14

ヒルベルト曲線

1890年、ペアノ（Giuseppe Peano）は平面曲線[1]で「平面空間を充填する（埋め尽くす）」というとても驚くべき特性を有するものを発見した。その曲線は、単位正方形の中をうねるように進み、各々の点(x, y)を最低でも1回は通過する。

ペアノ曲線は単位正方形の各々の辺を3等分し、元の正方形を九つの小さな正方形に分割することに基づいている。

彼の曲線はこれらの九つの正方形をある順番で辿っていく。そして九つの小さな正方形は、同様にしてより小さな九つの正方形に分割され、曲線はこれらの正方形をある順番で辿っていくように修正される。この曲線は基数3の小数を使って説明できる。事実、ペアノが最初に説明に使ったのがこの方法である。

1891年にヒルベルト（David Hilbert）［Hil］は、ペアノ曲線の一つの変種を発見した。これでは、正方形の各々の辺を2等分し、元の正方形を四つの小さな正方形に分割する。そして小さな正方形は、同様にしてより小さな四つの正方形に分割され、…と続く。この分割の各々の段階に対して、ヒルベルトは各々の正方形を辿る曲線を与えた。しばしば「ペアノ-ヒルベルト曲線」と呼ばれるヒルベルト曲線とは、この分割処理の極限の曲線である。ヒルベルト曲線は基数2の小数つまり2進小数を使って説明することができる。

図14-1はヒルベルトの1891年の論文に従って、彼の空間充填曲線を説明する最初の3ス

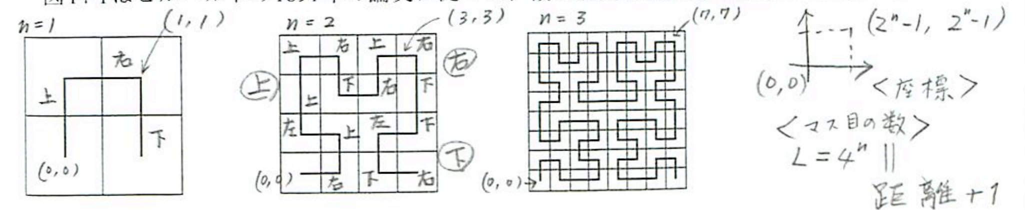


図14-1　ヒルベルト曲線を定義する手順における最初の三つの曲線

1　「曲線」とは、1次元空間からn次元空間への連続的な写像であることを思い出そう。

テップを示している。

ここでは、ちょっと違ったやり方でいく。「ヒルベルト曲線」という語を、極限がヒルベルトの空間充填曲線になる曲線列に属する任意の曲線に対して使っていく。「次数nのヒルベルト曲線」とは、その曲線列のうちのn番目の曲線を意味する。図14-1では、曲線の次数は1、2、そして3である。ここで、曲線の角が、図の四角の各線（辺）が交差する所と一致するように、下と左にずらす。最後に、曲線の角の座標が整数になるように、次数nの曲線を2^nの倍数で拡大する。したがって次数nのヒルベルト曲線は、角のx座標もy座標も0から2^n-1の範囲の整数になる。正の方向を、曲線に沿って$(x, y) = (0, 0)$から$(2^n-1, 0)$になるものとする。図14-2に、我々のいうところの「ヒルベルト曲線」で、次数1から6のものを示す。

14-1 ヒルベルト曲線を生成する再帰的なアルゴリズム

いかにしてヒルベルト曲線を生成するかを見るために、図14-2の曲線を見てみよう。次数1の曲線では、上→右→下と動く。次数2の曲線でも、全体的にこうなる。最初に、結果的に上に行くU形曲線を描く。次に1単位上に行く。3番目にU形曲線、1単位移動、もう一つのU形曲線を、どれも右行きで描く。最後に1単位下に行き、U形曲線を描く。

次数1の逆U形曲線は、次数2のY形曲線に変換される。

任意の次数のヒルベルト曲線を、ある方向への単位長の移動を挟んだ、いろいろな向きのU形曲線の連なりと見なすことができる。ある次数のヒルベルト曲線を次の次数に変形するには、各々のU形曲線を全体として同じ向きを持つY形曲線に変形し、各々の単位長の移動を同じ方向の単位長の移動に変形する。

次数1のヒルベルト曲線（結果的に右行きの時計回りのU形曲線）から次数2のヒルベルト曲線への変形は以下のようになる。

(1) 上行きの反時計回りのU形曲線を描く。
(2) 上に単位長行く
(3) 右行きの時計回りのU形曲線を描く。
(4) 右に単位長行く
(5) 右行きの時計回りのU形曲線を描く。
(6) 下に単位長行く
(7) 下行きの反時計回りのU形曲線を描く。

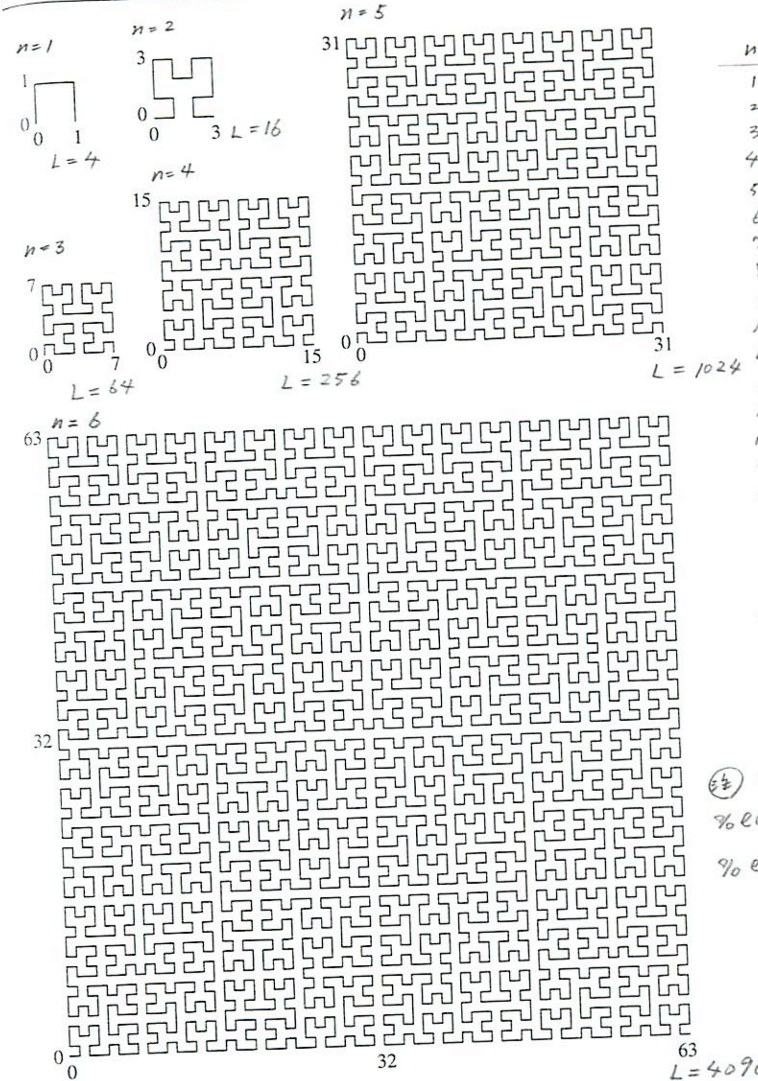


図14-2 次数1から6のヒルベルト曲線

よく見ると、次数1のヒルベルト曲線と同じ向きのすべてのU形曲線は、同じ方法で変形されることが分かる。他の向きのU形曲線についても、類似の規則の組で変形できる。これらの規則を再帰プログラムとして実装したものを図14-3に示す［Voor］。このプログ

```
void step(int);

void hilbert(int dir, int rot, int order) {

   if (order == 0) return;

   dir = dir + rot;
   hilbert(dir, -rot, order - 1);
   step(dir);
   dir = dir - rot;
   hilbert(dir, rot, order - 1);
   step(dir);
   hilbert(dir, rot, order - 1);
   dir = dir - rot;
   step(dir);
   hilbert(dir, -rot, order - 1);
}
```

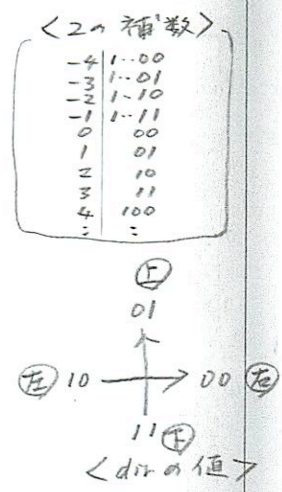
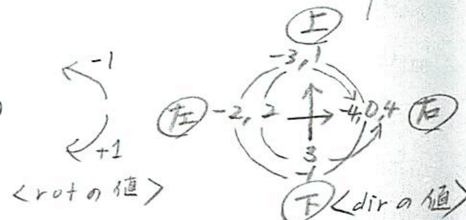


図14-3　ヒルベルト曲線生成器

ラムでは以下のように、U形曲線の向きを、最終的な直線方向と回転方向の二つの整数値で規定している。

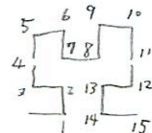
dir=0：右　　　rot=+1：時計回り
dir=1：上　　　rot=−1：反時計回り
dir=2：左
dir=3：下



実はdirは他の値をとり得るが、重要なのはその4を法とした合同値である。

図14-4は、プログラムhilbertで使われるドライバプログラムと関数stepである。このプログラムは、構築されるヒルベルト曲線の次数が与えられると、移動方向、各線分の終端までの、曲線に沿った道のり、それに各線分の終端の座標を含む、線分のリストを表示する。例えば、次数2では以下のように表示される。

```
   0   0000   00 00
   0   0001   01 00
   1   0010   01 01
   2   0011   00 01
   1   0100   00 10
   1   0101   00 11
```




```
#include <stdio.h>
#include <stdlib.h>

int x = -1, y = 0;                  // グローバル変数
int s = 0;                          // 曲線の道のり
int blen;                           // 表示する長さ

void hilbert(int dir, int rot, int order);

void binary(unsigned k, int len, char *s) {
/* 符号なし整数kを2進の文字の形に変換する。結果は長さlenのストリングs */
    int i;

    s[len] = 0;
    for (i = len - 1; i >= 0; i--) {
       if (k & 1) s[i] = '1';
       else       s[i] = '0';
       k = k >> 1;
    }
}
void step(int dir) {
    char ii[33], xx[17], yy[17];

    switch(dir & 3) {
       case 0: x = x + 1; break;
       case 1: y = y + 1; break;
       case 2: x = x - 1; break;
       case 3: y = y - 1; break;
    }
    binary(s, 2*blen, ii);
    binary(x, blen, xx);
    binary(y, blen, yy);
    printf("%5d   %s   %s %s\n", dir, ii, xx, yy);
    s = s + 1;                      // 道のりをインクリメント
}
int main(int argc, char *argv[]) {
    int order;

    order = atoi(argv[1]);
    blen = order;
    step(0);                        // 最初の点を表示
    hilbert(0, 1, order);
    return 0;
}
```

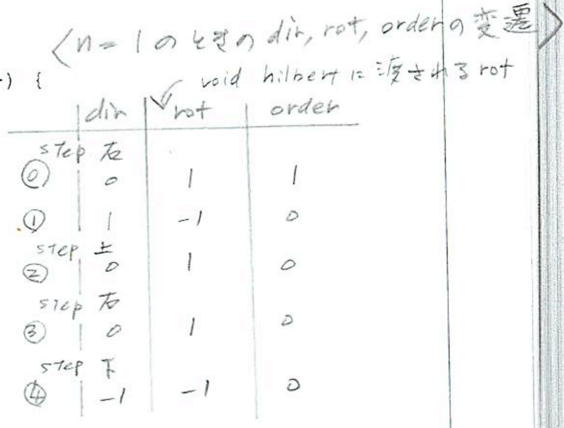
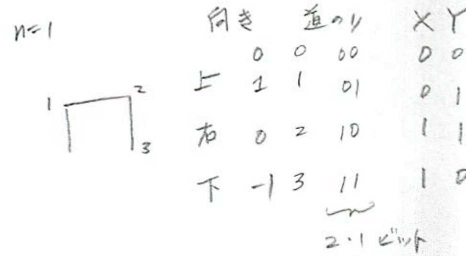


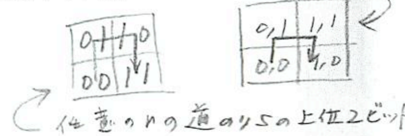
図14-4　ヒルベルト曲線生成器のドライバプログラム



14-2　ヒルベルト曲線の道のりから座標を求める

次数nのヒルベルト曲線で道のりsの地点の(x, y)座標を求めるのに、$2n$ビットの整数sの最上位2ビットが、その地点の存在する象限を決定していることに気づくだろう。これは任意の次数のヒルベルト曲線が次数1の曲線の全体的なパターンに従っているからである。もし最上位の2ビットが00であるなら点は左下の象限のどこかにあり、01なら左上の象限にあり、10なら右上の象限にあり、11なら右下の象限にある。したがって、sの最上位の2ビットは、nビット整数x, yの最上位ビットを以下のように決定する。

sの最上位の2ビット	(x, y)の最上位ビット
00	(0, 0)
01	(0, 1)
10	(1, 1)
11	(1, 0)



どのヒルベルト曲線でも、八つの描き得るU形曲線うちの四つしか現れない。これらを表14-1に、U形曲線の図示とともに、sの2ビットからxとyのそれぞれ1ビットへの対応付けとして示す。

表14-1　四つの可能なマッピング

A	B	C	D
00→(0, 0)	00→(0, 0)	00→(1, 1)	00→(1, 1)
01→(0, 1)	01→(1, 0)	01→(1, 0)	01→(0, 1)
10→(1, 1)	10→(1, 1)	10→(0, 0)	10→(0, 0)
11→(1, 0)	11→(0, 1)	11→(0, 1)	11→(1, 0)

図14-2を見ると、すべての場合において、マップA（⊓↓）で表されるU形曲線は、次の詳細レベルでは、元のレベルのマップAの中での道のり0、1、2、または3に応じて、マップB、A、A、またはDのU形曲線となる。同様にしてマップB（⊐）で表されるU形曲線は、次の詳細レベルでは、元のレベルのマップBの中での道のり0、1、2、または3に応じて、マップA、B、B、またはCのU形曲線となる。

これらの観察から、表14-2の状態遷移表が得られる（状態は表14-1の対応付けの状態に対応）。

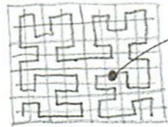
表14-2　sから(x,y)を計算するための状態遷移表

現在の状態	sの次（右側）の2ビット	(x,y)に追加される値	次の状態
A	00	(0, 0)	B
A	01	(0, 1)	A
A	10	(1, 1)	A
A	11	(1, 0)	D
B	00	(0, 0)	A
B	01	(1, 0)	B
B	10	(1, 1)	B
B	11	(0, 1)	C
C	00	(1, 1)	D
C	01	(1, 0)	C
C	10	(0, 0)	C
C	11	(0, 1)	B
D	00	(1, 1)	C
D	01	(0, 1)	D
D	10	(0, 0)	D
D	11	(1, 0)	A

この表を使うには、先ず状態Aから出発する。整数sの先頭には、長さが$2n$になるように0を詰め込んでおく（ここでnはヒルベルト曲線の次数）。sの左から右に、2ビット一組に走査する。表14-2の最初の行は、もし現在の状態がAで、走査したsの2ビットが00ならば、出力は(0, 0)で状態Bに行けということを意味している。そしてsの次の2ビットに移る。同様に次の行は、現在の状態がAで走査されたビットが01であるなら、出力は(0, 1)で状態Aに留まっていろ、ということを意味する。

出力ビットは左から右の順番で集められる。sの最後に達すると、xとyのnビットの出力が決定される。

例えば、$n=3$で



$s=110100$

とする。処理は状態Aから出発し最初に走査されたビットは11であるので、(1,0)を出力し状態Dに入る（4行目）。そして状態Dで01が走査されるので、(0,1)を出力し状態Dに留まる。最後に(1,1)を出力し状態Cに入る。しかしながら、ここでは状態はどうでもよい。

こうして出力は(101,011)、つまり$x=5$、$y=3$となる。

これらの処理をCのプログラムとして実現したものを図14-5に示す。このプログラムでは、現在の状態は0から3の整数を状態AからDにそれぞれ対応させて表現している。変数rowへの代入では、現在の状態とsの次の2ビットを連結して0から15の整数値を得て、表14-2の行を指定する。変数rowは、表14-2の右側の二つの列の値を表すビット列になっている（16進数で表された）整数値をアクセスするのに使われる。つまり、これら列へのアクセスはレジスタ内のテーブル検索（表引き）になっていて、16進の値を左から右に見ると、表14-2の値を下から上に向かって見ていくことになる。

```
void hil_xy_from_s(unsigned s, int n, unsigned *xp,
                                       unsigned *yp) {

   int i;
   unsigned state, x, y, row;

   state = 0;                                  // 初期化
   x = y = 0;

   for (i = 2*n - 2; i >= 0; i -= 2) {        // n回繰り返す
      row = 4*state | (s >> i) & 3;            // テーブルの行
      x = (x << 1) | (0x936C >> row) & 1;
      y = (y << 1) | (0x39C6 >> row) & 1;
      state = (0x3E6B94C1 >> 2*row) & 3;   // 次の状態
   }
   *xp = x;                                    // 結果を
   *yp = y;                                    // 戻す
}
```

図14-5 sから(x,y)を計算するプログラム

W. M. LamとJ. M. Shapiroは全く異なるアルゴリズムを与えている［L&S］。図14-5のアルゴリズムと異なって、これではsのビットを右から左へ走査していく。このアルゴリズムはsの最下位の2ビットを(x,y)に次数1のヒルベルト曲線に基づいてマップすることができるという観察に基づいており、それからsの次の左の2ビットを検査する。もしその値が00なら、計算したばかりのxとyの値を交換しなければならない。これは次数1のヒルベルト曲線を直線$x=y$について折り返す（対称移動する）ことに対応している（図14-1の次数1と2の曲線を参照されたい）。もし二つのビットの値が01か10なら、xとyの値は変更されない。もしその値が11なら、xとyの値は交換され、反転される。sの左側のビットに対しても、以上と同じルールが順々に適用される。これらを実装すると、表14-3と図14-6のプログラムのようになる。いささか奇妙なのは、先ずビットをxやyの先頭にくっつけて、その後で、新たに先頭にくっつけたビットも込みで交換や反転をしてもよいことである。結果は同じになる。

表14-3 sから(x,y)を計算するLamとShapiroの方法

sの左側の次の2ビットの値	操作	(x,y)の先頭にくっつける値
00	xとyを交換	(0, 0)
01	何も変えない	(0, 1)
10	何も変えない	(1, 1)
11	xとyを交換し、反転する	(1, 0)

```
void hil_xy_from_s(unsigned s, int n, unsigned *xp,
                                       unsigned *yp) {

  int i, sa, sb;
  unsigned x, y, temp;

  for (i = 0; i < 2*n; i += 2) {
    sa = (s >> (i+1)) & 1;               // sのi+1ビット目を得る
    sb = (s >> i) & 1;                   // sのiビット目を得る

    if ((sa ^ sb) == 0) {                // sa,sb = 00または11の場合、
      temp = x;                          // xとyを交換
      x = y^(-sa);                       // sa = 1の場合、
      y = temp^(-sa);                    // xとyを反転する
    }
    x = (x >> 1) | (sa << 31);           // saをxの先頭にくっつける
    y = (y >> 1) | ((sa ^ sb) << 31); // (sa ^ sb)をyの先頭にくっつける
  }
  *xp = x >> (32 - n);                   // xとyを右揃えし、
  *yp = y >> (32 - n);                   // 呼び出し側に戻す
}
```

図14-6 sから(x,y)を計算するLamとShapiroの方法

図14-6では、変数xとyは初期化されておらず、コンパイラによってはエラーメッセージを表示するかもしれない。しかしxとyの初期値がどんな値でも、コードは正しく動作する。

図14-6のループ中の分岐は、2-19節の「三つのexclusive or」の技を使った交換を用いて取り除くことができる。このifブロックは以下のコードで置き換え可能である。変数swapとcmplは符号なし整数である。

```
swap = (sa ^ sb) - 1;   // 交換すべき場合は-1、そうでなければ0
cmpl = -(sa & sb);      // 反転すべき場合は-1、そうでなければ0
x = x ^ y;
y = y ^ (x & swap) ^ cmpl;
x = x ^ y;
```

しかしながら、これは9命令になるのに対して、ifブロックは2または6命令になるので、分岐のコストがよほど高くない限りはこのコードは良い選択にはならない。

[L&S]の「交換と反転」のアイディアから、ヒルベルト曲線を生成する論理回路が思い浮かぶ。この回路のアイディアの骨子は、次数nの曲線を経路（パス）に沿って追いながら、sのビットのペアを表14-1のマップAに従って(x, y)にマッピングしていくということである。追っていくうちにいろいろな領域に入って行くと、マッピングの出力は交換されたり、反転されたり、あるいは両者を施される。図14-7の回路図は、各々の段階での必要な交換や反転を追跡し、sの2ビットから(x_i, y_i)への適切なマッピングを行い、次の段階に対して交換や反転を指示する信号を与えている。

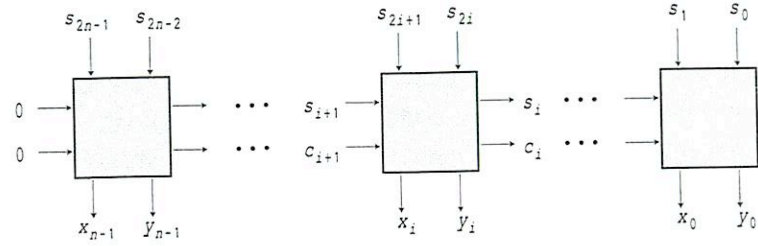
ここではパスの長さsが入るレジスタと、その値をインクリメントする回路があると仮定する。ヒルベルト曲線の次の点を調べるには、まずsをインクリメントし、次にその値を表14-4を使って変換する。この変換が左から右に向かって行う手続きであるのに対して、sのインクリメントは右から左に行うというのは、ちょっと問題である。したがって、次数nのヒルベルト曲線の新しい点を求めるには、$2n$（sのインクリメント）に加えてn（sから(x, y)への変換）に比例する時間がかかるということになる。

図14-7は、これに対する論理回路としての計算を表している。この図ではSは交換信号を意味し、Cは反転信号を意味する。

表14-4 sから(x,y)を計算するロジック

sの右側の次の2ビットの値	(x,y)にくっつける値	swapとcmplの値
00	(0, 0)*	swap = $\overline{\text{swap}}$
01	(0, 1)*	何も変えない
10	(1, 1)*	何も変えない
11	(1, 0)*	swap = $\overline{\text{swap}}$, cmpl = $\overline{\text{cmpl}}$

*交換や反転を施される可能性がある



$$x_i = [s_{2i+1} \oplus (s_{2i} S_{i+1})] \oplus C_{i+1}$$
$$y_i = [s_{2i} \oplus s_{2i+1} \oplus (s_{2i} S_{i+1})] \oplus C_{i+1}$$
$$S_i = S_{i+1} \oplus (s_{2i} \equiv s_{2i+1})$$
$$C_i = C_{i+1} \oplus (s_{2i} s_{2i+1})$$

図14-7 sから(x,y)を計算する論理回路

図14-7の論理回路は、sから(x, y)を計算するもう一つの方法を示唆している。交換や反転の信号が左から右にn段階を経て伝播していくことに注意されたい。これは並列プレフィクス演算を使って、各々の段階の交換と反転の信号の伝播を（$n-1$でなくて$\log_2 n$のステップに）高速化できる可能性を暗示している。したがってxとyを計算するのに図14-7の式を使っていくつかのワードレベル並列論理演算を行う。xとyの値がワードの偶数と奇数のビット位置で混ぜ合わされているので、それらは逆シャッフル演算で分離されなければならない（113ページを参照）。これは、ちょっと込み入っていて、nの値が大きくないと元がとれないように思える。しかしこの動きを見ていこう。

この演算の手順を図14-8に示す[GLS1]。この手順はワード全体を演算の値として使い、最初にsの左に01のビットパターンを詰め込んでおく。このビットの組み合わせは値の交換や反転に影響を与えない。次にcs（complement-swap、反転-交換）の値を計算する。こ

のワードは、cscs...csの形をしており、表14-3に従って、各c（1ビット）は1ならば対応するビットの組を反転することを意味し、各sは対応するビットの組を交換することを意味する。言い換えれば、この二つの文はsのビットの組の各々を以下のようにマッピングする。

s_{2i+1}	s_{2i}	cs
0	0	01
0	1	00
1	0	00
1	1	11

これが我々が並列プレフィクス演算を適用したい値である。左から右へのPP-XORが、ここで使うべき並列プリフィクス演算である。なぜなら、反転か交換かについて、連続する1ビットの持つ意味が、exclusive orと同じだからである。つまり、二つの連続する1のビットは、互いに打ち消し合う。

両方の信号（反転と交換）は、各々がcsの一つおきのビット同士と演算をし合いながら、同じPP-XOR演算で伝播していく。

次の四つの代入文は、xが奇数番目（左側）のビット位置に、yが偶数番目のビット位置に置かれているとして、sのビットの組の各々を(x,y)の値に変換していく働きを持つ。このロジックは分かりにくいかもしれないが、sのビットの各組が、図14-7の最初の二つのブール式で変換されているのを確かめるのは難しくない（注意：tとsrが奇数番目のビット位置では0であるという事実を使って、偶数番目のビット位置と奇数番目のビット位置で変換されていく様子を、個別に考えよ）。

残りの部分の意味は、読めば分かるだろう。この手順は66の基本RISC命令（定数、分岐不用）を要するが、図14-6のコードはおよそ$19n+10$（平均）命令を要する（コンパイル後のコードで。本質的には何もしていない関数出入り口コードも含む）。したがって、$n \geq 3$の場合に、並列プレフィクス法のコードのほうが高速である。


```
void hil_xy_from_s(unsigned s, int n, unsigned *xp,
                                        unsigned *yp) {
   unsigned comp, swap, cs, t, sr;

   s = s | (0x55555555 << 2*n);     // sの左に01の組（何も変えない）を
   sr = (s >> 1) & 0x55555555;      // 詰め込む。
   cs = ((s & 0x55555555) + sr)     // 反転と交換の信号を
        ^ 0x55555555;               // 2ビットの組で計算する
   // 並列プレフィクスxor演算は反転と交換の信号の両方一緒に
   // 左から右に伝播する。
   // "cs ^= cs >> 1"のステップがないので、
   // 結局これは二つの16ビットが一つおきに交互に配置された組に対して
   // 二つの依存しないプレフィクス演算を計算する

   cs = cs ^ (cs >> 2);
   cs = cs ^ (cs >> 4);
   cs = cs ^ (cs >> 8);
   cs = cs ^ (cs >> 16);
   swap = cs & 0x55555555;          // 交換と反転のビットを
   comp = (cs >> 1) & 0x55555555;   // 分離する

   t = (s & swap) ^ comp;           // それぞれ奇数ビット位置と
   s = s ^ sr ^ t ^ (t << 1);       // 偶数ビット位置のxとyを計算する。
   s = s & ((1 << 2*n) - 1);        // 左の不要なものをクリアする

   // ここで「逆シャッフル」をxのビットとyのビットについて行い分離する

   t = (s ^ (s >> 1)) & 0x22222222; s = s ^ t ^ (t << 1);
   t = (s ^ (s >> 2)) & 0x0C0C0C0C; s = s ^ t ^ (t << 2);
   t = (s ^ (s >> 4)) & 0x00F000F0; s = s ^ t ^ (t << 4);
   t = (s ^ (s >> 8)) & 0x0000FF00; s = s ^ t ^ (t << 8);

   *xp = s >> 16;                   // tの半分ずつを
   *yp = s & 0xFFFF;                // xとyに代入する
}
```

図14-8　sから(x,y)を計算する並列プレフィクス法

14-3　ヒルベルト曲線上の座標の距離

ヒルベルト曲線上の点に座標が与えられると、原点からその点までの距離は、表14-2と同じような状態遷移表によって計算できる。表14-5がそのようなテーブルである。

表14-5　(x,y)からsを計算する状態遷移表

現在の状態	(x,y)の次（右側）の2ビット	sの後ろにくっつける値	次の状態
A	(0, 0)	00	B
A	(0, 1)	01	A
A	(1, 0)	11	D
A	(1, 1)	10	A
B	(0, 0)	00	A
B	(0, 1)	11	C
B	(1, 0)	01	B
B	(1, 1)	10	B
C	(0, 0)	10	C
C	(0, 1)	11	B
C	(1, 0)	01	C
C	(1, 1)	00	D
D	(0, 0)	10	D
D	(0, 1)	01	D
D	(1, 0)	11	A
D	(1, 1)	00	C

その解釈は前節のものと同様である。まず、xとyは、nビット長になるよう0を前に詰め込む（ただしnはヒルベルト曲線の次数）。次にxとyのビットは左から右に走査されて、sは左から右へと作り上げられていく。

図14-9はこのステップを実装するCプログラムである。

［L&S］は(x, y)からsを計算するアルゴリズムを与えているが、それは反対方向に進む、sから(x, y)を計算するアルゴリズム（表14-3）に似たものである。すなわち、左から右のアルゴリズムで、表14-6と図14-10に示されている。

表14-6　(x,y)からsを計算するLamとShapiroの方法

(x,y)の次（右側）の2ビット	操作	sの後ろにくっつける値
(0, 0)	xとyを交換	00
(0, 1)	何も変えない	01
(1, 0)	xとyを交換し、反転する	11
(1, 1)	何も変えない	10


```
unsigned hil_s_from_xy(unsigned x, unsigned y, int n) {

   int i;
   unsigned state, s, row;

   state = 0;                                    // 初期化
   s = 0;

   for (i = n - 1; i >= 0; i--) {
      row = 4*state | 2*((x >> i) & 1) | (y >> i) & 1;
      s = (s << 2) | (0x361E9CB4 >> 2*row) & 3;
      state = (0x8FE65831 >> 2*row) & 3;
   }
   return s;
}
```

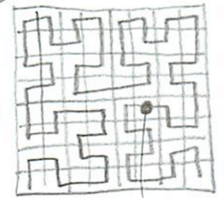


図14-9　(x,y)からsを計算するプログラム

```
unsigned hil_s_from_xy(unsigned x, unsigned y, int n) {

   int i, xi, yi;
   unsigned s, temp;

   s = 0;                             // 初期化
   for (i = n - 1; i >= 0; i--) {
      xi = (x >> i) & 1;              // xのビットiを得る
      yi = (y >> i) & 1;              // yのビットiを得る

      if (yi == 0) {
         temp = x;                    // xとyを交換し、
         x = y^(-xi);                 // xi = 1ならば、
         y = temp^(-xi);              // それらを反転する
      }
      s = 4*s + 2*xi + (xi^yi);       // sの後ろに2ビットをくっつける
   }
   return s;
}
```

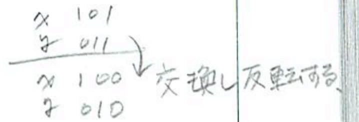


図14-10　(x,y)からsを計算するLamとShapiroの方法

14-4 ヒルベルト曲線の座標のインクリメント

n次のヒルベルト曲線上の点の(x, y)座標が与えられているとして、いかにして次の点の座標を求めたらよいだろうか。一つの方法は、(x, y)をsに変換して、sに1を加え、sの新しい値を上記のアルゴリズムで変換して(x, y)に戻すことである。

（劇的にではなく）ちょっとだけましな方法は、ヒルベルト曲線に沿って移動すると、一つの移動でxかyの両方ではなく片方だけが1インクリメントされるかデクリメントされるという事実に基づいている。これから説明するアルゴリズムは、最も右の2ビットが表すU形曲線の種別を決定するために、座標の値を左から右に走査する。そして、U形曲線と最も右の2ビットの値に基づいて、xかyのどちらか一方をインクリメントするかデクリメントする。

基本的には以上の通りだが、U形曲線の終わりにいる場合（4ステップに1回起こる）は話がややこしくなる。この場合行くべき方向は、xとyの一つ前のビットと、それらのビットに対応するより高い次数のU形曲線で決まる。もしその点も、さらにそのU形曲線の終わりであるなら、その一つ前のビットと対応するU形曲線で決まる。…以下同様。

このアルゴリズムを表14-7に示す。表のA、B、C、Dは表14-1にあるU形曲線をそれぞれ表す。この表を使うには、先ずnをヒルベルト曲線の次数であるとして、xとyの先頭に0を詰め込んでnビットの長さにする。状態Aから出発して、xとyのビットを左から右に走査する。表14-7の最初の行は、現在の状態がAで走査されたビットが(0, 0)であるなら、yをインクリメントすることを表す値に変数をセットし、状態Bに行くということを意味する。マイナス符号が付いた個所はその座標をデクリメントするとして、他の行についても同様に解釈する。3列目にあるダッシュ（—）は、座標の変化を記録する変数を変えないことを意味する。

xとyの最後（最も右側）のビットが走査された後は、変数の最後の値に従って、適切な座標をインクリメントかデクリメントする。

図14-11に、これらの手順を実装したCのプログラムを示す。変数`dx`の初期値は、何度も呼び出されるとアルゴリズムが循環して、同じヒルベルト曲線を繰り返し生成するようになっている（しかしながら一つのサイクルと次のサイクルを結ぶステップは、1単位ステップではない）。

表14-7 ヒルベルト曲線における次の1ステップ

現在の状態	(x,y)の次（右側）の2ビット	増減させる座標	次の状態
A	(0, 0)	y+	B
A	(0, 1)	x+	A
A	(1, 0)	—	D
A	(1, 1)	y−	A
B	(0, 0)	x+	A
B	(0, 1)	—	C
B	(1, 0)	y+	B
B	(1, 1)	x−	B
C	(0, 0)	y+	C
C	(0, 1)	—	B
C	(1, 0)	x−	C
C	(1, 1)	y−	D
D	(0, 0)	x+	D
D	(0, 1)	y−	D
D	(1, 0)	—	A
D	(1, 1)	x−	C

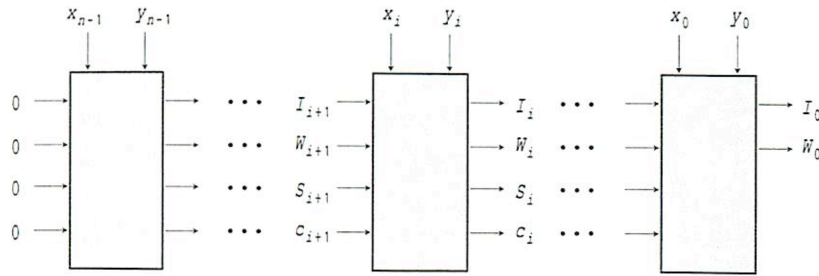
```
void hil_inc_xy(unsigned *xp, unsigned *yp, int n) {

   int i;
   unsigned x, y, state, dx, dy, row, dochange;

   x = *xp;
   y = *yp;
   state = 0;                        // 初期化
   dx = -((1 << n) - 1);             // -(2**n - 1)に初期化
   dy = 0;

   for (i = n-1; i >= 0; i--) {             // n回繰り返す
      row = 4*state | 2*((x >> i) & 1) | (y >> i) & 1;
      dochange = (0xBDDB >> row) & 1;
      if (dochange) {
         dx = ((0x16451659 >> 2*row) & 3) - 1;
         dy = ((0x51166516 >> 2*row) & 3) - 1;
      }
      state = (0x8FE65831 >> 2*row) & 3;
   }
   *xp = *xp + dx;
   *yp = *yp + dy;
}
```

図14-11 ヒルベルト曲線における次の1ステップ



$$X=(S_{i+1}y_i+\overline{S_{i+1}}x_i)\oplus C_{i+1}$$
$$Y=(S_{i+1}x_i+\overline{S_{i+1}}y_i)\oplus C_{i+1}$$
$$I_i=\overline{C_{i+1}}\bar{X}+C_{i+1}XY+I_{i+1}X\bar{Y}$$
$$W_i=\overline{S_{i+1}}\bar{X}Y+S_{i+1}(X\equiv Y)+W_{i+1}X\bar{Y}$$
$$S_i=S_{i+1}\equiv Y$$
$$C_i=C_{i+1}\oplus(X\bar{Y})$$

図14-12　ヒルベルト曲線で(x,y)を1ステップだけインクリメントする論理回路

表14-7は、図14-12に示すように、容易に論理回路で実装できる。この図で、各変数は以下のような意味を持つ。

- x_i：　入力xのiビット目
- y_i：　入力yのiビット目
- X, Y：　S_{i+1}とC_{i+1}に従って、x_iとy_iを交換・反転したもの
- I：　1ならインクリメント、0ならデクリメント（1だけ）
- W：　1ならxをインクリメントするかデクリメントする
  0ならyをインクリメントするかデクリメントする
- S：　1ならx_iとy_iを交換する
- C：　1ならx_iとy_iを反転する

SとCで表14-7の「状態」を決める。$(C, S)=(0, 0), (0, 1), (1, 0)$そして$(1, 1)$であることは、それぞれ状態のA、B、C、そしてDを表す。出力信号はI_0とW_0で、それぞれがインクリメントするのかデクリメントするのかと、どちらの変数を変更するのかを表す（図の回路に加えて、xとyのどちらをインクリメント／デクリメント回路に通すのかをさばく選択回路（MUX）が付いたインクリメント／デクリメント回路や、xやyの値を保持するレジスタへ、

変更された値を書き戻すのをさばく回路が必要である。その代わりとして、二つのインクリメント／デクリメント回路を使うことも考えられる）。

14-5　非再帰型の生成アルゴリズム

表14-2と表14-7は、任意の次数のヒルベルト曲線を非再帰的に生成するアルゴリズムを与えている。いずれのアルゴリズムも大した困難もなくハードウェアで実装可能である。表14-2に基づくハードウェアは、sを保持するレジスタを含み、これは各ステップでインクリメントされ、(x, y)座標に変換される。表14-7に基づくハードウェアは、sのためのレジスタを必要としない代わりに、アルゴリズムが複雑になる。

14-6　その他の空間充填曲線

先に述べたように、ペアノが1890年に最初に空間充填曲線を発見した。それ以降発見された多くの変種は、しばしば「ペアノ曲線」と呼ばれる。ヒルベルト曲線の興味ある変種の一つは、ムーア（Eliakim Hastings Moor）によって1900年に発見された。これは、終点が出発点から1ステップのところにあるという意味で「循環的」である。

次数3のペアノ曲線と次数4のムーア曲線を図14-13に示す。ムーア曲線は、次数1の曲線が上→右→下（┌┐）なのにこの形が他の高い次数の曲線では登場しないという不規則さを持つ。この小さな例外を除けば、ムーア曲線を取り扱うアルゴリズムはヒルベルト曲線のそれと非常に似通っている。

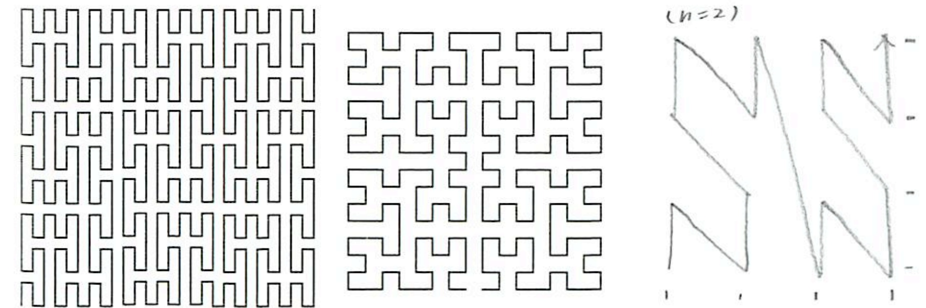
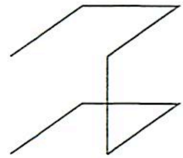


図14-13　ペアノ曲線（左）とムーア曲線（右）

ヒルベルト曲線は任意の長方形や3次元やより高次元の空間に一般化されてきた。3次元のヒルベルト曲線の基本部品を図14-14に示す。これは2×2×2の立方体の八つの点すべてを通る。[Sagan]では、これらを含む他の多くの空間充填曲線について議論されている。

Sagan, Hans: "Space-Filling Curves." Springer-Verlag, 1994



2次元の方がポピュラー。

例えば(1次元の)IPv4アドレスを3次元空間に可視化した ヒートマップと呼ばれる図に応用されている。

IPアドレスはCIDRで管理されているため、管理単位で正方形を為すので極めて見やすくなる。

図14-14 3次元のヒルベルト曲線の基本部品

14-7 応 用

http://www.xkcd.org/195/を見よ!

空間充填曲線は、圧縮やハーフトーン化、テクスチャ解析といった、画像処理へ応用できる[L&S]。他の応用としては、グラフィクスレンダリング技法の一つである、レイトレーシングでの計算効率の改善がある。従来は、風景を投影する光線を、通常はラスタ走査の順序(スクリーンの左から右、次に上から下)で走査していく。光線が風景をシミュレートしているデータベースの物体に当たると、その点における物体の色やその他の属性が決定され、その結果が光線が当てられた点を光らせるのに使われる(この説明はあまりにも単純化されすぎているが、ここでの目的には充分だ)。問題の一つは、データベースはしばしば大きく、走査光線が多くの物体に当たるに応じて、各々の物体のデータをページインするか追い出すかしなければならないことである。光線が走査線を跨いで走査するとき、直前の走査で当たった物体の多くに当たるので、それらは再度ページインされる必要がある。走査にある種の局所性があればページング操作は減るであろう。例えば、ある象限の走査に移る前に別のもう一つの象限の走査が完全に終わっていると、助かると思われる。

ヒルベルト曲線はこの局所性の要求を備えているように見える。ヒルベルト曲線では、別の象限をスキャンする前に別のもう一つの象限を完全に走査し終わっており、走査は再帰的なので、象限から象限へ移動するときに長いジャンプをすることはない。

Douglas Voorhies [Voor] は、従来の単一方向、ペアノ曲線、それにヒルベルト曲線を走査線として追跡を行う場合について、ページングの振る舞いのシミュレーションを行った。彼の方法は、あるサイズの円をランダムにスクリーン上に撒き散らすというものである。走査が円に当たると、新しい物体に当たったということを意味し、それをページインする。しかし走査が円から外れても、走査がその「物体」の円の半径の2倍の中にある間は、物体のデータは主記憶に残っていると仮定する。したがって、もし走査が物体をほんのちょっとの距離だけ離れた後にその物体に戻ってくるならば、ページングが起きないと想定できる。彼はこの実験を多くの異なるサイズの円で、サイズ1024×1024の模擬スクリーンの上で行った。

ある物体の円に入ってから、それを取り囲む円から離れることが、1ページング操作を表すと考えよう。すると従来の走査線では、直径Dピクセルの(そんなに大きくない)円を覆うのに、D回のページング操作を引き起こすのは明らかである。というのは、その円に入る走査線は必ずそれを取り囲む円から出るからである。Voorhiesのシミュレーションの興味ある結果は、ペアノ曲線では一つの円を走査するのにかかるページング操作は約2.7回で、驚くべきことに、おそらく円の直径に依存しないということである。ヒルベルト曲線では約1.4回で、やはり円の直径に依存しない。したがってこの実験の結果は、ページング操作の削減という点で、ヒルベルト曲線はペアノ曲線よりも優れており、普通の走査線よりもはるかに優れているということを示唆している(ページングの回数が円の直径に依存しないということは、たぶん外側の円のサイズが物体に対応する円のサイズに比例していることに起因するでっち上げである)。

(まとめ)
紹介されているアルゴリズムは、一見 $O(4^n)$ の計算量が掛かってしまうような問題が $O(n)$ の計算量で作れてしまう、そのような思慮を重ねた素晴らしいコード群と言える。

(補足)
本文で使われている表記と
巻末の出典
※良書なので あなたも 是非一冊!!

本書の訳語について

本書で使われている訳語で、他の書籍の用語とは異なるかもしれないものをご参考までに列挙しておきます。

原著の語	本書の訳語	他の書籍の用語例
bit/byte reverse	ビット/バイト逆転	ビット/バイト反転
branch	分岐	ブランチ
branch free	分岐不用	分岐なし、分岐不要
condition register	条件レジスタ	コンディションコードレジスタ
index	添字	インデックス
overflow	オーバフロー	溢れ
propagating	伝播	プロパゲーション
rotate-shift	循環シフト	回転シフト
shuffle	シャッフル	攪拌

なお、本書で、演算を表す単語、例えば"add"を「加算」と訳したり、そのまま英綴りにしたりしている理由は、原著でそれぞれ正体および斜体と使い分けているのを訳で反映したからです。つまり、正体のaddを「加算」、斜体の*add*を"add"としています。

1

序　論

1-1 表　記

本書では通常の数学の計算式とコンピュータによる演算を記述する式を区別する。「コンピュータ演算式」では、オペランドはある固定長のビット列またはビットベクタである。コンピュータ演算での式は通常の計算式に似ているが、変数はコンピュータレジスタの内容を意味する。コンピュータ演算式の値は単なるビット列で、特別な意味はない。しかしながら、演算子はオペランドを何らかの方法で独自に解釈する。例えば、比較演算子はオペランドを符号付き2進整数あるいは符号なし2進整数と解釈する。本書のコンピュータ演算表記では符号付きか符号なしかの比較のタイプを明確にするために、はっきりそれと識別できる記号を使用している。

コンピュータ演算と通常の算術計算との主な違いは、コンピュータ演算では、加算(add)、減算(subtract)、乗算(multiply)の結果が、mod 2^n(つまり2^nを法とする)になることである。ここでnはマシンのワード長である。もう一つの違いは、コンピュータ演算には、たくさんの演算があるということだ。四つの基本算術演算に加え、論理積(and)、排他的論理和(exclusive or)、論理比較(compare)、左シフト(shift left、左桁送り)等々がある。

特に指定がない限り、数は、ワード長が32ビットである符号付き整数を2の補数形式で表すものとする。

コンピュータ演算式は通常の計算式と似た形で記述するが、コンピュータレジスタの内容を表す変数はボールド体で表記する。この慣習はベクタ代数で通常使われているものである。コンピュータの1ワードをビット群のベクタと考える。定数も、それがコンピュータレジスタの内容を表すときはボールド体で表記する(これはベクタ代数とは異なる。ベクタ代数では、定数を記述する方法はベクタの各要素を表示する以外にない)。shift命令

の即値フィールドのように、定数が命令の一部である場合は、細字体を使用する。

"+"のような演算子にボールド体のオペランドがある場合、その演算子はコンピュータでの加算（「ベクタ加算」）を表している。オペランドが細字体ならば、演算子は通常のスカラ算術演算を表している。細字体の変数xは、ボールド体変数$\mathbf{x}$の算術的な結果を表すのに使用するが、これは文脈から明確にすべき解釈（符号付きなのか符号なしなのか）に基づいて表記される。すなわち、$\mathbf{x}$=**0x80000000**および$\mathbf{y}$=**0x80000000**の場合、符号付き整数の解釈に基づけば、$x=y=-2^{31}$、$x+y=-2^{32}$、$\mathbf{x}+\mathbf{y}=\mathbf{0}$である。ここで、**0x80000000**は1のビットの後ろに0のビットが31個続くビット列の16進表現である。

ビットは右から番号が付けられ、最右（rightmost、最下位）ビットがビット0である。「ビット」、「ニブル」、「バイト」、「ハーフワード」、「ワード」、「ダブルワード」という用語は、それぞれ1、4、8、16、32、64ビット長を指す。

簡単な短いコードは、代入演算子（左矢印）とたまに`if`文を使ってコンピュータ代数で記述する。コンピュータ代数はアセンブリ言語コードを書くよりも多少マシンから独立した方法として使われる。

それよりも複雑な長いコンピュータプログラムは、C++プログラミング言語で書かれている。C++のオブジェクト指向機能は一切使用していない。つまりプログラムは基本的にISO（ANSI）Cプログラムで、コメントがC++形式なだけである。特に区別する必要がない場合、言語は単に"C"と記述している。

Cの完全な説明は本書の範囲外であるが、表1-1に本書で使用するC［H&S］のほぼすべての構成要素の概要を収めている。これは何らかの手続き型プログラミング言語に精通しているがC言語は知らないという読者の助けになるだろう。表1-1はまた本書で使用するコンピュータ代数演算言語の演算子も示している。演算子は優先順位（緊密束縛）の高いものから低いものへ順番にリストされている。優先順位の列で、Lは左結合、つまり、

$$a \bullet b \bullet c = (a \bullet b) \bullet c$$

を意味する。

Rは右結合を意味する。優先順位と結合性に関して、本書のコンピュータ代数表記はCに従う。

説明で必要な場合、表1-1にある表記の他にブール代数および標準数学表記も使用する。

表1-1 Cおよびコンピュータ代数の式

優先順位	C	コンピュータ代数	説明
	`0x...`	**0x..., 0b...**	16進定数、2進定数
16	`a[k]`		k番目の構成要素を選択
16		$\mathbf{x}_0, \mathbf{x}_1, \ldots$	別々の変数、またはビット選択（どちらかは本文で分かる）
16	`f(x, ...)`	$f(\mathbf{x}, \ldots)$	関数の評価
16		abs($\mathbf{x}$)	絶対値（ただし、abs($-\mathbf{2}^{31}$) = $-\mathbf{2}^{31}$）
16		nabs($\mathbf{x}$)	絶対値のマイナス
15	`x++, x--`		後置インクリメント、後置デクリメント
14	`++x, --x`		前置インクリメント、前置デクリメント
14	(*type name*) `x`		型変換
14 R		$\mathbf{x}^k$	$\mathbf{x}$のk乗
14	`~x`	$\neg\mathbf{x}$、$\bar{\mathbf{x}}$	ビット単位のnot（否定）（1の補数）
14	`!x`		論理not（否定）（if $\mathbf{x}$=**0** then **1** else **0**）
14	`-x`	$-\mathbf{x}$	算術マイナス
13 L	`x*y`	$\mathbf{x} * \mathbf{y}$	乗算、ワード長を法とする
13 L	`x/y`	$\mathbf{x} \div \mathbf{y}$	符号付き整数除算
13 L	`x/y`	$\mathbf{x} \overset{u}{\div} \mathbf{y}$	符号なし整数除算
13 L	`x%y`	rem($\mathbf{x}, \mathbf{y}$)	($\mathbf{x} \div \mathbf{y}$）の剰余（負数も可）、引数は符号付き引数
13 L	`x%y`	remu($\mathbf{x}, \mathbf{y}$)	($\mathbf{x} \overset{u}{\div} \mathbf{y}$)の剰余、引数は符号なし引数
		mod($\mathbf{x}, \mathbf{y}$)	$\mathbf{y}$を法として$\mathbf{x}$を区間[0, abs($\mathbf{y}$)−1]で求める：引数は符号付き引数
12 L	`x + y`、`x - y`	$\mathbf{x}+\mathbf{y}$、$\mathbf{x}-\mathbf{y}$	加算、減算
11 L	`x << y`、`x >> y`	$\mathbf{x} \ll \mathbf{y}$、$\mathbf{x} \overset{u}{\gg} \mathbf{y}$	0充填左シフト、右シフト（「論理」シフト）
11 L	`x >> y`	$\mathbf{x} \overset{s}{\gg} \mathbf{y}$	符号充填右シフト（「算術」または「代数」シフト）
11 L		$\mathbf{x} \overset{rot}{\ll} \mathbf{y}$、$\mathbf{x} \overset{rot}{\gg} \mathbf{y}$	循環左シフト、右シフト
10 L	`x < y`、`x <= y`、 `x > y`、`x >= y`	$\mathbf{x}<\mathbf{y}$、$\mathbf{x} \le \mathbf{y}$, $\mathbf{x}>\mathbf{y}$、$\mathbf{x} \ge \mathbf{y}$	符号付き比較
10 L	`x < y`、`x <= y`、 `x > y`、`x >= y`	$\mathbf{x} \overset{u}{<} \mathbf{y}$、$\mathbf{x} \overset{u}{\le} \mathbf{y}$, $\mathbf{x} \overset{u}{>} \mathbf{y}$、$\mathbf{x} \overset{u}{\ge} \mathbf{y}$	符号なし比較
9 L	`x == y`、`x != y`	$\mathbf{x}=\mathbf{y}$、$\mathbf{x} \ne \mathbf{y}$	等式、不等式
8 L	`x & y`	$\mathbf{x} \& \mathbf{y}$	ビット単位のand（積）
7 L	`x ^ y`	$\mathbf{x} \oplus \mathbf{y}$	ビット単位のexclusive or（排他的論理和）
7 L		$\mathbf{x} \equiv \mathbf{y}$	ビット単位のequivalence（等価）（$\neg(\mathbf{x} \oplus \mathbf{y})$）
6 L	`x | y`	$\mathbf{x} \mid \mathbf{y}$	ビット単位のor（和）
5 L	`x && y`	$\mathbf{x} \overset{\rightarrow}{\&} \mathbf{y}$	条件付きand（if $\mathbf{x}$=**0** then **0** else if $\mathbf{y}$=**0** then **0** else **1**）
4 L	`x || y`	$\mathbf{x} \overset{\rightarrow}{\mid} \mathbf{y}$	条件付きor（if $\mathbf{x} \ne \mathbf{0}$ then **1** else if $\mathbf{y} \ne \mathbf{0}$ then **1** else **0**）
3 L		$\mathbf{x} \Vert \mathbf{y}$	連結
2 R	`x = y`	$\mathbf{x} \leftarrow \mathbf{y}$	代入

本書のコンピュータ代数は、"abs"、"rem"など以外にも関数を使用するが、それらは出てきたときに定義することにする。

Cでは、式x < y < zはx < yを評価して、0/1（0または1）の値を得、それからその結果をzと比較する。コンピュータ代数では、式$x<y<z$は$(x<y)$ & $(y<z)$を意味する。

Cには三つのループ制御文、while、do、forがある。

while文は以下のように記述する

```
while (式) 文
```

最初に式が評価され、**true**（非ゼロ）ならば文が実行され、それから式を再び評価するために制御が戻される。式が**false**（0）ならば、whileループは終了する。

do文もこれに似ているが、ループの一番下で検査が行われる。以下のように記述する。

```
do 文 while (式)
```

最初に文が評価されてから、式が評価される。これが**true**なら、処理は繰り返され、**false**ならループは終了する。

for文は以下のように記述する。

for (e_1; e_2; e_3) 文

最初にe_1が評価される。これは通常代入文である。次にe_2が評価される。e_2は通常比較である。**false**の場合は、forループは終了する。**true**の場合は文が実行される。最後に、これも通常代入文であるe_3が実行され、それから制御が再びe_2の評価に戻る。お馴染みの「do i = 1 to n」は、以下のようになる。

```
for (i = 1; i <= n; i++)
```

（これは後置インクリメント演算子を使う一つの例である。）

1-2 命令セットと実行時間モデル

アルゴリズム間の大まかな比較を可能にするため、Compaq Alpha、SGI MIPS、IBM RS/6000といった現在の汎用RISCコンピュータの命令セットに似た命令セットを備えたマシン用に、アルゴリズムがコーディングされていると考える。マシンは3-アドレスマシンで、かなり多くの、つまり16あるいはそれ以上の汎用レジスタを持つ。特に指定がない場合、レジスタは32ビット長である。汎用レジスタ0には固定的に0が入り、それ以外のレジスタは一律に何にでも使うことができる。

単純にするため、条件レジスタや状況ビットを保持するレジスタといった「専用レジスタ」はない。また、本書の範囲外なので浮動小数点演算については説明しない。

本書では2種類のRISCを認めている。「基本RISC」は表1-2にある命令を持ち、「完全RISC」は基本RISCの全命令に加えて表1-3にある命令を持つ。

この簡単な命令記述では、ソースオペランドのRAとRBは、実際にはそれらレジスタの内容を意味する。

現実のマシンには分岐とリンク（サブルーチン呼び出しに使用）、レジスタ内にあるアドレスへの分岐（サブルーチンからの戻りと「スイッチ」に使用）、おそらくさらに専用レジスタを処理する命令もある。もちろん、たくさんの特権命令とスーパバイザサービスを要求する命令もある。浮動小数点命令もあるかもしれない。

RISCコンピュータに備わっている可能性のあるそのほかの演算処理命令について表1-3に規定しておく。これらは後続の章で説明する。

マシンのアセンブラにいくつか「拡張簡略表記」があると便利である。これらはマクロのように働き、普通単一の命令に展開される。いくつかの例を表1-4に示している。

(注) 欄外書き込みは以下の者が書きました。
山田泰司
E-mail: taiji@aihara.co.jp

ハッカーのたのしみ

本物のプログラマはいかにして問題を解くか

2004年9月17日　初版第1刷発行
2007年4月20日　初版第5刷発行

著　者　ヘンリー・S・ウォーレン、ジュニア
訳　者　滝沢　徹、鈴木　貢、赤池英夫、葛　毅、藤波順久、玉井　浩
発行者　富澤　昇
発行所　株式会社エスアイビー・アクセス
〒183-0015 東京都府中市清水が丘3-7-15
TEL: 042-334-6780／FAX: 042-352-7191
Webサイト：http://www.hh.iij4u.or.jp/~sib-tom/
ダウンロード専用Webサイト：http://sib-download.ddo.jp/~sib/
発売所　株式会社星雲社
〒112-0012 東京都文京区大塚3-21-10
TEL: 03-3947-1021／FAX: 03-3947-1617
印刷製本　三美印刷株式会社
Translation Copyright © 2004 株式会社エスアイビー・アクセス

Hacker's Delight by Henry S. Warren, Jr.

Authorized translation from the English language edition, entitled HACKER'S DELIGHT, 1st Edition, ISBN: 0201914654 by WARREN, HENRY S., published by Pearson Education, Inc, publishing as Addison-Wesley Professional, © 2003

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

JAPANESE language edition published by SIBaccess Co. Ltd., Copyright © 2004.

printed in Japan　　ISBN 978-4-434-04668-3

●本書記載の会社名と製品名等は、それぞれ当該各社の登録商標または商標です。それら団体名、商品名は本書制作の目的のためにのみに記載されており、出版社としては、その商標権を侵害する意志、目的はありません。
●本書の一部あるいは全部について、著作権法の定める範囲を超え、小社に無断で複写、複製することは禁じられています。
●落丁・乱丁本はお取り替えいたします。
●本書の内容に関するご質問は（株）エスアイビー・アクセスまでファックスまたは封書にてお寄せください（電話によるお問い合わせはご容赦ください）。また、本書の範囲を超えるご質問等につきましてはお答えできかねる場合もあります。あらかじめご承知おきください。

SiB access　SiB means *Small is Beautiful and/or Simple is Better.*