第1章 アセンブラとは何か

見本

はじめに

現在，パソコンは日常生活において不可欠といえる道具になりました．ふつうは，パソコンを使う人は，パソコンがどのようなしくみで動いているのかを知る必要はありません．それは，文章を書いたり，画像や音などのデータを加工したり，インターネットに接続したりと，自分が必要とするアプリケーションを巧みに使いこなすことさえできれば良いからです．

一方で，パソコンのしくみを詳しく知っていなければならない人もたくさんいます．たとえば，パソコンを作っている技術者は当然ですが，それ以外にもパソコンのアプリケーションを作る人，パソコンの周辺装置を作る人，パソコンを計測や制御に利用しようと考えている人などです．

① アプリケーション・プログラムを作成する場合はアセンブリ言語は必要ない

ほんの少し前までは，このような目的を実現するためにはパソコンのハードウェアについて詳細に知っている必要がありました．たとえば，プログラムを処理するプロセッサの内部レジスタの動作，周辺装置をコントロールする専用LSIの内部レジスタの動作などです．そして，これらを理解して制御しようとすると，どうしてもアセンブリ言語[注1]によるプログラミングが必要でした．

しかし最近では，WindowsやLinuxといったOS，周辺機器の制御といった低レベルの仕事を代行するようになりました．したがって，アプリケーション・プログラムを作成する場合は，C言語を始めとするC++やJava，BASICといった高級言語と呼ばれるプログラミング言語を使用し，これらのプログラムからOSが用意したインターフェース・モジュールを呼び出すことで周辺機器の制御ができるようになりました．そのため，今ではアプリケーション・プログラムの作成を行うような場合には，アセンブリ言語を使う必要はまずありません．

② 直接ハードウェアを制御するプログラムはアセンブリ言語が必要

それでは，アセンブリ言語に関する知識はまったく不要になってしまったのでしょうか．

その答は，「YES」でもあり「NO」でもあります．OSの上で動作するアプリケーション・プログラムだけを作成するような場合は，C言語を代表とする高級言語だけを使用すれば目的を達成できるので，アセンブリ言語を使用する必要はありません．

しかし，直接ハードウェアを制御するプログラムであるデバイス・ドライバ[注2]を開発するような場合は，アセンブリ言語に関する知識が必要になってきます．また，組み込み製品を開発するような場合にも，デバッグするときは機械語レベルでプログラムの動作を追う必要が生じるので，アセンブリ言語を理解しなければならなくなります．さらに，組み込み用に使われているCPUのプログラムの開発などでは，メモリの制限などから，アセンブリ言語が今もまだ多く使われています(**図1**)．

これらは，ごく一部の人だけに必要な知識と言えるかもしれませんが，アセンブラを理解するともう一つ重要なことが理解できるようになります．それは，パソコンのハードウェアを理解できるようになるという

注1：アセンブリ言語とアセンブラの関係は，C言語とCコンパイラという関係に相当する．

注2：Windows(95/98/NT/2000)のDDKと呼ばれるデバイス・ドライバを開発するためのツールには，MLと呼ばれるアセンブラが付属している．Windows95/98の仮想デバイス・ドライバを開発するには，C言語も使用できるが，アセンブラで作成することが基本となっている．

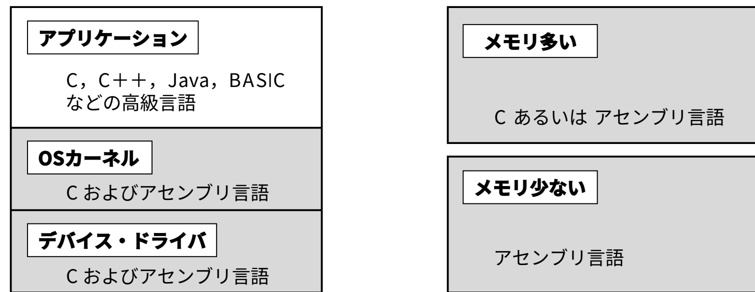


図1
アセンブリ言語の使用分野

(**a**) パソコンで使用されるプログラム言語　(**b**) 組み込みシステムで使用されるプログラム言語

ことです．C言語などでプログラムを書いていると，ハードウェアを制御するプログラムを書くことはできるのですが，具体的にどのようなしくみで動いているのか感覚的に理解することができません．しかし，アセンブリ言語のプログラムを見ると，データがどのように伝えられて制御が行われているのかが，手にとるようにわかります．

パソコンのアプリケーションを作成している人は，どうしてこのようにプログラムを書くと思ったとおりに制御ができるのか，つねに疑問を感じている人が多いはずです．そのようなとき，アセンブラを理解していればそのような疑問を必ず解消されます．ほかの例で言うと，ICやLSIを使うために中の回路で使われているトランジスタ回路まで理解する必要はありませんが，理解していればICやLSIの動作がより理解できることに似ています．

そこで本書では，パソコンの内部をより理解し，よりよいアプリケーションを作成するために役立つアセンブリ言語について学習することをめざします．

1.1 CPUとアセンブリ言語

一口にアセンブリ言語と言っても，CPUごとに異なった仕様のアセンブリ言語が存在します．アセンブリ言語はCPUの機能そのものを表しているプログラム言語だからです．つまり，アセンブリ言語を理解することは，コンピュータのハードウェア，とくにCPUを理解することにつながります．

これは逆に，アセンブリ言語を理解するためには，対象とするCPU，およびハードウェアを理解する必要があるということです．アセンブリ言語が使われなくなった理由の一つに，ハードウェアが変われば，それまでに蓄積したノウハウの再利用ができないということもありました．

そこで，本書ではハードウェアを理解する上で重要ではあるものの，現在では説明される機会が少なくなったアセンブラについて，初心者でも理解できるように，現在，もっとも普及しているPentium系CPUを対象にして解説します．CPUの内部構造とアセンブリ言語との関係，そしてアセンブリ言語によるプログラミングがおもな内容です．

「アセンブリ言語はCPUの機能そのものを表している」と述べましたが，ここではCPUとアセンブリ言語の関係についてもう少し詳しく見てみることにします．

1.1.1　機械語とアセンブリ言語

CPUがプログラムを実行する場合，CPUが理解できるのは機械語(machine language)と呼ばれるメモリ上のビットのONとOFFの組み合わせで表される命令です．言い方を変えると1と0の組み合わせで作られた命令コードだけです．

だからといって，人間が機械語命令でプログラムを作るのはたいへんな作業です．そこで，使われるようになったのがアセンブリ言語です.アセンブリ言語は，この1と0の組み合わせで表された機械語命令を，人間が理解できるようなニモニック・コードと呼ばれる単語で表すことにより，プログラムを作りやすくしたものです(**図2**)．

ニモニックには，機械語命令の動作がわかるような単語が使われます．そのため，人間にとっては数字の羅列でしかない機械語命令に比べ，機械語命令の動作がわかるような単語で表されたニモニックでプログラミングしたほうがはるかにプログラムが簡単になります．

実際のアセンブリ言語では，ニモニックはさらにオペランドと呼ばれる定数やCPUのレジスタ指定，メモリのアドレスといった値をもちます．

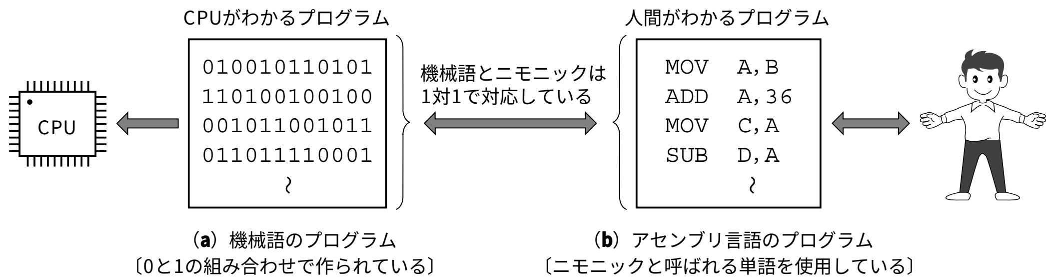


図2 機械語とアセンブリ言語

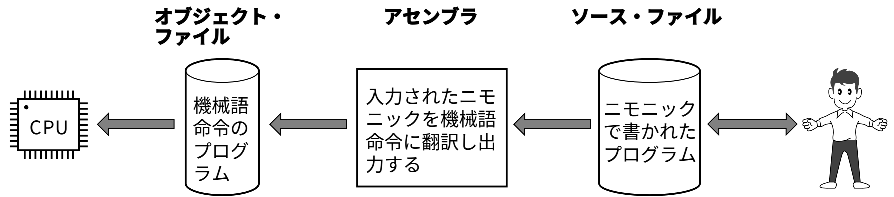


図3 アセンブラの動作

このニモニックを機械語命令に翻訳するソフトウェアのことをアセンブラ(assembler)と呼びます．そして，ニモニックを機械語命令に翻訳する作業をアセンブル(assemble)あるいはアセンブリ(assembly)と言います．

実際のアセンブラは，ユーザがアセンブリ言語で書いたソース・ファイルを入力し，ニモニックをCPUがわかる機械語命令に翻訳(アセンブル)し，オブジェクト・ファイルとして出力しています(**図3**)．

1.1.2 コンピュータの基本的な構成

先に述べたように，アセンブリ言語を理解するためには，まずコンピュータのハードウェアを知る必要があります．

そこで，ここではコンピュータの基本構成について，その概略を説明します．

① CPUとメモリ，I/O

コンピュータは，大きくCPUとメモリ，そしてI/Oに分けられます(**図4**)．

CPUは，“Central Processing Unit”を略したもので，日本語にすると「中央処理装置」となります．このCPUという名称は，コンピュータが巨大なロッカーのような形状をしていた時代に付けられたものです．そのため，このような「中央処理装置」という大げさな名前が付けられたわけです．その当時は，メモリも含めてCPUと呼ばれていました．

ロッカー・サイズだったCPUも，今ではマイクロプロセッサとなり，その核となる部分は指先に載るほどの小さなICチップになっています．ただし，現在使われているマイクロプロセッサは，その小ささゆえに汎用品においては，大容量のメモリを内蔵することはできないため，メモリは外部に接続するようになっています．

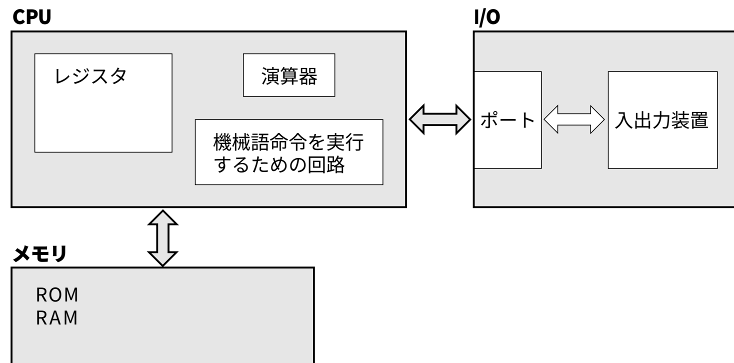


図4 コンピュータの基本構成

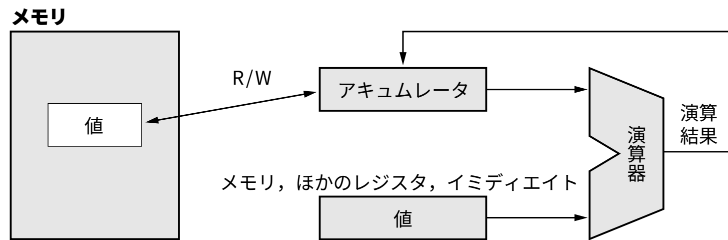


図5
アキュムレータの役目

CPUの基本構成は，どのようなCPUでも「レジスタ」,「演算器」,「機械語命令を実行するための回路」から作られています．実際のCPUでは，これに機械語命令の実行を速くするためのくふうや，演算の精度を上げたり，高速に演算するためのくふうがなされています．

I/Oとは，“ Input/Output ”のことで，データを入出力する装置，あるいは入出力装置に接続するためのポートのことをいいます．

マイクロプロセッサの接続方法には，I/OポートをRAMやROMのメモリ空間とは別のI/Oポート専用の空間に接続する方法と，RAMやROMが接続されているメモリ空間にI/Oポートも接続するという方法の二つの方法が使われています．

Intel社が開発したマイクロプロセッサは，過去のCPUから現在のPentiumに至るまで，I/Oポート専用の空間をもつ方式を採用しています．

実は，このI/Oポート専用の空間をもつ方法は，CPUの構成としては特殊であり，他社の多くのマイクロプロセッサは，I/OポートをRAMやROMが接続されているメモリ空間にそのまま接続する方法を採用しています．このように，RAMやROMが接続されているメモリ空間にI/Oポートを接続する方法を「メモリ・マップドI/O」と呼びます．

ちなみに，辞書によるとポート(port)には「貿易港」という意味があります．つまり，コンピュータで処理するデータの内部と外界との出入り口という意味でポート(貿易港)ということばが使われているのです．

② レジスタ

レジスタ(register)は，CPU内にあるデータを一時的に記憶しておくための回路のことです．CPUは，一般に複数個のレジスタをもっています．

レジスタは，その用途ごとに「アキュムレータ」,「ポインタ」,「コントロール」,「フラグ」に分けることができます．

ただし，CPUによっては，「アキュムレータ」と「ポインタ」は用途別にレジスタを分けるのではなく，すべてのレジスタが「アキュムレータ」にも「ポインタ」にも使用できるようになっているものがあります．

また，CPU内に複数個ある「アキュムレータ」や「ポインタ」を総称して，「汎用レジスタ」と呼ぶことがあります．

(a) アキュムレータ

アキュムレータ(accumulator)は，日本語では「累算器」と訳されます．CPUによっては「データ・レジスタ」と呼ばれる場合もあります．

アキュムレータは，演算処理で使われるデータを格納するために使用されます．

CPUが演算を行う場合，

- アキュムレータとほかのレジスタ
- アキュムレータとメモリ

といったぐあいに，このアキュムレータを中心として演算が行われます(**図5**)．

アキュムレータのビット数は，CPUのデータのビット数に合わせて決められています．たとえば，データが8ビットのCPUなら8ビット長，16ビットのCPUなら16ビット長，32ビットのCPUなら32ビット長といったぐあいです．

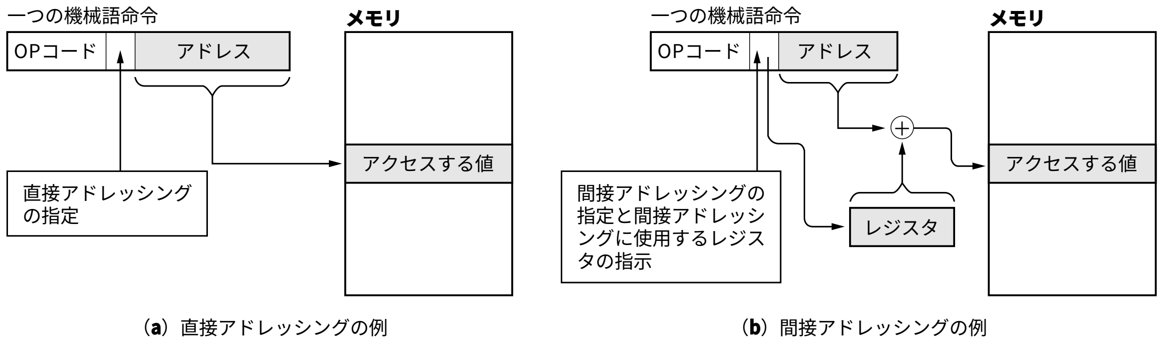
(b) ポインタ

ポインタ(pointer)は，メモリをアクセスするときに使用するアドレスを格納するためのレジスタで，CPUによっては「インデックス・レジスタ」とか「アドレス・レジスタ」とも呼ばれています．

CPUがメモリをアクセスする方法には，「直接アドレッシング(direct addressing)」と「間接アドレッシング(indirect addressing)」の二つの方法があります．

直接アドレッシングとは，機械語命令にアクセスするメモリのアドレスが，直接指定されているもののことです〔**図6(a)**〕．

間接アドレッシングでは，機械語命令はアクセスするメモリのアドレスを格納しているレジスタを示します．実際にアクセスされるデータは，レジスタの値を



（a）直接アドレッシングの例　（b）間接アドレッシングの例

図6　直接アドレッシングと間接アドレッシングの例

アドレスとしてアクセスする方法です〔**図6**(**b**)〕.

ポインタは，この間接アドレッシングを使うためのレジスタです．ポインタのビット数は，CPUのアドレスのビット数に合わせて決められています．たとえば，アドレスが16ビットのCPUなら16ビット長，32ビットのCPUなら32ビット長になります．

(c) コントロール

コントロール(control)は，CPU自体をコントロール，つまり制御するためのレジスタで，CPUによってその内容が異なります．構造が簡単なCPUには，このコントロール・レジスタがないものもありますが，現在よく使われている汎用的なマイクロプロセッサは，ほとんどこのコントロール・レジスタをもっています．

たとえば，CPUが仮想記憶をサポートしている場合は，そのためのコントロール・レジスタをCPUはもっています．また，キャッシュ・メモリをもつCPUなら，キャッシュを制御するためのコントロール・レジスタがあります(**図7**)．

(d) フラグ

フラグは，CPUの状態や演算の結果を表すために使用したり，CPUを制御したりするビットの集まりをレジスタにしたものです(**図8**)．CPUによっては，「ステータス・レジスタ」と呼ぶ場合もあります．

フラグ・レジスタは，アクセスする単位としてレジスタの形態をとっていますが，個々のフラグは独立しています．そのため，使用頻度が高いフラグについては，フラグを検査，セット，リセットするための専用の機械語命令が用意されています．

演算の結果を表すフラグとしては，ゼロ，サイン，

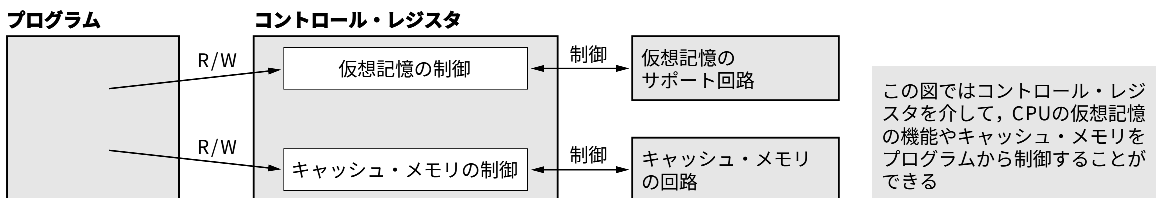


図7　コントロール・レジスタの例

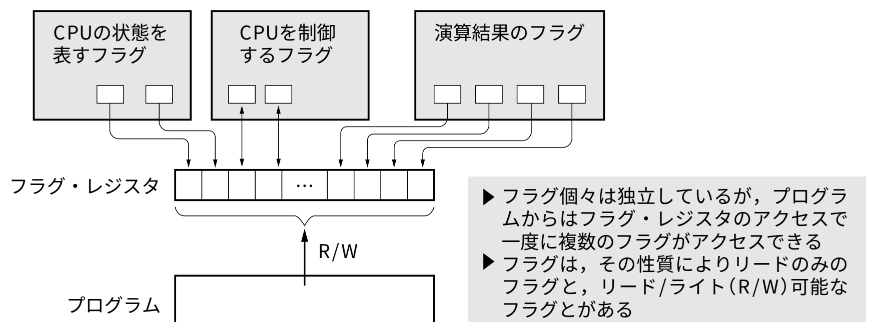


図8
フラグ・レジスタの例

表1
演算結果を表すフラグの例

フラグ	内　容
ゼロ	演算の結果がゼロになるとONになるフラグ
サイン	演算の結果が負の値になるとONになるフラグ
キャリ	加算の結果，桁上がりが発生するとONになる 減算の結果，桁借りが発生するとONになる
オーバフロー	演算の結果がオーバフローするとONになるフラグ

キャリ，オーバフローといったフラグがあります(**表1**).

また，CPUの状態を表したり，CPUを制御するビットもフラグとして定義されている場合があります．たとえば，現在のCPUの実行モードを表すフラグや，割り込みの許可/禁止を行うフラグなどです．これらのフラグへのアクセスはフラグ・レジスタで行いますが，その内容はコントロール・レジスタと同じであり，CPU自体をコントロールする目的で使われます．

③ 演算器

CPUの性能は，整数の論理演算と加減算しか行えないものから，浮動小数点の演算も可能なもの，さらには複数の整数や浮動小数点の値を同時に演算できる高性能なものまでさまざまです．

(a) 整数演算

CPUは，最低でも論理演算とシフト/ローテート，整数の加減算が行える演算器をもっています．最近のマイクロプロセッサでは，整数の加減算のほかに，乗除算も行えるものがほとんどです(**図9**).

論理演算として，論理否定(NOT)，論理積(AND)，論理和(OR)，排他的論理和(XOR)が行えます．シフトではビットの左右への移動，ローテートではビットの左右への回転が行えます．

整数演算では，2進数で表される値による演算を基本としています．演算ビット数は，CPUの処理ビット数に合わせ，その単位で演算されます．

たとえば，8ビットCPUなら8ビット演算，16ビットCPUなら16ビット演算，32ビットCPUなら32ビット演算というようになります．

(b) 浮動小数点演算

CPUによっては，浮動小数点演算をサポートしているものもあります．浮動小数点演算をサポートしているCPUでは，一昔前のマイクロプロセッサの場合，浮動小数点演算はコプロセッサと呼ばれる別のICチップをCPUに接続することで処理していました．しかし，現在のマイクロプロセッサは，この浮動小数点演算を行うユニットをCPUと同じチップに内蔵しているものがほとんどです．

浮動小数点演算では，IEEE規格754で規定されている2進浮動小数点形式のデータを演算対象としています．

浮動小数点演算といっても，CPUによってサポー

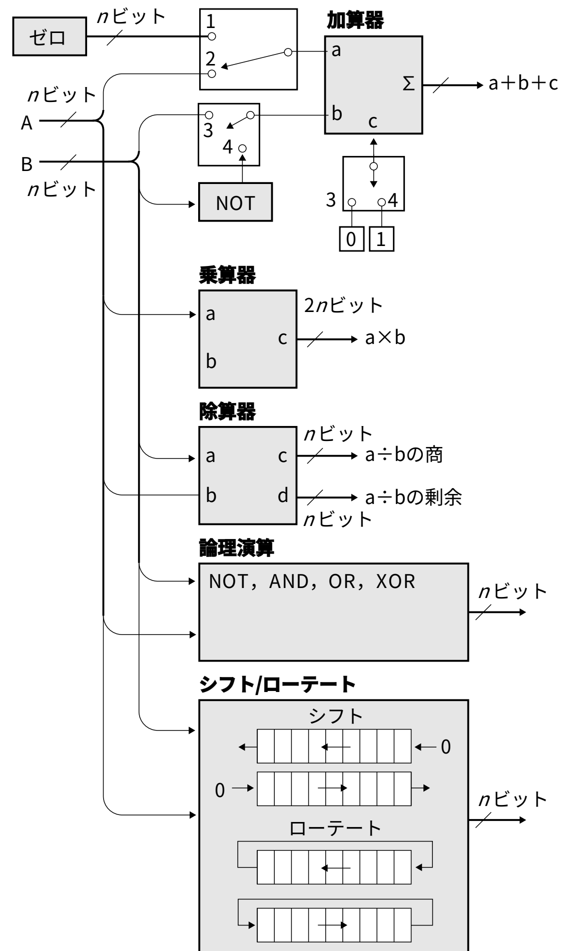


図9　整数演算

トしている演算に違いがあります．4バイト長の単精度の四則演算のみのものから，8バイト長の倍精度，あるいは10バイト長の拡張精度の四則演算が行えるものまであります．また，平方根や三角指数関数の演算機能をもったCPUもあります(**図10**)．

(c) 並列演算

CPUによっては，複数の2進整数や2進浮動小数点の値を，一つの命令で同時に演算する機能をもっています(**図11**)．

たとえば，PentiumがもつSIMD命令がそれです．SIMDとは“Single Instruction Multiple Data”の略です．PentiumのSIMD命令では，整数演算のMMXと浮動小数点演算のSSEに分けられます．

MMXでは8ビット整数なら8個，16ビット整数なら4個，32ビット整数なら2個のデータを，一つの命令で同時に演算する機能があります．SSEでは，4バイト(32ビット)長の単精度浮動小数点を4個，一つの命令で同時に演算することが可能です．

MMX，SSEとも，四則演算のほかに，論理演算も行うことができます．

従来，複数のデータに対し，同じ演算を行う場合，データ一つ一つに対して同じ演算を繰り返して行っていました．しかし，CPUにこの並列演算機能があれば，複数のデータに対して，一回の演算で複数のデータの処理が済むことになり，演算時間の短縮を図ることができます(**図12**)．

④ 機械語命令の実行

コンピュータを構成するレジスタや演算器，メモリやI/Oが単体で存在しているだけでは，何もできません．コンピュータは，プログラムがあってこそ動作し，存在する意義があります．このプログラムを実行するためのユニットがあってこそ，コンピュータといえます．

前にも述べたように，コンピュータが理解できるのは機械語のみです．その機械語をメモリから読み込み，解析し，データの流れや演算の指示を出すことで，コンピュータは機械語プログラムを処理していくのです．

(a) プログラム・カウンタ

メモリ上にあるCPUが次に実行する機械語命令のアドレスを指し示しているのが，プログラム・カウンタ(program counter)，あるいは命令カウンタ(instruction counter)と呼ばれるレジスタです．

プログラム・カウンタは，CPUリセットによって特定の値に初期化されます．つまり，CPUリセット

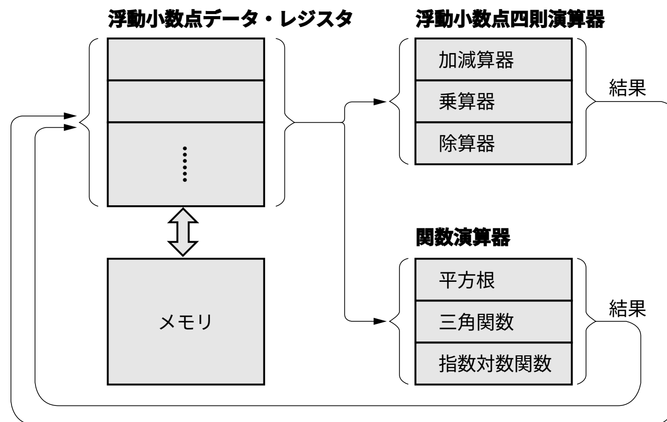


図10
浮動小数点演算

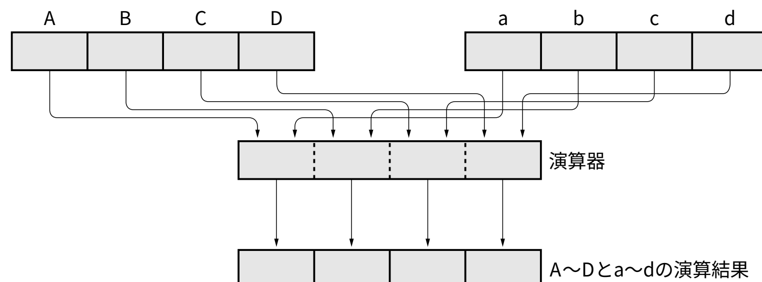


図11
並列演算

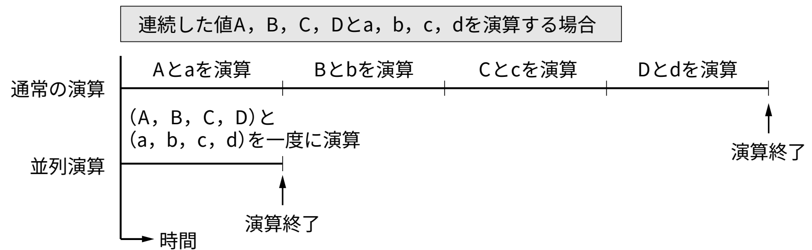


図12
並列演算のメリット

によってプログラムを開始するアドレスというのは，CPUによって事前に決められているのです．

プログラム・カウンタは，一つの機械語命令を実行すると，その機械語命令のサイズ分だけ，値がプラスされます．ただし，分岐に関係する機械語命令を実行した場合のみ，その分岐命令で指定されたアドレスに書き換えられます(**図13**)．

(b) 機械語命令の構成

CPUが実行する機械語命令は，CPUによってそのビット構成や意味が異なります．しかし，CPUに共通する機械語命令の構成というものはあります．

その構成は，先頭から「オペレーション・コード」，「アドレッシング・モード」，「アドレス値」，「イミディエイト値」となっています(**図14**)．機械語命令にとって，オペレーション・コードは必須項目です．後のアドレッシング・モード，アドレス値，イミディエイト値は，オペレーション・コードの解析の結果，必要な場合のみ存在する項目です．

●オペレーション・コード(OPコード)

機械語命令の先頭には，かならずこのオペレーション・コードのフィールドがあります．オペレーション・コードではCPUの大まかな動作が指定されます．オペレーション・コードは，略されて「オペコード」とか「OPコード」とも呼ばれます．

たとえば，CPUはこの命令が転送なのか演算なのか，それとも分岐なのか，CPUの制御なのかといったことをオペレーション・コードによって判断します．

また，オペレーション・コードにより，これから処理するデータの型やサイズといったものが決定される場合もあります．

レジスタやメモリのアクセスがない，あるいはオペレーション・コードのみでアクセス対象のレジスタやメモリが特定される場合は，機械語命令はオペレーション・コードのみとなります．

●アドレッシング・モード

オペレーション・コードで，アクセス対象があると判断される場合は，オペレーション・コードの次に「アドレッシング・モード」を指定するフィールドがあります．アドレッシング・モードによってアクセス対象のレジスタやメモリのアクセス方法が指定されます．

アクセス対象が一つの場合，アドレッシング・モードでは，アクセス対象のレジスタあるいはメモリのアクセス方法が指定されます．

アクセス対象が二つの場合は，一つがアクセス対象をレジスタのみに限定し，もう一つでアクセス対象のレジスタあるいはメモリのアクセス方法を指定します．

●アドレス値

アドレス値は，CPUによってはディスプレースメントと呼ばれる場合もあります．アドレッシング・モードでメモリ・アクセスであると判断され，さらにアドレス値が必要な場合にのみ，このアドレス値が存在します．

アドレス値は，直接アドレッシングでは，値そのものがメモリのアドレスとなります．間接アドレッシングでは，ポインタのレジスタに加算された結果がメモリのアドレスとなります．

●イミディエイト値

オペレーション・コードの解析の結果，定数値が必要と判断された場合に，このイミディエイト値が付きます．イミディエイト値は，アドレッシング・モードで指定されたレジスタやメモリに，定数として設定されます．

(c) 命令デコーダと機械語命令の実行

機械語命令を実行する場合，CPUはまずプログラム・カウンタが示すメモリから機械語を読み込みます．機械語をリードした後，プログラム・カウンタは今リードした機械語のサイズ分をプラスされた値に更新します．

リードされる機械語は，命令デコーダ(instruction decoder)に渡され，解析が行われ，データの種類を特定し，動作を決定します．また，解析の結果，機械

メモリ

機械語プログラム

サイズ m

プログラム・カウンタ

x

プログラム・カウンタには次に実行する機械語命令のアドレスが入っている

① 命令実行前

メモリ

機械語プログラム

今実行した命令

次に実行する命令

プログラム・カウンタ

$x+m$

機械語命令を実行すると，プログラム・カウンタは次に実行する命令を指すように更新される

② 命令実行後（分岐命令以外）

メモリ

機械語プログラム

今実行した命令

y

次に実行する命令

プログラム・カウンタ

y

分岐命令を実行すると，プログラム・カウンタは分岐先のアドレスに書き換えられる

③ 命令実行後（分岐命令の場合）

図13 プログラム・カウンタの動作

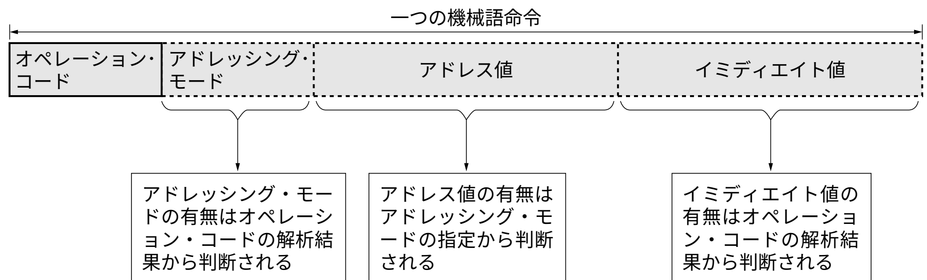


図14 機械語命令の構成

語の続きがあると判断された場合は，さらにプログラム・カウンタが示すメモリから機械語の残りを読み込むことになります．

実際の動作を指示するのは，コントロール・ユニットと呼ばれる部分です．コントロール・ユニットでは，機械語から抽出したオペレーション・コードから実行動作を決定し，アドレッシング・モードに従ってアクセスするレジスタとメモリのアドレスを計算します．そして，最終的にレジスタあるいはメモリをアクセスし，転送，演算といった処理を行います(**図15**)．

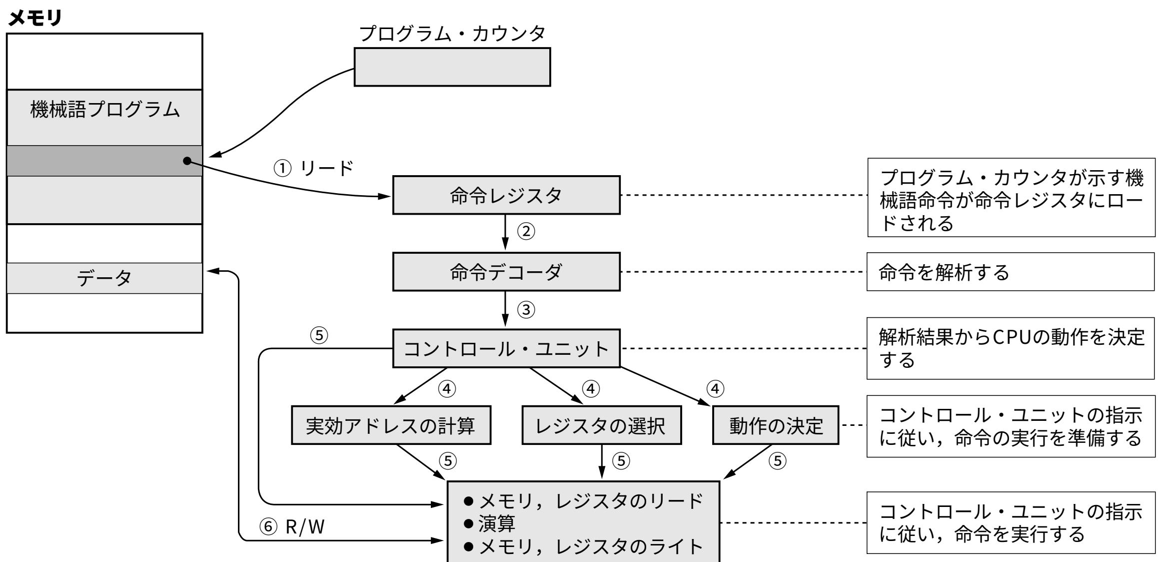


図15　機械語命令の実行

(d) CISCとRISC

現在使用されているCPUをおおまかに分類すると，CISCとRISCの二つがあります．

CISCは，“Complex Instruction Set Computer”の略で，CISCは扱えるデータの種類やアドレッシングも多く，転送や演算の種類が豊富であることが特徴です．そのため，複雑な処理であっても少ない機械語命令で処理できるというメリットがあります．

その一方でCISCは機械語命令の数が多く，一つの機械語命令でも1バイトのものから数バイトに及ぶものまであります．また，CISCの命令デコーダは複雑で，機械語命令の解析に時間がかかり，それがCPU全体の処理を高速化できない足かせになっています．

RISCは，“Reduced Instruction Set Computer”の略で，CPUで使われる機械語命令を極力少なくするように設計されています．また，RISCでは一つの機械語命令が16ビットや32ビットの固定長となっています．

そのため，RISCではCISCに比べて扱えるデータの種類も少なく，アドレッシングも多くありません．CISCでは1命令で実行できる処理も，RISCでは数命令が必要になる場合があります．

しかし，RISCは機械語命令の数を少なくし，機械語命令自体を固定長とすることによって命令デコーダを簡素化することができました．そしてその分，機械語命令の実行速度を上げることができるという大きなメリットがあります．

そのため，同じクロック周波数なら，CISCが1命令を実行する時間で，RISCは数命令実行できることになり，処理によってはCISCよりRISCで実行したほうが速いという場合もあります．

過去のマイクロプロセッサは，すべてCISCで作られていました．しかし，CPUの機能を上げるには，クロック周波数を上げることがもっとも効果的なので，新しく開発されるマイクロプロセッサは，クロック周波数を上げるのに有利なRISCタイプのものがほとんどになってしまいました．

たとえば，ARMやPowerPC，MIPS，SHシリーズといった最近のマイクロプロセッサは，すべてRISCとなっています．ところが，現在もっともよく使用されているPentiumなどのx86系のCPUは，依然としてCISCタイプのマイクロプロセッサです．これは昔の8086 CPUから続くプログラムの互換性を維持するという目的から，PentiumはCISCになっているといえます．とはいえ，Pentium系のCPUの命令体系はCISCのままですが，技術的にはRISCの要素を取り入れて，クロック周波数をどんどん高くしているのが現状です．

⑤ メモリ

x86系CPUに限らず，多くのマイクロプロセッサのCPUは，8ビットを1バイトとした単位でアドレスを振っています．そのため，8ビットより大きなデータは，複数のバイトで構成されることになります．つまり，8ビットより大きなデータは，つねに8の倍数