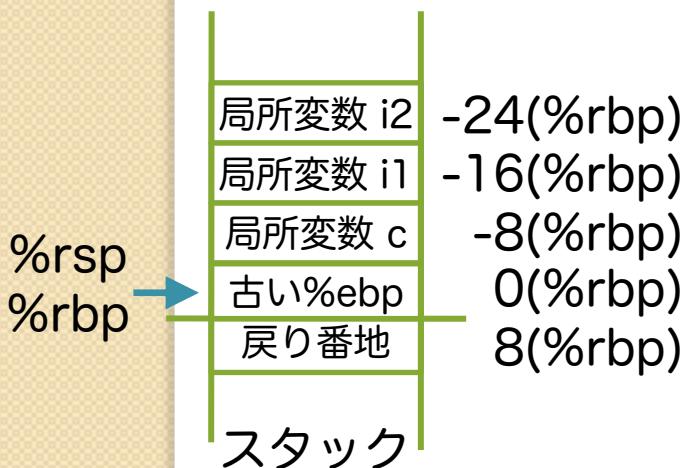
```
.text
.globl _main
.align 4, 0x90    # パディングにNOPを使用
_main:
    pushq %rbp
    movq %rsp, %rbp # ここで%rspは16の倍数
    subq $32, %rsp # 8+8+8=24を16の倍数に繰上げ
    movl $0, %eax
    leave # movq %rbp, %rsp; popq %rbp
    ret
```

ここではcharやintにも8バイト割り当てている
(簡単のため)

局所変数：方法2

- red zone を使う
 - %rsp を移動させず、%rspより上を勝手に使う

```
int main ()
{
    char  c;
    long  i1;
    long  i2;
    return 0;
}
```

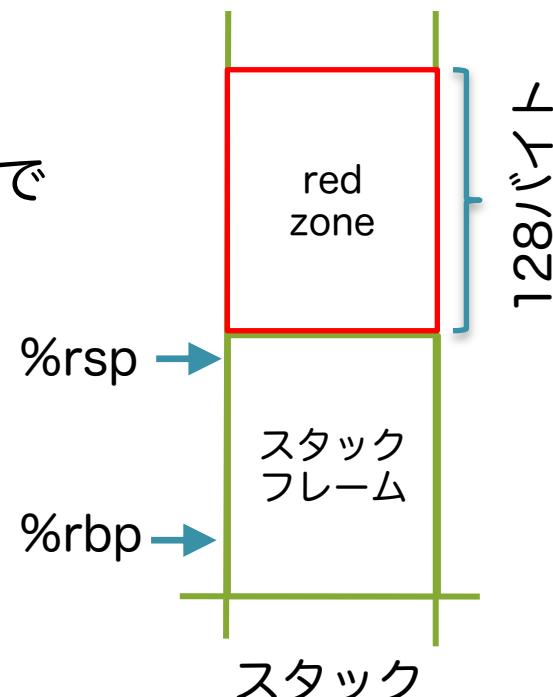


```
.text
.globl _main
.align 4, 0x90    # パディングにNOPを使用
_main:
    pushq %rbp
    movq %rsp, %rbp
    movl $0, %eax
    popq %rbp
    ret
```

別の関数をコールすると破壊されるので要注意

red zone

- 通常は%rspを動かして局所変数等の場所を確保。
 - 確保 : `subq $16, %rsp`
 - 解放 : `movq %rbp, %rsp`
- red zone は右図の場所
 - 局所変数や一時データ置き場に使用可
 - シグナルハンドラや割り込みハンドラで破壊されないことをABIが保証
- red zoneを使うと確保の`subq`命令を使わずに済む→高速化





コード生成：式



式のコード生成（概要）

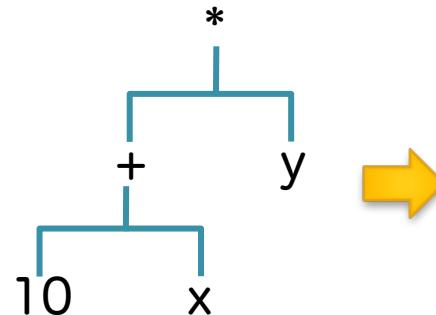
- スタック機械
 - レジスタの無いコンピュータ
- 中置記法と後置記法
- 後順序での木の訪問
- スタック使用のお約束（規約）
- 式をアセンブリコードに変換



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

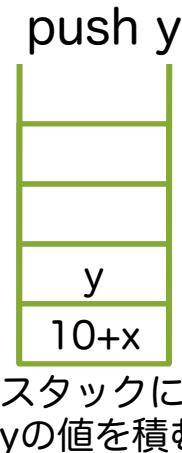
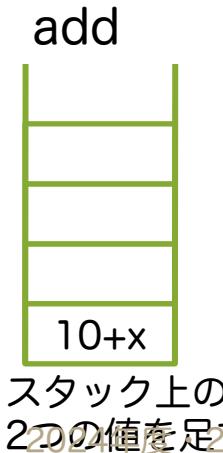
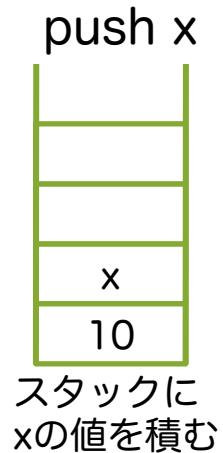
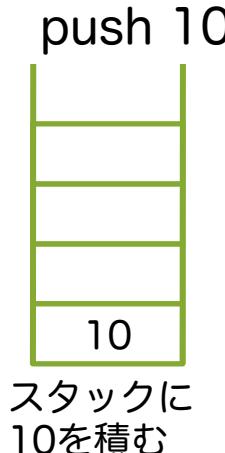
$10 \times + y *$

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

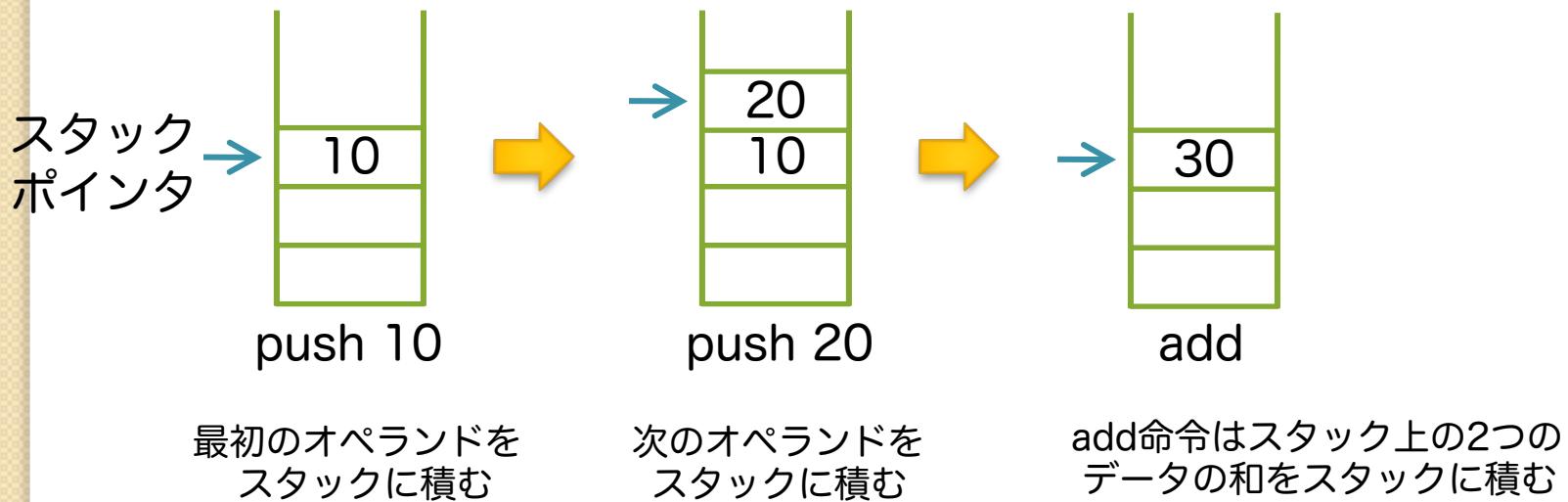
定数や変数はpush命令,
演算子は演算子命令に.





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、 push (pop () + pop ()) を計算する.
- 例：10+20の計算



後置記法(postfix notation)

- 後置記法 = 演算子をオペラントの後に書く。
 - 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.
 - オペラント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.

$10\ x\ +\ y\ *$ 

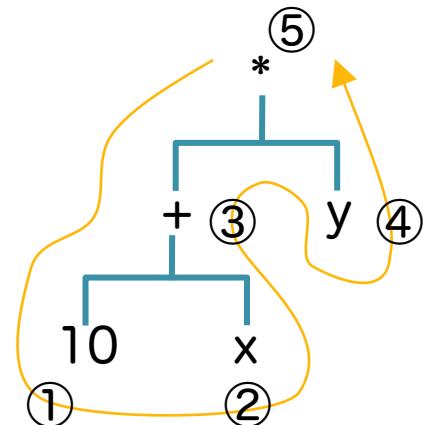
push 10
push x
add
push y
mul

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）
 - まず子ノードに後順序で（左から右に）訪問する.
 - 次の自分のノードを処理する.

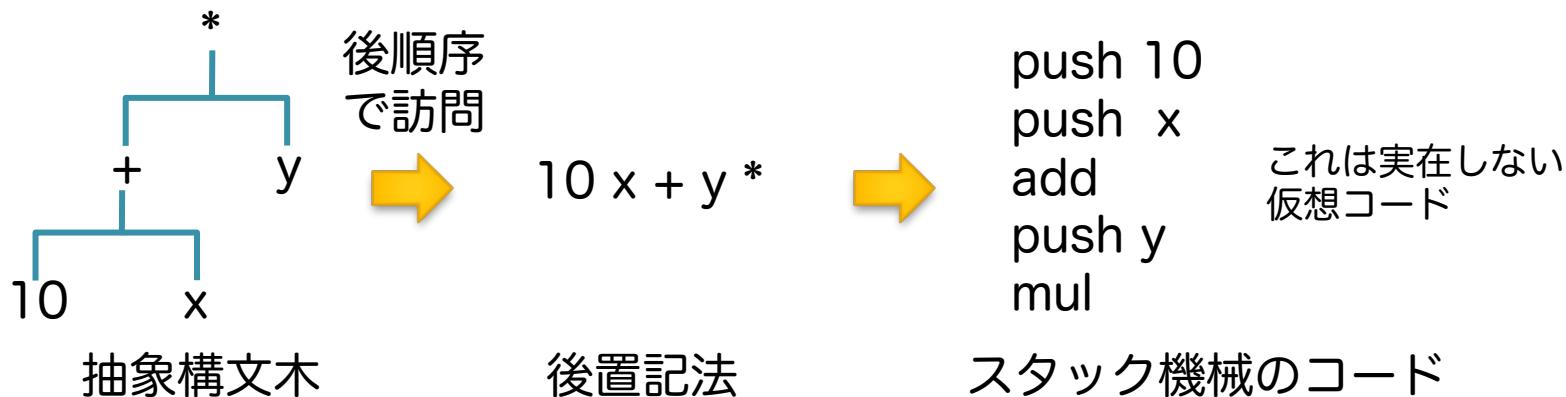
C言語での記述例

```
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）

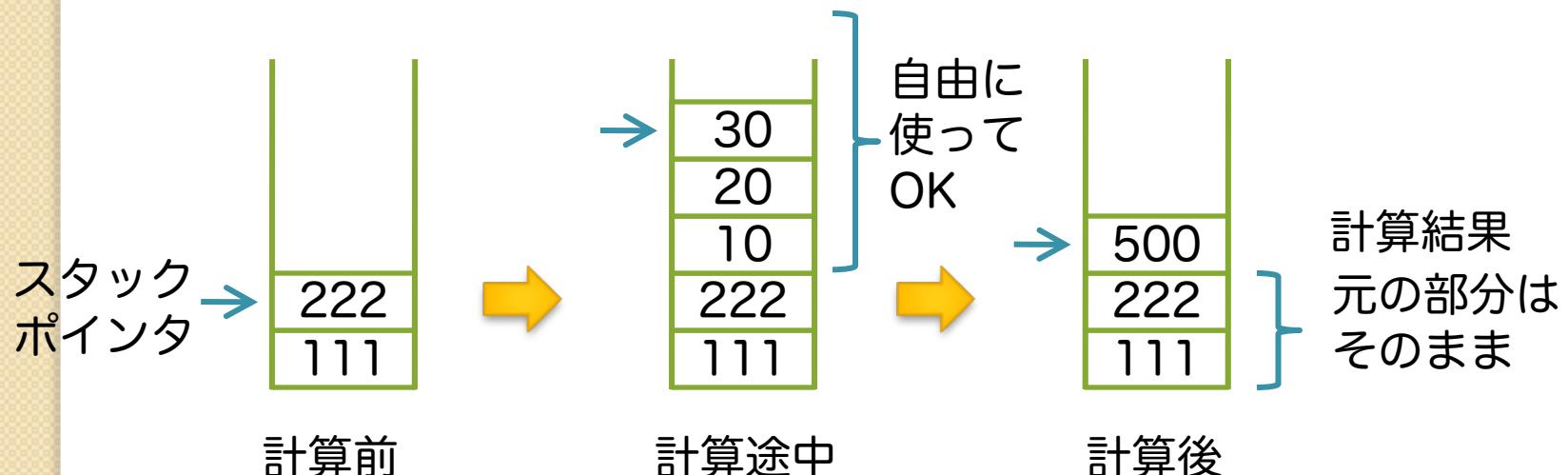
- スタック機械での式のコード生成：
 - 式の後置記法を左から右に読み、次の書き換えを行う。
 - 定数や変数 → push命令。 (例：push 10)
 - 演算子 → 演算子命令。 (例：add)
- 例： $(10+x)*y$ のコード生成。





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上の部分に限るに限る。
- 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）

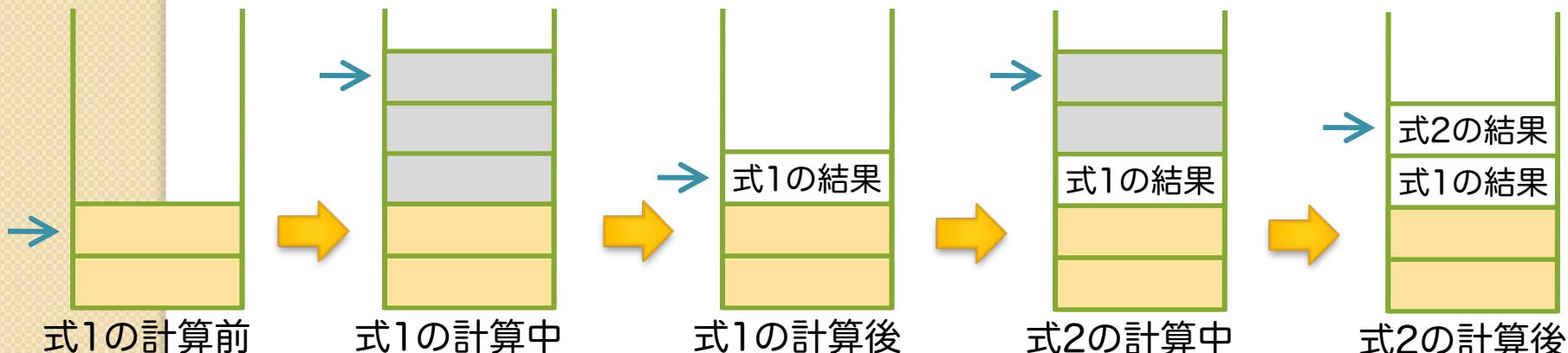
- 「式1+式2」のコード生成

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push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。
 - 簡単のため、レジスタは %rax, %r10, %r11のみを使う。
- 「式1+式2」のコード生成 (int型同士の場合)

式1 のコンパイル結果

式2 のコンパイル結果

```
popq  %r10
popq  %rax
addq  %r10, %rax
pushq %rax
```

式1の計算結果をスタックに積む

式2の計算結果をスタックに積む

式2の計算結果をレジスタ%r10にポップ
 式1の計算結果をレジスタ%raxにポップ
 %r10 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 %r10
popq %rax
addq %r10, %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：定数

- int型の定数は定数値をスタックにプッシュ
 - char型の定数は存在しない. 'A' も '\0' も long型で.

```
pushq $999  
pushq $'A'
```

- 文字列定数はアドレス値をスタックにプッシュ

```
leaq L_.str(%rip), %rax  
pushq %rax
```

```
L_.str:  
.ascii "hello\n\0"
```

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

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

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）

_x:
.quad 10



プッシュすると
低いアドレス
方向に伸びる。

局所変数3	-24(%rbp)
局所変数2	-16(%rbp)
局所変数1	-8(%rbp)
古い%rbp	0(%rbp)
戻り番地	8(%rbp)
第7引数	16(%rbp)
第8引数	24(%rbp)

スタック

式のコード生成：変数（2）

- 関数プロローグで第1～6引数をスタックに退避すると、以下のレジスタも自由に使える。
 - %rdi, %rsi, %rdx, %rcx, %r8, %r9

```
pushq  %rdi  
pushq  %rsi  
pushq  %rdx  
pushq  %rcx  
pushq  %r8  
pushq  %r9
```

- この資料では簡単のため退避せず、式の計算には %rax, %r10, %r11のみを使う

局所変数3	-72(%rbp)
局所変数2	-64(%rbp)
局所変数1	-56(%rbp)
第6引数	-48(%rbp)
第5引数	-40(%rbp)
第4引数	-32(%rbp)
第3引数	-24(%rbp)
第2引数	-16(%rbp)
第1引数	-8(%rbp)
古い%rbp	0(%rbp)
戻り番地	8(%rbp)
第7引数	16(%rbp)
第8引数	24(%rbp)

スタック



式のコード生成：変数（3）

- char型の「変数x」の右辺値コード生成
 - long型に変換する.
 - 全部64ビットに揃えるため. int型も同様.

```
movsbq -8(%rbp), %rax  
pushq %rax
```

movsbq (intel形式ではmovsx) は
バイトをクワッドワードに符号拡張して転送

整数やポインタと同じサイズで扱うと
コンパイラのコード生成が楽になる。

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 %r10

代入する値を%r10 に代入

movq %r10, 0(%rax)

%r10 の値を%raxが指すメモリ中に代入

pushq %r10

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 %r10
movq %r10, 0(%rax)
pushq %r10
```

ポインタ演算

- + と - は、オペラントがポインタ型の時、意味が変わる

```
int *p, *q, i;  
p + i;  
p - i;  
p - q;  
p + q;  
i - q;
```

$p + i * \text{sizeof } (*p)$ を計算
 $p - i * \text{sizeof } (*p)$ を計算
 $(p - q) / \text{sizeof } (*p)$ を計算
コンパイルエラー
コンパイルエラー

== のコード生成

- 「式1 == 式2」のコード生成

式1 の右辺値のコード

式2 の右辺値のコード

popq %rax	
popq %r10	
cmpq %rax, %r10	%raxと%r10を比較 (%rflagsの値が変化)
sete %al	等しいか否かの結果 (1か0) を%alに代入
movzbq %al, %rax	%al の値をゼロ拡張して %rax に転送
pushq %rax	

movzbq (intel形式では movzx) は
バイトをクアッドワードにゼロ拡張して転送

sete ではなく, je を使う別解あり



&& と || のコード生成

- 「左から右への評価」 (left-to-right evaluation) を行う
 - まず左オペラントを先に計算する
 - 式全体の計算に必要な場合のみ、右オペラントを計算する
- 例 : if ((p != NULL) && (*p > 0)) { ...
 - ($*p > 0$)を計算するのは、($p \neq \text{NULL}$)が真(1)の時だけ
 - &&の場合、左オペラントの計算結果が偽(0)の場合は、右オペラントを計算せずに、式全体の値を偽(0)にしなければならない

単項演算子のコード生成：& と *

- 「&式」の右辺値のコード生成（左辺値は無し）

式 の左辺値のコード

- 「*式」の右辺値のコード生成

式 の右辺値のコード

計算結果はアドレス（のはず）

popq %rax
movq 0(%rax), %rax
pushq %rax

アドレスを%raxに格納
そのアドレスが指すメモリ値を%raxに格納

- 「*式」の左辺値のコード生成

式 の右辺値のコード

計算結果はアドレス（のはず）



コード生成：関数呼び出し

%rsp の 16バイト境界制約に注意

- macOS では、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.
 - 分かっていても、うっかり間違えやすい.

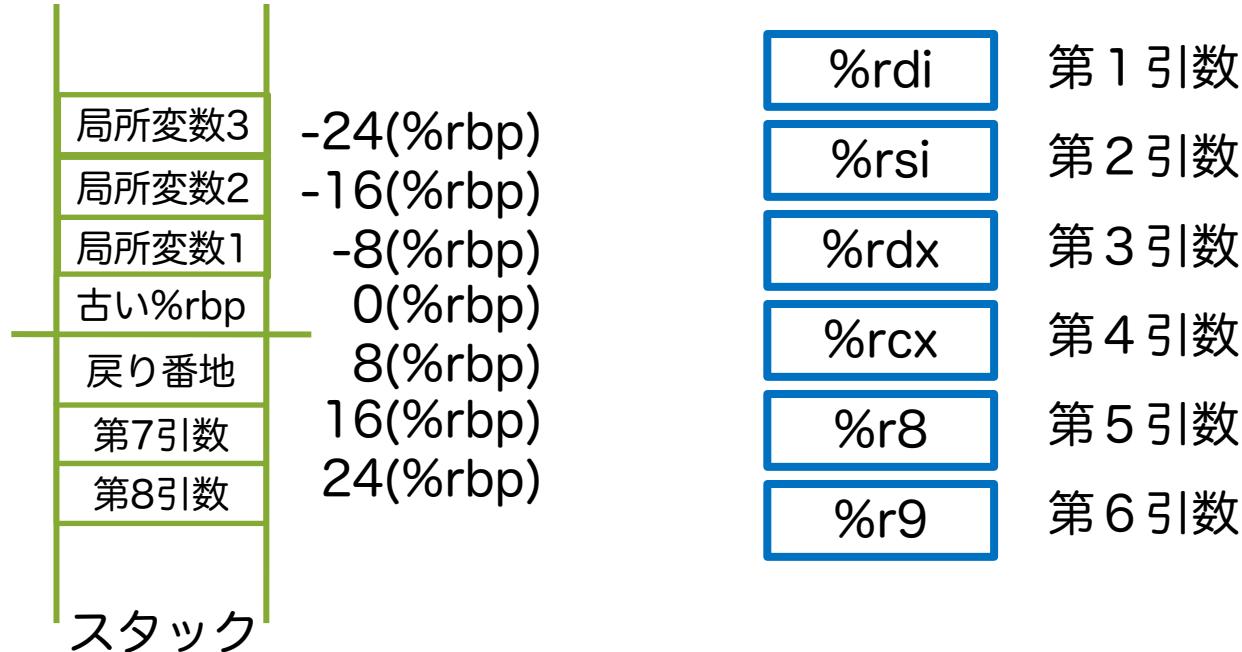
```
pushq %rbp  
movq %rsp, %rbp  
# ここで16の倍数  
...  
popq %rbp  
retq
```

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

```
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.

引数の渡し方 (AMD64 ABI)



ライブラリ関数 (printfなど)の場合、
第7引数以降に int (4バイト) を積む必要がある場合もあるが、
それは考えず、常に long (8バイト) を積む戦略で（手抜きで）

call ラベル near・相対・直接
call *%rax near・絶対・間接

関数呼び出し式のコード生成（1）

- 「式0 (式1, 式2, …, 式n)」のコード生成
 - 式0は多くの場合、単なる関数名。だが一般的には式。
 - 引数を逆順にスタックに積んで、call命令を実行する

subq \$n, %rsp nは16バイト境界調整のためのバイト数

式n の右辺値のコード

⋮

式1 の右辺値のコード

式0 の右辺値のコード

popq %r11

popq %rdi

⋮

call *%r11

addq \$args_size , %rsp

pushq %rax

leaq _fact(%rip), %rax; pushq %rax とか

計算結果は関数へのアドレス（のはず）

第1～6引数はレジスタにセット

$args_size = n + \text{第7引数以降のサイズ合計}$
関数呼び出し後にスタックから除去
関数からの返り値をスタックにプッシュ

関数呼び出しのコード生成 (2)

- 「foo (10, 20)」のコード生成例

```
pushq $20
pushq $10
leaq _foo(%rip), %rax
pushq %rax
popq %r11
popq %rdi
popq %rsi
call *%r11
pushq %rax
```

関数fooのアドレスを
スタックにプッシュ



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
 - 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 $0x10000fb1$ 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 = 0x000000010000fa6  a.out`main + 6 at
foo.c:3:7
(lldb) memory read --format d 0x100001000
0x100001000: 999
(lldb) quit
```



位置独立コード（3）

- global offset table (GOT)
 - 間接ジャンプテーブル（＝アドレスの配列）
 - 別の共有ライブラリ中の関数を呼ぶときに参照
 - .textセクションから固定相対位置に置かれる

.textセクション

```
movq _puts@GOTPCREL(%rip), %rax
```



静的に相対アドレス
が決まる

.gotセクション

```
GOTの puts のエントリ
```



ロード時（またはコール時に）
putsの絶対アドレスが格納される



位置独立コード（4）：lldbで確認

$0xf96 + 0x6a = 0x1000$

```
#include <stdio.h>
int main ()
{
    int (*fp)(const char *) = puts;
    fp ("10\n"); // putsを間接コール
}
```

```
% objdump -D a.out | more (出力略)
__text:
100000f8f: movq 0x6a(%rip), %rax
100000f96:
__got:
100001000: 0x0000000000000000
```

中身は（実行前なので）ゼロ

```
% lldb a.out
(lldb) b main
(lldb) r
(lldb) memory read --size 8 --format x 0x100001000
0x100001000: 0x00007fff6d3019e2 0x0000000000000000
(lldb) dis
a.out`main:
0x100000f8f <+15>: movq    0x6a(%rip), %rax
; (void *)0x00007fff6d3019e2: puts
```

printfなどのライブラリ関数（1）

```
movq _printf@GOTPCREL(%rip), %r11
```

- `_printf@GOTPCREL`とは
 - GOT中の `printf` のエントリへの相対アドレスを表す
 - そこに実行時に共有ライブラリ `printf` の絶対アドレスが入る
 - 類：`@GOT`, `@GOTPLT`, `@GOTOFF`, `@PLT`

どうやってライブラリ変数を区別するか
→ ライブラリ関数一覧表をxccが持つことによる

caller-saveレジスタをスタックに要退避・回復

printfなどのライブラリ関数（2）

```
movq    _printf@GOTPCREL(%rip), %r11  
....  
movb    $0, %al  
call    *%r11
```

- x86-64 ABIでのお約束
 - 可変長引数を持つ関数、あるいはプロトタイプ宣言が無い関数を呼び出す場合、%al が隠し引数として使われる。
%al には使用するベクタレジスタの数を格納する。
 - ベクタレジスタ=ベクトル演算に使うレジスタ。
 - 例：XMMレジスタ
- ここではベクタレジスタを使わないので0をセット
- %allは%raxの一部なので、%raxは使えない



コード生成：関数定義

典型的な関数プロローグ と関数エピローグ

関数プロローグ

```
pushq %rbp  
movq %rsp, %rbp  
pushq %rbx  
pushq %r12  
pushq %r13  
subq $24, %rsp
```

新しいスタックフレームを作成

callee-saveレジスタを
スタック上に退避

必要な
場合だけ

スタックフレーム上に領域を確保
(自動変数や引数のための領域など)

必要なサイズは
関数ごとに異なる

関数エピローグ

```
addq $24, %rsp  
popq %r13  
popq %r12  
popq %rbx  
leave  
retq
```

自動変数や引数などのための領域を解放

スタック上に退避していた
callee-saveレジスタを回復

必要な
場合だけ

スタックフレームを破棄

leave=mvq %rbp, %rsp; popq %rbp



今の macOS環境では実行できない

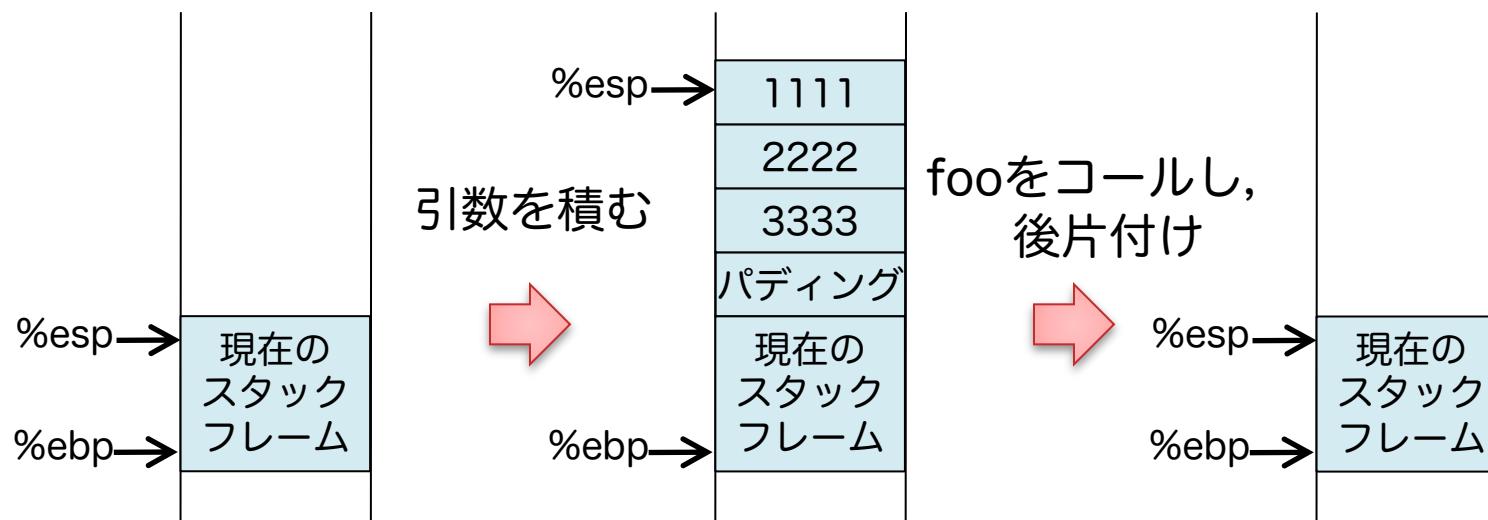
典型的な関数呼び出し (1)

一昔前のGCCが吐くコード。
引数をpushするので自然。

```
void foo (int a, int b, int c);
void foo2 (void)
{
    foo (1111, 2222, 3333);
}
```

```
% gcc -mtune=i386 -static -S foo.c
```

```
subl $4, %esp
pushl $3333
pushl $2222
pushl $1111
call _foo
addl $16, %esp
```



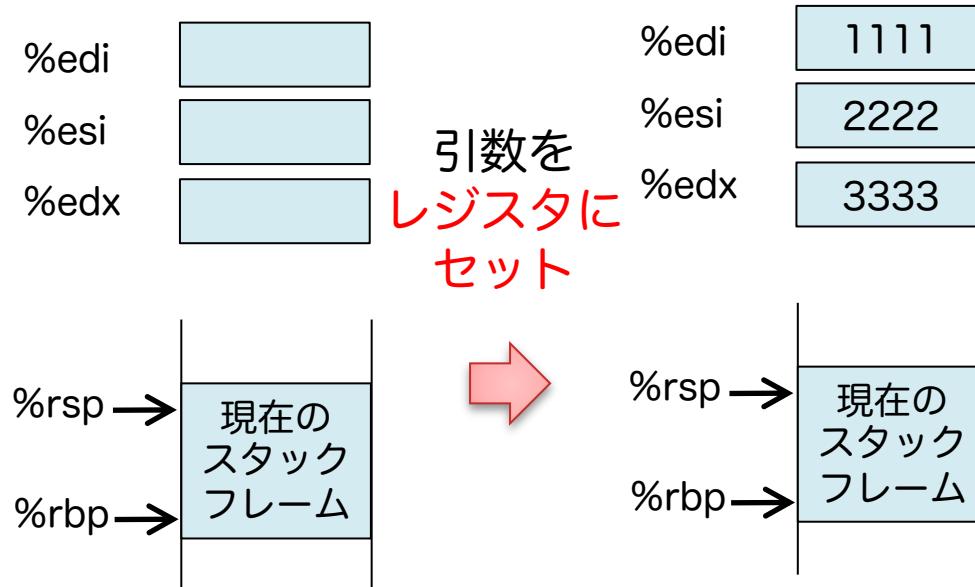
macOSのABIが、**関数呼び出し時にスタック
ポインタが16バイト境界を指すことを要求するので
パディングが入っている。**

最近のGCCが吐くコード。
引数をレジスタにセット。

典型的な関数呼び出し（2）

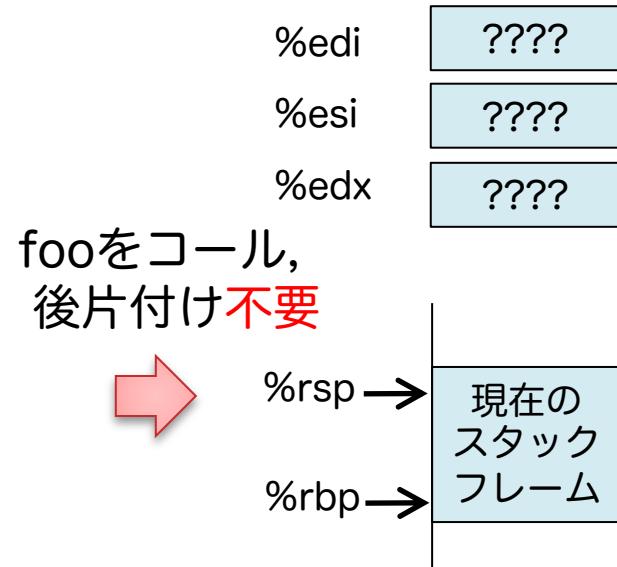
```
void foo (int a, int b, int c);  
void foo2 (void)  
{  
    foo (1111, 2222, 3333);  
}
```

```
% gcc -S foo.c
```

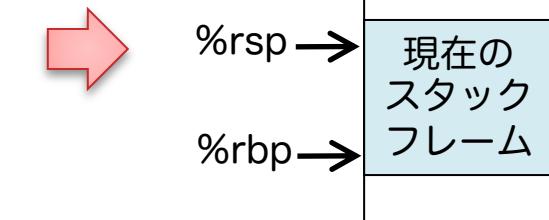


```
movl $3333, %edx  
movl $2222, %esi  
movl $1111, %edi  
callq _foo
```

caller-saveレジスタ
なので必要なら退避と
復旧は必要



fooをコール,
後片付け不要



第7引数以降は (pushではなく)
movでstack上にセット



関数定義のコード生成例

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

```
.text
.globl _add5
.p2align 4, 0x90
_add5:
    pushq %rbp
    movq %rsp, %rbp
    pushq %rdi
    pushq $5
    popq %r10
    popq %rax
    addq %r10, %rax
    pushq %rax
    popq %rax      # return value
    jmp L.XCC.RE.add5
L.XCC.RE.add5:
    movq %rbp, %rsp
    popq %rbp
    retq
```



LLDBの使用例メモ

<https://lldb.llvm.org/index.html>



レジスタの値を読む

```
% lldb ./a.out
(lldb) b main
(lldb) r
(lldb) s
-> 0x100000e94 <+4>: pushq %rdi
    0x100000e95 <+5>: pushq %rsi
    0x100000e96 <+6>: pushq %rdx
    0x100000e97 <+7>: pushq %rcx
(lldb) reg read rdi
    rdi = 0x0000000000000001
(lldb) reg read rsi
    rsi = 0x00007fffeefbff768
(lldb) quit
```

ブレークポイント設定
実行開始
ステップ実行

%rdi の値を表示



-16(%rbp) の値を読む

```
% lldb ./a.out
(lldb) b main
(lldb) r
(lldb) reg read rbp
    rbp = 0x00007fffeefbff758
(lldb) memory read 0x00007fffeefbff758
0x7fffeefbff758: 00 00 00 00 00 00 00 00
(lldb) x $rbp
0x7fffeefbff758: 00 00 00 00 00 00 00 00
(lldb) x $rbp-16
0x7fffeefbff748: 15 00 27 6d ff 7f 00 00
```

指定番地のメモリ内容を表示

x は memory readの省略形

レジスタ参照は \$rbp-16
などと指定可能



Segmentation faultした

```
% llDb ./a.out
(llDb) r
* thread #1, queue = 'com.apple.main-thread', stop reason =
EXC_BAD_ACCESS (code=1, address=0x0)
    frame #0: 0x0000000100000f70 a.out`main + 64
a.out`main:
-> 0x100000f70 <+64>: movq    (%rsi), %rsi
    0x100000f73 <+67>: movl    %eax, -0x14(%rbp)
(llDb) reg read rsi
    rsi = 0x0000000000000000
```

どこで SEGVしたか一発で分かる