Chapter.1

アセンブラ短歌入門

1.1 ● アセンブラ短歌を見てみる …… 02
1.2 ● アセンブラ短歌を作る …… 10
1.3 ● 長い文字列を出力してみる …… 22
1.4 ● さらに長い文字列に挑戦する …… 36

1.1 アセンブラ短歌を見てみる

1.1.1 三十一バイトの機械語コードで何ができるか

前置きは抜きにして，アセンブラ短歌というものに早速入門してみよう．まずはリスト1.1を見てみてほしい．三十一バイト（みそひとバイト）の機械語コードで何ができるか．

リスト1.1: waka.S

```
001:        .section .text
002:        .global main
003:        .type   main, @function
004: main:
005:        mov     $0x616b6157, %eax
006:        push    %ebx
007:        push    %eax
008:        mov     $4, %edx
009:        mov     $1, %ebx
010:        mov     $4, %eax
011:        mov     %esp, %ecx
012:        int     $0x80
013:        pop     %eax
014:        xor     %eax, %eax
015:        pop     %ebx
016:        ret
```

これはx86向けのアセンブラによるプログラムだ．このプログラムをアセンブルして実行すると，単に「Waka」と出力される．本当は「Tanka」（短歌）と出したかったのだが，4バイトが何かと都合が良いので出力文字列は「Waka」（和歌）としてみた．

リスト1.1で行っているのは，各種パラメータの設定をしてwriteシステムコールを呼び出しているだけだ．似たような動作をするプログラムをC言語で書くとするならば，リスト1.2のようになるだろうか．

リスト1.2: waka.c

```
#include <unistd.h>

int main()
{
  write(1, "Waka", 4);
  return 0;
}
```

リスト1.1とリスト1.2は，だいたい等価なプログラムとも言える．いや実はリスト1.2は関数コール，リスト1.1は自前でシステムコール命令を呼んでいるため厳密に等価ではないのだがまあ深くは追求しないことにしよう．大枠は同じだ．

リスト1.1のアセンブラ・プログラムの解説を簡単にしておこう．これの全体はmain()関数に相当する．まず1行目はおまじないだとして気にしなくていい．また2〜4行目はmain()関数の定義なので，これも気にしなくていいだろう．本書の目的はアセンブラ短歌なので，枝葉は追わずに本題を見よう．重要なのは5行目以降だ．

1.1.2 アセンブラで書かれた関数を理解する

5行目には「mov」という命令がある．これは値の移動を行う命令だ．ここではmov命令によって，「EAX」というレジスタに4バイトのデータを格納している．レジスタはCPUが持つ，数値の保存場所だ．つまりこれによって「EAX」という場所になんらかのデータが保存されたと思えばいい．レジスタというのがよくわからなければ，まあCPUが持っている変数だと思ってくれてもいい．そして変数名は固定で，個数に限りがある感じだ．

EAXに格納したデータは，7行目のpush命令によってスタックに保存されている．「push」は値をスタックに保存する命令で，具体的にはスタックポインタである「ESP」というレジスタを，保存する値のサイズぶんだけ減算して，その指す先のメモリ上に値をコピーする．このような操作を「スタックにデータを保存する」と言っているわけだ．これは「スタックを獲得する」などと言うこともある．

7行目の前には，6行目でやはりpush命令によってEBXレジスタの内容をスタックに保存している．その前にESPを初期化しているような部分は見当たらないので，スタックポインタを初期化せずに利用していることになる．ただmain()関数が呼

ばれるときには，main()の呼び出し元であるスタートアップ・ルーチンによってスタックポインタが初期化されていると考えていいだろう．

さらに8〜11行目でEDX,EBX,EAX,ECXレジスタを順に設定している．どうやらレジスタはEAXとEBXの他にも，ECXやEDXといったものが使えるようだ．そしてmov命令のオペランドの順番に注意だ．リスト1.1では，「コピー元，コピー先」の順になっているようだ．結果として，EDXとEAXレジスタには4，EBXレジスタには1，そしてECXレジスタにはESP，つまりスタックポインタの値が格納される．

リスト1.1のプログラムは，実はLinux向けだ．Linuxカーネル上でシステムコールを呼び出すときには，第1引数をEBXレジスタ，第2引数をECXレジスタ，第3引数をEDXレジスタのように格納し，さらにEAXレジスタに呼び出したいシステムコール番号を格納して int 0x80という命令を呼べばいいことになっている．8〜11行目ではwriteシステムコールを呼び出すためのレジスタの準備をしているわけだ．そして12行目で int 0x80 によりシステムコールを呼び出している．

EBXレジスタには1が，EDXレジスタには4が格納されているわけだから，これらがwriteシステムコールの第1引数と第3引数として渡される値になる．リスト1.2を見ればわかるように，これらはファイルディスクリプタと出力サイズだ．

そしてちょっとわかりにくいのは第2引数だ．ここにはどのような値が渡されることになるのだろうか．

リスト1.1の7行目ではpush命令でスタック上にデータを積み，11行目ではESPレジスタをECXレジスタにコピーしている．つまりECXレジスタはスタックに積んだデータを指していて，これが第2引数として渡されることになる．そしてwriteシステムコールの第2引数は，書き込むデータのアドレスを指している．

ということはC言語風に書くと，以下のようなwriteシステムコールの呼び出しが行われることになる．

```
write(1, ESP, 4);
```

つまり標準出力に4バイトのデータを書き込んでいるわけだ．書き込まれるデータはESPの指す先なので，pushによってスタック上に積まれた4文字が出力されることになる．

その後は13行目でpop命令によってスタックポインタを元に戻している．popはpushの逆の動作をする命令だ．具体的にはスタックポインタの指す先から値をコピーし，スタックポインタを値のサイズぶんだけ加算する．これは「スタックから値を取得する」「スタックを解放する」などと呼ばれる．ここでは7行目のpushによってスタック上に保存されたデータがEAXにコピーされ，スタックが解放されることになる．実際にはEAXに代入した値をこの後利用することはないのでEAXの利用はダミーであり，popの目的はスタックを解放してスタックポインタの辻褄を合わせることだ．

14行目はxorという命令が出てきている．これはまた後で説明するが，ここではひとまずEAXをゼロクリアしているということだけ説明しておく．

さらに15行目では，pop命令によってEBXレジスタの内容をスタックから復旧している．これにより，6行目で保存していた値がEBXに戻されることになる．EBXレジスタは9行目で「1」という値が設定されているのだが，15行目のpopにより関数呼び出し前の状態に戻されるわけだ．他のレジスタはとくにこのようなことは行っていないが，EBXレジスタのみ内容を保存・復旧しているのは，これが関数内部で値を破壊されないという約束になっているからだ．

16行目のretは，関数から戻るための命令だ．これによってmain()関数から呼び出し元に返ることになる．関数の戻り値はEAXレジスタで返されるが，EAXは14行目でゼロが設定されているので，戻り値ゼロでmain()関数から返ることになる．つまりC言語的に言えば「return 0」をしているわけだ．そしてmain()からの戻り値はそのままexit()の引数に与えられるので，このプログラムは終了コードがゼロで終了することになる．

さてリスト1.1には実は，本書で今後扱うアセンブラの知識が凝縮されている．

リスト1.1で利用されている命令は「mov」「push」「pop」「int 0x80」「xor」「ret」の6つだけだ．そして出てくるレジスタはEAX，EBX，ECX，EDX，ESPの5つだけだ．実は本書で扱うアセンブラ・プログラムは，そのほとんどの部分がこれらの命令とレジスタの組み合わせで構成されているのだ．

そしてスタックの操作やレジスタの保存と復旧，システムコール呼び出し，スタックの解放，戻り値の設定といった動作も理解が必要だが，これらもリスト1.1に出てきている．

なのでリスト1.1の内容がきちんと理解できれば，本書はそれほどの苦はなく読み進めることができるだろう．本書の目的はアセンブラの理解ではなく，アセンブラ短歌を楽しむことだ．しかし理解の不足のために「楽しむ」ということに注力できなかったとしたらそれはもったいない．そのためにもリスト1.1は，理解できるまで繰り返し読み込んでみてほしい．

1.1.3 実行してみよう

見るだけではつまらないので，実行してみよう．

リスト1.1のプログラムはLinux/x86向けに書かれているので，ＰＣ上で何らかのGNU/Linuxディストリビューションを利用することで動かすことができる．ディストリビューションはUbuntuやCentOSなど，何でもいいだろう．ただしコンパイラやアセンブラなどのプログラミング環境をインストールしておく必要がある．まあ要するにgccをインストールしておいてということだが，これはapt-getやyumで可能だろう．

まずはリスト1.1のプログラムを「waka.S」というファイル名で作成して保存する．空白文字とタブ文字の区別は，あまり気にしなくていい．そして以下のコマンドを実行することで，Waka.Sをアセンブルして実行ファイルを作成することができる．注意はファイルの拡張子を大文字の「S」にしておくことで，後はgccが拡張子を見て適切に判断して実行ファイルを作成してくれる．ちなみに大文字の「S」はアセンブル前にプリプロセッサを通す，という意味になる．小文字の「s」だとプリプロセッサを通さないので，#defineなどを使いたい場合には要注意だ．

```
$ gcc waka.S -o waka
$ ./waka
Waka$
```

このように「Waka」と出力されれば成功だ．改行コードが無いためにbashのプロンプトである「$」が続けて出てしまっているが，まあ細かいことは気にしない．

ちなみに今後，生成した実行ファイルを解析する機会があるかもしれない．このようなとき，実行ファイルが静的リンクされていると理解しやすい．gccで静的リンクするためにはリンクオプションである -static を利用するが，静的リンク用のスタティックライブラリがインストールされていないとうまくいかない．これは標準でインストールされているディストリビューションもあれば，そうでないものもあるようだ．以下のように -static を付加してうまくアセンブルできるかどうか試してみるといいだろう．

```
$ gcc waka.S -o waka -static
```


これがうまくいかなかった場合には，例えばCentOSでは以下のようにしてスタティックライブラリをインストールしておくといいかもしれない．プロンプトが「#」になっているのは，スーパーユーザーで実行するという意味だ．

```
# yum install glibc-static
```

さらに注意としてFreeBSDなどのBSD系ＯＳでは，うまく動かないプログラムもある．システムコール体系が異なるためだ．リスト1.1のプログラムも，そのままではFreeBSDでは正常に動作しない．実はFreeBSDのLinuxエミュレーション機能をうまく使えば動作させることはできそうなのだが，興味のある読者のかたは試してみてほしい．

1.1.4 5・7・5・7・7の機械語コード

さてここまででwriteシステムコールの説明をしてきたが，何がどう短歌なのかがわかっていない．機械語コードを見てみよう．

まずは生成された実行ファイルである「waka」を逆アセンブルしてみる．逆アセンブルには，おそらくobjdumpが使えるだろう．次のように実行してみよう．

```
$ objdump -d waka > waka.d
```

生成されたwaka.dはそれなりに大きなサイズだが，main()関数を検索するとリスト1.3のような箇所があるはずだ．アドレスは微妙に異なるかもしれないが，あまり気にする必要はない．

リスト1.3: waka.d

```
08048404 <main>:
 8048404:       b8 57 61 6b 61          mov    $0x616b6157,%eax
 8048409:       53                      push   %ebx
 804840a:       50                      push   %eax
 804840b:       ba 04 00 00 00          mov    $0x4,%edx
 8048410:       bb 01 00 00 00          mov    $0x1,%ebx
 8048415:       b8 04 00 00 00          mov    $0x4,%eax
 804841a:       89 e1                   mov    %esp,%ecx
 804841c:       cd 80                   int    $0x80
 804841e:       58                      pop    %eax
```

```
804841f:      31 c0                  xor    %eax,%eax
8048421:      5b                     pop    %ebx
8048422:      c3                     ret
```

リスト1.3を詳しく見てみよう．左端には配置先アドレス，右側にはアセンブラのニーモニック，そして中央には実際の機械語コードが出力されているようだ．この機械語コードが実際にメモリ上に配置され，CPUによって順に実行されるわけだ．

ここで，機械語コードをリスト1.4のように区分けしてみるとどうだろうか．

リスト1.4: waka.dを5・7・5・7・7で区分けする

```
8048404:      b8 57 61 6b 61         mov    $0x616b6157,%eax

8048409:      53                     push   %ebx
804840a:      50                     push   %eax
804840b:      ba 04 00 00 00         mov    $0x4,%edx

8048410:      bb 01 00 00 00         mov    $0x1,%ebx

8048415:      b8 04 00 00 00         mov    $0x4,%eax
804841a:      89 e1                  mov    %esp,%ecx

804841c:      cd 80                  int    $0x80
804841e:      58                     pop    %eax
804841f:      31 c0                  xor    %eax,%eax
8048421:      5b                     pop    %ebx
8048422:      c3                     ret
```

これが本書で言う，「5・7・5・7・7の機械語コードでプログラムを書く」ということだ．リスト1.4では最初のmov命令が5バイト，次の2つのpush命令とmov命令が1バイト，1バイト，5バイトで計7バイトだ．さらに次のmov命令は5バイト，次の2つのmov命令は計7バイト，最後の5命令は2バイト，1バイト，2バイト，1バイト，1バイトで計7バイトだ．

つまり，機械語コードの区切りが5・7・5・7・7のバイト数に揃っている．このような機械語コードになるようなアセンブラ・プログラムを，本書ではアセンブラ短歌と呼んでいる．

バイト列を抜き出して5・7・5・7・7に並べると，以下のようになるだろうか．このようにすると，実に短歌っぽく見えることだろう．

```
b8 57 61 6b 61
53 50 ba 04 00 00 00
bb 01 00 00 00
b8 04 00 00 00 89 e1
cd 80 58 31 c0 5b c3
```

1.2 アセンブラ短歌を作る

1.2.1 文字列出力をアセンブラで書く

リスト1.1のようなアセンブラ短歌は，どのようにして作成できるだろうか．アセンブラ短歌の作りかたの理解のために，ゼロから実際に作っていく手順を説明しよう．

まずはベースとして，単に文字列を出力するだけのプログラムをアセンブラで書いてみる．文字列の出力はwriteシステムコールで行える．そしてLinuxでは各種レジスタに引数を積んでEAXレジスタにシステムコール番号を設定し，int 0x80 を実行すればシステムコールが呼ばれる．あとはLinuxカーネルの仕事だ．アプリケーション側では int 0x80 を呼ぶまでのレジスタ設定を責任持ってやっておけば，あとはLinuxカーネルがよきにはからってくれる．

「Waka」という文字列をwriteシステムコールによって出力したいときに，int 0x80 の呼び出しまでに必要な設定内容をもう一度まとめてみよう．まず第1引数は，EBXレジスタに格納する．writeシステムコールの第1引数は出力先のファイルディスクリプタだが，これには標準出力を指す「1」を格納すればよい．

第2引数はひとまず飛ばして，第3引数について考える．第3引数はEDXレジスタに格納するが，writeシステムコールの第3引数は出力サイズだ．「Waka」は4文字なので，「4」という値を格納すればいいことになる．

少し面倒なのは第2引数だ．これは出力データのアドレスを渡すことになる．そして第2引数を格納するのはECXレジスタだ．よって「Waka」という文字列をどこかに作成し，そこのアドレスをECXレジスタに格納すればいい．一番簡単なのはスタックに文字列をデータとして積み，スタックポインタの値をECXレジスタにコピーしてしまうことだ．「Waka」という文字列はアスキーコードにすると 0x57 0x61 0x6b 0x61 になる．よって 0x57616b61 という4バイトデータをスタックに積んでみよう．

残るのはEAXレジスタへのシステムコール番号の設定だ．Linux/x86でのシステムコール番号は最新のカーネル・ソースコードではarch/x86/syscalls/syscall_32.tbl に記述されていて，writeシステムコールは以下のように4番になっている．よってEAXレジスタには「4」という値を設定すればいい．

```
4       i386    write                   sys_write
```

これらの設定後に int 0x80 を呼び出し，あとはret命令で返ればいいだろう．これを5・7・5・7・7を意識せずにアセンブラで書くと，リスト1.5のようになる．ret命令の直前のaddは加算命令で，スタックポインタを4加算することで，1命令目のpushで獲得したスタックを解放している．このようにすればpop命令を使わなくても，スタックは解放できる．

リスト1.5: waka-00.S

```
001:        .section .text
002:        .global main
003:        .type   main, @function
004: main:
005:        push    $0x57616b61
006:        mov     %esp, %eax
007:        mov     $4, %edx
008:        mov     %eax, %ecx
009:        mov     $1, %ebx
010:        mov     $4, %eax
011:        int     $0x80
012:        add     $4, %esp
013:        ret
```

実行ファイルを作成して動作を確認してみよう．リスト1.5を「waka-00.S」というファイル名で保存し，以下のようにしてアセンブルする．

```
$ gcc waka-00.S -o waka-00
```

実行してみよう．

```
$ ./waka-00
akaW$
```

ううむ，「akaW」と出力されてしまった．文字列が逆転しているようだ．

1.2.2 エンディアンの考慮

リスト1.5を見直してみよう．文字列の逆転は，おそらくx86がリトルエンディアンであるためだろう．これはリスト1.5の5行目でpushしているデータ列を逆転させれば解決する．

ということでリスト1.6のような修正をしてみよう．なお本書ではプログラムを修正する際には，修正量が少ない場合はリスト1.6のようなunified diff形式による差分で示すことにする．このほうが修正点が明確になるので，慣れている人にはわかりやすいだろう．unified diff形式を簡単に説明すると，先頭が「-」の行は削除し，先頭が「+」の行は追加するということを示す．つまりリスト1.6は，「push $0x57616b61」の行を「push $0x616b6157」のように書き換えるという意味になる．

リスト1.6: waka-00.S → waka-01.S

```
        .section .text
        .global main
        .type   main, @function
 main:
-       push    $0x57616b61
+       push    $0x616b6157
        mov     %esp, %eax
        mov     $4, %edx
        mov     %eax, %ecx
        mov     $1, %ebx
        mov     $4, %eax
        int     $0x80
        add     $4, %esp
        ret
```

waka-00.Sにリスト1.6の修正をして，waka-01.Sというファイルを作成する．アセンブルして実行してみると「Waka」のように出力された．今度は問題なさそうだ．x86のアセンブラ短歌では，エンディアンに注意しなければならないということか．


```
$ gcc waka-01.S -o waka-01
$ ./waka-01
Waka$
```

1.2.3 調整して短歌にする

出力はできたので，これを5・7・5・7・7の短歌にしていってみよう．まずは実行ファイルであるwaka-01の逆アセンブル結果を見てみるために，objdumpを実行する．

```
$ objdump -d waka-01 > waka-01.d
```

生成したwaka-01.dの中からmain()関数の箇所を検索すると，リスト1.7のような部分が見つかった．さすがに都合良く5・7・5・7・7になっているようなことはなさそうだ．実際に5・7・5・7・7に区切ろうとすると，5命令目のmovが5バイト命令で邪魔をしている．またリスト1.7の機械語コードのバイト数を数えると，30バイトになっている．三十一バイトには1バイト不足しているようだ．

リスト1.7: waka-01.d

```
08048404 <main>:
 8048404:       68 57 61 6b 61          push   $0x616b6157
 8048409:       89 e0                   mov    %esp,%eax
 804840b:       ba 04 00 00 00          mov    $0x4,%edx
 8048410:       89 c1                   mov    %eax,%ecx
 8048412:       bb 01 00 00 00          mov    $0x1,%ebx
 8048417:       b8 04 00 00 00          mov    $0x4,%eax
 804841c:       cd 80                   int    $0x80
 804841e:       83 c4 04                add    $0x4,%esp
 8048421:       c3                      ret
```

4命令目と5命令目を逆転して，さらに終端にnopを入れて調整してみよう．nopは「No OPeration」の略で，「何もしない」という命令のことだ．x86はnopが1バイト命令なので，こういうときに便利に使える．具体的には，リスト1.8のような修正だ．レジスタの設定順番を変えているだけなので，これでも論理的には問題ないはずだ．修正後のファイルはwaka-02.Sというファイル名にしよう．

リスト1.8: waka-01.S → waka-02.S

```
        .section .text
        .global main
        .type   main, @function
 main:
        push    $0x616b6157
        mov     %esp, %eax
        mov     $4, %edx
-       mov     %eax, %ecx
        mov     $1, %ebx
+       mov     %eax, %ecx
        mov     $4, %eax
        int     $0x80
        add     $4, %esp
+       nop
        ret
```

waka-02.S をアセンブルして，さらに逆アセンブルした結果を見てみよう．

```
$ gcc waka-02.S -o waka-02
$ objdump -d waka-02 > waka-02.d
```

waka-02.d はリスト1.9のようになった．こんどは5・7・5・7・7にきれいに区分けできる．

リスト1.9: waka-02.d

```
08048404 <main>:
 8048404:       68 57 61 6b 61          push   $0x616b6157
 8048409:       89 e0                   mov    %esp,%eax
 804840b:       ba 04 00 00 00          mov    $0x4,%edx
 8048410:       bb 01 00 00 00          mov    $0x1,%ebx
 8048415:       89 c1                   mov    %eax,%ecx
 8048417:       b8 04 00 00 00          mov    $0x4,%eax
 804841c:       cd 80                   int    $0x80
 804841e:       83 c4 04                add    $0x4,%esp
 8048421:       90                      nop
 8048422:       c3                      ret
```


1.2.4 戻り値を設定する

さて筆者は，このようなサンプルプログラムの作成時はMakefileを書いて，makeコマンドによってビルドや実行などが行えるようにしてある．

しかし実はここまでのプログラムはmakeコマンドにとっては都合が悪い．main()からの戻り値が未設定のため，ゼロ以外の終了コードでexit()してしまうのだ．このためmakeから実行するとエラー扱いになってしまう．回避手段は当然あってMakefile中のコマンド実行の先頭に「-」を付けるだけで対策できるのだが，このままでは完成度が低いことは否めないだろう．

ということで終了コードを設定してみよう．関数からの戻り値はEAXレジスタで返されるから，ret命令の前にEAXにゼロを代入しておけば，main()がreturn 0 で返ることと同等になる．nop命令を削って，そのかわりにmov命令でEAXレジスタにゼロを代入してみよう．リスト1.10のような修正だ．

リスト1.10: waka-02.S → waka-03.S

```
        .section .text
        .global main
        .type   main, @function
 main:
        push    $0x616b6157
        mov     %esp, %eax
        mov     $4, %edx
        mov     $1, %ebx
        mov     %eax, %ecx
        mov     $4, %eax
        int     $0x80
        add     $4, %esp
-       nop
+       mov     $0, %eax
        ret
```

リスト1.10の修正を加えてアセンブル・逆アセンブルしてみると，リスト1.11のようになる．機械語コードのバイト数を数えてみよう．うーむ，5・7・5・7・7にはなっていない．アセンブラ短歌が崩れてしまっているわけだ．

リスト1.11: waka-03.d

```
08048404 <main>:
 8048404:       68 57 61 6b 61          push   $0x616b6157
 8048409:       89 e0                   mov    %esp,%eax
 804840b:       ba 04 00 00 00          mov    $0x4,%edx
 8048410:       bb 01 00 00 00          mov    $0x1,%ebx
 8048415:       89 c1                   mov    %eax,%ecx
 8048417:       b8 04 00 00 00          mov    $0x4,%eax
 804841c:       cd 80                   int    $0x80
 804841e:       83 c4 04                add    $0x4,%esp
 8048421:       b8 00 00 00 00          mov    $0x0,%eax
 8048426:       c3                      ret
```

リスト1.10の問題はret命令の直前のmov命令で，リスト1.11を見ると，mov命令の命令長は5バイトにもなってしまっている．これはEAXレジスタにゼロを代入しているのだが，ゼロが「0x00000000」という4バイト値で表現されているために命令長が伸びてしまっているようだ．

1.2.5 命令サイズの削減

単なるゼロクリアを，もっとシンプルに記述することはできないだろうか．実はこのようなときによく使われる「xor命令」を使った書き方があるのだが，この方法でも機械語コードは2バイトにできるだけで三十一バイトには収まらず解決しない．またここでmov命令を使うことは，その前の引数の設定部分でのmov命令の利用に対して韻をふんでいるようにも思えて可能ならば残してみたい．ということでここでは全体の命令サイズを削減することで，伸びた命令長のぶんを調整してみよう．

ということで不要な命令を削ってみる．リスト1.10を見ると，まず2命令目の「mov %esp, %eax」と，5命令目の「mov %eax, %ecx」が無駄なように思える．ESPレジスタを一旦EAXレジスタにコピーして，その後にECXレジスタにコピーしているからだ．つまりEAXレジスタを中継させているわけだがこれはなんとなくそうしてしまっただけで，とくに意味はない．「mov %esp, %ecx」のようにして，ECXレジスタに直接コピーしてしまって構わないだろう．これで1命令節約できる．

さらに8命令目の「add $4, %esp」はスタック解放のためのものだが，これはpop命令でスタックから値を取り出すことでも代用できる．そしてpop命令は1バイト命令なので，popにすることで命令サイズを節約できたりもする．するとリスト1.12のように書けることになる．

リスト1.12: waka-03.S → waka-04.S

```
        .section .text
        .global main
        .type   main, @function
 main:
        push    $0x616b6157
-       mov     %esp, %eax
        mov     $4, %edx
        mov     $1, %ebx
-       mov     %eax, %ecx
+       mov     %esp, %ecx
        mov     $4, %eax
        int     $0x80
-       add     $4, %esp
+       pop     %eax
        mov     $0, %eax
        ret
```

逆アセンブル結果を見てみよう．今度はどうなっているだろうか．

リスト1.13: waka-04.d

```
08048404 <main>:
 8048404:       68 57 61 6b 61          push   $0x616b6157
 8048409:       ba 04 00 00 00          mov    $0x4,%edx
 804840e:       bb 01 00 00 00          mov    $0x1,%ebx
 8048413:       89 e1                   mov    %esp,%ecx
 8048415:       b8 04 00 00 00          mov    $0x4,%eax
 804841a:       cd 80                   int    $0x80
 804841c:       58                      pop    %eax
 804841d:       b8 00 00 00 00          mov    $0x0,%eax
 8048422:       c3                      ret
```

リスト1.13は5・7・5・7・7にはなっていないが，全体は三十一バイトになっているので，命令の順番を工夫することでアセンブラ短歌にできそうだ．よくよく見ると3命令目と4命令目を逆転させるだけで，論理的な変更なく調整できそうに

思える．リスト1.14のような修正だ．

リスト1.14: waka-04.S → waka-05.S

```
        .section .text
        .global main
        .type   main, @function
 main:
        push    $0x616b6157
        mov     $4, %edx
-       mov     $1, %ebx
        mov     %esp, %ecx
+       mov     $1, %ebx
        mov     $4, %eax
        int     $0x80
        pop     %eax
        mov     $0, %eax
        ret
```

逆アセンブル結果はリスト1.15のようになった．

リスト1.15: waka-05.d

```
08048404 <main>:
 8048404:       68 57 61 6b 61          push   $0x616b6157
 8048409:       ba 04 00 00 00          mov    $0x4,%edx
 804840e:       89 e1                   mov    %esp,%ecx
 8048410:       bb 01 00 00 00          mov    $0x1,%ebx
 8048415:       b8 04 00 00 00          mov    $0x4,%eax
 804841a:       cd 80                   int    $0x80
 804841c:       58                      pop    %eax
 804841d:       b8 00 00 00 00          mov    $0x0,%eax
 8048422:       c3                      ret
```

今度はきれいに5・7・5・7・7に収まった．mov命令の4連続も，韻を踏んでいて小気味良い．

1.2.6 EBXを保存する

さてここまでで，レジスタは自由に使ってしまってきている．これはこれで果していいのだろうか．関数内で値を保存・復旧する必要はないのだろうか．

関数内でのレジスタの使いかたは一般的にABI（Application Binary Interface）という規約によって決められており，コンパイラはそれに従ったアセンブラコードを出力するように作られる．そうでないと，異なるコンパイラでビルドしたオブジェクトファイルどうしをリンクしたときに，レジスタの使いかたに食い違いが出るからだ．逆にいえばABIに従ってさえいれば異なるコンパイラで生成したオブジェクトファイルをリンクできるし，アセンブラで書いたプログラムも問題なくリンクできる．

そして関数内でのレジスタの扱いには，「関数から返るときに，値が変更されていてもいい」というものと「関数から返るときには，値が変更されていてはいけない」というものの2種類がある．前者は「揮発性」（volatile），後者は「不揮発性」（non-volatile）などと呼ばれたりする．どのレジスタが揮発性でどのレジスタが不揮発性なのかは，ABIで定義されている．

これはCPUのハードウェア仕様ではなく，レジスタの使いかたとしてのソフトウェア仕様だ．揮発性のレジスタは関数内部で自由に使ってよいが，不揮発性のレジスタを関数内部で使う場合には，スタックに値を保存して関数からのリターン時には値を復旧するなどの処理が必要になる．関数の呼び出し元からすれば，不揮発性のレジスタは値が変化していないことを期待して，そのようなコードがコンパイラによって生成されているからだ．

x86向けgccではEAX,ECX,EDXは揮発性だが，ESPとあと実はEBXは不揮発性のレジスタとして扱っている．このためEAX,ECX,EDXは関数内部で自由に使ってよいが，EBXはそうはいかない．

よってリスト1.14には問題がある．EBXレジスタの値が書き換えられてしまうため，main()からリターンした後の動作の保証がないのだ．とはいってもmain()からのリターン後は戻り値をそのまま引数にしてexit()が呼ばれるだけということが多いため実害はなさそうで，まあとりあえずは動作しているようだが，完成度としてはいまひとつだ．

EBXはシステムコールの呼び出しのために値の設定が必要なため，使わないということはできない．ということでリスト1.14に，EBXを保存・復旧するコード

を追加してみよう．

リスト1.16: waka-05.S → waka-06.S

```
        .section .text
        .global main
        .type   main, @function
 main:
+       push    %ebx
        push    $0x616b6157
        mov     $4, %edx
        mov     %esp, %ecx
        mov     $1, %ebx
        mov     $4, %eax
        int     $0x80
        pop     %eax
        mov     $0, %eax
+       pop     %ebx
        ret
```

リスト1.14はちょうど三十一バイトに収まっていたがリスト1.16では命令数が増加してしまったため，このままでは5・7・5・7・7にすることは難しそうだ．節約できる箇所はないだろうか．

以前にEAXレジスタでの戻り値設定の際に少しだけ触れたが，x86ではレジスタのゼロクリアにxor命令が利用できる．具体的には「xor %eax, %eax」のような命令を利用することで，EAXレジスタをゼロクリアすることができる．

これは「EAX ^ EAX → EAX」のような演算を行うもので，同じものどうしをXORするとゼロになることから，特定のレジスタをゼロクリアする際の（x86での）常套手段として利用されるものだ．このやりかたの良いところは，機械語コードが少ないバイト数で済むということだ．

xor命令を使って戻り値設定を書き換えて，さらに機械語コードを5・7・5・7・7にそろえるために細かい命令調整をしてみたのがリスト1.17だ．

リスト1.17: waka-07.S

```
001:        .section .text
002:        .global main
003:        .type   main, @function
004: main:
005:        mov     $0x616b6157, %eax
006:        push    %ebx
007:        push    %eax
008:        mov     $4, %edx
009:        mov     $1, %ebx
010:        mov     $4, %eax
011:        mov     %esp, %ecx
012:        int     $0x80
013:        pop     %eax
014:        xor     %eax, %eax
015:        pop     %ebx
016:        ret
```

関数の終了時にはEBXレジスタのpopが必要だが，これは（必須ではないが）なるべくならretの直前に行ったほうがすっきりする．そのためには文字列データのpushの前にEBXレジスタのpushを行う必要があるが，リスト1.16は命令サイズが5バイトのmov命令が多く，調整が難しい．そこで文字列データをスタックに直接pushするのでなく，いったんEAXレジスタに格納してからpushすることにした．これならばEBXレジスタのpushを文字列データのpushよりも前に持っていくことができる．

さて実は，これが冒頭に紹介したリスト1.1のアセンブラコードになる．

リスト1.17を見てどうだろうか．確かに工夫次第ではコード量はもっと削減できるし，他にも技術的な意味での改善の余地は多くあるだろう．

しかしリスト1.17は，mov命令で各レジスタを順に設定していて整然としているし，pushに対するpopの対称もとれている．戻り値設定や不揮発性のレジスタの保存も行っていて関数としての体裁もある．引数設定部分はmov命令の連続で，韻をふんでいるような趣きもある．これがきちんと5・7・5・7・7に整列していると考えたとき，見てて飽きないような美しさを感じてしまうのは筆者だけであろうか．

なのでこれはこれで整った，味わい深いコードと言えるのではないだろうか．アセンブラ短歌は，単なるショート・コーディングを競うものではない．そもそも命令コードを5・7・5・7・7に整列させることに技術的な意味や目的はないわけだから，このような「味わい深さ」や「美しさ」もアセンブラ短歌のテーマの1つだと思う．

1.3 長い文字列を出力してみる

1.3.1 8文字の出力

さてここまでで「Waka」という文字列を出力するアセンブラ短歌を書いてみたわけだが，もっと長い文字列は出力できないものだろうか．

現在はpush命令で4バイトのデータをスタックに格納することで，出力文字列を準備している．2回pushを行えば，8バイトの文字列が準備できるはずだ．やってみよう．

しかしリスト1.17はmov命令でEAXを経由して文字列設定をしているためやりにくい．ということでリスト1.14をベースにしてpushを増やしてみる．EBXの保存が行われないことになるが，ひとまずは気にしない．

するとリスト1.18のようになった．waka-05.Sからの変更としては，2つのpush命令で8バイトの文字列を用意した．さらに格納データはバイト列で表現するとわかりにくいので，文字表記とシフト演算で表現するようにした．もちろんリトルエンディアンなので，データ表現としては逆転させた形になっている．出力させる文字列は「Waka」（和歌）に続けて短歌に関係ある言葉ということで，「若山牧水」から「Wakayama」としてみた．酒と旅を愛した，自然主義の歌人だ．

さらにpushの直後のmov命令でEDXレジスタに設定する値を，8に変更している．これはwriteシステムコールの第3引数，つまり出力文字数に相当する値だ．さらにスタックの辻褄合わせのため，終端でのpopをもう1つ追加している．

リスト1.18: waka-05.S → waka-08.S

```
        .section .text
        .global main
        .type   main, @function
 main:
-       push    $0x616b6157
-       mov     $4, %edx
+       push    $('W'|'a'<<8|'k'<<16|'a'<<24)
+       push    $('y'|'a'<<8|'m'<<16|'a'<<24)
+       mov     $8, %edx
        mov     %esp, %ecx
        mov     $1, %ebx
        mov     $4, %eax
        int     $0x80
        pop     %eax
+       pop     %eax
        mov     $0, %eax
        ret
```

実行してみよう．

```
$ gcc waka-08.S -o waka-08
$ ./waka-08
yamaWaka$
```

相変わらずbashのプロンプトが続いて出力されてしまっているが，それよりも問題は「Wakayama」でなく「yamaWaka」と出力されてしまっていることだ．スタックへのデータの積みかたを間違えたようだ．

x86のスタックは「フル下降」というタイプだ．いわゆる下方伸長のためスタックポインタの減算方向にスタックは伸び，さらにスタックポインタはスタック上のデータの先頭を指している．スタックには他にも上方伸長や，またスタックポインタが空のスタックの位置を指しているような場合もあるが，x86はそうではない．

下方伸長のため，スタックには「yama」「Waka」の順に文字列を積む必要がある．「フル下降」であるためスタックポインタはデータの先頭を指しているので，ECXレジスタには変わらずそのままスタックポインタの値を代入すればいい．ということで，pushする順番を逆転させてみよう．（リスト1.19）

リスト1.19: waka-08.S → waka-09.S

```
        .section .text
        .global main
        .type   main, @function
 main:
-       push    $('W'|'a'<<8|'k'<<16|'a'<<24)
        push    $('y'|'a'<<8|'m'<<16|'a'<<24)
+       push    $('W'|'a'<<8|'k'<<16|'a'<<24)
        mov     $8, %edx
```

```
        mov     %esp, %ecx
        mov     $1, %ebx
        mov     $4, %eax
        int     $0x80
        pop     %eax
        pop     %eax
        mov     $0, %eax
        ret
```

実行してみよう．今度はうまくいくだろうか．

```
$ gcc waka-09.S -o waka-09
$ ./waka-09
Wakayama$
```

うん，期待通りに出力されているようだ．

1.3.2 xor 命令の利用

出力はできたが，機械語コードはどのようになっているのだろうか．waka-09の逆アセンブル結果はリスト1.20のようになる．

リスト1.20: waka-09.d

```
08048404 <main>:
 8048404:       68 79 61 6d 61          push   $0x616d6179
 8048409:       68 57 61 6b 61          push   $0x616b6157
 804840e:       ba 08 00 00 00          mov    $0x8,%edx
 8048413:       89 e1                   mov    %esp,%ecx
 8048415:       bb 01 00 00 00          mov    $0x1,%ebx
 804841a:       b8 04 00 00 00          mov    $0x4,%eax
 804841f:       cd 80                   int    $0x80
 8048421:       58                      pop    %eax
 8048422:       58                      pop    %eax
 8048423:       b8 00 00 00 00          mov    $0x0,%eax
 8048428:       c3                      ret
```

5・7・5・7・7になっていないどころか，全体で37バイトにもなってしまっていて三十一バイトにはほど遠い．単純に命令数が増加しているためだ．サイズ削減の調整をする必要がありそうだ．

まず考えられるのは，ゼロクリアにxor命令を使うことだ．リスト1.19の戻り値設定の部分を，xor命令を使って書き直してみよう．リスト1.21のような修正になる．

リスト1.21: waka-09.S → waka-10.S

```
        .section .text
        .global main
        .type   main, @function
 main:
        push    $('y'|'a'<<8|'m'<<16|'a'<<24)
        push    $('W'|'a'<<8|'k'<<16|'a'<<24)
        mov     $8, %edx
        mov     %esp, %ecx
        mov     $1, %ebx
        mov     $4, %eax
        int     $0x80
        pop     %eax
        pop     %eax
-       mov     $0, %eax
+       xor     %eax, %eax
        ret
```

リスト1.21をアセンブル・逆アセンブルするとリスト1.22のようになる．

リスト1.22: waka-10.d

```
08048404 <main>:
 8048404:       68 79 61 6d 61          push   $0x616d6179
 8048409:       68 57 61 6b 61          push   $0x616b6157
 804840e:       ba 08 00 00 00          mov    $0x8,%edx
 8048413:       89 e1                   mov    %esp,%ecx
 8048415:       bb 01 00 00 00          mov    $0x1,%ebx
 804841a:       b8 04 00 00 00          mov    $0x4,%eax
 804841f:       cd 80                   int    $0x80
 8048421:       58                      pop    %eax
 8048422:       58                      pop    %eax
 8048423:       31 c0                   xor    %eax,%eax
 8048425:       c3                      ret
```

関数終端での戻り値設定の5バイトのmov命令が，xor命令に変更されて2バイトで済んでいる．3バイトの節約だ．しかし全体としては34バイトで，まだまだ削減が必要だ．

レジスタをxor命令でゼロクリアできるということを利用して，さらにサイズを削減できないだろうか？例えばリスト1.21では，レジスタに「8」や「1」といった定数値を代入するために5バイトの命令を用いている．これは代入する値を4バイトの定数値として命令中に保持しているためだが，もっと工夫できないだろうか．

1.3.3 レジスタの下位バイトのみ設定する

x86のアーキテクチャは8080の8ビット・アセンブラがニーモニック互換で16ビット拡張されて8086となり，さらにそれが80386に進化して32ビット拡張されているという歴史的な経緯がある．このためレジスタを8ビット，16ビット部分のみ操作できるという特徴があるのだが，レジスタ全体をいったんゼロクリアして，さらに定数値を8ビット単位で代入することで，命令サイズを削減することができる．例えば

```
mov     $8, %edx
```

のような代入は，

```
xor     %edx, %edx
mov     $8, %dl
```

のようにしても動作は等価だ．EDXは32ビットレジスタだが，DLはその中の下位8ビット部分を指している．まあ厳密にはフラグレジスタの変化などがあるのかもしれないので完全に等価というわけではないかもしれないが，ここでは気にする必要はない．

そのようにしてリスト1.21のレジスタ代入を書き換えてみよう．リスト1.21はEDX，EBX，EAXレジスタに定数値を代入しているので，それらの3箇所をすべてxorとmov命令の組み合せに変更してみるとリスト1.23のようになる．

リスト1.23: waka-11.S

```
001:         .section .text
002:         .global main
003:         .type   main, @function
004: main:
005: /*5*/   push    $('y'|'a'<<8|'m'<<16|'a'<<24)
006: /*5*/   push    $('W'|'a'<<8|'k'<<16|'a'<<24)
007: /*2*/   xor     %edx, %edx
008: /*2*/   mov     $8, %dl
009: /*2*/   mov     %esp, %ecx
010: /*2*/   xor     %ebx, %ebx
011: /*2*/   mov     $1, %bl
012: /*2*/   xor     %eax, %eax
013: /*2*/   mov     $4, %al
014: /*2*/   int     $0x80
015: /*1*/   pop     %eax
016: /*1*/   pop     %eax
017: /*2*/   xor     %eax, %eax
018: /*1*/   ret
```

リスト1.23では命令長をわかりやすくするために，ニーモニックの頭にコメントで命令長を追加してみた．mov命令による4バイト値の代入だと1命令が5バイトになっていたが，xor命令との組み合わせだと2＋2の4バイトで書くことができる．つまり1箇所につき1バイトの削減だ．3箇所を修正することで3バイトを節約し，全体としてはちょうど三十一バイトになっている．

1.3.4 8文字を出力するアセンブラ短歌

リスト1.23はそのままだと5・7・5・7・7になっていないのだが，終端にあるpop命令が1バイトなので，これを前に持ってくることでうまく調整できそうだ．1つは5命令目の後，もう1つは7命令目の後に移動することで，論理が破綻することなく調整できる（もっともこれでは，解放後のスタックを参照してしまうのだが）．そのように移動して，さらに5・7・5・7・7の区切りで空行を入れたのがリスト1.24だ．

リスト1.24: waka-11.S → waka-12.S

```
        .section .text
        .global main
        .type   main, @function
 main:
 /*5*/  push    $('y'|'a'<<8|'m'<<16|'a'<<24)
+
 /*5*/  push    $('W'|'a'<<8|'k'<<16|'a'<<24)
 /*2*/  xor     %edx, %edx
+
 /*2*/  mov     $8, %dl
 /*2*/  mov     %esp, %ecx
+/*1*/  pop     %eax
+
 /*2*/  xor     %ebx, %ebx
 /*2*/  mov     $1, %bl
+/*1*/  pop     %eax
 /*2*/  xor     %eax, %eax
+
 /*2*/  mov     $4, %al
 /*2*/  int     $0x80
-/*1*/  pop     %eax
-/*1*/  pop     %eax
 /*2*/  xor     %eax, %eax
 /*1*/  ret
```

リスト1.24はうまくアセンブラ短歌にできている．アセンブルして実行してみよう．


```
$ gcc waka-12.S -o waka-12
$ ./waka-12
Wakayama$
```

実行も問題ないようだ．逆アセンブル結果はリスト1.25のようになる．

リスト1.25: waka-12.d

```
08048404 <main>:
 8048404:       68 79 61 6d 61          push   $0x616d6179
 8048409:       68 57 61 6b 61          push   $0x616b6157
 804840e:       31 d2                   xor    %edx,%edx
 8048410:       b2 08                   mov    $0x8,%dl
 8048412:       89 e1                   mov    %esp,%ecx
 8048414:       58                      pop    %eax
 8048415:       31 db                   xor    %ebx,%ebx
 8048417:       b3 01                   mov    $0x1,%bl
 8048419:       58                      pop    %eax
 804841a:       31 c0                   xor    %eax,%eax
 804841c:       b0 04                   mov    $0x4,%al
 804841e:       cd 80                   int    $0x80
 8048420:       31 c0                   xor    %eax,%eax
 8048422:       c3                      ret
```

リスト1.25は一応アセンブラ短歌にはなっている．しかしダミーのpopが散らばっているし，解放後のスタックを参照してしまってもいる．EBXの保存もなく，これはちょっと完成度が低いかなと思わなくもない．

1.3.5 push/popによる値の設定

さらにコードサイズを削減して余裕を作ってみよう．

リスト1.24ではxor命令とmov命令でのレジスタ下位8ビットへの代入を利用することで，レジスタへの値の代入を4バイトの命令で表現している．

スタックを利用することで，さらに小さなサイズで書くことはできないだろうか．例えばEDXレジスタに「8」という値を代入したいときには，「8」という定数値をいったんスタックにpushし，さらにEDXレジスタにpopするとどうなるか．「8」はスタックを経由してEDXレジスタに代入されることになる．リスト1.24の定数値の代入をそのようにごっそり書き換えたのがリスト1.26だ．

リスト1.26: waka-13.S

```
001:         .section .text
002:         .global main
003:         .type   main, @function
004: main:
005: /*5*/   push    $('y'|'a'<<8|'m'<<16|'a'<<24)
006: /*5*/   push    $('W'|'a'<<8|'k'<<16|'a'<<24)
007:         push    $8
008:         pop     %edx
009: /*2*/   mov     %esp, %ecx
010:         push    $1
011:         pop     %ebx
012:         push    $4
013:         pop     %eax
014: /*2*/   int     $0x80
015: /*1*/   pop     %eax
016: /*1*/   pop     %eax
017: /*2*/   xor     %eax, %eax
018: /*1*/   ret
```

リスト1.26をアセンブルし，さらに逆アセンブルした結果はリスト1.27のようになる．

リスト1.27: waka-13.d

```
08048404 <main>:
 8048404:       68 79 61 6d 61          push   $0x616d6179
 8048409:       68 57 61 6b 61          push   $0x616b6157
 804840e:       6a 08                   push   $0x8
 8048410:       5a                      pop    %edx
 8048411:       89 e1                   mov    %esp,%ecx
 8048413:       6a 01                   push   $0x1
 8048415:       5b                      pop    %ebx
 8048416:       6a 04                   push   $0x4
 8048418:       58                      pop    %eax
 8048419:       cd 80                   int    $0x80
 804841b:       58                      pop    %eax
 804841c:       58                      pop    %eax
 804841d:       31 c0                   xor    %eax,%eax
 804841f:       c3                      ret
```


リスト1.27を見ると，スタックへの1バイト値のpush命令は2バイトの機械語コード，EDXレジスタへのpopは1バイトの機械語コードで表現できるようだ．機械語コードは合計で3バイトなので，xor命令の利用の場合に対してさらに1バイト削減できる．リスト1.27では3箇所について修正しているので，全体としては28バイトになっている．なおEAXへのダミーのpopによるスタック調整は関数の終端に戻しておいた．これにより解放後のスタックを参照してしまうという問題は解決する．

余っている3バイトをnopで埋めて三十一バイトにすると，偶然にもリスト1.26はそのままアセンブラ短歌になる．機械語コードのサイズをコメントで追加して，さらに5・7・5・7・7に分割したのがリスト1.28だ．

リスト1.28: waka-14.S

```
001:         .section .text
002:         .global main
003:         .type   main, @function
004: main:
005: /*5*/   push    $('y'|'a'<<8|'m'<<16|'a'<<24)
006:
007: /*5*/   push    $('W'|'a'<<8|'k'<<16|'a'<<24)
008: /*2*/   push    $8
009:
010: /*1*/   pop     %edx
011: /*2*/   mov     %esp, %ecx
012: /*2*/   push    $1
013:
014: /*1*/   pop     %ebx
015: /*2*/   push    $4
016: /*1*/   pop     %eax
017: /*2*/   int     $0x80
018: /*1*/   pop     %eax
019:
020: /*1*/   pop     %eax
021: /*2*/   xor     %eax, %eax
022: /*1*/   nop
023: /*1*/   nop
024: /*1*/   nop
025: /*1*/   ret
```

1.3.6 nopの削除

そしてここでリスト1.28の終端にある3バイトのnopと，EAXレジスタによる戻り値の返しかたについて考えてみる．

mov命令による4バイト値の代入は，5バイトの命令長になる．ということはxorによるゼロクリアをmov命令による代入に戻せば，ちょうど三十一バイトに収まることになる．リスト1.29のような修正だ．

リスト1.29: waka-14.S → waka-15.S

```
        .section .text
        .global main
        .type   main, @function
 main:
 /*5*/  push    $('y'|'a'<<8|'m'<<16|'a'<<24)

 /*5*/  push    $('W'|'a'<<8|'k'<<16|'a'<<24)
 /*2*/  push    $8

 /*1*/  pop     %edx
 /*2*/  mov     %esp, %ecx
 /*2*/  push    $1

 /*1*/  pop     %ebx
 /*2*/  push    $4
 /*1*/  pop     %eax
 /*2*/  int     $0x80
 /*1*/  pop     %eax

 /*1*/  pop     %eax
-/*2*/  xor     %eax, %eax
-/*1*/  nop
-/*1*/  nop
-/*1*/  nop
+/*5*/  mov     $0, %eax
 /*1*/  ret
```

リスト1.28とリスト1.29を比較して，どのように感じるだろうか？ショート・コーディングという視点で見ると，優れているのはもちろんリスト1.28だ．EAXへのゼロ代入を2バイトのxor命令で行えていて，3つのnopを作り出すことができているからだ．

しかしアセンブラ短歌は単純に短さだけを競うものではなく，美しさや完成度を求めるものだ．そういった観点で考えると，リスト1.28のように余った部分をとりあえずnopで埋めておくようなコードは，ちょっと安易だといえるだろう．

リスト1.29はそのような「安易なnop」を無くすために，ゼロ代入にxorでなくあえてmov命令を使うことでちょうど三十一バイトに合わせているわけだ．このようなものは，より完成度の高い歌であると言うことができるだろう．

1.3.7 やっぱりEBXを保存する

しかししばらく忘れていたが，EBXは不揮発性のレジスタだ．このため関数内部でEBXレジスタを利用する場合には，本来は保存と復旧が必要だ．この対応はリスト1.16で追加されたが，リスト1.18の作成のときに削られてしまってそのままになってしまっている．

ということでリスト1.29のmovはやっぱりxorに戻して，空いた3バイトでEBXレジスタの保存と復旧を行うようにしてみる．movをxorにそのまま戻してEBXのpopを追加し，全体的に命令を後ろにずらすことで前のほうにスペースを作りEBXのpushを追加することを考える．やってみたら1バイト余るので，xorでのゼロクリアはpushとpopに置き換えることでサイズ調整し，nopで埋めずに済むようにしてみたのがリスト1.30だ．

リスト1.30: waka-16.S

```
001:         .section .text
002:         .global main
003:         .type   main, @function
004: main:
005: /*1*/   push    %ebx
006: /*2*/   push    $1
007: /*2*/   push    $4
008:
009: /*1*/   pop     %eax
010: /*1*/   pop     %ebx
011: /*5*/   push    $('y'|'a'<<8|'m'<<16|'a'<<24)
012:
```

```
013: /*5*/   push    $('W'|'a'<<8|'k'<<16|'a'<<24)
014:
015: /*2*/   mov     %esp, %ecx
016: /*2*/   push    $8
017: /*1*/   pop     %edx
018: /*2*/   int     $0x80
019:
020: /*2*/   push    $0
021: /*1*/   pop     %eax
022: /*1*/   pop     %ebx
023: /*1*/   pop     %ebx
024: /*1*/   pop     %ebx
025: /*1*/   ret
```

実行結果も確認しておこう．

```
$ gcc waka-16.S -o waka-16
$ ./waka-16
Wakayama$
```

問題ない．そして逆アセンブル結果は，リスト1.31のようになった．

リスト1.31: waka-16.d

```
08048404 <main>:
 8048404:       53                      push   %ebx
 8048405:       6a 01                   push   $0x1
 8048407:       6a 04                   push   $0x4
 8048409:       58                      pop    %eax
 804840a:       5b                      pop    %ebx
 804840b:       68 79 61 6d 61          push   $0x616d6179
 8048410:       68 57 61 6b 61          push   $0x616b6157
 8048415:       89 e1                   mov    %esp,%ecx
 8048417:       6a 08                   push   $0x8
 8048419:       5a                      pop    %edx
 804841a:       cd 80                   int    $0x80
 804841c:       6a 00                   push   $0x0
 804841e:       58                      pop    %eax
 804841f:       5b                      pop    %ebx
 8048420:       5b                      pop    %ebx
 8048421:       5b                      pop    %ebx
 8048422:       c3                      ret
```


リスト1.31はEBXレジスタの保存を行っており関数としての体裁を保っている．スタックもレジスタ保存のあとに文字列のためのワーク領域として使っており，さらに関数終了時は戻り値設定，スタック解放，EBXの復旧という標準的な順番になっている．解放済みのスタックを参照しているようなこともない．実に様式美が感じられる歌になっている．

そして安易なnopもないし，int 0x80 の呼び出しが7バイトの句の終端にあって，最後の句は関数から戻るための処理としてまとまっていることも爽やかだ．これはリスト1.29に比べて，歌としての完成度はより上がっているように思える．

ううむ美しい．なのでこれもアセンブラ短歌による文字列出力のひとつの完成型として，ここでバイト列表記しておこう．

```
53 6a 01 6a 04
58 5b 68 79 61 6d 61
68 57 61 6b 61
89 e1 6a 08 5a cd 80
6a 00 58 5b 5b 5b c3
```

1.4 さらに長い文字列に挑戦する

1.4.1 12文字の出力

8文字の出力を行うことはできたが，もっと長い文字列を出力することは可能だろうか？

そのためにはさらにスタックにデータを積む必要がある．リスト1.26が28バイトで8文字を出力できているので，これをベースにするのが改造しやすそうだ．さらに4文字追加して，12文字を出力してみよう．修正したのがリスト1.32だ．リスト1.30でせっかく行ったEBXの保存対応だが，リスト1.32ではまた戻ってしまっている．しかしこれ以上の出力拡張のためにはもう無理だろうから，これはもうあきらめることにしよう．

リスト1.32: waka-17.S

```
001:         .section .text
002:         .global main
003:         .type   main, @function
004: main:
005: /*5*/   push    $('B'|'o'<<8|'k'<<16|'u'<<24)
006: /*5*/   push    $('y'|'a'<<8|'m'<<16|'a'<<24)
007: /*5*/   push    $('W'|'a'<<8|'k'<<16|'a'<<24)
008: /*2*/   push    $12
009: /*1*/   pop     %edx
010: /*2*/   mov     %esp, %ecx
011: /*2*/   push    $1
012: /*1*/   pop     %ebx
013: /*2*/   push    $4
014: /*1*/   pop     %eax
015: /*2*/   int     $0x80
016: /*3*/   add     $12, %esp
017: /*1*/   ret
```

リスト1.32では大改造を行っているので，少し詳しく説明しよう．まず12文字の出力のために，スタックへのpushを追加している．出力する文字列は「若山牧水」として続けて，「WakayamaBoku」としてみた．さらに文字列の長さの指定として，EDXレジスタに代入する値を8から12に変更した．

関数末尾でのダミーのpopによるスタックの調整は，3つ必要になる．これには3バイトが必要になるが，実はスタックポインタを直接減算して調整するとしても3バイトの命令になるので変わらない．ならばとスタックポインタの減算による1命令にまとめてしまった．

スタックへのpushは5バイト命令で，それが3つもあることになるので，そのままではさすがに三十一バイトに収めることは難しい．ということで戻り値の返却のためのEAXレジスタの設定は，この際あきらめて削除した．

リスト1.32のコードサイズは惜しくも32バイトだ．あと1バイト，サイズを削減する必要があるのだが，ひとまず命令順を調整して，5・7・5・7・7に区分けしてみよう．3つ目のpush命令とECXへのスタックポインタのコピーを後ろのほうに持っていき，EBXレジスタへのpopの位置をずらすことでうまくまとめられそうだ．するとリスト1.33のようになる．

リスト1.33: waka-18.S

```
001:         .section .text
002:         .global main
003:         .type   main, @function
004: main:
005: /*5*/   push    $('B'|'o'<<8|'k'<<16|'u'<<24)
006:
007: /*5*/   push    $('y'|'a'<<8|'m'<<16|'a'<<24)
008: /*2*/   push    $12
009:
010: /*1*/   pop     %edx
011: /*2*/   push    $1
012: /*2*/   push    $4
013:
014: /*1*/   pop     %eax
015: /*1*/   pop     %ebx
016: /*5*/   push    $('W'|'a'<<8|'k'<<16|'a'<<24)
017:
018: /*2*/   mov     %esp, %ecx
019: /*2*/   int     $0x80
020: /*3*/   add     $12, %esp
021: /*1*/   ret
```

リスト1.33は5・7・5・7・7の区切りで空行を入れてあるが，最後の句が8バイトになっている．つまり字余りだ．まああくまで短歌と考えて字余りを許せばこれでも可なのだが，やはりもう少し完成度を上げたい．1バイト，なんとか節約できないものだろうか．

リスト1.33では関数終端でのスタックの解放を，add命令によりESPレジスタに12を加算することで行っている．これは3バイト命令になっているが，工夫のしがいはないだろうか．

1.4.2 システムコールの戻り値を利用する

ここまで来ると，ちょっとズルい方法を使わないと無理だろう．実はこの直前ではint 0x80によってwriteシステムコールが呼ばれている．writeシステムコールの返り値は，出力したバイト数だ．そしてLinux/x86ではシステムコールの返り値はEAXレジスタで返される．ということはこの直後，EAXレジスタには12という値が格納されていることになる．

これをESPレジスタに加算してやれば，結果としてはESPレジスタは12加算され辻褄が合うことになる．つまり「add %eax, %esp」のようなことをすればいい．実際に試してみて逆アセンブラを出力させてみると，これは2バイト命令であることがわかる．これでちょうど三十一バイトになるわけだ．ついでにEDXレジスタのpopの位置を移動して，pushとpopをまとめてみよう．そのように修正したのがリスト1.34だ．

リスト1.34: waka-19.S

```
001:        .section .text
002:        .global main
003:        .type   main, @function
004: main:
005: /*5*/   push    $('B'|'o'<<8|'k'<<16|'u'<<24)
006:
007: /*5*/   push    $('y'|'a'<<8|'m'<<16|'a'<<24)
008: /*2*/   push    $12
009:
010: /*2*/   push    $1
011: /*2*/   push    $4
012: /*1*/   pop     %eax
013:
014: /*1*/   pop     %ebx
015: /*1*/   pop     %edx
016: /*5*/   push    $('W'|'a'<<8|'k'<<16|'a'<<24)
017:
018: /*2*/   mov     %esp, %ecx
019: /*2*/   int     $0x80
020: /*2*/   add     %eax, %esp
021: /*1*/   ret
```


実行してみよう．

```
$ gcc waka-19.S -o waka-19
$ ./waka-19
WakayamaBoku$
```

12文字が出力されている．逆アセンブラはどうなっているだろうか．objdumpによりwaka-19.dを作成してみるとリスト1.35のようになった．

リスト1.35: waka-19.d

```
08048404 <main>:
 8048404:       68 42 6f 6b 75          push   $0x756b6f42
 8048409:       68 79 61 6d 61          push   $0x616d6179
 804840e:       6a 0c                   push   $0xc
 8048410:       6a 01                   push   $0x1
 8048412:       6a 04                   push   $0x4
 8048414:       58                      pop    %eax
 8048415:       5b                      pop    %ebx
 8048416:       5a                      pop    %edx
 8048417:       68 57 61 6b 61          push   $0x616b6157
 804841c:       89 e1                   mov    %esp,%ecx
 804841e:       cd 80                   int    $0x80
 8048420:       01 c4                   add    %eax,%esp
 8048422:       c3                      ret
```

どうだろうか．pushとpopが整然と並んでいて，だいぶきっちりと詰めた感じだ．そして12文字も出力できていて，実に勢いが感じられる歌だ．

戻り値ゼロを返していないこととEBXレジスタの値が破壊されていることが心残りではあるが，これはこれでかなり完成度の高いアセンブラ短歌だと言えるので

はないだろうか．最後にこれをバイト列表記して，本章は終りとしよう．

```
68 42 6f 6b 75
68 79 61 6d 61 6a 0c
6a 01 6a 04 58
5b 5a 68 57 61 6b 61
89 e1 cd 80 01 c4 c3
```