計算機科学基礎'11 ＃1 – 計算機とは/ハードウェア/性能評価

久野　靖*

2011.4.13

1　計算機とその位置付け

1.1　計算機とは

まず、「**計算機**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?

△ 計算機とは、計算をするための装置である。

もしそうだったら、計算機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計算機である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、現在の計算機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計算機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。計算機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。

*経営システム科学専攻

[1]イギリスで作成された暗号解読用の電子計算機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、「弾よりも速い計算機」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。

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、これは計算機の出現以前には「人間が頭で」やっていたことで、それを計算機が肩代りしてくれているわけです。

これを整理すると、計算機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、計算機の定義は次のようになるのではないでしょうか。

○ 計算機とは、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。

1.2 計算機とデジタル情報

皆様は**デジタル**、**アナログ**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、計算機が扱う情報はすべてデジタル情報なのです。[4]

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。[5]

ビットが最小単位なのですから、デジタル情報すなわち計算機が扱う情報とはひらたく言えばビットの並び、**ビット列**だということになります。そして、ビット列の長さを長くすることで、いくらでも多くの場合を区別することができます。たとえば 1 ビットでは 0 と 1 の 2 通りしか区別できませんが、2 ビットでは 4 通り、3 ビットでは 8 通り、一般に N ビットでは 2^N 通りの場合が区別できます。

ビット列によって数値を表す場合、**2 進法**を使います。この場合、一番右の桁が「1 の桁」、すぐ左の桁が「2 の桁」、その左が「4 の桁」、「8 の桁」、…というふうに桁が進むごとに表す値が倍々になります。たとえば、表 1 は長さ 4 のビット列とその 2 進数としての値を対応させたものです。この中の「1010」がなぜ 10 かというと、

$$\underline{1} \times 8 + \underline{0} \times 4 + \underline{1} \times 2 + \underline{0} \times 1 = 10$$

だからなのです。

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** 表記と呼びます。たとえば

[4] 厳密に言えばアナログ計算機というものもあるのですが、一般には計算機といえばデジタル計算機のことを指します。

[5] bit とは「binary digit」つまり「2 進数の 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)。なぜ **2 進数**かというと、**10 進数**では 1 つの桁は 0〜9 のどれか、8 進数では 1 つの桁は 0〜7 のどれかになりますが、これと同様にして、2 進数では 1 つの桁は 0〜1 のどれか、つまり「0」「1」のどちらかになるからです。

表 1: 4 ビットのビット列とその値

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

```
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値は 16 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。

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、計算機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、計算機では決まったビット数を使って計算するので、たとえば「-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。

実際には 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**」と言います。コンピュータの内部でもこの方法を使って柔軟性を高めていますが、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、計算機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)。

実際にこのことを、簡単な C 言語のプログラムで確かめてみましょう。この授業では C 言語の細かい書き方は扱わないので、「見よう見まね」で理解してください。

```
/* mulsub.c --- simple calculation on 'double' data */
main() {
  double x, y;
  printf("x> "); scanf("%lg", &x);
```

```
  printf("y> "); scanf("%lg", &y);
  printf("x*y = %20.18lg\n", x*y);
  printf("x/y = %20.18lg\n", x/y);
}
```

このプログラムは次のことをします。

1. /* ... */はコメント (注釈) で、動作には関係しません。
2. main() { ... } がプログラムの実行される範囲。
3. double(倍精度実数) 型の変数 (データ領域)x, y を用意。
4. 「x>」と表示し、x に値を入力。
5. 「y>」と表示し、y に値を入力。
6. 小数点以下 18 桁表示で x*y(掛け算の結果) を表示。
7. 小数点以下 18 桁表示で x/y(割り算の結果) を表示。

このプログラムを動かすには次のようにします。

1. mulsub.c というファイルに上記のプログラムを保存。なお、「l」は小文字の L であり数字の「1」ではないことに注意。
2. 「gcc mulsub.c」でコンパイル (機械命令に変換)。
3. 「a.out」で実行開始。

複数回実行したければ一度変換した後は 3 だけ何回でもやればよいです。やってみましょう。

```
% gcc mulsub.c
% a.out
x> 1
y> 2
x*y =                   2  ←掛けたら確かに 2
x/y =                 0.5  ←割ったら確かに 0.5
% a.out
x> 1
y> 3
x*y =                   3  ←掛けたら確かに 3
x/y = 0.333333333333333315  ←あれ?
% a.out
x> 1e20                     ←「e20」は「10 の 20 乗」の指数部指定
y> 2e20
x*y = 2.00000000000000006e+40  ←あれ?
x/y =                 0.5  ←こっちは正しい…
%
```

「あれ?」と思うことが多々ありますが、これは先に述べたように「有限桁数」の計算で精度が限られていることから来ています。もっと簡単に言えば、**計算機での実数計算は近似値での計算に過ぎない**と言うことです。もっとも、計算のつど下位を「丸め (四捨五入)」するので、その結果「たまたま」正しい値になることもあります。

1.3 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、計算機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、計算機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**コード系**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? 計算機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、これらの情報は計算機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、全部計算機だけで済むわけです。これが、計算機が「何でも扱える」理由です。[6]

1.4 計算機と情報処理

計算機が「情報を処理」する装置であり、計算機が扱う「情報」はビット列である、ということは分かりました。では計算機がそれを「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?

なにしろ計算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、

○ 計算機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、これが計算機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[7]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

[6]逆に言えば、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。「におい」などはその代表でしょう。

[7]$1100_{(2)} = 8 \times 1 + 4 \times 1 + 2 \times 0 + 1 \times 0 = 12_{(10)}$

るわけです。もちろん、四則演算は計算機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計算機の中に内蔵されています。

別の典型的なビット列の加工演算として、**and 演算**というものがあります。これは、受け付けた 2 つのビット列の対応する位置にともに 1 があるときだけ、その位置に 1、それ以外には 0 があるような結果を作り出します。たとえば「01100001」と「11110000」という 2 つのビット列の and 演算の結果は「01100000」というビット列となります。

```
     01100001
and) 11110000
-------------
     01100000
```

しかし、そんな簡単なことをわざわざ計算機なるものを使ってやるまでもないのでは、と思われる方もいるかも知れません。もちろん上の場合だけならそうでしょうが、計算機によるビット加工では「データ (ビット列) の内容に応じた処理内容の変更や繰り返し」が可能であり、これによって人間が手でやるのはとても困難な「加工」が実現できます。

たとえば、ある 0 以上の数 Y の平方根を求める、という問題を考えます (もちろん、Y は計算機の中ではビット列で表現します)。これを計算機で (効率は悪くても) 行うには、たとえば次のようにすればできます。

- 数 X を 0.0 とする。
- X^2 を計算してみて、これが Y 以上なら X が求める答え。
- そうでなければ、 X に 0.00001 を足したものを新たに X とする。
- そして、ふたたび X^2 を計算してみて、これが Y 以上なら X が求める答え。
- そうでなければ、 X に 0.00001 を足したものを新たに X とする。
- そして、ふたたび X^2 を計算してみて、...(以下同様)
- ...

人間だったらとてもこんなことをやる気にはならないでしょうが、計算機ではこれが可能ですし、十分実用的でもあります。

しかし、これでは求まる値は近似値であって正確ではないって? もともと、計算機の中で扱える数値はすべて有限の桁数の近似値であると先に説明しました。ですから、どの程度まで正確な近似値を求めるか (これは調整可能です) を決めて、その正確さで計算するので十分ですし、それ以上のことはできない (し、やろうとしても意味がない) のです。[8]

これが、計算機でディジタル情報を扱う時の注意点です。つまり、アナログ情報や我々人間が考える「理想的な」情報は、ディジタル情報に (つまり「とびとびの値に」) 変換するときに、細かい部分が捨てられます。このことは常に注意を払っている必要があります。

また、同様のことは情報の処理についても言えます。計算機は非常に高速にビット列を加工することができますが、そのためには「どのように加工するか」ということが厳密に定まっている必要があるのです。言い替えれば、コンピュータはやり方が決まったことを大量/高速に、繰り返しこなすのを得意とします。一方、人間が行うような「大まかな印象を見てとる」「感じ取る」といった曖昧な作業は苦手です (「印象の計算方法」なるものを無理矢理厳密に決めて計算することはできますが、そうして計算したものは人間の感覚とはかけ離れていることが多いでしょう)。

[8] そもそも多くの数の平方根は無理数ですから、小数点以下何百桁を表示しても、それで「きっかり正確」ということはあり得ないわけです。

1.5 計算機とソフトウェア

では、計算機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。計算機にはそれが行える電子回路は備わっていますが (その構造についてはすぐ後で説明します)、計算機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。

たとえば、2 つの数値を入力して、1 つ目のものを 2 倍してから 2 つ目のものを引く (数式で書けば $2X - Y$) ものとします。このような電子回路を組み立てることは簡単ですが、それ「しか」できないのでは計算機とは言えません。では、さまざまな計算を組み込んで、切り替え可能にしておけばよいでしょうか? 実際には、「2 倍ではなく 3 倍にしたい」とか、「Y が正の数の時に限って引きたい」とか、やりたい計算 (ビット列の加工) には無限のバリエーションがありますから、これらすべての計算をあらかじめ用意しておくことは不可能です。

そこで、計算機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 (四則演算やビットごとの and 演算など) 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任意のビット列加工が行えるようにしてあります。

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、現代の計算機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、計算機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 1)。

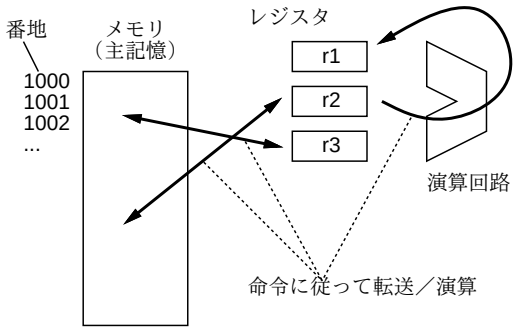

図 1: 計算機と命令

- すべてのデータは**メモリ** (**主記憶**) に格納する。メモリには**番地**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。
- データを加工するには**レジスタ**と呼ばれる場所に (メモリから) 持って来て演算命令を使う。加工が終ったらまたメモリにしまう。
- 持って来たり、取り出したり、演算するのは**命令**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**なのです。

たとえば、数値 X が 1000 番地、数値 Y が 1004 番地に格納されていたとします。それに対して $2X - Y$ の計算を行って結果を 1008 番地[9][10]に格納したいとすれば、次のような命令列を使えばよいのです。

[9] 番地が 4 飛びになっているのは、1 つの数値を入れるのに 4 つの連続した場所が必要—具体的には、1 つの番地には 8 ビットが格納でき、1 つの数値は 32 ビットで表す—な計算機を想定しているからです。

[10] 実際には入力装置からデータを受け取ったり結果を出力装置に送り出したりする命令が必要ですが、それは略しました。

```
2000  load 1000,r1
2004  add r1,r1,r2
2008  load 1004,r3
2012  sub r2,r3,r4
2016  store r4,1008
```

この命令列で起こることは次のとおりです。

1. 1000 番地に格納されているデータ (2 進数表現のビット列) をレジスタ 1 番に持って来る。
2. レジスタ 1 番の内容とレジスタ 1 番の内容を加算して (つまり 2 倍になる)、結果をレジスタ 2 番に入れる。
3. 1004 番地に格納されているデータをレジスタ 3 番に持って来る。
4. レジスタ 2 番のデータからレジスタ 3 番のデータを引き算して、結果をレジスタ 4 番に入れる。
5. レジスタ 4 番のデータを 1008 番地に格納する。

面倒くさいですが、このような命令列をいろいろに変えることでビット列の加工方法はどうにでも変えられるわけです。

しかし、命令列 (プログラム) とは具体的にはどんなものなのでしょうか? 紙に書いてあっても、それを計算機が直接扱うことはできませんね? 上の命令列の左側に数値が書いてあるのに気がつきませんでしたか。実はこれはメモリ上の番地です。つまり、「命令もビット列で表し、計算機のメモリの中に格納しておく」わけです。こうすることで、計算機に次のような性質を持たせられます。

A. 命令列を入力装置からメモリに読み込んで来ることで、自由に計算機の動作を設定できる。
B. メモリの内容は命令で書き換え可能 (その番地を指定して格納命令を使えばよい) だから、命令を変更したり作り出したりするようなプログラムも存在できる。

この原理を**プログラム内蔵方式**とよび、フォン・ノイマンの発明だとされています。[11]この発明のおかげで、計算機は「プログラムを取り換えることで何でもできる機械」になったわけです。

なお、計算機の装置本体を**ハードウェア**、そこに読み込ませるプログラムやプログラムが必要とするデータなどを**ソフトウェア**と呼ぶことが一般的です。装置は金属などでできた「硬い」もの、命令列やデータはビット列であり、形のない (柔らかい?) ものなので、そう呼ぶようになったようです。

しかし、上の命令列の例ではどうして「判断や繰り返し」ができるのか釈然としないかもしれません。もう 1 つの例として、平方根を求める命令列も考えてみましょう。こんどは数値 X を 3000 番地、Y を 3004 番地として、X は最初 0 になっているものとします。

```
4000  load 3000,r1
4004  load 3004,r2
4008  mul_f r1,r1,r3     ※1
4012  if_gt_f r3,r2,4028
4016  load 5000,r4
4020  add_f r1,r4,r1
4024  j 4008
4028  store r1,3000      ※2
...
5000  0.00001
```

この説明は次のとおりです。

- 4000 番地の内容をレジスタ 1 に持ってくる。

[11]ただし計算機の黎明期における発明・発見についてはさまざまな異説があります。

- 4004 番地の内容をレジスタ 2 に持ってくる。
- (※ 1)X^2 を計算して結果をレジスタ 3 に入れる。
- レジスタ 3 とレジスタ 2 の内容を比較し、レジスタ 3 の内容の方が大きければ次は 4028 番地 (※ 2) へ行く。
- レジスタ 4 に 5000 番地の内容 (0.00001) を持って来る。
- レジスタ 1 とレジスタ 4 の内容を加え、結果をレジスタ 1 に入れる。
- 次は 4008 番地 (※ 1) へ行く。
- (※ 2) レジスタ 1 を 4000 番地に格納する。

このように、「次に実行する命令の番地を指定する命令」(**ジャンプ命令**) や、「値の条件に応じてジャンプする命令」(**条件ジャンプ命令**) を使うことによって、繰り返しや処理内容の枝分かれが自由に行えるのです。

なお、上の命令の末尾に「`_f`」とついていたのは何でしょう? それは、これらの足し算命令や比較命令が「ビット列を小数点つきデータとして扱う」命令なのでこれまでの足し算命令や比較命令と違う名前にしたのです。つまり、ビット列が整数を表しているか、小数点つきの数を表しているかは、人間 (プログラムを書く人) にしか分からないので、きちんと区別して命令を使う必要があるのです。なお、`load` や `store` はビット列を転送するだけだから (ビット列の長さが同じ— ここでは 32 ビットであるものとしました — である限り)、命令を区別する必要はありません。

2 計算機ハードウェアのしくみ

2.1 ビットの表現 option

まず、電気回路ではどうやってビット列を扱うのでしょう? 直観的に分かりやすい例として、ビットを入力するのにはスイッチを用い、出力するのにはランプを用いるものとします。図 2 のように配線すれば、入力 A を 1(ON) にすれば出力 B も 1 になるし、入力が 0(OFF) なら出力も 0 のままです。

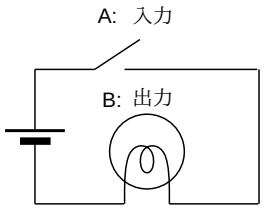

図 2: ランプとスイッチによる入力と出力

これでは出力が入力そのまんまだからあまりおもしろくありません。そこで、入力を A と B の 2 つとして、「A と B が両方 1 のとき出力 C が 1」および「A と B の少なくとも一方が 1 のとき出力 C が 1」、つまり **and 演算** と **or 演算**にしてみたのが図 3 です。何のことはない、理科の実験でやる「直列と並列」ですね。

では、この方法で回路を組み合わせればビットのどんな加工でもできるのかというと、残念ながらそうは行きません。早い話が「入力 A が `1` のとき出力 B が `0`」つまり not 演算すら作れないのです[12]。

[12] スイッチについている表示を書き換えて「`0`」と「`1`」を反対にしてしまうという手もありますが、ちょっとずるいですし、これで not 演算を作ったとは言えないでしょう。

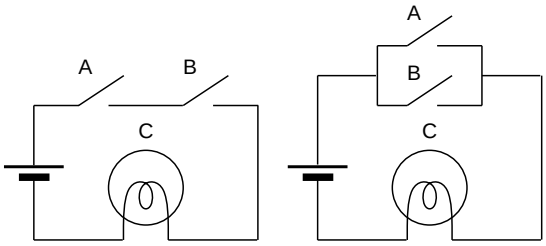

図 3: ランプとスイッチによる and と or

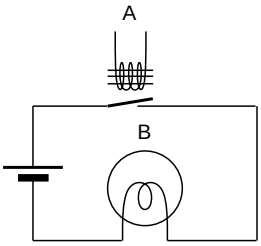

図 4: リレー回路

2.2 演算素子とゲート option

そこで次に、**リレー**を使うことにします。リレーというのは手のかわりに電磁石で on/off するスイッチのことで、ここでは図 4 のようにふだんはバネの力でスイッチが閉じていて、入力回路に電流が流れると電磁石の力でスイッチが切れるようなものを使うことにします。これで確かに not 回路をつくることができました。

そのほかの種類の演算もリレーで行わせることができます。たとえば図 5 のように 2 つの入力回路 A、B の両方に電流が流れた時だけスイッチが切れるようにバネを調節しておけば、 not (A and B) の演算 (**nand 演算**と呼ばれる) を行わせることができます。

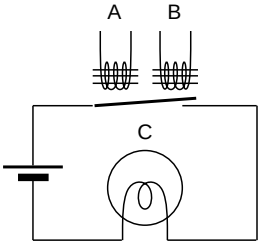

図 5: リレーによる nand 回路

組み合わせ回路が複雑になってくると、いちいちリレーの形を書くのは煩雑です。そこで、もっと抽象化した記号を用いて、電池の両極につながる線も省略して on/off される線だけを書くことにしましょう。具体的には図 4、図 5 を図 6 のように書きます。このような抽象化された演算素子のことを**ゲート**と呼びます。図 6 では上のものは「not ゲート」、下のものは「nand ゲート」を表しています。

ところで、リレーのもう 1 つの重要な性質は、エネルギーを増幅できることです。一般に 1 つのスイッチに流すことのできる電流は限られているので、スイッチ 1 個にたくさんの電球をつなぐことはできません。また、スイッチには固有の抵抗がありますから、100 個の入力の and をとろうと思っても、100 個のスイッチを全部直列につないだら十分な電流が流れない可能性があります。しかし、リレーがあれば少ない電流で大きな電流を on/off できるので、これらの問題を解決することができ、十

分高度なビット加工を行う回路を組み立てることができます。実際、電子計算機ができる以前に、リレーを演算素子とした計算機 (リレー計算機) が作られていたこともあります。

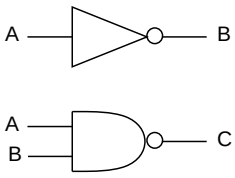

図 6: 抽象化された回路記号

2.3 VLSI option

リレー計算機はスイッチの鉄片がかちゃかちゃ動くのに時間がかかりますし、接触不良が多く信頼性が低いという問題もあります。そこでこれらの問題を解決して、高速かつ信頼性の高い情報処理を可能にしたのが**電子計算機**です。同じ電子計算機でもゲートに使われる演算素子は真空管 (第 1 世代)、トランジスタ (第 2 世代)、IC(第 3 世代)、VLSI (第 4 世代) と変遷してきています。ここでは現在使われている素子である **VLSI** の原理をごくおおざっぱに説明しましょう。

まず、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、微細な図 7 のような模様をデザインしてその模様にそって拡散を行えば、好きな形の電気配線がウエハー上に作れます。

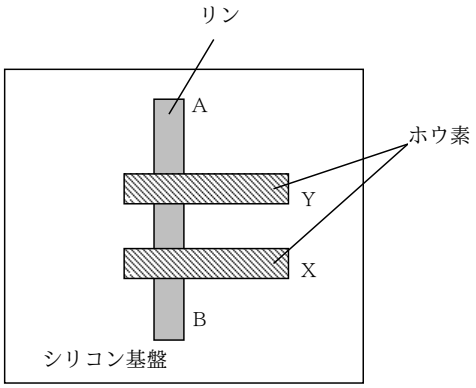

図 7: VLSI の構造

ここで、リンを加えた部分は (リンがシリコンより多くの電子を原子の外周に持つため) 電子が余分に生じて電気を流すという性質を持ち (**n 型**領域)、逆にホウ素では (シリコンより外周の電子が少ないため) 電子の不足 (正孔) が生じて電気を流すという性質を持ちます (**p 型**領域)。ここで、断面図が図 8 のように、n 型領域が途切れていて間に電極 (これもパターン焼きつけで作れる) を取り付けた形になっている部分を作ります。すると、電極が「+」の時にはそのプラス電荷に引き付けられて来た電子がギャップの所に集まって電気が通り、電極が「-」の時には逆に電子がマイナス電荷と反発して逃げてしまうので電気が通らなくなります。

これでつまり、電極の+/-に応じてスイッチが on/off できるのです。このような半導体素子を**電界効果トランジスタ** (FET) と呼びます。半導体素子はリレーと違って動く部分がありませんし、微細な領域で低電圧で動作させられるため高速です。なお、n 型の代りに p 型を使えばこれと反対の性質を持つ (つまり電極が「-」の時に電気が通る) トランジスタも作れます。今日では、その両者をペアに組み合わせて回路を構成する **CMOS** 型 VLSI が主流になっています。

ところで、図7は実はNANDゲートの一部になっています。なぜだか分かりますか? 実際のNMOS回路では、点Aを抵抗を介して電源(+極)につなぎ、また、点Bは直接アース(- 極)につなぎます。点Aから点Bまでの間の回路は、XとYの端子がともに「+」のときに導通します。そのときは、点Aはアース(- 極)と同じ電圧になります。それ以外、つまりXかYか少なくとも一方が「- 」だとAB間は絶縁されていますから、A点は抵抗を介してつながる「+」側になります。これはNANDゲートの動作ですね。このようにして、微細な配線で多数のゲートを作り、それらをつないで回路ができるわけです。

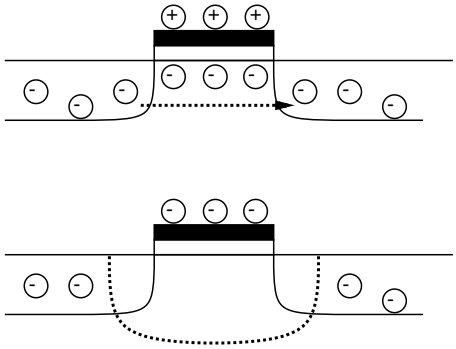

図 8: VLSI 上の電界効果トランジスタ

では、どうやってウェハーの上に微細な模様に従って拡散を行わせるのでしょう? それにはまず、模様を普通の大きさで作って、それを写真に撮ってフィルムを作ります。次に、ウェハーの上に光に感応する樹脂を塗り、フィルムを通した光をレンズで縮小して投射すると、光があたった部分だけ樹脂が変質します(図9)。その後で洗浄液につけると、変質した部分だけが流れ落ちるので、結果として模様通りの幕がウェハーに残ります。幕のついたウェハーをホウ素やリンがガス化した炉に入れると、幕がなくなった部分に模様にそって分子が拡散します。実際にはまずホウ素、次にリンという風にこれを何段階か繰り返して、非常に多数のトランジスタを持つ回路をウェハー上に焼き込むのです。これをVLSIといいます。

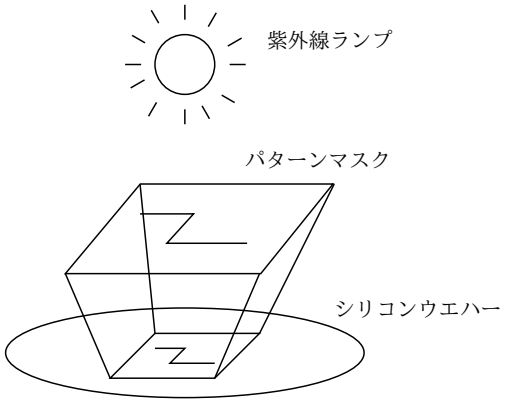

図 9: VLSI の製作原理

2.4 フリップフロップとラッチ option

もう少し具体的に計算機に使われる回路を見てみましょう。図10にフリップフロップと呼ばれる回路を示します。この図で一番左の状態をまず見てください。A、Bはともに1、上のゲートのもう

1 方の入力は 1 だから上のゲートの出力 C は 0、従って下のゲートの片方の入力が 0 だから下のゲートの出力 D は 1、そこで上のゲートの入力が 1 で、つじつまが合っています。[13]

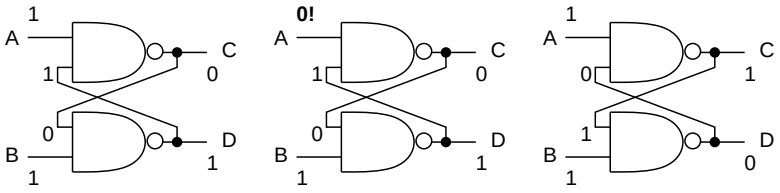

図 10: RS フリップフロップ

ここで、図中央のように入力 A をちょっとの間 0 にしたとしましょう。すると、上のゲートの出力は 0 → 1 に変化し、その結果下のゲートの出力は 1 → 0 に変化することになります。そして、入力 A が再び 1 に戻った後も、この回路はさっきとは逆に C が 1、D が 0 という状態を保持します。つまり、フリップフロップは与えられた情報 (最後に A と B のどちらが 0 だったか) を保持することができるのです。

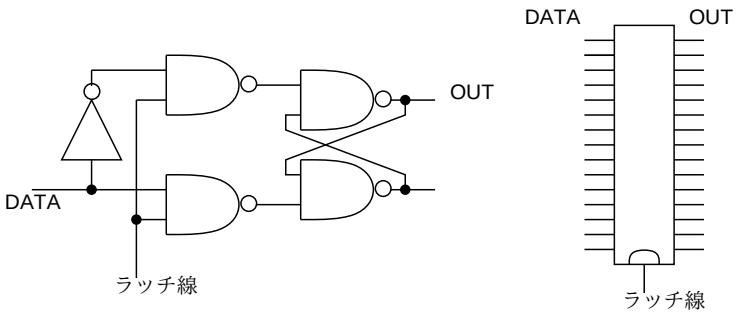

図 11: RS フリップフロップ

さて、実際に計算機の回路としてこれを使う場合には、図 11 左のように**ラッチ線**と呼ばれる信号線を持つ前段ゲートを追加して使うのが普通です。この場合、ラッチ線を普段は 0 にしておきます。すると前段のゲート出力は 1 なので、OUT の状態は変化しません。入力を記憶したい場合にはラッチ線を 1 にします。すると DATA 線の状態がフリップフロップに記憶され、OUT は DATA と同じになります。ラッチを 0 に戻すとその状態は固定され、あとは DATA が変化しても記憶された状態が保持されます。このような回路を「入力をそのまま固定する」ことから**ラッチ** (latch、掛け金の意味) と呼びます。通常はこのような回路をデータ幅 (32 ビット CPU なら 32 個) ぶん並べて、1 本のラッチ線で制御します。(これを図 11 右のように描きます。)

2.5 メモリ option

計算機がビット列を加工していく途上では、大量のデータを扱う必要があります。そこで、もっぱら大量のビット列を格納しておくことだけを行う VLSI チップが多く作られ売られています — これを**メモリチップ**といいます。実は 1 章で出て来た主記憶 (メモリ) は、このメモリチップをたくさん並べたものです。

メモリチップの中で実際にビットを格納する部分 (**メモリセル**) には、前述のフリップフロップが使えます。この方式は電源を切らない限り特に何もしなくても記憶内容が静的に保持されるので **SRAM**(static random access memory) と呼ばれます。SRAM は 1 セルあたり 4 つのゲートが必要であり、記憶容量の点からは不利です。もう 1 つの主要な方式は **DRAM**(dynamic RAM) であり、

[13] もしつじつまが合わないと、回路は不安定になり、発振 (状態が振動すること) します。

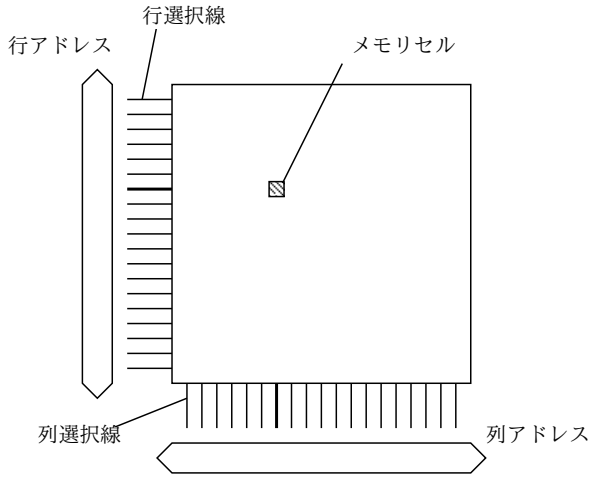

図 12: メモリチップの構成

これは小さなコンデンサ (電荷を蓄えるような構造) を使って記憶を保持します。ただし、コンデンサに蓄えた電荷は時間がたつと自然放電してなくなってしまうので、定期的に蓄えた値を読み出して再度書き込む必要があります。これを**リフレッシュ**と呼び、DRAM のチップ上にはそのための回路が組み込まれています。

DRAM でも SRAM でもメモリチップの内部は概念的には図 12 のようになっていて、メモリセルが格子状に配列されています。このなかで特定のセルをアクセスするためには、行と列の選択線をそれぞれ 1 本だけ「1」にすることで、その交点にあるセルが選ばれてアクセスされます。

実際には選択線の外側にある**アドレス線**に「0110」「0111」などのビットパターン (メモリ番地の 2 進数表現) を与えることで、セルを選択します。この、アドレス線から選択線のどれか 1 つを選ぶ部分をアドレスデコーダと呼びます。図 13 に、アドレス線が「0101」の時だけ出力が 1 になる回路を示しました。[14]

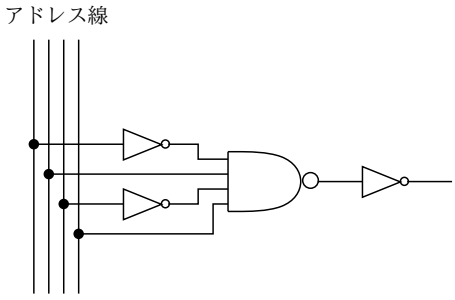

図 13: アドレスデコーダ

標準的な DRAM では 1 チップの容量が 1〜4Gbit 程度 (1G = 1024 × 1024 × 1024) で、実際には 1 つのアドレスを指定すると 4 ビットが同時に読み書きできるといった構成を取ります。このようなチップを 8 個とか 16 個、小さな板の上に取り付けたものを **SIMM**(single inline memory module、両面に付けたものは **DIMM** — dual inline memory module) と呼び、本物の計算機ではこれをソケットに差し込んで計算機本体に取り付けたものが主記憶になるわけです。[15]

[14] N 入力の nand ゲートは VLSI 上では図 7 のようなパターンを少し拡張することで簡単に作れます。

[15] 8 個とか 16 個というのは、計算機のメモリは 32 ビットとか 64 ビット単位でアクセスされることが普通なので、複数のメモリチップを並べておいてそのビット数ぶん同時にアクセスするようになっているものです。メモリチップ 1 個が 4 ビット幅であれば、16 個並べると 64 ビット幅で同時に読み書きできます。

2.6 演算回路 option

データを記憶しておく回路ができたので、次に演算を施す回路を考えましょう。一般に、n ビットの入力から任意の演算を行って 1 ビットの出力を行う場合、その演算規則を「積和標準形」つまり入力のうちいくつか (およびその not) に and 演算を行った「項」を複数 or する、という形で現すことができます。

表 2: 2 桁どうしの足し算

A_1	A_0	B_1	B_0	C	O_1	O_0
0	0	0	0	0	0	0
0	1	0	0	0	0	1
1	0	0	0	0	1	0
1	1	0	0	0	1	1
0	0	0	1	0	0	1
0	1	0	1	0	1	0
1	0	0	1	0	1	1
1	1	0	1	1	0	0
0	0	1	0	0	1	0
0	1	1	0	0	1	1
1	0	1	0	1	0	0
1	1	1	0	1	0	1
0	0	1	1	0	1	1
0	1	1	1	1	0	0
1	0	1	1	1	0	1
1	1	1	1	1	1	0

例えば 2 進数 2 桁の足し算を考えましょう。入力 (2 桁 $\times$2) と出力 (2 桁および桁上がり 1 桁) の関係を現すと表 2 のようになります。

ここで 3 つの出力をそれぞれ積和標準形で現すと

$$
\begin{aligned}
O_0 &= A_0\overline{B_0} + \overline{A_0}B_0 \\
O_1 &= \overline{A_0}A_1\overline{B_1} + \overline{A_0A_1}B_1 + A_1\overline{B_0B_1} + \overline{A_1B_0}B_1 + A_0\overline{A_1}B_0\overline{B_1} + A_0A_1B_0B_1 \\
C &= A_1B_1 + A_0A_1B_0 + A_0B_0B_1
\end{aligned}
$$

となります。なお、AB は A と B の and、$A+B$ は A と B の or、$\overline{A}$ は A の not を表します。

これを論理回路にするには、例えば図 14 のような規則的なやり方が可能です。ここではまず、各入力からそのままと not したものを作ります。次に、各論理式の項 (and でつながったもの) について 1 個ずつ nand ゲートを用意し、その入力を対応する入力線 (またはその not) につなぎます (N 入力の nand ゲートが簡単に作れるという説明は先にしました)。そして、各出力に対しても nand ゲートを用意し、各項に対応するゲートの出力をここに接続します。

このようにして、任意の入力と出力の関係を実現する論理回路を組み立てることができます。足し算、引き算など複数の演算を切り替えたければ演算の種類を指定する制御線も入力として扱い、それを含めた論理式から設計すればよわけです。このような回路は、ゲートが規則的に配置されているので**ゲートアレイ**と呼ばれます。

ただし、大きな桁数の加算や乗算を一段で行わせようとすると回路が巨大になってしまい、また実際の計算機で使われる演算器は高速性が要求されますから、本当の演算器 (**ALU** — Arithmetic

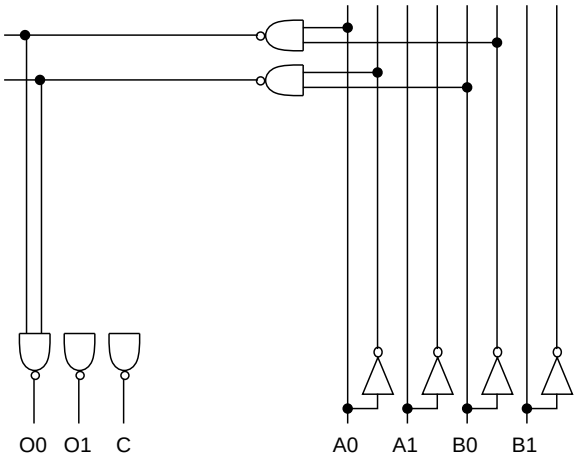

図 14: ゲートアレイによる論理回路

Logic Unit) はこのような規則的な設計とは別の方法で構成した論理回路を用います。

2.7 クロックとシーケンサ option

ここまでのところで一応、データを格納したり演算したりする部品はできました。しかし、これらの部品を組み合わせて思ったように動かす方法についてまだ説明していませんね。

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**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 15 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)。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。

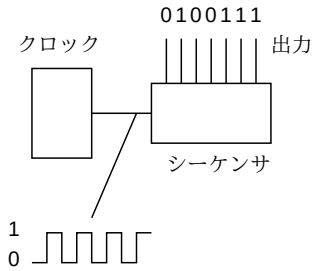

図 15: クロックとシーケンサ

次に、クロックの出力を**シーケンサ**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 (その回路までやっていると大変なのでここでは略します)。

さて、これでどうやって部品が組み合わせられるのでしょうか? たとえば図 16 のような回路を考えてみます。ここで A、B、C はラッチで真ん中のは加算回路です。そして、たて線のところはデータ線の接続を on/off するゲートが設けてあります (データ線の幅は 4 ビットでも 32 ビットでもなんでも構いません)。かりに A、B に入力装置からデータをセットできたとしましょう。ここで B の値に A の値の 2 倍を加えた値を最終的に出力するには、

- まず x と t を on にしてラッチ C に $A + B$ を計算する。
- t と y を off、z と x を on にして C の値と A の値を計算したものを再度 C に設定する。

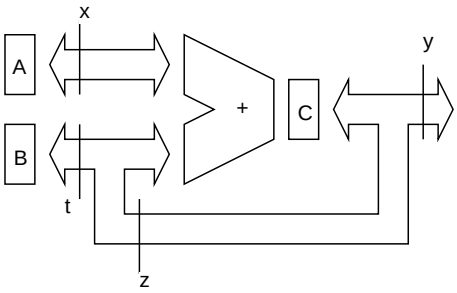

図 16: 簡単な計算システムのダイアグラム

- 最後に z を off、y を on にして C の値を出力する。

という順序で制御を行えばよのです。それは、シーケンサが x〜t の 4 本の線をクロックのタイミングに従って on/off することで行えるわけです。

2.8 プログラムと CPU

さて、前節最後に示した様な方法では、せっかく組んだ回路に、ある特定の計算しか行わせることができません。これではもったいないですね。ではどうしたらいいのでしょう? 同じような質問を既にしましたが、ここでは電子回路の面から考えてみます。その解答は…各制御線の on/off をシーケンサ回路として作ってしまう代わりに、「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」に他なりません (先の説明では命令はそれらしい文字列で書かれていましたが、実際には命令の種別、レジスタの番号、メモリ番地などを符号化したビット列の形でメモリ上に格納されるわけです)。

また、並んだ一連の命令が順番に実行されると説明してきましたが。これは、ある命令を実行し終ったら、メモリの次の場所にある命令を持って来るようにすればよいわけです。これを繰り返すことで、メモリに書いてある命令語の列、すなわちプログラムを順番に実行していくような回路が作れます。これが計算機の **CPU**(中央処理装置) の動作の本質なのです。

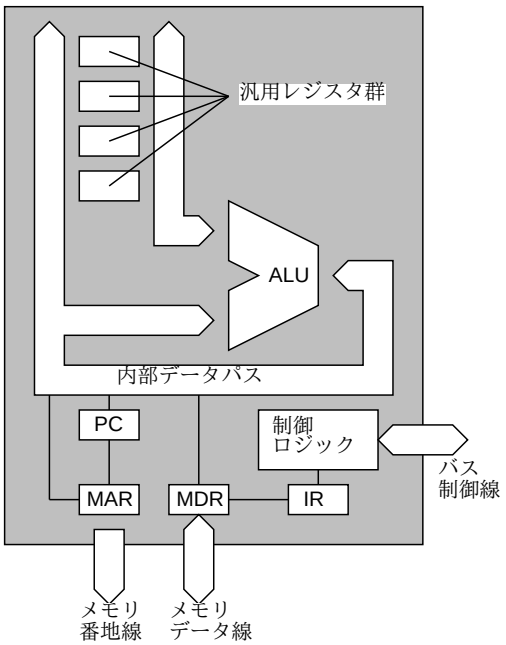

図 17: 計算機 CPU のブロックダイアグラム

ごく簡単化された CPU のブロックダイアグラムを図 17 に示します。ここで、メモリアドレスレジスタ (MAR) にメモリアクセスを行いたい番地を入れて CPU から外部に送り出すと、メモリアクセス (読み出しまたは書き込み) が行えます。読み出しの場合は、メモリから読まれたビット列はメモリーデータレジスタ (MDR) に格納されます。書き込みの場合には、MDR の内容がメモリに転送されます。

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。命令を読み出す番地は、**プログラムカウンタ** (**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。

なお、CPU 内部の制御線すべてを命令のビットと対応させると 1 命令がひどく大きくなってしまうので、普通の CPU では命令はもっと「詰めあわせた」形をしています。これを適当に分解して各制御線を on/off するのが**制御ロジック**の役割です。CPU 内部には、先に出て来たように、データとしてのビット列を入れておくためのレジスタ群もあります。これを**汎用レジスタ**と呼びます。命令の種類としては、汎用レジスタとメモリとの間でデータをやりとりするもの、汎用レジスタの値を ALU に送って演算し結果を再び汎用レジスタに格納するもの、次に実行する命令の番地を設定するもの、などがあります。

最後の種類のものでは、命令の実行が連続した列から「飛び出して」別の位置に移ることになります。これが先に説明した**ジャンプ命令** です。ジャンプ命令の実行とは単に、PC に次の命令の番地を設定することです。これらはすべて、CPU 内部のデータパスを通してデータを転送することで行えます。

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 の内容を MDR に転送して命令の読み出しを開始し、PC の値は次の番地に進める。
- 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。

2.9 計算機システムの構造

計算機システムには 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? これは計算機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 18 のように「**バス**」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、計算機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。[16]

PC などの中を開けると**マザーボード**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**と言います。これについては少し後でまた説明します。

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

[16] メモリについては CPU とのやりとりを高速にするため、共通のバスとは別の、専用のバスで CPU とつながっている場合も多い。

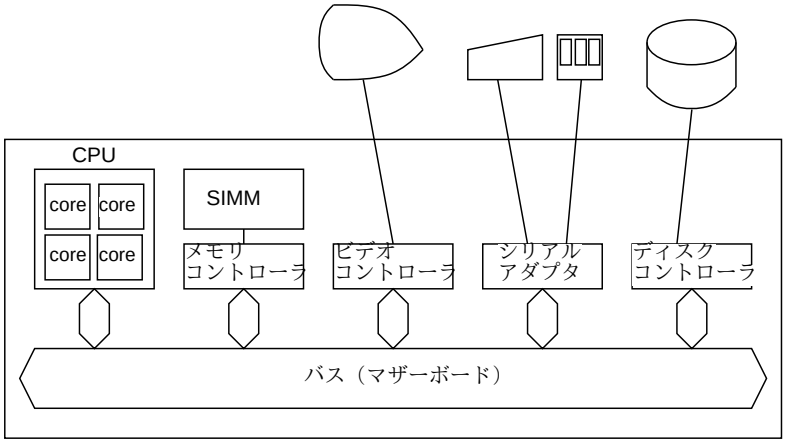

図 18: 計算機システムの構造

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**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、ミニ計算機になっています。

PC やワークステーション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)。

3 アーキテクチャと性能評価

3.1 命令セットアーキテクチャ

再び命令の話に戻りますが、CPU がどのようなレジスタ群や命令を持ち、それぞれの命令がどのようなビット列で表され、どのように動作するかの決まり一式を**命令セットアーキテクチャ**(Instruction Set Architecture、ISA) と呼びます。今日の PC の多くは Intel の IA-32 アーキテクチャを採用していますが、そのほかに IA-64、SPARC、PowerPC、ARM など多くのものがあります。

なぜこのように多数の命令セットアーキテクチャがあるのでしょうか。それは 1 つは「歴史的事情」(各メーカが独自のアーキテクチャを開発して優劣を競った) ですが、もう 1 つにはアーキテクチャの設計によって得手不得手が変わるからです。たとえば汎用 PC には性能を高くしやすいアーキテクチャが採用されますが、モバイル機器には多少遅くても消費電力が小さく、プログラムをコンパクトにしやすいアーキテクチャが有利です。

命令セットアーキテクチャの主要な分類基準として、次のものがあります。

- レジスタ数 — CPU 内でデータを扱うのにレジスタを使いますが、この面で「少数のそれぞれ固有の機能を持ったレジスタ」を備えるアーキテクチャと「多数の互換性のあるレジスタ」を備えるアーキテクチャがあります。後者を**汎用レジスタ**方式と言います。汎用レジスタ方式ではレジスタは「0 番」「1 番」など番号で呼びますが、専用レジスタ方式ではレジスタはそれぞれ固有の名前を持ちます。現在では汎用レジスタ方式が主流ですが、IA-32 は歴史的事情により専用レジスタ方式です。

- CISC と RISC — **CISC**(Complex Instruction Set Computer) とは、プログラムで活用しやすいように多数の命令を備えるタイプのアーキテクチャです。Intel のアーキテクチャがその代表です。これに対し、**RISC**(Riduced Instruction Set Computer) とは、どうせ機械語はコンパイラが生成するのだから、命令の種類数を減らして単純にし、その分クロック周波数などを高めて性能を向上させようというタイプのものです。ただし、今日では技術の進歩により (たとえば Intel のチップはアーキテクチャは変えずに内部で RISC 的な設計を採用しています)、性能面での違いはなくなっています。
- **データ幅** — 主要なレジスタのビット数により、8 ビット、16 ビット、32 ビット、64 ビットなどのアーキテクチャがあります。IA-32 は 32 ビット、IA-64 は 64 ビットです。ビット数が多いと一度に多くの情報が扱えますが、コストは高くなりますし、たとえば 32 ビットのデータをおもに扱うのなら 64 ビットのレジスタが有利とは言えないわけです。

1 つの命令セットアーキテクチャについても、その実装 (つまり CPU チップ) は複数種類作られるのが普通です。たとえば IA-32 についても Intel から Pentium III、Pentium IV、Pentium M などのチップでクロック周波数が違うものが複数種類ありますし、AMD や Cylix など他の企業も IA-32 アーキテクチャの CPU を製造しています。これはアーキテクチャが同じであれば、ソフトが同じで済むので、用途や値段に応じて違う CPU を採用してもソフトを変えなくて済むためです (ただし、OS までそのまま使うには命令セットだけでなく入出力まわりも同じである必要があります)。

3.2 命令/実行時間/キャッシュ

では、実際に IA-32 の命令セットを見てみましょう。まず、次の C 言語プログラムを見てください (たった 4 行です):

```
main() {
  int i = 0;
  while(i < 500000000) i = i + 1;
}
```

C 言語の細かい知識はさておいても、これは「i という変数をまず 0 にして、それから 5 億回、1 ずつ増やして行く」というものだというのは見れば大体分かると思います。

これをコンパイルして、アセンブリ言語コードを観察します。それには、まず上の 4 行をたとえば「count.c」というファイルに Emacs などのエディタで打ち込み、続いて次のようにしてコンパイルし、アセンブリ言語コードを表示させます。

```
% gcc -S count.c
% cat count.s
(先頭部分省略)
        movl    $0, -4(%ebp)
.L2:
        cmpl    $499999999, -4(%ebp)
        jg      .L3
        leal    -4(%ebp), %eax
        incl    (%eax)
        jmp     .L2
.L3:
        leave
        ret
(以下略)
```

ここで「-4(%ebp)」が変数 i の場所に相当します。movl 命令は変数 i に 0 を入れ、次に cmpl で i と 499999999 を比較し、もし i の方が大きければ jg でラベル.L3 へジャンプして終わります。そうでない場合は leal で変数 i の番地をレジスタ%eax に持って来て、次の incl でその番地の内容を 1 増やし、jmp で.L2 つまり比較の直前へジャンプします。結構読めるでしょう?

さて、このプログラムの実行時間はほとんどが cmpl から jmp までの 5 命令の実行に費されます (なにしろそれぞれ 5 億回実行されるわけですから)。では、この 25 億命令の実行に何秒掛かると思いますか? 実際にやってみましょう。それには、アセンブリ言語コードをさらに機械語に変換し、時間計測コマンドの下で実行させます。

```
% gcc count.s
% time a.out
real    0m0.894s
user    0m0.891s
sys     0m0.001s
```

ここで「user」の次に表示されているのがこのプログラムを実行させたときの CPU 使用秒数です。およそ 0.9 秒、ですから 1 命令の実行時間はおよそ 25 億分の 1 秒ということになります。

ところで、この機械語プログラムはかなり無駄があり、コンパイラの最適化機能を ON にすることで、もっと効率のよいコードを生成してもらえます。それもやってみましょう。

```
% gcc -S -O count.c
% cat count.s
(前略)
        movl    $0, %eax
        .p2align 2,,3
.L4:
        incl    %eax
        cmpl    $499999999, %eax
        jle     .L4
        leave
        ret
(以下略)
```

今度は変数 i をメモリ上に置くかわりに、レジスタ%eax を変数 i として使用するコードになっています。最初に movl でこのレジスタに 0 を入れ、次の命令はジャンプの速度を早めるためにラベル.L4 の位置を調整するもので、その後 incl でレジスタを直接 1 増やし、cmpl で 499999999 と比較し、まだ以下なら jle で.L4 へジャンプします。これにより、ループ内の命令数は 3 命令となりました。では実行時間はどうでしょうか?

```
% gcc t01.s
% time a.out
real    0m0.300s
user    0m0.297s
sys     0m0.001s
```

15 億命令で 0.3 秒、つまり 1 命令あたり 50 億分の 1 秒になりました。命令によって全然実行時間が違うことが分かりますね?

その理由は、メモリをアクセスする命令は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 19 左)。

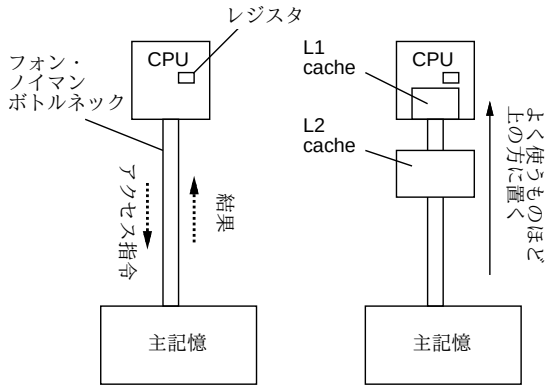

図 19: フォン・ノイマンボトルネックとメモリ階層

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**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**と呼ぶ。
- データについては、CPU のそばに**キャッシュメモリ**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**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、キャッシュメモリは通常のもの (CPU とは別のチップになっているもの) のほかに、CPU チップの上にもさらに高速なものがあって、さらによく使う少数のデータを保存するようになっていることが普通です。この場合、CPU にくっついているものを L1(レベル 1) キャッシュ、メモリとの間にあるものを L2(レベル 2) キャッシュと呼びます。

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 → 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 19 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。

3.3 CPU の高速化技術 option

ここまで、ずいぶん「速い」にこだわって来ましたが、なぜそんなにこだわるのでしょうか。それは、計算機の場合は「速いものは遅いものを兼ねる」ので、コストに見合う限り、できるだけ速いものを量産して低コストで供給する方が競争力があるからです。

また、速さがあってはじめて価値のある計算機の用途というのも多数あります。天気の数値予報 (計算機シミュレーションで将来の天気の状況を予測する) などがその代表です。24 時間後の天気を計算するのに 48 時間掛かったのでは役に立ちませんから。

では、CPU の速さは何によって決まるのでしょうか。先に 1 秒間当たりの命令実行数を計測しましたが、その値は概念的には次の式で求まります:

毎秒命令実行数 = クロック周波数 × CPI × 並列度

これらについて説明しておきましょう。

- クロック周波数 — これは回路技術によって決まる部分と、CPU の設計によって決まる部分とがあります。
- CPI(Clock Per Instruction) — 1 命令あたり平均何クロックで実行できるかをあらわす値。
- 並列度 — ALU などを複数備えることで、1 つの CPU 上で、複数の命令を同時に実行する技術があります。常に 2 命令ずつ実行できれば、他の値が同じだとして、1 命令ずつ実行する CPU の 2 倍の性能になるわけです。

一般に CPI とクロック周波数の間にはトレードオフがあります。つまり、1 クロックの間に多くの作業を行うように CPU を設計すると、少ないクロック数で 1 命令が実行できますが、クロック周波数は高めにくくなります。

この問題を克服して CPI を高める技術の 1 つに、**パイプライン化**があります。これは 1 つの命令を複数のステージに分けて実行し、隣接する命令の実行をステージ単位で重ね合わせる方式です。具体的には、CPU が個々の命令を実行するときはいくつかのステージを順番に経ることになります: [17]

- 命令フェッチ — 命令を取り出す。
- 命令デコード — 命令を解読し、実行の準備をする。
- データリード — 命令が使うデータを取り出す。
- 実行 — 実際に命令の動作を行う。
- ライトバック — 実行結果をレジスタやメモリに反映させる。

ここで ALU が必要なのは演算を行う実行ステージだけですから、素朴に命令を 1 つずつ実行していると、5 ステージのうち 4 つまでは ALU は遊んでしまいます。命令取り出しや解読など他の回路も同様です。そこで、パイプライン化により命令を重ね合わせることで、各回路を連続して稼働させ、CPI を高めるわけです (図 20)。

ただし、5 ステージパイプラインにしたからといって、CPI が 5 倍になるわけではありません。なぜなら、命令どうしの間には依存関係があるものがあり、ある命令の実行結果が出ないと次の命令が実行に入れないものがあるからです。典型的には、比較命令の後の条件ジャンプ命令などがそうです。このような場合は、パイプライン内で次の命令の実行を遅延させ、結果が出るのを待ちます (**パイプラインハザード**)。ハザードが多いと CPI は低下するわけです。

[17] CPU の設計によって、ステージの段数やそれぞれの機能は変わってきます。

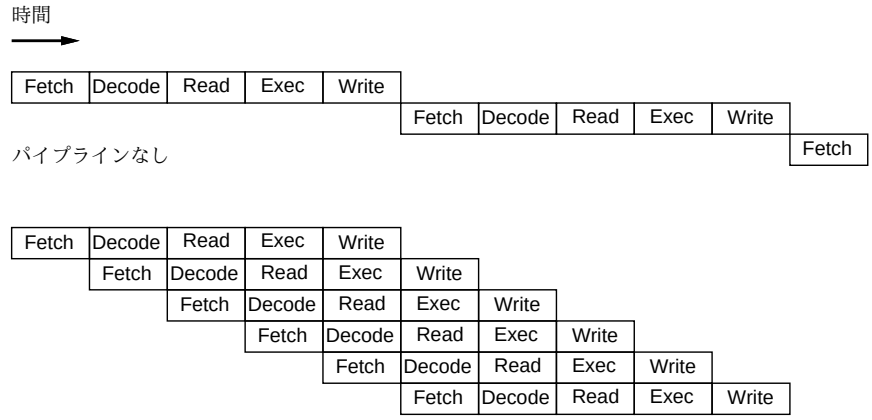

図 20: パイプライン化による性能の向上

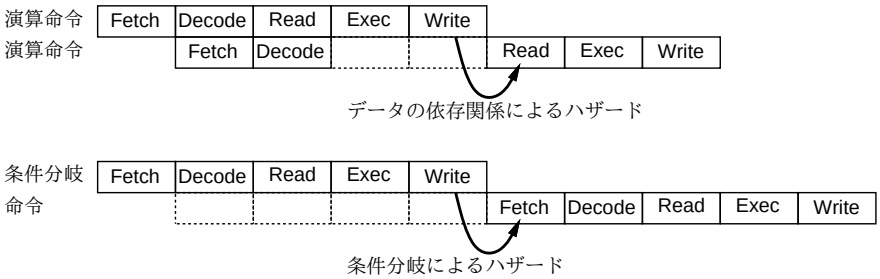

図 21: パイプラインハザード

並列実行についても同様です。内部で並列実行を行う CPU の場合、ALU などの回路が複数あって、それらを同時に使用することで命令実行数を稼ぎますが、ある命令と次の命令で依存関係があると、それを並列に実行するわけには行かなくなります。さらに、依存関係があるのに間違って並列に実行してしまうと、正しくない結果が得られてしまいますが、それを防ぐことの方がより重要であり、しかも難しいのです。このような並列実行の制御方式には、CPU のハードウェア中に依存関係の制御と複数命令の発行を行う回路を含む方式 (**スーパースカラー**) と、コンパイラが機械語を生成するときにあらかじめ依存関係を調べておき、依存関係のある命令が一緒に実行されないように並べ換える方式 (**VLIW**) があります。VLIW(Very Loing Instruction Word) 方式ではコンパイラが並列度の分だけの命令をくっつけた長い命令を出すので、命令セットアーキテクチャが独自のものになりますが、スーパースカラーではそのような必要はないので、現在の CPU ではスーパースカラーの方が一般的です。

3.4 計算機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、そもそも計算機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、命令セットアーキテクチャが違えば同じ仕事をするのに要する命令数も変わりますから、命令実行数で計算機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 (**マルチプロセサ**) 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**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。

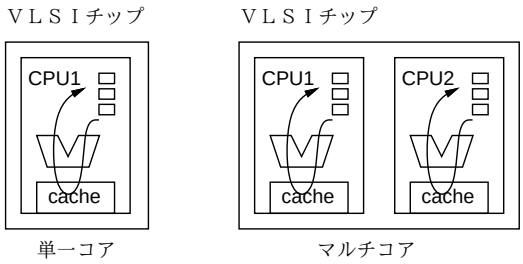

図 22: マルチコア CPU

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**と呼びます (図 22 右)。

今後の計算機システムでは、ネットワークにより多数の計算機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「5 億回足し算」を手元のマシンで動かします。

```
% time a.out
real    0m1.043s
user    0m1.008s
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 ---
% time a.out
real    0m1.916s
user    0m1.008s
sys     0m0.001s
----- 窓 2 ---
real    0m1.803s
user    0m1.008s
sys     0m0.000s
```

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 ---
% time a.out
real    0m1.163s
user    0m1.162s
sys     0m0.001s
----- 窓 2 ---
real    0m1.167s
user    0m1.166s
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。

4 まとめと演習問題

この回では計算機とは何かという話題から始めて、計算機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。

1-1. 「掛け算と割り算のプログラム」を動かしてさまざまな値を計算させ、「どのような場合に誤差があるか/ないか」を検討してみなさい。また、「10 で割る」のと「0.1 を掛ける」のに違いがあるかどうか予想し、実際に確認してみなさい。なぜそうなのかも考えてみてください。

1-2. 計算機のハードウェアに強い人に頼んで、計算機のケースを開けてもらい、その中にどんな部品があるか、どれが何であるかの「同定」を試み、どのような部品はどのようだったか (思ったのとどう違ったかも含めて) 報告しなさい。

1-3. 我々が使う PC に入っているような CPU チップ単体の値段はいくらくらいか予想し、ネット検索により調べてみなさい (円でもドルでもよい)。さらに、何によって価格が違ってくるのかをまとめなさい。資料として参照したサイトの URI を明記すること。

1-4. 自分の手元のマシンで「5 億回の足し算」の計測を実行し、平均命令実行時間を算出しなさい (注: 平均命令実行時間＝総所要時間÷総命令実行数。総命令実行数＝ループ内命令数×ループ回数)。最適化の有無によってどれくらい違うかも報告すること。

1-5. 手元のマシンで「5 億回の足し算」を複数 (2 個、3 個、…) 同時に動かす計測を行い、自分の手元のマシンのコア数がいくつかを調べてみなさい。コア数が 1 だった場合は、さらに sma/smb/smc で (他の人が計測していない時に) 同様に計測してみなさい (注: コマンド `uptime` を実行してみて負荷が 0.00 なら CPU を消費するプログラムは動いていないと判断できます。)

1-6. 計算機システムの性能評価ベンチマークとしてどんなものがあるか、ネット検索により調べてみなさい。資料として参照したサイトの URI を明記すること。