コンピュータネットワークのせいで不便になること：裸のアーキテクチャに興味はありますか?

佐藤, 周行
九州大学大型計算機センター研究開発部

https://doi.org/10.15017/1470248

出版情報：九州大学大型計算機センター広報. 27 (2), pp.93-110, 1994-03-15. 九州大学大型計算機センター
バージョン：
権利関係：

コンピュータネットワークのせいで不便になること

— 裸のアーキテクチャに興味はありますか? —

佐藤周行*

1 序にかえて – コンピュータネットワークが整備されて変わること

コンピュータネットワークの整備が全国的に急速に進んでいる。その整備に対応して計算機環境もどんどん便利になっている(と思われている)。しかし、便利になるばかりではない。従来たいしたことではなかった問題が急に露見することがある。本文ではその手の問題のうち、データの流通性の点から光を当てて解説めいたものを試みることにする。

1.1 超高速モデムとして – 通信手段としてのネットワーク

コンピュータネットワークが整備されると何ができるようになるか? それに対する答えは文字通り「必要に応じて」である。

コンピュータネットワークに対する需要のうち、大きいものは現在以下のようなものであると言われていて、飛び交うパケットもこれらに関するものが実際多い。

1. リモートの計算機に対する端末
2. 電子メール
3. ファイル転送

1. と 2. については何も問題ない。皆、仲良くやれば良いのである。計算 / 計算機に対する意識は従来のままで十分間に合う。だって通信手段なんだもの。モデムがず〜っと速くなったと思えば良いのだ。

3. も原則として問題がない。モデムが速くなればファイルの転送に要する時間は短縮されるというわけだ。いままで手元のパーソナルコンピュータにファイルを持ってくるための線がモデム経由 9.6kbps といった環境が、10Mbps の線になれば(単純に計算すれば)1000 倍のスピードになるわけである。(わざと乱暴な計算をすれば)従来の環境ではファイル転送に(例えば)10 時間かかっていたところがいきなり数秒でファイル転送が終ることが可能になるわけである。

平成 6 年 1 月 14 日 受理

*九州大学大型計算機センター研究開発部

ただし、これには落し穴がある。普通のテキストファイルのやりとりなら基本的に問題ないが、(計算機のアーキテクチャを色濃く反映した)バイナリファイルのやりとりまで便利になってしまう。アーキテクチャの違いをどこでどう吸収するか? この問題はいったんはまると結構面倒臭い。

1.2　分散・協調計算へ – 計算モデルの変化

実はコンピュータネットワークのポテンシャルは非常に高く、上のような「基本的」な利用にとどまらない利用ができる可能性を持っている。キーワードは「サーバ」と「分散」である。

1. ファイルサーバ、CPU サーバ、DB サーバなどの各種サーバ

2. 分散・協調計算

計算機の得意分野ごとに機能を分散させ、自分は PC や WS からそれらにアクセスして全体として計算を進めていくという計算モデルは今でもかなり一般的になってきている。例えば、メインフレーム (センター設置の M1800 等) はファイルサーバとしての機能は他の追随を許さず、さらにベクトル計算機 (同じく VP2600 等) は特に数値計算サーバとしての機能に優れている。従来はこれらで全部まかなうという感じであったのだが (なぜならばそれしかなかったから)、今やグラフィックスに関しては専用プロセッサで機能強化している GWS の方がよいし、電子メールなどはある意味でメインフレームを使うのはもったいない。

また、計算機群をネットワークでつなぎ、計算結果を交換しあって全体として一つの「計算」をしていくという分散協調型のモデルは (比較的低速なイーサネットなどで結合しておいて実効が上がるかどうかはともかく) 今後スーパーコンピューティングの有力な手法のひとつになることがかなりの確率で予想される[1]。

1.3　ネットワークのせいで不便になること

計算機ユーザにとってはそんな哲学の問題など興味はないだろうけど、それでも考えておかなければならない厄介な問題がいくつかある。今までは「計算機が一つ」という意識だったから日本語コード体系にしても、浮動小数点数のフォーマットにしても計算機側が勝手に決めることができた。だって「他の計算機」という概念がないんだもの[2]。

これからはファイル転送にしても、分散計算にしても、ユーザに対して種類の異なる計算機が生で見えてくるようになるわけである。例えば、パーソナルコンピュータで簡単な前処理だけやって、後はベクトル計算機をぶんまわすという計算はネットワークの高速化とパーソナルな計算環境の整備でこれから (今よりも) ずっと一般的になるだろう。この時にパーソナルコンピュータとベクトル計算機という最低 2 種類の計算機がデータのやりとりを行うわけである。

これから、このような時に大問題になるデータフォーマットのうち数種類について解説していこうと思う。最大の問題は計算機どうしがデータのやりとりをするときにはデータフォー

[1]計算機センター等に生息し、マクロ的な計算機システムの構築を考える人で今後、例えば 10 年前の大型計算機センターのようなシステム構築方法から頭が離れられない人はそれだけで失格である。– と言い切ると天に唾しているような気がしてしょうがない。

[2]某国の通産省がその昔、独自仕様計算機を作り続ける国内メーカに対してその路線を転換して当時世界業界標準であった IBM クローンを製造するように強力に指導したのは有名な話。

マットの違いに注意しなければならない、という面倒な状況がユーザに投げ出されたままであることである。コンピュータネットワークはこの流れを加速する。無防備なユーザは奔流に飲みこまれてしまうかもしれない。タイトルの「コンピュータネットワークのせいで不便になること」とは実はこのことを指すのである。この奔流から身を守る方法として、一番原始的だが、実は一番効果的な方法としてはたとえば

- 単一種類の計算機 (とそのネットワーク) しか使わない

と言うものがある。パーソナルコンピュータでも規模の大きいネットワークが構築できるようになっているので、PC なら PC、Mac なら Mac だけでネットワークを構築し、その中に閉じて仕事をすることは (仕事にもよるが) 実はそれほど奇異なことではない。

しかし良く考えて欲しいのだが、パーソナルコンピュータなど 100 台つなげたって、浮動小数点数演算に関してはできることなどたかが知れている。スーパーコンピュータを中核として複数種類のアーキテクチャと上手につきあって、研究効率をあげた方が絶対お得である。— と、さりげなく大型計算機センターの宣伝をして本文に移ろう。

2 アーキテクチャによって違うもの

まず、世の中に存在する計算機がどれほどバラエティに富んでいるかを表によって示すことにしよう。

	M1800/VP2600		VAX	R4000	i486	SPARC	MAC	CRAY
	MSP	UXP	VMS	IRIX	MS-DOS	SunOS	Sys7	UNICOS
ワード長[i]	32	32	32	64	32	32	32	64
バイトオーダ[ii]	B	B	L	B	L	B	B	B
文字コード[iii]	E	A	A	A	A	A	A	A
日本語コード[iv]	J	E	E	–	S	E	S	–
浮動小数点数[v]	M	M	V	I	I	I	I	C

i ワード長とは「1 ワード」が何ビットからなるかを示す。「32bit マシン」などと言えば、普通 1 ワードが 32 ビットからなっているマシンのことをいう。現在の主流は 1 ワード =32 ビットである[3]。

ii B は Big Endian、L は Little Endian を表す。

iii E は EBCDIC、A は ASCII を表す。ただし、これは主として OS に依存する話であって、チップに依存する話ではない。

iv J は JEF、E は EUC、S はシフト JIS。これもチップに依存する話ではないが、データ交換をしようとする (最近の) ユーザが最初に引っかかる所でもある。

v M は IBM 仕様、V は VAX 仕様、C は CRAY 仕様、I は IEEE 規格。

以下、少し詳説を試みることにしよう。

[3]余談であるが、昔、パーソナルコンピュータといえば 16 ビット機が全盛のころに某ソフトハウスでパーソナルコンピュータ関係のアルバイトをしていた某君は「1 ワード」といえば 16 ビットのことだと思う習性がついてしまい、大型計算機センターに就職する前にリハビリが必要だったとか。今となっては懐かしい話である。

3　ワード長

1 ワードが何ビットからなるかを表す。整数 (通常 1 ワード) が何ビットからなるかを誤解していると、思わぬ失敗をすることがある。64 ビットを整数と思っているマシンと 32 ビットを整数と思っているマシンでのデータ交換はどちらかが注意しないと一般にうまくいかない。

64 ビットマシンで src を用意し、

```
r4000% cat src.c
#include <stdio.h>

main()
{
      int i = 1;
      fwrite(&i, sizeof(int), 1, stdout);
}
r4000%
```

さらに 32 ビットマシンで dst を用意する。

```
sparc% cat dst.c
#include <stdio.h>
main()
{
      int i;
      fread(&i, sizeof(int), 1, stdin);
      printf("%d\n", i);
}
sparc%
```

次のようなことをやると、

```
r4000% src | rsh sparc dst
0
r4000%
```

となる。要注意。

ワード長についてちょっと現状を解説しておこう。現状の主流は 32 ビットを 1 ワードとするものである。メインフレームは言うに及ばず、PC や WS の分野でも i286 がようやく駆逐され、32 ビットマシンが主流になってきた[4]。

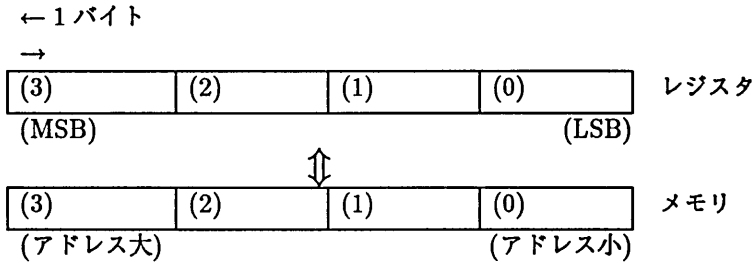
ところが、最近では新たに 64 ビットマシンが出始めて来ている。当初は CRAY だけの酔狂で済むかと思われていたが、さにあらず。MIPS R4000 はすでに市場に出ており、さらに SPARC にも 64 ビット化の計画がある。メインフレームにはその計画は少なくとも表には出ていないが、このままでいくと完全に取り残されるのではないかと他人事ながら気になる。

[4]制御用マシンではこんなことはない。念のため。

4 バイトオーダ

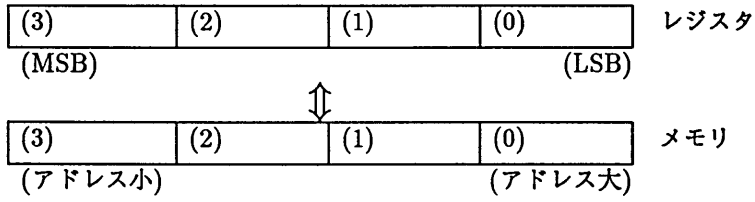
ここでは簡単のために32ビットのチップで話をする。レジスタは4バイトからなる。

i486やDECから出ているマシン群ではレジスタとメモリの間でデータを転送する時に以下のようになっている。



これがいわゆるLittle Endianと呼ばれているものである。

ところが、M1800やVP2600をはじめ、SPARCなどでは以下のようにしている。

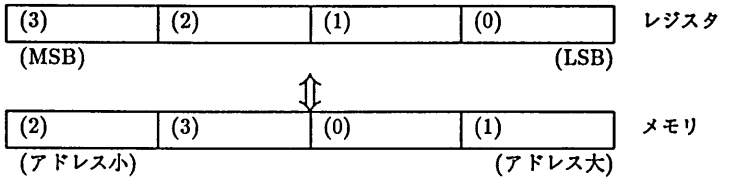


これは一般にBig Endianと呼ばれている。世の中のバイトオーダは大まかにいってこの2つ[5]に分類される。ファイルやLANを通じてデータのやりとりをする場合、これがまず問題になる。ただし、インターネットの世界では「ネットワークバイトオーダ」というものが定められている。これに従えば、Big Endianでデータ交換をすべしということになる。

5 日本語コード

異なった計算機システム間でファイルを移動させる時にまず最初につまずく(であろう)のが日本語コード体系である。現在、日本では日本語コードに対してはJIS[6]が定められており、

[5]ところで、筆者の回りには以下のような方式をとっている32ビットチップが存在する。これも一応はLittle Endianと呼ぶべきなのであろう。



[6]日本語コードの扱いというのは本来文化的な側面があるものであるが、その標準を通産省が定めあまつさえ第一水準、第二水準などという形で変に差別化しているのは問題であるという意見もあることにはある。読者は「電算処理による事務の効率化」の美名の下、JISコードにないからと勝手に名前の表記が変更されたら心穏やかでいられるだろうか?

情報処理では一応標準[7]ということになっている。ところが、計算機システムによっては扱いやすいように適宜 JIS をモディファイしているところがある。現在主流なのが JIS、シフト JIS、そして EUC と呼ばれるコード体系である。もうひとつ、JEF という MSP 標準の日本語コード体系があるが、すべて JIS から簡単な変換規則で得られる。

OS によっては標準の日本語コード体系を定めているところがある。例えば、MS-DOS と SONY の NEWS-OS ではシフト JIS が、UXP では EUC が標準のコード体系である。さらに、メールを交換する時のコード体系は JIS であることと定められている。念のためにいっておけば、これはハードウエアには直接関係なく、せいぜい OS レベルの話である。

5.1　日本語コード自動判別

いつのころからか以下の主張が有効になってきている。すなわち:
「日本語コード体系は任意でよい。ソフトの側で対応しましょう。」

上の 3 つ (JEF まで含めれば 4 つ) のコード体系が淘汰されない理由のひとつに、統一することにあまり意味がなくなっていることがある。

上の 3 つのコード体系についてはある簡単な方法を用いれば、ほとんど確実に 3 つのうちどれを用いているかが判別できる。この方法は筆者の回りでは 1980 年代の半ばにはすでにコード化されていた[8]。

上の主張が有効であるのは、じつは使用コード体系がほとんど自動判別できるということにも依存している。

現在、ファイルの日本語コードを自動判別してくれるソフトウエアのうち、センターでサービスしているものには以下のものがある。

- emacs の日本語版。
  これは電総研の小方、半田両氏によってなされた。
- nkf [9]
  これは富士通研の市川氏によってなされた。

5.2　日本語コードの相互変換 (I) – ツールの利用

さて、次のようなことがしたいとする:
「センターの UXP の上で日本語を含むデータを作成し、それをパーソナルコンピュータ (OS は MS-DOS) に転送し、こまごまとした編集を行う。」

UXP は特に何も指定しないと EUC を用い、片や MS-DOS ではシフト JIS が標準になっている。この時に EUC からシフト JIS へのコード変換が必要になる。現在センターでは次のようにすれば変換ができる。

5.2.1　emacs を用いる方法

emacs を用いてコード変換ができる。emacs を起動すると、下の方に図のようなタイトルバーが出てくる (ディスプレイによっては色がついていたり、白黒反転になっていたりする)。

[7]筆者は国粋主義者ではないつもりだが、最近の某コードのように日本語を使用しない人間が日本に自分たちが勝手に作った日本語コードを強制するのはどう考えてもおかしい。

[8]筆者が理解していたかどうかは別問題

[9]nkf の処理速度は遅いと言うのが定評である。しかし「プログラム資産を共有し、皆で貢献していく」というネットワーク文化 (というものがあるとしたらそ) の最も良質な部分を体現しているものとしての評価が高い。

```
[----]-----NEmacs: file.euc              (EEE-:Fundamental)--All--------------
                                          ↑(ファイルコード)
```

その真中辺りにある四組のアルファベットのうち、一番左側がファイルに使う日本語コード体系を示している。EUC は E(デフォルト)、JIS は J、(シフト JIS) は S で表す。
^X^K^F を何回かやると、これが次のようになるはずである。

```
[----]-----NEmacs: file.euc              (SEE-:Fundamental)--All--------------
```

こうしてからファイルをセーブすると、シフト JIS にコード変換がなされることになる。

5.2.2 nkf を用いる方法

nkf を用いる方法では、出力先のコード体系を指定するだけで良い。

```
kyu-cc% nkf -s < file.euc > file.sjis
```

主なオプションは以下の通りである。

nkf	-s	出力先コードはシフト JIS
	-e	出力先コードは EUC
	-j	出力先コードは JIS

5.2.3 fconv

UXP の上の fconv(/usr/uxp/fconv) はもともと MSP と UXP の間の様々なファイルフォーマット間の変換用に作られた強力なフィルタであるが、コード変換 (JEF+EBCDIC ⟷ EUC) にも使える。

JEF ⟶ EUC この場合、output-file は MSP の固定長フォーマットになる。

```
kyu-cc% fconv -c < input-file > output-file
```

EUC ⟶ JEF ただし、input-file が MSP の可変長フォーマットの場合。

```
kyu-cc% fconv -c -i V < input-file > output-file
```

5.3　日本語コードの相互変換 (II) – 自分でプログラミングする

コードの変換アルゴリズムは単純であるから、効率さえ無視すれば、自分でプログラミングしてもたいした手間ではない[10]。ここでは基本的なアルゴリズムを述べるにとどめることにする。ついでに JEF コードについても述べることにしよう。ただし、半角カタカナ[11]や、特殊コード[12]については考慮しないことにする。

以下、説明のために以下を仮定する。

- 日本語は 1994 年現在、16 ビットコードで表現するのが主流である。ここでも 16 ビットコードで表現することにする。変換前のコードのうち、上位 8 ビットを c2、下位 8 ビットを c1 とする。

c2	c1

変換後のコードについても、上位 8 ビットを c2、下位 8 ビットを c1 とする。

- char は 8 ビット、 int は 16 ビット以上であることを仮定する。
- JIS と JEF では、日本語コードはある特定のシークエンスで囲む (その前後に何があるかはとりあえず考えないことにする) 方式を取っている。例えば、文字列「,例 1」の表現方法を考えると、JIS(+ASCII) と、JEF(+EBCDIC) ではそれぞれ以下のように表現される。

JIS+ASCII	0x2c	0x1b 0x24 0x40	0x4e 0x63	0x1b 0x28 0x4a	0x31
JEF+EBCDIC	0x6b	0x28	0xce 0xe3	0x29	0xf1
	,	囲み始まり	例	囲み終り	1

ここでは日本語文字列を囲むシークエンスは以下のようになっている。

	囲み始まり	囲み終り
JIS	0x1b 0x24 0x40	0x1b 0x28 0x4a
JEF	0x28	0x29

これらのシークエンスの挿入 / 変換 / 除去については省略する。

5.3.1　JEF ⟷ EUC

両者のコード自体は同一である。

$$c2_{JEF} = c2_{EUC};$$
$$c1_{JEF} = c1_{EUC};$$

[10] し、適当な頭の体操にもなるのでこの手のフィルタを作った経験のある人はどこにでもいるはずである。一部ではスピード競争が繰りひろげられ、新しいフィルタが開発され続けられているらしい。

[11] あの貧相な字体はなんとかならないものか。

[12] ここで NEC-PC ユーザに一言申し上げておきたいのであるが、例えば① のような○囲み数字は NEC-PC の特殊コードであり、JIS コード外である。無反省に① などと使うのは自制してほしい。

5.3.2 JIS ⟷ EUC

EUC から JIS へは各バイトの最上位ビットを落すだけ。

$$\begin{aligned} c2_{JIS} &= c2_{EUC} \,\&\, 0x7f; \\ c1_{JIS} &= c1_{EUC} \,\&\, 0x7f; \end{aligned}$$

逆に JIS から EUC への変換は各バイトの最上位ビットを立てるだけ。

$$\begin{aligned} c2_{EUC} &= c2_{JIS} \mid 0x80; \\ c1_{EUC} &= c1_{JIS} \mid 0x80; \end{aligned}$$

5.3.3 JIS ⟷ シフト JIS

使い道としてはこの相互変換が一番役に立つであろう。高速計算機に載っている OS の世界ではシフト JIS がほとんど使われていないのにパーソナルコンピュータの標準漢字コードがシフト JIS であることがこれらの変換を「役に立た」せている最大の原因である。今となってはどちらかが譲歩できるというような状況ではなくなってしまった。

まず、JIS からシフト JIS への変換。

$$\begin{aligned} c2_{SJIS} &= ((c2_{JIS}-1)>>1) + \begin{cases} 0x71 & c2 \geq 0x5e; \\ 0xb1 & \text{otherwise.} \end{cases} \\ c1_{SJIS} &= c1_{JIS} + \begin{cases} 0x1f & \text{if } c2_{JIS} \text{ is odd, and } c1 < 0x60; \\ 0x20 & \text{if } c2_{JIS} \text{ is odd, and } c1 \geq 0x60; \\ 0x7e & \text{if } c2_{JIS} \text{ is even} \end{cases} \end{aligned}$$

逆変換についてはわざと下のように書くが、これをコーディングする場合は工夫の余地がだいぶある。

$$\begin{aligned} c2_{JIS} &= 2 \cdot c2_{SJIS} - \begin{cases} 0xe1 & \text{if } c1_{SJIS} < 0x9f \text{ and } c2_{SJIS} \leq 0x9f; \\ 0xe0 & \text{if } c1_{SJIS} \geq 0x9f \text{ and } c2_{SJIS} \leq 0x9f; \\ 0x161 & \text{if } c1_{SJIS} < 0x9f \text{ and } c2_{SJIS} > 0x9f; \\ 0x160 & \text{if } c1_{SJIS} \geq 0x9f \text{ and } c2_{SJIS} > 0x9f; \end{cases} \\ c1_{JIS} &= c1_{SJIS} - \begin{cases} 0x1f & \text{if } c1_{SJIS} \leq 0x7f; \\ 0x20 & \text{if } 0x7f < c1_{SJIS} < 0x9f; \\ 0x7e & \text{if } c1_{SJIS} \geq 0x9f; \end{cases} \end{aligned}$$

6 英数字

太古の昔、世の中には EBCDIC というコードが定められており、計算機 (当時は IBM とその互換機が全盛であった) の英数字のコード体系といえば、EBCDIC と決まっていた。ところが、今は ASCII と呼ばれるコード体系[13]が主流である。

面倒なことに、1993 年度現在[14]でセンターの主 OS である MSP のコード体系だけが EBCDIC なのである。ここでは EBCDIC と ASCII の間のコード変換プログラムを設計しよう。それだけではつまらないので、速いフィルタを書くための一般的なテクニックのひとつであるテーブル方式を解説することにしよう。

[13] より正確にいえば ASCII を英数字のコード体系に採用しているシステム

[14] あくまでも「現在」である。この点については一寸先は闇であることを明言しておこう。

6.1 速いフィルタはどう書くか?

簡単な例として、32 ビット整数を 16 進表示するプログラムを考えよう。速いフィルタを書こうとする時には、以下のように変換テーブルを内部に持っておくとよい。

```
void d2h(int i, char *ans)
{
        static char hex[16] = {'0','1','2','3','4','5','6','7','8',
                               '9','a','b','c','d','e','f'};
        ans[4] = 0;
        ans[3] = hex[i & 0x0f];
        ans[2] = hex[(i & 0x0f0) >> 4];
        ans[1] = hex[(i & 0x0f00) >> 8];
        ans[0] = hex[(i & 0xf000) >> 12];
}
```

これをいちいち例えば、

```
        {int ii = i & 0x0f; ans[3] = ii + ((ii < 10)? '0': ('a'-10))}
```

などとやると、途端に計算が遅くなる。ただしこれも程度問題で、例えばシフト JIS から JIS への変換テーブルを全部持とうなどということは現実的でない[15]。EBCDIC⟷ASCII のフィルタは変換テーブル方式でプログラムをかける問題のサイズを持っている自明でない例になっている。プログラムは以下の通り。

```
char ascii2ebcdic[256] =
{'\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0',
 '\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0',
 0x40,0x4f,0x7f,0x7b,0xe0,0x6c,0x50,0x7d,0x4d,0x5d,0x5c,0x4e,0x6b,0x60,0x4b,0x61,
 0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0x7a,0x5e,0x4c,0x7e,0x6e,0x6f,
 0x7c,0xc1,0xc2,0xc3,0xc4,0xc5,0xc6,0xc7,0xc8,0xc9,0xd1,0xd2,0xd3,0xd4,0xd5,0xd6,
 0xd7,0xd8,0xd9,0xe2,0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0x4a,0x5b,0x5a,0x5f,0x6d,
 0x79,0x81,0x82,0x83,0x84,0x85,0x86,0x87,0x88,0x89,0x91,0x92,0x93,0x94,0x95,0x96,
 0x97,0x98,0x99,0xa2,0xa3,0xa4,0xa5,0xa6,0xa7,0xa8,0xa9,0xc0,0x6a,0xd0,0xa1,0x07,
 '\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0',
 '\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0',
 '\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0',
 '\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0',
 '\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0',
 '\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0',
 '\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0',
 '\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0'
}

char ebcdic2ascii[256] =
{
```

[15]組み合わせてスピードアップということは十分考えられるが。

```
 '\0','\0','\0','\0','\0','\0','\0',0x7f,'\0','\0','\0','\0','\0','\0','\0','\0',
 '\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0',
 '\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0',
 '\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0',
 0x20,'\0','\0','\0','\0','\0','\0','\0','\0','\0',0x5b,0x2e,0x3c,0x28,0x2b,0x21,
 0x26,'\0','\0','\0','\0','\0','\0','\0','\0','\0',0x5d,0x5c,0x2a,0x29,0x3b,0x5e,
 0x2d,0x2f,'\0','\0','\0','\0','\0','\0','\0','\0',0x7c,0x2c,0x25,0x5f,0x3e,0x3f,
 '\0','\0','\0','\0','\0','\0','\0','\0','\0',0x60,0x3a,0x23,0x40,0x27,0x3d,0x22,
 '\0',0x61,0x62,0x63,0x64,0x65,0x66,0x67,0x68,0x69,'\0','\0','\0','\0','\0','\0',
 '\0',0x6a,0x6b,0x6c,0x6d,0x6e,0x6f,0x70,0x71,0x72,'\0','\0','\0','\0','\0','\0',
 '\0',0x7e,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7a,'\0','\0','\0','\0','\0','\0',
 '\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0','\0',
 0x7b,0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,'\0','\0','\0','\0','\0','\0',
 0x7d,0x4a,0x4b,0x4c,0x4d,0x4e,0x4f,0x50,0x51,0x52,'\0','\0','\0','\0','\0','\0',
 0x24,'\0',0x53,0x54,0x55,0x56,0x57,0x58,0x59,0x5a,'\0','\0','\0','\0','\0','\0',
 0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39,'\0','\0','\0','\0','\0','\0'
}


void a2e(char *astr, char *ans)
{
        while(*astr)
             *ans++ = ascii2ebcdic[*astr++];
        *ans = '\0';
}


void e2a(char *estr, char *ans)
{
        while(*estr)
             *ans++ = ebcdic2ascii[*estr++];
        *ans = '\0';
}
```

ここではa2eがASCII文字列をEBCDIC文字列に、e2aがその逆変換をするルーチンになっている。

7 ftpのテキストモードとは? – 普通のテキストだって安心できない

インターネットの世界でファイル転送を行う場合、ftpが一般に使われる。コード系の変換がでたついでに、ftpでファイル転送をする場合の変換フィルタについてちょっと解説しておこう。

ftpにはテキストモードとバイナリモードの2つのモードが存在する。それらの違いは基本的には行末を表すコードの調節をする(テキストモード)か、しないか(バイナリモード)だけにある[16]。

```
qviss% ftp kyu-cc
```

[16]ただしMSPとのファイル転送でテキストモードの場合はEBCDICとASCIIの変換くらいはやってくれる。

```
220 kyu-cc FTP server (UXP/M) ready.
Name (kyu-cc:xxxxxx): a79999a
331 required for a79999a.
Password:
230 User a79999a logged in.
ftp> bin
200 Type set to I.
ftp> ascii
200 Type set to A.
...
```

上のセッションで bin(バイナリモードへ) と ascii(テキストモードへ) がこれらの調節をするサブコマンドである。

では、行末コードを調整する必要があるかどうかという話であるが、Unix、MS-DOS と Mac ではそれぞれが異なる行末コードを使っている[17]。

OS	Unix	MS-DOS	Mac Sys.7
行末コード	0x0a	0x0a0d	0x0d

したがって、テキストをバイナリモードだけでやりとりすると、しばしば喜劇的なことが起きる。ftp のモードはこれを調節するためにも重要である。

Mac ⟶ Unix vi で編集しようとすると「1 行が長過ぎて編集できない」というメッセージがでたら、まずこれだと思ってよい。emacs を使って、^M を ^J に変換すればよい。

MS-DOS ⟶ Unix vi でも emacs でも同じだが、行末に ^M が表示されるファイルはこれだと思ってよい。見苦しい。

8　浮動小数点数フォーマット

センターの M1800 と VP2600 システムの浮動小数点数フォーマットは M 形式と言われる IBM 互換形式である。ところが、ほとんどのワークステーションやパーソナルコンピュータでは IEEE 形式と呼ばれるフォーマットで浮動小数点数を表現している。両者は (当然のことながら) 異なる。バイナリデータでの交換[18]をしようと思ったら、形式変換が必要になる。

8.1　浮動小数点数の表現

浮動小数点数フォーマットには一般に単精度のものと、倍精度のものがある。32 ビットアーキテクチャでは単精度といったら 32 ビットでの浮動小数点数の表現、倍精度といったら 64 ビットでの浮動小数点数の表現を指す場合が多い。

s	指数部 (e)	仮数部 (m)

[17]MSP ではこれに代わるものとしてレコードの概念があり、面倒なので取り敢えず説明しない。もちろん、ftp を使って MSP と他のシステムとの間でデータセットのやりとりをする時にこれらの変換はやってくれる。

[18]これをやろうとすると、浮動小数点数フォーマットの変換の他にもバイトオーダや、(Fortran の場合) レコードのフォーマットの整合性をとる必要があるので、実は結構面倒くさい。

上で s は符合部を表し、1 ビットをあてる。この他に基数 (b) と、指数部にはかせるげた (q) の情報が必要になる。

これらを使って、浮動小数点数はどう表現されるかということを考えると。e と m をそれぞれのビット幅をもった符合なしの整数とみなすと

$$(-1)^s \times b^{e-q} \times Rep(m)$$

となる。ここで、$Rep(m)$ は m の表す小数であるとする。この場合正規化の条件 ($b^{-1} \leq Rep(m) < 1$) をつけるのが一般的である。

8.1.1 M 形式

M1800 と VP2600 ではこの形式を使っている。

	単精度	倍精度
e の桁数	7	7
m の桁数	24	56
b	16	16
q	64	64
$Rep(m)$	$m \cdot 16^{-6}$	$m \cdot 16^{-14}$

ただし、$Rep(m)$ が正規化条件を満たしていないとだめ。

8.1.2 IEEE 形式

i386 以上のチップ、WS に用いられているチップのほとんどは IEEE 形式を用いている。

	単精度	倍精度
e の桁数	8	11
m の桁数	23	52
b	2	2
q	127	1023
$Rep(m)$	$1.0 + m \cdot 2^{-23}$	$1.0 + m \cdot 2^{-52}$

ただし、単精度では e が 255 の場合、m が 0 ならば負の∞、それ以外では NaN(Not a Number)、倍精度でも e が 16384 の場合、m が 0 ならば負の∞、それ以外では NaN(Not a Number) を表すことになっている。さらに、$e = 0$ かつ $m \neq 0$ の場合、単精度なら $(-1)^s \cdot 2^{-126} \cdot m \cdot 2^{-23}$、倍精度なら $(-1)^s \cdot 2^{-1022} \cdot m \cdot 2^{-52}$、を表すことになっている。

注意すべきは $e = 0$ の場合を除き、$Rep(m)/2$ が自動的に正規化の条件を満たしていることである。

8.1.3 その他

昔の CDC や VAX、新しいところでは CRAY は独自の形式で浮動小数点数を表現している。その他、基数を 10 にとるフォーマットを受けつけるプロセッサもある。

そう、「異機種間、バイナリで浮動小数点数をやりとりするのはとても面倒」なのである。でも、そうもいってられないし、適当につき合うことでメリットがあるなら、つきあってやろうじゃないかというと、やはり、フォーマット変換が必要になってくる。

8.2　フォーマット変換

ここではとりあえず問題になりそうなM←→IEEEの間の変換を考えることにしよう[19]。例えばMacで作ったTIFF形式のイメージデータをUXPで表示させようとするときに役に立つ[20]。

簡単のため、1ワードは32ビット、バイトオーダはBig Endianとする。さらに簡単のため、$-\infty$やNaNも無視する。エラーがおきてもしらんぷりという最低限の条件で作ることにしよう。

8.2.1　準備

以下の操作ではビットごとのシフト、AND、ORが必要になり、さらにビット幅までを考慮に入れる必要がある。アセンブラでプログラミングすれば好き勝手にできるのは事実であるが、生産性を考えればアセンブラというのは最悪の選択である[21]。ここでは、アセンブラプログラミングは好事家にまかせておいて、Cでプログラミングすることにしよう。

ここで重宝するのがビットフィールドとunionである。

Cにおけるビットフィールドはデータのビット幅を指定する時に用いる。M形式、IEEE形式の浮動小数点数フォーマットは次のように表現される。

```
typedef struct {
        unsigned      sign:1;
        unsigned      exp:7;
        unsigned      mant:24;
} mfloat;
/* M Float sign:1, exponent:7, mantissa:24 */
typedef struct {
        unsigned      sign:1;
        unsigned      exp:8;
        unsigned      mant:23;
} ie3float;
/* IE3 Float sign:1, exponent:8, mantissa:23 */


typedef struct {
        unsigned      sign:1;
        unsigned      exp:7;
        unsigned      mant:24;
        unsigned      mant2;
```

[19]ただし、スーパーコンピュタが学内に3システム(しかもいずれも大規模)もころがっていて、M形式、CRAY形式、IEEE形式の浮動小数点数のフォーマットにそれぞれ気を配らなければならないという■■大のような、九大などかすんでしまうような環境に「栄転」すると、「とりあえず」などということは言えなくなってしまうけど...

[20]センターではxv (/usr/local/bin/X11/xv) を使うとこれができる。

[21]し、そもそもRISCに代表されるようにプロセッサの複雑度がましている現在、人間の書くアセンブラコードがコンパイラの吐き出すコードより上等であるというのは迷信にすぎない。

```
} mdouble;
/* M Double sign:1, exponent 7, mantissa:(24+32) */
typedef struct {
        unsigned     sign:1;
        unsigned     exp:11;
        unsigned     mant:20;
        unsigned     mant2;
} ie3double;
/* IEEE Double sign:1, exponent 11, mantissa:(20+32) */
```

実はこれでも不十分。昔のCはいざ知らず、現在の規格では、ビットフィールドは指定されたビット幅を格納するだけに十分な幅を取らなければならないとされているだけで、それ以外の規定はない。これを指定幅にきちんと収めるためには外から規制を加える必要がある。それがunionである。

```
typedef union {
        ie3float      ie3;
        mfloat        m;
        float         f;
} float_t;

typedef union {
        ie3double     ie3;
        mdouble       m;
        double        d;
} double_t;
```

ここまでやれば、(まず)大丈夫。

8.2.2 単精度浮動小数点数

エラー処理も何もしていないので、実際に使う場合は適宜改良すること。

```
float_t m2ie3float(float_t fp)
{
     int shift;
     float_t rfp;

     if (fp.f == 0.0) return fp;
     shift = (fp.m.mant & 0x800000? 1:
              fp.m.mant & 0x400000? 2:
              fp.m.mant & 0x200000? 3: 4);
     rfp.ie3.sign = fp.m.sign;
     rfp.ie3.exp = 4 * fp.m.exp - shift - 128;
     rfp.ie3.mant = ((fp.m.mant << shift) & 0x7fffff);
```

```
    return rfp;
}

float_t ie32mfloat(float_t fp)
{
    int shift;
    float_t rfp;

    if (fp.f == 0.0) return fp;
    shift = 3 - ((fp.ie3.exp-127) % 4);
    rfp.m.sign = fp.ie3.sign;
    rfp.m.exp = ((fp.ie3.exp - 127) >> 2) + 65;
    rfp.m.mant = (fp.ie3.mant | 0x800000) >> shift;
    return rfp;
}
```

8.2.3　倍精度浮動小数点数

エラー処理も何もしていないので、実際に使う場合は適宜改良すること。

```
double_t m2ie3double(double_t dp)
{
      int shift;
      double_t rdp;

      if (dp.d == 0.0) return dp;
      shift = (dp.m.mant & 0x800000? 3:
              dp.m.mant & 0x400000? 2:
              dp.m.mant & 0x200000? 1: 0);
      rdp.ie3.sign = dp.m.sign;
      rdp.ie3.exp = 4 * dp.m.exp + shift + 767;
      rdp.ie3.mant = ((dp.m.mant >> shift) & 0x0fffff);
      rdp.ie3.mant2 = ((dp.m.mant | ((1 << shift)-1)) << (32-shift))
                              | (dp.m.mant2 >> shift);
      return rdp;
}

double_t ie32mdouble(double_t dp)
{
      int shift;
      double_t rdp;

      if (dp.d == 0.0) return dp;
      shift = (dp.ie3.exp-1023) % 4;
```

```
        rdp.m.sign = dp.ie3.sign;
        rdp.m.exp = ((dp.ie.exp - 1023) >> 2) + 65;
        rdp.m.mant = ((dp.ie3.mant | 0x80000) << shift)
                        | (dp.ie3.mant2 >> (32-shift));
        rdp.m.mant2 = dp.ie3.mant2 << shift;
        return rdp;
}
```

8.3 Fortran で提供されていること

以上はスクラッチから作る人のための情報であるが、センターの UXP と MSP の Fortran には浮動小数点数のフォーマット変換のために以下の機能が提供されている。ただし、Fortran の場合はこれ以前にレコードフォーマットの変換をしなければならないので、気を配らなければならないことが存在することには変わりない。

8.3.1 UXP の場合

IEEE 形式のバイナリデータの I/O をする時には以下に示すように実行時オプションをつければ良い (変換できなかった時にはメッセージを出してくれることになっている)。

```
kyu-cc% frt test.f
kyu-cc% a.out -Wl,-C,-M < ieee.in.data > ieee.out.data
```

8.3.2 MSP の場合

MSP では IEEE 形式のバイナリデータの I/O をするためには以下のようにする。

- この対象になるのは書式なし (順次 or 直接 or 索引) 入出力文だけである。
- やりかたは FLIB のオプションとして指定する。下の例では基番 1 と 2 が IEEE 形式データの I/O をしている。

```
FLIB(CONVIE=(1,2))
```

MSP ではさらにサブルーチンとして IETOM と MTOIE が提供されている。本稿で作ったルーチンにエラー処理を追加したものと考えればよい。

1. IETOM

   IEEE 形式を M 形式に変換する。

2. MTOIE

   M 形式を IEEE 形式に変換する。

形式 (IETOM) CALL IETOM(IE3, M, ITYPE, RETCD [,MSG])

形式 (MTOIE) CALL MTOIE(M, IE3, ITYPE, RETCD [,MSG])

引数解説

IE3 変換前 / 後の IEEE 形式データ領域 (変数または配列要素名)

M 変換前 / 後の M 形式データ領域 (変数または配列要素名)

ITYPE 単精度の場合 0、倍精度の場合 1 を指定する。

RETCD 復帰コード。

RETCD	意味
0	正常終了
4	仮数部の一部損失
8	オーバーフロー / アンダーフローの発生
12	ITYPE または MSG の値が不正
16	引数不足

MSG 0 の場合診断メッセージ出力、1 の場合抑制。

9　結びにかえて – 問題は解決されていない

ここまで、アーキテクチャや OS(総称してプラットフォームということにする) によるデータ表現の違いとそれを吸収するためのフィルタについて (ごく狭い範囲に限って) 解説してきた。

現在、アーキテクチャの種類は多様になって収拾が不可能な状態であるし、また、別に収束させる必要もない。ただし、ユーザがどう対処するかということは今後の大問題である。序文でも触れたが、

1. 計算機については同一機種しか使わないというのは、小規模の計算しかしない人にとっては最適解である可能性がある。「自分はいったい何がしたいのだろう」と一度考えてみよう。

2. 大規模計算をしようと思えば、大型計算機センターのメインフレームの大容量メモリはまだまだ価値がある。あきらめて、上手に複数のアーキテクチャとつきあおう。

3. 超高速計算をしようと思えば、将来にわたって、GFLOPS、TFLOPS といった計算能力を持つマシンとつきあう必要がある。この場合も潔くあきらめて、上手に複数のアーキテクチャとつきあおう。

しかし、どちらにしてもユーザに責任を押しつけていることに変わりはない。とにかく昔のような単一の大きな計算サーバで何もかもやるということで仕事の効率が上がらなくなってきているのは事実である。特に大規模高速計算 (= スーパーコンピューティング) を指向しているユーザにとっては受難の時代である。ただでさえ (自動並列化コンパイラがまだこなれていないので) 裸の並列アーキテクチャと格闘しなければならない時代に嫌気がさしていることとは思うが、この小文で述べたこともついでに頭の片隅にとどめておいてくれれば幸いである。