計算機科学基礎'13 ＃1 – 計算機とそのハードウェア

久野　靖*

2013.4.10

1　コンピュータについて学ぶことの必要性

情報社会である今日では、コンピュータとネットワークが非常に多くの場面で使われており、大きな役割を果たしています。そこで本科目「計算機科学基礎」では、コンピュータやネットワークとは何でありどのように作られているのか/どのように動いているのかという、原理に関することがらを、一通り学んで頂くことを目的としています (後続科目である「計算機科学」ではさらに、これらの主要な応用分野について個別に学ぶようになっています)。

ではなぜ、コンピュータやネットワークの原理について学ぶ必要があるのでしょうか? 別に原理など知らなくても、PC やスマホで必要なアプリケーションを動かしてやりたいことができれば十分なのではないでしょうか?

もちろん、とりあえず目の前にやることがあって、それをこなすためのソフトウェアがあるのなら、それを使うことは当然です。しかし皆様が「本当に」しなければならないことは、次のようなことではないのでしょうか?

- まだ存在していない「何か」を可能にするようなビジネスモデルやそれを実現するシステムを構想し、それを (たぶんソフトウェア開発をする多くの人たちと協力して) 実現する。
- コンピュータやネットワークの上で現に起きているか、またはこれから起きる可能性のある「何か」を予期し、それを解決したりそれに備えたりするような方策を (たぶん技術の専門家と協力して) 実現する。

これらのタスクをこなすためには、当然のことですが、今あるアプリをそのまま (普通に) 使うだけでは不十分です。では何が必要かというと、コンピュータやネットワークとはそもそも何であり/どのように作られていて、その結果、どんなことは原理的に可能/不可能かのかを知っていることが必要です。そうしないと、不可能なことを追い求めて時間を無駄にしてしまったり、本当なら可能なことを検討もせずにあきらめてしまったりしますから。

たとえば、次のものはすべて、ほんの 20 年ちょっと前、1990 年にはまだ存在していませんでした。

- World Wide Web、ブラウザ
- 検索サイトや地図サイトなどの情報サービス
- Twitter や Facebook などのコミュニケーションサービス
- iTunes Store などの音楽配信サービス
- YouTube などの動画共有サービス

しかし今ではこれらの無い生活などほとんど考えられないですよね? そしてこれから数年のうちに、これらと同じくらい我々の生活に変化を与えるものが、また次々と出て来るでしょう。そのとき、それを自分で生み出したり、自ら生み出さないまでも乗り遅れないうちに対応できるようになるためには、単なる「ソフトに使われているお客様」ではいけないでしょう、というのが本科目の主旨です。

そしてあともう 1 つ、本科目には「隠された主題」があります。それは、「科学的に考える」「自分の頭で考える」ことの練習ということです。日本の社会では、「勉強」というのは単に言われたこと

*経営システム科学専攻

を覚えておいてテスト用紙に向かって吐き出せばそれで済む、という側面が色濃くあり、大抵の学生さんはそれに毒されています。

もちろん皆様は社会人ですから、会社における現実の問題がそんなものではないことは十分お分かりと思います。しかし、会社を離れた「大学 (院) の勉強」もそうだということは、あまり意識されていないのではないでしょうか。この科目に出て来る情報技術のさまざまな事柄は、すべて「調べれば分かる」ことであり、「なぜそうなるのかには理由がある」「筋道を立てて (科学的に) 考えれば分かる」ことです。そのことを納得し、一見知識を問われているように思える問題であっても、自分で考えて答えに到達するという練習を、この科目ではして頂きたいと思っています。

以下、まず第 1 回では、コンピュータの目に見える装置部分である**ハードウェア** (hardware) とその原理について説明します。その後、第 2〜3 回は目に見えないけれどコンピュータが何をするかを定める主要部分である**ソフトウェア** (software)、第 4〜5 回は今日の社会において大きな影響をもたらすようになった**ネットワーク** (network) を取り上げて行く予定です。

2　コンピュータとその本質

2.1　コンピュータとは

まず、「**コンピュータ** (computer) とは何か?」という質問を投げかけてみましょう。あなたならどのような答を考えつくでしょうか? それは、たとえば次のようなものでしょうか?

△ コンピュータとは、計算をするための装置である。

もしそうだったら、コンピュータと電卓は同じものでしょうか? そうではないですね。ここで「大規模な」とか「大量の」とか「高速に」とつけ加えようとする人がいるかも知れませんが、それらの方も残念ながら「いまいち」です。確かに世界最初の実用化された電子コンピュータである ENIAC[1]は、大量の数値を「計算」するため[2]に作られたのですが、現在のコンピュータの用途としては数値の「計算」はマイナーな用途になっています。[3]

ではどうしましょうか? 世の中にはすでに大量のコンピュータが使われているわけですから、実際に何に使われているかを考えてみてはどうでしょう? たとえば、ちょっと考えてみただけでも、次のような「使われ方」を思いつくはずです。

- 銀行の窓口の裏側で、口座のお金の出し入れなどを記録し管理している。
- JR や旅行代理店の窓口などで、切符を発行したり座席の予約を受け付けている。
- ビデオゲーム機の中にあって、ありとあらゆるゲームを動かしている。
- 洗濯機やエアコンの中で、衣類をうまく洗ったり部屋をうまく空調してくれる。
- インターネットの向う側からさまざまな情報を含んだ画面を取り出して表示してくれる。
- 見ため美しい文書を作成させてくれたり、絵を描いたりさせてくれる。

まだまだ考えつくはずですが、とりあえずこれくらいにしておいて、元の質問に戻りましょう。コンピュータにこれらのことができるとして、ではこれらに共通するのは何でしょう? ほとんど絶望的にバラバラのように思えますか?

実は、これらのことがらに共通することが 1 つあります。それは「実体がない」ということです。たとえば、口座のお金の出し入れというのは誰のどの口座にいくら入金/出金があったか、ということ

[1] イギリスで作成された暗号解読用の電子コンピュータの方が先に完成していたという説もあります。

[2] 砲弾の発射と同時にその砲弾の着弾点を計算し始めると、実際に着弾する前に計算が完了するので、「弾よりも速いコンピュータ」と呼ばれたという逸話があります。

[3] ただし、一見計算と関係無さそうに思えるコンピュータグラフィクス (とくに動画) では、大量の数値計算が必要です。グラフィクスの計算には汎用 CPU ではなく専用のプロセサ (GPU) を使うのが一般的ですが、GPU の計算能力をグラフィクス以外の計算にも利用する流れもあります。

を記録することであって、その作業自体には見える実体がありません。もちろん、振替用紙を渡したり ATM でお金を出し入れしたりはしますが、それは作業の枝葉の部分であって本質ではないのです。

切符の発行や洗濯の制御や文書の印刷も同じことで、物理的な部分はありますが、それは切符を打ち出す装置や洗濯機やプリンタが受け持つ部分であって、予約を押えたり、どれくらい水をかき混ぜるか決めたり、文書を組み立てるという中心部分はやはり「実体がない」のです。極論すれば、これはコンピュータの出現以前には「人間が頭で」やっていたことで、それをコンピュータが肩代りしてくれているわけです。

これを整理すると、コンピュータがやっていることは、人間が頭でやっていたことのうち、比較的「単純労働な」部分を遂行していると言えるでしょう。これをもう少しかっこよく言えば、コンピュータの定義は次のようになるのではないでしょうか。

○ コンピュータとは、情報を処理するための機械である。

もちろん、この定義が役に立つためには「情報」とは何で「処理」とは何かをもう少し具体的に考えなければなりません。以下の 2 つの節で、これらの点について検討していきましょう。

2.2 コンピュータとデジタル情報

皆様は**デジタル** (digital)、**アナログ** (analog) という用語を聞いたことがあるはずです。いちばん身近なのはおそらく「デジタル時計」(文字で表示される時計) と「アナログ時計」(針が連続的に動く時計) という言葉かも知れません。また、体重計などもデジタル式 (数字で表示されるもの) と、アナログ式 (昔ながらの、針が動くもの) がありますね。

では、デジタルとアナログの区別は何でしょう? 上の例からだと数字と針の違い、ということになりそうですが、もっと一般的に言えば、デジタルとは値が有限個の決まった値のどれか 1 つという形で表されるもの、アナログとは値が連続的に変化し得るもの、ということになるでしょう。デジタル式体重計は「○○○. ○ Kg」という表示窓がついているとすれば、表示できる体重は全部で 10,000 通りしかありません (その代わり、ぱっと見てすぐ分かります)。一方、アナログ式体重計の針の位置はいくらでも細かく区別できます (しかし細かく読み取るのは大変ですし、そんなに正確に計れているのかはまた別の問題です)。そして、コンピュータが扱う情報はすべてデジタル情報なのです。

デジタルな情報の最小単位は「ある」「ない」のどちらか、「はい」「いいえ」のどちらか、「0」「1」のどちらか、といったものだと考えられます。これを「0」「1」で代表させ、**ビット** (bit) と呼びます (bit とは「binary digit」つまり「2 進法の 1 桁」を縮めて作った造語です。また「ちょびっと」という意味の英単語でもあります)。そして、すべてのディジタル情報はこの「0」「1」を沢山並べるだけで表現可能です。

どういう意味か分かりますか。1 ビットすなわち「0」「1」では 2 通りのどちらかしか表せませんが、これを 2 つ並べれば、つまり 2 ビットを使えば、「00」「01」「10」「11」の 4 通りが表せます。3 ビットなら「000」「001」「010」「011」「100」「101」「110」「111」の 8 通りですね (これ以外に無いことを確認してみてください)。では一般には? N ビットあれば、2 を N 回掛けた数、つまり「2^N 通り」の場合のどれであるかという情報が表せます。「$2^1 = 2$」「$2^2 = 4$」「$2^3 = 8$」だということを確認してみてください。では 4 ビットでは? 8 ビットでは?

2.3 0 と 1 による数値の表現

実際には、コンピュータでは数を扱うことも多いので、ビットの列を数に対応させる方法があると便利です。次のように考えてみましょう。図 1 を見てください。この 5 枚のカードのそれぞれの状態を、表だったら 1、裏だったら 0 として、5 枚のカードの状態を 5 ビットで表すものとします。そして、この 5 ビットが表す「値 (整数)」は、見えている丸の個数であるものと定義します。たとえば「01010」だと、「8」「2」のカードが表なので、この 2 つを足した値「十」を表すわけです。この方法で、いくつからいくつまでの整数が表現できるでしょうか? それは何故でしょうか?

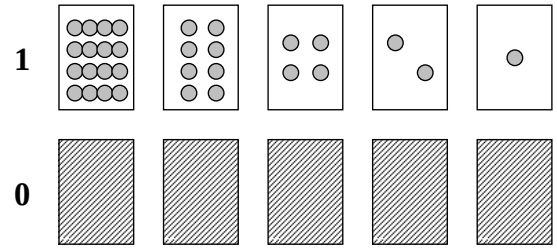

図 1: 5 枚のカード

実は、この方法で数を表すやり方を **2 進法** (binary system) と呼びます。なぜそうなのかを、私たちが普段使っている数の表し方である**十進法** (decimal system) と対比させて説明します。十進法では、数字として「0」〜「9」の十種類を使い、数字を並べて数を表しますね。そして、数を並べて書いたとき、最も右の桁が「1 の桁」、次が「十 ($=(10)^1$) の桁」「百 ($=(10)^2$) の桁」「千 ($=(10)^3$) の桁」と進んで行きます。そして

```
235
```

と書いた場合、これは「百が 2 個、十が 3 個、1 が 5 個」を意味します。3 桁で表せる最も大きい数は「999」で、それより 1 多いのは「1000」つまり次の位の値 1 個 (千) ということになります。

これに対し 2 進法では、数字として「0」「1」の 2 種類を使い、数を並べて書いたとき、最も右の桁が「1 の桁」、次が「2=(2^1) の桁」「4=(2^2) の桁」「8=(2^3) の桁」と進んで行きます。そして

```
101
```

と書いたとき、これは「4 が 1 個、2 が 0 個、1 が 1 個」(ということは 5) を意味します。数字が 0 と 1 しかないので簡単ですね。そして、3 桁で表せる最も大きい数は「111」で、それより多いのは「1000」つまり次の位の値 1 個 (8) ということになります。

なぜコンピュータでは 2 進法を使うのでしょうか。それは、電子回路では「信号のある/ない」だけを区別するようにすることで、回路の設計が単純化され、高速化や高密度化が容易になるからです (実はコンピュータの初期には十進法を使ったものもありましたが、性能的に不利なので消滅しました)。

表 1: 4 ビットのビット列とその値

ビット列	値	16 進	ビット列	値	16 進
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	a
0011	3	3	1011	11	b
0100	4	4	1100	12	c
0101	5	5	1101	13	d
0110	6	6	1110	14	e
0111	7	7	1111	15	f

このように、コンピュータ上では 0 と 1 の並びつまり**ビット列** (bit sequence) によってあらゆる情報を表します。しかし、ビット数が増えて 1 と 0 が何十個も並んで来ると、見るのも書き写すのも大変になります。そこで、4 ビットを 1 つの桁と考えて表 1 の右側にある `0` から `f` までの 1 文字で表す方法がよく使われます (`9` から先は数字がないため、`a`、`b`、`c`、`d`、`e`、`f` を充てているわけです)。これを **16 進** (hexadecimal) 表記と呼びます。たとえば

```
1010 0001 0010 1011
```

であれば

```
 a    1    2    b
```

になるわけであす。なぜ「16 進」かというと、これを数値として見た場合、1 つ桁があがるごとに (2 進法では 4 桁ぶんなので) 値は 16(=2^4) 倍になるからです。そして、「a12b」は値としては

$$\underline{10} \times 4096 + \underline{1} \times 256 + \underline{2} \times 16 + \underline{11} \times 1 = 41253$$

を表すことになります。

2.4 コンピュータで扱う数値の有限性

これまでの議論で、ビット列で表す限り値は飛び飛びであり、連続した値は表せないことがお分かりだと思います。でも、コンピュータで天体の位置のような連続的な数値も計算しているではないか、と思った方もいらっしゃるかも知れませんね。しかし実際には、コンピュータでは決まったビット数を使って計算するので、たとえば十進で合計 8 桁まで使えるとすると (あと適宜符号も加えてよいものとすると)、「−9999.0000 から +9999.0000 まで、0.0001 きざみ」といった形 (範囲も精度も有限) で数値を表さざるを得ないのです。しかしこの方法では「10000」がもうあふれて表現できませんし、小さい数も「0.00001」は表現できません。

ここで 1.2345×10^{23} のような表現方法 (指数表記) を適用することでもう少し柔軟に値を表すことができます。このような数値の表現方法を**浮動小数点** (floating point) と言います。ちなみに、この方法で**仮数** (mantissa) に 6 桁、**指数** (exponent) に 2 桁 (さらに各々に符号ビット) を割り当てたとすると、表せる最も大きな値は「9.99999×10^{99}」、小さい数も「0.00001×10^{-99}」までになります。コンピュータの内部では 2 進法の浮動小数点を使うことで、これと同様にして柔軟性を高めています。しかしそれでも、表せる数に限界があることに変わりはありません。ですから、コンピュータで扱うすべての小数点つき数は「有限の桁数の近似値」であり、さらに「これ以上大きな/小さな値は表せない」という限界もあるものだと考えてください (整数についてはこのような誤差はありませんが、表せる範囲の限界があるという点は同じです)。

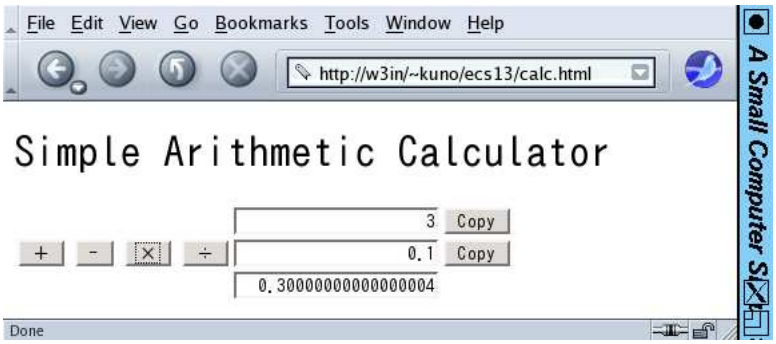

図 2: 簡単な計算ページ

実際にこのことを、JavaScript で作った「簡単な計算ページ」で確認してみましょう (図 2)。JavaScript では計算は上で説明した浮動小数点計算 (ただし 2 進法の) を用いて計算しています。この浮動小数点は IEEE 754 という標準によって定められているもので、今日のほとんどのコンピュータの実数計算はこれにしたがって行われています。

上の欄と 2 番目の欄に数値を入力した後で、「＋」「- 」「×」「÷」のボタンを押すと、2 数に対してその計算をした結果が 3 番目の欄に表示されます。その結果を使ってさらに計算したければ、「Copy」ボタンを押すことで結果欄のコピーを 1 番目/2 番目の欄にコピーして来られます。

やみくもに計算してみる前に、どのような誤差があり得るか考えてみてください。

- 上に説明したように、「決まった桁数の仮数で」計算するので、その桁数ぶんしか精度はありません。たとえば「1234512345 + 0」を計算すると、10 桁の精度があれば答えは「1234512345」ですが、10 桁ない場合はどこかで省略されて「1234512300」みたいになるでしょう (実際は 2 進法でこれが行われます)。これを「丸め誤差」と言います。
- また、十進法と 2 進法の違いが問題になることもあります。たとえば、「0.1」というのは十進法ではキリのいい値ですが、2 進法では無限小数になるので、それ自体に丸め誤差が含まれます。このため、ある数を「10 で割る」のと「0.1 倍する」のとで結果が違うことがしばしばあります。これも丸め誤差の一種です。
- 浮動小数点は非常に大きな数まで対応できますが、それでも限界はあります。たとえば、上で例に挙げた「指数 2 桁」だと、「1.0×10^{50}」と「1.0×10^{50}」を掛けると「1.0×10^{100}」となり、2 桁で表せません。これを**オーバーフロー** (overflow) と言います。同様に、「1.0×10^{-50}」を「1.0×10^{50}」で割ると「1.0×10^{-100}」になり「小さすぎて表せない」ことになります。こちらは**アンダーフロー** (underflow) と言います。通常、オーバーフローが起きると「Infinity(無限)」という特別な値になります。また、アンダーフローが起きたときは (極めて 0 に近い値ということなので)「0」が結果になります。

ちなみに、コンピュータでは指数を表すのに「e」という文字を使い、たとえば「1.0×10^{-12}」であれば「1.0e-12」のように書きます。指数つきで入力したり表示を読む時のために覚えておきましょう。

演習 1-1 「簡単な計算ページ」を用いて、次のいずれか (できれば全部) を確認してみなさい。

a. IEEE 754 浮動小数点形式では、十進に直すとおよそ何桁ぶんの精度 (仮数の桁数) があるでしょうか。

b. IEEE 754 浮動小数点形式では、指数部は十進に直すとおよそ何桁ぶんあるでしょうか。

c. 小数点つきの数の計算をしたとき、一般にどのような値だと誤差があり、どのような値だと誤差なく計算できるでしょうか。

いずれも、「何桁」「どのような場合」を記述したあと、そのことを確認できるような計算例を「2 つ以上」挙げ、なぜその計算例で確認できるのかを説明すること。

「◎」の条件: (1) 小問を 2 問以上解答しており、(2)「2 つ以上の計算例」について適切な (と担当が考える) 考察が書いてあること。

2.5 数値以外の情報のディジタル表現

ところで、ここまでに出て来た例はすべて数値でしたが、コンピュータでは数値以外の情報も表せるのではなかったでしょうか? もちろんそうです。たとえば、コンピュータで英字を表すことを考えましょう。その場合には、決まった長さのビット列を考え、その 0 と 1 のパターンと文字とを対応させます。たとえば次のようにするわけです。

```
01100001  'a'
01100010  'b'
01100011  'c'
...
```

このような、ビット列と文字の対応を定める規則を**文字コード系** (character coding system) と呼びます。なお、上の例は 8 ビットのビット列と英数字記号を対応させる **ASCII** と呼ばれる文字コード系の一部です。8 ビットでは $2^8 = 256$ 種類の文字しか表せないので、漢字などを扱う場合にはもっとビット数の多い文字コード系を使用します。もちろん、1 つの文字ではなくもっと長いもの、たとえば文字の並びとか文章を扱う場合には、ずっと長いビット列を使用します。

では、画像はどうでしょうか? コンピュータで画像を扱う場合には、画像全体を多数の点に分割して、それぞれの点の光の 3 原色 (RGB) の強さを数値として表します。つまり数値の並んだものですから、やはりビット列になります。動画は画像を短い時間間隔で並べたものだと考えればよいですね。音もこれに似ていて、要するに短い時間間隔で信号 (ないし空気の圧力) の強さを計測して数値化し並べることで表せます。

このように、我々が普段接している情報の多くはつまりディジタル情報として (つまりビット列で) 表せます。したがって、これらの情報はコンピュータで取り扱うことができるわけです。

これを従来のアナログ情報と比較してみてください。アナログ情報の時代には、文字を記録するには紙、音を記録するにはテープレコーダ、画像を記録するには写真、動画を記録するには VTR というふうに、全部それ専用の装置が必要でした。これに対し、これらをディジタル情報として扱うのであれば、(人間が直接見るのはアナログ情報の部分ですから) ディジタル情報として取り込む部分、アナログ情報に戻す部分を除けば、全部コンピュータだけで済むわけです。これが、コンピュータが「何でも扱える」理由です。[4]

2.6 コンピュータと情報処理

コンピュータが「情報を処理」する装置であり、コンピュータが扱う「情報」はビット列である、ということは分かりました。ではコンピュータがそれを「処理」する、というのは具体的にはどういうことでしょうか?

なにしろコンピュータが処理するものはすべてビット列なのですから、処理した結果もやはりビット列です。そこで、「処理する」などと難しい言葉を使わずに、もっとひらたく「加工する」と言い替えたらどうでしょう。そうすると、

○ コンピュータとはビット列を加工する装置である

ということになります。つまり、あるビット列を加工して別のビット列にすること、これがコンピュータの行えることのすべてなのです。

具体的には「加工」とはどういうことでしょう? たとえば、ビット列「0010」はこれを 2 進数として見たとき「2」を表し、ビット列「1010」は「10」を表します。そして、これら 2 つのビット列を[5]受け付けて「1100」というビット列を作り出すような「加工」は、つまり足し算という計算をしているわけです。もちろん、四則演算はコンピュータの中では非常によく使われるので、そのようなビット列の加工をするための回路がコンピュータの中に内蔵されています。先に挙げた浮動小数点による演算などについても、同様です。

2.7 コンピュータとプログラム

では、コンピュータの中ではどのようにして「ビット列の加工」を行っているのでしょうか。数値の計算のように基本的なものについては、そのための回路が組み込まれていることは説明しました。しかし、コンピュータに行わせたいような「ビット列の加工」のバリエーションすべてをあらかじめ電子回路として組み込んでおくことは明らかに不可能です。

そこで、コンピュータでは特定の計算を電子回路に組み込む代わりに、電子回路ではごく基本的なビット列の加工だけを用意しておき、それらを後で自由に組み合わせることによってさまざまな加工を行います。しかし、各種の加工を行う回路を「自由に組み合わせる」には、そういう配線を行う必要があるのでは、と思いますね? そこが実は重要なポイントで、現代のコンピュータでは配線を行う代わりに「どう組み合わせるか」を「命令として与える」ことで自由な加工を実現しています。ここがまさに、コンピュータを作り出した人たちの偉大なアイデアなのです。具体的には、次のようにしています (図 3)。

[4]一方、ディジタル化する方法がまだ未開発のものは計算機でうまく扱えません。「香り」などはその代表でしょう。

[5]$1100_{(2)} = 8 \times 1 + 4 \times 1 + 2 \times 0 + 1 \times 0 = 12_{(10)}$

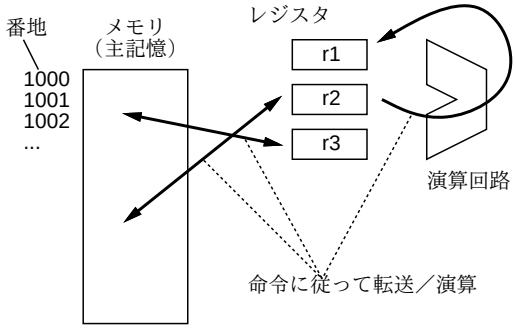

図 3: コンピュータと命令

- すべてのデータは**メモリ** (memory) ないし**主記憶** (main storage) に格納する。メモリには**番地** (address) がついていて、番地を指定してデータを格納したり、取り出して来たりできる。
- CPU はデータをメモリから取り出し、レジスタと呼ばれる記憶場所を利用しながら、さまざまな加工を行い、結果をまたメモリに戻す。
- データの移動や加工はすべて**命令** (instruction) で指定する。

実際には 1 つの命令では簡単な動作 1 つしかできないので、命令を並べてそれを順番に実行していくことで、より込み入った動作を行わせます。この、命令を並べたものが**プログラム** (program) なのです。プログラムもまたメモリに格納されており、データの一種として扱えます。この方式を**プログラム内蔵** (stored program)、ないし考案した科学者の名前から**ノイマン型** (Neuman architecture) などと呼びます。

2.8 小さなコンピュータのシミュレータ

本物の CPU の命令は複雑なので、ここでは簡単化した (架空の)「小さなコンピュータ」を想定して、その命令を動かしてみましょう。この小さなコンピュータは JavaScript で実現されていますが (図 4)、そのことは置いておいて、とりあえず使ってみます。

ここで「プログラム」と記された欄に、1 行に 1 つずつ、命令を書いて行きます。たとえば、次の 7 行のコードを打ち込んでから「実行」してみましょう (**コード** (code) というのは、「プログラムないしその断片」を表す一般的な用語です)。

```
    load X
    add Y
    store Z
    stop
X: 3
Y: 5
Z: 0
```

ここで「load」は、数値をメモリの指定場所 (この場合は X という名前のついている番地) からアキュムレータ (Acc) というレジスタに取り出して来る命令、「add」は、メモリを指定場所 (この場合や Y という名前のついている番地) から取り出し、それを Acc の内容に足し込む命令、「store」は、Acc の内容をメモリの指定場所 (この場合は Z という名前のついている番地) に格納する命令です。これらの命令はいずれも「どこからどこへ」のように 2 つの場所を本来は指定する必要がありますが、その一方は Acc になっているので場所を 1 つ指定すれば済むのです。そして最後の「stop」は、その名前通りプログラムの実行を停止する命令です。

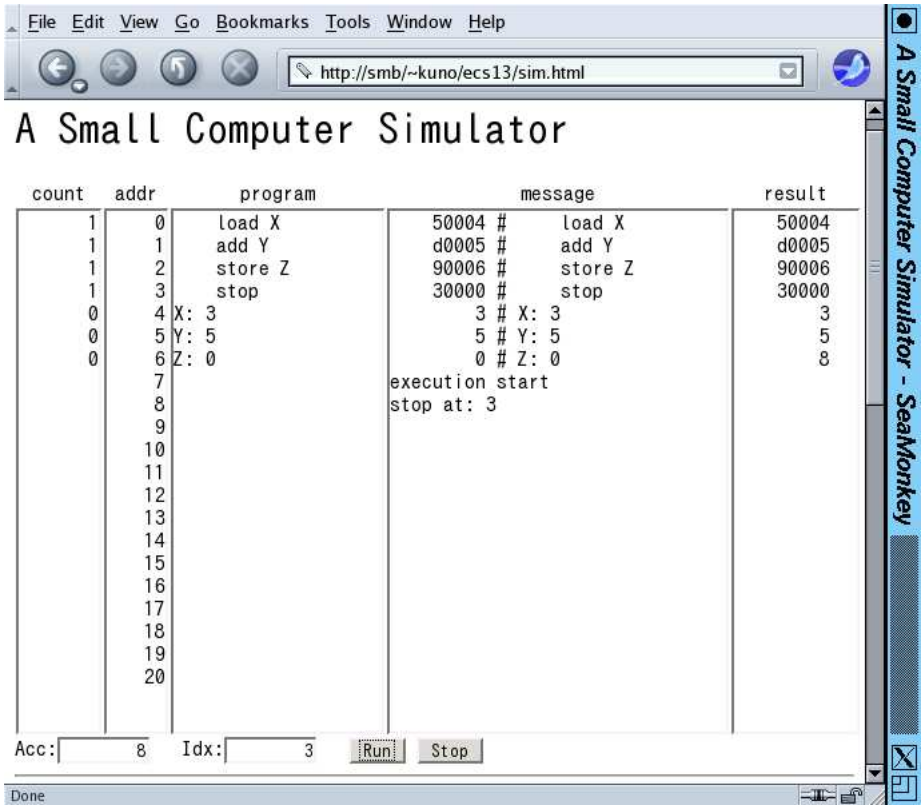

図 4: 「小さなコンピュータ」の画面

プログラムの後には、X、Y、Z という名前のついた場所を用意し、それぞれに 3、5、0 という値を入れておきます。そうすると、プログラムを実行した結果、X と Y の値を足した結果 (8 ですね) が Z の場所に格納され、確かに足し算ができています。

私たちが普段使っているコンピュータでは、画面やマウスなどを使ってデータをやりとりしますが、この「小さなコンピュータ」ではごく基本的な命令しか用意していないので、このようにメモリに直接データを用意して動かすようにしています。

実は、このように命令に (後で出て来るように場所にも) 名前をつけて表すプログラムの書き方を**アセンブリ言語** (assembly language) と呼びます。アセンブリ言語のプログラムは**アセンブラ** (assembler) と呼ばれるプログラムによってビット列、つまり 0 と 1 だけから成るプログラムに変換され、CPU はそれを実行します。この、CPU が直接実行する形のプログラムのことを**機械語** (machine language) のプログラムと呼びます。「小さなコンピュータ」では、実行開始時に機械語 (アセンブラの変換出力) を表示するようになっています。

上の例は「足し算」でしたから順番に命令を 4 つ実行すれば終わりでした。しかし実は、プログラムでは「計算した結果によって処理を切り替える」ということが可能であり、これによって複雑な処理が行えます。そのために、命令の中に「分岐 (ジャンプ) 命令」「条件分岐命令」があります。具体的には、通常の命令はその命令を実行し終わると「次の」命令に進むのに対し、分岐命令は番地を指定し、次はその番地の命令の実行に進むようにさせます。そして条件分岐命令は、「Acc が 0 でないならば」のように条件を指定して、その条件が成り立っている時だけ分岐します (成り立っていなければ、次の命令に進む)。

たとえば、今度は X と Y のうち「より大きい値を」求めてそれを Z に入れることを考えます。そのためには、X から Y を引いてみて、マイナスなら Y の方が大きいと分かります。この考えに基づいてプログラムを作ってみましょう。

```
load X
store Z
```

```
      sub Y
      ifp Skip
      load Y
      store Z
Skip: stop
X:    3
Y:    5
Z:    0
```

「sub」は引き算の命令です。このプログラムではまず、X を Acc に取り出し、とりえあず Z に入れます。次に、Acc の内容から Y を引き算します。ここで、もしプラスなら X の方が大きくて OK ですから、Skip という場所に「飛びます」(ifp は Acc の内容がプラスなら指定した場所に分岐する条件分岐命令です)。プラスでないなら、もう 1 回 Y の値を Acc に持って来て、Z に格納します。いずれにせよ Skip の所に合流して、そこでプログラムは止まります。このように、条件判断して自動的に処理を切り替えることで、コンピュータは複雑な処理が行えているのです。

では、合計に戻って、もっと沢山の数を合計したければどうしましょうか? X、Y、Z…のように沢山変数を並べていてはプログラムが長くなって大変そうです。そこで、Acc のほかにもう 1 つ、インデックスレジスタ (Idx) というものが用意されています。そして、load 命令の拡張版 loadx では「指定した番地より Idx の値だけ先の場所」からデータを取り出すことができます。これを使って、並んだデータを順番に取り出し、合計して行けばよいのです。プログラムを見てみましょう。

```
       iload 0
Loop:  loadx Data
       ifz End
       add Sum
       store Sum
       iadd 1
       jump Loop
End:   stop
Sum:   0
Data:  1
       2
       5
       0
```

Data というラベルの後に数行ぶんのデータがありますが、これを順番に持って来て合計するわけです。

「iload」命令は Idx に指定した値 (この場合は 0) をロードします。次に、load 命令で Acc に値を持って来ますが、最初は Idx は 0 なので、ちょうど Data の場所の値が持って来られます。もしその値が 0 なら、これは終わりの印なので (合計を取りたいのに 0 をデータに入れる必要はないでしょうから)、End へ分岐します。そうでなければ、Acc の値と Sum を加え、その結果を Sum に入れます。続いて、「iadd」命令で Idx を 1 増やしてから、「jump」命令で無条件に Loop へ戻ります。すると、次は Data の次の場所から値が取り出せるわけです。これを繰り返して次々に値を足して行き、0 が現れたら End へ来て止まりますが、そのときには Sum には総計が入っています。

このプログラムの「肝」は、前方向への分岐命令を使って一群の命令列を「繰り返し」実行させることで、短いプログラムでも沢山の処理を行わせられる、というところにあります。これが、今日のコンピュータの重要な原理だと言えます。最後に、この「小さなコンピュータ」が持っている命令の一覧を掲載しておきます (表 2)。[6][7]

[6]条件分岐命令が沢山ありますが、その覚え方は次のようになります。ifz〜「if zero」、ifnz〜「if not zero」、ifp〜「if

表 2: 「小さなコンピュータ」命令一覧

名前	コード	命令の動作
nop	00,01	何もしない
stop	02,03	プログラムの実行を停止
load	04,05	Acc に値を持って来る
loadx	06,07	〃(値/番地に Idx を足す)
store	08,09	Acc の値を格納する
storex	0a,0b	〃(番地に Idx を足す)
add	0c,0d	Acc に値を足す
sub	0e,0f	Acc から値を引く
iload	10,11	Idx に値を持って来る
iadd	12,13	Idx に値を足す
isub	14,15	Idx から値を引く
ifz	16,17	Acc ＝ 0 なら分岐
ifnz	18,19	Acc ≠ 0 なら分岐
ifp	1a,1b	Acc ＞ 0 なら分岐
ifn	1c,1d	Acc ＜ 0 なら分岐
jump	1e,1f	無条件に分岐
neg	20,21	Acc の符号を反転

演習 1-2 「小さなコンピュータ」で以下の処理から 1 つ以上 (できれば全部) 選び、それを実行するプログラムを 1 作成してください。

a. 正負とりまえぜて複数の整数を与えておき (0 が終わりの印)、それらの数値の「絶対値の合計」を求める。データ例:「1 -2 3 -4 5 0」→結果例: 「15」
b. 2 つの整数 (いずれも 0 以上) を与えておき、その 2 つの積 (掛けた結果) を求める。ただし掛け算命令を使わないこと。データ例:「3 5」→結果例:「15」
c. 複数の整数を与えておき (0 が終わりの印)、それらの数値の「最大」を求める。データ例:「-3 -5 -2 0」→結果例:「-2」

いずれにおいても、作成したプログラムに上記の入力例を与えた時、どの命令は何回実行されるか「理由を含めて説明」してください。

「◎」の条件: (1) 小問を 2 問以上解答しており、(2)「各命令の実行回数」について適切な (と担当が考える) 説明がなされていること。

3 コンピュータのハードウェア

3.1 個別素子から VLSI へ

コンピュータの CPU は「電子回路」ですから、もともとは**真空管** (vaccum tube) や**トランジスタ** (transistor) などの個別素子を相互に配線して作っていました。しかし、CPU には非常に多数の素子や配線が必要なので、装置が巨大で高価だったり、信頼性が低いなどの問題がつきまとっていました。

positive」、ifn〜「if negative」

[7]「コード」はその命令を表現するビット列です。すべて 2 つずつコードがありますが、大半の命令はどちらでも動作は同じです。値を取り出す命令 (add、sub、mul、load) のみ、「命令の後に指定した数値を取り出す」「命令の後に指定したメモリの場所から取り出す」の 2 通りに分かれています。

今日ではこの問題は、小さな結晶の上に多数の素子と回路を直接作り込む、**VLSI**(Very Large Scale Integrated-circuit) という技術によって克服されています。その原理を簡単に説明しましょう。VLSI を作るには**シリコン** (圭素、Si) の単結晶のうす切り (ウエハー) を用意します。これ自体は電気を通しませんが、この上に**ホウ素**や**リン**の分子をごく微量加える (**拡散**させる) と、その部分は電気を通すようになります。だから、微細な模様をデザインしてその模様を縮小したマスクに沿って拡散を行えば (図 5)、好きな形の電気配線がウエハー上に作れます。

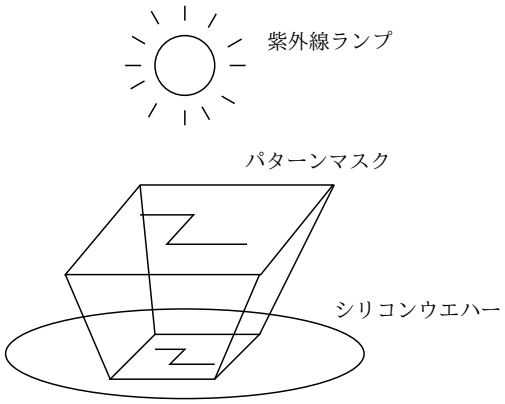

図 5: VLSI の製作原理

さらに、リンとホウ素を拡散した部分が交わったところは、(細かい理屈は省略しますが) トランジスタとして動作します。つまり、多数の細かい模様をウエハー上に作り込むことで、大量のトランジスタを持つ複雑な回路が作れるのです。実際には 1 つのウエハーには同じパターンを (小さく縮小して) 多数焼き付け、それを切り離した個々の**チップ** (chip) が 1 つの VLSI という形になります。

VLSI の何がそんなにすごいのでしょう。それは、個別の素子を使って作るのと比べると、ずっと小さい手間で多数の回路作れること、そして回路を細かくすることで「焼き付けて現像する」という同じ作業のままで同じ 1 つのチップに一層多くの回路を詰め込めるようになります。さらに、技術が進歩して回路が小さくなると、高速で動作させることが可能になります。

1965 年に、ゴードン・ムーア (後の Intel 社の共同創立者) が「ある一定サイズのチップに搭載できる素子数は、毎年 2 倍ずつになっている」という記事を発表しました。具体的な比率はその後、もう少し低い値 (たとえば 18 か月ごとに 2 倍) が正しそうだとされていますが、この「一定期間ごとに倍」という知見は**ムーアの法則** (Moor's law) として知られ、今日でもほぼ成り立ち続けています。

たとえば、VLSI を使って最初に作られた CPU は 1971 年の Intel 4004 ですが、その素子数は 2300 ほどでした。これに対し、2006 年の Intel Core 2 Duo では素子数は 3 億 (!) であり、今日の最先端のチップでは 50 億 (!!) と言われています。

もちろん、最先端のチップを量産するには原理は同じ「焼き付けて現像」であっても、極めて高価な設備が必要です。日本のメーカーはどこも、もはやこの設備投資について行けていないわけですが…

演習 1-3 あなたが過去に使ったことがあったり、よく耳にしたことがある CPU チップを新しいものから古いものまで数個程度選び、その発売年と素子数を調べなさい。その上で、それらをグラフに描き (素子数の方は対数目盛りにする)、ムーアの法則を「検証」してみなさい。なお、それぞれの CPU チップについて、自分との関わり (例:はじめて所有したパソコンだとか) を簡単に説明すること。

「◎」の条件: (1) 適切なデータが調べられていること。(2)「どれくらいの期間ごとに倍」かについてグラフから適切に読み取れていること。

3.2 クロックと同期型回路

皆様は PC などについて「クロックが 4GHz の…」などという言葉を聞かれたことがあると思います。**クロック** (clock) は名前の通り「時計」であってシステム全体を動かすタイミングを制御します。具体的には、図 6 のように決まった周期で 0/1 を反復する出力を出すような回路がクロックです。この反復回数が 1 秒間に 4×10^9 回だと 4GHz となります。この「G」などの略号は国際単位系 (SI) の標準で決まっています (表 3)。なお、クロックの回路はいわゆる発振器で、どちらかというとアナログ回路の部類です。

表 3: SI(国際単位系) の表

倍率	略号	読み	日本語
10^3	K	キロ	千倍
10^6	M	メガ	百万倍
10^9	G	ギガ	十億倍
10^{12}	T	テラ	一兆倍
10^{16}	P	ペタ	千兆倍

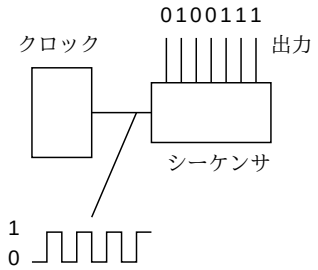

図 6: クロックとシーケンサ

次に、クロックの出力を**シーケンサ** (sequencer) に与えます。シーケンサはクロックのきざみに従って、一定の規則に従って出力を 0/1 に切り替えるものです。これを用いると、複雑な回路を計画した通りに動作させることができます。

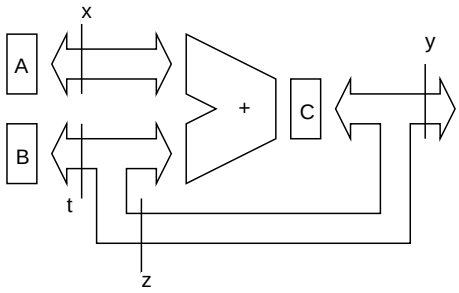

図 7: 簡単な計算システムのダイアグラム

たとえば図 7 のような回路を考えてみます。ここで A、B、C のところに決まったビット数 (CPU の種類によって 8、16、32、64 など) のデータを保持している素子があり、真ん中に演算をおこなう素子群があります。x、y、z、t のところにも素子があって、シーケンサの出力に応じてデータの「通る」「通らない」を切り替えられます。まず x と t をあけると、演算が行われ、結果が C に入ります。次に z と t をあけると、演算結果が B に戻されます。このように、順番に「通る」「通らない」を決まった順で切り替えて行くことで、あちこちにデータを送ったり計算したりができるのです。

このような、クロックにしたがって動作するディジタル回路のことを**同期式回路** (synchronous circuit) と呼びます。今日のディジタル回路は大半が同期式で設計されています。

ところで、同期式回路はクロックに従って動作するわけですから、クロックの刻み速度を倍にすれば、同じ時間で倍の処理ができることになりますね? もちろん、そのためにはクロックだけ速くするのでは済まないので、すべての回路をその速度で間に合うように設計する必要がありますが…

ところが、VLSI の製造技術が進歩するということは、回路が小さくなるということであり、回路が小さくなると遅延も短くなり、クロックが速くできることにつながるのです。たとえば、最初の VLIS CPU である Intel 4004 のクロックは最大 740KHz でしたが、今日の CPU では最速のもので 8GHz くらいまでのものが存在します。

ただ、VLSI でクロック周波数を高めると急速に発熱が増すため、これ以上のクロック周波数の増大は難しい (燃えてしまうから)、というのが今日の状況です。そこで、今日では周波数の増大よりも、回路を高度化したり、1 つのチップに多数の CPU を搭載して並列動作させるなどの形で、処理を高速化する方法が取られるようになっています。

3.3 CPU の構造

話を元に戻して、上に示したようにシーケンサで直接制御線を on/off しているだけでは、決まった動作しかしません。これでは、さまざまな処理を自由に行わせることはできません。

ではどうしたらいいのでしょう? その答えが、「メモリからビット列を読み出してきて、その 0/1 に従って制御線を on/off する」ことなのです。そして、このビット列が「命令」であり、それが並んだものがプログラムなわけです。つまり、CPU は「命令を次々に読み取って、そこに指定された処理を行うことを繰り返す」回路に他ならないわけです。

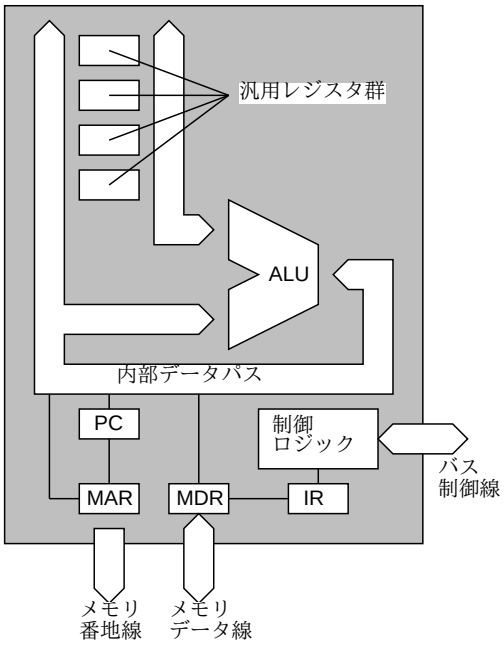

図 8: コンピュータ CPU のブロックダイアグラム

ごく簡単化された CPU のブロックダイアグラムを図 8 に示します。CPU は 1 実行サイクルごとにメモリから命令を読み出し、それを命令レジスタ (IR) にセットします。IR には制御ロジックがつながっていて、これが CPU 内の各箇所にある制御線を on/off して命令の動作を実行します。

命令を読み出す番地は、**プログラムカウンタ** (program counter, PC) と呼ばれるレジスタによって指定されます。PC には 1 命令読み出すごとに内部の値を命令の大きさぶん増やす回路が組み込まれ

ているので、これによって次々と連続した命令列を実行することができます (ただし、ジャンプ命令では命令の動作として PC の値を別の番地に変更します)。

以上をまとめると、CPU の動作の 1 サイクルぶんは次のようになります。

- PC の番地から命令を読み出し、PC を次の番地に進める。
- 命令が読めて来たら IR に入れてその内容を分解する。
- 制御ロジックによって命令の実行を開始する。

これを無限に繰り返すのが CPU の動作です。そして反復動作自体はクロックとシーケンサによって制御されているわけです。

3.4 コンピュータシステムの構造

コンピュータシステムには CPU だけでなく、メモリや入出力装置が備わっているたことを思い出してください。これらは CPU とどのようにしてつながっているのでしょう? これはコンピュータシステムの機種によって多少違いがあるのですが、我々がいちばんよく目にするようなシステムでは図 9 のように**バス** (bus) と呼ばれる信号線がすべての要素を結んでいます。このバスは文字通り乗り合いバスの bus で、コンピュータ内部の部分が共通に信号をやりとりするための配線なのです。[8]

PC などの中を開けると**マザーボード** (mother board) と呼ばれる基板が入っていて、その上には CPU とメモリとバスが搭載されています。そして、バスについてはただの配線なのですが、その上にたくさん端子のついたソケットが数個くっついています。そして、CPU とメモリ以外の要素 (つまり入出力のための回路) はこのソケットに基盤を差し込むことで接続します。こうしておけば、このソケットを抜き差しするだけで、さまざまな入出力装置を増設したり外したりできるわけです。

ところで、CPU の中に「core」と書いてありますが、これは何でしょう。実は今日の PC などに使う CPU では、1 つのチップの中に 2〜8 個の「小さい CPU」が入っていて、それらが同時に処理を行うことで性能を高くしています。これを**マルチコア** (multi-core) と言います。これについては少し後でまた説明します。

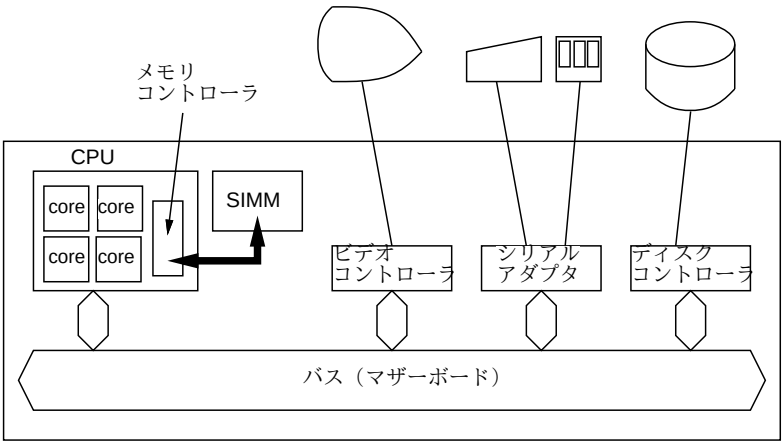

図 9: コンピュータシステムの構造

この、抜き差しする基盤の反対側には、また別のさまざまな形のソケットがついていて、ここと実際の入出力装置 (CRT、ディスク、ネットワークなど) をケーブルやコネクタで接続します。では、基盤の上に載っている回路は何でしょう? これは**コントローラ** (controller) と呼ばれ、バス上の信号と各入出力装置の間の橋渡しを行う機能を持ちます。このようにすることで、CPU からはどの入出力

[8] メモリについては CPU とのやりとりを高速にするため、別の経路で直接 CPU につながっています。

装置でもバス上の同じ信号で制御でき、それぞれの装置に固有の信号の送受はコントローラにまかせることができるのです。

たとえばディスクコントローラであれば、ディスク装置に内蔵されているモータの起動や停止を行ったり、ディスクの特定の位置に記憶されているデータを読み書きする作業を行います。このような高度な機能を持つコントローラはそれ自体が CPU を搭載していてプログラムで制御される、小さなコンピュータになっています。

PC などのシステムにはおおむね、ビデオコントローラ (画面)、シリアルアダプタ (キーボード、マウス)、ディスクコントローラ、メモリコントローラなどが備わっています (メモリは入出力装置ではないが、コントローラを通じてチップのテストやエラー記録の取得などの制御ができることが多い)。

演習 1-4 コンピュータのハードウェアに強い人に頼んで、コンピュータのケースを開けてもらい、その中にどのような部品があるか、どれが何であるかをできるだけ見分けなさい。見分けた部品について、その「大きさ」「搭載位置」「その他気付いたこと」を一覧に示し、全体としてどのようなことが分かるか述べなさい。

「◎」の条件: (1)5 個以上の要素について、適切に見分けられていること。(2) 観察の記述として適切な (と担当が考える) 内容が書かれていること。

4 コンピュータの性能向上とその意義

4.1 ディジタル革命の本質

ここまでで、コンピュータの原理や CPU の機能、ハードウェアの構成について説明してきましたが、結局コンピュータの何がこれほどのインパクトを世の中に与えているのでしょうか? いくつか挙げてみましょう。

- 汎用的なディジタル情報 — 世の中の多くの情報は、ディジタル表現することができ、その結果、コンピュータで扱うことができる。
- 汎用的な処理装置 — コンピュータは、プログラムを取り替えることで、ディジタル情報の「どのような」処理であっても、(その処理の方法が分かっている限り) 行うことができる。
- コンピュータの低価格化 — VLSI 製造技術の発達により、コンピュータはどんどん安価に作ることができるようになってきた。その結果、どこでも専用の機械や電子回路を組み立てて使うより、コンピュータを組み込んでソフトで処理を記述する方が安くて柔軟に処理できるようになった。
- コンピュータの高速化・大容量化 — ハードウェア技術の進化により、これまでは「扱えなかった」「計算できなかった」処理がどんどん実用的な時間でこなせるようになってきている。

4.2 本物の CPU の命令実行速度

ここで、一番最後の「速さ」について、少し調べてみましょう。先にやった「小さいコンピュータ」はソフトで実現しているため、命令の実行速度もごく低いものでした (1 秒間に 100 命令とか)。本物の CPU ではどれくらいの速さで命令を実行しているのでしょうか? 実際に測ってみましょう。まず、10G(百億) 回ループを周回するだけの i386 プログラム (アセンブリ言語) を用意しました。

```
.globl main
main: movl    $1000000000, %eax
loop: subl    $1, %eax
      jg     loop
      ret
```

「.globl main」というのは外部から参照できるラベルを指定するというアセンブリ言語の指示です。そして main というラベルから実行が始まります。最初の movl 命令では、レジスタ%eax に値 1G(十億) を設定します。2 番目の命令 subl では、そこから「1」を引きます。3 番目の命令 jg では、「引き算の結果が 0 より大きければ」loop へ分岐します。なので、subl と jg がともに 1G 回、繰り返し実行されます。最後の ret はこのプログラムを呼び出したところに戻り、プログラムを終了させます。

では、これを動かしてみましょう (以下の演習は Unix 上で行ってください)。まず、上のプログラムを「test.s」というファイルに打ち込みます (打ち込むのには Emacs などのエディタを使います)。なお、最後が「.s」で終わるのはアセンブリ言語のプログラムという意味になります。その後は、次のようにして実行させ、また時間を測ります (「%」は向こうから表示してくるコマンドプロンプトを表すので、打ち込まないように)。

```
% gcc test.s    ←機械語に変換 (アセンブラが動作)
% time a.out    ←時間計測しつつ実行
real     0m0.511s ←プログラムの実行実時間
user     0m0.504s ←プログラムの CPU 消費時間
sys      0m0.001s
%
```

およそ 0.5 秒で 2G(20 億) 命令を実行したわけですから、これを試したマシンの CPU の平均命令実行速度は 1 秒あたりおよそ 4G(40 億) 命令ということになります。また、1 命令あたりの平均所要実行時間は $\frac{1}{4\times10^9} = 25 \times 10^{-11}$ 秒ということになります。

演習 1-5 gcc コマンドの使えるシステム (典型的には GSSM のシステム) で、上の方法で時間計測を行いなさい。その上で、次の課題から 1 つ以上 (できれば全部) 解答しなさい。

a. 計測した CPU の、1 秒間あたりの平均命令実行数を示しなさい。また、1 命令あたりの平均実行所要時間を示しなさい。

b. 上の例では、2 つの命令とも同じ回数だけ実行していた。これを手直しした版 (たとえば、2 回引き算してから条件分岐する) についても同様に計測し、それぞれの命令の平均実行所要時間を求めよ。

c. 上の例では、アキュムレータ (%eax) を使って計算をしていた。代わりにメモリを使った版を示す。これについて計測をおこない、レジスタとメモリで subl 命令の実行速度がどれだけ違うか調べよ。

```
.globl main
main: movl    $1000000000, mem
loop: subl    $1, mem
      jg      loop
      ret
     .data
mem: .long  0
```

いずれも、計測した数値をまず示し、その上で計算式を示し、計算結果に基づいて論じること。

「◎」の条件: (1) 小問のうち 2 問以上を解答しており、(2)「適切な計算式および計算結果に基づいて」解答されていること。

4.3 命令/実行時間/キャッシュ

順番が前後しましたが、演習 1-5 の c について、つまりレジスタの代わりにメモリを使って 1G 回ループするプログラムの実行時間を示しましょう (もちろん、先の例と同じマシンです)。

```
% gcc test1.s
% time a.out
real    0m1.687s
user    0m1.677s
sys     0m0.001s
%
```

4 倍近く遅い!ですね。その理由は、メモリをアクセスする命令はとても時間が掛かるからです。これは、先に学んだように、メモリは CPU とは別の部品でバスにつながっていて、信号の行き来に時間が掛かるためです。これに対し、レジスタは CPU の内部にあるため、レジスタの内容だけを読み書きすれば済む命令は高速に実行できます。この違いが上の違いになっているわけです (図 10 左)。

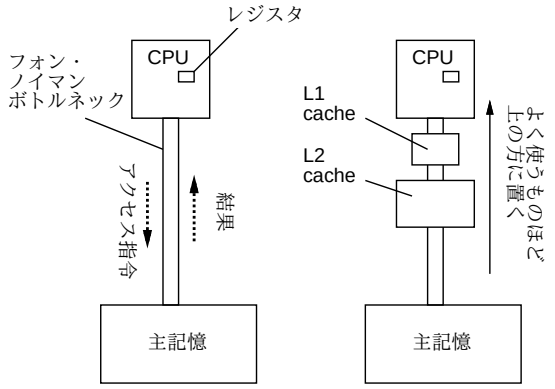

図 10: フォン・ノイマンボトルネックとメモリ階層

このように、今日のノイマン型のコンピュータでは、データやプログラムが全部主記憶に入っているため、そこと CPU の間の転送に時間が掛かることが性能向上の障害になています。これを**フォン・ノイマン・ボトルネック** (Von Neuman bottleneck) と呼びます。これをいくらかでも解消するため、次のような方法が採用されています。

- 命令は基本的に順番に実行されるため、必要な時点よりも早めに CPU に取り寄せておき (先読み)、必要になったらすぐ使えるようにする。この先読みした命令を入れておく場所を**命令バッファ** (instruction buffer) と呼ぶ。
- データについては、CPU のそばに**キャッシュメモリ** (cache memory) と呼ばれる高速な専用メモリを置いておき、主記憶からデータを取り寄せたら「何番地の内容はこれこれ」という形で記憶しておく。その後、CPU から同じ番地の読み書きがあった場合はキャッシュの内容を使って済ませる。

「高速なメモリ」があるのなら、なぜ最初から全部そちらに入れておかないのでしょう? それは、高速なメモリは容量が限られるため、主記憶ほど内容が入れられないためです。

このため、一度 CPU が使った番地の内容はキャッシュに入りますが、実行が進むにつれて次々にキャッシュに内容が取り込まれると入れる場所がなくなり、最近使っていない内容は捨てて (必要なら主記憶に書き戻して) 場所をあけ、新しい内容を入れるようになっています。従って、沢山の番地をバラバラにアクセスするプログラムよりも、少ない番地だけをアクセスするプログラムの方が高速に実行できます。このような性質を**参照の局所性** (locality of reference) と言います。

命令についても、命令バッファのほかに最近使った命令を保持しておくキャッシュがついています。こちらの場合も、参照の局所性がある、つまりごく少ない番地の命令をぐるぐる実行しているプログラムの方が、バラバラにさまざまな番地の命令を実行するプログラムより性能が良くなります。

最近の CPU では、VLSI に多数の素子が集積できるようになったため、CPU チップ上に複数レベルのキャッシュが搭載されています。これらを、CPU に近い側から L1(レベル 1) キャッシュ、L2(レ

ベル 2) キャッシュ、…のように呼びます。この場合、使うデータを入れる場所は次のように順番につながっています。

レジスタ → L1 キャッシュ → L2 キャッシュ → L3 キャッシュ → 主記憶

ここでは、左へ行くほど容量が小さくアクセスが高速、右へ行くほど容量は大きくアクセスは低速になります。このような構造を「メモリ階層」などと呼ぶことがあります (図 10 右)。そして、参照の局所性があるほど、上位階層だけで用が足りるので、プログラムが速く実行できます。

もう 1 つ重要なことは、同じプログラムでもどのように機械語を生成するかで所要時間が大きく変化するということです。このため、以前は「人間がアセンブリ言語で注意深くプログラムを組むのが一番よい」という考えが主流だったことがあります。しかし今日では、コンパイラの技術が進歩したため、人間が一生懸命組むよりもコンパイラで最適化したコードの方が速いというのが普通になっています。

4.4 コンピュータシステムの処理能力

ここまでは 1 秒あたりの命令実行数について考えて来ましたが、そもそもコンピュータシステムの処理能力は 1 秒間あたりの命令実行数で表せるのでしょうか?

1 秒間あたりの命令実行数といっても、命令の種類ごとに実行時間が違うのは上で見た通りですし、CPU ごとにどのような命令群があるかが違うので、同じ同じ仕事をするのに要する命令数も変わります。このため、命令実行数でコンピュータシステムの処理能力を比べることはできません。また、CPU の処理能力だけが高くても、主記憶の速度、ディスク装置などの入出力の速度などが見合うものでないと、実際のシステムとしては性能が出ないという側面もあります。

さらに、WWW サーバなど多数の人の仕事をさばく場合には、1 つのシステムに複数の CPU を搭載した**マルチプロセサ** (multiprocessor) とすることで、全体の処理能力を向上させることができます。このようなケースの場合は、1 つの仕事に対する処理能力ではなく、「一定時間あたりどれだけの量の仕事がこなせるか」の数値を比べる必要があります。このような指標を**スループット** (throughput) と呼び、情報システムの設計では重要な考え方です。たとえば「東名高速で東京から名古屋まで何分で行けるか」が処理能力だとすれば、「東名高速の東京名古屋間は、最大毎時何台の車両を通行させられるか」がスループットだと言えば分かるでしょうか。

これらを総合すると、本当の処理能力を計るには、実際に使いたいシステムを作ってそれを動かして計測するのが一番です。しかしそれではシステムを作る前にハードウェアを選定するといった役に立ちませんから、実際には標準的な**ベンチマーク** (benchmark) を動かしてその数値をシステムの能力のめやすとする、というのが多く行われます。

もちろん、ベンチマークについても「数値計算むき」「汎用処理むき」「データベース/トランザクションむき」など用途によってさまざまなベンチマークが開発されて使われているわけです。

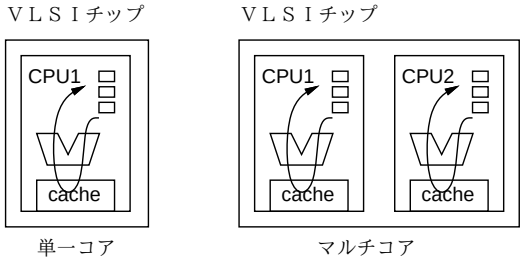

図 11: マルチコア CPU

近年では、CPU の回路自体の性能向上が頭打ち気味なのに対し、1 つの VLSI チップに盛り込める回路量は増大しています。このため、1 つの VLSI チップの中に複数の CPU 機能 (コア) を造り込み、

それらを並列に動作させることで性能を高める方法が普及してきています。これをマルチコアと呼ぶのでした (図 11 右)。

今後のコンピュータシステムでは、ネットワークにより多数のコンピュータどうしがつながって動作する、という側面と上記のような技術で 1 つのシステム内でも複数のプログラム群が並行に動く、という側面が合わさって、これまでのような「単一のプログラムを高速に動かす」形から「多数のプログラムが並列動作し、全体として 1 つの仕事をこなす」形に移行して行くことが考えられます。

ちょっと簡単な実験をしてみましょう。先程の「10G 回足し算」を手元のマシンで動かします。

```
% time a.out
real    0m0.511s
user    0m0.504s
sys     0m0.001s
```

これはさっきと同様ですね。では、手元のとあるマシンで「窓を 2 つ開いて」(できるだけ) 同時に 2 つ動かしてみます。

```
----- 窓 1 ---          ----- 窓 2 ---
real    0m0.859s         real    0m0.834s
user    0m0.509s         user    0m0.504s
sys     0m0.001s         sys     0m0.001s
```

CPU 使用時間 (user) は変わりませんが、実所要時間は倍近くになっています。それはそうで、CPU は 1 つしかないのですから、同時に 2 つ動かそうとすればその CPU が 2 つのプログラムを「掛け持ち」するので所要時間が延びてしまうのです。

ところが、それが「延びない」こともあります。たとえば別のマシンで同じことをやってみます。

```
----- 窓 1 ---          ----- 窓 2 ---
real    0m1.163s         real    0m1.167s
user    0m1.162s         user    0m1.166s
sys     0m0.001s         sys     0m0.001s
```

このマシンは単独の所要時間は先のマシンよりちょっと遅いのですが、2 つ同時に動かしても実所要時間が延びません。それは…このマシンはマルチコアなので 2 つのプログラムが「本当に同時に」動かせるからなのでした。では 3 つ、4 つと動かす数を増やしたら…動かす数がコア数以下であれば、同様に時間は延びず、コア数を超えたら余計に掛かるようになるはずですが、それは各自やってみてください。

今の場合は 2 つのプログラムを別々に動かしていましたが、仕事全体として複数のものに分けて同時に動かせれば、このようなシステムの方が全体として性能が高くなるわけです。

演習 1-6 自分の近辺のマシンで、「10G 回の足し算」を実行し、そのマシンのコア数を推定してみなさい。できれば、複数のマシンで試せるとよい。

「◎」の条件: (1) 計測結果がきちんと掲載されていること。(2) それに基づき、どのような観察からコア数を推定したかが記されていること。(3)2 種類以上のマシンについて実施していること。

5 まとめ

この回ではコンピュータとは何かという話題から始めて、コンピュータシステムの原理と構造について、ハードウェア面を中心に一通り説明しました。さらに、CPU が持つ命令やその実行時間、高速化のための技術についても触れました。コンピュータの歴史は始まってからほんの数十年ですが、その間にその性能、大きさ、価格などは驚異的に (望ましい方向に) 変化しており、その結果が今日の「ディジタル革命」となっているわけです。