Write Great Code

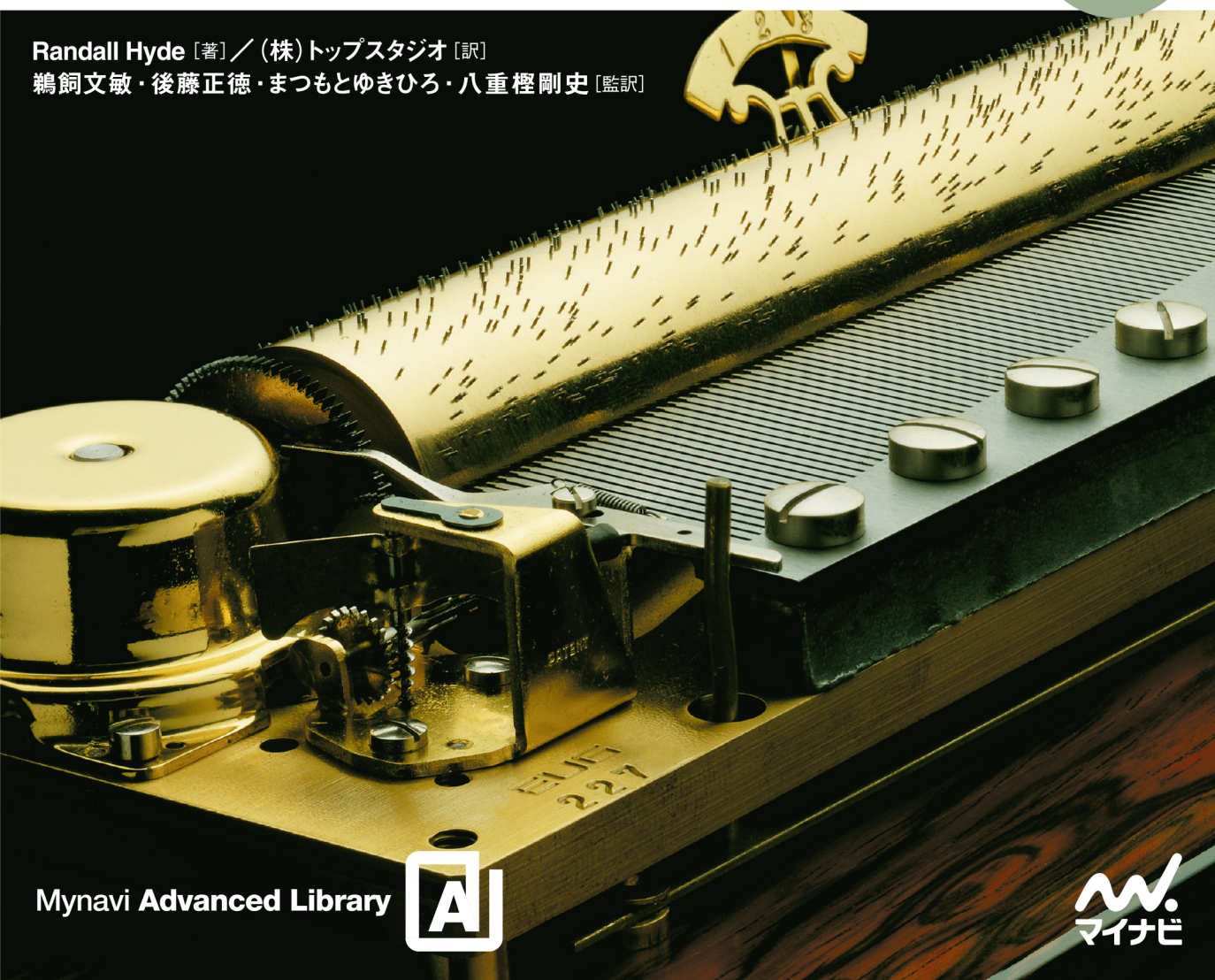
ライト・グレート・コード

Vol.2 — Thinking Low-Level, Writing High-Level

Vol. 2

低いレベルで考え、高いレベルで書く

Randall Hyde［著］／（株）トップスタジオ［訳］
鵜飼文敏・後藤正徳・まつもとゆきひろ・八重樫剛史［監訳］



NO STARCH PRESS

Mynavi Advanced Library

マイナビ

NO STARCH PRESS

Write Great Code

ライト・グレート・コード

Vol.2 — Thinking Low-Level, Writing High-Level

低いレベルで考え、高いレベルで書く

Vol. 2

Randall Hyde［著］／（株）トップスタジオ［訳］

鵜飼文敏・後藤正徳
まつもとゆきひろ・八重樫剛史［監訳］

Mynavi Advanced Library

マイナビ

Title of English-language original : **Write Great Code, Vol. 2**, ISBN 978-1-59327-065-0,
published by No Starch Press.

謝 辞

もともと、本書の内容は『Write Great Code, Volume 1』の最後の章として書かれました。しかし、Volume 1の担当編集者のHillel Heinsteinが、章が長すぎるわりには要点を十分に伝えきれていないのではないかという懸念を示したのです。そこで、我々は元の内容をふくらませて、1冊の本として独立させることにしました。したがって、本書を生み出すきっかけを作ってくれたHillelに最初に感謝しなければなりません。

もちろん、200ページ分の章を基に1冊の本を作り上げるというのは大がかりな作業で、本書の制作には多くの方々がかかわっています。この場をお借りして、それらの方々に感謝の意を表したいと思います。

私のすばらしき友人であるMary Philipsは、『The Art of Assembly Language』の原稿の整理を手伝ってくれましたが、その内容の一部が本書にも反映されています。

発行者のBill Pollockは、このシリーズの有用性を確信し、指導や精神的な援助を与えてくれました。

制作マネージャーでNo Starch Pressの主連絡窓口だったElizabeth Campbellは、このプロジェクトを先導し、現実のものとしてくれました。

編集者のKathy Grider-Carlyleは、文法上の誤りを正すのに力を貸してくれました。

担当編集者のJim Comptonは、本書を読みやすくするためにかなりの時間を費やしてくれました。

校正者のStephanie Provinesは、いくつかのタイプミスとレイアウトの不具合を指摘してくれました。

Riley Hoffmanは、ページレイアウトの調整に日々尽力し、プログラムリストを含む本書全体の体裁を整えてくれました。

同じく本書のレイアウトに携わったChristina Samuellは、制作全般にわたるさまざまな手助けをしてくれました。

テクニカルレビューアのBenjamin David Luntは、本書の技術的な品質を確保することに力を貸してくれました。

Leigh PoehlerとPatricia Witkinは、本書の販売とマーケティングに尽力してくれています。

また、No Starch Pressの前の編集者であるSusan BergeとRachel Gunnにも感謝の意を表したいと思います。両氏は本書の完成前に転任しましたが、プロジェクトに貴重な意見を残してくれました。

最後に、本書を私の姪や甥たち、Gary、Courtney（Kiki）、Cassidy、Vincent、Sarah Dawn、David、Nicholasに捧げます。自分たちの名前が本に載っているのを見て大喜びしてくれるのではないかと期待しています。

監訳者のことば

優れたプログラマーには「ブレーキの壊れた」人が多いように思います。普通の人には常識という名前のブレーキが備わっているので、どこかで「ここまでやるのは変だろう」「これくらいで十分だろう」と思ってしまうものですが、この種の「優れたプログラマー」はそんなことはお構いなしでどんどん先まで進んでいきます。まるで限界など存在しないかのようです。本書の著者であるRandall Hydeは、間違いなく「ブレーキが壊れて」います。いや、誉めているのです。彼は「グレートコード」を追求することに対して妥協がありません。

アセンブラプログラミングが広く教えられなくなってから、ずいぶん経ちます。CPUそのものの挙動を知ることは、「グレートコード」を書くことにとって重要です。しかし、普通はアセンブラを使うのは繁雑で面倒なので、CPUの一般的な知識と注意点を学ぶくらいで止めてしまうものです。しかし、Randallには妥協がありません。彼は自ら高級言語の制御構造を導入したアセンブラであるHLA（High Level Assembler）を開発してまで、アセンブラによる動くコードによる「グレートコード」を私たちに見せつけてくれているのです。

本書の主なテーマはパフォーマンスです。パフォーマンスはプログラマーにとって永遠のテーマの1つと呼んでもよいでしょう。近年見られない妥協ない姿勢でパフォーマンスを追求した本書は、凡庸なプログラマーにとどまりたくない皆さんが、もう一段高いレベルに到達する上で役に立つに違いありません。

2006年11月
まつもとゆきひろ

目 次

はじめに

グレートコードには多くの側面があります。実際、あまりに多すぎて、一冊の本ではとてもきちんと説明しきれません。そこで、『Write Great Code』シリーズの2巻目となる本書では、グレートコードの重要な1要素であるパフォーマンスに的を絞ります。コンピュータシステムのパフォーマンスがメガヘルツから数百メガヘルツ、そしてギガヘルツへと向上するにつれて、ソフトウェアのパフォーマンスはほかの問題ほど重要視されなくなりました。今日では、ソフトウェア技術者が「コードを決して最適化すべきではない」と声を大にして言うことも珍しくありません。奇妙なことに、アプリケーションのユーザーからそうした言葉を聞くことはあまりありません。

本書では効率的なコードの書き方を説明しますが、本書の主題は最適化ではありません。最適化はソフトウェア開発サイクルの終盤近くのフェーズです。ソフトウェア技術者は、この終盤フェーズで、期待したパフォーマンスが出ない原因を究明し、コードに手を加えて解決すべく努力します。しかし残念ながら、最適化フェーズまでアプリケーションのパフォーマンスをまったく考慮しなかった場合、最適化が功を奏する見込みはほとんどありません。アプリケーションにとって適切なパフォーマンスを確保する時期は、フェーズ全体の中で最初のほうにある設計フェーズと実装フェーズなのです。最適化によってシステムのパフォーマンスを微調整することはできても、奇跡を起こすことはめったにできません。

「早まった最適化は諸悪の根源だ」という言葉は、それを広めたDonald Knuth<監訳者注1>の発言として引用されることが多いのですが、最初にそう言ったのはTony Hoare<監訳者注2>です。この言葉はかなり前から、ソフトウェア開発サイクルの最後の最後までアプリケーションのパフォーマンスを考慮せずに済ませようとするソフトウェア技術者のスローガンとなってきました。この段階に至ると、経済的な理由や製品化に要する時間を理由に、最適化フェーズが無視されるのが通例です。しかし、Hoareは「アプリケーション開発の早い段階でパフォーマンスを意識することは諸悪の根源だ」と言ったわけではありません。彼は明確に「**早まった最適化**」と言っており、これは当時、アセンブリ言語コードでCPUサイクルと命令を数えることを意味していました。ベースとなるコードもかなり流動的な、初期のプログラム設計時に行うようなコーディングのことではなかったのです。したがって、Hoareの意見は的を射ていました。この言葉を深読みしすぎることの問題について述べた、Charles Cookの論評（`http://www.cookcomputing.com/blog/archives/000084.html`）の抜粋を次に示します。

> 私は常々、この言葉が本来の意図とは異なる問題領域に適用されてきたために、ソフトウェア設計者が重大な過ちを犯す要因を頻繁に作り出してきたと考えている。
> 元の発言は「97%くらいの時間は、小さな効率化のことを忘れるべきである。早まった最適化は諸悪の根源だ」というもので、私もこれには同意する。通常、パフォーマンスのボトルネックが明らかになる前に、コードのわずかな最適化に多くの時間を費やしても意味がない。その反面、ソフトウェアをシステムレベルで設計するときは、最初から常にパフォーマンスの問題を考慮しておく必要がある。優れたソフトウェア開発者は、パフォーマンスがどこで問題になるかという

<監訳者注1>組版ソフトウェアTeXの開発や『The Art of Computer Programming』シリーズの著者として知られる著名なコンピュータ研究者。
<監訳者注2>QuickSortアルゴリズムの開発者やホーア論理の提唱者として著名なコンピュータ研究者。

> 勘を養っているため、これを無意識に行う。しかし、経験の浅い開発者は、そうしたことを気にかけず、後の段階で少し微調整すればあらゆる問題が解決すると勘違いしている。

Hoareが実際に言ったのは、ソフトウェア技術者は個々のステートメントの実行に必要なCPUサイクル数をはじめとする旧来の最適化を気にする前に、適切なアルゴリズム設計やアルゴリズムの実装などのほかの問題を気にするべきだということでした。

確かに、本書の概念の多くは最適化フェーズで適用できますが、テクニックの大半は最初のコーディング時に行う必要があります。コードの完成に至るまで先延ばしにした場合は、そのソフトウェアにはまず反映されないでしょう。事後にこれらの概念を実装するのは手間がかかりすぎるからです。

本書では、最新の最適化コンパイラによって効率的なマシンコードに変換される適切な高級言語（HLL）ステートメントを選択する方法を説明します。多くの高級言語では、特定の結果を得るためにいろいろなステートメントを使って記述する方法がありますが、マシンレベルではその方法によって必然的に効率の差が発生します。効率の劣るステートメントシーケンスをあえて選択する相応の理由（たとえば読みやすくするため）が存在する場合もありますが、ほとんどのソフトウェア技術者は高級言語ステートメントの実行時コストのことを何もわかっていないというのが現実です。そうした知識がなければ、ステートメントを選択しようにも根拠を持って選び出すことは不可能でしょう。それを変えるのが本書の目標です。

経験豊富なソフトウェア技術者は、そのような個別のテクニックを実践してもわずかなパフォーマンスの向上しか得られないと主張するかもしれません。場合によってはこの評価は正しいものの、そのわずかな効果が蓄積していくということを忘れてはなりません。確かに、本書で推奨するテクニックの使い方を誤り、読みにくくて保守のしにくいコードを生み出す可能性はありますが、2つの（システム設計の観点からは）同等のコードシーケンスを示された場合に、より効率の良いほうを選択するのはごく当たり前のことです。残念ながら、現在のソフトウェア技術者の多くは、2つのどちらの実装がより効率的なコードを生み出すかを知りません。

効率的なコードを書くために専門のアセンブリ言語プログラマーになる必要はありませんが、これから本書でコンパイラの出力を学習するつもりであれば、少なくともアセンブリ言語を読み解く知識は必要です。第3章と第4章では、80x86およびPowerPCのアセンブリ言語の初歩について説明します。

第5章と第6章では、コンパイラの出力を調べることで高級言語ステートメントの品質を見極める方法を学習します。これらの章では、逆アセンブラ、オブジェクトコードダンプツール、デバッガ、アセンブリ言語コードを表示するためのさまざまな高級言語のコンパイラオプション、その他の便利なソフトウェアツールについて説明します。

残りの第7章から第15章では、さまざまな高級言語ステートメントとデータ型に対してコンパイラがどのようにマシンコードを生成するかを説明します。この知識を身に付ければ、効率的なアプリケーションを作成するのに最適なデータ型、定数、変数、制御構造を選択できるようになります。

本書を読む際は、「早まった最適化は諸悪の根源だ」というHoare博士の言葉を心にとどめておいてください。確かに、本書の情報を誤って適用し、読みにくくて保守しにくいコードを生み出す恐れはあります。コードが流動的で変わりやすい、プロジェクトの設計および実装の初期段階では、これは特に被害甚大です。しかし忘れないでください。本書の主題は、結果を無視して最も効率的なステートメントシーケンスを選択することではありません。選択肢があるときに、どのシーケンスを使用すべきかを知識

に基づいて決定できるように、さまざまな高級言語構文のコストを理解することなのです。時には、効率の劣るシーケンスを選択すべき合理的な理由が存在する場合もあります。けれども、ステートメントのコストを理解していなければ、より効率的な代替手段を選択することはできません。

「諸悪の根源」に関するほかの論評にも興味がある方は、次のWebページを見てみるとよいでしょう(出版後にこれらのURLが無効になっていたらお詫びします)。

- http://blogs.msdn.com/ricom/archive/2003/12/12/43245.aspx
- http://en.wikipedia.org/wiki/Software_optimization

第1章

低いレベルで考え、高いレベルで書く

「高級言語で最良のコードを書きたければ、
アセンブリ言語を学ぶことだ」
——プログラミングに関する一般的な助言

本書では、何も革新的なことは説明しません。グレートコードを書くため、長年の実績に裏打ちされた手法を説明し、自分が書いたコードが実際のマシンでどのように実行されるのかを理解できるようにします。長年の経験を持つプログラマーであれば、おそらく本書の内容に心当たりがあるでしょう。この手法をきちんと習得していない若いプログラマーが書くコードをあまり目にする機会がなければ、本書の内容を無価値とさえ考えるかもしれません。本書（および、このシリーズの第1巻）では、今の世代のプログラマーの教育に欠けている部分を補い、良質のコードを書けるようにすることを目指しています。

『Write Great Code』シリーズのこの巻では、次の概念について説明します。

- **高レベルなプログラムであっても低レベルでの実行を考慮することが重要な理由**
- **コンパイラが高級言語のステートメントからマシンコードを生成する仕組み**
- **コンパイラが低レベルの基本的なデータ型を使ってさまざまなデータ型を表現する仕組み**
- **コンパイラがより効率的なマシンコードを生成できるように高級言語コードを書く方法**
- **コンパイラの最適化機能を活用する方法**
- **高級言語コードを書きながらアセンブリ言語（低レベルの観点）で「考える」方法**

その出発点となるこの章では、次の話題を取り上げます。

- **標準的なコンパイラが生成するコードの品質に関するプログラマーの誤解**
- **アセンブリ言語の習得が今でも有益である理由**
- **高級言語コードを書きながら低レベルの観点で考える方法**
- **本書を読む前に知っておくべき事柄**
- **本書の構成**
- **グレートコードを構成するもの**

それでは、面倒なことはこれくらいにして本題に入りましょう。

1.1 コンパイラの品質に関する誤解

パーソナルコンピュータ革命の初期の時代には、パフォーマンスの高いソフトウェアはアセンブリ言語で書かれていました。時代の経過と共に、高級言語用の最適化コンパイラが改良され、コンパイラの作成者たちは、コンパイラで生成されるコードのパフォーマンスが手作業で最適化されたアセンブリコードの10～50%以内になったと主張し始めました。こうした宣言はPCのアプリケーション開発における高級言語の台頭を招き、アセンブリ言語に終焉の鐘を鳴らすものでした。多くのプログラマーが、「このコンパイラはアセンブリ言語で書かれた場合の90%の速度をたたき出すのだから、アセンブリ言語を使うのは馬鹿げている」といった具合に、数字を引き合いに出すようになりました。問題は、彼らが手作業で最適化されたアセンブリ言語バージョンのアプリケーションをわざわざ作成し、自説を確かめようとはしなかったことです。多くの場合、コンパイラのパフォーマンスに関する彼らの憶測は誤りです。

最適化コンパイラの作成者たちが嘘をついていたわけではありません。適切な条件下では、最適化コンパイラはアセンブリ言語を使って手作業で最適化した場合とほぼ同等のパフォーマンスを持つコードを生成します。ただし、このパフォーマンスレベルを実現するには、高級言語コードを適切な方法で書く必要があります。そのためには、コンピュータの動作とソフトウェア実行の仕組みを十分に理解していなければなりません。

1.2 アセンブリ言語の習得が今でも有益である理由

アセンブリ言語を捨てて高級言語を使い始めた当初のプログラマーは、たいていは自分たちが使っている高級言語ステートメントが低レベルではどのように表現されるかを理解しており、高級言語ステートメントを適切に選択することができました。しかし残念ながら、彼らの後の世代のコンピュータプログラマーは、アセンブリ言語を習得する機会に恵まれませんでした。そのため、効率的なマシンコードへ変換できる高級言語のステートメントとデータ構造を賢明に選択することができませんでした。彼らのアプリケーションは、もしも手作業で最適化された同等のアセンブリ言語プログラムのパフォーマンスと比較したら、間違いなくコンパイラの作成者を狼狽させるでしょう。

この問題を認識したベテランのプログラマーたちは、新人のプログラマーに「優れた高級言語コードを書く方法を習得したければ、アセンブリ言語を学ぶ必要がある」という賢明な助言を与えました。アセンブリ言語を学ぶことによって、プログラマーはコードの低レベルにおける意味を考察できるようになり、高級言語でアプリケーションを記述する最良の方法に関して、十分な情報に基づく決定を下すことができます。

1.3 アセンブリ言語の習得が必須ではない理由

経験豊富なプログラマーがアセンブリ言語のプログラミングを学ぶことはおそらく有益ですが、実際には優れた効率的なコードを書く上で、アセンブリ言語の習得は必要条件ではありません。重要なのは、高級言語がステートメントをマシンコードに変換する仕組みを理解し、適切な高級言語のステートメントを選択できるようになることです。

その方法を身に付ける1つの手段はアセンブリ言語の熟練プログラマーになることですが、それにはかなりの時間と労力が必要なだけでなく、たくさんのアセンブリコードを書く必要があります。

そこで、「プログラマーはアセンブリ言語に精通しなくても、マシンの低レベルの性質を学習するだけで高級言語コードの品質を高めることができるのか」という疑問が浮かびます。答えは条件付きのイエスです。シリーズの2巻目である本書の目的は、アセンブリ言語の熟練プログラマーになることなく、グレートコードを書くために知っておくべき事柄を説明することです。

1.4 低いレベルで考える

1990年代後半にJava言語が普及し始めたころ、次のような不満の声が聞かれました。

> **Javaのバイトコードでは、ソフトウェアを書くときにかなり余計に注意を払うことを強いられる。C/C++でやるように線形検索を使って済ませることができない。2分検索のような優れた（そして実装が面倒な）アルゴリズムを使う必要がある。**

このような意見は、まさに最適化コンパイラを使用することの主な問題点を表しています。それは、プログラマーに怠け癖が付くということです。最適化コンパイラはこの数十年で飛躍的に進歩しましたが、どんな最適化コンパイラを使おうと、高級言語で書かれた出来の悪いソースコードの欠点を補うことはできません。

もちろん、多くの無邪気な高級言語プログラマーは、最新のコンパイラの最適化アルゴリズムがいかに優れているかを説明した文章を読み、コンパイラに何を与えても効率的なコードが生成されると思い込んでいます。しかし、この考え方には1つ問題があります。きちんと書かれた高級言語コードを効率的なマシンコードに変換する分にはコンパイラは見事な働きをしますが、出来の悪いソースコードをコンパイラに与えると、最適化アルゴリズムがすぐに行き詰まってしまうのです。現実に、自分のプログラムの書き方のせいでコンパイラがお粗末な働きをしていることにまるで気付かず、コンパイラのすばらしさを吹聴しているC/C++プログラマーを見かけることも珍しくありません。問題は、コンパイラが高級言語ソースコードから生成したマシンコードを、彼らが実際には一度も見ていないことです。彼らは、「コンパイラが生成するコードは、アセンブリ言語の熟練プログラマーが作成するコードとほとんど変わらない」と聞かされているために、コンパイラは優れた働きをしているとやみくもに信じています。

1.4.1 コンパイラの働きはソースコード次第

言うまでもなく、コンパイラがソフトウェアのパフォーマンスを改善するためにアルゴリズムを変更することはありません。たとえば、2分検索ではなく線形検索を使用した場合に、コンパイラがそれをより効率的なアルゴリズムに置き換えてくれることはありません。確かに、オプティマイザによって線形検索の速度が定数倍だけ（たとえば、コードの速度が2倍または3倍に）向上することもありますが、より効率的なアルゴリズムを使った場合の速度の向上とは比較になりません。実際、十分な大きさのデータベースがある場合に、最適化せずにインタプリタで処理される2分検索のほうが、最良のコンパイラで処理される線形検索アルゴリズムよりも高速に実行されることを証明するのはとても簡単です。

1.4.2 コンパイラがより効率的なマシンコードを生成できるようにする

たとえば、アプリケーションに最適なアルゴリズムを選択し、余分なお金をかけて最高のコンパイラを手に入れたとしましょう。この上さらに効率的な高級言語コードを書くために、何かできることはあるでしょうか。一般に、その答えはイエスです。

コンパイラの世界では公然の秘密ですが、ほとんどのコンパイラベンチマークは操作されています。大半のコンパイラベンチマークでは、使用するアルゴリズムが指定されていますが、個々の言語でのアルゴリズムの実装はコンパイラベンダーに任されています。コンパイラベンダーは、普通は特定のコードシーケンスを与えたときに自社のコンパイラがどのように動作するかを承知しているので、可能な限り最も効率的な実行コードを生成するコードシーケンスを作成します。

これをインチキだと思う人がいるかもしれませんが、実際はそうではありません。通常の状況下でコンパイラが同じコードシーケンスを生成できるのであれば（つまり、コード生成の仕掛けをベンチマーク専用に作ったのでない限り）、コンパイラのパフォーマンスを誇示することに何の問題もありません。それに、コンパイラベンダーがこのようなちょっとした仕掛けを使えるとすれば、あなたにも同じことができるわけです。高級言語ソースコードで使用するステートメントを慎重に選択することで、コンパイラが生成するマシンコードを「手動で最適化」できます。

手動による最適化には、いくつかのレベルがあります。最も抽象的なレベルでは、ソフトウェアにより適したアルゴリズムを選択してプログラムを最適化します。このテクニックは、コンパイラや言語に依存しません。

1段下のレベルでは、使用中の高級言語が何であるかを踏まえて、コードを手動で最適化します。ただし、使用している高級言語のコンパイラ製品や実装には依存しない最適化にとどめておきます。このような最適化はほかの言語には適用できませんが、同じ言語の別々のコンパイラ間では適用できます。

さらに1段下のレベルでは、特定のベンダーや、場合によってはベンダーの特定のバージョンのコンパイラにしか適用できないような最適化を行ったコードの構成を考え始めることができます。

おそらく最も下のレベルでは、コンパイラが出力するマシンコードを考慮し、何らかの望ましいマシン命令のシーケンスが生成されるように、高級言語のステートメントの書き方を調整し始めることになります。Linuxカーネルは、この手法の一例です。言い伝えによると、カーネルの開発者たちは、GCCコンパイラが生成する80x86マシンコードを制御するために、Linuxカーネルとして書かれたCコードを絶えず微調整していたそうです。

この開発プロセスは少し誇張されているかもしれませんが、確かなことが1つあります。このプロセスを用いるプログラマーが、最も効率的なマシンコードを生成するということです。まともなアセンブリ言語プログラマーが作成するコードに匹敵するのは、このタイプのコードであり、コンパイラは手書きのアセンブリ言語に匹敵するコードを生成すると高級言語プログラマーが主張するときに自慢したいのは、この種のコンパイラ出力なのです。多くの人々は高級言語コードを書くときにそこまで極端に走らないという事実は、ここでは一切考慮していません。それでも、慎重に書かれた高級言語コードが、まともなアセンブリコードと同じくらい効率的になり得るという事実は残ります。

コンパイラは、アセンブリ言語の熟練プログラマーが書くのと同等の品質のコードを生成できるようになるでしょうか。正確な答えはノーです。しかし、Cなどの高級言語でコードを書く注意深いプログラマーが、コンパイラがプログラムを効率的なマシンコードへ容易に変換できるような形で高級言語コードを書けば、品質をそれに近付けることができます。そこで、「コンパイラが最も効率的に変換できるように高級言語コードを書くにはどうすればよいのか」という疑問が浮かびます。実は、その疑問に答えることこそが本書の主題なのです。しかし、手短に言ってしまうと、「アセンブリ言語で考え、高級言語で書く」ということになります。その方法を簡単に見てみることにしましょう。

1.4.3 アセンブリ言語で考えながら高級言語コードを書く方法

高級言語コンパイラは、その言語のステートメントを1つまたは複数のマシン語（またはアセンブリ言語）命令のシーケンスに変換します。アプリケーションが消費するメモリ空間の量、およびアプリケーションの実行にかかる時間は、マシン命令の数と、コンパイラが出力するマシン命令の種類に直接関係しています。

ただし、高級言語の2つの異なるコードシーケンスを使って同じ結果が得られるからといって、どちらの場合もコンパイラが同じマシン命令のシーケンスを生成しているわけではありません。高級言語のifステートメントとswitch/caseステートメントは、その代表例です。多くのプログラミングの入門書では、一連のif-elseif-elseステートメントとswitch/caseステートメントを同じものとして扱っています。次の簡単なCの例を見てください。

```
switch( x )
    {
        case 1:
            printf( "X=1\n" );
            break;

        case 2:
            printf( "X=2\n" );
            break;

        case 3:
            printf( "X=3\n" );
            break;

        case 4:
            printf( "X=4\n" );
            break;

        default:
            printf( "X does not equal 1, 2, 3, or 4\n" );
    }

/* 同等のIFステートメント */

    if( x == 1 )
            printf( "X=1\n" );
    else if( x == 2 )
            printf( "X=2\n" );
    else if( x == 3 )
            printf( "X=3\n" );
    else if( x == 4 )
            printf( "X=4\n" );
    else
            printf( "X does not equal 1, 2, 3, or 4\n" );
```

この2つのコードシーケンスは意味的には等しい（つまり、同じ実行結果が得られる）ものの、コンパイラが両方に対して同じマシン命令のシーケンスを生成するという保証はまったくありません。

より効率的なのは、どちらのコードシーケンスでしょうか。コンパイラがこのようなステートメントをマシンコードに変換する仕組みを理解し、さまざまなマシン命令ごとの効率の違いについての基礎知識を持たない限り、シーケンス同士を評価して一方を選択することはできません。コンパイラがこの2つのシーケンスをどのように変換するかを十分に理解しているプログラマーであれば、コンパイラによって生成されるコードの品質に基づいて、いずれか一方のシーケンスを賢明に選択できます。

高級言語コードを書くときに低レベルの観点で考えることにより、プログラマーは最適化コンパイラが生成するコードの品質レベルを、手作業で最適化したアセンブリ言語コードのレベルに近付けることができます。残念ながら、たいていはその逆も当てはまります。高級言語コードが低レベルではどのように表現されるかをプログラマーが考慮しない場合は、コンパイラが品質の高いマシンコードを生成することはめったにありません。

1.5 高いレベルで書く

高レベルのコードを書くときに低レベルの観点で考えることの1つの問題は、この方法で高級言語コードを書くのはアセンブリコードを書くのと同じくらい大変だということです。そのため、開発時間の短縮、読みやすさ、保守のしやすさなど、高級言語でプログラムを書く場合におなじみの利点の多くが帳消しになります。高級言語でアプリケーションを書くことの利点を犠牲にするくらいなら、初めからアセンブリ言語で書けばよいのではないでしょうか。

実際には、低レベルの観点で考えても、全体のプロジェクトスケジュールは思ったほど長くなりません。確かに最初のコーディングには時間がかかりますが、書き上がった高級言語コードはやはり読みやすくて移植性があり、きちんと書かれたグレートコードのほかの属性をそのまま維持しています。しかし、もっと重要なのは、低レベルの観点で考えなければ得られなかった効率も得られることです。いったんコードを書いてしまえば、ソフトウェアライフサイクルの保守フェーズと拡張フェーズを通して、コードについて常に低レベルの観点で考える必要はありません。つまり、最初のソフトウェア開発段階の間に低レベルの観点で考えれば、付随するデメリットなしに、低レベルと高レベルの両方のコーディングのメリット（効率と保守のしやすさ）が維持されるということです。

1.6 前提

本書は、読者に一定の予備知識があることを前提に書かれています。次の前提を満たすスキルを持っている人であれば、本書の内容を最大限に活かすことができるでしょう。

- 少なくとも1つの命令型（手続き型）プログラミング言語をある程度使いこなせること。具体的にはCとC++、Pascal、BASIC、アセンブリ言語などで、Ada、Modula-2、FORTRANなどの言語でも構いません。
- ちょっとした問題を設定し、その問題についてソフトウェアの設計から実装まで一連の作業を遂行できること。大学の一期または半期の標準的な課程（あるいは数ヶ月の経験）で得られる程度の知識があれば十分です。
- ハードウェア構成およびデータ表現の基礎知識があること。16進数および2進数に関する知識があること。符号付き整数、文字、文字列などのさまざまな高レベルのデータ型がコンピュータのメモリ内でどのように表現されるかを理解していること。この後のいくつかの章でマシン語の初歩について説明しますが、事前にその知識があれば大いに役立つでしょう。ハードウェア構成の知識に多少不安がある読者には、『Write Great Code, Volume 1:Understanding the Machine』<監訳者注1>で詳しく説明しています。

1.7 言語に依存しない手法

本書は、読者が少なくとも1つの命令型言語に精通していることを前提に書かれていますが、特定の言語に依存しているわけではありません。本書の考え方はプログラミング言語の垣根を越えて通用します。できるだけ多くの読者の役に立つように、本書のプログラム例では複数の言語（C/C++、Pascal、BASIC、アセンブリ言語など）を交代で使用しています。例を示すときはコードの動作を具体的に説明しているので、そのプログラミング言語に不慣れでも、そこに書かれている説明を読めば動作を理解できます。

本書の例で使用しているプログラミング言語とコンパイラは次のとおりです。

- C/C++：GCC、Microsoft Visual C++、Borland C++
- Pascal：Borland Delphi/Kylix
- アセンブリ言語：Microsoft MASM、Borland TASM、HLA（High-Level Assembler）、GNUアセンブラ（Gas）
- BASIC：Microsoft Visual Basic

アセンブリ言語の操作に不慣れでも心配はいりません。アセンブリ言語に関する2つの章の入門編とオンラインリファレンス（http://www.writegreatcode.com/）を参考にすれば、コンパイラの出力を解読できるようになります。アセンブリ言語の知識をさらに広げたい読者には、拙著『The Art of Assembly Language』（No Starch Press、2003年）をお勧めします。

<監訳者注1>邦訳『Write Great Code―ハードウェアを知り、ソフトウェアを書く』（鵜飼文敏、後藤正徳、まつもとゆきひろ監訳、トップスタジオ訳、毎日コミュニケーションズ、2006年、ISBN4-8399-1820-1）。

1.8 グレートコードを特徴付けるもの

「**グレートコード**」とはどのような意味合いを持つのでしょうか。このシリーズの第1巻で、優れたコードのいくつかの特性を示しました。本書の目標を定めるために、ここでその説明を繰り返すことにします。

グレートコードの定義はプログラマーによってさまざまに変化するので、すべてのプログラマーを満足させるような包括的な定義を与えることは不可能です。しかしながら、グレートコードが備えるべき特性として、ほぼすべてのプログラマーが認めるものがあります。それらの共通の特性に基づいて、本書における定義をまとめます。本書の目的にかなう特性は、次のようなものです。

- **CPUを効率的に使用する（つまり、コードが高速である）**
- **メモリを効率的に使用する（つまり、コードが小さい）**
- **システムリソースを効率的に使用する**
- **可読性に優れ、メンテナンスが簡単である**
- **一貫したスタイル指針に従う**
- **ソフトウェア工学的に確立された規約に基づいて系統的に設計されている**
- **拡張性に富む**
- **十分テストされ、堅牢である（つまり、確実に機能する）**
- **ドキュメントが整備されている**

これ以外にも、必要とあれば、いくらでも付け足すことができます。たとえば、移植性が高いこと、取り決めたプログラミングスタイル指針に従うこと（あるいは特定の言語で記述**されていない**こと）などをグレートコードの特性として求めるプログラマーもいるでしょう。また、グレートコードはできるだけわかりやすく記述されるものだと言う人もいれば、短時間で記述されるものだと言う人もいるかもしれません。あるいは、期限と予算を守って作られるものと考える人だっているかもしれません。きっと、あなたも別の特性を考えているのではないでしょうか。

では、グレートコードとは一体何でしょうか？そこそこ妥当な定義を示します。

> **グレートコードとは、一連の適切なソフトウェア特性（それらは一貫性があり、重要なものから順に並べられている）に基づいて記述されたソフトウェアのこと。特に、グレートコードは、アルゴリズムをソースコードの形で実装する際にプログラマーが決定を下すために使ういくつかの規則に従う。**

本書では、グレートコードを書く際の効率の側面に重点を置いています。効率は必ずしもソフトウェア開発作業の主要な目標ではない場合もありますが、効率の悪いコードはグレートコードではないという点には、ほとんどの人がおおむね同意するでしょう。これは、効率を最大限に引き出していないコードはグレートコードではないという意味ではありません。しかし、著しく効率の悪い（つまり、目に見えて非効率な）コードは決してグレートコードとは見なされません。そして、効率の悪さは現在のアプリケーションの大きな問題の1つなので、強調すべき重要な話題です。

1.9 本書で使用する環境

本書では包括的な情報を扱いますが、説明の内容によっては特定のシステムを対象にせざるを得ない部分があります。現在圧倒的に普及しているのはIntelアーキテクチャのPCなので、本書で特定のシステムに依存する概念を説明するときは、Intelプラットフォームを使うことにします。しかし、それらの概念はほかのシステムやCPUにも（たとえば、以前のPower MacintoshシステムのPowerPC CPUやUnixボックスのRISC CPUなどにも）当てはまります。もっとも、ある概念が現在使用中のプラットフォームに適合しているかどうかは、よく調べる必要があるかもしれません。

本書の大部分の例はWindowsとLinuxのどちらでも動作します。例を作成するときに、オペレーティングシステムとのインターフェイスについては、できる限り標準ライブラリを使うように努めました。グレートコードを書くためにやむを得ない場合だけ、オペレーティングシステム固有のシステムコールを使用しています。

本書に掲載されている具体的な例のほとんどは、最新モデルのIntelアーキテクチャCPU（AMDを含む）で動作します。対応するオペレーティングシステムはWindowsとLinuxです。また、最新モデルのPCにおいて標準的で適度なサイズのRAMと周辺装置を装備しているものとします[注1]。ソフトウェア自体は別として、その概念はMacintosh、Unixボックス、組み込みシステム、さらにはメインフレームにも当てはまります。

1.10 さらなる知識のために

グレートコードを書くために知っておくべき事柄を1冊の本ですべて網羅することはできません。そこで、本書では優れたソフトウェアの作成に特に関係する分野に重点を置き、最も効率的なコードを書くことに関心がある人々に90%の解決策を提供します。残りの10%を手に入れるには、ほかの情報源を当たる必要があります。次にいくつかの参照先の候補を示します。

- **アセンブリ言語の熟練プログラマーになる。少なくとも1つのアセンブリ言語をすらすら書けるようになると、本書では得られない多くの詳細な部分が埋まります。本書の目的は、実際にアセンブリ言語プログラマーにならずに、最も効率的なコードを書く方法を示すことです。しかし、新たな努力を重ねることにより、低レベルの観点で考える力が向上します。アセンブリ言語の学習には、拙著『The Art of Assembly Language』（No Starch Press、2003年）をお勧めします。**
- **コンパイラの構成論を学習する。これはコンピュータサイエンスの上級トピックですが、コンパイラがコードを生成する仕組みを理解するには、コンパイラの背景にある理論を学習するのが何よりの方法です。この理論を扱う参考書は多岐にわたりますが、前提条件となる資料がかなりの量にのぼります。本を購入する前に内容を詳しく調べて、自分の技量に適したレベルで書**

<注1>PowerPCアセンブリ言語の例など、Intelマシンでは動作しない例もいくつかありますが、ごくわずかです。

かれているかどうかを確認してください。検索エンジンを使い、優れた手引書をインターネットで見つけることもできます。

- 先進的なコンピュータアーキテクチャを学習する。ハードウェア構成とアセンブリ言語プログラミングは、コンピュータアーキテクチャの研究対象の一部です。独自のCPUを設計する方法を知る必要はないにしても、コンピュータアーキテクチャを学習することで、高級言語コードをよりうまく書くための新たな方法が見つかるかもしれません。『Computer Architecture: A Quantitative Approach』（Patterson、Hennessy、Goldberg共著、Morgan Kaufmann、2002年）<監訳者注2>は、この話題を扱っている好著です。

<監訳者注2>邦訳『コンピュータ・アーキテクチャ――設計・実現・評価の定量的アプローチ』（富田真治、新実治男、村上和彰訳、日経BP社、1994年、ISBN4-8222-7152-8）。なお、英語版は、マルチプロセッサなどの最新のトピックを増補した第4版が刊行されています。

第2章

アセンブリ言語習得の勧め

本書では、アセンブリ言語に習熟しなくても、より効率的なコードを書く方法を説明しています。ただし、本当に優秀な高級言語プログラマーはアセンブリ言語を知っており、その知識こそ彼らがグレートコードを書ける根拠の1つになっています。単に高級言語でグレートコードを書きたいと望んでいる読者は、本書で90%の解決策を得ることができますが、残りの10%を埋めるにはアセンブリ言語を習得する以外にありません。アセンブリ言語に習熟する方法を説明することは本書の範疇を超えていますが、この話題について考察し、読者が本書を読んだ後に100%の解決策を追求する場合の情報源を示すことは、やはり重要です。この章では、次の概念について説明します。

- **アセンブリ言語の習得にかかわる問題**
- **高級アセンブラと、それらを利用したアセンブリ言語の習得**
- **Microsoft Macro Assembler（MASM）、Borland Turbo Assembler（TASM）、HLAなどの実際の製品を利用したアセンブリ言語プログラミングの習得**
- **アセンブリ言語プログラマーの考え方（アセンブリ言語プログラミングパラダイム）**
- **アセンブリ言語プログラミングの習得に役立つ情報源**

2.1 アセンブリ言語の習得を妨げるもの

アセンブリ言語を習得し、**本当に**自在に操れるようになることには2つの利点があります。第1に、コンパイラが生成するマシンコードを**完全に**理解できることです。アセンブリ言語に習熟することで、前述の「100%の解決策」を実現し、より効率的な高級言語コードを記述できるようになります。第2に、たとえ手動で最適化しても高級言語コンパイラでは効率の良いコードを生成できない場合に、アセンブリ言語のレベルに降りて、アプリケーションの重要な部分をアセンブリ言語でコーディングできることです。したがって、これ以降の章の内容を身に付けて高級言語のスキルを磨いた後で、アセンブリ言語の習得に着手することは非常に有益です。

アセンブリ言語の習得には1つだけ落とし穴があります。従来、アセンブリ言語の習得は、時間のかかる、面倒な、いらだたしい作業でした。アセンブリ言語プログラミングパラダイムは高級言語プログラミングとは大きくかけ離れているため、ほとんどの人はアセンブリ言語を学習するときに一からやり直すような気分になります。C/C++、Java、Pascal、Visual Basicなどのプログラミング言語で実現する方法はわかっているのに、それをアセンブリ言語で解決する方法がわからないというのは非常にいらだたしいものです。

ほとんどのプログラマーは、何か新しいことを学習するときに、過去に学んだことをうまく応用したいと考えます。しかし残念ながら、アセンブリ言語プログラミングを習得するための従来の手法では、高級言語プログラマーは過去に学んだことを忘れるように強いられがちです。これはどう見ても、既存の知識の効率的な活用とは言えません。アセンブリ言語を習得するときに、既存の知識を活用できるようにする方法が求められていました。

2.2 本書を役立てる

本書を読み終えると、次の3つの理由により、アセンブリ言語をはるかに習得しやすくなります。

- **アセンブリ言語に習熟することが、より効率的なコードを書くために役立つ理由がわかるので、アセンブリ言語に対する学習意欲が向上する**
- **本書にはアセンブリ言語に関する2つの入門編（80x86アセンブリ言語およびPowerPCアセンブリ言語）が含まれるので、たとえアセンブリ言語を初めて目にする読者でも、本書を読み終わるころにはある程度のアセンブリ言語の知識が身に付いている**
- **一般的なすべての制御構造とデータ構造について、コンパイラがどのようにマシンコードを出力するのかがわかるので、初心者のアセンブリプログラマーが直面する最も困難な課題の1つ（高級言語でのやり方がわかっていることをアセンブリ言語でどのように実現するか）を習得できる**

本書では、アセンブリ言語の熟練プログラマーになる方法こそ説明しませんが、コンパイラが高級言語をどのようにマシンコードへ変換するかを示すたくさんのサンプルプログラムを通じて、アセンブリ言語のさまざまなプログラミングテクニックを紹介します。本書を読み終えてアセンブリ言語を学ぼうと決心したときには、これらのサンプルプログラムが役立つはずです。

確かに、すでにアセンブリ言語の知識があれば、本書を読みやすくなるでしょう。しかし、本書を読み終えるとアセンブリ言語を習得しやすくなるという点も重要です。この2つ（アセンブリ言語を習得することと本書を読むこと）を比較した場合、おそらくアセンブリ言語を習得するほうが時間がかかるので、本書を先に読むほうが効率的なやり方と言えます。

2.3 高級アセンブラを役立てる

話は1995年にさかのぼりますが、あるとき、筆者はカリフォルニア大学リバーサイド校のコンピュータサイエンス学部長と議論をしていました。筆者は、学生がアセンブリ言語の講義をとるときに、最初からすべてをやり直す必要があり、非常に多くの事柄を学び直すためにかなりの時間がかかることを残念に思っていました。議論が進むにつれて、問題はアセンブリ言語自体にあるのではなく、既存のアセンブラ（Microsoft Macro Assembler：MASMなど）の構文にあることがはっきりしてきました。アセンブリ言語の習得は、いくつかのマシン命令を覚えれば済むような生やさしいものではありませんでした。第1に、新しいプログラミングスタイルを覚える必要があります。アセンブリ言語に習熟するには、いくつかのマシン命令のセマンティクスを覚えるだけでは足りず、それらの命令を組み合わせて現実の問題を解決する方法も学ぶ必要があります。そして、**これこそが**アセンブリ言語の習得で苦労する部分なのです。

第2に、**純粋な**アセンブリ言語は、いくつかの命令を一度に効率的に覚えられるようにはできていません。最も単純なプログラムを書くだけでも、かなりの量の知識と、数十個以上のマシン命令のレパートリーが要求されます。学生が通常のアセンブリ言語の講義で覚えなければならないほかのすべてのハード

ウェア構成のトピックに、そのレパートリーが加わると、「手取り足取りして教えた」単純なアプリケーション以外のものをアセンブリ言語で書けるようになるまでに数週間かかることも少なくありません。

1995年当時のMASMが備えていた1つの重要な機能は、.if、.whileなどの高級言語に似た制御ステートメントのサポートでした。これらのステートメントは真のマシン命令ではないものの、学生は時間をかけて十分なマシン命令を覚え、低レベルのマシン命令を使ってアプリケーションを書けるようになるまで、講義の初期には慣れ親しんだプログラミング構文を使うことができます。学期の早い時期にこれらの高レベルの構文を使うことによって、学生はアセンブリ言語プログラミングのほかの側面に専念できるようになり、すべてを一度に身に付ける必要がなくなります。これによって、学生は講義の早い段階でコードを書き始めることができ、その結果、より広範な内容を学期末までに修得できます。

高級言語に似た制御ステートメントを（同じ働きをする従来の低レベルのマシン命令のほかに）備えたアセンブラを**高級アセンブラ**と呼びます。MicrosoftのMASM（バージョン6.0以降）やBorlandのTASM（バージョン5.0以降）は、高級アセンブラの好例です。理屈の上では、これらの高級アセンブラを使ってアセンブリ言語プログラミングを学ぶ適切な教科書があれば、学生は講義の1週目から簡単なプログラムを書き始めることができます。

MASMやTASMなどの高級アセンブラの唯一の問題は、高級言語に似た制御ステートメントとデータ型をわずかしか備えていないことです。高級言語プログラミングに慣れ親しんだ人にとって、それ以外はほとんどが異質のものです。たとえば、MASMとTASMのデータ宣言は、多くの高級言語のデータ宣言とはまったく異なります。高級言語に似た制御ステートメントがあるにもかかわらず、初心者のアセンブリプログラマーは、やはりかなりの量の知識を学び直さなければなりません。

2.4 High-Level Assembler（HLA）

学部長と議論をした直後、筆者はアセンブリ言語のセマンティクスを変えずに、より高レベルの構文をアセンブラに取り入れればよいということにふと気付きました。たとえば、整数型の配列変数を宣言する次のようなC/C++とPascalのステートメントがあるとします。

```
int intVar[8]; // C/C++

var intVar: array[0..7] of integer; (* Pascal *)
```

同じものをMASMで宣言すると次のようになります<監訳者注1>。

```
intVar sdword 8 dup (?) ;MASM
```

<監訳者注1>これはintVarがグローバル変数の場合のみです。自動変数としてスタックに確保する場合は、このようには記述しません。

C/C++とPascalの宣言は互いに異なるものの、アセンブリ言語の宣言はそのどちらとも根本的に異なっています。C/C++のプログラマーは、たとえPascalのコードを一度も見たことがなくても、おそらくPascalの宣言を理解できるでしょう。これは逆の場合についても当てはまります。しかし、PascalとC/C++のプログラマーは、おそらくアセンブリ言語の宣言を解読できないでしょう。これは、高級言語プログラマーが初めてアセンブリ言語を学ぶときに直面する問題のほんの一例です。

残念なのは、アセンブリ言語の変数の宣言が高級言語の宣言とここまで根本的に違っていなければならない理由が何一つないことです。アセンブラが変数の宣言にどの構文を使用しても、最終的な実行可能ファイルには何も違いはありません。それなら、アセンブラでもっと高級言語に近い構文を使用して、高級言語から移行する人たちがアセンブラを学びやすいようにすればよいのではないでしょうか。1996年にアセンブリ言語の講義について学部長と議論したときに考えていたのが、この疑問でした。それをきっかけに、筆者は高級言語をすでに習得した学生がアセンブリ言語プログラミングを学ぶための新しいアセンブラである**HLA**（High-Level Assembler）を開発しました。HLAでは、たとえば先ほどの配列の宣言は次のようになります。

```
var intVar:int32[8]; // HLA
```

構文はC/C++やPascalと少し異なりますが（実際は両者の組み合わせになっています）、ほとんどの高級言語プログラマーがこの宣言の意味を理解できるでしょう。

HLAの設計目標は、**本物の**アセンブリ言語プログラムを書く能力を犠牲にせずに、従来の（命令型の）高級プログラミング言語にできるだけ近いアセンブリ言語プログラミング環境を作ることです。マシン命令と無関係なコンポーネントでは慣れ親しんだ高級言語の構文を使用する一方で、マシン命令は依然として基になる80x86マシン命令に一対一ベースで対応付けられます。

HLAをさまざまな高級言語にできるだけ似せることによって、アセンブリ言語プログラミングを学ぶ学生は、根本的に異なる構文の習得にそれほど時間をかけずに済むようになります。代わりに、既存の高級言語の知識を応用できるので、アセンブリ言語をより簡単に短期間で習得できます。

アセンブリ言語の学習効率をできるだけ高めるために必要なのは、親しみやすい宣言用の構文と、高級言語に似たいくつかの制御ステートメントだけではありません。アセンブリ言語の習得に関して非常によく聞かれる不満の1つは、プログラマーに対するサポートがほとんど用意されていないという点です。プログラマーはアセンブリコードを書くときに、いつもわざわざ一から作り直す必要があります。たとえば、MASMまたはTASMを使ってアセンブリ言語プログラミングを学ぶときにすぐに気付くのは、アセンブリ言語には整数値をユーザーのコンソールに文字列として出力できるような便利なI/O機能がないことです。そうしたコードはアセンブリプログラマーが自分で書かなければなりません。困ったことに、まともなI/Oルーチンを書くには、アセンブリ言語プログラミングの高度な知識が必要です。にもかかわらず、その知識を身に付けるにはかなりの量のコードを書く以外に方法がなく、I/Oルーチンなしにそうしたコードを書くのは困難です。したがって、アセンブリ言語の教育用ツールは、初心者のアセンブリプログラマーが自分でI/Oルーチンを書けるようになるまで、簡単なI/O処理（整数値の読み書きなど）が行える一連のI/Oルーチンを備えている必要があります。HLAには、この機能が**HLA標準ライブラリ**を装った形で用意されています。これは、呼び出すだけで複雑なアプリケーションを非常に簡単に書けるようになるサブルーチンとマクロの集合です。

HLAアセンブラの利用者は増え続けており、HLAはWindowsとLinuxに対応したフリー、オープンソース、パブリックドメインの製品なので、本書ではアセンブリ言語にかかわるコンパイラに依存しない例にはHLA構文を使用します。

2.5 高いレベルで考え、低いレベルで書く

HLAの目的は、初心者のアセンブリプログラマーが高級言語の観点で考えながら低レベルのコードを書けるようにすることです（言い換えると、本書で目指していることの正反対です）。もちろん、最終的にはアセンブリプログラマーは低レベルの観点で考える必要があります。しかし、アセンブリ言語に初めて接する学生にとって、高レベルの観点で考えられることは天の恵みです。アセンブリ言語プログラミング特有の問題に直面したときに、ほかの言語で学んだテクニックを応用できるからです。

最終的に、アセンブリ言語の学生は高レベルの制御構造を手放し、低レベルの構造を使う必要があります。しかし初期の段階では、こうした高レベルのステートメントを利用することで、学生は低レベルのほかのプログラミング概念に専念し、それらを身に付けることができます。学生が新しい概念を身に付ける速度を調整することによって、教育過程の効率を高めることができます。

最終的な目標は、当然ながら低レベルのプログラミングパラダイムを習得することです。そしてそれは、高級言語に似た制御構造を捨てて、純粋な低レベルのコードを書くことを意味します。つまり、「低いレベルで考え、低いレベルで書く」ということです。それでも、「高いレベルで考え、低いレベルで書く」ところから始めることは、アセンブリ言語プログラミングを学ぶ上で有効な方法です。これは、ニコチン含有量をさまざまに調整したパッチを使用する禁煙療法によく似ています。パッチ使用者は、ニコチンへの欲求を徐々に断ち切ります。同様に、高級アセンブラを使うと、プログラマーは高レベルの観点で考える習慣を徐々に断ち切ることができます。この方法は、禁煙しようとするときと同じくらい、アセンブリ言語の習得に効果があります。

2.6 アセンブリ言語プログラミングパラダイム（低いレベルで考える）

アセンブリ言語のプログラミングは、一般的な高級言語のプログラミングとまったく異なります。このため、多くのプログラマーはアセンブリ言語でプログラムを書く方法を習得するのは難しいと感じています。幸いなことに、本書で必要なのは、コンパイラの出力を分析するためにアセンブリ言語を読み解く知識だけです。アセンブリ言語プログラムを一から書けるようになる必要はありません。つまり、アセンブリ言語プログラミングの難しい部分を克服する必要はありません。それでも、アセンブリプログラムの書き方がわかると、コンパイラが特定のコードシーケンスを出力する理由を理解できるようになります。そこで、これから時間をかけて、アセンブリ言語のプログラマー（およびコンパイラ）がどのように「考える」のかを説明していきます。

アセンブリ言語プログラミングパラダイム<注1>の最も基本的な側面は、実現しようとするタスクが、マシンで処理できるきわめて小さな部分に分割されることです。本質的に、CPUはきわめて小さいタスクを一度に1つしか処理できません（これはCISCプロセッサでも同じです）。このため、高級言語で使われるステートメントのような複雑な操作は、マシンが直接実行できる小さなコンポーネントに分割する必要があります。例として、次のVisual Basicの代入ステートメントを見てください。

```
profits = sales - costOfGoods - overhead - commissions<監訳者注2>
```

このVisual Basicステートメント全体を1つのマシン命令として実行できるCPUは、現実にはありません。そこで、ステートメントを分割し、個別のコンポーネントを計算する一連のマシン命令にする必要があります。たとえば、多くのCPUにはマシンのレジスタから1つの値を減算するための**減算**命令があります。この例の代入ステートメントは3つの減算で構成されているので、代入操作を少なくとも3つの減算命令に分割する必要があります。

80x86 CPUファミリには、かなり柔軟性のある減算命令のsubが用意されています。この命令は次のような形式で使用できます（HLA構文）。

```
sub( constant, reg );        // reg = reg - constant<監訳者注3>
sub( constant, memory );     // memory = memory - constant
sub( reg1, reg2 );           // reg2 = reg2 - reg1
sub( memory, reg );          // reg = reg - memory
sub( reg, memory );          // memory = memory - reg
```

元のVisual Basicコードのすべての識別子が変数を表していると仮定した場合、80x86 sub命令とmov命令を使い、次のHLAコードシーケンスで同じ操作を実装できます。

```
// salesの値をEAXレジスタに取り込む

mov( sales, eax );

// sales - costOfGoodsを計算する（EAX := EAX - costOfGoods）

sub( costOfGoods, eax );

// (sales-costOfGoods) - overheadを計算する
//  （注：EAXにはsales - costOfGoodsが格納されている）

sub( overhead, eax );

// (sales - costOfGoods-overhead) - commissionsを計算する
//  （注：EAXにはsales - costOfGoods - overheadが格納されている）
```

<注1>**プログラミングパラダイム**とは、プログラミングをどのように行うかをモデル化したものです。したがって、アセンブリ言語プログラミングパラダイムとは、アセンブリプログラミングをどのように行うかを記述したものです。
<監訳者注2>利益 = 売上 ― 製造原価 ― 諸経費 ― 手数料
<監訳者注3>constantは定数、regはレジスタ、memoryはメモリを意味しています。

```
sub( commissions, eax );

// EAX内に格納されている結果をprofitsに格納する

mov( eax, profits );
```

ここで重要なのは、1つのVisual Basicステートメントが5つのHLAステートメントに分割され、それぞれが計算全体の小さな部分を実行していることです。アセンブリ言語プログラミングパラダイムを理解する秘訣は、この例で示したように、複雑な操作を単純な一連のマシン命令に分割する方法を知ることにあります。この手順は第13章でもう一度取り上げます。

複雑な操作を単純な一連のステートメントに分割する場合のもう1つの要点が、高級言語の制御構造です。たとえば、次のPascalのifステートメントを見てください。

```
if( i = j ) then begin

    writeln( "i is equal to j" );

end;
```

CPUはifマシン命令をサポートしていません。そこで、代わりに2つの値を比較して**条件コードフラグ**をセットした後、**条件ジャンプ**命令を使ってこれらの条件コードの結果を評価します。高級言語のifステートメントをアセンブリ言語に変換する一般的な方法は、逆の条件（i <> j）を評価し、元の条件（i = j）の評価が真だった場合に実行されるステートメントを飛び越すことです。たとえば、次に示すのは、先ほどのPascalのifステートメントを（**純粋な**アセンブリ言語を使って、つまり高級言語に似た構文を使わずに）HLAに変換したものです。

```
    mov( i, eax );      // iの値を取り込む
    cmp( eax, j );      // jの値と比較する
    jne skipIfBody;     // i <> jの場合はifステートメントのボディ
                        // （文字列を出力するコード部分）をスキップする

    << 文字列を出力するコード >>

skipIfBody:
```

高級言語言語の制御構造のブール式が複雑になるにつれて、対応するマシン命令の数も増えていきます。しかし、手順は変わりません。後半の章で、コンパイラが高レベルの制御構造をアセンブリ言語に変換する仕組みについて説明します（第14章と第15章を参照してください）。

プロシージャや関数に引数を渡す操作、プロシージャや関数内のそれらの引数にアクセスする操作、そのプロシージャや関数にローカルなその他のデータへアクセスする操作でも、アセンブリ言語は通常の高級言語に比べてかなり複雑になります。ここではその方法を詳しく説明する（または簡単な例の意味を理解する）ための前提条件が整っていませんが、本書の後半でこの重要な問題を取り上げるので安心

してください（第16章を参照してください）。

要するに、何らかのアルゴリズムを高級言語から変換するときは、アセンブリ言語でコーディングできるように、問題をきわめて小さな部分に分割する必要があるということです。前に述べたように、単にアセンブリコードを読み解くだけであれば、どのマシン命令を使えばよいかを判断する必要はありません。それは元のコードを作成したコンパイラ（またはアセンブリプログラマー）がすでにやってくれているからです。必要なのは、高級言語コードとアセンブリコードを対応付けることだけです。それをどのように実現するかが、これ以降の本書の大部分の主題となります。

2.7 『The Art of Assembly Language』およびその他の情報源

HLAはアセンブリ言語を学ぶための優れたツールですが、HLAだけでは不十分です。HLAを使ってアセンブリ言語を学ぶには、HLAを使用する良質の教材がぜひとも必要です。幸いにも、そのような教材は存在します。実のところ、HLAはそれらの教材を補助するために特に作成されました（HLAを補助するために教材が作成されたのではありません）。HLAを使ってアセンブリ言語を学ぶのに最適な情報源は、『The Art of Assembly Language』（No Starch Press、2003年）です。この本は、この章に書かれている段階的な方法でアセンブリ言語を学ぶことを特に目的として書かれています。『The Art of Assembly Language』を読んだ大勢の学生や読者との経験が、このやり方の正しさを実証しています。高級言語の知識があり、アセンブリ言語プログラミングを学びたいと考えているのであれば、『The Art of Assembly Language』を一読することを真剣に検討してください。

もちろん、アセンブリ言語プログラミングに関する情報源は『The Art of Assembly Language』だけではありません。『Assembly Language Step-by-Step』（Jeff Duntemann著、Wiley、2000年）は、初めてのプログラミング言語としてアセンブリ言語プログラミングを学んでいる（つまり、高級プログラミング言語の経験がない）プログラマーに適しています。一般に、『Write Great Code』の読者はこの範疇には入りませんが、人によっては、このように別の方法でアセンブリ言語を学んだほうがうまくいくことがあるでしょう。

『Programming from the Ground Up』（Jonathon Barlett著、Bartlett Publishing、2004年）では、GNUのGasアセンブラを使ってアセンブリ言語プログラミングを解説しています。この本は、GCCの80x86出力を分析する必要がある人に特に役立ちます。この本の旧版が、Websterのサイト（`http://webster.cs.ucr.edu/AsmTools/Gas/index.html`）で無料で公開されています。

『Professional Assembly Language（Programmer to Programmer）』（Richard Blum著、Wrox、2005年）でも80x86でGNUアセンブラを使用しており、この本はGCCのGas出力を解読したい人には役立つでしょう。

Paul Carter博士もアセンブリ言語プログラミングに関する興味深いオンラインチュートリアルを公開しています。Carter博士の最新版のチュートリアルへのリンクは、Websterのサイトのリンクページ（`http://webster.cs.ucr.edu/links.htm`）にあります。

もちろん、Websterにはオンラインの電子テキストを含めて、アセンブリ言語プログラミングに関す

るたくさんの情報源があります。アセンブリ言語プログラミングについてさらに学習したい読者には、Websterのサイト（http://webster.cs.ucr.edu/）を訪れることをお勧めします。

アセンブリ言語プログラミングに関する情報源は、これだけではありません。適当な検索エンジンを使ってインターネットを検索するだけで、アセンブリ言語プログラミングに関する何千ページもの情報が見つかります。また、Usenetニュースグループのalt.lang.asmおよびcomp.lang.asm.x86では、アセンブリ言語プログラミングについてさまざまな質問を投稿できます。アセンブリ言語を扱ったいろいろなWebベースのフォーラムもあります。繰り返しになりますが、Googleなどの検索エンジンを使ってインターネットを検索すれば、すぐには読みこなせないほどたくさんの情報が見つかります。

第3章

高級言語プログラマーのための80x86アセンブリ言語

本書では、高級言語のコードを調べ、そのコードからコンパイラが生成したマシンコードとを比較します。コンパイラの出力の意味を理解するには、ある程度のアセンブリ言語の知識が必要です。アセンブリ言語の熟練プログラマーになるには、時間をかけて経験を積まなければなりません。幸いなことに、本書の目的のためには、そうしたスキルが必須というわけではありません。必要なのは、コンパイラやほかのアセンブリ言語プログラマーが生成したコードを読み解く能力だけです。アセンブリ言語プログラムを一から書けるようになる必要はありません。この章では、次の内容について説明します。

- **基本的な80x86マシンアーキテクチャ**
- **さまざまなコンパイラが生成する80x86出力を読み解くための80x86アセンブリ言語の概要**
- **32ビットの80x86 CPUがサポートするアドレス指定モード**
- **いくつかの一般的な80x86アセンブラ（HLA、MASM/TASM、Gas）で使われる構文**
- **アセンブリ言語プログラムで定数を使用する方法、およびデータを宣言する方法**

この章の内容に加えて、オンラインリソース（`http://www.writegreatcode.com/`）を参照することをお勧めします。このサイトには、コンパイラの出力を調べるときに必要となる最小限の80x86命令セットが掲載されています。

3.1 複数のアセンブリ言語習得の勧め

80x86以外のプロセッサ用のコードを書くつもりであれば、ぜひとも2種類以上のアセンブリ言語を読み解く方法を習得してください。そうすれば、高級言語で80x86用のコードを書いた後で、その「最適化」が80x86 CPUにしか有効でないことに気付くといった落とし穴にはまらずに済みます。そのために、第4章はPowerPC CPUアセンブリ言語の入門編になっています。両方のプロセッサファミリには共通の概念が多いものの、いくつかの重要な相違点もあります（相違点は付録にまとめています）。

おそらく、80x86などの複合命令セットコンピュータ（CISC）アーキテクチャと、PowerPCなどの縮小命令セットコンピュータ（RISC）アーキテクチャの主な違いは、メモリの使い方でしょう。RISCアーキテクチャのメモリアクセスは比較的融通が利かず、アプリケーションはメモリへのアクセスを回避するためにあらゆる手だてを尽くします。一方、80x86アーキテクチャではさまざまな方法でメモリにアクセスすることができ、アプリケーションは一般にこの融通性を利用します。どちらのやり方にも長所と短所がありますが、要するに、80x86で実行されるコードには、PowerPCなどのRISCアーキテクチャで実行される同等のコードとはいくつかの根本的な違いがあるということです。

3.2 80x86アセンブリ言語の構文

80x86プログラマーは幅広い選択肢の中からプログラム開発ツールを選ぶことができますが、こうした

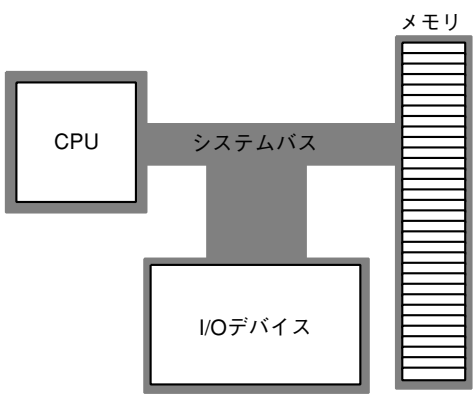
選択肢の多さは、構文上の非互換性という小さな欠点を伴います。同じ80x86ファミリ向けであっても、コンパイラとデバッガが違えば、まったく同じプログラムに対して異なるアセンブリ言語のリストが出力されることがあります。なぜなら、これらのツールが別々のアセンブラに対応したコードを出力するからです。たとえば、Microsoft Visual C++パッケージは、Microsoft Macro Assembler（MASM）と互換性のあるアセンブリコードを生成します。Borland C++コンパイラは、Borland Turbo Assembler（TASM）と互換性のあるコードを生成します。GNUコンパイラコレクション（GCC）は、Gas互換のソースコードを生成します（GasはFree Software Foundationが提供しているGNUアセンブラです）。コンパイラが出力するコードのほかに、FASM、NASM、GoAsm、HLAなどのアセンブラで書かれた膨大な量のアセンブリ言語プログラミングの例が見つかります。

本書全体を通して1つのアセンブラの構文だけを使用できればよいのでしょうが、本書の手法は特定のコンパイラに固有ではないので、よく使われる数種類のアセンブラの構文を考慮する必要があります。本書では、コンパイラの種類には依存しない例を示すときは、一般にHigh-Level Assembler（HLA）を使うことにします。したがって、この章ではMASM、TASM、Gas、HLAの4つのアセンブラの構文について説明します。幸いにも、1つのアセンブラの構文を習得した後で、別のアセンブラの構文を覚えるのはとても簡単です。これはBASICの1つの方言から別の方言に切り替えるようなものです。

3.3　80x86の基本アーキテクチャ

Intel CPUは、一般に**フォン・ノイマン型マシン**に分類されます。フォン・ノイマン型のコンピュータシステムには、**中央処理装置（CPU）**、**メモリ**、**入出力（I/O）デバイス**の3つの主要な基本構成要素があります。これらの3つのコンポーネントは（アドレスバス、データバス、コントロールバスから成る）**システムバス**を使って連結されます。この関係を示したのが図3-1のブロック図です。

図3-1　フォン・ノイマン型システムのブロック図



CPUはアドレスバスに数値を出力し、メモリ位置またはI/Oデバイスのポート番号の1つを選択してメモリやI/Oデバイスと通信します。メモリ位置とI/Oデバイスのポート番号には、それぞれ一意の2進数の**アドレス**が割り当てられています。その後、CPU、I/O、メモリデバイスは、データバスにデータを出力して互いにデータをやりとりします。コントロールバスには、データ転送の方向（メモリの書き込みか読み取りか、I/Oデバイスの書き込みか読み取りか）を決める信号が流れます。

3.3.1 レジスタ

レジスタは、CPU内で最も重要な機能です。80x86 CPUで実行されるほぼすべての計算で、最低1個のレジスタが使われます。たとえば、2つの変数の値を加算し、その合計をさらに別の変数に格納する場合は、変数の1つをレジスタにロードし、もう一方のオペランドをレジスタに加えた後、そのレジスタの値を格納先の変数に格納します。レジスタは、ほぼすべての計算の仲介役となります。そのため、80x86アセンブリ言語プログラムではレジスタが非常に重要です。

80x86 CPUのレジスタは、4つのカテゴリに分類できます。すなわち、汎用レジスタ、特殊用途アプリケーションアクセス可能レジスタ、セグメントレジスタ、特殊用途カーネルモードレジスタです。本書では、後の2つのレジスタについては考察しません。セグメントレジスタは、最近の32ビットオペレーティングシステム（Windows、BSD、BeOS、Linuxなど）ではあまり使用されません<監訳者注1>。また、特殊用途カーネルモードレジスタは、オペレーティングシステムやデバッガなどのシステムレベルのツールを作成するためのレジスタです。そのようなソフトウェアの作成は、本書の範疇を超えています。

3.3.2 80x86汎用レジスタ

80x86（Intelファミリ）CPUには、アプリケーションで使用するいくつかの汎用レジスタが用意されています。まず、次の名前を持つ8個の32ビットレジスタがあります。

```
EAX、EBX、ECX、EDX、ESI、EDI、EBP、ESP
```

名前の先頭のEは拡張（Extended）を意味します。この接頭語により、32ビットレジスタは次の名前を持つ8個の16ビットレジスタと区別されます。

```
AX、BX、CX、DX、SI、DI、BP、SP
```

さらに、80x86 CPUには次の名前を持つ8個の8ビットレジスタがあります。

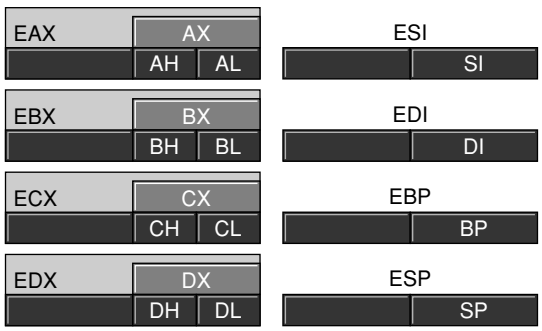
```
AL、AH、BL、BH、CL、CH、DL、DH
```

汎用レジスタに関する最も重要な注意点は、これらが互いに独立していないということです。つまり、80x86には24個の独立したレジスタはありません。代わりに、32ビットレジスタの一部は16ビットレジス

<監訳者注1>カーネルがセグメントレジスタの管理をしているので、アプリケーションレベルで操作することはありません。

タと重なり合い、16ビットレジスタの一部は8ビットレジスタと重なり合っています。この関係を示したのが図3-2です。

図3-2　Intel 80x86 CPUの汎用レジスタ



1つのレジスタを変更すると、最大で3個のほかのレジスタが変更される可能性があるという点は、特に強調しておく必要があります。たとえば、EAXレジスタを変更すると、AL、AH、AXの各レジスタも変更される可能性があります。コンパイラが生成するコードでは、80x86のこの特徴が使われることがよくあります。たとえば、コンパイラがEAXレジスタのすべてのビットをクリア（0にセット）した後、ALに1または0をロードして32ビットの真（1）または偽（0）の値を作るといった具合です。一部のマシン命令はALレジスタのみを操作するにもかかわらず、プログラムではそれらの命令の結果をEAXで返さなければならない場合があります。レジスタの重なり合いを利用することにより、コンパイラが生成したコードはALを操作する命令を使用して、その値をEAX全体で返すことができます。

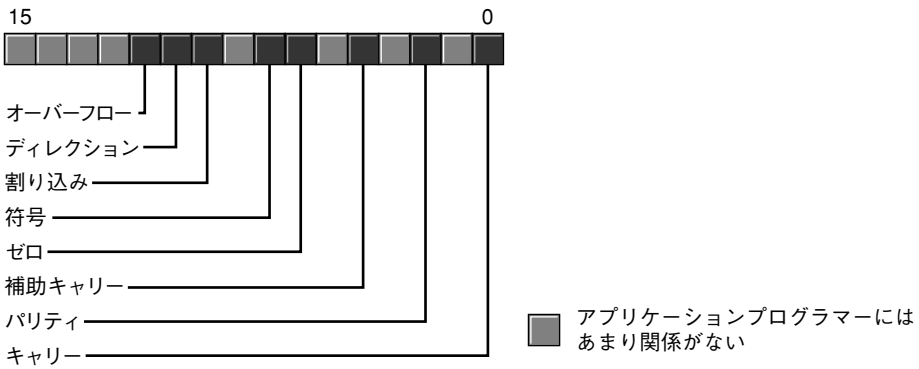
Intelではこれらのレジスタを**汎用**と呼んでいますが、すべてのレジスタを任意の用途に使用できるとは考えないでください。たとえば、SP/ESPレジスタには非常に特殊な用途があるため、実際上はそれ以外の用途には使用できません（これは**スタックポインタ**です）。同様に、BP/EBPレジスタにも特殊な用途があるため、汎用レジスタとしての有用性は制限されます。すべての80x86レジスタには個別の特殊な用途があり、特定の状況での使用が制限されます。こうした特殊な用途については、それらを使用するマシン命令の説明をするときに併せて説明します（オンラインリソースを参照してください）。

3.3.3　80x86 EFLAGSレジスタ

32ビットEFLAGSレジスタは、たくさんの単一ビットのブール（真/偽）値（**フラグ**）をカプセル化しています。これらのビットの大半は、カーネルモード（オペレーティングシステム）の機能用に予約されているか、アプリケーションプログラマーにはほとんど用がないかのどちらかです。アプリケーションプログラマーがアセンブリ言語コードを読む（または書く）ときに関係があるのは、オーバーフロー、

ディレクション、割り込み<注1>、符号、ゼロ、補助キャリー、パリティ、キャリーフラグの8つのビットです。図3-3に、EFLAGSレジスタの下位（LO：Low-Order）16ビット内のフラグの配置を示します。

図3-3 80x86EFLAGSレジスタの配置（下位16ビット）



アプリケーションプログラマーが使用できる8個のフラグのうち、特にオーバーフロー、キャリー、符号、ゼロの4個のフラグはきわめて役に立ちます。これらの4個のフラグを**条件コード**と呼びます。それぞれのフラグには（セットまたはクリアされる）状態があり、これを使って前の計算の結果を評価することができます。たとえば、2つの値を比較した後で条件コードフラグを調べると、一方の値がもう一方の値より小さいか、大きいか、それとも両方の値が等しいかがわかります。

3.4 リテラル定数

ほとんどのアセンブラは、数値リテラル定数<監訳者注2>、文字リテラル定数、文字列リテラル定数をサポートしています。あいにく、世の中に出ているほぼすべてのアセンブラが、リテラル定数に異なる構文を使用します。これ以降の節では、本書で使用するアセンブラの構文について説明します。

3.4.1 2進リテラル定数

すべてのアセンブラには、基数2（2進）リテラル定数を指定する機能があります。2進定数を出力するコンパイラはほとんどないので、おそらくコンパイラが生成する出力でこれらの値を目にすることはないでしょうが、手書きのアセンブリコードで見かけることがあるかもしれません。

<注1>アプリケーションプログラムでは割り込みフラグを変更できませんが、後からこのフラグを取り上げるので、ここで名前を挙げています。
<監訳者注2>リテラル定数とは、コード中で常に同じデータ（定数）が割当てされていることを意味します。たとえば数値リテラル定数は、ある数を表現するバイナリ値がマシンコード中に直接埋め込まれていることになります。

3.4.1.1 HLAの2進リテラル定数

HLAの2進リテラル定数は、パーセント記号（%）で始まり、その後ろに1つ以上の2進数（0または1）が続きます。2進数の任意の2つの数字の間にアンダースコア文字（_）を入れることもできます。慣習的に、HLAプログラマーはアンダースコアを使って4桁ごとに数字を区切ります。たとえば次のように指定します。

```
%1011
%1010_1111
%0011_1111_0001_1001
%1011001010010101
```

3.4.1.2 Gasの2進リテラル定数

Gasの2進リテラル定数は、0bという接頭語で始まり、その後ろに1つ以上の2進数値（0または1）が続きます。たとえば次のように指定します。

```
0b1011
0b10101111
0b0011111100011001
0b1011001010010101
```

3.4.1.3 MASMおよびTASMの2進リテラル定数

MASM/TASMの2進リテラル定数は、1つ以上の2進数値（0または1）と、bという接尾語から成ります。たとえば次のように指定します。

```
1011b
10101111b
0011111100011001b
1011001010010101b
```

3.4.2 10進リテラル定数

ほとんどのアセンブラの10進定数は、標準形式（特別な接頭辞や接尾辞が付かない1つ以上の連続する10進数の形式）をとります。10進リテラル定数は、コンパイラが出力する一般的な2つの数値形式の1つなので、コンパイラの出力を読むときに目にすることがよくあります。

3.4.2.1 HLAの10進リテラル定数

HLAでは、必要に応じて10進数の任意の2つの数字の間にアンダースコアを入れることができます。HLAプログラマーは一般に、アンダースコアを使って10進数を3桁ごとに区切ります。たとえば、次のような数字があるとします。

```
1024
1021567
```

HLAでは、次のようにアンダースコアを入れることができます。

```
1_024
1_021_567
```

3.4.2.2 Gas、MASM、TASMの10進リテラル定数

Gas、MASM、TASMでは、10進数値の並び（10進値の標準「コンピュータ」形式）を使います。たとえば次のように指定します。

```
1024
1021567
```

Gas、MASM、TASMでは、HLAと違って、10進リテラル定数にアンダースコアを入れることはできません。

3.4.3 16進リテラル定数

16進（基数16）リテラル定数は、アセンブリ言語プログラム（特にコンパイラが出力したプログラム）でよく目にする、もう1つの一般的な数値形式です。

3.4.3.1 HLAの16進リテラル定数

HLAの16進リテラル定数は、$という接頭語が付いた16進数値の並び（0..9、a..f、またはA..F<監訳者注3>）から成ります。必要に応じて、任意の2つの16進数の間にアンダースコアを入れることができます。慣習的に、HLAプログラマーはアンダースコアを使って4桁ごとに数字を区切ります。

たとえば次のように指定します。

```
$1AB0
$1234_ABCD
$dead
```

3.4.3.2 Gasの16進リテラル定数

Gasの16進リテラル定数は、0xという接頭語が付いた16進数値の並び（0..9、a..f、またはA..F）から成ります。たとえば次のように指定します。

```
0x1AB0
0x1234ABCD
0xdead
```

<監訳者注3>本書では、日本語における「〜」と同様に、「..」を範囲を表す記号として使用しています。

3.4.3.3 MASMおよびTASMの16進リテラル定数

MASM/TASMの16進リテラル定数は、hという接尾語が付いた16進数値の並び（0..9、a..f、またはA..F）から成ります。値の先頭は10進数（定数がa..fの範囲の数字で始まる場合は0）でなければなりません。たとえば次のように指定します。

```
1AB0h
1234ABCDh
0deadh
```

3.4.4 文字および文字列リテラル定数

文字および文字列データも、アセンブリプログラムでよく目にする一般的なデータ型です。MASMとTASMでは、文字と文字列のリテラル定数に区別はありません。しかし、HLAとGasでは、文字と文字列にそれぞれ別々の内部表現を使用するので、これらのアセンブラでは両者の形式を区別することが非常に重要になります。

3.4.4.1 HLAの文字および文字列リテラル定数

HLAの文字リテラル定数は、いくつかの異なる形式をとります。最も一般的な形式は、'A'のように単一の印字可能文字を2つのアポストロフィで囲んだものです。アポストロフィを文字リテラル定数として指定する場合は、2つのアポストロフィをアポストロフィで囲む必要があります（''''）。さらに、#記号の後ろに、文字のASCIIコードを示す2進、10進、16進数値のいずれかを使って文字定数を指定することもできます。たとえば次のように指定します。

```
'a'
''''
' '
#$d
#10
#%0000_1000
```

HLAの文字列リテラル定数は、引用符で囲んだ0個以上の連続する文字から成ります。文字列定数の中で引用符を使用する場合は、2つの隣接する引用符を使って表現します。

たとえば次のように指定します。

```
"Hello World"
"" -- 空の文字列
"He said ""Hello"" to them"
"""" -- 引用符文字が1つ含まれる文字列
```

3.4.4.2 Gasの文字および文字列リテラル定数

Gasの文字リテラル定数は、1つのアポストロフィの後ろに単一文字で表現します。たとえば次のように指定します。

```
'a
''
'!
```

Gasの文字列リテラル定数は、引用符で囲んだ0個以上の連続する文字から成ります。Gasの文字列リテラル定数では、Cの文字列と同じ構文を使います。Gasの文字列に特殊文字を挿入するときは、\によるエスケープシーケンスを使います。たとえば次のように指定します。

```
"Hello World"
"" -- 空の文字列
"He said \"Hello\" to them"
"\"" -- 引用符文字が1つ含まれる文字列
```

3.4.4.3 MASMおよびTASMの文字/文字列リテラル定数

MASM/TASMでは、文字および文字列リテラル定数は同じ形式をとります。どちらも0個以上連続する文字をアポストロフィまたは引用符で囲みます。これらのアセンブラでは、文字定数と文字列定数は区別されません。たとえば次のように指定します。

```
'a'
"'" - アポストロフィ文字
'"' - 引用符文字
"Hello World"
"" -- 空の文字列
'He said "Hello" to them'
```

3.4.5 浮動小数点リテラル定数

通常、アセンブリ言語の浮動小数点リテラル定数は、高級言語と同じ形式（一連の数字に、場合によって小数点が含まれ、その後ろに必要に応じて符号付きの指数が続く）をとります。たとえば次のように指定します。

```
3.14159
2.71e+2
1.0e-5
5e2
```

3.5 アセンブリ言語のシンボル定数

ほぼすべてのアセンブラには、（名前付き）シンボル定数を宣言するための仕組みが用意されています。実際に、ほとんどのアセンブラには、ソースファイルで値を識別子に関連付けるさまざまな方法が用意されています。

3.5.1 HLAのシンボル定数

HLAアセンブラでは、その名前のとおり、ソースファイルで名前付き定数を宣言するときに高レベルの構文を使用します。定数を定義する方法は3つあります。constセクションで定義する方法、valセクションで定義する方法、?コンパイル時演算子を使用する方法です。constセクションとvalセクションはHLAプログラムの宣言セクションにあり、構文は非常によく似ています。この2つのセクションの違いは、valセクションで定義する識別子には値を再割り当てできるのに対し、constセクションの識別子にはできないことです。これらの宣言セクションではさまざまなオプションがサポートされていますが、基本的な宣言は次のような形式をとります。

```
const
    someIdentifier := someValue;
```

この宣言の後でソースファイルに*someIdentifier*が出現すると、その識別子は*someValue*の値に置き換わります。たとえば次のように指定します。

```
const
    aCharConst := 'a';
    anIntConst := 12345;
    aStrConst := "String Const";
    aFltConst := 3.12365e-2;

val
    anotherCharConst := 'A';
    aSignedConst := -1;
```

HLAでは、?ステートメントを使うと、ソースファイル中の空白を入れられる場所ならどこにでもval宣言を挿入できます。定数を宣言セクションで宣言することが常に好都合とは限らないので、時にはこれが役立つことがあります。たとえば次のように指定します。

```
?aValConst := 0;
```

3.5.2 Gasのシンボル定数

Gasでは.equステートメントを使用して、ソースファイルでシンボル定数を定義します。このステートメントの構文は次のようになります。

```
.equ        symbolName, value
```

次に、Gasソースファイル内の「同等化（equate）」の例を示します。

```
.equ        false, 0
.equ        true, 1
.equ        anIntConst, 12345
```

3.5.3 MASMおよびTASMのシンボル定数

MASMとTASMにも、ソースファイル内でシンボル定数を定義するための数種類の方法が用意されています。その1つは、次のようにequディレクティブを使う方法です。

```
false       equ    0
true        equ    1
anIntConst  equ    12345
```

また、次のように=演算子を使う方法もあります。

```
false       =    0
true        =    1
anIntConst  =    12345
```

この2つの方法にはわずかな違いしかありません。詳細については、MASMおよびTASMのマニュアルを参照してください。

> **NOTE** 多くの場合、コンパイラは=形式よりもequ形式で出力する傾向があります。

3.6　80x86のアドレス指定モード

アドレス指定モード（アドレッシングモード）とは、命令オペランドにアクセスするためのハードウェア固有の仕組みです。80x86ファミリには、レジスタオペランド、即値オペランド、メモリオペランドという3種類のオペランドがあります。これ以降の節では、それぞれのアドレス指定モードについて説明します。

3.6.1　80x86のレジスタアドレス指定モード

多くの80x86命令は、80x86の汎用レジスタセットを操作できます。レジスタにアクセスするには、レジスタ名を命令オペランドとして指定します。

80x86 mov（移動）命令を使い、アセンブラがこの方式をどのように実装しているかをいくつかの例で見ていきましょう。

3.6.1.1　HLAのレジスタアクセス

HLAのmov命令は、次のように指定されます。

```
mov( source, destination );
```

この命令は、**ソース**（source）オペランドから**デスティネーション**（destination）オペランドにデータをコピーします。8ビット、16ビット、32ビットのレジスタが、この命令の有効なオペランドであるのは当然です。唯一の制限は、両方のオペランドが同じサイズでなければならないことです。

次に、実際の80x86 mov命令を見てみましょう。

```
mov( bx, ax );       // 値をBXからAXにコピーする
mov( al, dl );       // 値をALからDLにコピーする
mov( edx, esi );     // 値をEDXからESIにコピーする
```

3.6.1.2　Gasのレジスタアクセス

Gasでは、各レジスタ名の先頭にパーセント記号（%）を付けます。たとえば次のように指定します。

```
%al、%ah、%bl、%bh、%cl、%ch、%dl、%dh
%ax、%bx、%cx、%dx、%si、%di、%bp、%sp
%eax、%ebx、%ecx、%edx、%esi、%edi、%ebp、%esp
```

Gasのmov命令の構文はHLAの構文に似ていますが、丸カッコとセミコロンがなく、アセンブリ言語ステートメントをソースコードの1物理行にきちんと収める必要がある点が異なります。たとえば次のように指定します。

```
mov %bx, %ax        // 値をBXからAXにコピーする
mov %al, %dl        // 値をALからDLにコピーする
mov %edx, %esi      // 値をEDXからESIにコピーする
```

3.6.1.3 MASMおよびTASMのレジスタアクセス

MASMおよびTASMアセンブラでは、HLAと同じレジスタ名を使い、Gasに似た基本構文を使用しますが、オペランドの順序が逆になる点が異なります。つまり、movなどの標準的な命令は次のような形式をとります。

```
mov destination, source
```

次にMASM/TASM構文のmov命令の例を示します。

```
mov ax, bx        ; 値をBXからAXにコピーする
mov dl, al        ; 値をALからDLにコピーする
mov esi, edx      ; 値をEDXからESIにコピーする
```

3.6.2 即値アドレス指定モード

レジスタオペランドとメモリオペランドを使用できるほとんどの命令では、即値（定数）オペランドも使用できます。たとえば、次のHLA mov命令は、対応するデスティネーションレジスタに適切な値をロードします。

```
mov( 0, al );
mov( 12345, bx );
mov( 123_456_789, ecx );
```

多くのアセンブラでは、即値アドレス指定モードを使うときに、さまざまな種類のリテラル定数を指定可能です。たとえば、数値を16進、10進、2進形式のいずれかで指定できるほか、文字定数をオペランドとして指定することもできます。このモードでは、定数がデスティネーションオペランドに指定されたサイズに収まらなければならないという規則があります。次にHLA、MASM/TASM、Gasの例をいくつか示します。

```
mov( $'a', ch );  // HLA
mov $'a, %ch      // Gas
mov ch, 'a'      ;MASM/TASM

mov( $1234, ax ); // HLA
mov $0x1234, %ax   // Gas
mov ax, 1234h     ; MASM/TASM
```

```
mov( 4_012_345_678, eax ); // HLA
mov $4012345678, %eax       // Gas
mov eax, 4012345678        ; MASM/TASM
```

ほぼすべてのアセンブラで、シンボル定数名を作成し、それらの名前をソースオペランドとして指定できます。たとえば、HLAにはtrueとfalseの2つのブール型定数があらかじめ定義されているので、次のように、これらの名前をmov命令のオペランドとして指定できます。

```
mov( true, al );
mov( false, ah );
```

アセンブラによっては、ポインタ定数やその他の抽象データ型の定数を使用できるものもあります（詳細については、使っているアセンブラのリファレンスマニュアルを参照してください）。

3.6.3 変位のみのメモリアドレス指定モード

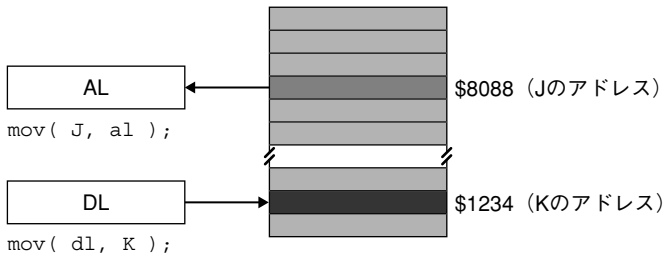
最も一般的で理解しやすいアドレス指定モードが**変位**（displacement：ディスプレースメント）**のみの**（**直接**）アドレス指定モードです。このモードでは、32ビットの定数でメモリ位置のアドレスを指定します。アドレスはソースオペランドまたはディスティネーションオペランドとして指定できます。

たとえば、変数Jがアドレス$8088に配置されているバイト変数であるとします。HLAのmov(J, al);命令は、メモリ位置$8088にあるバイトのコピーをALレジスタにロードします。同様に、バイト変数Kがメモリ内のアドレス$1234にある場合、mov(dl, K);命令はDLレジスタの値をメモリ位置$1234に書き込みます（図3-4を参照してください）。

変位のみのアドレス指定モードは、単純なスカラー変数へのアクセスに最適です。高級言語プログラムで静的変数またはグローバル変数にアクセスするときに通常使用するのが、このアドレス指定モードです。

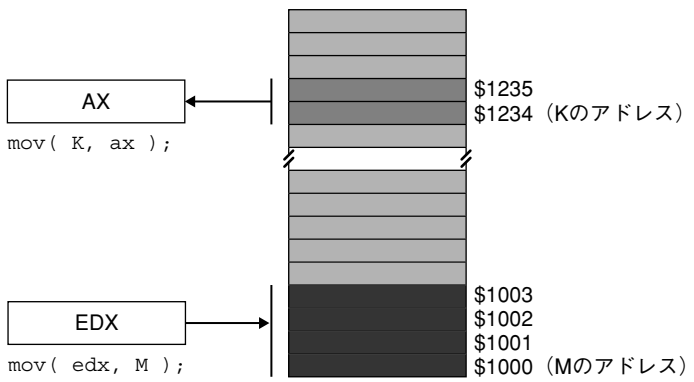
> **NOTE** **Intelがこのアドレス指定モードを「変位のみ」と名付けたのは、32ビットの定数（変位）がベースアドレスを指定せずにメモリ内のmovオペコードに続くからです。80x86プロセッサでは、この変位はメモリの先頭（つまりアドレス0）からのオフセットです。**

図3-4 変位のみの（直接）アドレス指定モード



この章の例では多くの場合、メモリ内のバイトサイズのオブジェクトにアクセスしています。しかし、80x86プロセッサでは、最初のバイトのアドレスを指定することにより、ワードやダブルワードにアクセスすることもできます（図3-5を参照してください）。

図3-5 直接アドレス指定モードによるワードやダブルワードへのアクセス



MASM、TASM、Gasでは、変位のみのアドレス指定モードにHLAと同じ構文を使います。つまり、オペランドにはアクセスするオブジェクトの名前をそのまま指定します。MASMやTASMのプログラマーの中には変数名を角カッコ（[と]）で囲む人もいますが、これは必須ではありません。

次にHLA、Gas、MASM/TASMの構文を示します。

```
mov( byteVar, ch );  // HLA
movb byteVar, %ch    // Gas
mov ch, byteVar      ;MASM/TASM

mov( wordVar, ax ); // HLA
movw wordVar, %ax   // Gas
mov ax, wordVar     ; MASM/TASM
```

```
mov( dwordVar, eax );    // HLA
movl dwordVar, %eax      // Gas
mov eax, dwordVar        ; MASM/TASM
```

3.6.4 レジスタ間接アドレス指定モード

80x86 CPUでは、レジスタ間接アドレス指定モードを使い、レジスタを介してメモリに間接的にアクセスできます。このモードを間接と呼ぶのは、オペランドが実際のアドレスではないからです。代わりに、オペランドの値で使用するメモリアドレスを指定します。レジスタ間接アドレス指定モードの場合は、レジスタの値でアクセスするアドレスを指定します。たとえば、HLAのmov(eax, [ebx]);命令では、EBXに格納されたアドレスの位置にEAXの値を格納するようにCPUに指示します。

3.6.4.1 HLAのレジスタ間接モード

80x86では、このアドレス指定モードに8つの形式があります。HLAの構文を使うと、次のようになります。

```
mov( [eax], al );
mov( [ebx], al );
mov( [ecx], al );
mov( [edx], al );
mov( [edi], al );
mov( [esi], al );
mov( [ebp], al );
mov( [esp], al );
```

これら8つのアドレス指定モードは、角カッコで囲まれたレジスタ（EAX、EBX、ECX、EDX、EDI、ESI、EBP、ESP）に格納されたオフセットにあるメモリ位置を参照します。

レジスタ間接アドレス指定モードには32ビットのレジスタが必要です。間接アドレス指定モードを使うときは、16ビットまたは8ビットのレジスタは指定できません。

3.6.4.2 MASMおよびTASMのレジスタ間接モード

MASMとTASMでは、レジスタ間接アドレス指定モードにHLAとまったく同じ構文を使い、レジスタ名を角カッコで囲みます。ただし、movなどの命令ではオペランドの順序が逆になります。

先ほどのHLAの命令をMASM/TASMで指定すると次のようになります。

```
mov al, [eax]
mov al, [ebx]
```

```
mov al, [ecx]
mov al, [edx]
mov al, [edi]
mov al, [esi]
mov al, [ebp]
mov al, [esp]
```

3.6.4.3 Gasのレジスタ間接モード

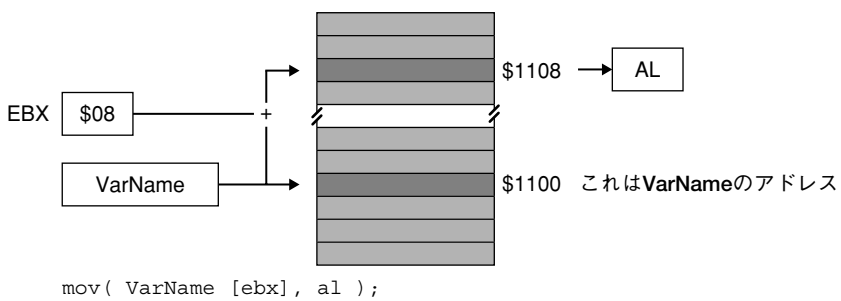
一方、Gasではレジスタ名を丸カッコで囲みます。前のHLAのmov命令をGasで指定すると次のようになります。

```
movb (%eax), %al
movb (%ebx), %al
movb (%ecx), %al
movb (%edx), %al
movb (%edi), %al
movb (%esi), %al
movb (%ebp), %al
movb (%esp), %al
```

3.6.5 インデックス付きアドレス指定モード

インデックス付きアドレス指定モードでは、角カッコ内の32ビットレジスタに格納されている値に、変数のアドレス（**変位**または**オフセット**とも呼ばれます）を加えて**実効アドレス**<注2>を計算します。その合計が、命令がアクセスするメモリアドレスになります。したがって、VarNameがメモリ内のアドレス$1100にあり、EBXに8が格納されている場合、mov(VarName [ebx], al);はアドレス$1108にあるバイトをALレジスタにロードします（図3-6を参照してください）。

図3-6 インデックス付きアドレス指定モード



<注2>実効アドレスは、すべてのアドレス計算が終了した後で命令がアクセスするメモリ内の最終的なアドレスです。

3.6.5.1 HLAのインデックス付きアドレス指定モード

インデックス付きアドレス指定モードでは、次のHLA構文を使います。VarNameは、プログラム内の何らかの静的変数の名前です。

```
mov( VarName[ eax ], al );
mov( VarName[ ebx ], al );
mov( VarName[ ecx ], al );
mov( VarName[ edx ], al );
mov( VarName[ edi ], al );
mov( VarName[ esi ], al );
mov( VarName[ ebp ], al );
mov( VarName[ esp ], al );
```

3.6.5.2 MASMおよびTASMのインデックス付きアドレス指定モード

MASMとTASMはHLAと同じ構文をサポートしていますが、ほかにもさまざまな形式でインデックス付きアドレス指定モードを指定できます。次にMASMとTASMでサポートされるいくつかの形式を示します。

```
varName[reg32]
[reg32+varName]
[varName][reg32]
[varName+reg32]
[reg32][varName]
varName[reg32+const]
[reg32+varName+const]
[varName][reg32][const]
varName[const+reg32]
[const+reg32+varName]
[const][reg32][varName]
varName[reg32-const]
[reg32+varName-const]
[varName][reg32][-const]
```

MASMとTASMでは、これ以外にも多くの組み合わせを使用できます。これらのアセンブラは、角カッコ内の並列する2つの項目を、+演算子で区切られているかのように扱います。加算には可換性があるため、数多くの組み合わせが生まれます。

次に示すのは、「3.6.5.1 HLAのインデックス付きアドレス指定モード」で示したHLAの例をMASM/TASMの構文に置き換えたものです。

```
mov  al, VarName[ eax ]
mov  al, VarName[ ebx ]
mov  al, VarName[ ecx ]
mov  al, VarName[ edx ]
mov  al, VarName[ edi ]
```

```
mov  al, VarName[ esi ]
mov  al, VarName[ ebp ]
mov  al, VarName[ esp ]
```

3.6.5.3 Gasのインデックス付きアドレス指定モード

レジスタ間接アドレス指定モードの場合と同様に、Gasでは角カッコではなく丸カッコを使用します。次にGasでインデックス付きアドレス指定モードに使用できる構文を示します。

```
varName(%reg32)
const(%reg32)
varName+const(%reg32)
```

前に示したHLAの命令をGasの構文に置き換えると、次のようになります。

```
movb VarName( %eax ), %al
movb VarName( %ebx ), %al
movb VarName( %ecx ), %al
movb VarName( %edx ), %al
movb VarName( %edi ), %al
movb VarName( %esi ), %al
movb VarName( %ebp ), %al
movb VarName( %esp ), %al
```

3.6.6 スケールドインデックス付きアドレス指定モード

スケールドインデックス付きアドレス指定モードは、インデックス付きアドレス指定モードに似ていますが、2つの点が異なります。スケールドインデックス付きアドレス指定モードでは、次の操作が可能です。

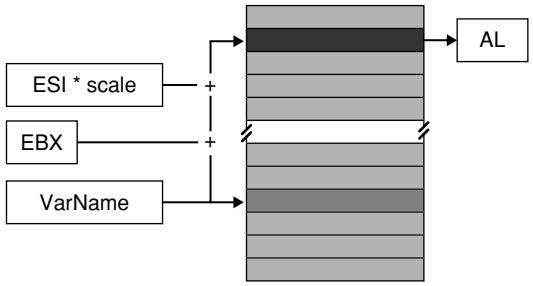
- **2つのレジスタと変位を組み合わせる**
- **インデックスレジスタにスケール値として1、2、4、8のいずれかを掛ける**

これで何ができるかを知るために、次のHLAの例を見てください。

```
mov( eax, VarName[ ebx + esi * 4 ] );
```

スケールドインデックス付きアドレス指定モードとインデックス付きアドレス指定モードの主な違いは、`esi * 4`の部分が含まれていることです。この例では、ESIに4を掛けた値を加えて実効アドレスを計算しています（図3-7を参照してください）。

図3-7 スケールドインデックス付きアドレス指定モード




```
mov( VarName [ebx + esi * scale], al );
```

3.6.6.1 HLAのスケールドインデックス付きアドレス指定

HLAの構文では、さまざまな方法でスケールドインデックス付きアドレス指定モードを記述できます。次に、各種の構文形式を示します。

```
VarName[ IndexReg32 * scale ]
VarName[ IndexReg32 * scale + displacement ]
VarName[ IndexReg32 * scale - displacement ]

[ BaseReg32 + IndexReg32 * scale ]
[ BaseReg32 + IndexReg32 * scale + displacement ]
[ BaseReg32 + IndexReg32 * scale - displacement ]

VarName[ BaseReg32 + IndexReg32 * scale ]
VarName[ BaseReg32 + IndexReg32 * scale + displacement ]
VarName[ BaseReg32 + IndexReg32 * scale - displacement ]
```

これらの例で、`BaseReg32`は任意の汎用32ビットレジスタを表します。`IndexReg32`はESPを除く任意の汎用32ビットレジスタを表します。`scale`は1、2、4、8のいずれかの定数でなければなりません。`VarName`は静的変数名を表します。

3.6.6.2 MASMおよびTASMのスケールドインデックス付きアドレス指定

MASMとTASMは、スケールドインデックス付きアドレス指定モードにHLAと同じ構文をサポートしていますが、インデックス付きアドレス指定モードの場合と同様に、それ以外の形式もサポートしています。これらの形式は、構文上は+演算子の可換性によって変形されたものにすぎません。

3.6.6.3 Gasのスケールドインデックス付きアドレス指定

例によって、Gasでは角カッコではなく丸カッコを使ってスケールドインデックスのオペランドを囲みます。また、ほかのアセンブラのように算術式の構文を使わずに、Gasではオペランド3個の構文を使って**ベースレジスタ、インデックスレジスタ、スケール値**を指定します。Gasのスケールドインデックス付

きアドレス指定モードの構文は次のとおりです。

```
expression( baseReg32, indexReg32, scaleFactor )
```

具体的には、次のように指定します。

```
VarName( ,IndexReg32, scale )
VarName + displacement( ,IndexReg32, scale )
VarName - displacement( ,IndexReg32, scale )

( BaseReg32, IndexReg32, scale )
displacement( BaseReg32, IndexReg32, scale )

VarName( BaseReg32, IndexReg32, scale )
VarName + displacement( BaseReg32, IndexReg32, scale )
VarName - displacement( BaseReg32, IndexReg32, scale )
```

3.7 アセンブリ言語のデータ宣言

80x86には、個々のマシン命令で操作する低レベルのマシンデータ型がほんのわずかしか用意されていません。用意されているデータ型は次のとおりです。

- 任意の8ビット値を保持する**バイト**（**byte**）
- 任意の16ビット値を保持する**ワード**（**word**）
- 任意の32ビット値を保持する**ダブルワード**（**dword**）
- 32ビット単精度浮動小数点数値を保持する**Real32オブジェクト**（**Real4オブジェクト**とも呼ばれる）
- 64ビット倍精度浮動小数点数値を保持する**Real64オブジェクト**（**Real8オブジェクト**とも呼ばれる）

> **NOTE** 80x86アセンブラは一般に、TByte（ten-byte）およびReal80/Real10データ型をサポートしていますが、ここではそれらの使用については考慮しません。ほとんどの高級言語コンパイラは、これらのデータ型を使用しないからです （ただし、いくつかのC/C++コンパイラはlong doubleデータ型を使ってReal80の値をサポートしています）。

3.7.1 HLAのデータ宣言

HLAアセンブラは高級言語の性質を備えており、文字、符号付き整数、符号なし整数、ブール型、列挙型などの多彩な1バイトのデータ型をサポートしています。アセンブリ言語で実際にアプリケーションを書くのであれば、このように各種のデータ型が（HLAの型チェックと共に）揃っているととても便利です。しかし本書の目的に必要なのは、バイト変数用に記憶域を割り当て、より大きいデータ構造用にバイトブロックを確保することだけです。8ビットオブジェクトと配列オブジェクトにはHLA byte型があれば十分です。

HLAのstaticセクションでbyte型のオブジェクトを宣言するには、次のように指定します。

```
static
    variableName : byte;
```

バイトブロック用の記憶域を割り当てるには、次のHLA構文を使います。

```
static
    blockOfBytes : byte[ sizeOfBlock ];
```

これらのHLAの宣言では、**初期化されていない**変数が作成されます。厳密に言うと、HLAは常にstaticオブジェクトを0で初期化するので、実際に初期化されていないわけではありません。肝心なのは、このコードがこれらのバイトオブジェクトを値で明示的に初期化しないという点です。ただし、次のようなステートメントを使用して、オペレーティングシステムがプログラムをメモリにロードするときに、バイト変数を値で初期化するように指定できます。

```
static
    // InitializedByteに初期値の5を格納する：

    InitializedByte : byte := 5;

    // InitializedArrayを0、1、2、3で初期化する

    InitializedArray : byte[4] := [0,1,2,3];
```

3.7.2 MASMおよびTASMのデータ宣言

MASMやTASMでは、通常は.dataセクション内でdbまたはbyteディレクティブを使用して、バイトオブジェクトまたはバイトオブジェクトの配列用の記憶域を確保します。単独の宣言の構文は、次のいずれかの形式をとります。

```
variableName    db      ?
variableName    byte    ?
```

試し読みはお楽しみ
いただけましたか？

ここからはManatee
おすすめの商品を
ご紹介します。

Tech Book Zone Manatee

3.15
2017

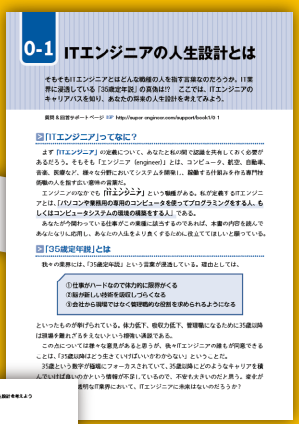
おすすめ電子書籍

Tech Book Zone Manatee

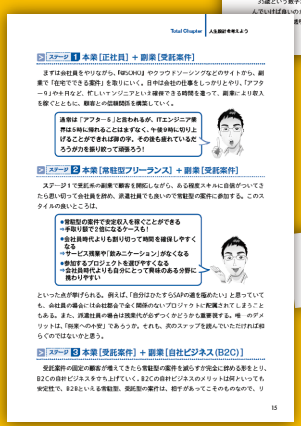
1

副業? 独立? それとも?
ITエンジニアの人生設計の決定版!

組織を束ねるマネジャーになるか、現場のスペシャリストであり続けるか。自分の技術を活かして独立するか、副業を考えるか……。意外に悩ましいITエンジニアの人生設計。会社に依存しない、転職や独立も射程に入れた「マインドセット」の持ち方から、お金、営業戦略、顧客対応術、ビジネスモデルの構築といった「ビジネスロジック」まで。求人情報ポータルサイト「@SOHO」の開発者が、自身の知見と経験から得たノウハウを教えます。



トピックスごとにオンラインの公式サイトで質問コーナーを用意。疑問に答えてくれる



ITエンジニアがどのように人生を描き切り開いていけるか、著者の実体験をもとに解説

「開発技法」

推奨職種を併記。アーティストやレベルデザイナなど専門ごとに読むべき章がわかる

プログラムやスクリプトの勉強をしたことがない人でも読み進められる

2

UE4におけるゲーム制作で
必須の知識と経験が身につく!

『Unreal Engine 4で極めるゲーム開発』は、Unreal Engine 4（UE4）の機能を単に紹介するだけでなく、一本のサンプルゲーム開発に順序よく盛り付けていく構成になっており、機能の組み合わせ方や、実践的な使い方をラーニングできます。3Dゲーム開発の一般的なワークフローやプロセスも解説し、章の構成も実際の開発プロセスに近づけました。今使っている人にも、これから始める人にも、すべてのUE4ユーザーにオススメの一冊!

開発系読み物

IT エンジニアのための「人生戦略」の教科書
技術を武器に、充実した人生を送るための「ビジネス」と「マインドセット」

マイナビ出版
平城寿（著者） 256 ページ
価格：2,462 円（PDF・EPUB）

開発ツール

Unreal Engine 4 で極めるゲーム開発

ボーンデジタル
湊和久（著者）
592 ページ
価格：4,860 円（PDF）

Tech Book Zone Manatee

「開発技法」

3.15 2017

ワークフローを疑似体験！ GitHub が初歩からわかる

開発ツール

3

Web 制作者のための GitHub の教科書 チームの効率を最大化する 共同開発ツール

Web 制作における「GitHub」の使い方が、実際のワークフローをイメージしながら理解できます。「そもそもどんなサービスなの ?」「どういうときにどの機能を使えばいいの ?」といった初歩の疑問から解説します。

インプレス
塩谷啓・紫竹佑騎・原一成・平木聡（著者）
224 ページ　価格：2,052 円（PDF）

Docker が利用される 現場のノウハウが凝縮！

開発ツール

4

Docker 実践ガイド

Docker が利用される環境や背景をはじめ、導入前のシステム設計、基本的な利用方法、Dockerfile による自動化の手法、プロセッサ、ネットワーク、ストレージなどの資源管理、管理・監視ツールについて解説します。

インプレス
古賀政純（著者）
328 ページ　価格：3,240 円（PDF）

チーム改善に活かす ITIL 悩めるリーダーにオススメ

開発系読み物

5

新米主任 ITIL 使ってチーム改善します！

化粧品メーカーで主任に昇格した友原京子。異動先は問題だらけのハチャメチャ部署だった…。『新人ガール ITIL 使って業務プロセス改善します !』の第 2 弾。英国生まれの IT 運用ノウハウ「ITIL」をチーム改善に活かします。

シーアンドアール研究所
沢渡あまね（著者）
304 ページ　価格：1,750 円（PDF）

プロトタイピングによって 初期段階での可能性を探る

開発系読み物

6

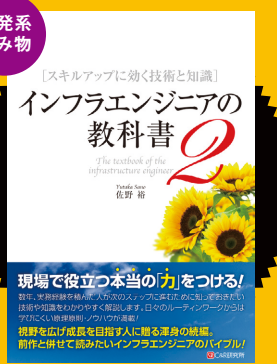
プロトタイピング実践ガイド スマホアプリの効率的なデザイン手法

本書で解説するプロトタイピングは、紙などを使った「低精度プロトタイピング」を中心とした手法です。設計フェーズの早期段階から作成し、検証と改善によって、機能要件や UI 設計、デザインを具現化していけます。

インプレス
深津貴之・荻野博章（著者）
240 ページ　価格：2,592 円（PDF）

インフラエンジニアの 必須知識をていねいに解説

開発系読み物



7

インフラエンジニアの教科書 2 スキルアップに効く技術と知識

数年間インフラエンジニアの経験を積んでいても「自分は詳しく知らないし、他の人に説明できない」といったことがあります。本書は実務経験を積んだインフラエンジニアを対象に、必須知識をわかりやすく解説します。

シーアンドアール研究所
佐野裕（著者）　価格：2,070 円（PDF・EPUB）

エミュレータ制作を通して コンピュータの中身を理解

開発系読み物

8

自作エミュレータで学ぶ x86 アーキテクチャ コンピュータが動く仕組みを徹底理解！

機械語やアセンブリ言語が CPU でどう実行されるか意識することはめったにありません。本書ではエミュレータの制作を通して x86 CPU の仕組み、メモリ・キーボード・ディスプレイといった部品と CPU の関わりを学びます。

マイナビ出版　内田公太・上川大介（著者）　196 ページ
価格：2,324 円（PDF）