コンピュータリテラシ＃4 – コンピュータの動作原理

久野 靖 (電気通信大学)

2017.5.19

1　今回の目標

今回の目標は次の通りです。

- 整数の表現 (符号なし、2 の補数) とその操作について理解する — コンピュータ上の演算が持つ特性について知っておくことで実験や研究のための計算での失敗を減らすことができます。
- コンピュータの基本的な動作原理について理解する — コンピュータの動作原理を知ることで何が得意/不得意かという的確な判断ができ研究がスムーズに進められます。
- 機械語 (アセンブリ言語) レベルの簡単なプログラミングを体験する — プログラムを動かしてみることでコンピュータの特性が具体的に理解できます。

2　コンピュータとデジタル情報

2.1　デジタル情報とビット exam

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、ということになります。

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)。つまり数字で表したものは常にデジタルです。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。そして、すべてのデジタル情報はこの「0」「1」を沢山並べるだけで表現可能です。

たとえば、現在の天気を「雨が降っていない」「雨が降っている」の 2 通りの場合に分けたとすると、その情報をたとえば次のように 1 ビットの情報として表すことができます:

ビット表現	意味
0	雨が降っていない
1	雨が降っている

1 ビットはデジタル情報の最小単位ですが、複数のビットを並べた**ビット列**とすることで、より多くの情報を表現できます。たとえば、雨が降っている/いないでは大まかすぎるので、もっと詳しい情報として「晴れ」「曇」「雨」「雪」のどれであるかが知りたいとします。これは、たとえば次のように 2 ビットに対応させて表現できます。

ビット表現	意味
00	晴れ
01	曇
10	雨
11	雪

このように、ビット列の長さを 1 増やすと、表せる場合の数は 2 倍になり、一般に N ビットのビット列では 2^N 通りの場合を表すことができます。そして、デジタル情報とは「いく通りかの場合のうちのどれか」という情報なので、すべてのデジタル情報は (必要なだけの長さを決めることによって) ビット列で表すことができます。

コンピュータとはひらたくいえば、ビット列を蓄積/転送/加工するための装置であり、その機能によってあらゆるデジタル情報を取り扱うことができます。さらに、これから実際に見ていくように、人間の介在なしに自動的に処理を行える、という点も重要です。

2.2 2 進法 — 0 と 1 による数値の表現 exam

コンピュータでは数を扱うことも多いので、ビット列を数に対応させる方法があると便利です。図 1 のような 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、合わせて 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「10_{10}」を表すわけです。

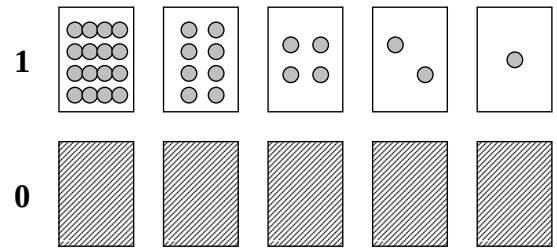

図 1: 5 枚のカード

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 **10 進法** (decimal system) と対比させて説明します。10 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(=$(10)^0$) 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(=2^0) 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。

たとえば、2 進法による足し算を見てみましょう。1 桁の足し算の表は次のようになります。

+	0	1
0	0	1
1	1	10

つまり、数字が 0 と 1 しかないわけですから、「0+0=0」「0+1=1」「1+0=0」「1+1=10」これだけです。最後のがびっくりしたかも知れませんが、1+1 の結果は 2 進表現の 1 桁では表せないので上の桁に桁上がりが起きて「10」になるわけです (そして先に学んだようにこれが「2」に相当します)。

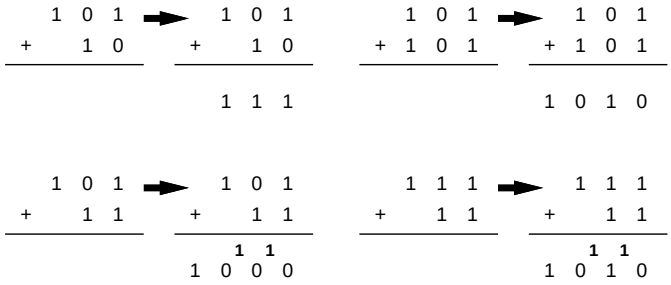

図 2: 2 進法の足し算

皆様は小学生のときに「0〜9 の値と 0〜9 の値を足すといくつになるか、また桁上がりがあるかどうか」を覚えさせられたはずですが、それに比べると上の規則はずっと簡単ですね? ですからコンピュータの回路にもしやすいわけです。桁上がりをつかって複数桁の足し算をしている例を図 2 に示しておきます。

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

2 進法は、ビット数が増えると、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。このほか、2 進法を 3 桁ずつ区切って表現する **8 進法**が使われることもあります。8 進法の場合は各桁の数字は 0〜7 の範囲ということになります。

2.3　2 の補数による負数の表現 exam

実際にコンピュータで整数を扱うときには、演算回路の大きさやメモリに格納するデータの大きさに応じて 32 ビット、64 ビットなどの決まったビット数で扱うことが普通です。一般に前節で説明した形の 2 進表現が N ビットの場合、2^N 通りのものが表せるので、0〜2^N-1 までの範囲の整数が表現できます。これを負の数を含まないことから**符号なし表現**といいます。

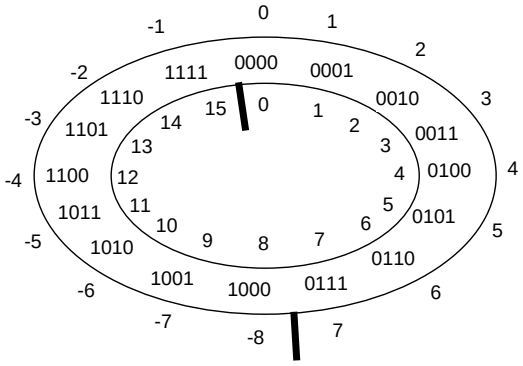

図 3: 4 ビットの 2 の補数表現と符号なし表現

では負の数を扱う場合はどうでしょうか。ここでは多くのコンピュータで使われている **2 の補数** (two's complement) と呼ばれる表現を説明します。2 の補数表現では、一番上位の (左の) ビットが符号ビットであり、これが 0 だと「0 以上の値」、1 だと「負の値」を表します。たとえば 4 ビットの 2 の補数であれば、1 ビットが符号ですから、残り 3 ビットで 8 通りの場合が表せるので、0 以上の数としては「0〜7」、負の数としては「-8〜-1」のそれぞれ 8 通りが表せます。[1]

これを具体的には図 3 のように割り当てます。ドーナツ型の上には 4 ビットの 2 進表現、内側にはそれを符号なし整数として解釈した場合の値、外側には 2 の補数として解釈した場合の値を記してあります。つまり、負の数の側は符号なし表現としての値から 16 を引く (マイナス側に 16 ずらす) ようになっています。

この表現がなぜ広く使われているかというと、負の数の計算がこれまでにやった符号なし計算と同じ回路で済むからです (そして引き算は符号を反転して足せばよい)。それが分かるためには、符号反転の方法がまず分かる必要がありますね。符号の反転は「0 と 1 を交換して、それから 1 を足す」ことでできます。たとえば 3 は「0011」ですから、その 0 と 1 を入れ換えて「1100」、それに 1 を足して「1101」これが-3 ということになります。

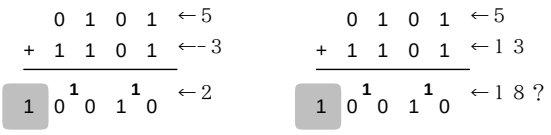

図 4: 4 ビットの 2 の補数表現での足し算

では「5+(-3)」をやってみましょう。図 4 左のように、これまでの足し算と同じ操作により、確かに期待通りの値「2」が得られます。しかし一番上の桁上がりは? それは、回路のビット数が 4 ビットなのですから、この網掛けになっている 5 ビット目は失われて消えるので、単に無視すればいいのです。

なんだか疑問ですね? 同じビットを符号なしとして解釈したらどうなのでしょう? それは図 4 右にありますが、「1101」を符号なしで解釈すると 13 ですから、5 と足すと 18 になるはずです。しかし 4 ビットの符号なし表現では 0〜15 しか表せないはずです。つまり桁上がりが無視された結果、正しくない答えである 2 が結果になります (これは正しい答え 18 を 16 ずらした値です)。

[1] 0 以上の方には 0 が含まれるので、2 の補数で表せる正の数は負の数より 1 つ少ないことに注意。

このように、コンピュータは有限のビット数で計算をするため、その範囲内で表せない結果になるような計算は正しく求まりません。これを**オーバフロー** (overflow、あふれ) と呼びます。たとえば 4 ビットの場合、図 3 の太線のところを計算結果が超えるとオーバーフローになります (上で見たように、符号なしの場合と 2 の補数の場合でオーバーフローの起こる境界は違います)。

演習 0 2 進表現の練習として次のことをやりなさい。

a. 次の 10 進表現を 2 進表現に直しなさい。8, 17, 25, 107
b. 次の 2 進表現を 10 進表現に直しなさい。1011, 100100, 111010
c. 次の 10 進表現を 4 ビットの 2 の補数表現に直してから足し算しなさい。3+7、7+(-5)、(-6)+(-3)
d. 4 ビットの 2 の補数表現した 2 つの数を足すとオーバーフローになり正しく結果が表現できない例を作りなさい。「正の数+正の数」の例と「負の数+負の数」の 2 例作ること。

3 コンピュータと情報処理

3.1 コンピュータとプログラム exam

この講義の＃ 1 において「コンピュータとは情報を扱う機械である」と述べましたが、前節の説明から、その情報はデジタル情報であり、ビット列として表されていることが分かりました。そこで定義を次のように具体化します。

◎ コンピュータとはビット列を処理する装置である

ここで「処理する」というのは、転送する (あちこち移動する)、蓄積する (記録・保管する)、そして加工することを表します。とくに最後の加工が大切で、これによってコンピュータは新しい情報を作り出したりできるわけです。

では、コンピュータの内ではどのようにして「ビット列の加工」を行うのでしょうか。数値の計算など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ます。しかし、コンピュータに行わせたい「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むことは不可能です。

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「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? 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です。具体的には、次のようにしています (図 5)。

- 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**(central processing unit、中央処理装置) はデータをメモリから取り出し、**レジスタ** (register) 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。

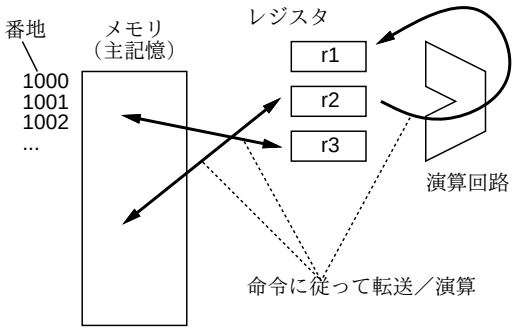

図 5: コンピュータと命令

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。ということは、新しいプログラムをよそから持って来ることも、さらにはプログラムを動かすことで新しいプログラムを作り出すことも、可能になるのです。非常に画期的だと思いませんか? そういうわけで、この「プログラムもメモリに格納されたデータである」という構成を、**プログラム内蔵** (stored program) 方式、ないし考案した科学者の名前から**ノイマン型** (Neuman architecture) と呼びます。

3.2 小さなコンピュータのシミュレータ exam

本ものの 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 言語で記述されていて、ブラウザ上で動作します (図 6)。とりあえず使ってみましょう。

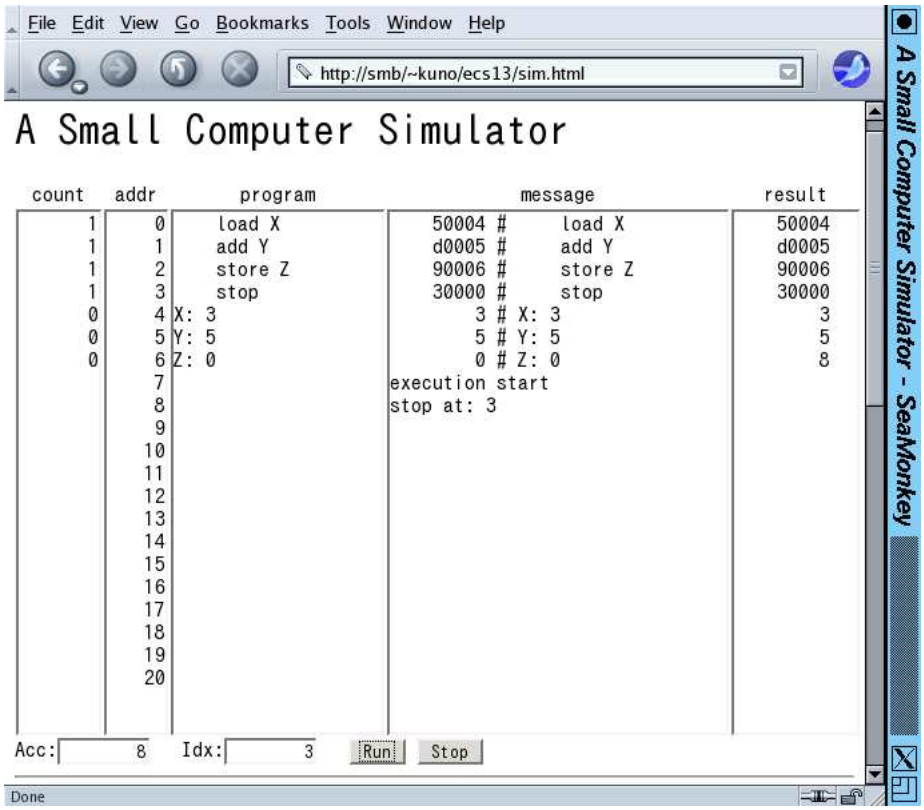

図 6: 「小さなコンピュータ」の画面

ここで「プログラム (program)」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 (run ボタンをクリック)」してみましょう。なお、

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 (図 7 上)、「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 (図 7 下)。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。この命令が無いと、コンピュータは次の番地の内容を命令だと思って実行してしまいます (上の例だと次には「3」「5」等が入っていて、これらを実行して行き、正しく止まりません)。

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ることが分かります。

実行開始時には、**プログラムカウンタ** (program counter、PC) というもう 1 つのレジスタ (画面には見えていない) に 0 を入れてから開始します。CPU は PC が指す番地 (最初は 0 番地) から命令を取り出し、PC をその命令の次の番地に変更します。最初の命令を実行し終ったらまた PC の指している番地から命令を取り出し、PC を次の番地にします。これを繰り返して実行が進みます。

なお、表示の見かたですが、message の欄はコードを命令に翻訳した様子や実行開始/停止の情報が表示されます。result の欄はプログラムが停止したときのメモリの内容が表示されます。

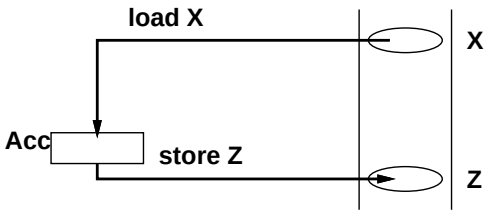

図 7: ロード命令とストア命令

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、このように命令やメモリの場所に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 message 欄に機械語 (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。

演習 1 「小さなコンピュータ」でここまでに出て来た例題をそのまま動かしてみなさい。うまくできたら、以下の処理を実行するプログラムを作成してみなさい。

a. 5 つの値を A、B、C、D、E というラベルの番地に入れておき、その合計を Z というラベルの番地に入れて止まる。データ例: 1, 2, 3, 4, 5 → 結果 15 (16 進では F)。
b. A というラベルのついた番地に入れておいた数値を 5 倍し、結果を Z というラベルのついた番地に入れて止まる。データ例: 4 → 20 (16 進では 14)。
c. 3 つの値を A、B、C というラベルのついた番地にそれぞれ入れておき、その 3 つの最大値を求めて、Z というラベルのついた番地に入れて止まる。データ例: 3, 6, 2 → 結果 6。

いずれにおいても、プログラムがどのように動作するか説明すること。

3.3 ループのあるプログラム

先の課題では、5 つの値の合計とか、5 倍とかなので、その数だけ足し算命令を使えば済んでいました。

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? A、B、C…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 (図 8)。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。

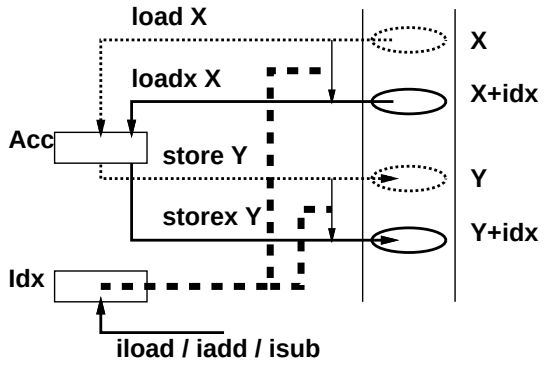

図 8: Idx を使うロード命令とストア命令

```
       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、loadx 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、手前 (上) 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)。[2][3]

演習 2 「小さなコンピュータ」で上に説明した N 個のデータの合計プログラムを動かし、さらにデータの個数を増やしたり減らしたりして、動作を確認しなさい。うまくいったら、以下の課題から 1 つ以上をやりなさい。

a. 「小さなコンピュータ」では左端の count 欄にそれぞれの命令を実行した回数が表示され

[2]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 positive」、ifn〜「if negative」

[3]「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名前をつけ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。

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

ます。合計するデータの個数 N と、データを取り出す loadx 命令の実行回数の関係を調べ、なぜそうなるのかを検討しなさい。

b. 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」を求めるプログラムを作りなさい。データ例:「1 -2 3 -4 5 0」→結果例: 「15」

c. 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) を求めるプログラムを作りなさい。データ例:「X: 3, Y: 5」→結果例:「15」。ヒント: この問題では Idx レジスタは使う必要がありません。つまり iload や lodax 命令は使う必要がありません。

b と c については、プログラムがどのように動作しているか説明すること。

4 コンピュータの性能向上とその意義

4.1 CPU の製造技術

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。

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 Integration) 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、またその配置に応じてトランジスタの働きをするようになります。だから、微細な模様をデザインしてその模様を焼きつけたマスクに沿って拡散を行うことで、好きな形の素子の配置と電気配線がウエハー上に作れます (図 9)。

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が量産できること、そして回路を細かくすることで「焼き付けて現像する」

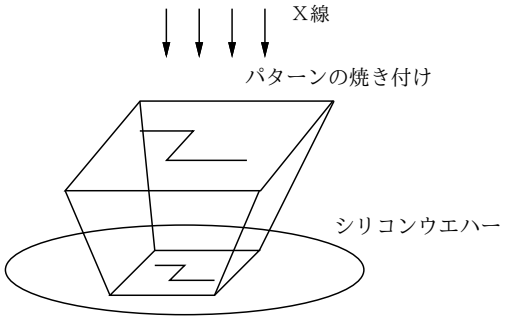

図 9: VLSI の製造方法

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) に修正されていますが、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。

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…

4.2 デジタル革命の本質

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。

- 汎用的なデジタル情報 — 世の中の多くの情報は、デジタル表現でき、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と小型化 — 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。たとえば、数ミリ角のチップで完全なパソコンと同じ機能を持つ CPU が (速度は遅いが) 実現できる。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。
- 通信とコンピュータの融合 — 以前の通信は「文字や音声を伝達する」だけのものだったが、インターネットに代表されるコンピュータネットワークでは、ネットワークを介してやり取りされるデジタル情報を直接コンピュータにより制御・加工することで、人間の介在なしに膨大かつ多種多様な情報をあらゆる場所に廉価に流通させることを可能とした。今やインターネットの存在しない世界はほとんど考えることすらできない状況である。

演習 3 (チャレンジ問題) 以下の課題から 1 つ以上を選んでやってみなさい。

a. 「小さなコンピュータ」の 1 命令あたりの実行時間 (または同じことですが 1 秒間あたりの平均命令実行数) を測ってみなさい。どのようにして計測したか、また答えの求め方の根拠を必ず記すこと。

b. 「小さなコンピュータ」で、これまでにまだ使ったことのない命令を使うプログラムを作りなさい。どのようなプログラムか、どのように動作するかの説明と、実行例をつけること。

c. 本もののコンピュータの 1 命令あたりの実行時間 (または同じことですが 1 秒間あたりの平均命令実行数) を測ってみなさい。対象とするコンピュータや測定方法はお任せしますが、レポートを読んだ人に数値の根拠が分かるように説明すること。

本日の課題 4A

本日の課題は「演習 1」「演習 2」「演習 3」に含まれる小問 (合計で 9 個) の中から 1 つ以上を選択し、結果をレポートとして報告して頂くことです。LMS の「レポート＃ 4」の入力欄に直接入力してください (別途作成したものをコピーペーストで貼っても構いません)。以下の内容がこの順に含まれるようにしてください。

- 冒頭に題名「コンピュータリテラシレポート＃ 4」、学籍番号、氏名、提出日付を書く。(グループでやったものはグループのメンバー全員の氏名も別途書く。)
- 課題の再掲を書く (どんな課題であるかをレポートを読む人が分かる程度に要約する)。
- レポート本体の内容 (やったこととその結果) を書く。必ず自分が書いたプログラムを掲載すること。
- 考察 (課題をやった結果自分が新たに分かったことや考えたこと) を書く。
- 以下のアンケートに対する回答。

  Q1. コンピュータの動作原理やアセンブリ言語によるプログラムについてどれくらい知っていましたか。新たに知ったことで面白かったことは何ですか。

  Q2. 「小さなコンピュータ」でプログラムが組めるようになりましたか。

  Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

なお、課題はグループでやって構いません。その場合も、(メンバー全員の氏名を明記した上で) レポートは必ず各自で執筆してください。レポート文面が同一 (コピー) と認められた場合は同一であると認めた全員について点数にペナルティを科すことがあります。