最新 アセンブリ言語 詳解

北山洋幸◉著

ソフトウェア開発の能力を深化させる
アセンブリ／ベクトルプログラミングの基本と応用

CUTT
カットシステム

■**サンプルファイルのダウンロードについて**

本書掲載のサンプルファイルは、下記 URL からダウンロードできます。

https://----------

・本書の内容についてのご意見、ご質問は、お名前、ご連絡先を明記のうえ、小社出版部宛文書（郵送または E-mail）でお送りください。
・電話によるお問い合わせはお受けできません。
・本書の解説範囲を越える内容のご質問や、本書の内容と無関係なご質問にはお答えできません。
・匿名のフリーメールアドレスからのお問い合わせには返信しかねます。

はじめに

近年においてソフトウェアエンジニアがハードウェアのアーキテクチャを意識する機会は激減しました。それは悪いことではないでしょう。ソフトウェアエンジニアがハードウェアを意識していては、開発したソフトウェアはハードウェア依存となりポータビリティやスケーラビリティなどを失ってしまいます。

ソフトウェアをアセンブリ言語で開発していた時代は遙か昔のこととなり、現在では C++ 言語などは言うにおよばす、仮想マシン上で動作する言語を利用する機会が多くなりました。より抽象化が進んだ環境を使用するようになったと言ってよいでしょう。つまり、現代のソフトウェアエンジニアが、CPU の動作やアセンブリ言語を理解することが必要となる機会は減っています。

とはいえ、ソフトウェアエンジニアを自負するなら、基礎を習得することは重要です。基礎を理解していると、時代が移りインプリメント法やアーキテクチャが変わろうとも、新しい時代へ柔軟に対応できるでしょう。なぜならば、動作の表面だけを理解しているのではなくシステムの基本を理解しているからです。

本書の目的は、抽象化の進んだ言語のみを利用しているソフトウェアエンジニアに、アセンブリ言語を理解してもらうことです。その延長上で CPU 自体の動作についても、理解できるでしょう。アセンブリ言語をひととおり理解しようとすると大変な労力がかかり、必要に迫られていない限り途中で挫折するでしょう。そこで、本書はリファレンス的にアセンブリ命令のすべてを網羅して理解するのではなく、比較的簡単な命令を理解することによって、アセンブリ言語、ひいては CPU がどのようにプログラムを解釈実行するかを理解していただくことにつとめました。リファレンスよりアプリケーションに重きを置き、興味を失わないように努めました。

リファレンスは用意しましたが、紙面の関係もあり十分でありません。アセンブリ命令を詳しく知りたい人は、参考文献を参照してください。

本書の対象読者は，以下のような人を対象としています。

- 低水準言語が、どのようなものか体験したい人
- アセンブリ言語を理解したい人
- SIMD などベクトル処理を理解したい人

是非、本書を参考に現代の CPU やアセンブリ言語の実際を理解し、コンピューターの基礎の理解や、高性能なプログラムの開発に役立ててください。微力ながら本書が学習の助けになれば幸いです。

謝辞

出版にあたり、お世話になった株式会社カットシステムの石塚勝敏氏に深く感謝いたします。

2023 年冬　都立東大和南公園にて 北山洋幸

本書の使用にあたって

開発環境、および、実行環境を説明します。

- **プラットフォーム**

  Windows 10 Home を開発・実行環境とします。一部のプログラムは、Ubuntu 22.04.1 LTS、Windows 10 Pro でも確認しています。

- **オペレーティングシステムのビット**

  主に 64 ビット Windows を使用します。いくつかのプログラムは、Ubuntu 20.04.2 LTS で確認しました。

- **コンパイラー**

  Windows では、無償の Visual Studio Community 2022 を使用します。Ubuntu では、g++ バージョン 11.3.0、clang++ バージョン 14.0.0-1ubuntu1 を利用しました。

- **CPU**

  最新の命令を使用しているプログラムは、比較的古い CPU が搭載されたパソコンでは直接実行できません。ただ、エミュレーターが提供されていますので、速度は若干低下しますが、古いパソコンであっても動作させるのは可能です。

- **エミュレーター**

  自身の環境が最新の命令を使用できる CPU でない場合、Intel Software Development Emulator（Intel SDE）を利用してください。これについては、付録 A で詳しく解説します。

- **URL**

  URL の記載がある場合、執筆時点のものであり変更される可能性もあります。リンク先が存在しない場合、キーワードなどから自分で検索してください。

- **バイト、ワード、ダブルワード、クワッドワード**

  バイトは 8 ビット（1 バイト）、ワードは 16 ビット（2 バイト）、ダブルワードは 32 ビット（4 バイト）、クワッドワードは 64 ビット（8 バイト）です。高級言語では使用しませんが、アセンブリ言語を記述する場合、このような表現が使用されます。

- **アドレス**

  あるオブジェクトが格納されているメモリー位置のことです。

- **アセンブリ言語**

  プログラミング言語の一つで、コンピューターが直接解釈・実行できるマシン語と 1 対 1 に対応した言語です。いわゆる低水準言語（低級言語）の代表です。アセンブリ言語はマシン語の命令と 1 対 1 に対応するニーモニック（英単語）などを使い記述します。現在では、抽象化された高水準言語が主流ですが、CPU 性能を限界まで引き出すチューニングを行いたい場合や、組み込みソフトウェアの開発などハードウェアに近い分野の開発などでは、今でも使われています。

- **アセンブラー**

  アセンブリ言語で記述されたソースコードを、コンピューターが解釈できるマシン語へ翻訳するソフトウェアのことです。C++ 言語などに対するコンパイラーと同じ位置付けです。

- **カタカナ語の長音表記**

  「メモリー」や「フォルダー」など、最近は語尾の「ー」を付けるのが一般的になっています。なるべく、最近の表現に合わせましたが統一していません。

- **ソースリストとソースコード**

  基本的に同じものを指しますが、ソースリストと表現する場合はソース全体を、ソースコードと表現する場合はソースの一部を指す場合が多いです。

- **命令やレジスター名**

  SIMD 命令やレジスター名の表現で、大文字と小文字の表現が混在します。例えば、`ZMM1` と `zmm1`、そして `vmovdqa64` と `VMOVDQA64` は同じものです。

- **Linux と Ubuntu**

  Linux と Ubuntu を混在して使用していますが、本書のプログラムを動作させたのは、すべて Ubuntu 上のみです。ただ、Ubuntu に限らず Unix（Linux）系と説明したほうが良さそうな場合は、Linux を用います。

- **AVX-512 と AVX512**

  AVX512 と記述する場合と、AVX-512 と記述する場合がありますが同じものを指します。

- **BOM コード**

  サンプルのソースファイルは BOM コードが付与されているときがあります。一般的に BOM コードの有無は処理に影響しませんが、まれに BOM コードを意識しない環境があり、正常にファイルを扱えないときがあります。そのような場合は BOM コードを削除してください。BOM コードはファイルの先頭に格納されています。

目次

第1章

はじめてのアセンブリ言語

1

早速ですが、簡単な代入をアセンブリ言語で記述してみましょう。本書は主に、Windows の Visual Studio に付属する ml64 や cl コマンドを利用します。もちろん、Visual Studio のプロジェクトからビルドしても構いません。これに加え、Linux 系 OS（Ubuntu）の g++ や clang++ のインラインアセンブラーを利用する方法も解説します。

1.1 Windows の例

まず、Windows の例を示します。以降に、ソースリストを示します。

リスト1.1●mov.cpp（01begin）

```
#include <iostream>

int main(void)
{
    int a = 0;

    __asm   mov a, 1

    std::cout << "a = " << a << std::endl;

    return 0;
}
```

C++ 言語内にアセンブリ言語の記述を行う場合、インラインアセンブラを使用すると簡単です。この例では、__asm キーワードに続きアセンブラコードを記述します。mov 命令はデータを移動する命令です。__asm に続き、アセンブラコードを記述しますが、コードの最後にセミコロンは必要ありません。本プログラムは、1 を変数 a へ格納します。このプログラムの実行結果を示します。

```
a = 1
```

この例ではアセンブリ命令が一つだけですので、__asm キーワードに続けてすぐにその命令を記述できます。複数のアセンブリ命令を記述したい場合は、それらを {} でくくったものを __asm キーワードに続けて記述します。もちろん、命令が一つだけであっても {} で囲んで構いません。次の二つはどちらも a に 1 を代入します。

```
__asm   mov a, 1
```

```
__asm
{
    mov a, 1
}
```

mov 命令の構文を参照すると分かりますが、ソースが 1 でデスティネーションが a です。つまり、変数 a に 1 が代入されます。

1.1.1 命令の解説

mov 命令

mov 命令を簡単に説明します。名前から類推できるように、mov 命令は、データ転送命令です。8 ビット、16 ビット、または 32 ビット（64 ビットモードでは 64 ビット）のデータを CPU のレジスター同士、メモリーと即値、あるいはレジスターとメモリーの間で移動する命令です。

ニーモニック

```
mov デスティネーションオペランド，ソースオペランド
```

ニーモニックには、Intel 形式と ATT 形式があり、記述方法やオペランド位置に違いがあります。どちらかを理解してしまえば、別のニーモニックを記述するのに苦労しません。ただし、転送元と転送先が逆になるためケアレスミスを起こしやすくなるでしょう。本書では主に Intel 形式のニーモニックを採用します。

ニーモニック

基本的にプログラム内ではアセンブリ命令のニーモニックは小文字入力します。ニーモニックは、大文字でも小文字でも構いません。

■ ビルドと実行

簡単なプログラムなので、プロジェクトは作成せず、Visual Studio をインストールすると同時に使用できるようになる「x86 Native Tools Command Prompt for VS 2022」を使ってビルドします。ビルドと実行の例を次に示します（コマンドラインに入力した部分を太字で示します）。

```
C:¥>cl /EHsc mov.cpp
Microsoft(R) C/C++ Optimizing Compiler Version xx.xx.xxxxx for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

mov.cpp
Microsoft (R) Incremental Linker Version xx.xx.xxxxx.x
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:mov.exe
mov.obj

C:¥>mov
a = 1
```

1.2 Ubuntu の例

前節の Windows 用のプログラム（リスト 1.1）を Ubuntu 用に書き直します。ソースリストは次のようになります。

リスト1.2●movGpp.cpp（01begin）

```
#include <iostream>

int main(void)
{
    int a = 0;

    __asm__ __volatile__ ("movl $1, %0¥n": "=r" (a));

    std::cout << "a = " << a << std::endl;

    return 0;
}
```

__asm__（または asm）はインラインアセンブラを使用することを示します。続く __volatile__（または volatile）は省略可能ですが、そうすると内容によってはアセンブリ言語で記述した部分が最適化対象となってしまいます。アセンブリ言語で記述したコードを確実に実行させたいなら、__volatile__ を指定するのが適切でしょう。

1.2.1 ビルドと実行

gcc（g++）では、コンパイルオプションを指定しない限り、Intel 形式ではなく AT&T 形式を用います。Intel 形式と AT&T 形式では、オペランドの並び順が逆になります。インテル社のマニュアル類は当然 Intel 形式で記述されているため、混乱しないように注意してください。

以降に、ビルドと実行の例を示します（太字は入力した部分です）。

```
$ g++ movGpp.cpp
$ ./a.out
a = 1
```

■ g++ で Intel 形式

gcc（g++）にコンパイルオプションを指定すると、Intel 形式で記述できます。Intel 形式で記述したソースリストを次に示します。

リスト1.3●movGppIntel.cpp（01begin）

```
#include <iostream>

int main(void)
{
    int a = 0;

    __asm__ __volatile__ ("mov %0, 1¥n": "=r" (a));

    std::cout << "a = " << a << std::endl;

    return 0;
}
```

命令が Intel 形式になるとともに、オペランドの位置も Intel 形式となります。Visual Studio、g++、そして clang++ などを併用する場合は、Intel 形式で記述する方が混乱は起きないでしょう。

ビルドと実行の例を次に示します。Intel 形式で記述したときは、コンパイルオプションに **-masm=intel** を指定してください。

```
$ g++ -masm=intel movGppIntel.cpp
$ ./a.out
a = 1
```

■ clang++

コンパイラーを g++ から clang++ へ変更したときのビルドと実行の例も示します。

```
$ clang++ movGpp.cpp
$ ./a.out
a = 1
$ clang++ -masm=intel movGppIntel.cpp
$ ./a.out
a = 1
```

1.3 64 ビット

Windows の 64 ビット対応プログラムを作成する場合、C++ 言語内にインラインアセンブラを記述できません。ここでは、Windows における、64 ビットのアセンブリ言語を解説します。アセンブリ言語の記法は、使用するプロセッサー、オペレーティングシステム、そしてコンパイラーやアセンブラーに依存します。本書は、環境や記法の説明を目的としておらず、アセンブリ言語へフォーカスしたいため、一つの環境に絞って解説します。一つの環境でアセンブリ言語を理解できれば、異なる環境に適合させるのは容易です。プロセッサーが異なる場合は若干困難が伴うでしょうが、基本的な考えを理解しておけばそれほど苦にはなりません。Linux 上で C++ を理解した人が Windows 上の C# を理解するのに大きな困難を伴わないのと同様です。

これ以降は、Windows の 64 ビット環境をメインに解説します。ただ、いくつかの例では g++ などの解説も行います。g++ などはアセンブリ言語をインラインで記述できるため、本節で解説する呼び出し規約やレジスター保護に注意する必要はありません。ただその反面、記述法が少し面倒で、破壊するレジスターを明示的にコンパイラーに知らせる必要があります。

1.3.1 インラインアセンブラが使えない

Visual Studio を使用する場合、プラットフォームに x64 を選ぶとインラインアセンブラを使用することはできません。理由はスタックやレジスターの管理など、いろいろあるのでしょう。インラインアセンブラを使用できないため、アセンブリコードは別のファイルに単独で記述しなければなりません。以降では簡単な例を挙げて、64 ビット環境で記述するときの手順を示します。

1.3.2 ファイル構成

これまでのプログラムのソースファイルは一つでした。ところが、64 ビットでは C++ 言語ファイル中にアセンブリコードを記述できないため、アセンブリコードを記述したファイルが、もう一つ必要です。ファイルとコンパイラーやアセンブラーの概念図を次に示します。

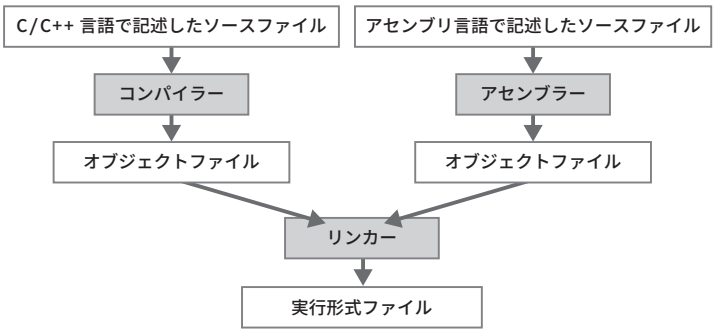

図1.1●ファイルとコンパイラーやアセンブラーの関係

Visual Studio の統合環境を利用する場合は、プロジェクトに含まれるソースファイルは自動的にコンパイラーもしくはアセンブラーによって翻訳されます。リンカーなども自動で起動されますので、C++ 言語で記述したファイルと、アセンブリ言語で記述したファイルが混在しても、単にビルドを選択するだけです。ビルドに関しては、C++ 言語で記述したファイルのみで構成されたプロジェクトと何ら変わりません。ただし、プロジェクトの設定を少し変更する必要があります。これについては付録で解説します。

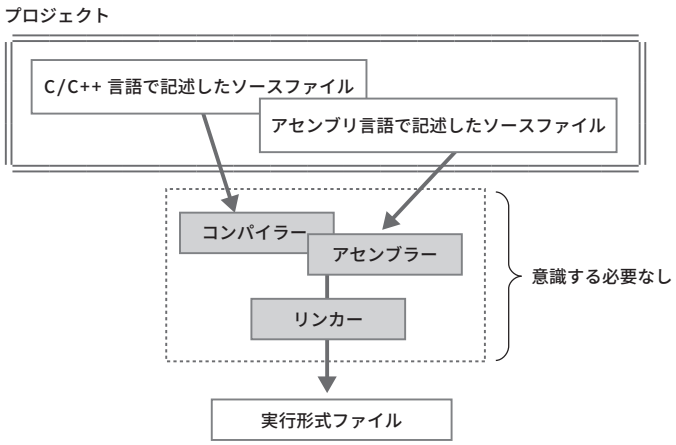

図1.2●Visual Studioプロジェクトのファイル関係

1.3.3 アセンブリ言語で関数を作る

以降に、C++ 言語で記述したソースリストとアセンブリ言語で記述したソースリストを示します。

リスト1.4●movRegister64.cpp（01begin）

```
#include <iostream>
#include <iomanip>

extern "C" void asmCode(char* ca, short* sa, int* ia);

using namespace std;

int main(void)
{
    char ca = 0;
    short sa = 0;
    int ia = 0;

    asmCode(&ca, &sa, &ia);

    cout << hex << uppercase << setfill('0') << right;
    cout << "ca = 0x" << setw(2) << (unsigned int)ca << endl;
    cout << "sa = 0x" << setw(4) << (unsigned int)sa << endl;
    cout << "ia = 0x" << setw(8) << (unsigned int)ia << endl;

    return 0;
}
```

char 型変数 ca、short 型変数 sa、そして int 型変数 ia へ値を設定する簡単なプログラムです。

リスト1.5●movRegister64Asm.asm（01begin）

```
;
; rcx   = &ca
; rdx   = &sa
; r8    = &ia
;asmCode(&ca, &sa, &ia);
;
_TEXT   segment

        public asmCode
```

```
        align  16

asmCode proc
        mov     eax, 01020304h          ; eax=0x01020304
        mov     byte ptr [rcx], al
        mov     word ptr [rdx], ax
        mov     dword ptr [r8], eax
        ret
asmCode endp

_TEXT   ends
        end
```

rcx レジスターに ca のアドレス、rdx レジスターに sa のアドレス、そして r8 レジスターに ia のアドレスが渡されます。まず、eax レジスターに 01020304（16 進数）を設定します。そして、ca に al レジスターを、sa に ax レジスターを、そして ia に eax レジスターの値を代入します。

このプログラムのビルドと実行の例を示します。簡単なプログラムなので、プロジェクトは作成せず、Visual Studio をインストールすると同時に使用できるようになる「x64 Native Tools Command Prompt for VS 2022」を使ってビルドします。Visual Studio をインストールしていても C++ の環境をインストールしていない場合、「x64 Native Tools Command Prompt for VS 2022」は現れません。そのようなときは、Visual Studio に「C++ によるデスクトップ開発」も追加インストールしてください。このプロンプトを使ってビルドすると、64 ビットでビルドされます。

```
C:¥>ml64 /c movRegister64Asm.asm
Microsoft (R) …
…

C:¥>cl /O2 /EHsc movRegister64.cpp movRegister64Asm.obj
…

C:¥>movRegister64
ca = 0x04
sa = 0x0304
ia = 0x01020304
```

当然ですが、64 ビット対応のビルド環境ではインラインアセンブラを記述できませんので、以前紹介した方法でプログラムを記述するとエラーが発生します。

1.3.4 Ubuntu

簡単なプログラムですので、Ubuntu の g++ や clang++ 用のソースリストも示します。Visual Studio と違い、g++ などでは 64 ビット対応のプログラムもインラインでアセンブリ言語を記述できます。

リスト1.6●movRegister01gpp.cpp（01begin）

```
#include <iostream>
#include <iomanip>

using namespace std;

int main(void)
{
    char ca = 0;
    short sa = 0;
    int ia = 0;

    __asm__ __volatile__ (
        "movl $0x01020304, %%eax¥n"
        "movb %%al,  %0¥n"
        "movw %%ax,  %1¥n"
        "movl %%eax, %2¥n"
        : "=r" (ca), "=r" (sa), "=r" (ia)
        :
        : "%eax"
        );

    cout << hex << uppercase << setfill('0') << right;
    cout << "ca = 0x" << setw(2) << (unsigned int)ca << endl;
    cout << "sa = 0x" << setw(4) << (unsigned int)sa << endl;
    cout << "ia = 0x" << setw(8) << (unsigned int)ia << endl;

    return 0;
}
```

ビルドと実行の例を次に示します。