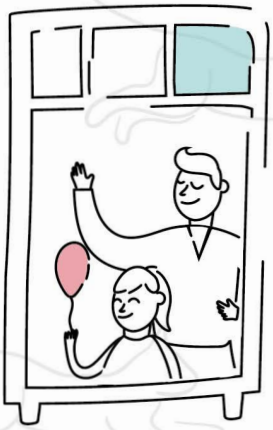
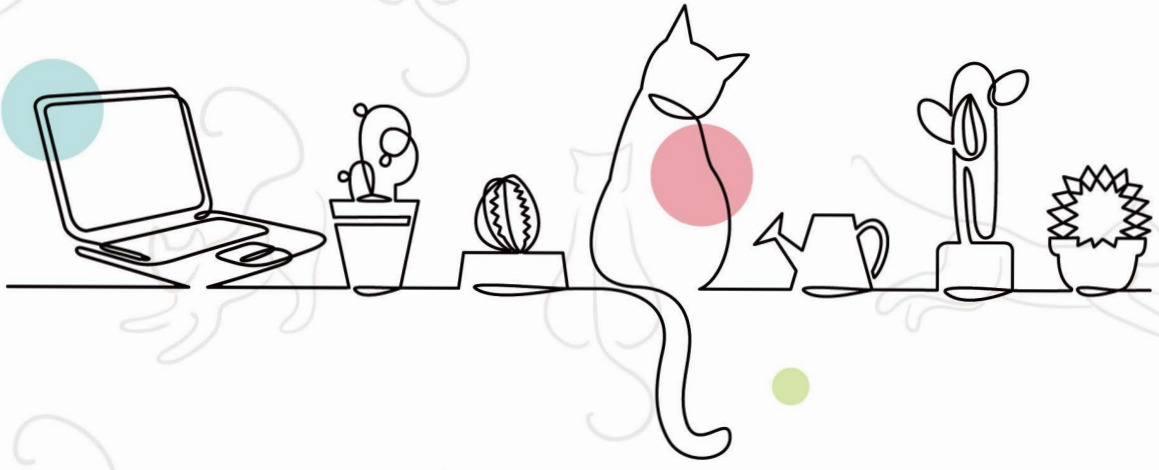
# もっとCPUの気持ちが知りたいですか？

著書　出村成和





PEAKS

# もっとCPUの気持ちが知りたいですか？

著書　出村成和

PEAKS

# はじめに

本書は、CPU の気持ちを理解するための本です。CPU の気持ちを理解するといっても、最近の半導体技術や AI の発展により CPU が自我を持つようになったため、CPU が持つ気持ちを理解するのが重要になってきた、ということではありません。

CPU の気持ちとはいうまでもなく比喩であり、「CPU の仕組みや、行われている処理をちゃんと理解してあげましょうね」という意味が込められています。

ソフトウェア開発者の方であっても、CPU の役割はなんとなく理解しているが具体的な仕組みを全然知らない、考えたこともなかった、といった方もいらっしゃることでしょう。それでもコンピューターを利用する上で困ることはありませんし、Web アプリケーション開発などで困ることはほぼないでしょうしね。

ただですね、どんなプログラミング言語で開発した場合であってもソフトウェアを実行しているのは CPU です。これは紛れもない事実なのです。

ですので、ある程度は CPU の気持ちを理解し、CPU の合わせて動作環境を揃えたり、ソースコードを記述したりするとよいでしょう。これにより、CPU も処理を実行する際には、それら開発者の願いや想いに応えてくれることでしょう。

それに、CPU の気持ちを理解することで得られるメリットは他にもあります。それは CPU により近いソフトウェアを開発できるようになったり、CPU に近いトラブルも解決できるようになったりする、という点です。

具体的には「OS などの低レイヤー寄りのソフトウェア開発ができるようになる」「実行中のソフトウェアが強制終了した原因を調査する道筋がみえるようになる」といった感じです。

CPU の気持ちをまったく理解していなかった頃には手が出せなかったさまざまなことについて、CPU の気持ちを理解することで手が出せるようになるのです。

CPU と聞くと何か凄く難しいことをしているイメージを持っているかもしれません。CPU の中で動いているコードは、現代のプログラミング言語と比較してみるとずっとシンプルな構文です。シンプルである分、いろいろと奥が深いです。

世界にはいろいろな会社から CPU がリリースされてます。ただ、それらの CPU にそう大きな違いはありません。

そのため、本書で CPU の仕組みや動作の基礎を理解し、その後に Intel などそれぞれの CPU を理解することで効率よく習得できるのでは、と考えています。

では本書で、CPU の気持ちを習得する第一歩を踏みしめていきましょう。

## 対象読者

本書では次のような読者を想定しています。

- CPU の内部で行われていること（主にソフトウェア方面）を理解したい人
- CPU の挙動を理解した上でコードを書きたい人

## 書かれていること

- CPU の気持ちを理解するのに必要なあれこれ

## 書かれてないこと

- マルチコアに関する解説
- 特定 CPU アーキテクチャーのアセンブリ言語の解説
- CPU の回路設計などのハードウェア関連

## 想定環境

本書では表 1 の環境で動作確認をしています。

●表 1　想定環境

| | |
|---|---|
| MacBook Air | M1, 2020 |
| macOS | Monterey |

本書で利用している C コンパイラーは macOS Monterey 標準のものを利用しています。

```
$ clang -v
Apple clang version 12.0.5 (clang-1205.0.22.9)
Target: arm64-apple-darwin21.5.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
```

また、本書で登場するアセンブリ言語のコードは断りがない限りは、MacBook Air（M1, 2020）

にて生成されたコードとなっています。

## クラウドファンディングと PEAKS

本書は技術書クラウドファンディング・サービスである「PEAKS」のプロジェクトとして開始され、680 人もの支援者のサポートによって作られました。出資者特典である「アーリーアクセス」でいただいたご意見も反映されております。

PEAKS ではこんな本を作りたい！という方を募集しています。次の窓口からご連絡いただければ幸いです。

- https://peaks.cc/requests

## 免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、著者はいかなる責任も負いません。

# 目次

第1章

# CPUの気持ちを知るということ

本書はCPUの気持ちについて解説している本です。 CPUの気持ちを解説する前に、まずCPUの気持ちを知ると、どんなよいことがあるのかを先に解説します。この本を読みすすめるために少しモチベーションが上がるといいな、と思っています。

前提条件として、本書を読んでいる人はFlutterといったクロスプラットフォームに対応したフレームワークの利用経験がある人を想定して進めていきます。

## 1.1 時代はクロスプラットフォーム！

時代はクロスプラットフォームなんですよ。IntelやArmといったCPUベンダーなんて関係ない、OSの違いも関係ない、動作対象のデバイス（たとえばスマフォやパソコンなど）も関係ない、1度ソフトウェアを書いてしまえばどんな環境でも動くソフトウェア開発環境があればよいのです[注1)]。

そんな時代なので、今時のCPUのことを勉強するのは時間のムダなんじゃね？と考えてしまうのは、まぁ当然かなと思います。

古今東西、さまざまなクロスプラットフォーム対応フレームワークがリリースされては役目を終えて終了するという流れが続いているのが事実です。どの時代においても何かしらのクロスプラットフォームが存在しています。

## 1.2 単一プラットフォームの夢をみるか？

とはいえですよ、動作する機器、スマートフォンやPCなど、OSの違い、CPUの違いは間違いなく存在しているのですよ。

ということは、あなたが開発で使用しているクロスプラットフォーム対応フレームワークがOSやCPUを問わず使えるのは、どこかのだれかが頑張ってOSの違いやCPUの違いを意識せずに開発できるよう環境を整えてくれているからであり、これはまぎれもない事実なのです。

という話をすると「すべてを統べるただ1つのOS、ただ1つのCPUがあればいいんだー」み

注1)　そういえば、Write Once, Run Anywhere という標語のプログラミング言語があったなー、なんて個人的には思ったりします。

たいな方向に話が進みそうですが、それは本書の趣旨とは外れていくので、その話はいったん横に置いておきます。

それはさておき、クロスプラットフォーム対応フレームワークを利用してアプリケーションを開発している人は、裏で頑張ってくれている誰ががいることをゆめゆめ忘れないようにしましょう。そして、その誰かがあなたになる可能性がないわけではないのです。

### 1.2.1 クロスプラットフォームの限界

クロスプラットフォーム対応フレームワークを使ってアプリケーションを開発するのだからCPU や OS の違いを理解しなくてよいといった風に考えてしまうかもしれませんが、世の中そんな完璧なソリューションは存在しません。というのも、多かれ少なかれそれぞれのプラットフォームに依存する実装が必要な箇所は存在します。

クロスプラットフォーム対応フレームワークを利用したアプリケーション開発においては、プラットフォームに依存しないソースコードは 8〜9 割で、残り 1〜2 割が OS やプラットフォーム依存の実装が必要という肌感覚があります。

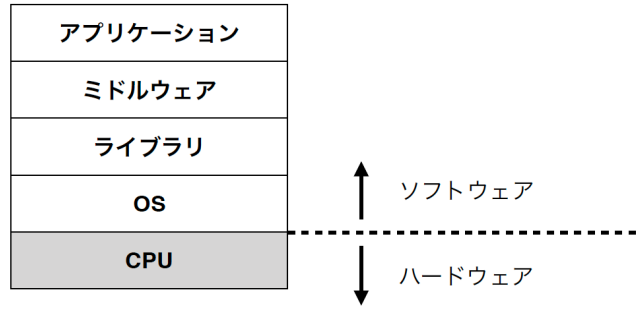
で、この 1〜2 割が開発において厄介な箇所だったりします。「泥臭い作業となりがち」「だれもやりたがらない」「できる人も少ない」「この 1〜2 割を実装しないと、そもそもアプリがリリースできない」といった感じです。

それらの泥臭いところを実装しようとすると、幅広い知識が要求されます。プラットフォーム、OS、CPU…。少なくとも、そのクロスプラットフォームで採用されている開発言語やミドルウェア以外のあらゆる知識、もっといえば OS などの低レイヤーの知識が要求されます。そして、これまで挙げてきたさまざまな知識の深さ（ソフトウェアレイヤーとして）で、その実装ができるか否かを決めるのではないかと考えています。

## 1.3 よくあるソフトウェアレイヤーの構成

スマートフォンの内部で動作しているソフトウェアは、ミルフィーユのように役割別の階層構造で構成されているのが一般的です（図 1.1）。図の上の方がユーザの目に触れやすいレイヤー、下の方がユーザの目に触れにくい、つまりハードウェアに近いレイヤーを表しています。

この図で言いたいのは「上位レイヤーのソフトウェアは、下位レイヤーのソフトウェアがあるからこそ動作している」ということです。




●図 1.1　よくあるソフトウェアレイヤー

ユーザが触れるアプリケーションは図 1.1 をみても分かるとおり、最上位に位置しています。そして、本書で取り上げている CPU は OS のさらに下のレイヤーに位置しています。いうまでもなく CPU はソフトウェアではなくハードウェアですからね。

ということで、開発しているアプリケーションと本書にて学ぶ CPU はソフトウェアアーキテクチャーのレイヤーとしては、もっとも遠い場所に位置しています。なので、普段からソフトウェア開発している方であっても CPU に馴染みがないといわれても何ら不思議はないですね。

## 1.4　CPU の知識＝筋肉説

CPU の知識＝筋肉説という話は聞いたことがあるでしょうか？　恐らくないでしょう。私が言い出した理論なので。

個人的には CPU の知識って割と筋肉に近い気がするんですよね。アセンブリ言語（CPU が実行しているコード）が読めれば、技術の習得や内容の調査など、技術でなんとかできる幅が広がります。

実際のソフトウェア開発において直接役に立つことがなくても「私には CPU の知識があるんだ」「いざとなれば、CPU の知識も使って調査していけば、解決できない問題はない」みたいな感じで心の安寧というか安心感というか、そういったモノが得られるのではないかなと思うのです。

さすがに CPU の知識だけで、ありとあらゆるすべての問題が解決できるわけではありません。しかし、CPU の知識がソフトウェアレイヤーとしては最下層となるので、それより上位のレイヤーはソフトウェアの話となり、CPU よりは馴染みがあるため習得しやすいかなと考えています。

## 1.5 CPU を知ることで得られるメリット

「CPU の知識は筋肉同様にあった方がよいから覚えろ！」といわれて「はい、わかりました」と答える人は多くないでしょう。

ということで、もう少し具体的に CPU の知識を得ることのメリットについて説明してみます。もっと分かりやすいメリットがないと学習するモチベーション湧かないですよねぇ。

メリットとしては次の 2 つがあるかな、と考えています。

- 開発できるソフトウェアの種類の幅が広がる
- 開発で考えられる範囲が深くなる

では、それぞれについてみていきます。

### 1.5.1 開発できるソフトウェアの幅が広がる

CPU をはじめとするハードウェアについて、詳しく知らないと開発ができないソフトウェアは存在します。その最たるモノが OS、コンパイラーなどのハードウェアに近いソフトウェアです[注2)]。

これは言い方を変えると、CPU の知識や C 言語の知識があると、より幅広いジャンルのソフトウェアが開発できるということを意味しています。たとえば、Python といったスクリプト言語で開発したソフトウェアは Mac などのパソコン、もしくは Web サーバといった OS が搭載されていることが前提のコンピューターで動作させるのが一般的でしょう。

それらに加えて CPU の知識や C 言語の知識があると、Mac などのパソコン、各種サーバ上で動作するソフトウェアの他に、ワンボードマイコン（小さなコンピューター）といった安価なコンピューターで動作するソフトウェアも開発できるようになります[注3)]。

たとえば M5 Stack を取り上げてみます。これは液晶ディスプレイやボタンなどが付いた小さなマイコンです。このマイコンを扱うためのプログラミング言語としては MicroPython と C/C++ から選択できます。ちなみに、このマイコンボードには OS は搭載されていませんので、ソースコードから直接ハードウェアを制御します。

それらに向けたソフトウェアや接続された周辺機器も含めて制御するソフトウェア開発する場合、CPU やその周辺のハードウェアの知識が必要となります。開発用ライブラリなども用意されていますので、一見するとそれらを利用してソフトウェア開発していけばなんとかなるのでは？と思うかもしれません。ただ、そのようなハードウェア特有のあれこれや、接続された周辺機器に対応したコードを書く必要があるので、最終的にはハードウェアの知識が必要となる場合が大半です。

---

注2)　低レイヤーと呼ばれることが多いですね。

注3)　最近では、ワンボードマイコンでも MicroPython といったスクリプト言語も動作しますね。

このように CPU を含めたハードウェアの知識があることで、パソコンのみではなくマイコンボードのような OS を搭載していないハードウェアや、小さなハードウェアを制御するソフトウェアを開発できるようになります。

プログラミングできる対象が広がると、ソフトウェアエンジニアとしての経験値がたまりやすくなるのはないでしょうか。

### 1.5.2 開発で考えられる深みが増す

CPU というのは先ほども解説したとおり、ユーザにもっとも近いハードウェアといえます。

アプリケーションなどが動作しているのは、ソフトウェアレイヤーでいうと最上位に位置します。

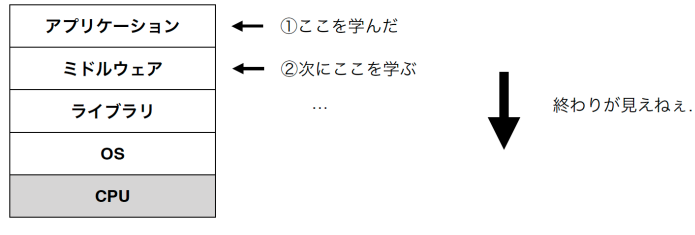
上位レイヤーのソフトウェアは、ハードウェアからもっとも離れているソフトウェアですので、ハードウェアはほぼ意識せずにアプリケーションが開発できます。参照する資料にしても、プログラミング言語の言語仕様であったり、ミドルウェアのドキュメントであったり、CPU などのハードウェアとはほぼ無縁の情報が多いことでしょう。

これら上位ソフトウェアレイヤーでの開発に関する知識は、アプリケーション開発で幾度となく活用するものですから、ある程度のソフトウェア開発経験がある方は設計、開発、デバッグなどの開発の基本は身についていることでしょう。

では、もうすこし深い所を学んでいきたいとします。その際の進め方はつぎの 2 つがあるのではないかと考えます。

1. ミドルウェア、ライブラリといった順に徐々にソフトウェアレイヤーの下に進んでいく
2. ハードウェア（つまり CPU）を学んだ後にソフトウェアレイヤーを上に進んでいく

ぱっと思いつくのが 1. の流れかと思います。割とスムーズに行くような気になりますが、ある意味、学んで行く上で終わりが見えず、またハードウエアの知識が徐々に要求されて割とツラい思いをするのではないかなーと考えてしまいます（図 1.2）。




●図 1.2　最上位レイヤーから下へ向かうと終わりが見えないので心折れそうになる

第 2 章

# CPU と友達になろう

CPU の気持ちを把握するためにも、まず CPU がどのような働きをしているのかを知っておきましょう。CPU がいつ、どんな時に、どのように働いているかを具体的に知らないことには、CPU の気持ちを理解することは到底不可能です。

本章では CPU を身近に感じられるよう、CPU とかなり親密な関係のプログラミング言語である C 言語入門の冒頭に書かれているようなソースコードから、実行ファイルのビルドまでを解説します。このあたりをすでに知っている方は本章を読み飛ばしちゃってください。

## 2.1 最初はお友達から

CPU の気持ちを知ろう！という内容の本です。いきなり気持ちを知ろうとするのは、クラス替えでたまたま席が隣になったあの娘へ「君のこと、もっと教えて！」というくらいには無謀かと思います。

そういう正面突破は、クラス中の女子から相手にされなくなってしまうなどのトラウマの元となったりするので、最初はお友達から始めるのがセオリーでしょうね。ということで、気持ちを知る前に CPU と親しくなるところからはじめていきます。やはり最初はお友達からね。

ではさっそく、CPU とお友達となるために、C 言語のソースコードから実行ファイルをビルドするところからはじめていきます。やはりね、CPU と C 言語との相性の良さは誰もが認めるところでしょうし。そして、その実行ファイルを実行して、最後に実行ファイルの仕組みをみていきます。外堀から埋める感じですね。そこからジワジワと弱みを握る…、ではなく、深層に迫っていきます。

●図 2.1　突然「君のこと教えて！」なんていわれたら、不審者認定間違いなしでしょうね

## 2.2　実行ファイルを作って実行してみよう

最初に C 言語のソースコードを元にビルド、ファイルの実行までの流れを説明します。さみしいことですが最近は C 言語を知っている人も減ってきたようですし。こんなの知ってるわ！という人は、本章を読み飛ばしてもらっても OK です。

早速 Hello, World を表示して…、ではなく、簡単な加算処理を書いてみます（リスト 2.1）。何かしらの演算処理が含まれていた方が説明しやすいのでね。C 言語に馴染みのない人は、#include って何やねんと思うでしょうが、そこは黙って見逃してください。

●リスト 2.1　加算処理を行う　calc1.c

```
#include <stdio.h>

int main(int argc, char** argv)
{
    int a = 1;
    int b = 2;
    int c = a + b;
    printf("c = %d\n",c);
    return 0;
}
```

リスト 2.1 をビルドし、実行ファイルを実行してみましょう（リスト 2.2）。C コンパイラーとし

て、今回は macOS の標準コンパイラーである clang を利用しています。インストールしていない場合はあらかじめインストールしておいてください[注1)]。

実行ファイルの実行結果については、実行する前から結果が分かっているので、実はそんなに重要じゃないです。実行するまでのプロセスが大事なのです。

**●リスト 2.2　calc1.c をビルド、実行してみた**

```
$ clang calc1.c -o calc1
$ ./calc1
c = 3
```

C 言語のソースコードから生成された実行ファイルには CPU で直接実行できるコードが含まれています。そして、CPU はそのコードを実行する、という仕組みとなっています。

Linux のパッケージ等で配付されている実行ファイルの中には、実は Python のコードで動いています、みたいなものもあります。そして、C 言語のソースコードをビルドした場合は、中身は CPU が直接実行できるコードに変換されます。間違っても、C 言語のコードがそのまま実行ファイルに含まれていることはありません。

## 2.3　実行ファイルの中身

そんな実行ファイルに何が含まれているのかをみていきます。気になるあの娘（といっても CPU ですが）が何を考えているのか、その一端を実行ファイルの内容から垣間見ることができます。

実行ファイルの内容を調査する手法はいくつかありますが、ここでは objdump コマンドを使って進めていきます。

ちなみに本書で重要視するのは、「実行ファイルの中身は何なのか？　どのように処理を行い、演算結果を画面へ出力しているのか？」という点です。

ということで、objdump コマンドを使って実行ファイルの中身を覗いてみました（リスト 2.3）。objdump コマンドは、実行ファイルなどの C コンパイラーが生成したバイナリファイルの内容を分かりやすく出力してくれるツールです。

今回は、M1 MacBook Air でビルドした実行ファイルの中身をみています。

**●リスト 2.3　objdump コマンドで実行ファイルの内容を出力した**

```
$ objdump -d ./calc1

./calc1:     file format mach-o-arm64  ←Mach-O arm64向けの実行ファイルである
```

注1)　"Mac Xcode インストール"などのキーワードでググるなりしてください。

```
Disassembly of section .text:

0000000100003f10 <_main>:              ←Mach-O arm64向けの実行ファイルである
   100003f10:    d10103ff     sub     sp, sp, #0x40
   100003f14:    a9037bfd     stp     x29, x30, [sp, #48]
   100003f18:    9100c3fd     add     x29, sp, #0x30
   100003f1c:    52800008     mov     w8, #0x0                        // #0
   100003f20:    b81fc3bf     stur    wzr, [x29, #-4]
   100003f24:    b81f83a0     stur    w0, [x29, #-8]
   100003f28:    f81f03a1     stur    x1, [x29, #-16]
   100003f2c:    52800029     mov     w9, #0x1                        // #1
   100003f30:    781ee3a9     sturh   w9, [x29, #-18]
   100003f34:    52800049     mov     w9, #0x2                        // #2
   100003f38:    781ec3a9     sturh   w9, [x29, #-20]
   100003f3c:    78dee3a9     ldursh  w9, [x29, #-18]
   100003f40:    78dec3aa     ldursh  w10, [x29, #-20]
   100003f44:    0b0a0129     add     w9, w9, w10
   100003f48:    781ea3a9     sturh   w9, [x29, #-22]
   100003f4c:    78dea3a9     ldursh  w9, [x29, #-22]
   100003f50:    90000000     adrp    x0, 100003000 <__mh_execute_header+0x3000>
   100003f54:    913ec000     add     x0, x0, #0xfb0
   100003f58:    910003eb     mov     x11, sp
   100003f5c:    aa0903e1     mov     x1, x9
   100003f60:    f9000161     str     x1, [x11]
   100003f64:    b90017e8     str     w8, [sp, #20]
   100003f68:    94000006     bl      100003f80 <_main+0x70>
   100003f6c:    b94017e8     ldr     w8, [sp, #20]
   100003f70:    aa0803e0     mov     x0, x8
   100003f74:    a9437bfd     ldp     x29, x30, [sp, #48]
   100003f78:    910103ff     add     sp, sp, #0x40
   100003f7c:    d65f03c0     ret

 ...  (以下略)  ...
```

出力結果をみると、実行ファイルに含まれている情報は大きく分けて次の 2 つであることが分かります。ちなみに、この構成はどの OS の実行ファイルでも大まかには同じです。

1. ヘッダー情報（動作環境 OS、CPU アーキテクチャー[注2)]など）
2. 実行プログラム[注3)]

では、それぞれについてみていきましょう。

注2)　CPU 設計の方針のこと。2022 年では Intel と Arm が主流。
注3)　OS によって呼び方が異なるので、本書では「実行プログラム」と呼ぶことにします。

第 3 章

# アセンブリ言語をなんとなく読む

本章では、第 2 章に登場したアセンブリ言語のコードがだいたい理解できるようになるためのあれこれを解説します。

アセンブリ言語は、どの CPU アーキテクチャーでも、機能的に大きな違いはなく似たり寄ったりの構成となっています。ですので、命令の読み方や規則性を 1 度理解してしまえば、CPU アーキテクチャーに関係なく、なんとなく読めるようになります。

ちなみに本書では完璧な理解は目指していません。

これは英語の長文問題を読んでいるときの、その中に知らない単語が出てきたとしても、文章を通して読んでいけば何が書いてある文章なのかなんとなくは理解できるよね、といったよくある試験テクニックの応用なので。

## 3.1 コンパイラーからアセンブリ言語を直接出力する

第 2 章「CPU と友達になろう」では、objdump コマンドを利用して実行ファイルを逆アセンブルしてアセンブリ言語のコードを生成しました。

C 言語のソースコードが手元にある場合はコンパイラーを通してアセンブリ言語のコードが直接出力できます。まぁ、最初からそうしろよ、という話でもありますが。

では、実際にやってみましょう。第 2 章で登場したソースコードに再登場してもらいます（リスト 3.1）。

●リスト 3.1　calc1.c　再び

```
#include <stdio.h>

int main(int argc, char** argv)
{
    int a = 1;
    int b = 2;
    int c = a + b;
    printf("c = %d\n",c);
    return 0;
}
```

第 2 章では実行ファイルを生成しました。コンパイルして実行ファイルを生成するという流れが

一般的ですが、今回は実行ファイルを生成する元となったアセンブリ言語のコードをみてみたい！ということで、リスト 3.2 を実行します。-S オプションはアセンブリ言語で出力するためのオプションです。

●リスト 3.2　アセンブリ言語のコードを出力する

```
$ clang -S calc1.c
```

実行すると、calc1.s ファイルが生成されます。あ、アセンブリ言語のソースコードの拡張子は一般的に.s です。

ちなみに出力されたアセンブリ言語のコードは最適化[注1)]されていません。最適化を指定しない場合は、先の C 言語のソースコードの内容をそのままアセンブリ言語へ置き換えたコードとなり、コンパイラーが何をどこまで考えてアセンブリ言語へ落とし込んでいるのか分かりやすくなります。

## 3.2　アセンブリ言語を読みすすめていく

では、calc1.s ファイルのコードを読んでいきましょう（リスト 3.3）。

ここでは 1 つ 1 つの命令の詳細な意味や解説はさておき、全体をさっと眺めて、なんとなく内容を理解できることを目的とします。

どのように変換されたかわかるよう C 言語のソースコードをコメントとして追記してあります。コメントを元に、どのように C 言語のソースコードからアセンブリ言語へ変換されたか見えてくることでしょう。

●リスト 3.3　calc1.s ファイルを読み解く

```
    .section    __TEXT,__text,regular,pure_instructions
    .build_version macos, 11, 0    sdk_version 11, 1
    .globl    _main                    ; -- Begin function main ←(1)
    .p2align    2
_main:                                  ; @main
    .cfi_startproc
; %bb.0:
    sub    sp, sp, #64              ; =64
    stp    x29, x30, [sp, #48]      ; 16-byte Folded Spill
    add    x29, sp, #48             ; =48
    .cfi_def_cfa w29, 16
    .cfi_offset w30, -8
    .cfi_offset w29, -16
    mov    w8, #0
    stur   wzr, [x29, #-4]
    stur   w0, [x29, #-8]
```

注1)　ムダな処理を一切省いたアセンブリ言語のコードに変換すること。コンパイラーの仕事なので人間は気にしなくていい。

```
    stur   x1, [x29, #-16]
    ;  int a = 1;
    mov    w9, #1           ←(2) w9レジスターに1を代入
    stur   w9, [x29, #-20]  ←(2) w9レジスターの値を指定したメインメモリへ格納
    ;  int b = 2;
    mov    w9, #2           ←(3) w9レジスターに2を代入
    str    w9, [sp, #24]    ←(3) w9レジスターの値を指定したメインメモリへ格納
    ;  c = a + b
    ldur   w9, [x29, #-20]  ←(4) 変数aの値(=1)をw9レジスターにコピー
    ldr    w10, [sp, #24]   ←(4) 変数bの値(=2)を取り出す
    add    w9, w9, w10      ←(4) a + bの結果をw9レジスターへ格納
    str    w9, [sp, #20]
    ldr    w9, [sp, #20]
                                       ; implicit-def: $x1
    ;  printf("c = %d\n",c);
    mov    x1, x9
    adrp   x0, l_.str@PAGE
    add    x0, x0, l_.str@PAGEOFF
    mov    x11, sp
    str    x1, [x11]
    str    w8, [sp, #16]           ; 4-byte Folded Spill
    bl     _printf          ←(5) printf関数を呼び出し
    ldr    w8, [sp, #16]           ; 4-byte Folded Reload
    mov    x0, x8
    ldp    x29, x30, [sp, #48]     ; 16-byte Folded Reload
    add    sp, sp, #64             ; =64
    ret
    .cfi_endproc
                                       ; -- End function
    .section    __TEXT,__cstring,cstring_literals
l_.str:                                ; @.str
    .asciz "c = %d\n"

.subsections_via_symbols
```

ひととおり読めたでしょうか？ まぁ、このリストをいきなり見せられても「は？ 意味が分からん」となるのが正直なところでしょう。当然の反応かと思います。

ということで、このコードの読み解き方を解説します。ちなみにアセンブリ言語で覚える命令などは多くありません。現在のスクリプト言語の文法を覚えるよりは遙かに少ないです。

まずはフィーリングで読めるようになってください。それが先だと考えています。よく判らなくてもふんふんと読みすすめてください。この本を最後まで読んだ後に、もう一度このコードを読むと読めるようになっているはずです。

ですので、ここで読めないからといってあきらめることはありません。

### 3.2.1 アセンブリ言語の解説

さて、長い前振りはここまでにして、お待ちかね、アセンブリ言語を説明していきましょう。

第4章

# CPUをざっくり把握する

コンピューターを扱う世界で仕事をしていると、「プロセスを殺す」や「ポートを叩く」など何も知らない人からすると物騒ないいまわしがよく登場します。

このような会話は駅のホームなど不特定多数がいる環境では口に出さない方がよいでしょう。通報される可能性が十分ありますので。

## 4.1 CPUを構成する主要な回路

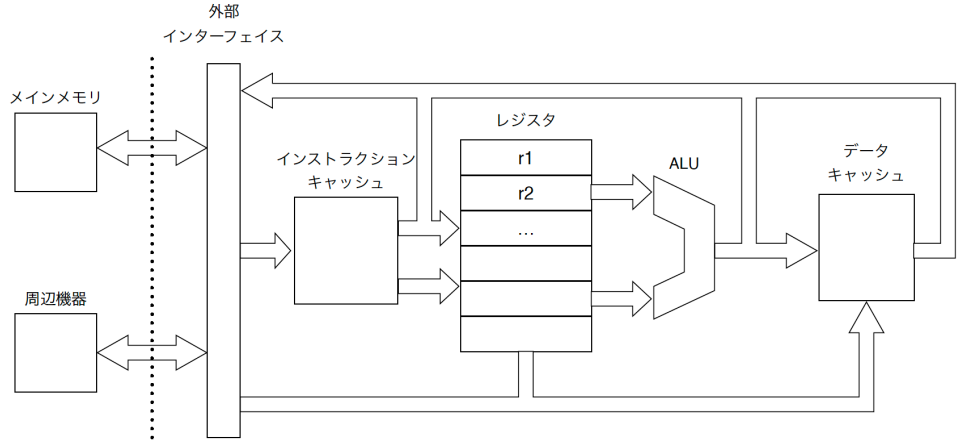
本章では、CPUの内部にあるさまざまな機能について個別に解説しています。と、その前に、本章で解説しているCPUの内部にある機能がどのように関連しているのかをあらかじめ解説します。

なんとなくでも、機械語のコードやデータがCPU内部において、どのような順序で処理が行われているのかを把握しておくと、本書のこの後の解説も理解しやすくなることでしょう。

ですので、CPUを構成する回路は「どんなものがあって」「どの回路同士が連係して動作しているのか」を解説します。CPUの気持ちを理解する上でおさえておくべき回路は、つぎの4つです。

- 外部インタフェース
- キャッシュメモリ
- レジスター
- ALU（Arithmetic and Logic Unit）

それ以外のモノとしては、「バス」「コントローラー」があります。機構的にも存在しており、どれもなくてはならないものですが、本章では存在意義を押さえておく程度で解説しておきます。




●図 4.1　CPU を構成している回路の関係性

### 4.1.1　外部インタフェース

CPU 内部と CPU の外側に接続されている周辺機器[注1)]との間を取り持つインタフェースです。家で例えるならば玄関ですね。普通の家であれば勝手口とか玄関はいくつかあるでしょうけど、CPU の場合は必ず玄関から家に入らないといけない、ということです。

隣に住んでいる幼なじみとは二階の自室の窓からよく行き来していた、といったようなことは許されていません（CPU に、そんな裏口的なものは存在しません）。幼なじみのようにどんなに親しくても、ちゃんと玄関から出入りしないといけないのです。

### 4.1.2　キャッシュメモリ

CPU の内部にあるメモリエリア　その 1 です。

キャッシュメモリには、直近で実行した、もしくはこのあと実行予定の命令やデータが記録されています。命令やデータが処理される際、必ずキャッシュメモリから取得します。実行結果もキャッシュメモリに反映されます。

キャッシュメモリは役割別に 2 つあり、1 つは命令をキャッシュするためのインストラクションキャッシュ、もう 1 つはデータをキャッシュするためのデータキャッシュです。

これらのキャッシュメモリは数十キロバイト〜数百キロバイトであり、その内容はつぎつぎと更新されます。1 メガバイトにも満たないメモリを、なぜわざわざ搭載して、そのメモリにある命令やデータを参照して実行しているのか？　その答えは第 11 章「手が届く範囲にモノがあると便利だよね（キャッシュメモリ）」で解説します。

---

注1)　メインメモリ、SSD 等のストレージ、GPU など CPU が制御したり、CPU 自身が必要としたりするあらゆる機器を指します

### 4.1.3 レジスター

CPU の内部にあるメモリエリア　その 2 です。レジスターは演算で参照する値や演算結果が格納されています。

レジスターは複数存在しているので、それらのレジスターたちを上手く取り回して演算しています。レジスターの詳細は第 5 章「値を扱う（レジスター）」にて解説します。

### 4.1.4 ALU（Arithmetic Logic Unit）

文字どおり四則演算と論理演算（AND、OR などですね）の処理を行う回路です。CPU のもっともコンピューターらしい処理を行う回路ですね。

ALU では 1 度に 1 つの演算のみを行います。複数の演算を効率的に ALU に送る処理はコントローラーの仕事となります。

### 4.1.5 バス（Bus）

それぞれの回路を結ぶ信号線です（⇒で示しているやつ）。CPU 全体を学校で例えるならば、それぞれの回路は教室、バスはさしずめ廊下といった感じでしょうか。

バスにはデータ転送速度の上限があります。そのため、大量のデータを転送しようとするとめちゃくちゃ時間がかかったりします。これは、学校の終業式のため全校生徒が一斉に体育館へ向かうと、体育館につながる渡り廊下がめっちゃくちゃ混んで進みが悪くなるのと同じですね。

### 4.1.6 コントローラー

ここでは一口にコントローラーとざっくり書いていますが、厳密には制御対象に応じて、さまざまなコントローラーが存在します（たとえば、キャッシュメモリを制御するキャッシュコントローラーなど）。が、コントールする対象は違っても、何かしらコントロールしているという役割は同じなので、ここでは一緒くたにコントローラーとして解説します。

コントローラーとは、各種回路やその間の信号線を流れるデータを、必要に応じてちゃんと準備して、いい感じに制御してくれるものです。先ほどあげた「外部インタフェース」「キャッシュメモリ」「レジスター」「ALU」の間を流れるデータは、このコントローラーによって制御されています。

## 4.2 処理の順序

ここからは全体的な流れを解説します。

CPU の全体の流れを把握した上で、それぞれの機能を理解したほうが理解しやすいでしょう。

第 5 章

# 値を扱う（レジスター）

レジスターでググってみると、小売店にある会計をするための機械が先頭に出てきます。確かにあれもレジスターですね。買い物で会計を済ませるときに、同伴者に「レジ行ってくる」とたまに言ったりします。この言い方を改めて考えると、USB メモリを指して「USB 持ってきて」と言っているのと変わらないのでは？という気がしてきました。

本章で説明するレジスターは、小売店にあるレジスターとはまったく関係ありません。

## 5.1 レジスターと変数の違い

ここでは、ここまでで何度か登場してきたレジスターをもっと詳しく解説します。レジスターの役割は、プログラミング言語によく登場する変数と同じです。値を一時的に保存しておく箱のような役割を果たしています。

●図 5.1　コンビニなどのレジスターとは関係ないです

ただし、レジスターと変数は別物です。もうちょっと具体的にいうと、変数はプログラミング言語上の概念であり実際の値の格納先はプログラミング言語やコンパイラーに依存します[注1)]。

レジスターは変数ではなく CPU の内部にある特殊な**メモリエリア**です。ですので、変数のような概念として存在しているモノではなく物理的に存在しています。メモリエリアですので、変数とは扱いが違うのだろうなぁ、ということはなんとなく想像がつくかと思います。

そのレジスターは、具体的に変数とはどのような点が異なるのでしょうか。主な違いは次の通りです。

- 名前が固定である
- レジスターの数は決まっている
- 用途が固定化されている

これらについて詳しくみていきましょう。

### 5.1.1 名前が固定である

レジスターは変数と違い、名前が自由に変更できず固定です。しかも名前は、CPU アーキテクチャーによって EAX、EBX、ECX…とか、r1、r2…といった具合に統一されてはいません。

変数のように名前が変更できないので、アセンブリ言語でプログラミングする際には「r1 レジスターには××の値が入っていて」と、開発者がレジスターに格納されている値の役割を認識しつつプログラミングしていく世界となります[注2)]。

### 5.1.2 レジスターの数は決まっている

レジスターの数自体は、Intel や Arm などの CPU アーキテクチャーによっても異なります。また同じ Arm アーキテクチャーでも、arm32（32 ビットアーキテクチャー）と arm64（64 ビットアーキテクチャー）で、これまたレジスターの表記や数が異なります。ちなみに arm64 での汎用レジスターの数は 31 個あります。

コンパイラー（もしくは人間）は、これらレジスターを駆使してさまざまな演算を行っています。処理内容によっては汎用レジスターの数がたりない場合もあるでしょう。その場合は、1 つの汎用レジスターの値をいったんメモリへ待避し、その汎用レジスターに値を代入して演算を行い、演算終了後にはメモリに待避した値を再度レジスターへ戻す、といった感じの処理を行ってレジスタ不足を回避しています。

---

注1)　変数の値はコンパイルすることで即値に変換されたり、メモリに格納されていたり…。つまり、変数ではなく別の方法で値が扱われているってことです。

注2)　大変なように思うかもしれませんが結局は慣れの問題です。

### 5.1.3 用途が固定化されている

レジスターは、演算に使用される複数の汎用レジスター、複数の浮動小数点レジスター、演算以外の用途で個別の役割を持つ専用レジスターに分類できます。

専用レジスターは、CPU の現在の状況を表すレジスター、命令を実行した結果を格納しているレジスターなどさまざまなモノがあります。専用レジスターは汎用レジスターと異なり、値をセットすることはほぼなく、格納されている値を参照する方が圧倒的に多いです。

## 5.2 レジスターの分類

レジスターは、用途別に用意されています。

内訳は「汎用レジスター」、「浮動小数点レジスター」、「専用レジスター」です。汎用レジスターは複数個あり、それらはどれも役割は同じ、ということになっています[注3)]。浮動小数点レジスターも複数個あり、それらはどれも役割は同じです。汎用レジスターと浮動小数点レジスターの違いは格納できる値の種類です。

専用レジスターは、1 つ 1 つが役割を持っています。数は少ないとはいえ、とってもよくお世話になるレジスターとなります。

## 5.3 汎用レジスター

汎用レジスターという文字だけを見ると、整数・小数問わず文字列でも何でも扱えるスゴイ変数みたいなイメージを持つかもしれません。名前に汎用とついていますからね。なんでもいけるっしょー的な感じで。

といいつつ、ここまで本書を読んできた聡明な読者の方々はもうすでに気づいていると思いますが、残念ながら整数しか格納できません。文字列など到底無理です。残念です。そんなイケているものではないのです。

汎用レジスターには整数が格納できます。ちなみに上限値は CPU が扱うビット数でも異なり、32 ビット CPU では $0〜2^{32}$-1 の値が扱えますし、64 ビット CPU では $0〜2^{64}$-1 の値が扱えます。

レジスターが扱える数値の範囲以上は扱えないのかというとそういう訳でもありません。たとえば 64 ビット CPU で 128 ビットの整数値の演算を行う場合は、汎用レジスターを 2 つ使い、1 つを上位 64 ビットの値、もう 1 つを下位 64 ビットの値を格納して表します。このあたりはコンパイラーが頑張るか、アセンブリ言語で人間が書いているのであれば人間が頑張るか、しないといけません。

---

注3) 特定の CPU アーキテクチャーでは汎用レジスターといいつつも、レジスターごとに得意な処理が異なっていたりします。詳しくは第 8 章「シンプルな CPU、複雑な CPU（RISC と CISC）」にて。

第 6 章

# CPUができることは多くない（命令）

本章では CPU が実行する命令について解説します。CPU が備える本質的な命令の種類や役割は Intel、Arm などの CPU アーキテクチャーによる違いはさほどありません。

ですので、本章では CPU アーキテクチャーに特化した話は少なめにして、命令の分類や機能をメインに解説します。このあたりをざっくり学んだ後に、それぞれの CPU アーキテクチャーでの解説書を読むと、割とスムーズに理解が進んでいくのではと考えています。

CPU の気持ちを理解する上では、アセンブリ言語のコードをなんとなくであっても読めるようになっている必要があります[注1)]。

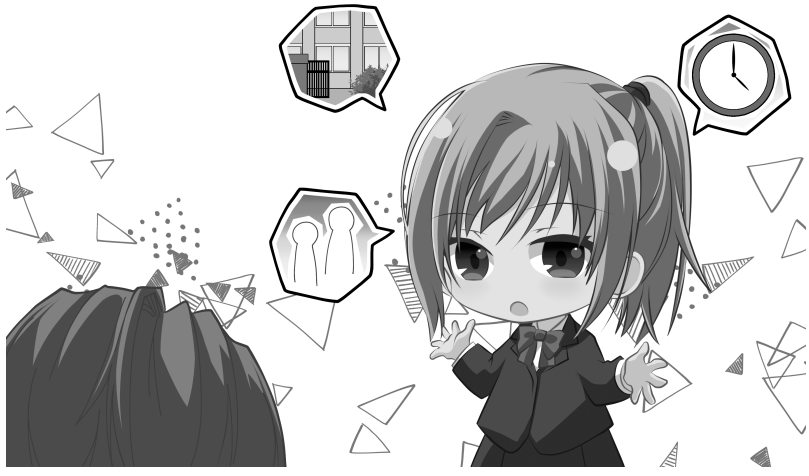
## 6.1 文法はあった、ような…

こういったプログラミング言語を覚える際には、printf("Hello, world") とか if (〜) {} みたいな文法を覚えるのが当然だと考えている人もいるでしょう。英語を勉強しはじめた時も「How are you?」といったような文法と「you」といったような単語の両方を覚えていった経験から、それが当然だと考えている方も多いかもしれません。

言語といえば文法が存在するのは人間が中心の世界です。機械が中心の世界では文法なんて不要なのです。

たとえるならば、C 言語などの高級プログラミング言語では「下校時に校門で待っているから一緒に帰ろう」と表記されるような文面を、アセンブリ言語では「下校時」「校門」「待っている」「一緒に帰る」といった具合に、キーワードが続くような表記をイメージするとよいでしょう。

注1) 人の気持ちも、なんとなくであっても読めるようになっていれば、もうちょっと違った学生時代が送れていたんだろうなぁ、と思うことがたまにあります。



●図 6.1　素っ気ない感じなのが、またいいのです

そんな感じなので、文法を覚えるというより単語を覚えるのがメインのような錯覚を覚えるかもしれません。見てのとおり文法なんてほとんど存在しないので。

アセンブリ言語で文法らしきものをみかけたときは、それは CPU が必要としているからではなく、アセンブラ（開発ツールの方ね）で要求しているからだ、程度に考えてもらった方がよいかもしれないです。

それぐらい文法が存在しません。アセンブリ言語では、先ほど登場した「下校時」といったキーワードは、命令で表します。

## 6.2　命令を詳しく

命令は、第 3 章で説明したとおり、外部から CPU へ処理指示を行う単位です。

アセンブリ言語のコードはすべて命令で構築されるため、一般的なプログラミング言語と考え方が若干異なります。これらを最初は意識しておきましょう。

学び始めた頃は少し混乱するかもしれませんが、そのうち慣れます。たぶん。知らんけど。

### 6.2.1　命令 ≠ 演算子

アセンブリ言語ではよくあるプログラミング言語のような文法らしい文法、演算子はありません。すべてが命令で構成されます。

たとえば加算処理を C 言語では、リスト 6.1 のように記述するでしょう。

●リスト 6.1　C 言語で加算処理

```
int c = a + b;
```

アセンブリ言語では、加算をプラス（+）演算子ではなく、加算命令である ADD 命令を使って記述します（リスト 6.2）。 ちなみに ; はアセンブリ言語でのコメントを表します。

●リスト 6.2　アセンブリ言語での加算処理

```
add w3, w1, w2  ; w3 ← w1 + w2
```

このように **CPU の処理はすべて命令として指示する**ことを覚えて…、いや慣れてください。

### 6.2.2　扱えるデータは 1 度に 1 つだけ

原則として 1 つの演算命令で得られる結果は 1 つだけです。アセンブリ言語のコードでは、そのような処理が延々と続くのです。

ただし、後ほど解説する SIMD 命令（複数の値を 1 度に演算できる命令）で扱われるレジスターは 1 つのレジスターに複数の値が代入できます。そして、SIMD 専用命令で処理することで、1 度に複数の結果が得られます。

### 6.2.3　命令の書式

命令の種類はいろいろありますが、まずは命令の読み解き方を先に頭へ入れておくとよいかもしれません。読み方がわかると、アセンブリ言語のコードもなんとなく読めるようになる、と考えています。

命令は 3〜4 字ぐらいの略語で構成されています。省略の仕方にパターンがあるので、そのあたりを覚えておけば OK です。この書式はだいたいどの CPU アーキテクチャーのアセンブリ言語でも似たり寄ったりです。

●表 6.1　命令の書式例

| 分類 | 命令 | 何の略称か |
|---|---|---|
| そのまま | add（加算） | |
| 先頭から数文字 | sub（減算） | substract |
| 母音抜き | cmp（比較） | compare |
| 頭文字から | bne（分岐） | branch not equal |

第 7 章

# 道は分かれる（分岐命令）

人生において、何かしら選択を迫られることはよくあります。小さな頃から一緒に遊んで居た幼なじみと、自分を慕ってくれるお金持ちの令嬢の二人から言い寄られ、どちらと結婚するのか決めないといけない、といったように。

そのような選択を迫られるのは CPU の内部では日常茶飯事です。本章では CPU で実行する命令でも、とくに重要な分岐命令について解説します。CPU の気持ちを知る上では、この分岐命令の理解はおさえておかないといけないのでね。

## 7.1 分岐命令とは

アセンブリ言語での分岐命令とは、次の命令実行先を変更する命令です。
分岐命令は大きく分けて 2 種類あります。

- 無条件分岐命令
- 条件分岐命令

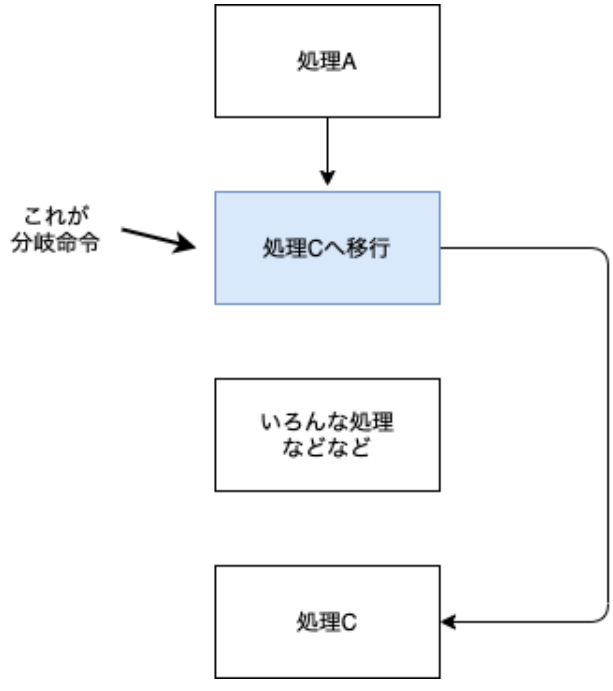
ではこれらについて解説します。

●図 7.1　どちらか選ぶなんて無理、なんてこともよくありますよね

### 7.1.1　無条件分岐命令とは

無条件分岐命令とは、問答無用で分岐…、というか、次に実行するメモリアドレスを変更（プログラムカウンターの値を変更）する命令です。別名ジャンプ命令と呼ばれることもあります。C 言語で例えるならば GOTO 文です。

これって分岐とはいわないのでは…、とツッコミの 1 つも入れたくなりますが、そんな野暮なツッコミをしないのが大人ってもんです。




●図 7.2　無条件分岐命令

### 7.1.2　条件分岐命令とは

条件分岐命令とは、次の二択で実行先が変わる命令です。

- 条件が合致したら、次は指定したメモリアドレスへ実行先を変更（プログラムカウンターの値を変更）して命令実行を継続
- 条件が合致しなかったら、条件分岐命令の次の命令を実行

これこそ分岐ですね。C 言語でたとえると、if 文 + GOTO 文という感じです。図 7.3 では、ループ処理での条件分岐命令の役割を表しています。

第 8 章

# シンプルなCPU、複雑なCPU（RISCとCISC）

CPU の仕組みの話をしていくと、RISC（Reduced Instruction Set Computer、縮小命令セットコンピューター）抜きには語れません。

RISC に触れず CPU の仕組みを解説しようとしても「あの機能は、RISC から始まり…」「この機能は、RISC で採用されたことから広まり…」「くそっ！　この世界線の CPU は RISC の影響を受けてないモノを探す方が難しいじゃないか」といったような、どこかの異世界転生ものの小説のような展開になるのです。

ということで、ここはすっぱり諦めて RISC の解説を進めていきます。

## 8.1 RISC とは

RISC とは、CPU の設計手法の一種です。RISC の詳細な説明は後にまわすとして。最近の CPU 設計は、多かれ少なかれ何かしらの RISC 的な設計手法を取り入れています。ちなみに、身近にある RISC と言い切れる CPU は Arm です。

RISC の説明を行う際、CISC（Complex Instruction Set Computer、複合命令セットコンピューター）という単語が対になって登場します。これは、RISC という新しい CPU アーキテクチャーが公表された際、それ以前の CPU アーキテクチャーを指し示すワードがないと何かと説明が不便なので CISC というキーワードが作られたという経緯があります。

**Column.** RISC と Arm

Arm 社の始まりはエイコーン（Acorn）社のスピンオフだったりします。エイコーン社はすでにありませんが、Apple Ⅱ、ファミコンの CPU である 6502 を設計した会社です。

Arm は、元々は Acorn RISC Machine の頭文字をとって ARM と付けられました。その後に Advanced RISC Machines へと変更され、今の Arm となります。

## 8.2 コンピューターの歴史をざっくり

RISC と CISC の説明をいきなり始める前に、おそらくコンピューターの歴史的経緯を知っておいた方が、何かと理解が早まるでしょうから、先に歴史的経緯をざっくりと解説します。詳細を知りたい人はウィキペディアをみるなりしてください。そちらの方がずっと詳しく書かれていますので。

1 台のコンピューターが冷蔵庫ぐらいの大きさだった頃の時代の話です。コンピューターという資源は非常に高価で貴重でした。ですので、なるべく高速に演算できるように、またメモリの消費量を極力抑えてコードを書くのが当時の主流でした。

この時代はコンパイラーの技術も未熟でしたので、これら貴重な資源であるコンピューターを有効活用するためには、人間が機械に歩み寄る方法、つまりアセンブリ言語でコードを書くのがもっとも理にかなった方法だったのです。

そのため、CPU の命令は人間がアセンブリ言語でコードを書きやすいように設計されました。たとえば、データの取得方法の指定も柔軟に指定できるよう○○命令を用意した、といった具合です[注1)]。

しばらくはこれで良かったのですが、時代が進むにつれて、いくつかの問題にぶち当たりました。大きめな問題としてはつぎの 3 つが挙げられます。

- 命令数の増加に比例して、CPU 全体の回路がどんどん複雑化していく
- 回路が複雑になり、1 命令の実行速度がどんどん遅くなる
- 使用されない命令が増えていく

これらの問題を解決すべく導きだされたのが「命令の数を減らした CPU を設計してみよう」という方向です。この方向性になったのも、もちろん根拠があってのことです。たとえば「既存のコードを調査した結果、CPU にある命令はすべて使われているわけではない」「命令の数を減らして設計してみると都合のよいことが多いと分かった」などです。

命令を減らし回路をシンプルにすることで、パイプラインなど他の機能も組み込めるようになり、実行速度の向上も進めやすくなりました。

といってもよいことづくめであった訳でもなく。

RISC は命令の機能をシンプルにして命令数自体も削減した結果、生成されたコードサイズは CISC でのそれよりも大きくなりました。

そして、命令を実行する上で CISC ではあり得ない制約も発生しました。この制約は処理速度とのトレードオフなので致し方ないところでしょう。技術の進歩によりコンパイラーの性能もよくなってきているので、これも大きな問題ではなくなりました。

---

注1)　モトローラ社の 68000 という CPU がそれを極めたもののひとつですね。

●図 8.1　同じ性能でも、時代によってコンピューターの大きさが全然違うんですよね

## 8.3 RISC の特徴

RISC という単語は何かしら明確な定義はありませんが、次の特徴を持つ CPU の意味で用いられていることが多いです。

- ロード・ストア　アーキテクチャーである
- 汎用レジスターがすべての命令で値の取得元や格納先として利用可能
- 命令が固定長である
- 命令は 1 クロックで動作する

このあたりを CISC と比較しつつ解説します。ここでは対比しやすいように CISC の代表例として Intel i386、RISC の代表として MIPS R2000 を元に解説します[注2)]。なんで MIPS かというと世界で最初に商業化された RISC の CPU ですからね。コンピューターサイエンスの界隈では著名な書籍である「コンピューターの構成と設計」（通称パタヘネ本）でも解説されているのでご存じの方もいらっしゃることでしょう。

ちなみに、Intel i386 は今の Windows PC の祖先に当たる DOS/V 機で採用されていた CPU で、MIPS R2000 は初代 PlayStation に搭載されていた CPU の一世代前の CPU です[注3)]。

注2)　i386 は 1985 年発売、R2000 は 1986 年発売でほぼ同時期に発売された CPU なのです。
注3)　初代 PlayStation は正確には R3000 ですが設計はほぼ同じです。

第 9 章

# 記憶の仕組み（メインメモリ）

CPU はそれ単体で動作するということは決してなく、その周辺回路とまぁいろいろ仲良く、時には足を引っ張りあいながら動作しています。

CPU の周辺回路のうち、非常に密接な関係があるのはメインメモリでしょう。CPU はメインメモリからコードやデータを取得して動作している訳ですから。

ということで、CPU とメインメモリの関係について解説します。CPU の気持ちを知る上では、CPU だけではなく、その周辺にも気を配る必要があるのです。

## 9.1 CPU とメインメモリの関係

メインメモリは、CPU がやりとりする周辺機器の中でも特に密接な関係があります。どれだけ密接かというと、実行するコードを呼びだしたり、CPU がデータを一時的に保管したりなど、とにかく CPU がもっとも頻繁にアクセスする周辺機器な訳ですからね。

CPU のいくつかの機能はメインメモリと密接に関わっているので、CPU からみたメインメモリの基礎知識を解説します。

やっぱねえ、仲良くなりたい女の子がいるときは、その友人たちともお互いに仲良くなっておくとよいのですよ。いろいろ協力してもらえたりしますからね。CPU と周辺機器との関係性は、人間関係にも共通するところがあるわけです。

## 9.2 メモリアドレス

メモリアドレスとはメモリに対して 1 バイトごとに割り当てられた番地（アドレス）のことです[注1)]。メインメモリにデータを読み書きする場合は、メモリアドレスを指定した上で読んだり書いたりします。たとえば、「メモリアドレスの 0x1000 番地から 4 バイト取得します」「0x1020 番地へ数値として 200 書き込みます」といった感じです。

注1) そうじゃないコンピューターも世の中に存在します。まー、そのようなコンピューターに触れる機会自体が稀なので考えなくてよいかな、と。

●表 9.1　メモリアドレスは 1 バイト毎に割り当てられている

| メモリアドレス | 0x1000 | 0x1001 | 0x1002 | 0x1003 | .. |
|---|---|---|---|---|---|
| 値 | 0x24 | 0x56 | 0xfb | 0x02 | .. |

ただ、扱う値がすべて 1 バイト以内に収まるということは稀で、実際には 65535 といった複数バイトで扱わないといけない値を扱う場面がほとんどでしょう。

そのような 2 バイト以上で表す数値をメモリへ格納する場合、次の 2 つは知っておくべきです。

- バイトオーダー
- アライメント

では、これらについてみていきましょう。

### 9.2.1　バイトオーダー

バイトオーダーとは 2 バイト以上で表す数値をメモリに格納する順序を指します。

たとえば、0x12345678 という数値をメモリに格納する際には、「下位バイト→上位バイトの順に格納する」「上位バイト→下位バイトの順に格納する」という 2 つの記録方法が考えられます。

数値を下位バイト→上位バイトの順に格納する手法をリトルエンディアン、上位バイト→下位バイトの順に格納する手法をビッグエンディアンといいます。

ちなみにリトルエンディアン・ビッグエンディアンのいずれにも設定できるものは、バイエンディアンといいます。

●表 9.2　バイトオーダー

| | 0x100 番地 | 0x101 番地 | 0x102 番地 | 0x103 番地 |
|---|---|---|---|---|
| リトルエンディアン | 0x78 | 0x56 | 0x34 | 0x12 |
| ビッグエンディアン | 0x12 | 0x34 | 0x56 | 0x78 |

現在の CPU アーキテクチャーは、かなりの割合でリトルエンディアンが採用されています。Arm は、設計自体ではバイエンディアンとなっておりリトルエンディアン・ビッグエンディアンのいずれも設定可能です。実際の機器を設計する際にリトルエンディアン・ビッグエンディアンのいずれかを決めるのですが、かなりの割合でリトルエンディアンが指定されています。

ということで、ビッグエンディアンを採用している CPU は絶滅危惧種といってもいいくらいに貴重です。

#### バイトオーダーを確認してみよう

ということで、メインメモリに値が格納されている様子を確かめてみましょう。

ここでは、M1 MacBook Air 上で動作している統合開発環境（CLion）をつかって、コンピューター内部のプログラム実行中のメインメモリの状況をみてみます。こういうものは実際に中身をみてもらった方が早いと思うので。

リスト 9.1 を実行中に一時中断しメモリダンプを表示してみました。

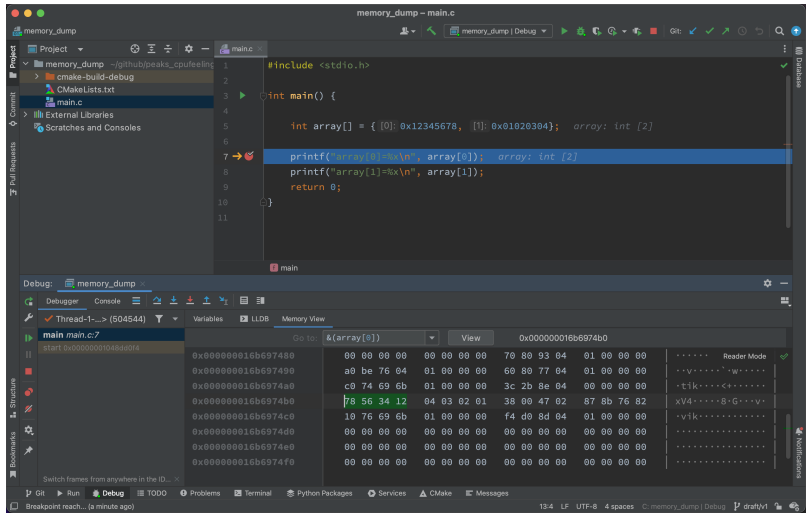
**●リスト 9.1　メモリダンプ確認**

```
#include <stdio.h>

int main() {

  int array[] = {0x12345678, 0x01020304};

  printf("array[0]=%x\n", array[0]);
  printf("array[1]=%x\n", array[1]);
  return 0;
}
```

実行中のメインメモリの状態は、図 9.1 の右下に表示されている Memory View に表示されています。この 16 進数の羅列は、一般的にはメモリダンプと呼ばれています。

リスト 9.1 の実行を一時中断し、array[0] の値がどのように格納されているのか確認してみました。

図 9.1 では、Memory View の array[0] の値が格納されているメモリエリアが色違いで表示されています。確認すると、0x12345678 という値はメインメモリには 0x78, 0x56, 0x34, 0x12 の順に格納されていることがわかります。




**●図 9.1　プログラム実行中のメモリの様子**

第 10 章

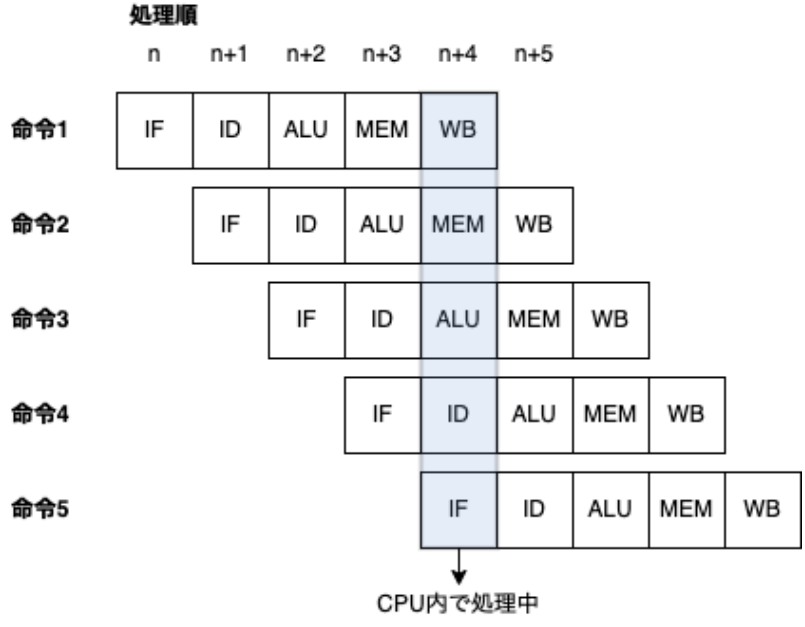
# 処理を効率よく実行する仕組み（パイプライン）

現在の CPU では、実行される命令はパイプラインを通して処理されている場合がほとんどです。パイプラインと聞くと、連想するのが「ゴルビーのパイプライン大作戦」というゲームだったりします。遊んだことないですけど。

ゴルビーの話は横に置いといて、パイプラインという仕組みは以前からあったものの、着目されるようになったのは MIPS で採用されてからです。その後はほぼすべての CPU でパイプラインが採用されるようになりました。RISC（というか MIPS）の設計思想は、その後の CPU 設計に大きく影響したことがほんとよく分かります。会社はなくなっても、その設計思想はずっと息づいているわけですねー。

## 10.1 パイプラインとは

パイプラインとは 1 つの命令を複数の処理に分割、複数の命令を別々にずらして実行する仕組みを指します（図 10.1）。




●図 10.1　パイプラインで命令が実行される様子

この図では、1 つの命令を IF、ID、ALU、MEM、WB という 5 つの処理に分解して実行しています（IF、ID って何やねん、というツッコミもあるでしょうが、これらはこの後解説します）。なお、IF、ID などの処理（回路）はすべて独立しています。そのため、命令 1 が IF を処理し、その 1 クロック後、命令 1 は ID の処理を行い、命令 2 は IF を処理する…、といった具合に連続して処理を行っています。

## 10.2　とんかつの調理でたとえてみる

先の解説だけでは、よく分からないというのが正直なところでしょう。

そこで、パイプラインの導入前と導入後をとんかつの調理でたとえて解説します。ほとんどの人が一度は食べたことがあるでしょうし、なんとなくでしょうがとんかつの作り方も知っていることでしょうし。

ここでは、とんかつが 1 枚揚がる＝ CPU の 1 命令が実行完了する、という感じでとらえてもらえば OK です。

さっそく、とんかつの調理に移ります。とんかつ用の肉をはじめ、とんかつを作るのに必要な具材や調理器具などは全部揃っているものとします。あとは調理するだけです。いやー、手際よいですね。

とんかつの調理は次の手順で行われます。あとは、こちらの都合で申し訳ないですが、各プロセ

スの調理時間は一律 1 分とします[注1)]。

1. 肉の筋切り
2. 下味をつける
3. 小麦粉をつける
4. 溶き卵をつける
5. パン粉をつける
6. 揚げる

さて、この流れで 3 枚のとんかつ（＝ 3 命令を実行する）を揚げていきましょう。

●図 10.2　とんかつ揚げるのは気を遣いますよね

### 10.2.1　1 枚づつ調理する

パイプライン導入前は、ちょうどとんかつを 1 枚づつ揚げていく様子に似ています。1 枚づつというのは、1 枚目のとんかつを肉の筋切り～揚げ終わり、次に 2 枚目のとんかつの筋切りを行い…、といった流れのことです。

この方法では 1 枚の調理に 6 分、3 枚では 3 × 6 ＝ 18 分かかる計算となります。この流れでも調理はできます。ただ、同時に 1 つのことしか行っていないため、他に利用されていないプロセス

注1)　技術解説の例なので、調理方法に厳密さは求めないで…

第 11 章

# 手が届く範囲にモノがあると便利だよね（キャッシュメモリ）

本章では、今時の CPU には当たり前のように搭載されているキャッシュメモリについて解説します。

そういえば、キャッシュの単語の綴りは cache です。稀に cash の綴りで表記されている書籍や Web サイトをみかけたりします。いうまでもなく cash だと現金の意味になります。ま、cache と cash はいくらあっても困らない点は同じですが。

## 11.1 コンピューターにおけるキャッシュが指すもの

コンピューターの世界で登場するキャッシュという単語は、「1 回目の処理時間はそこそこだけど、2 回目以降の処理は速くなる」的な仕組みを指している場合がほとんどです。

たとえば Web ブラウザにあるブラウザーキャッシュも、「とあるサイトの初回アクセス時の速度はそこそこだけど、2 回目の表示は速くなる」仕組みを実現しています。

初回アクセス時に画像データなどの静的データを PC に保存しておき、2 回目以降は PC に残されている画像データを参照しつつ、相手のサーバからもデータを取得して画像を表示する仕組みで速度アップを実現しています。

## 11.2 人間とメイドロボ

本章で説明しようとしているキャッシュメモリというのは「以前に実行した命令やデータを一時的に記録しておくメモリエリア」となります。目的はもちろん CPU の処理の高速化のためです。

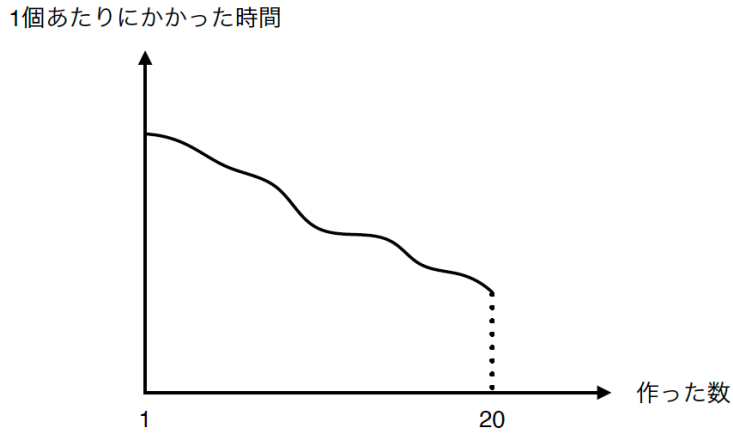
なんでキャッシュメモリを積むと高速化するかは後で説明するとして。キャッシュメモリがない場合とある場合でどの程度処理速度が違ってくるのかみてみましょう。

### 11.2.1 人間の場合

人が繰り返し作業をしていると、その処理能力が徐々に上がるというのはみなさんも経験があるでしょう。例としてクラス全員の男子に配る義理チョコをきれいにラッピングする女の子を想定し

てみます[注1)]。

ラッピングのような単純作業は、最初は 1 袋で 2 分掛かかっていたが最後は 30 秒でできるようになった、といったことはよくあることでしょう。人間は単純作業を繰り返すうちに、動きがスムーズになって徐々に早くなる、といったイメージですね。




●図 11.1　慣れてくるとスピード上がりますよね

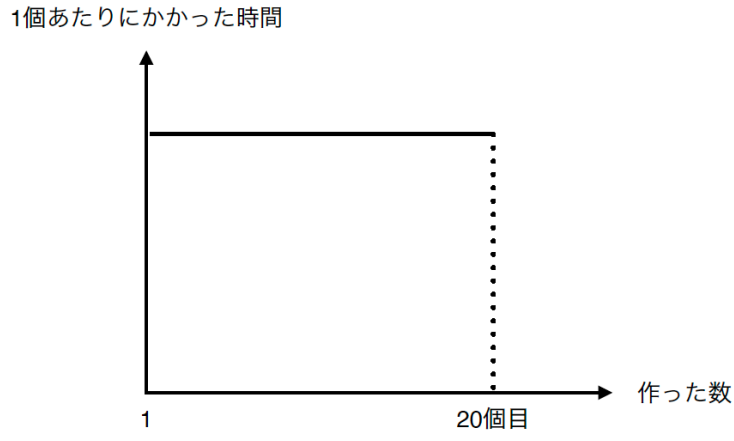
### 11.2.2　コンピューターの場合

コンピューターではキャッシュメモリの有無によって、処理効率は大きく異なります。

ということで、ここではちょっと試作型メイドロボを例にして、キャッシュメモリを搭載していない場合と、搭載した場合の違いをみていきます。試作型ですのでキャッシュメモリを搭載するスロットがあるものの、空きとなっていたとします。そんなメイドロボがどこかのお屋敷に仕えることとなりました。

その仕事の 1 つに 20 個のジャガイモの皮むきがありました。皮むきには 1 個あたり約 10 秒かかっており、20 個の皮むきでは約 200 秒かかっていました。

注1)　女の子に免疫がない男子にとって、割と罪深い行為ですね…。

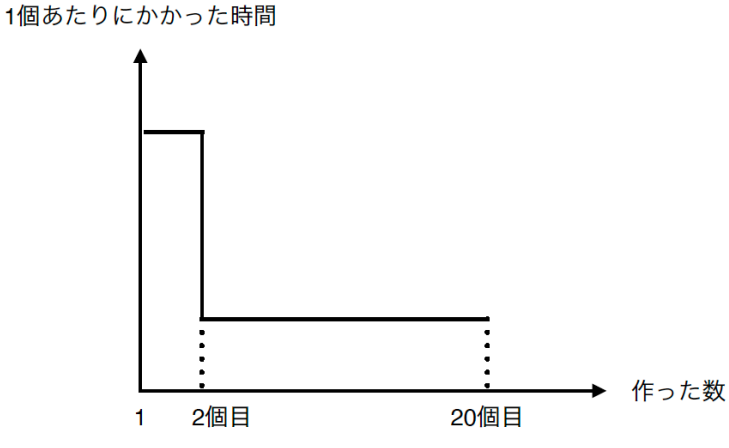



●図 11.2　キャッシュメモリがないと、つねに処理速度は同じ

ある日、このメイドロボにバージョンアップが施されることとなり、キャッシュメモリが搭載されて、お屋敷に戻ってきました。

そこで、以前と同様に 20 個のジャガイモの皮むきを頼んでみました。1 個目に 10 秒かかるのは以前と同様でしたが、2 個目以降の皮むきが 3 秒でできるようになりました。キャッシュメモリの効果ですね。

そのため、20 個のジャガイモの皮むきは、10+3 × 19=67 秒で終わらせることができるようになりました。いやー、キャッシュメモリの効果は絶大ですね。




●図 11.3　キャッシュメモリがあると、2 度目以降は格段に処理速度が向上する

第 12 章

# CPU と周辺機器との結びつき（I/O）

I/O（アイオーと読みます）という文字を見て何を連想するでしょうか？　その返答で、その人がどのような人生を歩んできたかみえてくる気がします。パソコンの周辺機器メーカーでしょうか？　コンピューター雑誌でしょうか？　それとも、それ以外でしょうか？　マルイを連想する人がいたら、それは私と同じです。

## 12.1 I/O とは

I/O とは Input/Output（入出力）を指します。コンピューターの五大装置[注1)]のうちの 2 つは入力と出力ですしね。これらをまとめて入出力と呼びます、この入出力はコンピューターには必要不可欠なものです。やはり入出力がないとコンピューターなんてタダの箱ですしね。

コンピューターの入門書には、入力はキーボードやマウス、出力はディスプレイやスピーカーが紹介されているでしょう。

確かにこれらは入力と出力に違いないですが、本書は CPU と友達以上恋人未満の関係になるための本です。なので、CPU から見た入力、出力について解説します。

## 12.2 CPU と周辺機器とのやりとり

CPU はさまざまな周辺機器とやりとりしながら処理を進めていることを第 4 章にて解説しました。では、具体的に CPU は周辺機器とどのようなやりとりをしているのかをみてみましょう。

例として CPU が SD カードのデータを取得する流れを擬人化して再現してみます。SD カードへの読み書きは専用のコンローラ（制御装置）で制御されていることが多いので、そのやりとりを再現してみます。このやりとりは CPU は関東出身、SD カードのコントローラは関西出身という設定です。この設定に深い意味はありません。

注 1)　五大装置とは、演算装置、制御装置、記憶装置、入力装置、出力装置のことです。演算装置、制御装置、記憶装置はここまでで解説済みです。

●リスト 12.1　CPU と SD カードのコントローラーとのやりとり

```
「コントローラの初期化をお願いします」
「初期化終わりましたか？」
「まだやで」
「初期化終わりましたか？」
「まだやで」
 ...
「初期化終わりましたか？」
「初期化終わったでー」

「ブロック番号1024番のデータ読み込みをお願いします」
「読み込み準備できましたか？」
「まだやで」
「読み込み準備できましたか？」
「まだやで」
 ...
「読み込み準備できましたか？」
「いつものメモリからデータ読めるようになったでー」
```

これで、いつものメモリエリア（データが読み込めるメモリエリアが決まっている）からブロック番号 1024 番のデータが取得できます。

先のやりとりをみて「SD カードからファイルを読み取る時って、sample.txt みたいなファイル名を指定しないの？」と疑問を持った人もいることでしょう。実はファイルといった概念は OS の仕事の範疇ですので、CPU が SD カードからデータを読み書きする場合は、先のようなブロック番号指定となるのです。

このやりとりは OS の内部で行われていることが多いので、ユーザが目にする機会はほとんどないんですよねぇ。

## 12.2.1　入力と出力

先ほどの CPU と SD カードのコントローラーとのやりとりの中にも入出力に関するキーワードが含まれています。そのあたりをみていきましょう。

### 入力

入力は周辺機器に対して信号を送ることを指します。リスト 12.1 では、次の言葉がそうですね。

●リスト 12.2　input1

```
「初期化をお願いします」
```

CPU から機器に対して何か指令といった信号を送信するのは入力ですね。

### 出力

出力は周辺機器から CPU へ返される信号を指します。

リスト 12.1 では、次の言葉がそうですね。

●リスト 12.3　output1

```
「まだやで」
「初期化終わったでー」
```

## 12.3　メモリマップド I/O

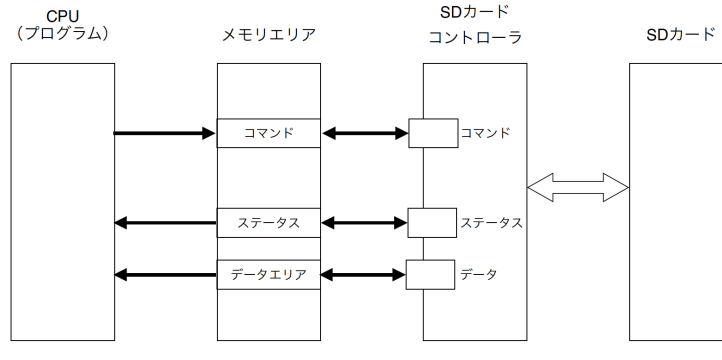
これらの入出力は、実際のソースコードで書くとしたら、どんな風に書けばよいのでしょう？

機器を直接制御するプログラムを作成したことがある方ならば、これらの処理はライブラリを呼びだして実現した、という方もいることでしょう。

そのライブラリの中では、どのような処理が行われているのか、というと、ライブラリの中では特定のメモリアドレスに対して読み書きしている処理をしています。

つまり、入力は特定のメモリエリアに値を書き込むことで実現できますし、出力は特定のメモリエリアの値を読み取ることで実現できる、ということです。

このようなメモリエリアの一部を周辺機器の I/O（入出力）として割り当てる方法をメモリマップド I/O（memory mapped I/O）といいます。ついでに説明すると、メモリマップという単語は文脈によっても異なりますが、基本的にはメモリの割り当て構造を指す単語です。




●図 12.1　メモリマップド I/O のしくみ

第 13 章

# 多くの仕事を差し込まれる立場です（割り込み）

社会に出て働くようになると 1 度ぐらいは"仕事中に電話が掛かってきたので、今取りかかっていることを中断して電話対応した"といった経験をするでしょう。

なんだかんだと電話の対応を終わらせた後、先ほどの仕事を再開しようとしますが「あれ？　どこまで進めていたっけ…？」と思い出すのに時間かかった。そんな経験はないでしょうか？　私はよくあります。

まぁ、思い出すのに時間がかかるようになってきたのは少し気になっていますが…。

## 13.1 割り込みとは

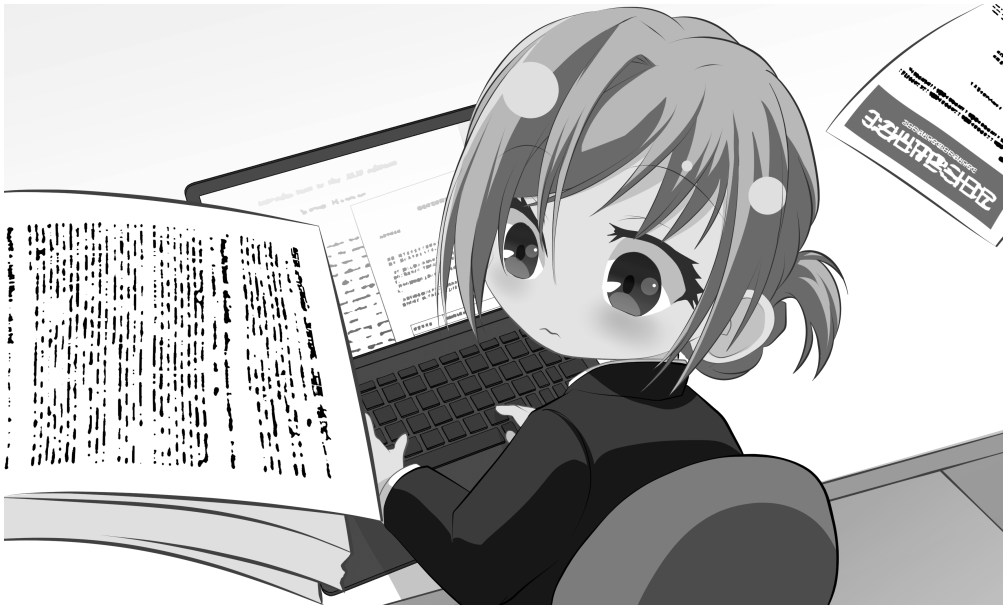
CPU の内部においても、「ゴメン、○○さん。仕事中に悪いんだけど、こっちの仕事を大至急で進めてくれないかな。今度昼飯奢るからさ。」みたいな出来事は頻繁に発生しています。残念ながら CPU は昼ご飯を食べないので、昼飯を奢ってもらう約束は永遠に果たされません…。

話をもどして。このような、外部から CPU に対して処理が依頼される現象を割り込みといい（そのままですね）、割り込みによって実行される処理を割り込み処理といいます。ちなみに CPU はこのような割り込みが入ることは想定内なので、昼飯を奢るほどのことではありません。

コンピューターでの割り込み処理は、元々は「信号が入力されるタイミングの不明な信号を優先的に処理するため」の仕組みとして用意されました。

たとえば、キー入力やマウス操作などの人間からコンピューターへ渡す情報がその代表例ですね。

ですが、この割り込みの処理は、OS などいろんなところで使われるようになりました。



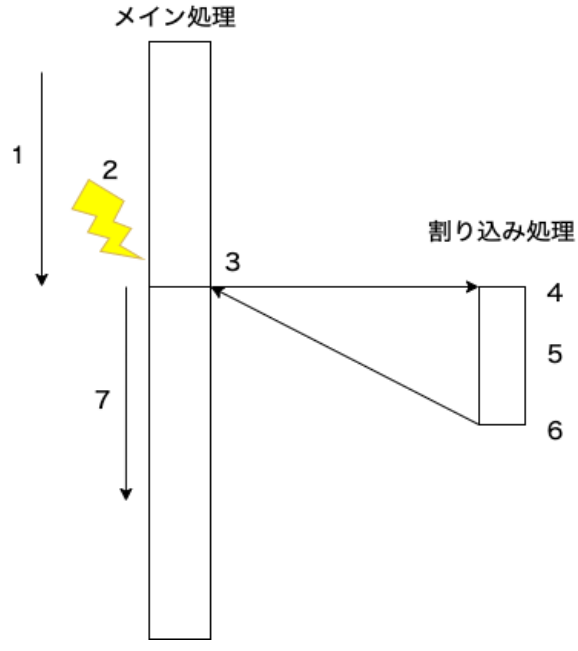
●図 13.1　仕事の最中に別の仕事を差し込まれること、よくありますよねぇ

## 13.2　割り込みの処理の流れ

割り込み処理の流れをざっくり解説すると次のようになります。

1. メインの処理が実行中である
2. 優先して処理すべきイベントが発生する（割り込みが発生）
3. メインの処理を中断する
4. 現在のレジスター情報、OS 固有データをスタックメモリへコピーする
5. 割り込み処理を実行、完了する
6. スタックメモリからレジスター情報、OS 固有データを戻す
7. メインの処理を中断した箇所から再開する

といった流れで処理されます。まずは、これが基本的な流れとなります。




●図 13.2　割り込み処理の流れ

さきほどの例を、外部に接続されたボタンで割り込みを発生させ LED を ON/OFF する、という処理を例に解説してみます。

1. メインの処理が実行中である
2. 外部に接続されたボタンが押された（割り込みが発生）
3. メインの処理を中断する
4. 現在のレジスター情報、OS 固有データをスタックメモリへコピーする
5. LED を点灯する処理を実行、完了する
6. スタックメモリからレジスター情報、OS 固有データを戻す
7. メインの処理を中断した箇所から再開する

このような感じで処理が進みます。

## 13.3　割り込みのありがたみを理解する

割り込みがあると何がうれしいのか、ちょっとピンとこないかもしれません。ですので、割り込みのありがたみを理解するためにも、ここでは、割り込みでよく処理されるキーボードからのキー入力を例に解説してみます。

付録 A

# 次に読むべき本

さて、本書で CPU の中身についてなんとなく理解できたことでしょう。

ということで、次のステップに進むための書籍をいくつかご紹介します。といっても、次のステップで何がしたいかにもよってオススメする本も違ってきます。

いくつかの方向性を示しつつ本を紹介しておきます。自分にあった本を探してみてください。

## コンピューターの構成と設計 第 5 版

通称パタヘネ本。コンピューターサイエンスを大学院などで学んだことがある人ならば、この本のタイトルについて一度は耳にしたことがあるでしょう。

本書で解説した話、とくに RISC や MIPS 関連についてはこの本がもっとも詳しいですね。それもそのはず、世界初の RISC チップである MIPS の設計者が書いた本ですし。

ということで、一度読んでおくとその後のソフトウェアエンジニア人生で十分役立つことでしょう。本の分厚さで軽く絶望するかもしれないけど、がんばって！

## プロセッサを支える技術

こちらは回路設計の技術者がソフトウェアエンジニア向けに、CPU というハードウェアについて解説した書籍です。この本で解説したメモリキャッシュやパイプラインについても、より詳細に解説しています。この本も古め（初版 2011 年）ではありますが内容的には今でも十分に役に立ちます。

パタヘネ本は内容や価格で人を選びますが、この本はもう少し気軽に読めることでしょう。

## 独習アセンブラ　新版

世間に流通している CPU 向けに、アセンブリ言語を書いたり読んだりしていきたい人向けの本です。この手の本は昔はいろいろ出版されていましたが、最近見かけることはぐっと減りました。

ということで、この手の解説が日本語で読める数少ない書籍の 1 冊です。

この書籍では、Intel アーキテクチャーのアセンブリ言語をメインに、Arm や COMET Ⅱまで

# 索引

## 著者

### 出村 成和 @checkela

北陸からIT企業にリモートワークしているソフトウェアエンジニア。弱小サークル「トゲトゲ団」を主宰。下はCPUから、上はAWS、ゲームエンジンまで多岐に渡るレイヤーでゲーム関連のあれこれをしています。