2014 年 7 月 10 日

Q言語コンパイラ 関連資料

三浦 欽也

IA-32 インテル(R)アーキテクチャ

Q言語コンパイラで考慮すべきレジスタは，4 つの汎用レジスタ(%eax，%ebx，%ecx，%edx)と，スタックの先頭を指すスタックポインタ(%esp)，スタック中の基準となるアドレスを指すベースポインタ(%ebp)である．いずれも 32bit の長さを持つ．(図 1 参照)

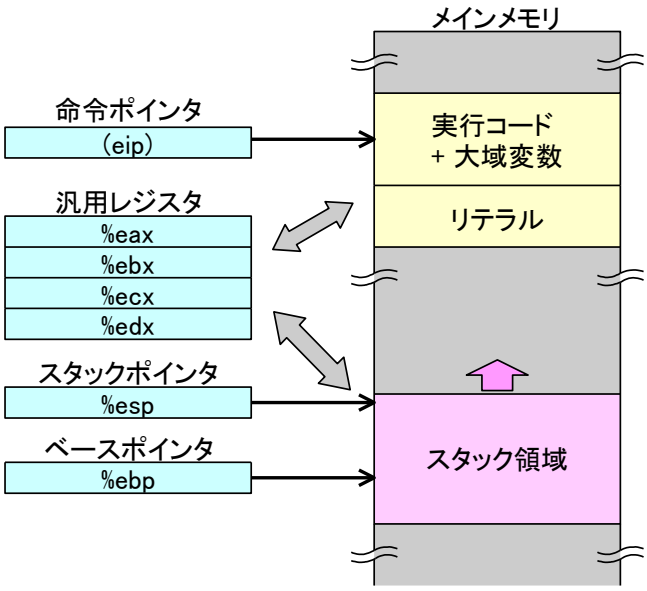


図1　IA-32 アーキテクチャ

変数の記憶域

Q言語の変数は，型から見れば，整数と整数の配列の 2 種類であり，変数のスコープから見れば，大域変数と局所変数がある．また，関数の仮引数も，変数に準じるもの(整数型，局所変数)と考えることができる．

これらのうち，大域変数の記憶場所は，実行コードと同じ領域に，実行コードと混在した形で割り当てられる．また，局所変数(と関数の仮引数)は，スタック領域の中に割り当てられる．局所変数(と関数の仮引数)の詳細については，次項を参照されたい．

関数フレーム

関数が呼び出されるごとに，それに対応して，毎回スタック上に確保される局所的な記憶領域を，関数フレームと呼ぶ．ここには，その関数をスコープとする局所変数や，関数呼び出し時に与えられた引数が含まれており，ベースポインタ(%ebp)に対する相対アドレッシングによってアクセスされる．Q言語の目的コードでは，右図のような構成を仮定している．

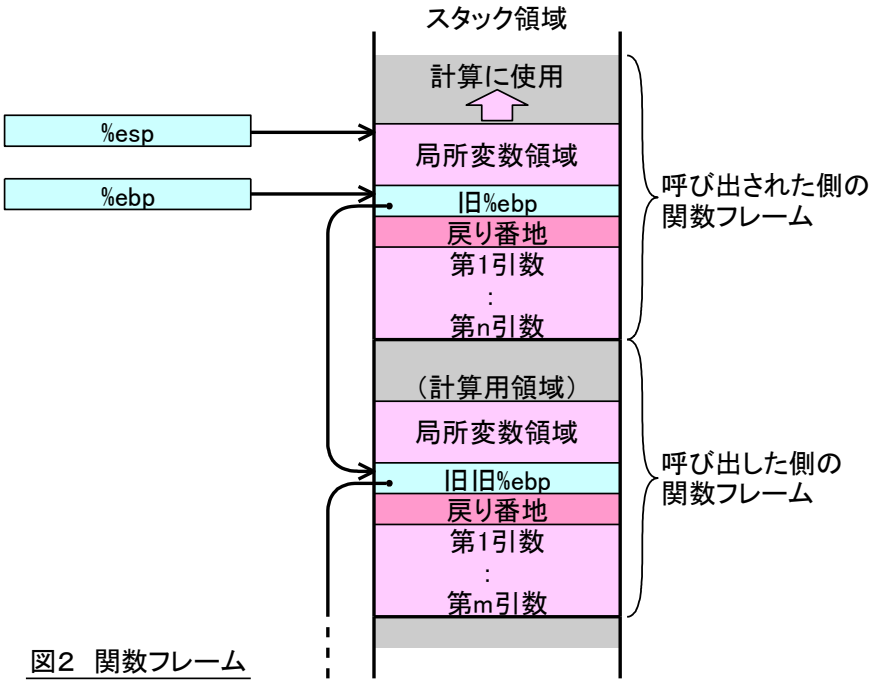


図2　関数フレーム

式の評価(計算)中に，関数が呼び出されると，まずその関数呼び出しの実引数を後ろから順に評価して結果をスタックに push し，関数を call する．その結果，スタック上に戻り番地が push され，プログラムの実行は関数の開始番地に分岐する．(cf.　下記「関数呼び出し」の項)

呼び出された関数の側では，呼び出し側の関数フレームで用いられていたベースポインタをスタックに退避(push)し，%esp を進めて，その関数をスコープとする局所変数の領域を確保する．(cf.　下記「関数定義の目的コード」の項)

関数の実行終了時には，局所変数領域を解放してベースポインタを復元し，ret 命令で戻り番地に復帰後に，引数の領域を解放する．(cf.　下記「関数呼び出し」の項)

大域変数宣言の目的コード

大域変数は，その宣言時に，ただちにその変数と同じ名前の共有領域を定義するアセンブリ命令を生成する．
その結果，図1にみたように，実行コードと混在する形でそれらの「共有領域」が生成される．
（cf. code.c, l. 73）

```
var x, y[n], … ;    ⇒    .comm  x, 4
                         .comm  y, 〈4*n〉
                          …
```

式の目的コード

式の目的コードは，その式の値をレジスタ%eax に残すようなコードとして生成される．

整数定数

```
C       ⇒    mov    %eax, C        /* code_exp.c, l. 92 */
```

文字列定数（リテラル）

```
"…"     ⇒    lea    %eax, 〈リテラル"…"のラベル〉  /* code_exp.c, l. 94 */
```

大域変数

```
x       ⇒    mov    %eax, x                /* code_exp.c, ll. 85-90 */
y[α]    ⇒    〈α のコード〉                  /* code_exp.c, l. 74 */
             mov    %eax, [%eax*4+y]       /* code_exp.c, l. 76 */
```

局所変数・関数の引数

変数のオフセット（関数フレーム内で，ベースポインタが指示する場所からの距離）は，コンパイル時（意味解析時）に値が定まり，オブジェクト構造体にその値が書き込まれているので，それを用いる．

```
x       ⇒    mov    %eax, [%ebp+〈x のオフセット*4〉]      /* code_exp.c, ll. 85-90 */
y[α]    ⇒    〈α のコード〉                                 /* code_exp.c, l. 74 */
             mov    %eax, [%ebp+%eax*4+〈y のオフセット*4〉]  /* code_exp.c, l. 77 */
```

関数呼び出し

関数の戻り値は%eax に格納されていると仮定する．

```
fn(α1, …, αn)    ⇒    〈αn のコード〉          /* code_exp.c, l. 157 */
                       push   %eax            /* code_exp.c, l. 158 */
                        …
                       〈α1 のコード〉          /* code_exp.c, l. 157 */
                       push   %eax            /* code_exp.c, l. 158 */
                       call   fn              /* code_exp.c, l. 66  */
                       add    %esp, 〈n*4〉     /* code_exp.c, l. 71  */
```

（cf. 図2）

単項演算式

```
!α    ⇒          〈α のコード〉          /* code_exp.c, l. 52 */
                 or      %eax, %eax     /* code_exp.c, l. 53 */
                 jz      loc            /* code_exp.c, l. 54 */
                 mov     %eax, 1        /* code_exp.c, l. 55 */
           loc:                         /* code_exp.c, l. 56 */
                 dec     %eax           /* code_exp.c, l. 57 */

-α    ⇒     〈α のコード〉  /* code_exp.c, l. 60 */
            neg     %eax   /* code_exp.c, l. 61 */
```

二項算術演算式

四則演算子 +, -, *, / (*OP*) を用いた二項演算式は，対応する算術演算命令 add, sub, imul, mdiv(*) (*op*) を用いて，下記のような形式のコードを生成する.

```
α OP β      ⇒     〈β のコード〉          /* code_exp.c, l. 145 */
                  push    %eax           /* code_exp.c, l. 146 */
                  〈α のコード〉          /* code_exp.c, l. 147 */
                  pop     %ebx           /* code_exp.c, l. 148 */
                  op      %eax, %ebx     /* code_exp.c, l. 149 */
```

(*) 除算については，他の演算と同じ形式で書けるように，下記のように定義されたマクロ命令 mdiv を用いる. (cf. code.c, ll. 25-28)

```
.macro mdiv   eax, SRC
       cdq
       idiv   ¥SRC
.endm
```

比較演算式

比較演算子 ==, !=, <, <=, >, >= (*OP*) を用いた二項演算式は，各々，条件分岐命令 jne, je, jge, jg, jle, jl (*op*) を用いて，下記のような形式のコードを生成する.

```
α OP β      ⇒          〈β のコード〉          /* code_exp.c, l. 132 */
                       push    %eax           /* code_exp.c, l. 133 */
                       〈α のコード〉          /* code_exp.c, l. 134 */
                       pop     %ebx           /* code_exp.c, l. 135 */
                       cmp     %eax, %ebx     /* code_exp.c, l. 136 */
                       mov     %eax, 0        /* code_exp.c, l. 137 */
                       op      loc            /* code_exp.c, l. 138 */
                       dec     %eax           /* code_exp.c, l. 139 */
                 loc:                         /* code_exp.c, l. 140 */
```

論理二項演算式

左辺の実行結果によって右辺を実行するか否かが決まるので，他の二項演算式とは扱いが異なる. 論理演算子 ||, && (*OP*) に対し，条件分岐命令 jnz, jz (*op*) を用いて，下記のような形式のコードを生成する.

```
α OP β      ⇒          〈α のコード〉          /* code_exp.c, l. 122 */
                       or      %eax, %eax     /* code_exp.c, l. 123 */
                       op      loc            /* code_exp.c, l. 124 */
                       〈β のコード〉          /* code_exp.c, l. 125 */
                 loc:                         /* code_exp.c, l. 126 */
```

大域変数への代入式

```
x = β        ⇒    〈β のコード〉                  /* code_exp.c, l. 105 */
                  mov    x, %eax                 /* code_exp.c, ll. 101, 106 */

y[α] = β     ⇒    〈α のコード〉                  /* code_exp.c, l. 110 */
                  push   %eax                    /* code_exp.c, l. 111 */
                  〈β のコード〉                  /* code_exp.c, l. 112 */
                  lea    %ebx, y                 /* code_exp.c, ll. 101, 108, 113 */
                  pop    %edx                    /* code_exp.c, l. 114 */
                  mov    [%ebx+%edx*4], %eax     /* code_exp.c, l. 115 */
```

局所変数・関数の引数への代入式

変数のオフセット(関数フレーム内で，ベースポインタが指示する場所からの距離)は，コンパイル時(意味解析時)に値が定まり，オブジェクト構造体にその値が書き込まれているので，それを用いる．(ファイルはすべて code_exp.c)

```
x = β        ⇒    〈β のコード〉                  /* l. 105 */
                  mov    [%ebp+〈x のオフセット*4〉], %eax   /* ll. 102, 106 */

y[α] = β     ⇒    〈α のコード〉                  /* l. 110 */
                  push   %eax                    /* l. 111 */
                  〈β のコード〉                  /* l. 112 */
                  lea    %ebx, [%ebp+〈y のオフセット*4〉]   /* ll. 102, 108, 113 */
                  pop    %edx                    /* l. 114 */
                  mov    [%ebx+%edx*4], %eax     /* l. 115 */
```

関数定義の目的コード

関数が必要とする局所変数領域のサイズは，コンパイル時(意味解析時)に値が定まり，オブジェクト構造体にその値が書き込まれているので，それを用いる．また，return 文のための分岐先として，__関数名__r というラベルを定義する．(ファイルはすべて code.c)

```
fn(…) α      ⇒          .globl fn                          /* l. 79 */
              fn:                                          /* l. 80 */
                         push   %ebp                       /* l. 81 */
                         mov    %ebp, %esp                 /* l. 82 */
                         sub    %esp, 〈fn の局所変数領域のサイズ〉   /* l. 83 */
                         〈α のコード〉                     /* l. 84 */
              _fn_r:                                       /* l. 85 */
                         mov    %esp, %ebp                 /* l. 86 */
                         pop    %ebp                       /* l. 87 */
                         ret                               /* l. 88 */
```

return 文の目的コード

その return 文を含む関数定義の末尾に分岐するため，上(関数定義の目的コード)で定義した__関数名__r というラベルを用いる．

```
return α     ⇒    〈α のコード〉   /* code.c, l. 38 */
                  jmp    _fn_r   /* code.c, l. 39, fn はこの文を含む関数の名前 */
```

文の目的コード

「式」文

式 α にセミコロン ； をつけて得られる文については，その式のコードそのものが，その文のコードとなる．
(%eax の値は，原則として捨てられる(使用されない))

```
α;                  ⇒   〈α のコード〉          /* code.c, l. 62 */
```

ブロック

ブロックの目的コードは，ブロックを構成する各文のコードを並べたものとなる．

```
{ ξ, η, … , ζ }     ⇒   〈ξ のコード〉          /* code.c, l. 41 */
                        〈η のコード〉
                          …
                        〈ζ のコード〉
```

if 文

```
if ( α ) ξ          ⇒       〈α のコード〉          /* code.c, l. 43 */
                            or      %eax, %eax    /* code.c, l. 44, %eax が 0 か check */
                            jz      loc1          /* code.c, l. 45, 0 なら分岐 */
                            〈ξ のコード〉          /* code.c, l. 46, true パート */
                        loc1:                     /* code.c, l. 48 */
```

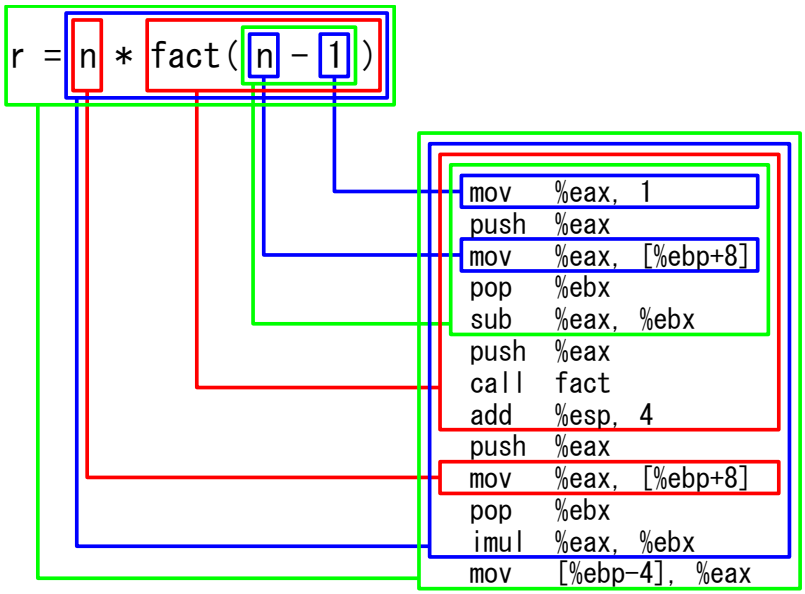
```
if ( α ) ξ else η
                    ⇒       〈α のコード〉          /* code.c, l. 43 */
                            or      %eax, %eax    /* code.c, l. 44, %eax が 0 か check */
                            jz      loc1          /* code.c, l. 45, 0 なら分岐 */
                            〈ξ のコード〉          /* code.c, l. 46, true パート */
                            jmp     loc2          /* code.c, l. 47, 無条件に分岐 */
                        loc1:                     /* code.c, l. 48 */
                            〈η のコード〉          /* code.c, l. 50, false パート */
                        loc2:                     /* code.c, l. 51 */
```

while 文

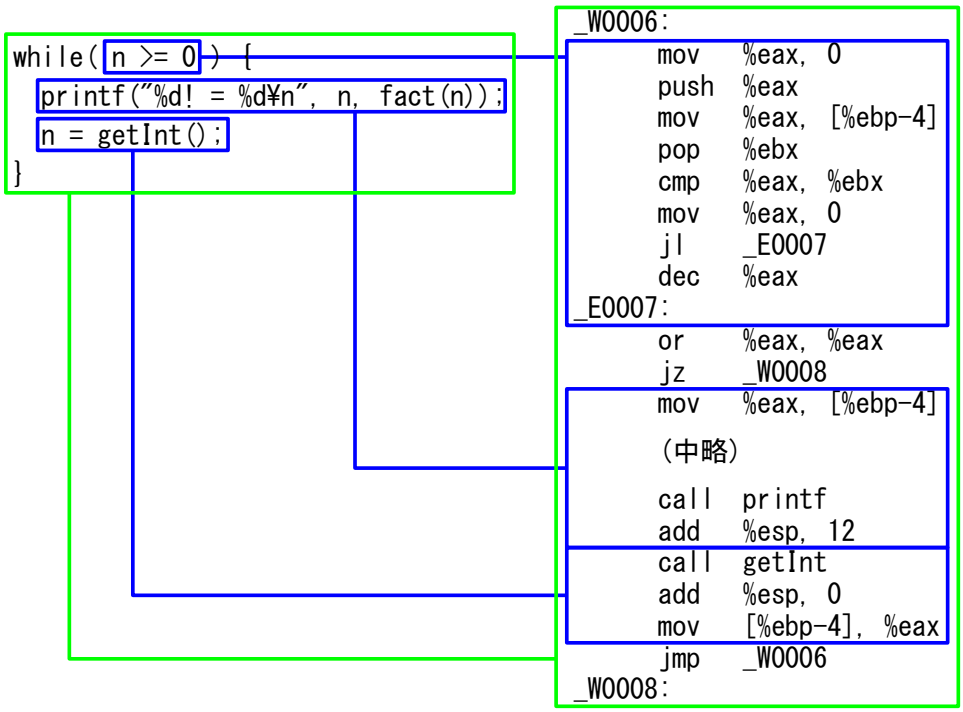
```
while ( α ) ξ
                    ⇒   loc2:                     /* code.c, l. 54 */
                            〈α のコード〉          /* code.c, l. 55 */
                            or      %eax, %eax    /* code.c, l. 56, %eax が 0 か check */
                            jz      loc1          /* code.c, l. 57, 0 なら分岐 */
                            〈ξ のコード〉          /* code.c, l. 58, 反復パート */
                            jmp     loc2          /* code.c, l. 59, 無条件に分岐 */
                        loc1:                     /* code.c, l. 60 */
```

コード生成の例

以下に，ソースコードの一部(fact.q, l. 9)と，それが生成した目的コード(fact.s, ll. 26-38)の例を示す．



while 文のコード生成の例を以下に示す．(fact.q, ll. 28-31, fact.s, ll. 80-107)



文字列定数の扱い

プログラム中に現れる文字列定数(リテラル)は，内部の文字列表に，新たに生成したラベルとともに格納される．同じ内容の文字列が既に文字列表に存在する場合は，新たに格納することはせず，そのエントリのラベルを用いる．いずれにしても，その際生成されるコードは，上述のように，そのラベルの実効アドレスを %eax に代入する命令となる．(cf. 上記「式の目的コード」の「文字列定数(リテラル)」の項; asm.c, ll. 76-90)

文字列データの実体は，大域変数宣言と関数定義が一通りコンパイルされた後，下のような形でまとめて生成される．(図 1 の「リテラル」の領域)(cf. qc.y, l. 158; asm.c, ll. 93-99)

〈ラベル〉:　.ascii 〈文字列〉, "¥0"

覗き穴最適化

Q言語コンパイラでは，-4 オプションをつけることで，簡単な覗き穴最適化を行うようになっている．以下にその例を示す．なお，一部はこのQ言語コンパイラが生成するコード特有の性質に依存している部分もあるので，必ずしも一般的に適用可能な最適化ばかりとは限らない．(行番号は，すべて optimize.c のもの)

1. 連続する無条件分岐の2つ目は無効なので除去する．(ll. 34-38)

```
jmp    loc1
jmp    loc2          ⇒      jmp    loc1
```

2. 直後のラベルに分岐する分岐命令は無意味なので除去する．(ll. 39-44, ll. 135-142)

```
       jmp    loc?
loc1:                        loc1:
 …                    ⇒       …
locn:                        locn:
```

3. 0 の加減算は無意味なので除去する．(ll. 45-48)

```
add    α, 0          ⇒      (除去)
sub    α, 0          ⇒      (除去)
```

4. 1 の加減算は inc/dec 命令に置き換える．(ll. 49-54)

```
add    α, 1          ⇒      inc    α
sub    α, 1          ⇒      dec    α
```

5. 1 の乗除算は無意味なので除去する．(ll. 55-58)

```
imul   α, 1          ⇒      (除去)
mdiv   α, 1          ⇒      (除去)
```

6. 0 の乗算は，定数の 0 と同じ．(ll. 59-61)

```
imul   α, 0          ⇒      mov    α, 0
```

7. レジスタ %eax 経由の push を直接 push．(ll. 62-76)

```
mov    %eax, α
push   %eax          ⇒      push   α


lea    %eax, loc
push   %eax          ⇒      push   offset loc
```

8. スタック経由のデータ移動を直接移動に.（ll. 77-92）

```
push   α
OP     r1, β        ⇒     mov    r2, α
pop    r2                 OP     r1, β
```
（ただし, *r2* ≠ *r1* かつ *r2* ≠ *β*）

9. レジスタ %ebx 経由の加減乗を, 直接行うかたちにする.（ll. 93-104）

```
mov    %ebx, α
mov    %eax, β        ⇒     mov    %eax, β
OP     %eax, %ebx           OP     %eax, α
```
（ただし, *α* は %eax を含まず, *β* は %ebx を含まず, *OP* は mdiv 以外）

10. %eax が 0 のときの %eax によるインデックスアドレッシングは無意味なので修正.（ll. 105-112）

```
mov    %eax, 0
mov    %eax, [%ebp+%eax*4± α]     ⇒     mov    %eax, [%ebp± α]
```

11. 関数本体の末尾で %esp を復元する直前の「add　%esp, *α*」は無意味なので除去.
（ll. 113-121, ll. 145-153）

```
add    %esp, α
jmp    _func_r        ⇒     jmp    _func_r
```

```
       add    %esp, α
loc1:                ⇒     loc1:
  …                            …
locn:                        locn:
_func_r:                     _func_r:
```

最適化の例

fact.q の 9 行目,「 r = n*fact(n-1); 」の目的コードの最適化された例を示す. push/pop 命令がかなり減っている.（fact3.s は -3 オプションでコンパイルした結果で, 通常の fact.s と同じ. fact4.s は -4 オプションでコンパイルした結果.）

〔最適化前　(fact3.s)〕

```
mov    %eax, 1
push   %eax
mov    %eax, [%ebp+8]
pop    %ebx
sub    %eax, %ebx
push   %eax
call   fact
add    %esp, 4
push   %eax
mov    %eax, [%ebp+8]
pop    %ebx
imul   %eax, %ebx
mov    [%ebp-4], %eax
```

〔最適化後　(fact4.s)〕

```
mov    %eax, [%ebp+8]
dec    %eax
push   %eax
call   fact
add    %esp, 4
mov    %ebx, %eax
mov    %eax, [%ebp+8]
imul   %eax, %ebx
mov    [%ebp-4], %eax
```