CycleC と Handel-C によるマシンビジョンシステムの設計

平井　慎一　　立命館大学　ロボティクス学科
山本 真也　　グローリー工業株式会社　研究開発センター　技術研究所

１．　はじめに

Design Wave Magazine でいきなりこんなことを書くのもどうかとは思いますが，筆者らは LSI の専門家ではありません．平井はロボティクス，山本はコンピュータビジョンをバックグラウンドとしています．ほんの数年前までは，論理合成って何？配置配線って何？という状況でした．ではなぜ，ロボティクスやコンピュータビジョンの研究者が LSI の設計に関わったかというと，次のような次第です．

筆者らは，ビジョンを用いて物体の位置（並進移動）と姿勢（回転移動）を検出し，ロボットの運動を制御するという，ビジュアルフィードバックの研究開発を進めていました．正規化相関法あるいはブロックマッチングとよばれるアルゴリズムでは，物体の並進移動を検出することができます．このアルゴリズムは MPEG でも使われており，専用の LSI があります．しかし，正規化相関法では，回転移動を検出することができません．筆者らは，並進移動と回転移動を検出できるアルゴリズムを開発していましたので，当然それを LSI 化することを考えました．しかし，ある LSI メーカの技術の方に打診したところ，「月産万単位以上のマーケットがないと ASIC にする意味はないですよ」と言われ，ASIC は断念しました．しょうがない，自分らで何とかするかと覚悟し，手がけたのが FPGA＋C 言語レベル設計です．

このような経緯で感じたのは，ロボティクスやコンピュータビジョンの人たちにとって，LSI 設計は，興味はあるけど最初の敷居が意外に高い技術ではないかということです(図１)．ロボティクスやコンピュータビジョンの世界では，興味を示す人は多いのですが，LSI 設計を経験した人は意外に少ないし，LSI の世界でコンピュータビジョンはともかくロボティクスやメカトロニクスに関わった人は少ないと思います．筆者らは，C 言語レベル設計を体験し，C 言語レベル設計がこのような敷居を低くすると信じるようになりました．この記事では，LSI の専門家ではない筆者らの体験をもとに，C 言語レベル設計を使ってどのようにビジョン回路を設計したかを書きます．LSI の専門家ではないけれど，LSI の回路設計を試してみたい，LSI の回路設計をしなくてはならなくなったという方々の参考になれば幸いです．

２．　われわれが実現したかったこと

ロボティクスやメカトロニクスでは，様々な信号処理を行います．近年，広く用いられているのが，ビジョンです．「そんなのコンピュータビジョンで数十年の研究開発があるやないか」という方，忘れがちなのがリアルタイム性とロバスト性です．機械システムを動かすときには，センシングから運動までの時間に制限があります．できればセンシングから運動まで 1msec で動いてほしいし，そこまで達しなくても速ければ速いほど性能が上がります．通常の CCD カメラでは 30fps ですから，我慢して約 33msec ということになります．一方，機械システムが実際に動くのは，きれいな環境ではありません．照明の明るさは変動しますし，影があったりオクルージョンが生じていたりと，画像の質が落ちていま

す．このような外乱に対応できる能力を，ロバスト性といいます．ロボティクスやメカトロニクスにおける信号処理で難しいのは，リアルタイム性とロバスト性の両方が求められるからです．

われわれが実現したかったのは，物体の位置と姿勢の検出です．図２を見てください．あらかじめ検出したい物体を撮影し，テンプレート画像とします．入力画像のなかから，テンプレート画像に写っている物体を探し，その位置と姿勢を求めることが目的です．入力画像内の四角の枠は，検出した位置と姿勢を表しています．図にあるように，入力画像には，オクルージョンや物体の変形，照明変動，ハレーションが生じることがあります．外乱があっても，きちんと位置と姿勢が検出できています．これが，ロバスト性です．

位置と姿勢を検出するロバストなビジョンアルゴリズムは，いくつか知られています．例をあげると，マッチトフィルタ法，位相限定相関法，ハフフーリエ変換法，一般化ハフ変換法などです．われわれは，ハフフーリエ変換法とマッチトフィルタ法を実装しています．図３にハフフーリエ変換法の概略を，図４にマッチトフィルタ法の概略を示します．ハフフーリエ変換法では，テンプレート画像と入力画像のハフ変換（後で説明します）と，ハフ変換のフーリエパワースペクトルを比較して，位置と姿勢を検出します．マッチトフィルタ法では，テンプレート画像と入力画像の二次元パワースペクトルと，その回転変換（後で説明します）を比較して，位置と姿勢を検出します．これらの方法はどれも二次元 FFT や逆 FFT，ハフ変換，極座標変換，回転変換といった計算量の多い演算が欠かせません．ただし，これらの演算は並列性が高く，LSI 上に回路を実現すると，計算時間を短くすることができます．

冒頭でも述べたように，筆者らは LSI の専門家ではありませんが，ビジョンアルゴリズムを LSI 上に実装したいという欲求を持っていました．どのように実現するかを考えたとき，解決策が FPGA＋C 言語レベル設計です．なぜ FPGA かというと，ビジョンアルゴリズムは個別性が強く，通信やコンピュータ用の LSI ほどのマーケットが望めないからです．なぜ C 言語レベル設計かというと，様々なビジョンアルゴリズムを C/C++言語で開発してきたからです．結論から言いますと，この方式は大成功であったと思います．

３．　Ｃ言語による画像処理回路の設計

筆者らは，ビジョンアルゴリズムを FPGA 上に実装するに当たって，可能な限り PC 上でＣ言語を用いて回路の開発と検証を行うという方針を立てました．第一の理由は，それまで PC 上でビジョンアルゴリズムの開発を行ってきたからです．第二の理由は，回路記述の検証です．ビジョンアルゴリズムの検証は，信号レベルではほとんど不可能で，画像を見ながら行ってきました．ハフ変換やフーリエ変換のプログラムも，変換の結果を画像化して，画面で確認することで検証しています．したがって，ビジョンアルゴリズムの回路設計においても，画像を見ながら回路を検証することが効率の面から必須であると考えました．そうすると，必然的に C 言語による回路設計を導入することになります．

図４に示す二つのハードウェアを構築して，ビジョンアルゴリズムを実装しています．図４(a)は，スタンドアロンタイプです．これは，エスケーエレクトロニクス社製 CM-69 の周囲に，三菱電機マイコン機器ソフトウェア社 Power Medusa シリーズのビデオデコーダボード，ビデオエンコーダボード，SRAM ボードを配置してあります．図４(b)は，PC ボ

ードタイプです．これは，ALPHA DATA parallel systems 社製 ADM-XRC と Matrox 社製画像入力ボード Meteor II/MC を，PC の PCI バスに接続しています．どちらも，FPGA として，ザイリンクス社 VertexE-2000 を使っています．これら二つのハードウェア上で，ビジョン回路を設計するために，CycleC (C Level Design, Inc.)と Handel-C (Celoxica Ltd.)を選択しました．スタンドアロンタイプでは CycleC，PC ボードタイプでは Handel-C を使っています．スタンドアロンタイプでは，Visual C/C++環境でシミュレーションができる CycleC を，PC ボードタイプでは，ハードウェアに特有な並列処理が記述できる Handel-C を選びました．

CycleC は，ANSI C/C++の文法のサブセットです．CycleC の記述スタイルに従って，論理回路を記述します．論理回路の挙動は C++のクラスとメソッドで記述します．実際の論理回路は，インスタンスに対応します．CycleC による記述は，通常のコンパイルを行うことにより，PC 上のプログラムとして実行することができます．一方，CycleC による記述は, トランスレータにより HDL へ変換することができます. すなわち, 通常のコンパイル・実行によるアルゴリズムの検証と，HDL への変換が両立しており，アルゴリズムの開発とハードウェアの記述を効率よく進めることができるというのが, うたい文句です. 表 1 は，CycleC を使ってセレクタを記述した例です．変数の宣言では，符号とビット幅を指定します．宣言 uint1 は，符号なし 1 ビット変数を表します．マクロ infer_clock(clk)は，信号 clk をクロックとみなし，立ち上がりで真となります．マクロ infer_reset(!rst)は，信号 rst をリセットとみなし，立ち下がりで真となります．したがって，infer_clock(clk) || infer_reset(!rst)は，posedge CLK or negedge RST と同じ記述です．クラス Selecter の定義には，セレクタが 1 ビットのレジスタ変数を持ち，その初期値が 0 であることが記述されています．セレクタの挙動は，メソッド Selecter::run に記述されています．出力信号 **y** への代入は，レジスタ変数を通して行います.

CycleC では，ループの回数が定数である必要があります．表２(a)に示す for 文は，8 回のループですので，CycleC の記述です．表２(b)に示す for 文は，ループの回数が不定ですので，CycleC の記述ではありません．これは，表２(c)のように記述する必要があります.

Handel-C は，ANSI C/C++に Occum スタイルの並列記述や回路設計特有の記述を追加した言語です．ご記憶の方もあるでしょうが，並列計算を実現するために，トランスピュータという LSI と Occum という言語がありました．Handel-C には，ANSI C/C++に並列計算を記述する Occum の構文，すなわち，並行プロセスを記述する par，並行プロセス間のチャネルを記述する!や?等が追加されています. ハードウェアの並列性を明確に記述することができるというのが，うたい文句です．Hendel-C では，代入処理に対応してレジスタが生成されます．クロックの立ち上がりに同期して代入が行われるため，値が有効となるのは次のクロックです．このように，Handel-C による記述では，サイクルアキュレートに処理を実行しますので, 動作タイミングが正確に予想できます. また, Handel-C の記述は，HDL 記述へ変換することも，直接ネットリストを生成することもできます.

Handel-C では, par を用いて並列処理を記述します. 構文 par の内部に記述した処理は，並列に実行されます．すべての並列処理が終わった時点で par 構文以降の処理が開始されます．たとえば，変数 a と b の値を交換するとき，ANSI C では表３(a)のように書きますが，Handel-C では表３(b)のように記述できます．また，par 構文を用いて，パイプライン

を記述できます．表４は，Handel-C によるパイプライン記述の例です．表のように，par 構文内部に順次処理を記述することにより，パイプラインを記述できます．

３.１　CycleC による設計例

CycleC を用いてハフ変換を記述した例を紹介します．ハフ変換について簡単に説明します．グレースケール画像の座標を(x,y)とします．格子点(x,y)における画素値を g(x,y)で表します．格子点(x,y)に対して，次式を満たす(ρ,θ)の組を求めます．

$$\cos\theta\ y - \sin\theta\ x = \rho$$

上式を満たす(ρ,θ)の組に対して，H(ρ,θ)の値を画素値 g(x,y)だけ増加させます．以上の操作を投票とよびます．すべての画素に対して投票を行い，その結果得られる H(ρ,θ)を，ハフ変換とよびます．図６がハフ変換を行う回路です．入力信号 X,Y が座標値を，入力信号 G が画素値を表します．実装例では，角度 0°～360°を，2.5°毎に 144 分割しています．角度 2.5k (k=0,1,...,143)に対する投票を回路 THETA(k)で行い，その結果得られる H(ρ,2.5k)をバッファ HOUGH_BUF(k)に格納します．投票回路には，SIN_k=sin(2.5k)と COS_k=cos(2.5k)の値を記憶しておきます．各々の角度に対する投票は，並列に実行することができるので，投票回路 THETA(k)とバッファ HOUGH_BUF(k)を 144 個並べて並列に動作させ，ハフ変換を計算します．

表５は，CycleC によるハフ変換回路のクラス定義です．数値は，符号ビットと絶対値で表しています. 正のとき符号ビットは 0，負のとき符号ビット 1 です. メソッド THETA::run では，座標 X,Y と定数 SIN,COS からρの値を計算します．信号 SIN,COS には，角度 2.5k に対するサインとコサインの値を定数で指定します．実際の計算は，組合せ回路 CALC_RHO で行います. 組合せ回路 CALC_RHO の記述を, 表６に示します. MULT は, 8 ビット整数と 9 ビット整数の積を計算し，16 ビット変数に代入するマクロです．マクロ get_bit と set_bit はビット操作を，マクロ get_ slice と set_ slice はビットスライス操作を行います.

表７は，ハフ変換回路のシミュレーション記述です．入力信号を生成して，メソッド THETA::run を実行しています．表８が Visual C によるプログラムです．Visual C プログラムによる変換結果と CycleC 回路記述のシミュレーションによる変換結果が一致しているので，ハフ変換の回路記述が正しいことを確認できます（図７）．その後，CycleC の記述を VerilogHDL に変換し，HDL レベルで投票回路とバッファを接続しました（表９）．図８は実行結果です．画面中央 256×256 ピクセルの領域をリアルタイムでハフ変換し，結果を画面下部に出力しています.

３.２　Handel-C による設計例

Hanel-C を用いて回転変換を記述した例を紹介します．回転変換について簡単に説明します．図９(a)に示す原画像を，画像中心を回りに角度θ回転させると，図９(b)に示す回転画像が得られます．この変換を回転変換とよびます．原画像の中心から水平方向を x 軸，垂直方向を y 軸とします．回転画像の中心から水平方向を adjU 軸，垂直方向を adjV 軸とします．座標(adjU, adjV)で表される回転画像上の点が，原画像で座標(x,y)に対応するとします．このとき，座標(adjU, adjV)と座標(x,y)の間には，

x = adjU * cosθ – adjV * sinθ

y = adjU * sinθ + adjV * cosθ

が成り立ちます．したがって回転変換は，(1) 座標(adjU, adjV)から座標(x,y)を計算する，(2) 原画像の座標(x, y)における画素値を読み出す，(3) 回転画像の座標(adjU, adjV)に画素値を書き込む，という手続きを繰り返すことにより，計算することができます．すなわち，変換後の座標系でラスタースキャンしながら，対応する原画像の輝度値を書き込みます．

表１０は，Handel-C による回転変換回路の記述です．水平方向走査の部分が，パイプライン処理になっています．パイプライン実行の様子を，図１０に示します．変数 u，v から座標 adjU, adjV を計算し，cosθ, sinθとの積を求め，加算，丸め処理を経て，座標 x, y を求めます．次に，座標 x, y が画像内にあるかどうかを判定し，原画像から読み出すアドレスを求め，読み出しを開始します．読み出しには 3 クロック要するので，validFlag の値を順次レジスタ変数に代入して，3 クロック待ちます．その間に，回転画像に書き出すアドレスを計算し，書き出しを開始します．パイプラインの最後で，読み出しと書き出しを続けて実行します．表中，宣言 signed 9 は，符号付き 9 ビット変数，unsigned 9 は符号なし 9 ビット変数を表します．マクロ adju は符号なし変数のビット幅を，マクロ adjs は符号付き変数のビット幅を変換します．記述 a[n]は変数 a の第 n ビットの抽出，a[n:m]は変数 a の n～m ビットの抽出，a<-n は変数 a の下位 n ビットの抽出，a¥¥n は変数 a の下位 n ビットの除去を意味します．また，a@b は，変数 a と変数 b の連結を意味します．SSRAM にアクセスするために，自作のマクロ関数 ZbtSetAddress と ZbtAccess を用いています．関数 ZbtSetAddress は，アドレスとコントロールを設定します．関数 ZbtAccess は指定されたデータにアクセスし，次のアドレスとコントロールを設定します．たとえば，データ 0,1,2 をアドレス 0,1,2 にパイプライン書き込みを行う場合，

ZbtSetAddress(バンク ID，アドレス 0, ZBT_WRITE);

ZbtAccess(バンク ID，データ 0, WRITE_FLAG，アドレス 1, ZBT_WRITE);

ZbtAccess(バンク ID，データ 1, WRITE_FLAG，アドレス 2, ZBT_WRITE);

ZbtAccess(バンク ID，データ 2, WRITE_FLAG，ダミーアドレス, ZBT_READ);

と記述します. Handel-C では, HDL コードや IP コアとの接続を記述できます. 表１１に，三角関数テーブルコアとの接続例を示します．宣言 interface を用いて，外部ポートの定義を行います．出力変数に値を代入すると，その値が IP コアに渡されます．IP コアからの出力は，SinCos9.COSINE あるいは SinCos9.SIN にように，構造体の形で参照できます．

マッチトフィルタ法を Handel-C で記述し，PC ボードタイプで実行した結果が，図１です．ビデオフレームレートで，物体の位置と姿勢を検出しています．

３.３　評価と個人的な感想

CycleC，Handel-C ともに，ビジョン回路を設計する上で有効でした．まず，CycleC は，Visual C/C++でシミュレーションができる HDL という印象を持ちました．CycleC の最大のメリットは，シミュレーションを簡単に実行できることです．Visual C/C++上で実行できるので，回路記述の途中に printf 文を挿入して変数の値を見るとか，二次元配列をファイルに落としてビューワで画像として見るということが簡単です．すなわち，画像の出力で確認できるので検証がわかりやすい，コンパイル・実行は数秒で終わるので，シミュレ

ーションを数多く試みることができる，という利点があります．これは，ビジョン回路の設計においては，強い道具になりました．一方，Handel-C は，本格的に回路を記述するための C 言語と感じました．Handel-C の最大のメリットは，C 言語に近い形で回路のほとんどの部分を記述できることです．Handel-C を用いると，C 言語に近い感覚で並列処理やパイプライン処理が記述できます．また，C 言語レベルで HDL コードや IP コアとの結合を記述できます．このように Handel-C は，CycleC よりソフトウェアに近い記述ができますので，ソフト技術者向きだと思います．

もちろん，すべてを CycleC や Handel-C で記述できるわけではありませんし，HDL で記述する方が簡単な場合も多くあります．スタンドアロンタイプでは，ビデオデコーダやエンコーダとの入出力は，すべて VerilogHDL で記述しました．当初は，画像の入出力を含めて CycleC で記述しようとし，ビデオデコーダやエンコーダのシミュレーション記述を CycleC で行おうとしましたが，止めました．理由は簡単で，ビデオデコーダやデコーダのインターフェースに関しては，HDL のサンプル記述があったからです．すべて CycleC で記述するという手間をかけるより，CycleC の記述を HDL に変換し，HDL 記述と接続するほうがはるかに簡単であることに気がついたという次第です．また，PC ボードタイプでは，Virtex-E の DLL(Delay Locked Loop)を使用しましたが，Handel-C では記述ができませんでした．そのため，トップモジュールを VHDL で記述し，その内で DLL を記述しています．また，今回の Handel-C 設計では，DK1(Handel-C 開発環境)のデバッガは用いませんでした．用いた FPGA ボードの外部 RAM は，PCI バス経由で PC からアクセスできるので，テストデータの書き込み，結果データの読み込みが PC 上で可能です．そこで，実際に FPGA ボード上で動作を確認することにより，デバッグを行いました．

CycleC や Handel-C で記述するにしても，HDL 記述や LSI のハードウェアに関するスキルはある程度必要です．アセンブラやマシンコードを経験することが，C 言語によるソフトウェアを直感的に理解するために必要であるのと同様，HDL 記述を経験することは，C 言語による回路記述を進めるうえで必須と感じます．著者らも，VerilogHDL や VHDL の講習会に参加しましたし，設計では HDL による記述も併用しています．逆に，CycleC や Handel-C を使うことにより，HDL 記述や LSI のハードウェアに関する知識を，必要最小限は習得できたと思います．

４．　おわりに

はじめにでも言いましたが，筆者らのバックグラウンドは LSI ではありません．そのような研究者が，一年間のプロジェクトで，ハードウェアの構築からビジョン回路の設計と実装までを達成することができました．これは，C 言語レベル回路設計の賜物です．C 言語レベル設計なしでは技術的なバリアが高すぎて，ビジョンアルゴリズムの実装は不可能だったと思います．

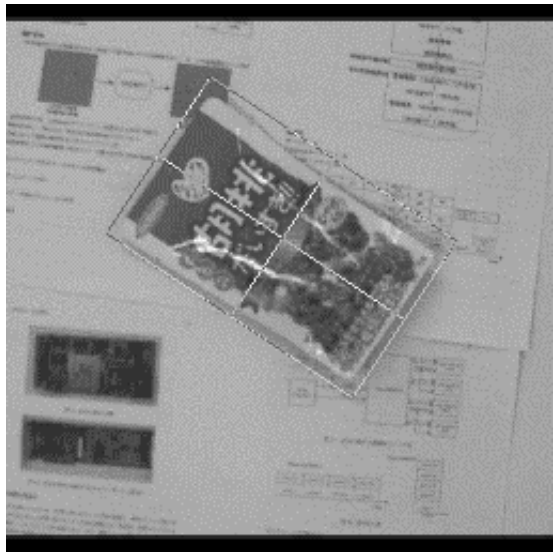
ロボティクスやメカトロニクス，コンピュータビジョンに関して，FPGA を使いたい，LSI を設計したいという方には，C 言語レベル設計を導入することをお勧めします．また，これらの分野には，LSI 化すると技術的にもマーケット的にも面白いネタがたくさんあります．「何を作るか」には事欠きませんし，MEMS との融合も期待できます．LSI の専門家の方も，こちらの分野にチャレンジしてはどうでしょうか．

9600 字＋図表 12 点
40 字×40 行＝1600 文字／ページ　9600 文字＝6 ページ

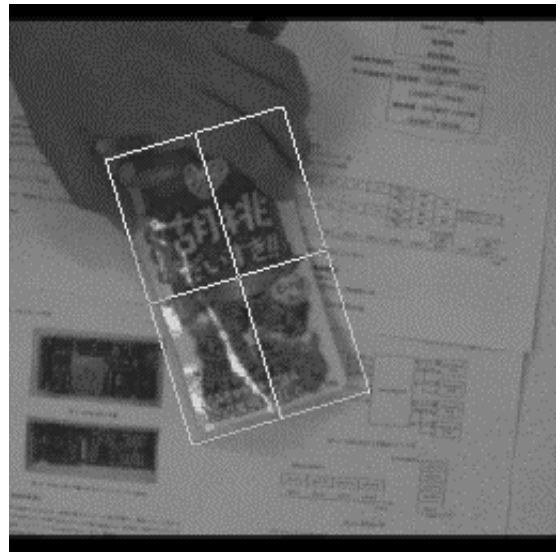
2002 年 11 月 20 日

図1　意外に大きい技術的な壁
ロボティクスやコンピュータビジョンの人たちにとって，LSI設計は興味はあるけど最初の敷居が意外に高い．京都のお茶屋さんで，興味はあるけど一見さんお断りと聞いて，敷居が高いと感じるようなところか．

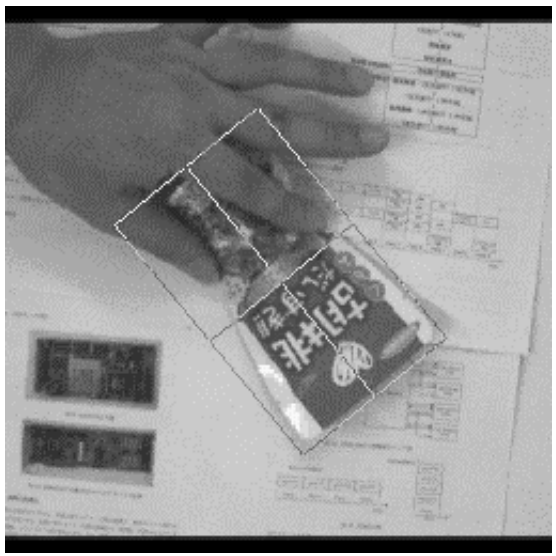
(a) テンプレート画像



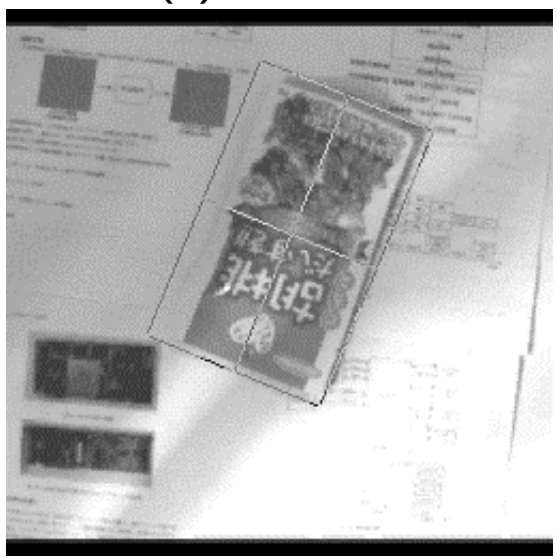
(b) 入力画像



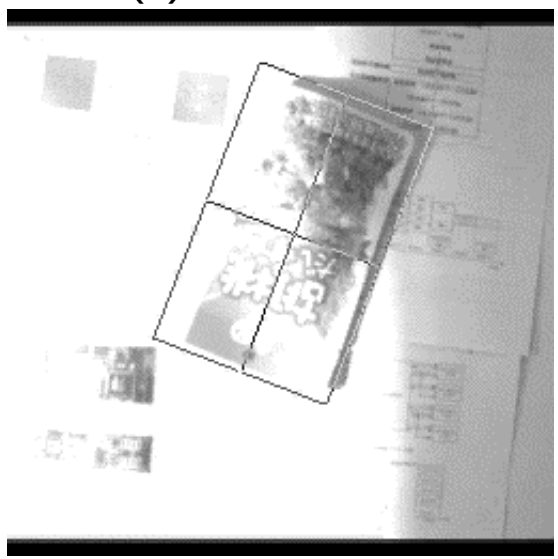
(c) オクルージョン



(d) 変形

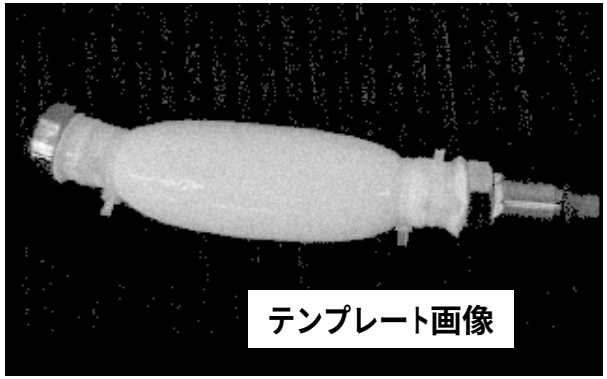


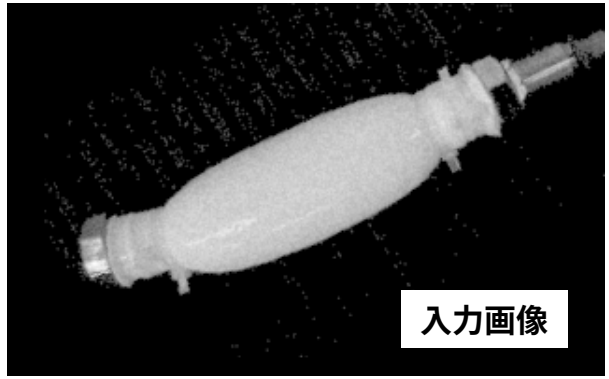
(e) 照明変動

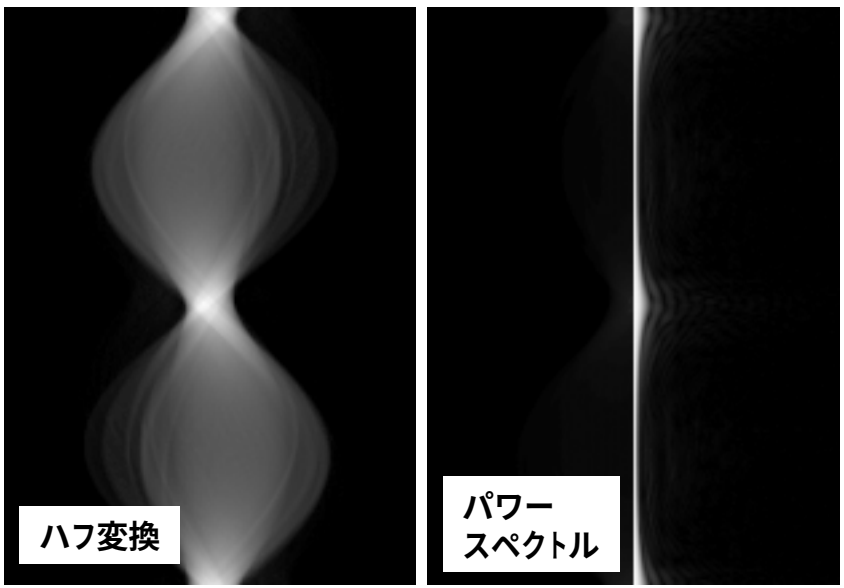


(f) ハレーション

図2　テンプレート画像と入力画像
テンプレート画像に写っている物体を，入力画像の中から探し，







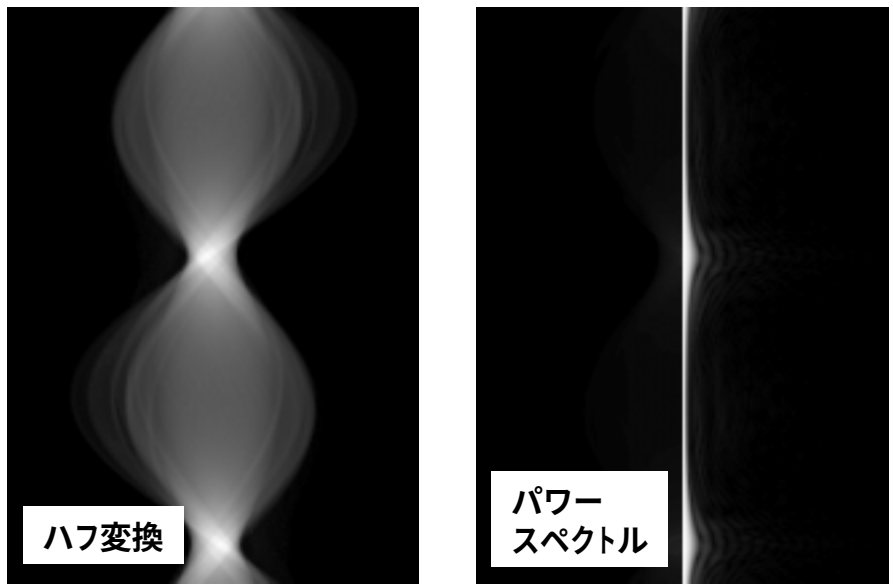


図3　ハフフーリエ変換法
テンプレート画像と入力画像のハフ変換のパワースペクトルを比較して，物体の姿勢を検出する．姿勢を検出後，ハフ変換を比較して，物体の位置を検出する．

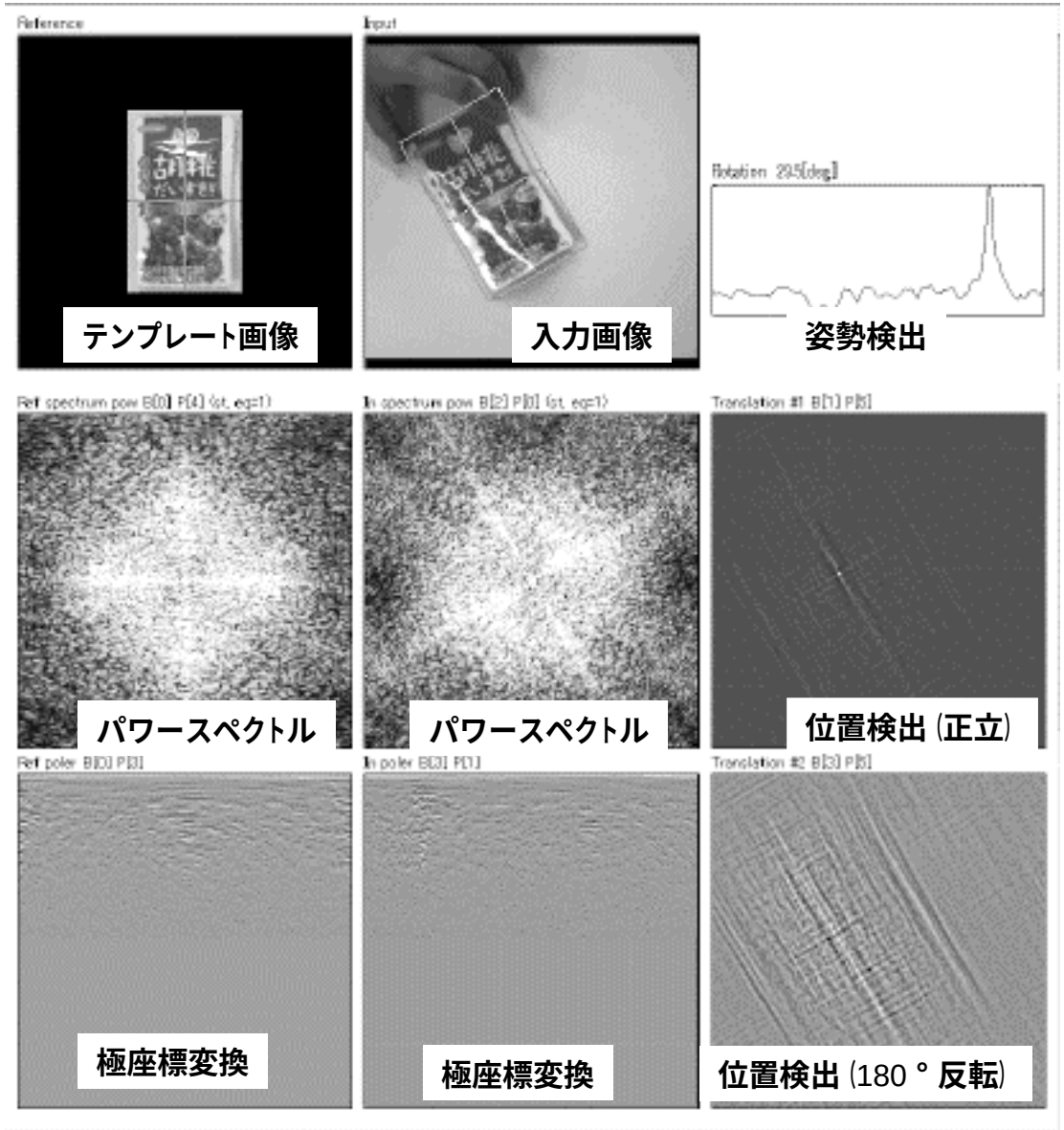
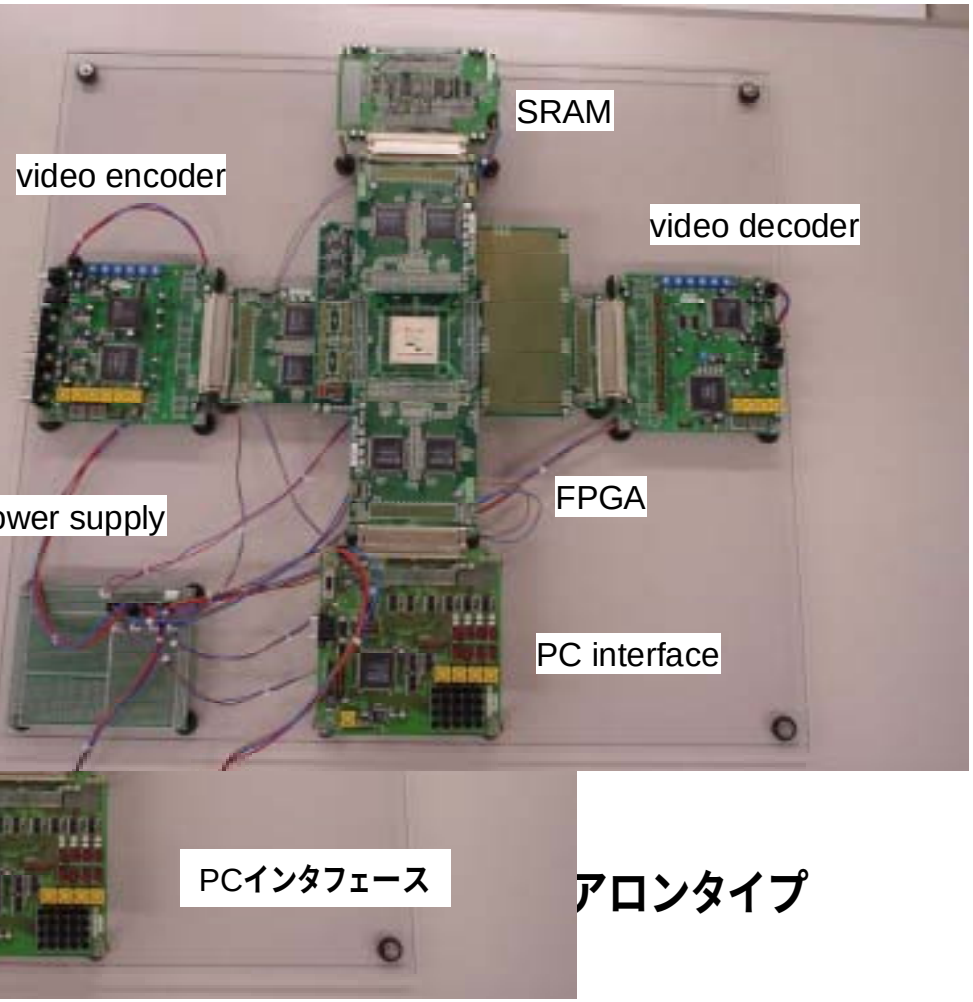
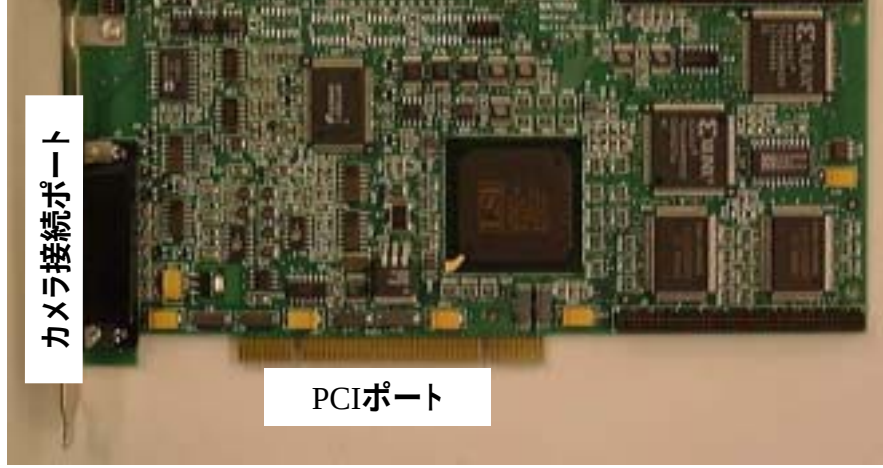
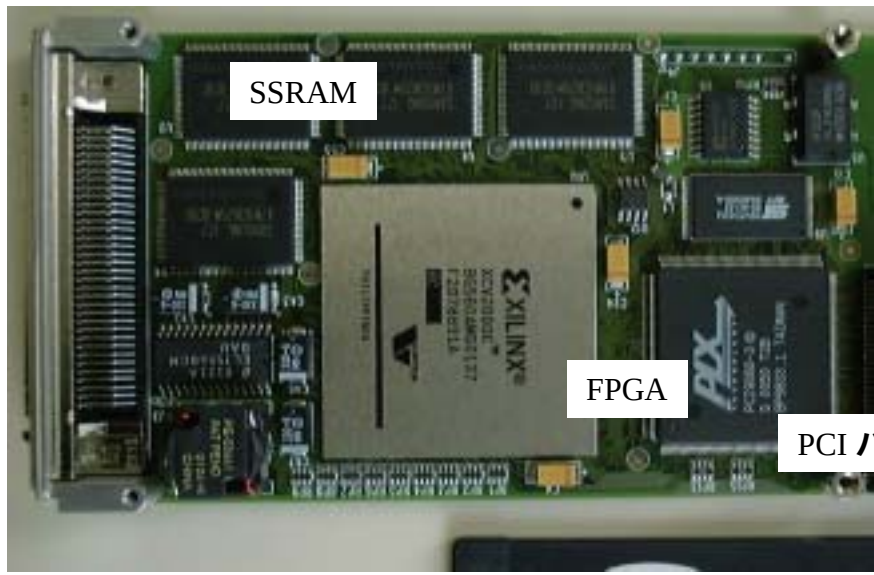


図4　マッチトフィルター法
テンプレート画像と入力画像の二次元パワースペクトルの極座標変換を比較して，物体の姿勢を検出する．姿勢を検出後，テンプレート画像と入力画像の二次元フーリエ変換を比較して，



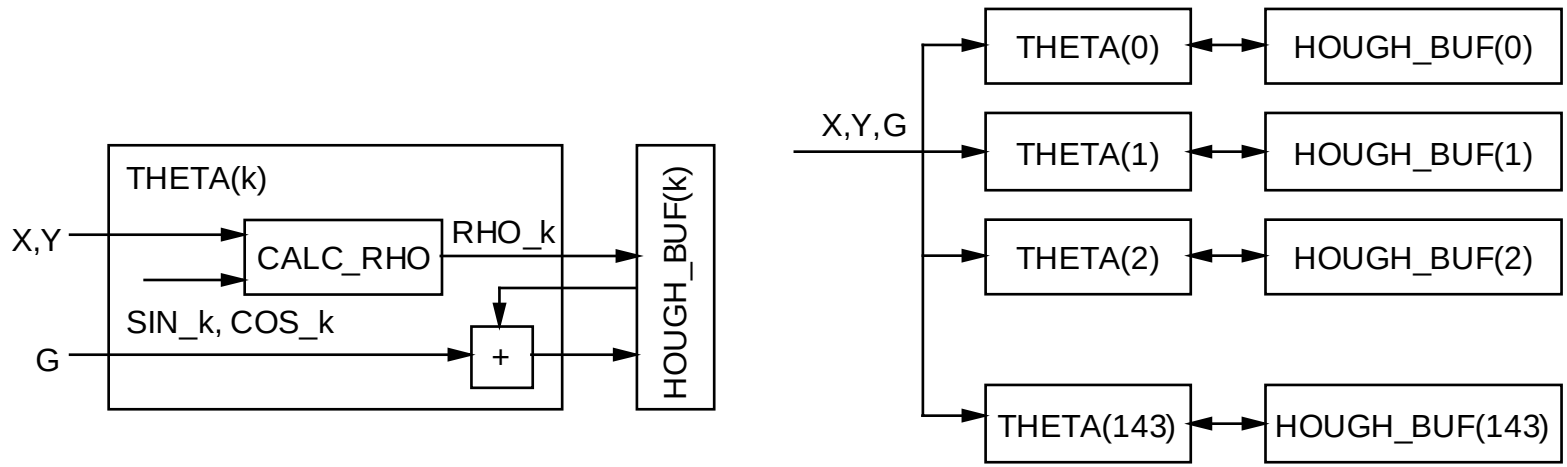
アロンタイプ





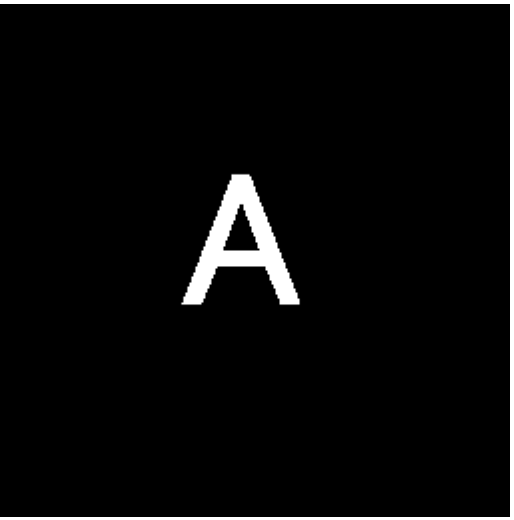
(b) PCボードタイプ

図5 ビジョンアルゴリズムを実装するハードウェア
スタンドアロンタイプは，エスケーエレクトロニクス社製CM-69**の周囲に**
三菱電機マイコン機器ソフトウェア社Power Medusa**シリーズのボードを配置**
してある．PC**ボードタイプは，**ALPHA DATA parallel systems**社製**
ADM-XRC**と**Matrox**社製画像入力ボード**Meteor II/MC**を使用している．**

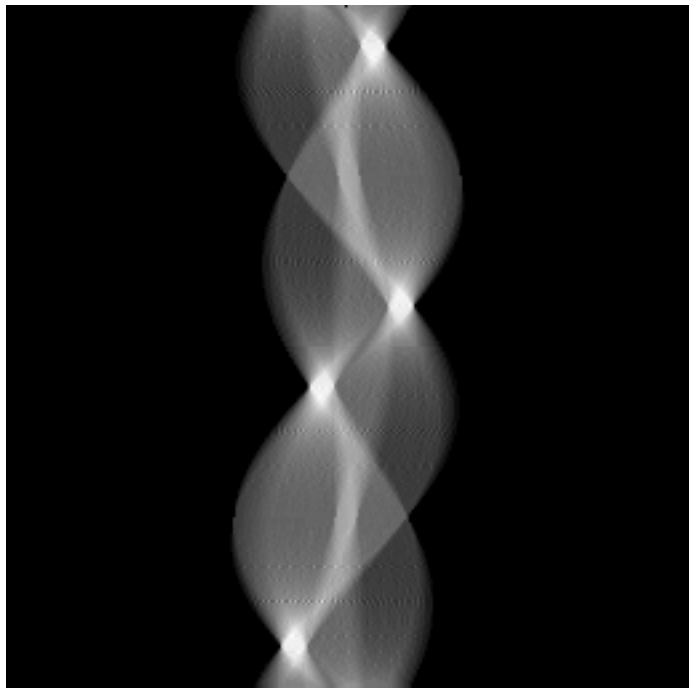


(a) **投票回路**　　(b) **並列ハフ変換**

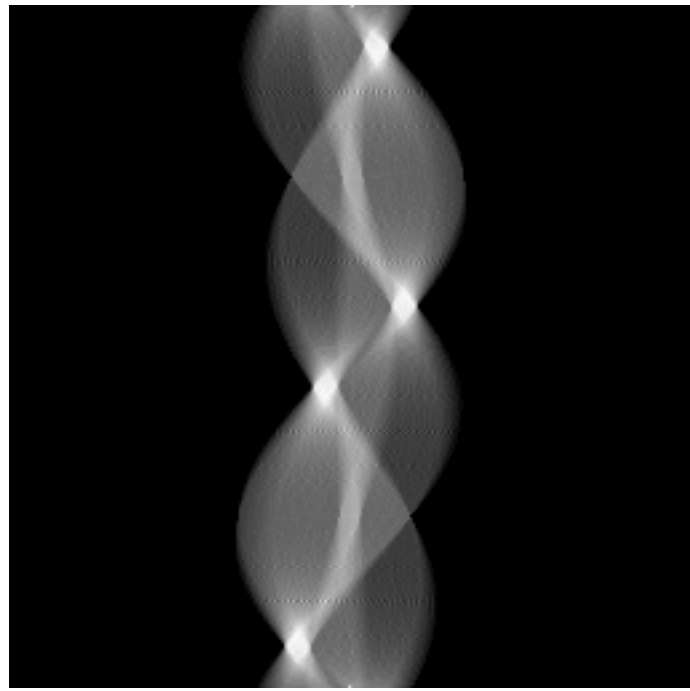
図6　ハフ変換の並列実行
角度0°～360°**を,** 2.5° **毎に**144**分割する. 角度**2.5k (k=0,1,…,143)**に**
対する投票を回路THETA(k)**で行い, 結果をバッファ**HOUGH_BUF(k)**に格納する.**
回路THETA(k)**とバッファ**HOUGH_BUF(k)**を,** 144**個並べて並列に実行させる.**



(a) **原画像**

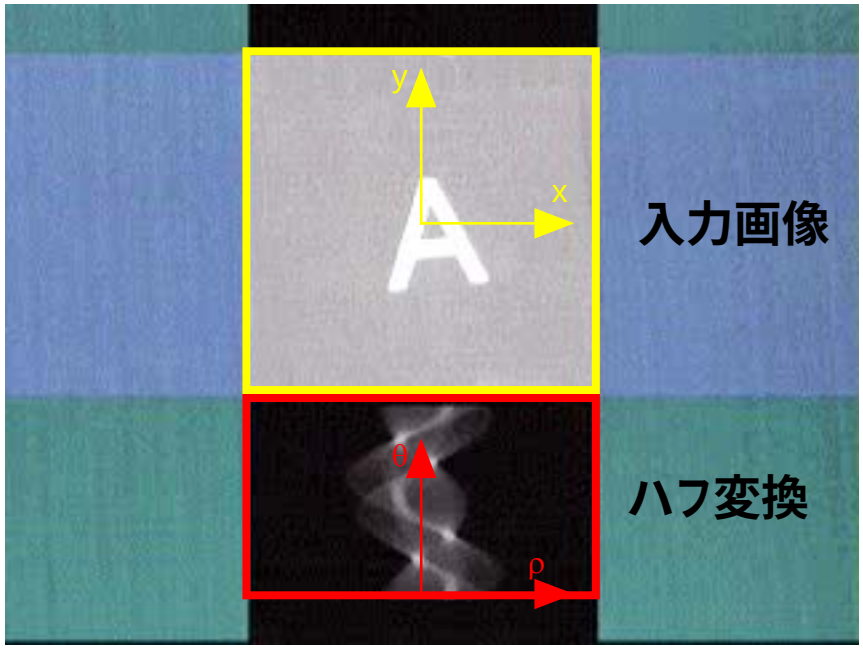


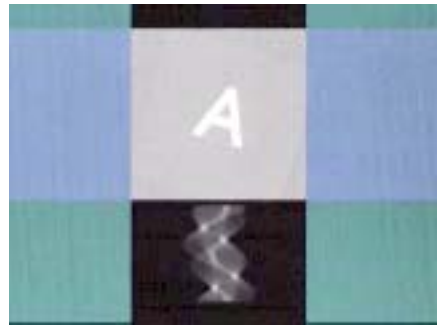
(b) Visual C/C++**プログラムによるハフ変換の結果**



(c) CycleC**回路記述のシミュレーションによるハフ変換の結果**

図7 CycleC**のシミュレーションによる回路検証**
Visual C/C++**プログラムによる変換結果と**
CycleC**回路記述のシミュレーションによる変換結果を比較する.**
双方が一致しており, ハフ変換の回路記述が正しいことを確認できる.





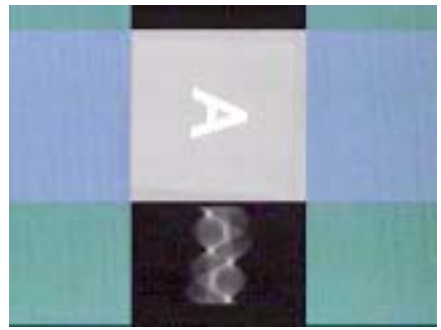
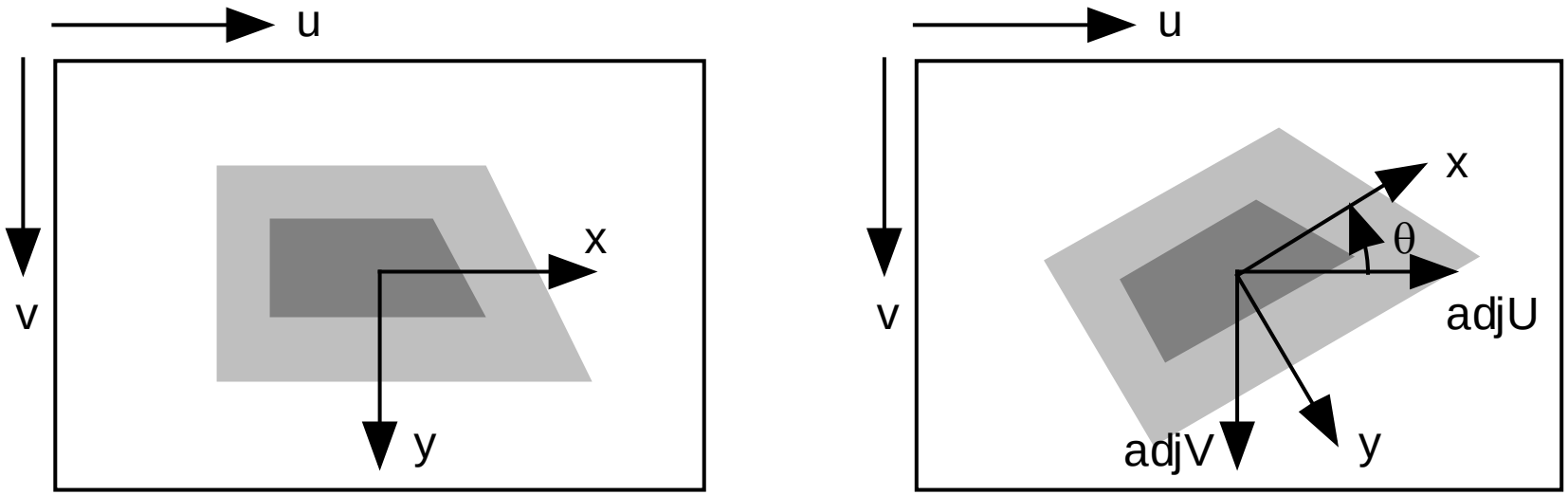


図8　リアルタイムハフ変換
入力画像の256×256ピクセルの領域を
ハフ変換し，結果を画像として出力する．



(a) **原画像**

(b) **回転画像**

図9　回転変換
画像(二次元データ)を中心周りに角度θ回転する.
回転画像の座標adjU,adjVに対応する原画像の座標x,yを計算し,
原画像の値を回転画像の対応する座標に書き込む.

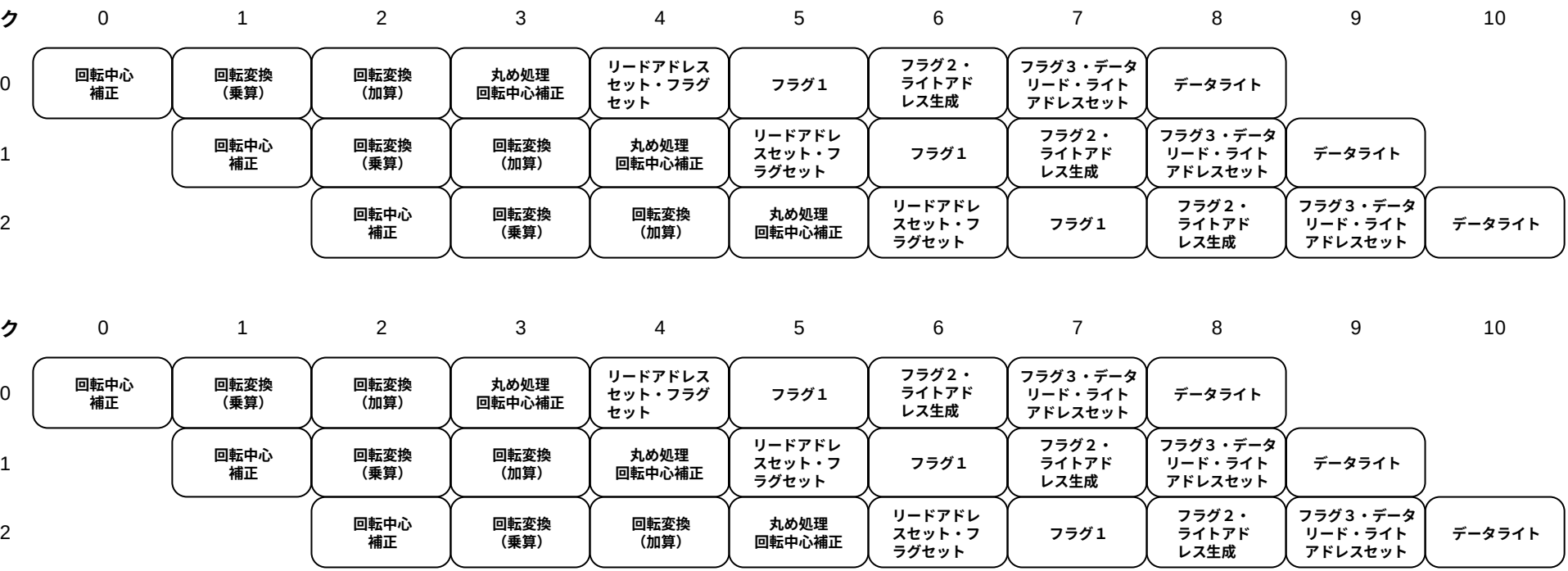


図１０　回転変換のパイプライン実行
回転後の座標値の計算，データの読み込み，
データの書き込みをパイプラインで実行する．

表１　CycleC によるセレクタの記述例

```
class Selecter {
public:
  uint1 reg;
  Selecter ( void ){ reg = 0;}
  void run ( uint1, uint1,
       uint1, uint1, uint1, uint1& );
};

void Selecter::run(uint1 clk, uint1 rst,
     uint1 a, uint1 b, uint1 s, uint1&y)
{
  uint1 temp = s ? a : b;

  if ( infer_clock(clk) || infer_reset(!rst) ) {
    if (!rst) {
      reg = 0;
    } else {
      reg = temp;
    }

    y = reg;
  }
}
```

表２　CycleC **で可能な** for **文と許されない** for **文**

(a)

```
for (i=0; i<8; i++) {
  out[i] = a[i] + b[i];
}
```

(b)

```
for (i=0; i<8; i++) {
  if (a[i] == 0) break;
  out = i;
}
```

(c)

```
flag=0;
for (i=0; i<8; i++) {
  if (a[i] == 0) flag=1;
  if (!flag) out = i;
}
```

表３　変数 a と b の交換の記述

(a) ANSI C

```
tmp = a;
a = b;
b = tmp;
```

(b) Handel-C

```
par {
   a = b;
   b = a;
}
```

表４　Handel-C によるパイプライン処理の記述例

```
a=-1; b=-1;
result=-1;

count=0;
while(1)
{
  par {
    a = count;  // clock 0
    b = a;      // clock 1
    result = b; // clock 2

    count++;
  }
}

// 結果
clock 0: a=0, b=-1, result=-1
clock 1: a=1, b= 0, result=-1
clock 2: a=2, b= 1, retult= 0
```

表５　CycleC によるハフ変換回路のクラス定義

```
class THETA {
public:
  void run( uint1, uint1,
     uint8, uint8, uint8, uint9, uint9,
     uint8&, uint8& );
  uint8 reg_rho;
  uint8 reg_image;
};

void THETA::run( uint1 CLK, uint1 RST,
     uint8 X, uint8 Y, uint8 IMAGE, uint9 SIN, uint9 COS,
     uint8& out_rho, uint8& out_image )
{
  if ( infer_clock( CLK ) || infer_reset( !RST ) ) {
    if ( !RST ) {
      reg_rho   = 0;
      reg_image = 0;
    } else {
      CALC_RHO( X, Y, SIN, COS, reg_rho );      // 組合せ回路で記述
      reg_image = IMAGE;
    }
  }
  out_rho   = reg_rho;    // レジスタを通してρの値を出力
  out_image = reg_image;  // レジスタを通して画素値を出力
}
```

表６　CycleC によるする組合せ回路の記述例

```
//  RHO=Y*COS-X*SIN を計算
inline void CALC_RHO( uint8 X, uint8 Y, uint9 SIN, uint9 COS, uint8& RHO )
{
  uint16 YCOS=0, XSIN=0, temp_rho=0;
  uint1  sign1=0, sign2=0, sign_rho=0;
  uint15 abs1=0, abs2=0, abs_rho =0;

  MULT( Y, COS, YCOS );         // YCOS(16 ビット) = Y(8 ビット)*COS(9 ビット)
  MULT( X, SIN, XSIN );         // XSIN(16 ビット) = X(8 ビット)*SIN(9 ビット)
  sign1 = get_bit(YCOS, 15);  abs1 = get_slice(YCOS, 14, 0);
  sign2 = get_bit(XSIN, 15);  abs2 = get_slice(XSIN, 14, 0);

  if( sign1 ) {
    if( sign2 ) {      // YCOS < 0 && XCOS < 0
      if( abs1 > abs2 ) {
        sign_rho = 1;  abs_rho  = abs1 - abs2;
      } else {
        sign_rho = 0;  abs_rho  = abs2 - abs1;
      }
    } else {           // YCOS < 0 && XCOS > 0
      sign_rho = 1;  abs_rho  = abs1 + abs2;
    }
  } else {
    if( sign2 ) {      // YCOS > 0 && XCOS < 0
      sign_rho = 0;  abs_rho  = abs1 + abs2;
    } else {           // YCOS > 0 && XCOS > 0
      if( abs1 > abs2 ) {
        sign_rho = 0;  abs_rho  = abs1 - abs2;
      } else {
        sign_rho = 1;  abs_rho  = abs2 - abs1;
      }
    }
  }

  set_slice( temp_rho, 14, 0, get_slice(abs_rho,14,0) );
  set_bit( temp_rho, 15, sign_rho );
  RHO = get_slice( temp_rho, 15, 8 );   // 上位 8 ビット
}
```

表７　CycleC によるハフ変換回路のシミュレーション記述

```
#define NTHETA 144      // 角度の分割数

for (i=0; i<255; i++) {
  for (j=0; j<255; j++) {
    x = i-127, y = j-127;
    // 信号 X を生成（符号ビット＋絶対値）
    if ( x >= 0 ) {
      X = x;
    } else {
      X = abs(x); set_bit(X, 7, 1);
    }
    // 信号 Y を生成（符号ビット＋絶対値）
    (信号 X の生成と同じ)
    IMAGE = image[i][j];

    for (t=0; t<NTHETA; t++) {  // 角度
      theta = 2.0*PI*(double)t/NTHETA;
      s = sin(theta);  c = cos(theta);
      // 信号 SIN を生成（符号ビット＋絶対値）
      if ( s >= 0 ) {
        SIN = (int)(s * 255);
      } else {
        SIN = (int)(fabs(s) * 255); set_bit(SIN, 8, 1);
      }
      // 信号 COS を生成（符号ビット＋絶対値）
      (信号 SIN の生成と同じ)

      THETA.run( CLK, RST,
                 X, Y, IMAGE, SIN, COS,
                 out_rho, out_image );

      // out_image の値を投票
      // 添字 0,1,…,255 が rho=-127,…,-1,-0,+0,+1,…,+127 に対応
      if ( out_rho >= 0 && out_rho < 128 ) {          // rho >= 0
        hough[t][out_rho+128] += out_image;
      } else if ( out_rho > 127 && out_rho < 256 ) {  // rho < 0
        hough[t][255-out_rho] += out_image;
      }
    } // End: t loop
  }
}
```

表８　Visual C プログラムによるハフ変換

```
#define NTHETA 144      // 角度の分割数

for (i=0; i<255; i++) {
  for (j=0; j<255; j++) {
    x = i-127, y = j-127;
    IMAGE = image[i][j];
    for (t=0; t<NTHETA; t++) {
      theta = 2.0*PI*(double)t/NTHETA;
      rho = y*cos(theta)-x*sin(theta);
      if ( rho >= 0 && rho < 128 ) {
        hough[t][int(rho)+128] += IMAGE;
      } else if ( rho > -127 && rho < 0 ) {
        hough[t][int(rho)+127] += IMAGE;
      }
    } // End: t loop
  }
}
```

表９　投票回路とバッファの接続

```
module hough (CLK, xRST, X, Y, image, SIN, COS, read_clk, web, act, ax, out);

  input         CLK, xRST;
  input [7:0]   X, Y, image;
  input [8:0]   SIN, COS;
  input         read_clk, web;
  input         act;  // ハフ変換中(画面中央部をスキャン中)は 1，それ以外は 0
  input [7:0]   ax;   // ハフ変換を画面に表示するときのアドレス
  output [15:0] out;

  wire [7:0]    rho;
  wire [7:0]    data;
  wire [7:0]    add_rho = (rho[7])? (8'b1111_1111-rho): {1'b1,rho[6:0]};
    // rho=-127,…,-1,-0,+0,+1,…,+127 に add_rho=0x00,0x01,…,0xfe,0xff が対応
  wire [7:0]    read_add = (act) ? add_rho : ax;
    // act=1 : 投票でアクセスするアドレス
    // act=0 : ハフ変換の表示でアクセスするアドレス
  wire [15:0]   out;  // 現在のハフ変換の値
  wire [15:0]   in = (act) ? (out + data) : 16'b0000_0000_0000_0000;
    // out + data : 現在のハフ変換の値に画素値を加算

  // 投票
  THETA theta( .CLK(CLK), .RST(xRST), .X(X), .Y(Y), .IMAGE(image),
               .SIN(SIN), .COS(COS),
               .out_rho(rho),           // rho : 符号 1 ビット，絶対値 7 ビット
               .out_image(data) );      // data : 符号なし 8 ビット(画素値)
  // ブロック RAM
  mem256 mem ( .ADDRA(read_add),        // 読み込みアドレス
               .CLKA(read_clk),         // 立ち上がりで読み込む
               .ADDRB(read_add),        // 書き込みアドレス(読み込みに一致)
               .CLKB(~read_clk),        // 立ち下りで書き込み
               .DIB(in),                // 書き込む値(ブロック RAM に)
               .WEB(web),               // 書き込み可能か
               .DOA(out) );             // 読み込む値(ブロック RAM から)
endmodule
```

表１０　Handel-C による回転変換の記述

```
//       回転変換
//   外部 RAM(バンク 0)上の 32 ビット二次元データ(256x256 点)を
//   データの真中を中心にある角度だけ回転させる.
//   回転角度のサインとコサインを tmpSin と tmpCos に指定する.
//   回転変換の結果を別の外部 RAM(バンク 1)に書き出す.
{
  unsigned 9    u, v;
  signed 9      x2, y2, tmpCos, tmpSin;
  signed 16     adjU, adjV, p1, p2, p3, p4;
  signed 10     x, y;
  unsigned 32   data32;
  unsigned 16   addr16;
  unsigned 1    validFlag, validFlag_q1, validFlag_q2, validFlag_q3, uBreakFlag;
  unsigned 2    dnWriteState;

  (中略)  // tmpSin, tmpCos をセット

  // 垂直方向走査
  for (v=0; v<256; v++) {
    par {
      dnWriteState=WRITE_IDLE;  // ライト状態初期化
      u=0; uBreakFlag=0;        // 水平方向ループ制御フラグ初期化
    }
    // 水平方向走査(パイプライン処理)
    while (1) {
      par {
        // Clock 0：回転中心補正
        adjU = adjs((signed)u-128, 8+8);
        adjV = adjs((signed)v-128, 8+8);
        // clock 1：回転変換(乗算)
        p1 = adjs( tmpCos, 8+8)*adjU; p2 = adjs(-tmpSin, 8+8)*adjV;
        p3 = adjs( tmpSin, 8+8)*adjU; p4 = adjs( tmpCos, 8+8)*adjV;
        // clock 2：回転変換(加算)
        x = (p1+p2)[15:6];
        y = (p3+p4)[15:6];
        // clock 3：丸め処理・回転中心補正
        x2 = (x\\1) + ((signed 8)0@x[0])+128;
        y2 = (y\\1) + ((signed 8)0@y[0])+128;
        // clock 4：回転後の座標の有効性判定
        if (x2>=0 && y2>=0 && x2<=255 && y2<=255) {
          validFlag = 1;
        } else {
          validFlag = 0;
```

```
        }
        // clock 4：リードアドレスセット
        // (clock 4+3 で data32 に読み出し結果(画素値)が格納される)
        ZbtAccess(0, data32, READ_FLAG,
                  (unsigned 3)4@(unsigned)(y2<-8)@(unsigned)(x2<-8), ZBT_READ);
        // clock 4+1
        validFlag_q1 = validFlag;
        // clock 4+2
        validFlag_q2 = validFlag_q1;
        // clock 4+3
        validFlag_q3 = validFlag_q2;
        // Clock 4+4-2：ライトアドレス生成
        addr16 = (unsigned)(v<-8) @ (unsigned)((u-6)<-8);
        if(u == 4+4 -2)     dnWriteState = WRITE_FIRST;  // データ 0
        if(u == 4+4 -1)     dnWriteState = WRITE_MIDDLE; // データ 1〜254
        if(u == 255+6+2 -1) dnWriteState = WRITE_LAST;   // データ 255
        // clock 4+4-1：データ 0 のライトアドレスセット
        if (dnWriteState == WRITE_FIRST) {
          ZbtSetAddress(1, (unsigned 3)3@(unsigned)(v<-8)@(unsigned 8)0,
                        ZBT_WRITE);
        }
        // clock 4+4：データライト
        if (dnWriteState == WRITE_MIDDLE) {  // データ 0〜254
          if (validFlag_q3) {  // データが有効なとき
            ZbtAccess(1, data32, WRITE_FLAG, (unsigned 3)3@addr16, ZBT_WRITE);
          } else {             // データが無効なとき
            ZbtAccess(1,      0, WRITE_FLAG, (unsigned 3)3@addr16, ZBT_WRITE);
          }
        }
        if (dnWriteState == WRITE_LAST) {    // データ 255
          if (validFlag_q3) {
            ZbtAccess(1, data32, WRITE_FLAG, (unsigned 3)3@addr16, ZBT_READ);
          } else {
            ZbtAccess(1,      0, WRITE_FLAG, (unsigned 3)3@addr16, ZBT_READ);
          }
        }
        u++;  // ループ変数加算
        if(u == 255+6+2) uBreakFlag = 1;  // 条件判定の遅延を削減
      } // End: Par Block
    if(uBreakFlag) break;
    } // End: u loop
  } // End: v loop
}
```

表１１　Handel-C による IP コアとの接続

```
//       三角関数テーブルコアとの結合
//       0～512 が 0 ° ～360 ° に対応
//
//  三角関数テーブルコアのコンポーネント宣言(VHDL)
//       component sin_cos_9to8_comb
//         port (
//         THETA:  IN  std_logic_VECTOR(8 downto 0);
//         SINE:   OUT std_logic_VECTOR(7 downto 0);
//         COSINE: OUT std_logic_VECTOR(7 downto 0));
//       end component;
//
unsigned 9 TrigoTheta9; // 出力変数(ポートに出力する値)
interface
  sin_cos_9to8_comb(signed 8 SINE, signed 8 COSINE) // コア名と入力の定義
  SinCos9(unsigned 9 THETA=TrigoTheta9)             // 実体名と出力の定義
  with{busformat="B<I>"};                           // バスフォーマットの定義

//  三角関数テーブルコアの参照
  TrigoTheta9 = Theta;                 // 出力変数のセット
  par {
    tmpCos = adjs(SinCos9.COSINE, 9); // 8 ビット入力 SinCos9.COSINE を 9 ビットに
    tmpSin = adjs(SinCos9.SINE,   9); // 8 ビット入力 SinCos9.SIN    を 9 ビットに
  }
```