ハードウェア設計コンテスト　自由課題部門

Mandelbrot 集合描画支援ハードウェア [Pyxis]

― 最終レポート ―

1994 年 12 月

中　島　　千　明

[概要]

Pyxis は,1 枚の画面描画のために膨大な計算量を必要とする Mandelbrot 集合のコンピュータグラフィックスを高速化するための専用ハードウェアエンジンである.

対象を Mandelbrot 集合のみに絞ったシステム全般にわたる徹底した最適化，アルゴリズムや回路構成の工夫などにより，特殊な部品を用いず，非常に低コストで高い性能を得ることができた.

なお，Pyxis は 1986 年に設計開始され，1988 年に完成した.

○目次

1．製作の目的

まずこの章では，Mandelbrot 集合のコンピュータグラフィックスについて説明し，従来の方法，すなわちソフトウエアによる描画法とその問題点について述べます．そして，ハードウェアによる解決法としてのPyxis の，内容説明への導入とします．

1.1　対象

フラクタル図形の代表として有名なものに，Mandelbrot 集合（以下M集合）のコンピュータグラフィックス（CG）があります．この不可思議な形状には，文字どおりの "無限" が醸し出す想像を超えた美しさ，神秘性が潜んでいます．従来より多数の書籍，雑誌の中で紹介されているほか，それらの CG ばかりを集めた本が出版されていることからも，その不思議な魅力にとりつかれた人の多さが想像できます．

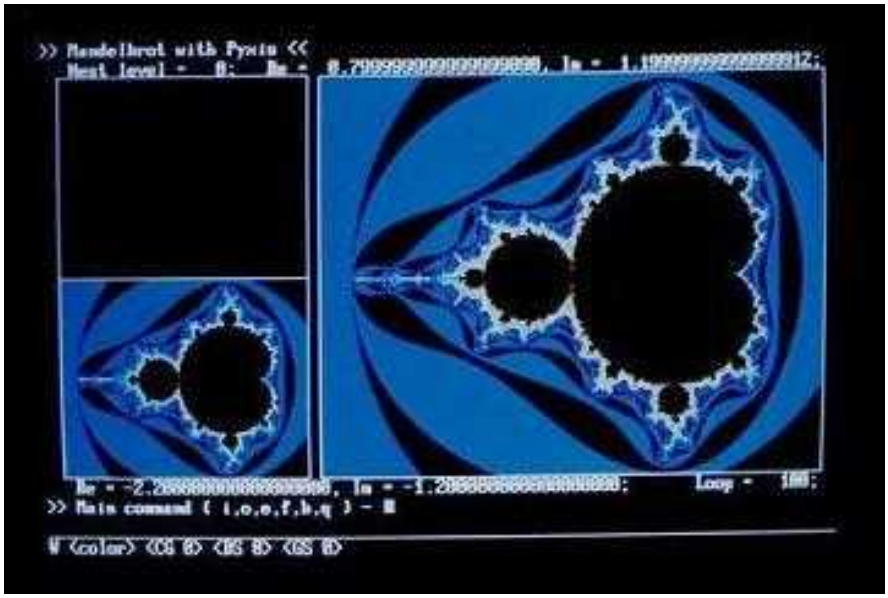
写真 1-1 から写真 1-7 は，Pyxis を用いた描画画面の例です．写真 1-1 はM集合の全体像，写真 1-2 以降は全体像の一部を拡大描画したものです．

ここで，M集合とその描画について簡単に説明しておきます．

Ｚを複素変数，Ｃを複素定数として，次式

$$Z_{n+1}=Z_n^2+C \quad (n=0,1,2,\cdots) \quad \cdots(1\text{-}1)$$

を $Z_0=0$ として反復演算します．この結果，反復回数ｎを無限大としても |Z |が無限に大きくならないＣの集合がM集合です．基本となるルールは，驚くべきことにたったこれだけです．

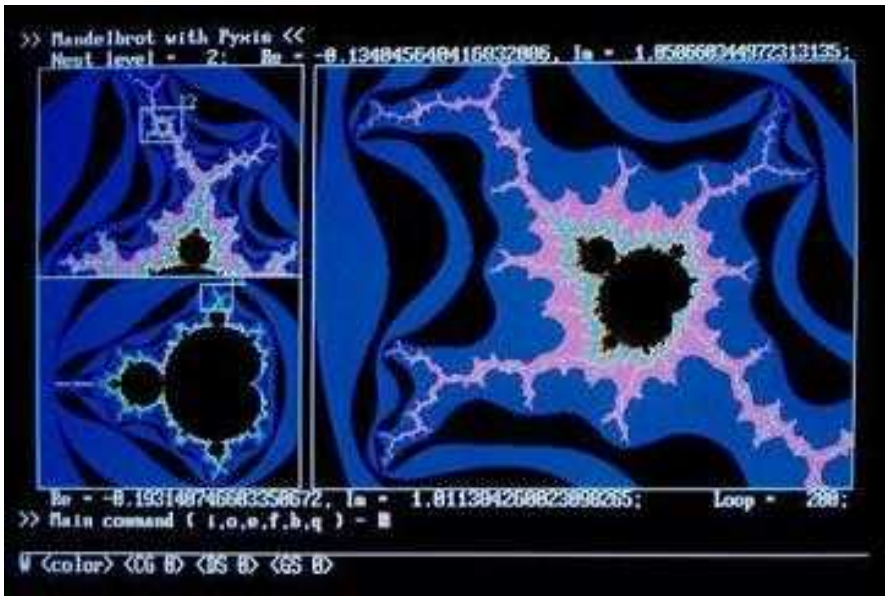


［写真 1-1］

M集合の全体像．

水色で囲まれた，黒い横倒しの雪だるま形状の部分がM集合．

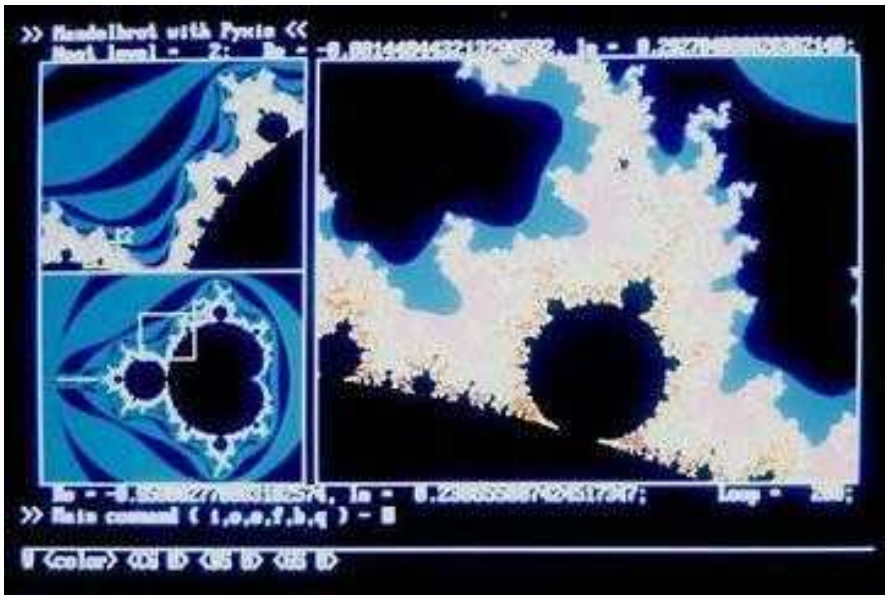
描画対象領域の左下と右上の点の座標がそれぞれ表示されている．



［写真１-2］

M集合上部を拡大したところ．細いヒゲの中に，全体と自己相似な図形があることがわかる．

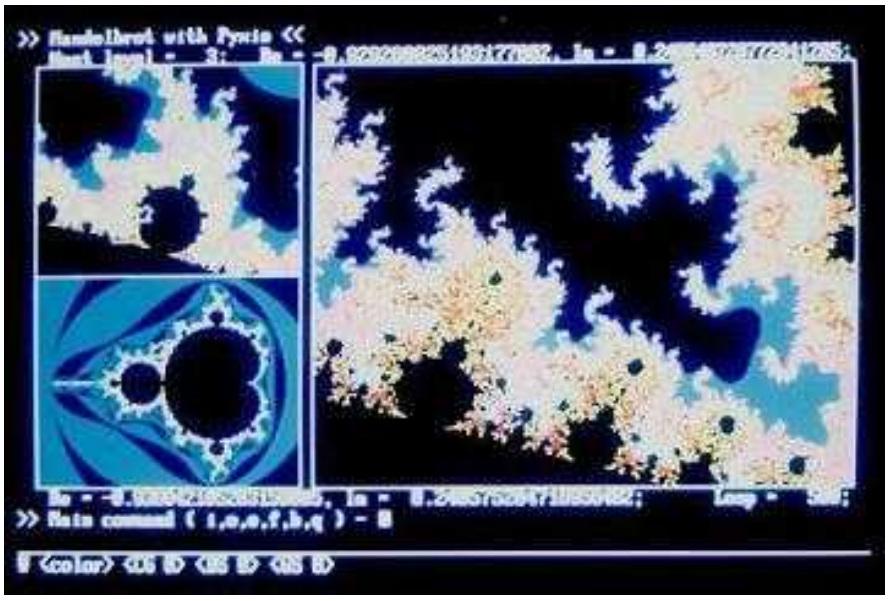
左の小さい枠内には，拡大した範囲が表示されている．



［写真１-3］

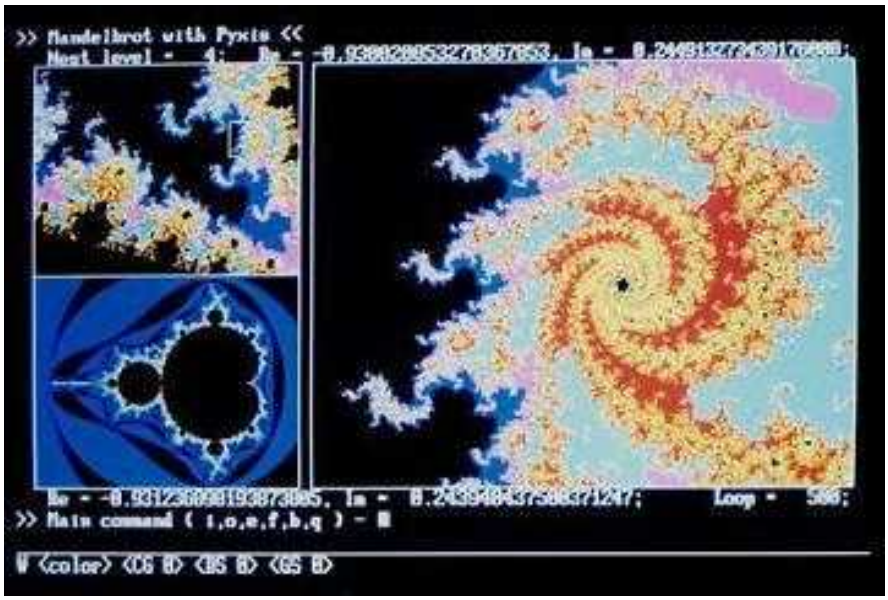
雪だるまの首部の拡大．

周囲にも，全体とよく似た形状が無限に付随していると想像できる．



［写真１-4］

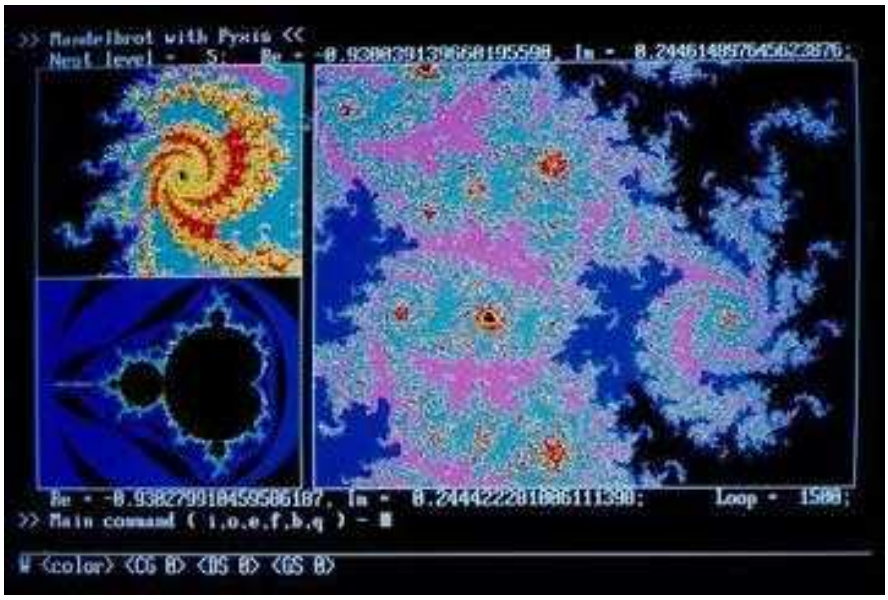
写真１-3 の一部をさらに拡大．



［写真１-5］

写真１-4の一部をさらに拡大．

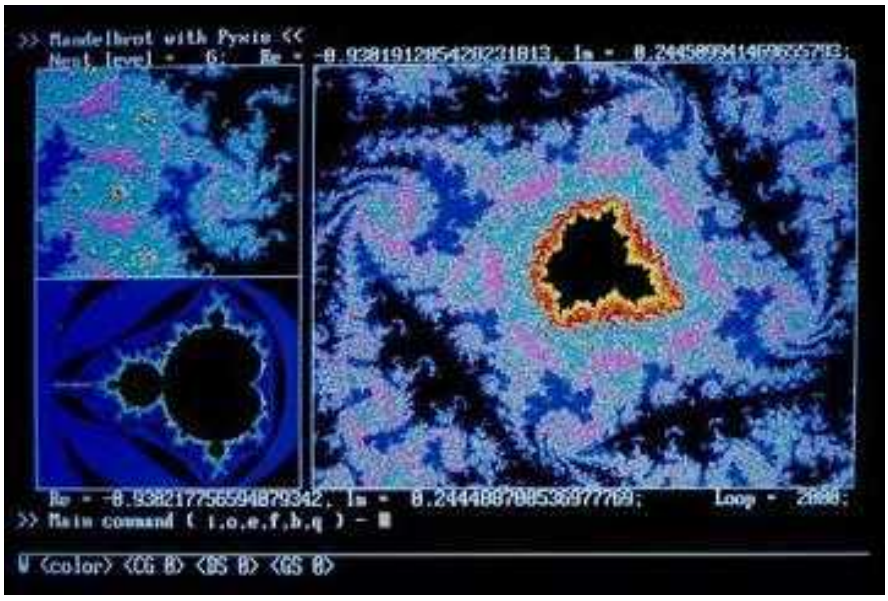
大きな,無限に続く渦が現れる．



［写真１-6］

写真１-5の，渦の一部をさらに拡大．

渦の中は非常に複雑な乱流になっている．

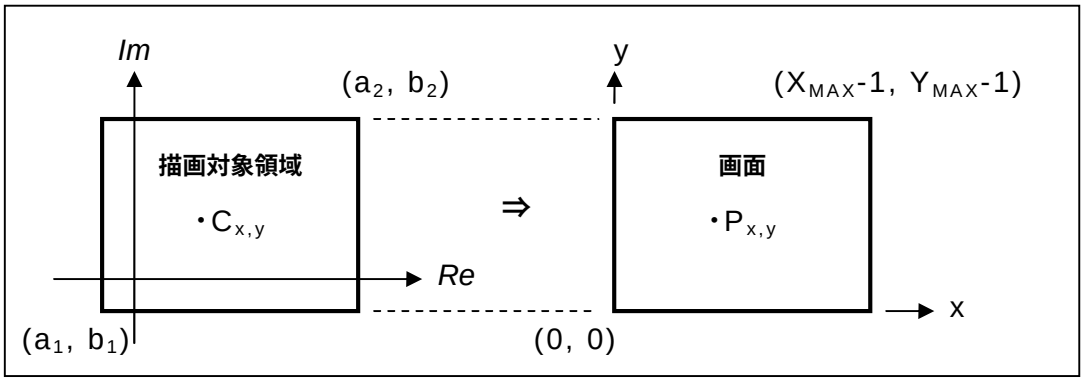


［写真１-7］

写真１-6の一部をさらに拡大．

乱流の中から，本体と自己相似な図形が現れた．その周囲には，複雑ながらも整然とした無限の自己相似図形が集まって，デザイン化された模様のようになっている．

［図 1-1］複素平面内の描画対象領域と表示画面



（3 頁からの続き）

これを画面に描画します．描画法にも種々あるようですが，ここではもっとも一般的と思われる方法について説明します．

図 1-1 のように，複素平面内に，ある描画対象領域 (a_1,b_1)-(a_2,b_2) を考え，これを画面に当てはめます．画面のサイズ（ピクセル数）を $X_{MAX}\times Y_{MAX}$，その中の画面座標 (x,y) のピクセルを $P_{x,y}$ で表すとします．さらに，各 $P_{x,y}$ に対応する描画対象領域内の点を $C_{x,y}$ として，これは

$$C_{x,y}=C_x+C_y i \quad \text{但し 虚数単位 } i^2=-1$$

であるとすれば，C_x，C_y は，

$$\begin{cases} C_x=a_1+x\cdot\dfrac{a_2-a_1}{X_{MAX}} \\ C_y=b_1+y\cdot\dfrac{b_2-b_1}{Y_{MAX}} \end{cases} \quad \cdots(1\text{-}2)$$

さらに，画像の変形をさけるため，x 軸と y 軸のスケールが等しくなるように，

$$\frac{b_2-b_1}{Y_{MAX}}=\frac{a_2-a_1}{X_{MAX}}=S \quad \cdots(1\text{-}3)$$

と置き換えれば，式(1-2)は，

$$\begin{cases} C_x=a_1+S\cdot x \quad (x=0,1,\cdots X_{MAX}-1) \\ C_y=b_1+S\cdot y \quad (y=0,1,\cdots Y_{MAX}-1) \end{cases} \quad \cdots(1\text{-}4)$$

となります．

ところで，実際の演算では無限大を扱うことはできませんから，$|Z|$ と n のそれぞれに対して，無限大とみなすための閾値を設定します．式(1-1)の反復において，ある n の時点で

$$|Z|\geqq 2 \quad \cdots(1\text{-}5)$$

が満たされれば，それ以降の反復により $|Z|$ は無限に大きくなりますので，これを判定基準とします[1)]．また n については，描画結果と描画時間との兼ね合いにより，予め適当な最大反復回数 N_{MAX} を設定しておくものとします．

さて，式(1-4)の $C_{x,y}$ を式(1-1)の C として反復演算を行います．そして N_{MAX} 回までの反復の中で，毎 n につき式(1-5)が満たされるか否かを調べていきます．

反復は $n=N_{MAX}-1$ か式(1-5)の成立にて終了とします．そして，その時の n の値を演算結果とします．これを $N_{x,y}$ とすると，すなわち

$$N_{x,y}=\begin{cases} n & \cdots N_{MAX}-1 \geqq n \text{ にて式(1-5)が成立した場合.} \\ N_{MAX} & \cdots N_{MAX}-1=n \text{ にて式(1-5)が不成立の場合.} \end{cases}$$

この結果に基づき，画面上の 1 ピクセル $P_{x,y}$ を描画します．$N_{x,y}=N_{MAX}-1$ ならば黒で描画し，$N_{x,y}<N_{MAX}-1$ ならば $N_{x,y}$ の値を使って適当な色で描画します．

以上の処理を x , y すべての組み合わせについて行えば，1 画面分の描画図形が得られます．そこでは，M集合の部分は黒で，そしてそれ以外の部分には，Z が無限遠点へ飛び去る速さの違いが色のパターンとなって表現されることになります．

このようにして，様々な対象領域Cに対する画像を得ることが，M集合の CG の目的となります．

（注）筆者が使用しているコンピュータは色数が少ないため，M集合以外の部分にも黒を使用しています．画面写真参照の際，注意してください．

1.2 問題点

以上の処理をソフトウエアで実現するのは簡単です（リスト 1 -1 参照）．しかし，問題は演算量です．

ZもCと同様に

$$Z=Z_x+Z_y i$$

で表されるとすると，Zの反復 1 回に含まれる演算は，リスト 1 -1 に注意しつつ，式(1 -1)より

$$\begin{cases} Z^2+C=(Z_x^2-Z_y^2+C_x)+(2Z_xZ_y+C_y)i \\ \qquad\qquad \cdots \text{反復用} \quad \cdots(1\text{-}6a) \\ |Z|^2=Z_x^2+Z_y^2 \quad \cdots \text{判定用} \quad \cdots(1\text{-}6b) \end{cases}$$

となりますから，最低 8 回の浮動小数点加減乗算が必要です．例えば，

$$X_{MAX}=400,\ Y_{MAX}=320,\ N_{MAX}=100$$

として，すべてのピクセルがM集合に含まれる場合（N_{MAX} 回反復する）を考えると，仮に判定用比較演算や，また正規化，加減算時の桁合わせなどを考えないとしても，演算量は 1 億回を越えます．これを例えば 1 分以内に描画するには，最低限約 2MFLOPS の性能が必要になります．

［リスト 1 -1］M集合の描画処理例

```
var
 x, y, n : word;
 Cx, Cy, Zx, Zy, Zx2, Zy2 : double;

begin
  Cy := b1;
  for y := 0 to Ymax -1 do begin
    Cx := a1;
    for x := 0 to Xmax -1 do begin
      Zx := 0;
      Zy := 0;
      n := 0;
      repeat
        Zx2 := sqr(Zx);
        Zy2 := sqr(Zy);
        Zy := 2 * Zx * Zy + Cy;
        Zx := Zx2  - Zy2 + Cx;
        n := inc(n)
      until ((n = Nmax) or (Zx2+Zy2 >= 4.0));
      draw_pixel(x,y,n);  (*ピクセルの描画*)
      Cx := Cx + S
    end;
    Cy := Cy + S
  end
end.
```

今でこそ 32 ビット MPU やオンチップ FPU，浮動小数点演算 DSP などが当たり前になっていますが，当時（1986 年頃）はまだ 8 ビットマシンを使っている人も多かった頃で，1 枚の画面を得るのに一晩かかるのが普通でした．PC98XL²(386+387)でも数時間，筆者の使っていた自作 CP/M マシンでは十数時間も必要でした．これでは，「この辺を拡大するとどうなっているのだろう」と思っても，結果がでるのは翌日です．これは精神衛生上，非常によくありません．

1.3 解決法

もちろん，高価なハードウェアを用意すればすぐにでも高速化はできますが，それでは面白くありません．そこで，専用ハードウェア（Pyxis）を自作することにしました．当時筆者は貧乏学生だったので，予算を労力でカバーすることを基本方針としました．

では，2 章以降で本論に入ります．Pyxis は回路規模が大きく，動作も複雑ですので，本レポートの説明は考え方を中心に進めたいと思います．

1.4 略記号について

本レポートでは説明簡略のため略記号を多用します．以下に主なものを挙げておきますので適宜参照してください．

a_1	:描画対象領域実軸原点
b_1	:描画対象領域虚軸原点
C	:式(1-1)における複素定数
$C_{x,y}$	:$P_{x,y}$ に対応する C
C_x	:$C_{x,y}$ の実数成分
C_y	:$C_{x,y}$ の虚数成分
CK_{PA}	:部分積加算クロック
L_E	:P_E 用結果レジスタ
L_D	:P_D 用結果レジスタ
n	:式(1-1)における反復回数
$N_{x,y}$	:$P_{x,y}$ についての演算結果
N_{MAX}	:最大反復回数
$P_{x,y}$	:画面座標(x,y)のピクセル
P_E	:ピクセルペアの内 x が偶数のピクセル
P_D	:ピクセルペアの内 x が奇数のピクセル
S	:1 ピクセルに対応する複素平面上の幅
X_{MAX}	:描画画面の x 方向のピクセル数
Y_{MAX}	:描画画面の y 方向のピクセル数
Z	:式(1-1)における複素変数
Z_x	:Z の実数成分
Z_y	:Z の虚数成分
Z_{nx}	:n を意識した Z_x
Z_{ny}	:n を意識した Z_y

2．システム概要

2.1　設計方針

Pyxisの目的は,"M集合のCGを高速に作成する" ためのハードウェアです．これを第一目標として設計するわけですが，当時の状況などの制約事項を考慮し，次のように設計目標を定めました．

①400×320 ピクセル分を 1 分以内で描画できること．

②IC 代で 2〜3 万円が上限．

③特殊な部品は使わない．

④拡大描画に耐える十分な演算精度を持つ．

⑤ホストの性能を期待しない．

全般に最適なトレードオフを得るためには，システム全体の様々な問題に対し上記の条件のもとで，あたかも "ニュートン法を使って連立方程式を解く" ような設計をする必要があります．その "連立方程式" の最適解を目標としてPyxisを設計しました．

2.2　システム的機能

図 2-1 にPyxisのホストとの接続形態を示します．

Pyxis は 8 ビット CPU バスに直結され，単純な I/O として動作します．

まず，ホストからは第 1 章で説明した 4 つのパラメータをPyxisの内部レジスタに設定します．

①描画対象領域実軸原点：a_1（図 1-1 参照）

②描画対象領域虚軸原点：b_1（図 1-1 参照）

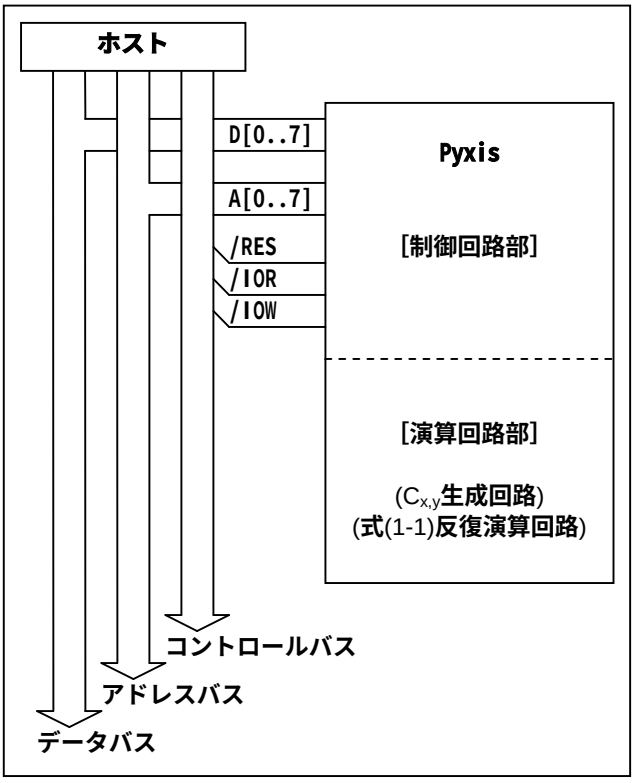
③ステップ：S（式(1-3)参照）

④最大反復回数：N_{MAX}

そしてリセットレジスタをクリアすると，Pyxis は演算を開始します．

この後ホスト側に必要な処理は，Pyxis から順次出力される全ピクセル分の演算結果 $N_{x,y}$ を読み出し，描画していくだけです．ホストの能力に依存せず高速な描画を実現するため，各ピクセルに対応する C_x，C_y の計算など，演算にからむ処理は Pyxis 内部ですべて処理されます．ですから，機能的にはリスト 1-1 の描画部分(draw_pixel())を除いた，すべての処理を取り込んだものとなっています．

［図 2-1］ホストとの接続形態



2.3 動作の概要

前項の通り，Pyxis は機能的にリスト 1-1 をハードウェア化したものとなっています．ここではその動作について概略を説明します．

(1)処理の進み方

処理は $P_{0,0}$ を開始点として，まず x 方向に 0 から X_{MAX}-1 まで，さらにこれを y 方向に Y_{MAX}-1 まで順次進行していきます.この時ピクセル毎の $C_{x,y}$ は，C_x 生成回路及び C_y 生成回路にて処理の進行にあわせて計算されます．

Pyxis はパイプラインにより 2 ピクセル分を同時に処理します（後述）．x 方向への処理進行において，まず $P_{0,0}$ と $P_{1,0}$，次に $P_{2,0}$ と $P_{3,0}$，･･･と 2 ピクセルずつ順次処理されていきます．この同時に処理される 2 ピクセルのことを以後ピクセルペアと呼び，x が偶数，奇数のピクセルをそれぞれ P_E，P_D で表すことにします．

(2)ピクセルペア毎の処理

演算処理はピクセルペア毎に行われます．n=0 から，式(1-1)の反復演算を実行していきます．

その処理の中で，P_E，P_D のそれぞれについて式(1-5)の成立がチェックされます．成立と判定された場合にはそのフラグが制御回路へ渡され，その時の n が結果レジスタ L_E もしくは L_D へロードされます．当然，P_E と P_D が同時に成立するとは限りません．先に成立した方は，他方の処理が終了するまで待たされます．

このようにして，P_E，P_D の両方が式(1-5)成立と判定されるか，または n が N_{MAX}-1 に到達するとそのピクセルペアの処理は終了となります．演算終了ステータスがセットされ，ホストに対して結果レジスタ L_E,L_D の読み出しが可能になったことを知らせます．演算終了ステータスは，L_D レジスタの読み出しでクリアされます．

(3)1 ピクセルペアの処理終了後

演算終了ステータスがセットされた後，Pyxis は直ちに次のピクセルペアの処理を開始します．ホストの読み出しを待ちませんので，回路の動作効率を非常に高めることができます．

しかし万一，ホストの処理が遅くて読み出しが間に合わない場合（すなわち，演算終了ステータスがセットされた状態で，処理中である次のピクセルペア P_E，P_D のどちらかの演算が終了した場合）には，結果レジスタがオーバーライトされる直前の状態で回路全体が停止し，ホストの読み出しの完了を待つようになっています．これにより，ホストが遅くても演算結果を読み落とす心配はありません．

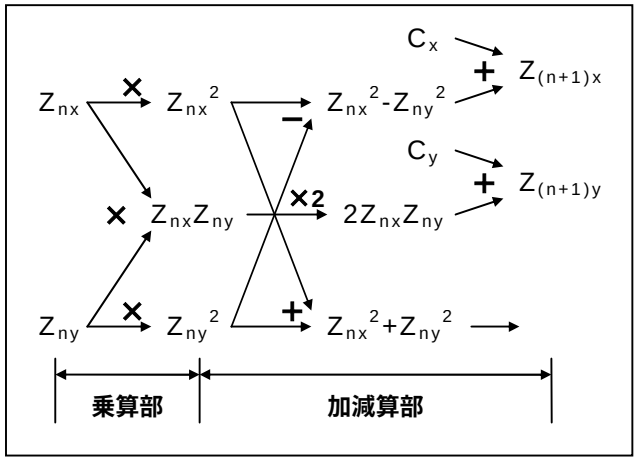
○ここまでで，Pyxis 全体の構成及び動作の概略を説明しました．次の章では，具体的な回路の説明の前に必要となる，数値データの表現法などシステム全体に共通する基本的な重要要素について説明します．

3．システム設計

3.1　演算フローの検討

式(1-6)及びリスト 1-1 をデータフローを中心に考えると，図 3-1 のように表すことができます．

［図 3-1］演算フロー



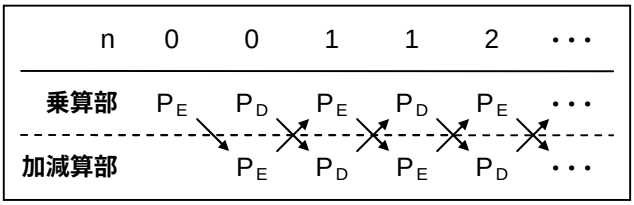
(1)パイプライン化

この図から，反復 1 回分の演算は前半の乗算部と後半の加減算部に分割でき，かつそのそれぞれが，互いの演算結果を利用し合う形のループになっていることがわかります．よって，乗算部と加減算部をパイプライン化することで，それぞれを休むことなく動作させることができます．そのため，このパイプラインループに 2 ピクセル分のデータをいれ，乗算部と加減算部を交互に回るように全体を構成しました．

まず P_E が乗算部へ入り，その結果が加減算部へ送られると同時に乗算部では P_D の処理が開始されます（図 3-2）．以後これを反復し，P_E が乗算部に戻ってくる毎に n が 1 増えることになります．

このように，P_E，P_D のそれぞれについて式(1-1)の反復演算が同時に行われます．

［図 3-2］パイプラインとピクセルペア



これにより通常動作中の演算回路の利用効率を 100%とすることができます．

(2)並列処理

図 3-1 のフローには乗算 Z_{nx}^2，$Z_{nx}Z_{ny}$，Z_{ny}^2 を中心とした，次の 3 つの大きな流れがあることがわかります．

$$\begin{cases} Z_{nx}^2-Z_{ny}^2+C_x & \cdots(3\text{-}1a) \\ 2Z_{nx}Z_{ny}+C_y & \cdots(3\text{-}1b) \\ Z_{nx}^2+Z_{ny}^2 & \cdots(3\text{-}1c) \end{cases}$$

これらはそれぞれ，式(1-6a)の実部と虚部，式(1-6b)に対応しています．

主な演算部を，この乗算を中心とした 3 つに分解し，その各部が並列に動作するようにしました．乗算器を 3 つ用意し，そのそれぞれから同時に出力される結果を，3 分割された各加減算部が並列に利用しています．なお式(3-1b)の 2 倍演算は，$Z_{nx}Z_{ny}$ の乗算結果をステージ間レジスタにロードする部分で 1 ビットずらしておくことで処理しています．

以上のように，演算フローの中の並列性を，コスト他の制限が許す範囲で最大限に利用しています．それによって，全体の処理速度，演算回路の利用効率をそれぞれ向上させています．

3.2　数値のデータ表現

演算数値データのフォーマットは，2 の補数固定小数点 64 ビットにしました．ビット割り当ては，符号 1 ビット，整数部 3 ビット，小数部 60 ビットです．

(1)ビット配分

式(1-5)の条件により，M集合の演算は実数部虚数部ともに±2 の範囲で考えれば十分です．これを有効利用すれば，無用に大きな数を表現する必要がなくなり，整数部のビット数を最小限に押さえられます．実際には，式(1-5)の判定を $Z_x^2+Z_y^2$ だけでなく，それぞれ 2 乗する前の Z_x，Z_y においても判定するようにして，整数部を 3 ビットに抑えています．これについては次項で説明します．

(2)2 の補数表現

Pyxis の乗算回路には，2 の補数をそのまま乗算できる 2 次 Booth アルゴリズムを用いています．これにより，演算回路のすべてでデータ表現を 2 の補数に統一することができます．符号反転や絶対値などの余分な回路を省略し，かつ加減算と相性の良い最適な表現法といえます．

(3)ビット数と固定小数点表現

データ長に 64 ビット用意したのは，非常に大きな拡大率の図形に対して十分な反復回数の演算ができるようにするためです．この固定小数点表現なら有効桁は十進数で 18 桁以上ありますので，対象の複素平面に対して一様（これが重要）に十分な精度が得られます．浮動小数点方式では，表現できる数値が対数的であることに加え，1.0 付近の数値を表現すると指数部が無意味になることから，ハードウェアが複雑になる以上にそのメリットはありません．

3.3　式(1-5)の判定法

(1) $Z_{nx}^2+Z_{ny}^2$ の判定

$|Z_n|$を得るためには$(Z_{nx}^2+Z_{ny}^2)^{1/2}$ を計算する必要がありますが，式(1-5)の判定が目的ならもちろんこれは不要で，次式の成立を調べれば十分です．

$$|Z_n|^2=Z_{nx}^2+Z_{ny}^2 \geqq 4 \qquad \cdots(3\text{-}2)$$

(2)判定法の改良

しかし，3.2 で説明したように，64 ビットの数値データを有効に使うためには，できるだけ小数部のビット数が多い方が有利です．

仮に，あるｎ-1 回目における式(3-2)の判定が

$$Z_{(n-1)x}^2+Z_{(n-1)y}^2=3.999\cdots$$

にて不成立となったとすると，次回(n)の判定の時には，例えば，

$$\begin{cases} Z_{(n-1)x}=\pm 1.999\cdots, & C_x=+1.999\cdots, \\ Z_{(n-1)y}=0.0, & C_y=+1.999\cdots \end{cases}$$

の時に，

$$Z_{nx}^2+Z_{ny}^2=39.9999\cdots$$

となってしまいます．この数を表せるようにするには，整数部に 6 ビットが必要です．

このビット数を削減するため,2 乗の前の Z_{nx}，Z_{ny} をそれぞれチェックする回路を追加しました．もちろんこのためのハードウェア量が大きければ本末転倒ですが,論理的にはわずかゲート 3 個で実現可能です．

式(3-1)において，

$$\begin{cases} |Z_{nx}| \geqq 2 & \cdots(3\text{-}3) \\ |Z_{ny}| \geqq 2 & \cdots(3\text{-}4) \end{cases}$$

のいずれかが成り立つならば，式(3-2)も成立します．

この式(3-3),式(3-4)の判定を，Z_{nx}，Z_{ny} が乗算部に入る直前に行います．このことは，式(3-2)の評価

の一部を1反復サイクル前に先取りして行うことと同じです．従って，式(3-3),式(3-4)の判定結果は制御回路にて 1 サイクル分遅延した後，式(3-2)と併せて評価するようにします．

これによるワーストケースは，式(3-3)，式(3-4)のどちらもを満たさない最大値，

$$\begin{cases} Z_{nx}=\pm 1.999\cdots, \\ Z_{ny}=\pm 1.999\cdots \end{cases}$$

の場合で，

$$Z_{nx}^2+Z_{ny}^2=7.9999\cdots.$$

以上より，式(3-3),式(3-4)の判定を加えることで，整数部のビット数を 3 ビットに抑えながら，式(1-5)の正確な判定が可能になります．

(3)実際の判定法

ここでは，数値データをビット展開して，次のように表現します．

$$D_0\ D_1\ D_2\ D_3.D_4\ D_5\ \cdots\cdots\ D_{62}\ D_{63}$$

↑
小数点

①式(3-2)の判定

判定基準は正数で 4 以上ですから簡単です．

式(3-2)=D_0+D_1.

②式(3-3),(3-4)の判定

式(3-3)=$(D_0+D_1+D_2)\cdot(\overline{D_0}+\overline{D_1}+\overline{D_2})$.
式(3-4)も同様．

このように，簡単な論理回路にて判定が可能です．

○ここまでで，構成，機能，動作，方法を，それぞれ抽象的な表現を中心に概説しました．次の章からは，これらがどのように具体化されているかの解説に移ります．まず全体のブロックダイヤグラムを示し，その中の構成要素を順に説明していきます．

4. 機能ブロックの解説

4.1 システムブロック

Pyxis の全回路は 4 つの階層に分けられた機能ブロックの集合体として構成されています．この階層構造については後述しますが，まず第 1 階層は次の 6 つの機能ブロックで構成されています．

①C_x 生成回路 ····· Ox
②C_y 生成回路 ····· Oy
③$Z_x^2-Z_y^2+C_x$ 演算回路 ····· Xx
④$2Z_xZ_y+C_y$ 演算回路 ····· Yy
⑤$Z_x^2+Z_y^2$ 演算回路 ····· Rr
⑥制御回路 ····· Cn

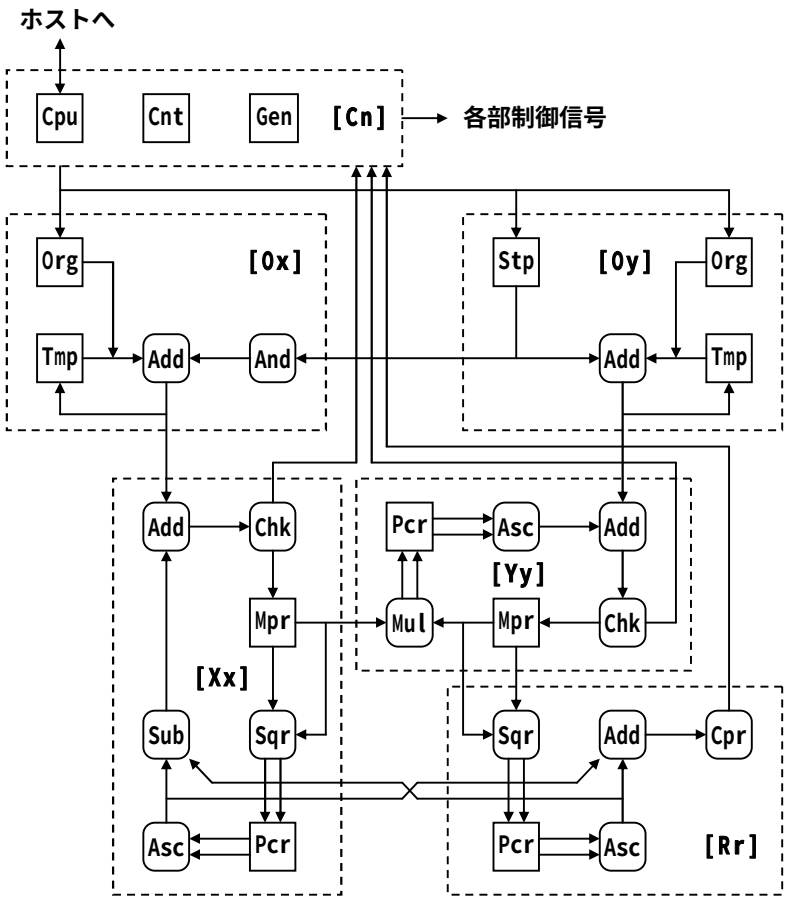
以後これらの部分を適宜右の 2 文字の略語で表します．

Ox,Oy は各ピクセルに対応する C_x，C_y を生成します．Xx,Yy,Rr は各ピクセルについて式(1-1)の反復演算を行う部分です．それぞれ，式(3-1a),式(3-1b),式(3-1c)を演算します．Cn はホストインターフェースを含む全体制御回路です．

図 4-1 に Pyxis のブロックダイヤグラムを示します．この図には，第 2 階層までが表現されています．制御回路部を除けば，図 3-1 の演算フローに忠実な構成になっています．

以降で，このシステムブロックを構成している各機能ブロックを説明していきますが，個別の説明に入る前に，まず演算回路部①～⑤に共通の基本要素である"加算・減算回路"と"乗算回路"について説明します．

［図 4-1］全体ブロック図



○Cn-Cpu：CPU インターフェース回路
○Cn-Cnt：全体制御用同期カウンタ
○Cn-Gen：制御信号生成回路

○Ox-Org:描画対象領域実軸原点 a_1 保持レジスタ
○Ox-Tmp：C_x 生成用テンポラリレジスタ
○Ox-Add：C_x 生成用加算回路
○Ox-And：C_x 生成用データゲート

○Xx-Mpr：Z_x 用乗数,被乗数レジスタ
○Xx-Sqr：Z_x^2 用乗算回路
○Xx-Pcr：Z_x^2 用乗算結果,残留桁上げレジスタ
○Xx-Asc：Z_x^2 残留桁上げ加算回路
○Xx-Sub：$Z_x^2 - Z_y^2$ 用減算回路
○Xx-Add：$(Z_x^2 - Z_y^2) + C_x$ 用加算回路
○Xx-Chk：式(3-3)判定回路

○Oy-Org：描画対象領域虚軸原点 b_1 保持レジスタ
○Oy-Tmp：C_y 生成用テンポラリレジスタ
○Oy-Add：C_y 生成用加算回路
○Oy-Stp：C_x, C_y 生成用 S 保持レジスタ

○Yy-Mpr：Z_y 用乗数,被乗数レジスタ
○Yy-Mul：$Z_x Z_y$ 用乗算回路
○Yy-Pcr：$2Z_x Z_y$ 用乗算結果,残留桁上げレジスタ
○Yy-Asc：$2Z_x Z_y$ 残留桁上げ加算回路
○Yy-Add：$2Z_x Z_y + C_y$ 用加算回路
○Yy-Chk：式(3-4)判定回路

○Rr-Sqr：Z_y^2 用乗算回路
○Rr-Pcr：Z_y^2 用乗算結果,残留桁上げレジスタ
○Rr-Asc：Z_y^2 残留桁上げ加算回路
○Rr-Add：$Z_x^2 + Z_y^2$ 用加算回路
○Rr-Cpr：式(3-2)判定回路

4.2　加算・減算回路

(1)状況の有効利用

パイプライン化により，加減算部でも乗算部と同じ演算時間を使うことができます．視点を変えて，これを有効に使えば，ハードウェアの規模を小さく押さえることができます．乗算より加減算の方が所要時間は短く，処理も少ないですから，許される時間内に演算が終わる程度のハードウェアを用意すれば十分です．

(2)アルゴリズム：加算回路

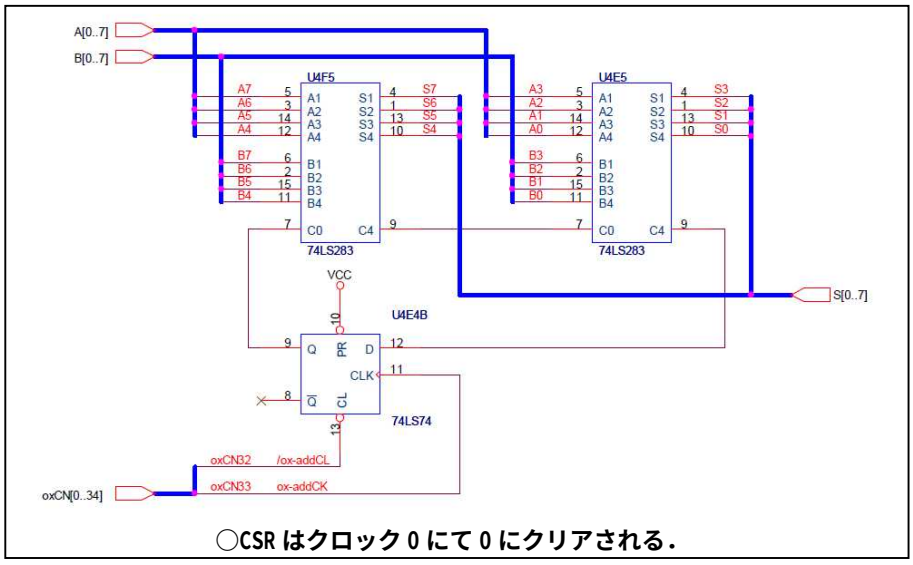
図 4-2 に Pyxis の加算回路を示します．64 ビットデータを 8 ビット毎 8 回に分けて，LSB 側よりこの回路に通します．毎回の桁上げは，桁上げ保存レジスタ CSR に保存され，次回にキャリー入力として加算されるようにします．これにより，64 ビット分の加算器を 8 ビット分のハードウェアで実現できます．もちろんこの回路のためには，64 ビットデータを 8 ビット毎にマルチプレクスして出力し結果をロードする回路がこのほかに必要となります．が Pyxis ではパイプライン化のためのステージ間レジスタがいずれにしても必要ですので，それにこの機能をあわせ持たせてあります．

(3)減算回路

また，この回路構成は簡単に減算回路にも対応できます．第 2 オペランド側の入力をビット反転し，CSR の初期値を 1 にします．これで 2 の補数となりますので，後は加算器と同様です（図 4-3）．

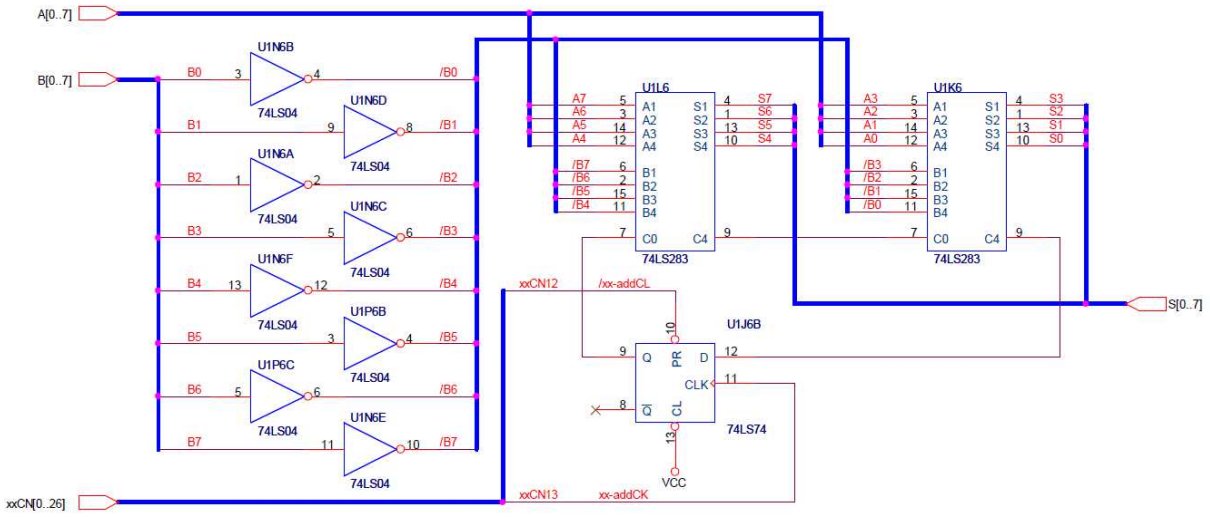
このように，加減算回路においては許容された時間を有効に利用することと，桁上げ保存を応用することで，ハードウェアの規模を大幅に縮小しています．

［図 4-2］加算回路



○CSR はクロック 0 にて 0 にクリアされる．

［図 4 -3］減算回路



◯CSR はクロック 0 にて 1 にセットされる.

4.3　乗算回路

(1)アルゴリズム

乗算には種々の方法がありますが，Pyxis では 2 次の Booth アルゴリズムを利用しました．

このアルゴリズムを用いると，64 ビットの固定小数点数を，ビット数の 1/2 の 32 クロックで，しかも 2 の補数表現のまま演算できることが大きな特長です．他の方法，例えば単純なシフト加算方式では処理に 64 クロックを必要とすることに加え，乗算器の入力を予め正数にしておく必要があります．またその際，符号は別に処理しなければなりません．加減算を含む各演算回路のデータ表現を統一することはシステム的に非常に有効です．そのためにも，乗算に Booth のアルゴリズムを使うことはこの場合最適といえます．

(2)構成と動作

①被乗数，乗数レジスタ

乗算回路の基本ブロック図を図 4-5 に示します．

被乗数レジスタは 64 ビットパラレル出力です．

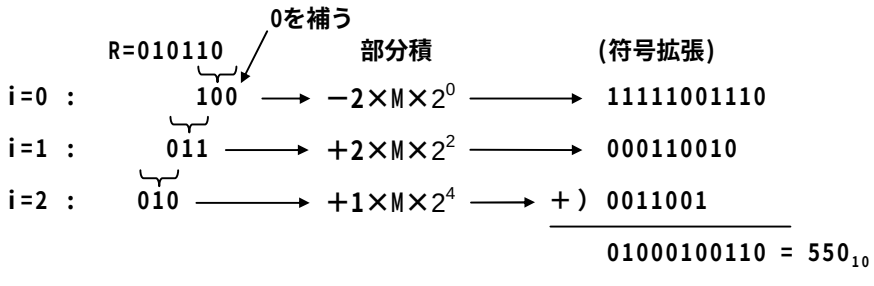
乗数はシフトレジスタにロードされますが，これは 2 次 Booth のため 2 ビットずつシフトする必要があります．そこで半分のサイズのシフトレジスタを 2 系列用意し，偶数番ビットと奇数番ビットを分けてそれぞれロード／シフトしています．そしてそのシフト出力（LSB 側）の 3 ビットを乗数側データとして 2 次 Booth 乗算器に送っています．

［図 4-4］2 次の Booth アルゴリズム

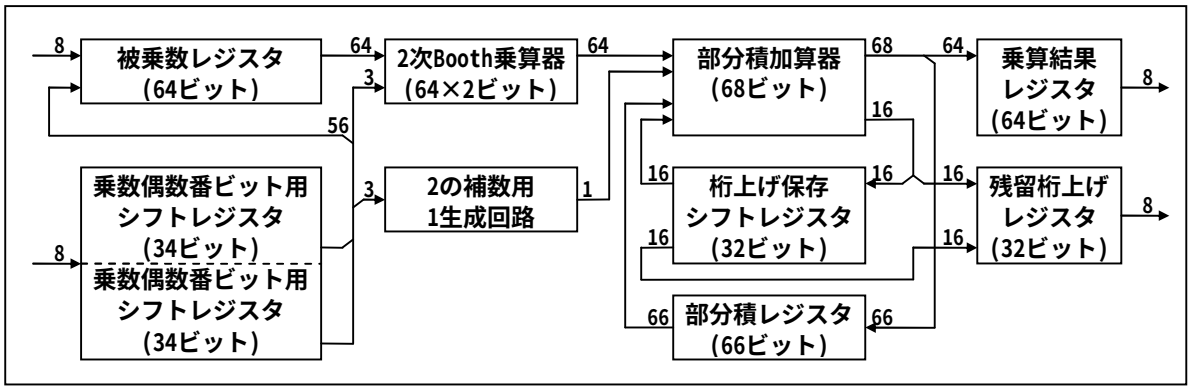
まず乗数の LSB 側に 0 を補う．つぎに LSB 側より，1 ビット重複しながら 3 ビットずつ評価する．この 3 ビットのパターンにより右の表の部分積を生成し，加算する．

評価中の3ビット R_{2i+1}	R_{2i}	R_{2i-1}	部分積
0	0	0	0
0	0	1	$+1\times M\times 2^{2i}$
0	1	0	$+1\times M\times 2^{2i}$
0	1	1	$+2\times M\times 2^{2i}$
1	0	0	$-2\times M\times 2^{2i}$
1	0	1	$-1\times M\times 2^{2i}$
1	1	0	$-1\times M\times 2^{2i}$
1	1	1	0

例として，乗数 $R=22_{10}=010110_2$，被乗数 $M=25_{10}=011001_2$ の場合を考える．



［図 4-5］乗算回路のブロック図



この両レジスタは，パイプラインのステージ間レジスタとしても機能しています．レジスタの入力側は加減算部（具体的には前項の加算・減算回路）と接続されており，そこからの出力は 64 ビットの LSB 側より 8 ビットずつ 8 回に分けて送られてきます．乗数シフトレジスタは乗算処理の進行により MSB 側から空いてきます．この空きが 8 ビットになる瞬間に，加算・減算回路からの 8 ビット出力がそこへロードされるタイミング構成になっています．結果，例えば P_E の乗算処理が終了した時点では，この乗数シフトレジスタには P_D のデータがすでに 64 ビット分ロードされていることになります．そして同時に，この最後の 8 ビットロードのタイミングで，乗数シフトレジスタにロード済みの 56 ビットとこの最後の 8 ビットを合わせて被乗数レジスタへロードし，被乗数を更新しています．

②2 次 Booth 乗算器

2 次 Booth 乗算器の実現には，LS261 を用いています．これにより，図 4-4 にある被乗数 64 ビット×乗数 2 ビット（±2,±1,0）の部分積が得られます．
ただし，生成される部分積が M×(-2, -1)の場合の LS261 からの出力は 1 の補数でしかないことに注意します．図 4-5 の中に "2 の補数用 1 生成回路" がありますが，ここで 2 の補数にするための 1 を生成しています．

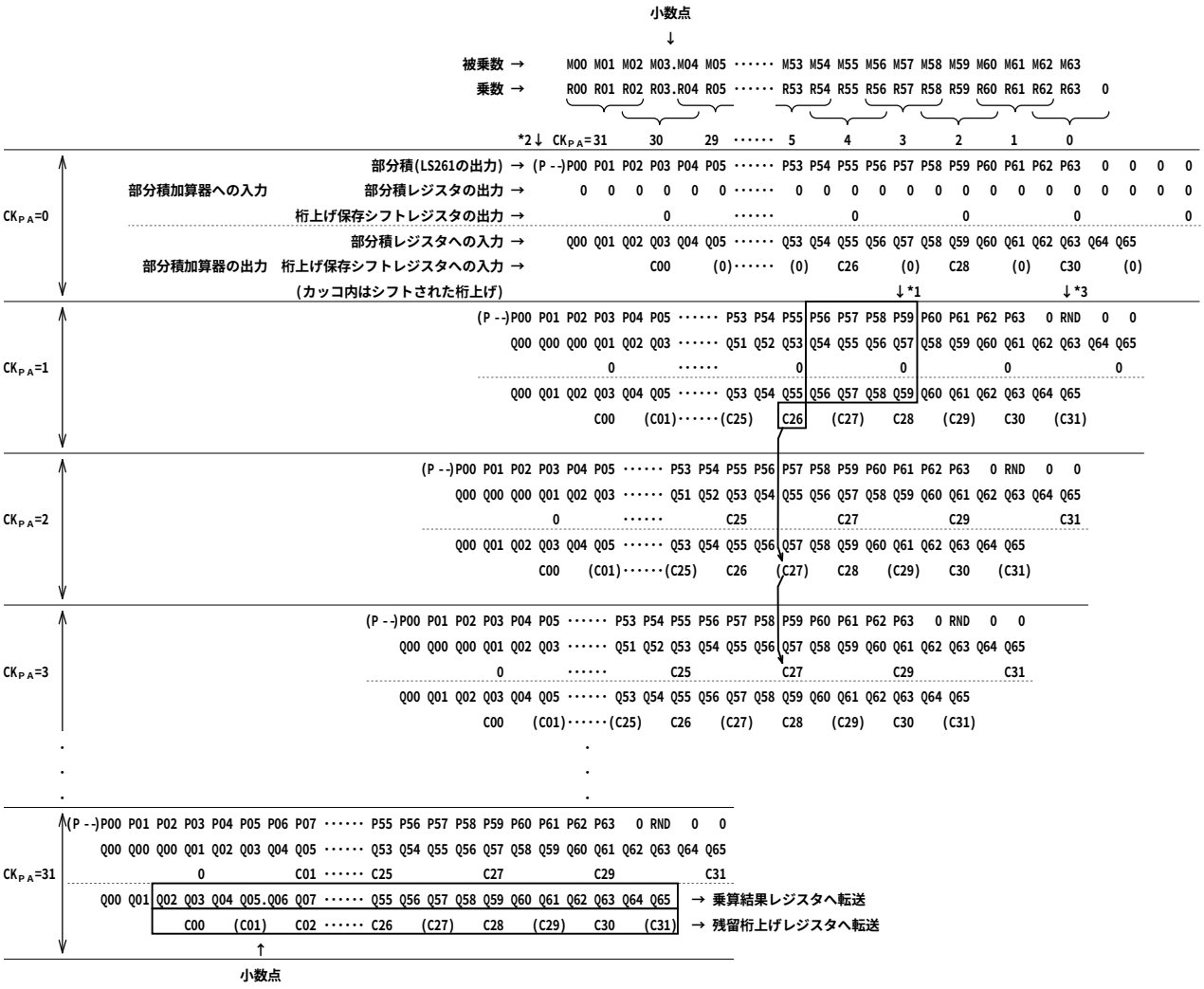
③高速部分積加算

次にこの部分積を加算します．部分積レジスタはクロック 0 でクリアされ，以後クロック 31 まで部分積加算器の出力をクロック毎に一時的に保存します．以後，この部分積加算クロックを CK_{PA} と表記します．

部分積加算器は LS283 を 17 個使った 68 ビット加算回路ですが，このキャリーを 1 クロック中ですべて伝搬させると非常に時間を消費してしまいます．そこで，ここでも桁上げ保存を応用しています．これにより 64 ビット加算でありながら，キャリー伝搬のための無駄な時間を完全に排除しています．

図 4-6 に，乗算回路の動作を示します．この図を用いて，部分積加算における桁上げの高速処理を説明します．

［図 4 -6］乗算回路の動作－高速部分積加算



***1:本文参照．**

***2:原理上では，被乗数を±2 倍したとき，ここのビットの処理が必要である．しかし，3.3 の式(1 -5) 判定法により，意味のある乗数，被乗数の絶対値は 2.0 未満であるので， M00 と M01 は常に同一である．よって，このビットの処理は不要である．**

***3:RND は 2 の補数補正用ビットで，本文②の説明における 2 の補数用 1 生成回路の出力がここで加算される．このビットは本来 P63 と同じ位置へ同時に加算されるべきものであるが，加算器にとって 4 つめのオペランドとなるため加算できない．そのため，桁上げ保存に似た処理で 1 加算サイクル遅らせた後，本来の位取り位置に合わせるため 2 ビットずらして加算している．**

例として，部分積の中の LS283 1 個分（P56～P59：図中*1）の，CK_{PA}＝1 での動きで説明します．

4 ビット全加算器 LS283 の入力には，それぞれ

・A4～A1：部分積レジスタの出力 Q54～Q57

・B4～B1：LS261 から出力された部分積 P56～P59

・C0(桁上げ入力)：C27 の出力(現在初期化により 0)

が入力され，その出力として Q56～Q59 が得られています．この出力は次の CK_{PA}＝2 の開始タイミングで，部分積レジスタの Q58～Q61 として 2 ビットずらしてロードされます．

これと同時に，LS283 の桁上げ先見出力 C4 は，桁上げ保存シフトレジスタの C26 にロードされます．そしてこの C26 の内容は，次の次の加算クロック CK_{PA}＝3 の開始タイミングで C27 にシフトされ，その出力が桁上げとして加算されます．

LS283 が 4 ビットの加算器であることと，乗数のシフトが 2 ビットずつであることから，保存された桁上げは本来の位取り位置に合わせるため 2 つ後の加算クロックにて処理しています．

少しややこしい処理ですが，桁上げを伝搬させないので高速な部分積加算が実現できます．このほかの動作は，図中の説明を参照してください．

④乗算結果レジスタ・残留桁上げレジスタ

乗算結果レジスタは 64 ビットパラレル入力のレジスタです．これには CK_{PA}＝31 の部分積加算器の出力が，次の CK_{PA}＝0 の開始タイミングでロードされます．これが乗算結果ですが，一部の桁上げが桁上げ保存シフトレジスタに残っていますので，これも同時に残留桁上げレジスタ（32 ビットパラレル入力）にロードします．そしてパイプラインの加減算部の先頭でこの両者を加算し，最終的な乗算結果を得ています．

この両レジスタは，乗数・被乗数レジスタ同様パイプラインのステージ間レジスタとしても機能しています．両レジスタの出力側がそれぞれ 8 ビットのバス接続になっており，64 ビットを 8 ビットずつ 8 回に分けて，時分割で加減算部へ送っています．

乗算回路の速度は，Pyxis の性能を左右する重要な要素です．以上説明したように，2 次 Booth アルゴリズムを用いることで，2 の補数 64 ビット同士の乗算を 32 回の部分積加算で実行します．また桁上げ保存の応用により，64 ビットの部分積加算を行いながらも高速な動作が可能です．

実際の動作は，LSTTL ベースの設計ながら，部分積加算クロックは 12.5MHz です．基本的にはシフト加算方式ですが，64×64 ビットを 2.56μsec で演算しています．

このほか，パイプラインステージ間レジスタと，乗数･被乗数･乗算結果･残留桁上げの各レジスタを共用しています．さらにそれらへのロード／ストアを部分積加算クロックと同一タイミングで実行しています．データ転送のための無駄なクロックタイミングがないので，すべてのクロックに対して演算回路は 100%の効率で動作します．

○次の項から，各機能ブロックの説明に入ります．その中で出てくる演算回路は，基本的に前述の加算・減算・乗算回路です．その組み合わせにより，大きな機能ブロックが構成されています．

4.4　Ox:C_x 生成回路

ここは，式(1-4)の0から X_{MAX}-1 までの各xに対応する C_x を，処理の進行に合わせて生成する部分です．

Oxのブロック図を図4-7に示します．

Org,Tmp,Stp はトライステート出力の 64 ビットレジスタで，出力は8ビット幅のバス構造になっています．これは4.2の加算回路構成である Ox-Add 及び Xx の加減算部への対応のため，8 ビットずつ，8 回に分けた時分割出力としています．

さて式(1-4)の乗算項は，もちろん乗算する必要はありません．x=0 に対応する Org を初期値として，これに Stp を加算し，結果を Xx の加減算部へ送ると同時に Tmp へ保存します．この Tmp と Stp の加算と保存を x の 1 増加毎に行うことで C_x を生成します．

これが C_x 生成法の基本ですが，パイプラインへの対応のためもう一工夫します．

And は 8 個の And ゲートで構成されており，8 ビットデータ(S)をそのまま通過させるか(×1)，または出力を 0 とするか(×0)を制御するためのゲートです．

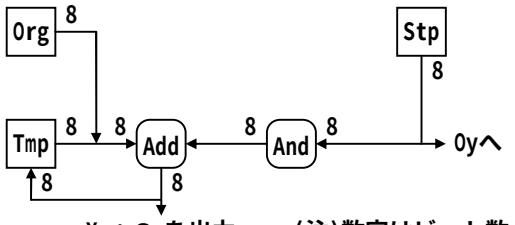
Xx の加減算部には，反復演算により P_E と P_D のデータが交互に回ってきますので，C_x の値もそれに同期して交互に生成する必要があります．このため，まず Tmp には P_E に対応する C_x だけが保存されるようにします．そして Stp は常時出力しておき，And の制御により 0 を加算するかSを加算するかを切り替えます．

- P_E の時：And を 0 にして P_E の C_x+0 を出力．
- P_D の時：And を 1 にして P_E の C_x+S を出力．

この間，Tmp は変更されません．

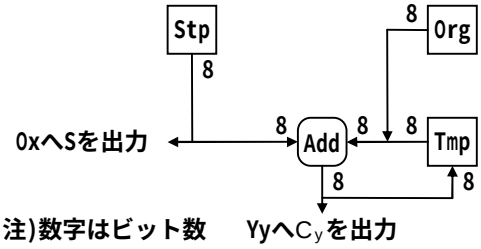
そしてピクセルペアの処理終了時に，Tmp にSを 2 回加算します．反復終了時の P_b に対応する出力 C_x+S を Tmp に保存し，さらに続いて開始される次のピクセルペアの処理の第 1 サイクルで And を 1 にすると同時に出力を Tmp にロードします．これで，次のピクセルペアに対する Tmp が準備されます．

［図 4-7］ Ox のブロック図



○Ox-Org：描画対象領域実軸原点a_1保持レジスタ
○Ox-Tmp：C_x生成用テンポラリレジスタ
○Oy-Stp：C_x，C_y生成用Sレジスタ(Oyと共用)
○Ox-Add：C_x生成用加算回路
○Ox-And：C_x生成用データゲート

［図 4-8］ Oy のブロック図



○Oy-Org：描画対象領域虚軸原点b_1保持レジスタ
○Oy-Tmp：C_y生成用テンポラリレジスタ
○Oy-Add：C_y生成用加算回路
○Oy-Stp：C_x，C_y生成用Sレジスタ(Oxと共用)

4.5 Oy:C_y 生成回路

ここは，式(1-4)の 0 から Y_{MAX}-1 までの各 y に対応する C_y を，処理の進行に合わせて生成する部分です．

ここの動作は Ox と同様ですが，パイプラインに関する処理が不要なのでその機能はありません．単純に y の 1 増加毎に Stp と Tmp の加算と保存を行って，C_y を更新しています．

Oy のブロック図を図 4-8 に示します．

4.6 Xx:Z_x^2-Z_y^2+C_x 演算回路

ここには，Z_x の 2 乗のための乗算回路と，Z_x^2-Z_y^2+C_x のための加減算回路があります．また，2 乗の前の式(3-3)のチェックがあります．

ブロック図を図 4-9 に示します．

4.7 Yy:$2Z_xZ_y$+C_y 演算回路

ここには，Z_xZ_y のための乗算回路と，$2Z_xZ_y$+C_y のための加減算回路があります．また Xx 同様，2 乗の前の式(3-4)のチェックがあります．

ブロック図を図 4-10 に示します．

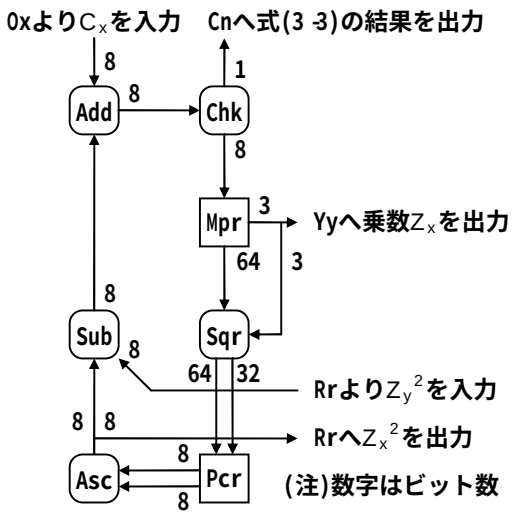
4.8 Rr:Z_x^2+Z_y^2 演算回路

ここには，Z_y の 2 乗のための乗算回路と，Z_x^2+Z_y^2 のための加減算回路があります．また，式(3-2)のチェックがあります．

ブロック図を図 4-11 に示します．

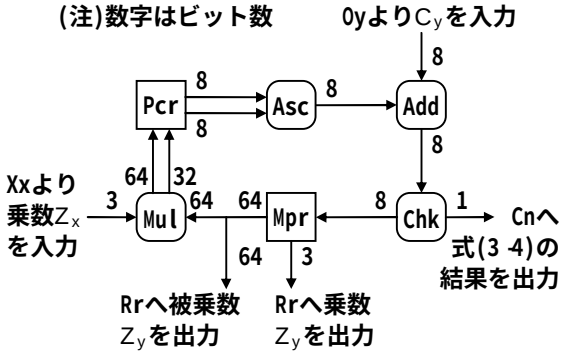
ここには乗数，被乗数レジスタはありません．Xx,Yy 内のレジスタ出力を同時に利用する事で，無駄な回路を省いています．

［図 4-9］ Xx のブロック図



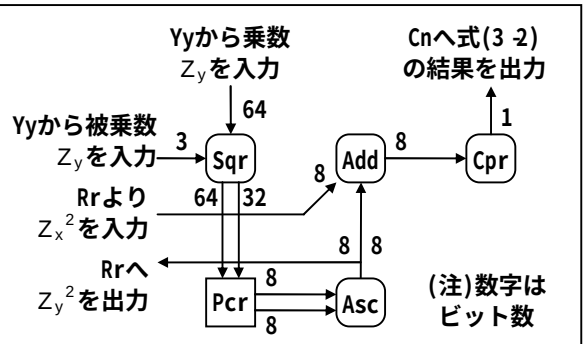
○Xx-Mpr：Z_x用乗数,被乗数レジスタ
○Xx-Sqr：Z_x^2用乗算回路
○Xx-Pcr：Z_x^2用乗算結果,残留桁上げレジスタ
○Xx-Asc：Z_x^2残留桁上げ加算回路
○Xx-Sub：Z_x^2－Z_y^2用減算回路
○Xx-Add：(Z_x^2-Z_y^2)+ C_x用加算回路
○Xx-Chk：式(3-3)判定回路

［図 4-10］ Yy のブロック図



○Yy-Mpr：Z_y用乗数,被乗数レジスタ
○Yy-Mul：Z_xZ_y用乗算回路
○Yy-Pcr：$2Z_xZ_y$用乗算結果,残留桁上げレジスタ
○Yy-Asc：$2Z_xZ_y$残留桁上げ加算回路
○Yy-Add：$2Z_xZ_y$+C_y用加算回路
○Yy-Chk：式(3-4)判定回路

［図 4-11］ Rr のブロック図



○Rr-Sqr：Z_y^2用乗算回路
○Rr-Pcr：Z_y^2用乗算結果,残留桁上げレジスタ
○Rr-Asc：Z_y^2残留桁上げ加算回路
○Rr-Add：$Z_x^2+Z_y^2$用加算回路
○Rr-Cpr：式(3-2)判定回路

［表 4-1］内部レジスタ

<入力>	I/O アドレス
リセットレジスタ (ビット 0 のみ有効:1 でリセット)	080H
描画対象領域実軸原点 a_1 レジスタ(8 バイト)	0B0H～0B7H
描画対象領域虚軸原点 b_1 レジスタ(8 バイト)	0A0H～0A7H
C_x, C_y 生成用ステップ S レジスタ(8 バイト)	0A8H～0AFH
最大反復回数 N_{MAX} レジスタ(2 バイト)	088H,090H
<出力>	**I/O アドレス**
結果レジスタ L_E(2 バイト)	09AH～09BH
結果レジスタ L_D(2 バイト)	09DH～09EH
演算終了ステータスレジスタ (ビット 0 のみ有効:1 で終了)	098H

4.9 Cn:制御回路

Cn は Pyxis 各部の制御信号すべてを生成している部分で，次の 3 部分より構成されています．

- Cn-Cpu：ホストインターフェース
- Cn-Cnt：制御信号生成用同期カウンタ
- Cn-Gen：制御信号生成回路

(1)Cn-Cpu：ホストインターフェース

2.2 での説明の通り，Pyxis は 8 ビット CPU バスに接続され，単純な I/O として動作します．表 4-1 に，内部レジスタの一覧を示します．これら内部レジスタへのインターフェース回路，アドレスデコーダなどが主な内容です．

なお,X_{MAX}，Y_{MAX} は特に変更する必要性を感じなかったので内部で固定されており，それぞれ 400,320 です．

(2)Cn-Cnt：制御信号生成用同期カウンタ

ここは，制御信号を生成するために必要となる，内部状態をカウントする同期カウンタ部です．

機能上これは 4 つに分けられており，分周比の小さい方から，

①パイプライン 1 段の中の処理クロックである，
　部分積加算クロックカウンタ（5 ビット）と，
　ピクセルペア区別用カウンタ（1 ビット）
②式(1-1)の反復回数カウンタ（20 ビット）
③ x カウンタ（12 ビット）
④ y カウンタ（12 ビット）

です．

しかし回路構成上では，この 4 つはシーケンシャルに接続された 52 ビットの完全同期カウンタになっており，12.5MHz の共通クロックで動作します．

通常の回路構成ではこのビット数を 12.5MHz でカウントすることはできませんので，ちょっと凝った回路

にしてみました．カウンタ IC に非同期キャリー出力の LS161 を 13 個使用し，それ 1 個に AND ゲートを 2 個ずつ付加して桁上げ先見しています．これにより飛躍的にクロックレートを上げることができます．

原理は簡単ですので説明は省略します．回路図を参照してください．

(3)Cn-Gen：制御信号生成回路

ここは，次の 3 つの部分で構成されています．

①Cn-Gen-Gn1～4：制御信号生成回路

②Cn-Gen-Syn：パイプライン制御回路

③Cn-Gen-Cnr：演算結果レジスタ

①Cn-Gen-Gn1～4：制御信号生成回路

前項の Cn-Cnt の出力を受けて，Pyxis 各部の状態を制御する信号を生成，出力する部分です．設計当時の資料ですが，この制御信号と出力条件の一覧を付録 1 に示します．

②Cn-Gen-Syn：パイプライン制御回路

主に，2.3 で説明した反復演算の終了処理に関する部分です．正直言って，ここの動作は複雑すぎて簡単に説明できません．この反復演算の終了シーケンスには 10 通りの場合があり（CPU の読み出しが遅い場合の一時停止の有無や，式(1-5)判定回路 3 つからの信号のパターン，P_E が先に終了，P_D が先に終了･･･の組み合わせ），そのそれぞれについての動作があります．設計時にも，動作，タイミングの検証にずいぶん時間がかかりました．

③Cn-Gen-Cnr：演算結果レジスタ

ピクセルペア P_E，P_D のそれぞれに対する演算結果がストアされるレジスタです．ホストの処理を軽減するため，反復が N_{MAX} 回行われたかどうかを示す 1 ビットのフラグ MAXCNT が，結果の MSB にアサインされています．

4.10　回路図の構成

Pyxis の回路図は OrCAD の階層構造を使ってかかれており，4 階層全 67 枚で構成されています．1 枚 1 枚がそれぞれ機能ブロックになっており，図面番号がその機能ブロックの内容を表現しています．

○図面番号の付け方

・図面番号はファイル名と共通で，最大 8 文字．

・第 1 階層は 1 枚のみで，図面番号は［Pyxis］．

・第 2 階層以下は，機能ブロックの名称を次のように付ける．

図面番号＝［2233344］．

第 2 階層は 2 文字で，Ox,Oy,Xx,Yy,Rr,Cn のいずれか．

第 3 階層は 3 文字で，所属する第 2 階層名称の後に付加する．第 4 階層も同様．

ここまでの説明の中で出てきた各機能ブロックの名称はこのような形で図面番号と対応しており，それぞれ階層と機能がわかるようになっています．

付録 5 に回路図の階層情報を示します．

付録 6 に Pyxis の全回路図を示します．

5．タイミング設計

タイミング設計として本章を設けましたが，具体的な回路の動作タイミングの説明については非常に繁雑となるため省略します．それに関してはタイムチャートを示すことにとどめ，ここではそのタイムチャートの表記法について説明します．そしてそれが，デバイスの実力値をあてにせず最大遅延時間に基づいたものであることにより，動作タイミングがきちんと確認されていることを示します．

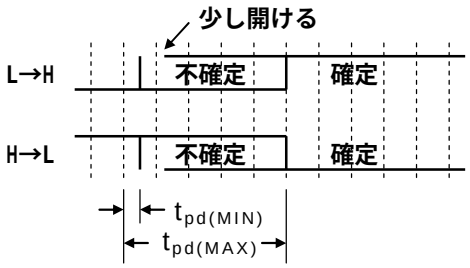
5.1　タイムチャートの表記法

この表記法は筆者の自己流で，従来より長く使っているものです．簡単ですが正確で便利な書き方ですので紹介します．

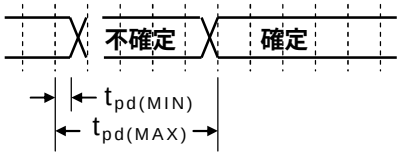
［図 5 -1］中島流表記法

①基本的に方眼紙に書く．
②回路要素の最小遅延時間$t_{pd(MIN)}$を，どんな要素でも一律方眼の最小目盛りの1/2とする．
　以上を基本に，次のように表記する．

○単信号の場合

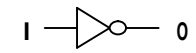


○バスの場合

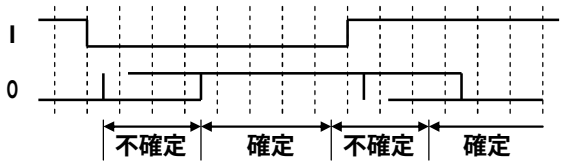


このように，左側が開いていて上下に線があるときは，その期間の信号は不確定であることを表す．

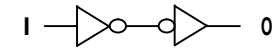
例1．インバータ



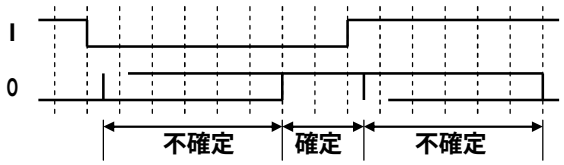
の場合．



例2．多段接続

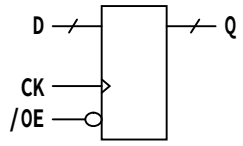


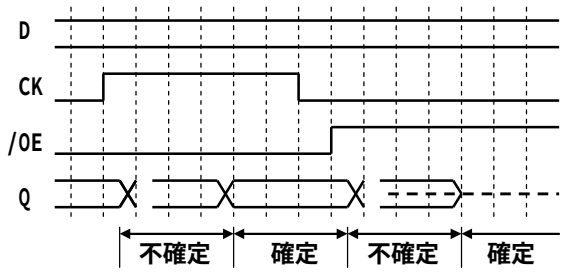
の場合．



この場合の不確定時間は，各$t_{pd(MAX)}$の合計．

例3．トライステート出力のレジスタ





このように，最小遅延時間を最小時間単位に設定して導入し，その表記法と併せてそれを明示しています．わずかな工夫ですが，これにより信号の遷移状態とともに信号がいつ確定し，いつまで有効なのかがひと目でわかるようになっています．また，最小遅延時間を 0 でなく 1/2 最小目盛りとすることによって，連動している複数の信号遷移の従属関係も表現できます．

5.2　タイムチャート

設計当時に作成したタイムチャートをそのまま示します．

（付録 2-1）

リセット状態から，リセット解除後反復 1 回分の全体の動作を検証するためのもの．

（付録 2-2～2-5）

ピクセルペアの処理終了時の動作を検証したもの．

（付録 2-6）

乗算回路のタイミングを検証したもの．

（付録 2-7）

Ｘx の加減算部のタイミングを検証したもの．

以上の表記法とタイムチャートにより，最大遅延時間に基づいたワーストケースの動作タイミングを確認しています．

6．使用部品

付録 3 に，Pyxis の部品表（半導体のみ）を示します．全 IC 数は 361 個，このうち 1 個はオシレータモジュールです．種類別の構成は，

・LS ‥‥‥‥‥	317 個
・ALS ‥‥‥‥‥	7 個
・F ‥‥‥‥‥‥	36 個
・オシレータ‥‥	1 個
合計	361 個．

F シリーズは，制御回路(Ｃn)内でタイミングの厳しい部分や重いファンアウトの部分に使用されています．

このように，すべて標準的な TTL シリーズの IC のみで構成されています．

7．実装設計

回路設計の結果，IC 数 361 個の巨大な回路となってしまいました．部品点数が多くなると，レイアウトなどにも十分配慮する必要があります．

7.1　基板

基板は 330㎜×245㎜ の，市販のユニバーサル両面基板を使用しました．デジタル回路用のパターンで，片面がグランドプレーンになっているものです．このグランドプレーンは IC の下やピン間にもパターンが通っており，グランドがまさにメッシュ上になっているすぐれものです．

この基板 4 枚に回路を分割して実装しました．分割の仕方は，Ｏx とＯy，Ｃn をあわせて 1 枚，他Ｘx，Ｙy，Ｒr に各 1 枚ずつです．

7.2　レイアウト

IC のレイアウトも難しくなってきます．特にＣn が問題で，ここからすべての回路へ制御信号が分配されます．そこで，レイアウト図面を作成して全配線を書き込み，妥当性を確認することを何度か行って決めました．決定基準は，

○同一タイミングの信号（特にクロック）の配線長をできるだけそろえる．

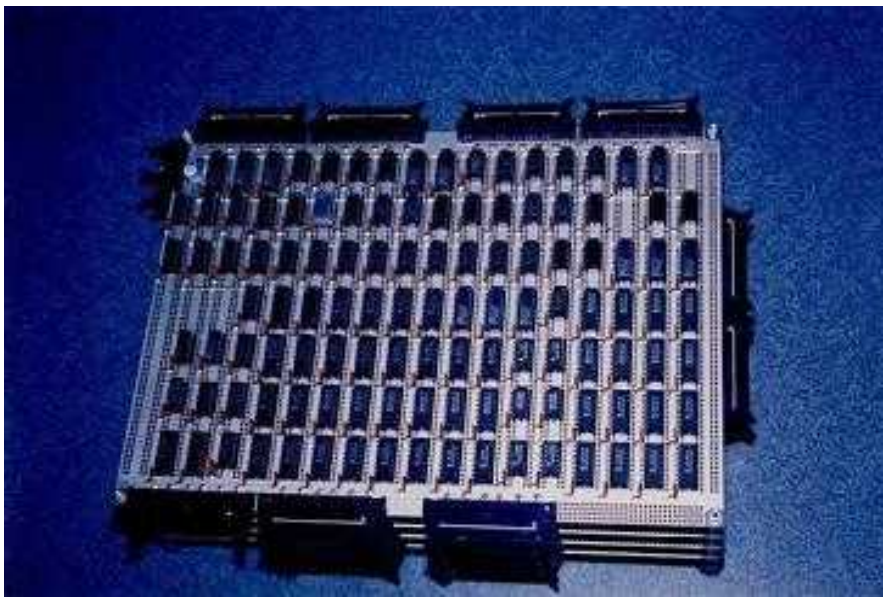
○長い経路の配線が生じないようにする．

○グランドのリターンも考慮する．

と，改めて書くまでもないような基本的なことです．ですが，シンクロもロジアナも無い環境でしたので，不具合が生じてからの "デバッグ" はまず不可能です．「後の祭り」を未然に防ぐためにも，非常に慎重に作

業しました．このことは，回路設計，配線など Pyxis の製作すべてに通じています．

これも当時の資料ですが，レイアウト図を付録 4 に示します．（なお，残念ながらレイアウト検証用の配線を書き込んだ図面は紛失した）

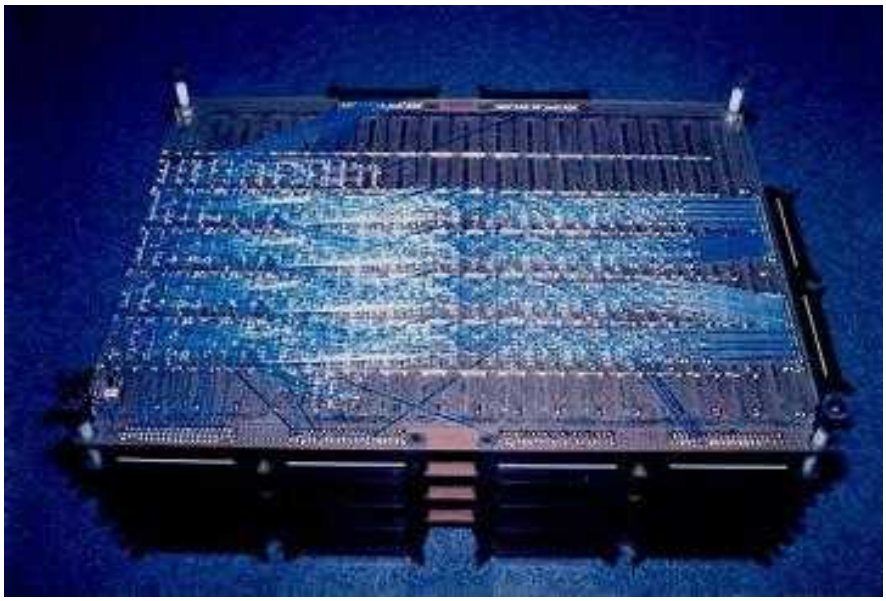


［写真 8-1］

Pyxis の全体像．

4 枚の各基板は，上部の 4 カ所と下部の 2 カ所の MIL コネクタを介して接続されている．右側の 2 カ所のコネクタは，2 枚目と 3 枚目の基板を接続している．

左上の小さなコネクタは電源供給用，左下はホストとの接続用．



［写真 8-2］

Pyxis の裏面．

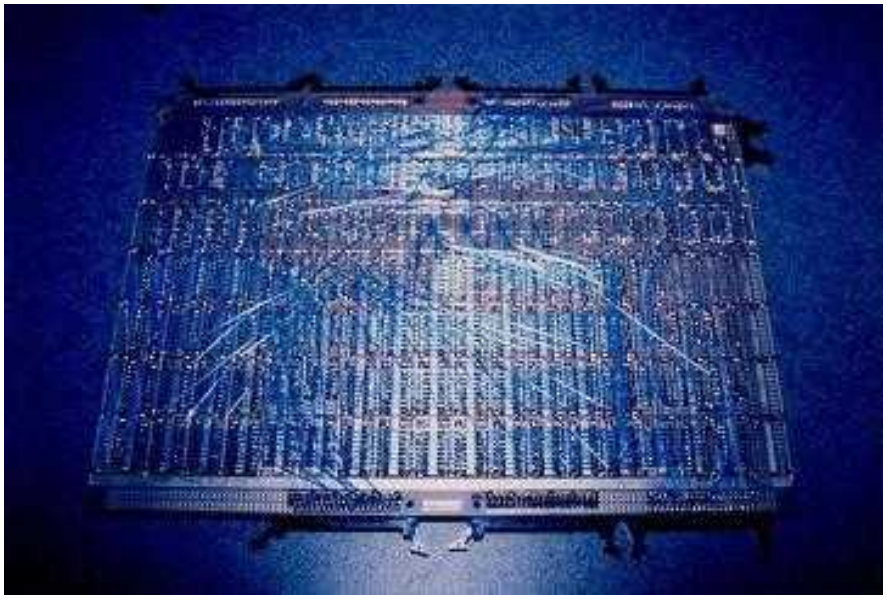
配線材であるブルーのラッピングワイヤーが見える．すべて手ハンダによる．

基板上で横に走っている銀のものは，グランド補強用のスズメッキ線．

8.製作

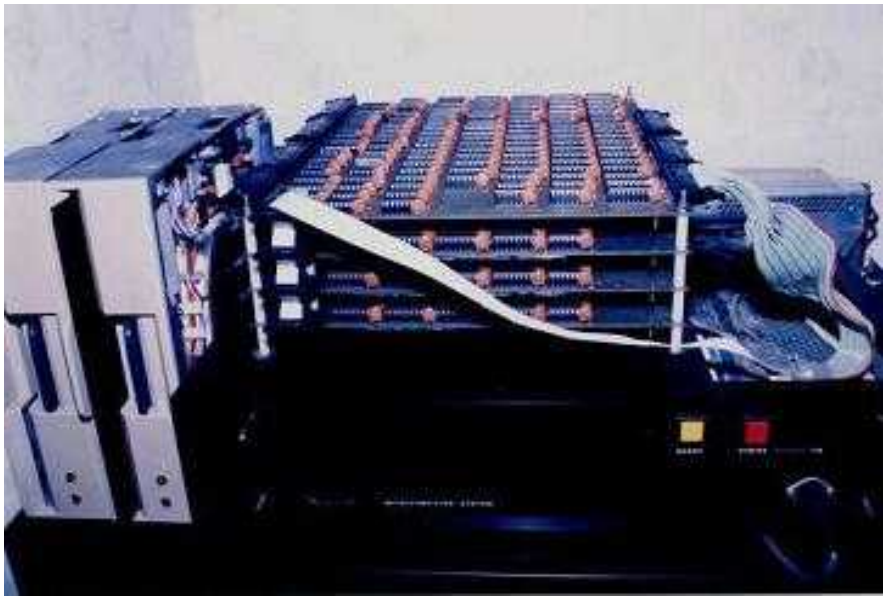
配線にはラッピングワイヤーを使用し，すべて手ハンダ，手配線で行いました．後になってネットリストから数えてみたところ，5000 本以上あったようです．配線だけで半年かかりました．趣味でやっているのですから文句も言えませんが．

写真 8-1 から 8-4 に Pyxis 本体を示します．



［写真 8-3］

Ox,Oy,Cn 基板の裏面．レイアウトは紙上シミュレーションして決めてあるので，繁雑になりやすい制御回路にもかかわらず配線はすっきりしている．



［写真 8-4］

システム全景．

下の黒いシャーシがホスト（自作 8 ビット機）．本レポートの描画画面の写真は，このコンピュータによるもの．現在も健在です．

9．ハンドリングソフトウエア

Pyxis のハンドリングのためにサポートソフトウエア（名称 MP）を作成しました．主な処理は 2.2 で説明したようなハンドリング処理と画面描画です．しかし折角ですので，M集合の探検に便利な機能（コマンド）を用意しました．

写真９ -1 は MP による描画画面です．

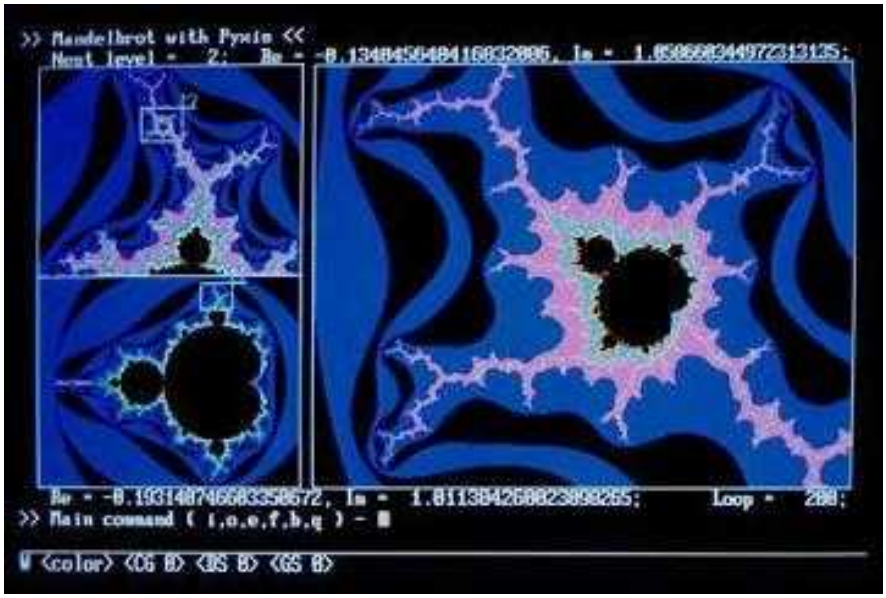
画面には 3 つの矩形描画領域があり，左下を全体領域，左上を親領域，右の大きな箱を子領域と呼びます．子領域は拡大描画の結果などが表示されるメインの領域で，これの左下と右上に座標(a_1,b_1)-(a_2,b_2)がそれぞれ表示されています．

親領域には，子領域に描画されている拡大を行う前の図形が表示されます．あわせて親領域内には，その中で子領域が対応する部分を示す枠と矢印が表示され，親の中のどこを拡大しているのかがわかるようになっています．

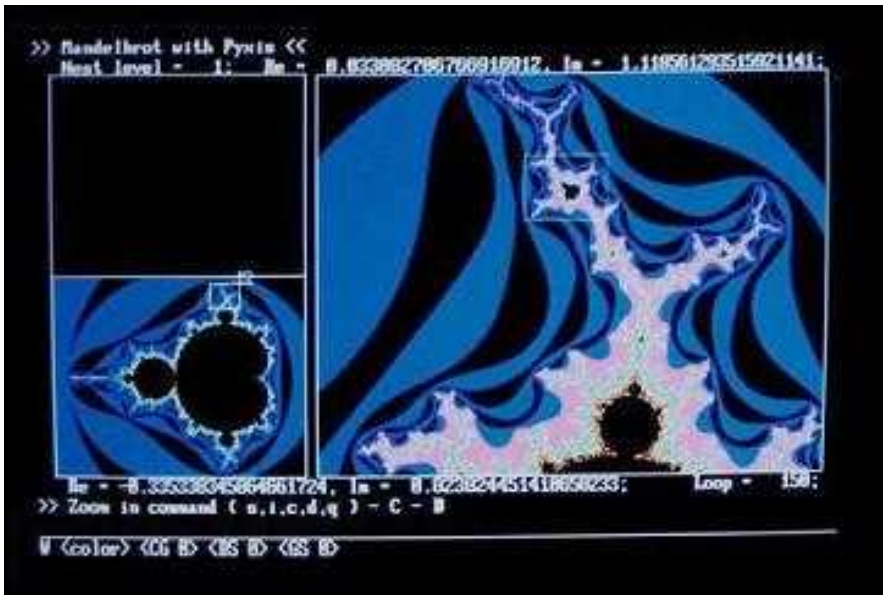
全体領域にはM集合の全体像が常に表示されており，同様に親領域が対応する部分を示す枠と矢印が表示されます．いま全体の中のどの辺を拡大しているのかがわかるようにしています．

拡大する部分の指定は，ズームインコマンドで行います．写真 9 -2 のように子領域内で枠を指定し，最大反復回数を入力します．すると子領域内の図形が親領域にコピーされ（写真 9 -3），指定した枠の部分が子領域に拡大描画されます（写真 9 -4）．この結果が写真 9 -1 です．

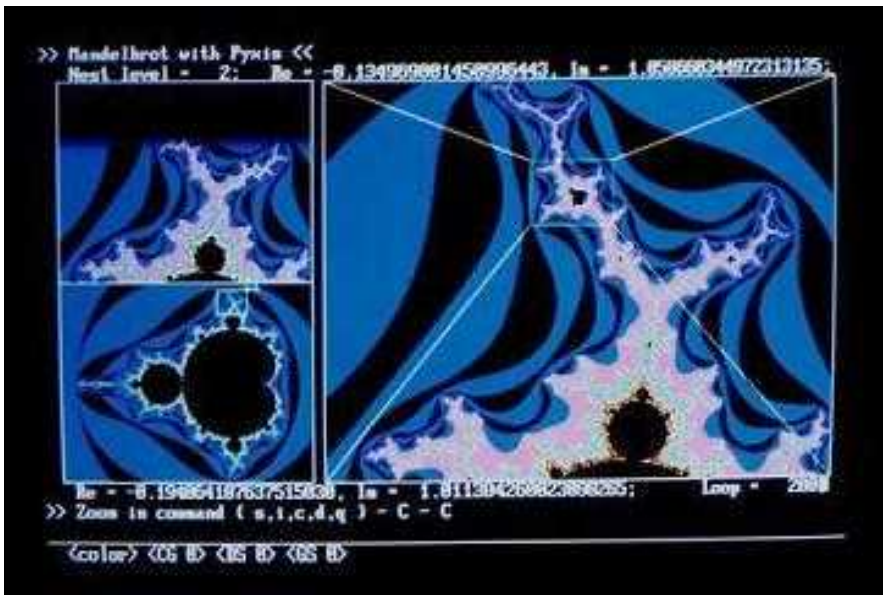
写真 9 -2 のように枠で指定した座標と最大反復回数のデータは，拡大指定情報としてメモリにツリー状に蓄えられます．次回からは同じ枠を指定しなくても，写真 9 -5 のように番号で指定できます．また，このツリー情報はファイルとしてセーブ／ロードできます．拡大の過程できれいな図形を見つけても，別の機会に改めて同じところへたどり着くことは非常に困難です．



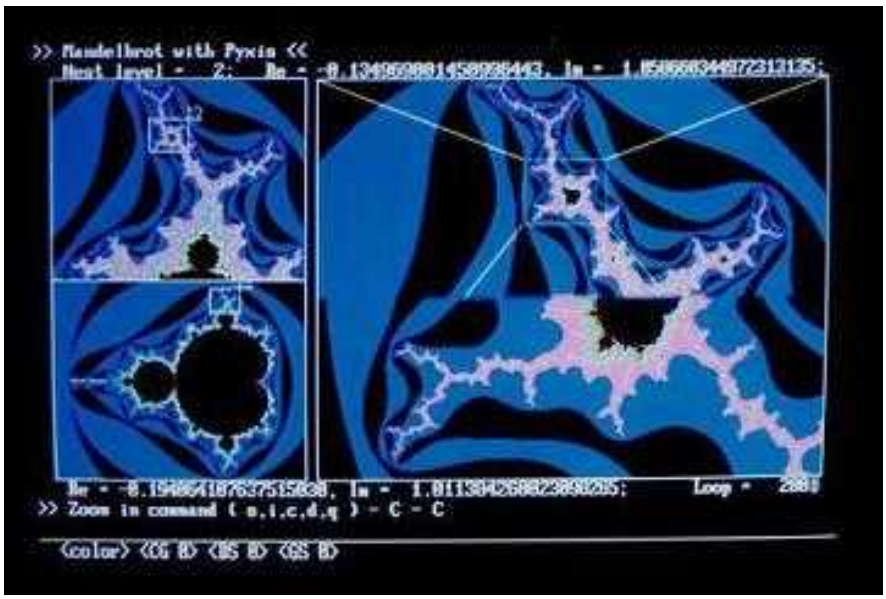
［写真 9 -1］



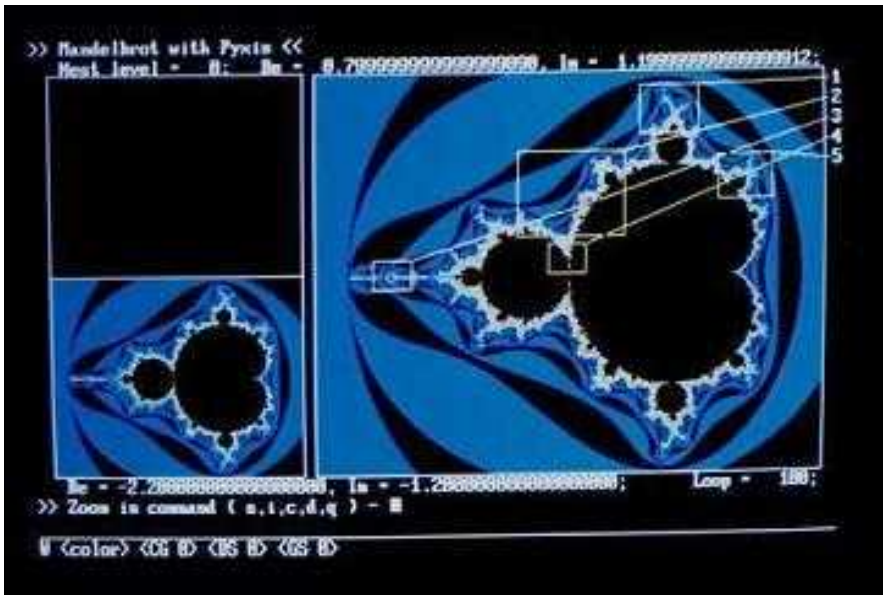
[写真 9 -2]



[写真 9 -3]



[写真 9 -4]



[写真 9-5]

[図 9-1] MP のコマンド体系

```
[MAIN]─┬─[Zoom In ]─┬─[Cursor]：枠指定による拡大描画
       │            ├─[Select]：番号選択による拡大描画
       │            ├─[Input ]：座標の数値入力による拡大描画
       │            ├─[Delete]：過去の拡大指定情報の削除
       │            └─[Quit  ]：Mainに戻る
       ├─[Zoom Out]──────────：ひとつ親の拡大描画に戻る
       ├─[Edit    ]─┬─[Move  ]：子領域の描画領域を移動し，再描画する
       │            ├─[Loop  ]：最大反復回数を変更し，再描画する
       │            └─[Quit  ]：Mainに戻る
       ├─[File    ]─┬─[Save  ]：拡大指定情報をファイルに保存する
       │            ├─[Load  ]：ファイルから拡大指定情報を読み込む
       │            └─[Quit  ]：Mainに戻る
       ├─[Batch   ]──────────：(＊)
       └─[Quit    ]──────────：MPを終了する
```

(＊)ツリー状の拡大指定情報を，深さ優先で自動的にたどって描画するコマンド．
ズームイン／アウトを自動的に実行し描画していくので，放っておくと環境ビデオのように楽しめる．

10．結果

10.1　実行時間

表 10-1 に，3 種類の環境における描画時間の測定結果を示します．この時間には，すべて画面への描画時間も含まれています．描画条件は，

$$\begin{cases} X_{MAX}=400,\ Y_{MAX}=320 \\ (a_1,b_1)=(0,0) \\ S=0.1/X_{MAX} \\ N_{MAX}=256 \end{cases}$$

これは，全ピクセルにおいて N_{MAX} 回反復する（全ピクセルがM集合に含まれる）場合になります．

この表に示されるように，期待通りの結果が得られました．②と比較しても十分に高速です．先に設計目標として掲げた "1 分" の根拠は，描画の際に CRT の前で待っていられる時間として考えた値です．描画を開始してから数分なら待てますが，85 時間では待っていられません．

因みに，M集合の全体像（写真 1-1）

$$\begin{cases} (a_1,b_1)=(-2.2,-1.2) \\ S=3.0/X_{MAX} \\ N_{MAX}=100 \end{cases}$$

の描画時間は 29 秒でした．このように，描画時間として十分に満足できる結果が得られました．

［表 10-1］結果：描画時間の比較（描画条件は本文中を参照）

環境	ハードウェア	ソフトウエア	描画時間	③との比較
①	自作 CP/M システム CPU:Z80A (4MHz)	Turbo Pascal Ver3	307200 秒 (85 時間 20 分)	3662 倍
②	PC98XL² (16MHz) CPU:80386 NDP:80387 付き	Turbo Pascal Ver4	12800 秒 (3 時間 33 分 20 秒)	153 倍
③	①＋Pyxis	Turbo Pascal Ver3	83.9 秒 (ストップウォッチによる測定 10 回の平均)	1 倍

ここで，回路の効率を検証してみます．

パイプライン 1 段あたりの所要時間は，乗算回路のところで説明したように 2.56μsec です．　従って，表 10-1③における理想的な演算所要時間は，

$$X_{MAX}\times Y_{MAX}\times N_{MAX}\times 2.56\times 10^{-6}\times 2^{注\ 1)}\times 0.5^{注\ 2)}$$

$$=\ 83.9[秒]$$

です．これは正確に③の結果と一致しています．③は描画を含んだ時間ですから，2.3 で説明した演算結果 $N_{x,y}$ のバッファリングが効果的に作用して，ホスト側による結果の読み出しまでの待ち時間が発生していないことがわかります．同時に，演算回路が設計通り効率 100%で動作していることが確認できました．

注 1)：パイプライン 2 段構成のため．

注 2)：2 ピクセル同時処理のため．

10.2　設計目標との対比

第2章のはじめに掲げた設計目標各項に対する結果をまとめてみます．

①描画時間について．

M集合全体を 29 秒で描画．

②IC 代について．

安い購入先を探すのに苦労しましたが，結局合計で \30,780．

③構成部品について．

オシレータモジュールを除き，すべて標準 TTL シリーズ(LS,ALS,F)で構成．

④演算精度について．

完全 64 ビット表現により，十進有効桁 18 桁以上．

⑤ホストへの依存度について．

10.1 にて説明の通り，8 ビット機でもハンドリングが十分可能な低依存度．

以上をまとめると，予算を若干オーバーしましたが総合的には十分目標を達成できたと考えます．

11．終わりに

Pyxis の製作にあたっては，慎重に，かつ時間をかけて，非常に複雑なパズルを解くような感覚で設計を楽しみました．思えば，本当に寝食を忘れてやっていたような気がします．回路は，すべてオリジナルです．

このレポートの中では，回路の詳細について残念ながらあまり説明できませんでした．自分で言うのも変ですが，全般に凝った設計なので，書きたいことが複雑かつたくさんあってとても書き切れません．説明が深みにはまらないように注意したつもりですが，いたらぬ点が多々あると思います．ご容赦ください．

今となっては Pyxis の性能も昔ほどではなくなってしまいましたが，当時のアマチュアレベルのコンピュータ環境においてはダントツに最速だったのではないかと自負しています．以前なら時間がかかりすぎて見当すらつけられなかったような彼方にある拡大図形を，心ゆくまで探し歩くことができるようになりました．

第 2 次レポートに添付しました描画例を参照され，この感覚を少しでも感じていただければ幸いです．

"Pyxis" の名前は，英語の "羅針盤座" という星座名からとりました．想像をはるかに超えた複雑さと広がりを持つM集合の CG において，文字通りの水先案内役を果たしてくれていることを製作者としてうれしく思っています．

［参考文献］

1)A.K.デュードニー,山崎秀記監訳,「コンピューターの顕微鏡で，数学でもっとも複雑な図形を拡大観察する」，『別冊サイエンス 82 コンピューター レクリエーション I』，日経サイエンス社

2)西橋敬一，「ジュリア集合とマンデルブロート集合」，『The BASIC』，1987 年 3 月号，技術評論社

3)H.-O.パイトゲン・P.H.リヒター，宇敷重広訳，『フラクタルの美 複素力学系のイメージ』，シュプリンガー･フェアラーク東京

4)K.Hwang,堀越監訳『コンピュータの高速演算方式』，近代科学社

5)『The Bipolar Digital Integrated Circuits Data Book PART1 -1982-』，日本テキサスインスツルメンツ

6)『1985 年版　最新 TTL IC 規格表』，ＣＱ出版社