システムＬＳＩ設計技術　103－7
（2001. 11. 29）

算術演算回路のレイアウトのためのビットスライス構造の抽出

小川　雄史　　高木　一義　　高木　直史

名古屋大学大学院工学研究科情報工学専攻
〒 464-8603 名古屋市千種区不老町
TEL: 052-789-5284, FAX: 052-789-3798

E-mail: {ogawa,ktakagi,ntakagi}@takagi.nuie.nagoya-u.ac.jp

あらまし　多くの算術演算回路はビットスライス構造と呼ばれる特徴的な回路構造をもつ。ビットスライス構造をもつ回路には入力データの各ビットを計算をする部分回路の間に規則性が存在する。この構造を回路から抽出し、レイアウトに反映させることによって総配線長の減少や小面積化が期待できる。本稿では算術演算回路のビットスライス構造の抽出手法を提案する。提案手法は、ネットリストに現れる規則的な回路構造の情報と HDL 記述の情報を利用してビットスライス構造を抽出する。本手法を実装し、実験により配列型乗算器や互除法に基づく有限体上の除算器などの算術演算回路でビットスライス構造が抽出できることを示した。

キーワード　算術演算回路, ビットスライス構造, 規則性抽出, ネットリスト

Bit-slice Extraction of Arithmetic Circuits for Layout Design

Yushi OGAWA, Kazuyoshi TAKAGI, and Naofumi TAKAGI

Department of Information Engineering, Nagoya University
Furo-cho, Chikusa-ku, Nagoya, 464-8603, Japan
TEL: 052-789-5284, FAX: 052-789-3798

E-mail: {ogawa,ktakagi,ntakagi}@takagi.nuie.nagoya-u.ac.jp

Abstract　Many arithmetic circuits have bit-sliced structure and exhibit regularities between each subcircuits that compute each bit of data. These regularities can be used to reduce the area and the wire length of the layout. In this report, we propose a method that extracts these bit-sliced structures of arithmetic circuits using information provided by the corresponding net-list and the HDL description. By using our method, extraction of bit-sliced structures of arithmetic circuits such as the array multiplier and the divider over a finite field based on Euclidean algorithm, was capable as shown in the experimental results.

Key words　arithmetic circuit, bit-sliced structure, regularity extraction, net-list

1. はじめに

近年の集積回路技術の進歩に伴い、様々な製品に特定用途向け回路 (ASIC) が搭載されるようになった。ASIC の性能をより高めるために、回路中で頻出する演算に特化した算術演算回路が ASIC に搭載されている。

算術演算回路は乗算器や除算器などで代表され、集積回路化に適した規則正しい回路構造を持つことが多い [1]。この規則正しい構造はビットスライス構造と呼ばれ、データの各ビットの計算を行う部分回路が同一かあるいは似た回路構造を持つ。ビットスライス構造を内在する回路においては、レイアウトの際に各ビットスライス内の記憶素子や論理素子を互いに近くに配置することによって配線長の減少や小面積化が期待できる。

従来、算術演算回路設計の際にビットスライス構造をレイアウトに反映させるために、設計者は経験や勘に基き手作業によりトランジスタの配置を決定していた。近年、ASIC に搭載する算術演算回路の設計においてはスタンダードセルベースの自動レイアウトツールを用いるようになってきている。しかし、従来の自動レイアウトツールは入力された回路をランダムロジックとして扱うために、ビットスライス構造をもつ算術演算回路の規則性はレイアウトに反映されない。そのため、ASIC ではデータパスを重視したレイアウトのためのツールも用いられているが、汎用のマイクロプロセッサなどを設計する際には設計者が手作業でレイアウトすることが多い。

今後ますます特定用途向けの算術演算回路をシステム毎に設計する必要性が高まり、回路に対して求められる性能は非常に高くなっていくと考えられる。また、IP の流通が加速した場合、HDL やネットリストの形態で流通しているソフト IP を ASIC に組み込むためにその IP の設計者以外の人がレイアウト作業をせざるを得ないという傾向が促進される可能性もある。そのためにもビットスライス構造をレイアウトに反映する自動レイアウトツールの実現は急務である。

現在までに行われてきた規則的な構造をもつ回路のレイアウトに関する研究として、データパス回路の規則性抽出に関するもの [2] [3] と抽出できたスライス内のセル配置に関するもの [4] [5] などが挙げられる。これらの手法はいずれも一般的なデータパス回路を対象としているので、配列型乗算器や複雑な回路など一部の算術演算回路のビットスライス構造を抽出するには十分でないこともある。また、[5] では規則性の抽出の対象がデータフローグラフ (DFG) であるため現在の標準的な設計フローに組み込むことは難しい。

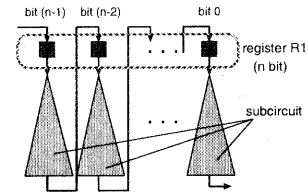

図 1　減算シフト型除算器の構造

本稿では算術演算回路のレイアウトのためのビットスライス構造の抽出手法を提案する。算術演算回路の回路構造が他の回路に比べて強い規則性をもつことに注目し、論理合成後のネットリストに現れる規則的な構造や、HDL 記述の情報を利用してビットスライス構造を抽出する。本手法はネットリストを抽出の対象としているため現在主流となる設計フローに柔軟に組み込むことができる。また、実験により配列型乗算器や互除法に基づく除算器などでビットスライス構造を抽出できることを示した。

2 節ではビットスライス構造の抽出問題を定式化し、3 節でビットスライス構造の抽出手法を提案する。4 節で実験結果を示し、5 節で本稿をまとめる。

2. 準　　備

乗算器や除算器などの算術演算回路の多くは、ビットスライス構造と呼ばれるデータの各ビットを処理する部分回路どうしが同一かあるいは似た構造をもつ。また、算術演算回路のアルゴリズム [6] [7] には漸化式に基づくものが多い。例えば、レジスタ Ri の値を ri とするとレジスタ $R1$ の値に関する漸化式は以下のように一般化される。

$$r1 = f(r1, r2, r3, \cdots)$$

この漸化式によりレジスタ R1 の値は更新される。実際の回路においては、レジスタ R1 の各ビットのフリップフロップから出たデータがそれぞれのビットの計算をする部分回路によって処理され、隣のビットや元のビットに戻るような構造により実現される (図 1)。ビットスライス構造をもつ算術演算回路ではこれらのフリップフロップと部分回路の組が各ビットにおいて同一かあるいは似た構造をもつ。本手法ではこの特徴を利用してビットスライス構造の抽出を行う。

図 2 にビットスライス構造抽出を含めた設計フローを示す。本手法の入力は HDL で記述された回路の論理合成後に得られるネットリストである。算術演算回路は他のデータパス回路と比べてより強い規則性をもつ。故に HDL 記述を自動合成ツールによって論理合成した後でもネットリストには規則的な構造が

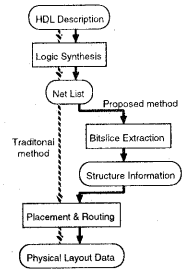

図 2　ビットスライス構造抽出を含む設計フロー

残る。また、ネットリストを用いることで現在主流の設計フローに対してビットスライス構造の抽出を柔軟に統合することが可能である。現在主流の設計フローではHDL記述の論理合成後に生成されたネットリストに従って配置配線をするが、ネットリストからビットスライス構造を抽出し、その結果を配置配線時に利用することで従来の設計フローでは得られなかった優れた回路性能が期待できる。

HDLで記述されたnビットの算術演算回路Cを表現する有向グラフ(ネットリスト)を$G(V,E)$(Vは2つの部分集合D, $L(=V-D)$に分割される)、ノード$v \in V$, $v' \in D$につけられるラベルを$l(v), n(v')$とする。Vは論理素子の集合に対応する。Dは回路の入力、出力、記憶素子の集合であり、データの入出力や保持を行う。実際の回路では入出力ポートやフリップフロップの集合に相当する。Lは論理ゲートや簡単な構成の演算器などの組合せ論理素子の集合である。$l(v)$はノード$v \in V$の論理機能を表す。$n(v')$はHDL記述時に設計者がつけた記憶素子$v' \in D$の名称に相当する。$v \in L$についてはラベル$n(v)$は定義されない。Eは配線の集合である。

あるグラフ$F(V_f, E_f)$と$F'(V_{f'}, E_{f'})$について、(1)FとF'が同型であり(つまりFとF'の間に同型写像ψが存在)、かつ(2)$v \in V_f$のラベル$l(v)$と$\psi(v) \in V_{f'}$のラベル$l(\psi(v))$が同一であるとき、グラフFとF'は“機能的に同型”であると呼ぶ。以下、“同型”は“機能的に同型”を意味する。また、Gの部分グラフ$G(V_i, E_i)$の全ての$v \in V_i - D$について、ある$u \in V_i$からの有向辺$e \in E_i$が少なくとも一つ存在する。以下、部分グラフはこの条件を満たしているとする。

ビットスライス構造抽出問題は以下のようになる。

ビットスライス構造抽出問題　グラフGで表現されたnビットの回路のkビット目の計算をする部分回路を表す部分グラフをB_kとしたとき、互いに同型な各B_k $(n_l < k < n - n_u)$を求める問題。ただし、$0 \leqq n_l < n-1$, $0 \leqq n_u < n-1$, $n_l + n_u < n-1$である。

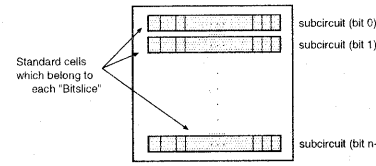

図 3　セルの配置

n_lとn_uはそれぞれ最下位ビットと最上位ビットから数えたビット数である。ビットスライス構造をもつ回路では$0 \leqq k \leqq n-1$のときに必ずしもB_kどうしが同型とならない。特に最下位ビットや最上位ビットは回路全体の制御に関わっているためB_0やB_{n-1}は$B_k(n_l < k < n - n_u)$と同型でないことが多い。そのため、ビットスライス構造抽出問題では下位のn_lビットと上位のn_uビットについては考慮していない。

3. ビットスライス構造の抽出

算術演算回路のビットスライス構造抽出問題を解くことで、データの各ビットを計算する部分回路を決定することができる。しかし、ビットスライス構造抽出問題では下位n_lビットと上位n_uビットのデータを計算する部分回路を表す部分グラフ$B_l, B_u (0 \leqq l \leqq n_l,\ n - n_u \leqq u \leqq n-1)$については述べられていない。しかし、$B_l$や$B_u$がそれ以外のビットの部分回路を表す$B_k(n_l < k < n - n_u)$と同型とならなくとも、図1のようなフリップフロップと部分回路という組を単位として考えれば、B_lやB_uの部分グラフはB_kの部分グラフと同型となる。つまり、B_lやB_uをB_kと“似た”回路として決定することができる。この点を利用し、提案手法では次の二つの方針でビットスライス構造抽出問題を解く。

下位ビットの部分回路を表す部分グラフB_lのノード集合をV_{B_l}としたとき、(a)ある$d \in (V_{B_l} \cap D)$とdから有向辺を順方向に推移的に辿った先に存在する全てのノードからなる集合$\{v | v \in (V_{B_l} - D)\}$の部分集合が導く部分グラフ$S_l$が$B_k$の部分グラフ$S_k(V_{S_k}, E_{S_k})$と同型であり、かつノード数$|V_{S_k}|$が最大となるように$B_k, B_l$を決定する。同様に$B_u$も決定する。また、(b)$n_l + n_u$を最小にする。

この方針で解くことで、B_kだけでなく、B_l, B_uも決定できる。また、S_kのフリップフロップに対応するノード$v \in (V_k \cap D)$のラベル$n(v)$を利用することでS_kが何ビット目の計算を行う部分回路であるかを判断する。

抽出できた各B_kのノード(スタンダードセル)を例えば図3のように配置することで、ビットスライス構造をレイアウトに反映させることができる。

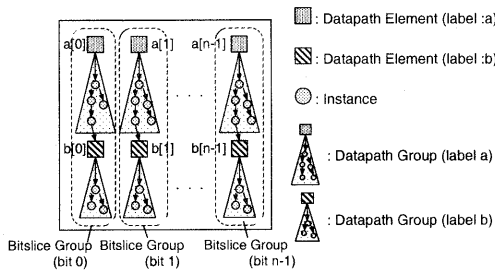

図4　データパスグループとビットスライスグループ

3.1 用　　語

本節以降で用いる用語を定義する。

データ依存関係　グラフ $G(V,E)$ において、$v_a \in V$ から有向辺を逆方向に辿っていき $v_b \in V$ に到達したとき v_a は v_b に対してデータ依存関係があるという。

データパスエレメント　レジスタや入出力ポート $d \in D \subseteq V$ のことである。回路を HDL で記述した際に設計者がつけた名称をもつ。名称はラベルとビット番号からなる。

ラベル、ビット番号　データパスエレメントにつけられた名称 ($label[bitnumber]$) である。ラベルは回路を HDL で記述した際に設計者がつけたものであり、ビット番号はそのデータパスエレメントが何番目のビットの情報を保持するかを表す。これらは論理合成ツールにより HDL からネットリストに継承される。

インスタンス　データパスエレメント以外の論理素子 $v \in L = V - D$ である。

データパスグループ　一つのデータパスエレメントと、それに対してデータ依存関係があるインスタンス (全てではない) の集合である。

ビットスライスグループ　各ビットの部分回路 B_k に相当する。データパスグループの集合である。

ビットスライス構造抽出問題は、データパスエレメントとインスタンスの集合であるデータパスグループがどのビットスライスグループに所属するかを規則性を考慮して決定することと考えることができ、以下の二つのステップで求めることができる (図 4)。

（1）　インスタンスがどのデータパスエレメントとデータ依存関係があるかを調べて、データパスグループを決定

（2）　各データパスグループがどのビットスライスグループに所属するかを決定

以下でそれぞれのステップについて述べる。

3.2 データパスグループの決定

インスタンスがどのデータパスグループに所属するかを決定する際には、算術演算回路がもつ規則的な回路構造を利用する。算術演算回路の各ビットスライスに対応する部分回路は同じ回路構造をしているはずである。そこで、提案手法ではラベルが同じデータパスエレメントを含んだデータパスグループの要素が導くグラフどうしは同型であり、かつ対応するノードの論理関数が同じになるようにインスタンスがどのデータパスグループに属すかを決定する。これを実現するためには同じラベルをもつ各データパスエレメントを起点として有向辺を辿っていき、各ビットの経路上でノードの論理関数の出現順が同じならば、それぞれの経路上のノードを対応するデータパスグループに所属させればよい。

以下の関数 **RegExtract()** によってデータパスグループを決定する。

```
RegExtract(G(V,E))
 begin
   /* データパスエレメント集合 D(⊆V) を「ラベル
   が同一という同値関係で分割 */
   D := {D_1, D_2, ..., D_f};
   while D ≠ φ do
     /* 各 D_i を根とする含まれるインスタンス数が最も多い
     部分木 T_i を決定 */
     for i=1 to n do
       PathTrace(D_i, |D_i|);
       D_i を根とする最もインスタンスが多くなる
       部分木 T_i を決定;
       T_i = {D_i, N_0, N_1, ...}
     end
     max := 0;
     /* 各 T_i の中から含まれるインスタンス数が最大となる
     T_max を決定 */
     for i=1 to n do
       /* Evaluate() は部分木のそれぞれのノードに含まれ
       るインスタンス数の合計を返す関数 */
       if Evaluate(T_i) > max
         T_max := T_i;
         max := Evaluate(T_i);
     end
     D_max := T_max ∩ D;
     /* T_max を各ビットについて分割し、データパスグループ
     を決定 */
     for i=1 to |D_max| do
       v_i ∈ D_max;
       /* S_i は n(v_i) をラベルとするデータパス
       グループ */
       S_i := {v_i};
       n := Next(v_i);
       N := Next(D_max);
       while n ∈ N do
         S_i := S_i ∪ {n};
         n := Next(n);
         N := Next(N);
       end
     end
     D := D − {D_max};
   end
 end
```

RegExtract() ではラベルと論理機能が同じデータパスエレメントの集合 (以下では同ラベルデータパスエレメント集合と呼ぶ) をまず作成する。つまり、同ラベルデータパスエレメント集合各 D_i の任意の要素 $v_i, v_j (i \neq j)$ に関して、"$Label(v_i) = Label(v_j)$ かつ $Logic(v_i) = Logic(v_j)$" が成り立つ (ただし、$Label(v)$ はノード v のラベルを返す関数であり、$n(v)$ のビット番号は含まれていない。$Logic(v)$ は $l(v)$ に対応する)。

RegExtract() は同ラベルデータパスエレメント集合に対して関数 **PathTrace**() を実行する。**PathTrace**() は引数として渡された集合の各要素の有向辺を順方向に辿った先に存在するインスタンスから成る集合 (ただし、インスタンスの論理機能は全て同じとする。この集合を同論理インスタンス集合と呼ぶ) を作成する。つまり、集合 G から作成される同論理インスタンス集合各 N_i の任意の要素 $v_i, v_j (i \neq j)$ に関して、"$Logic(v_i) = Logic(v_j)$ かつ $v_i = Next(u_i)$, $v_j = Next(u_j)$, $v_i \neq v_j$; $u_i, u_j \in G$" が成り立つ (ただし、$Next(x)$ はノード x の有向辺を辿った先に存在するノードを返す)。

```
PathTrace(G, |D_i|)
/* |D_i| は探索木の根ノードの要素数 */
begin
    G に関して同論理インスタンス集合 N_1, N_2, ..., N_k を列挙;
    for i=1 to k do
        if N_i ≠ φ
            if |N_i| ≧ |D| × TH      (0 ≦ TH ≦ 1)
                G の子として N_i を登録;
                PathTrace(N_i, |D_i|);
    end
end
```

例えば図5(円の中の番号はその論理関数の種類を表す) の同ラベルデータパスエレメント集合 $D1$ と $|D1|$ を **PathTrace**() の入力とした場合、生成される同論理インスタンス集合は $N1$ と $N4$ となる。

また、あるデータパスグループに所属するインスタンスを他のデータパスグループに重複して所属させないために "$v_i, v_j \notin F$" (F は既に決定済みのデータパスグループ) という条件を追加する。更に、一つのデータパスグループには一つのデータパスエレメントのみが含まれるので "$v_i, v_j \notin D$" という条件を加える。

作成した各同論理インスタンス集合を入力の集合の子とすることで探索木を作成し、各同論理インスタンス集合について **PathTrace**() を実行する。

PathTrace() 中の $TH (0 \leq TH \leq 1)$ は **PathTrace**() の入力となる集合 G の有効な同論理インスタンス集合を決定する条件を決める値である。部分回路の回路構造が全ビットで全て同じであると仮定すると、探索木の各ノードに含まれる要素数は探索木の根にあたる同ラベルデータパスエレメント集合の要素数と同じである。しかし、多くの算術演算回路では最上位ビットと最下位ビットにおいて回路の制御のための処理を行うため、他のビットを計算する部分回路と最上位、最下位ビットを計算する部分回路の構造が異なることがある。その際、探索木の各ノードの要素数が根ノードの要素数は等しくならない。例えば、図5で同論理インスタンス集合 $N1$ はその前の同ラベルデータパスエレメント集合 $D1$ と比べて要素数が一つ少ない。値 TH はこれを許す他、要素数が少なすぎる同論理インスタンス集合の生成を制限する。値 TH は回路に応じて経験的に定める。

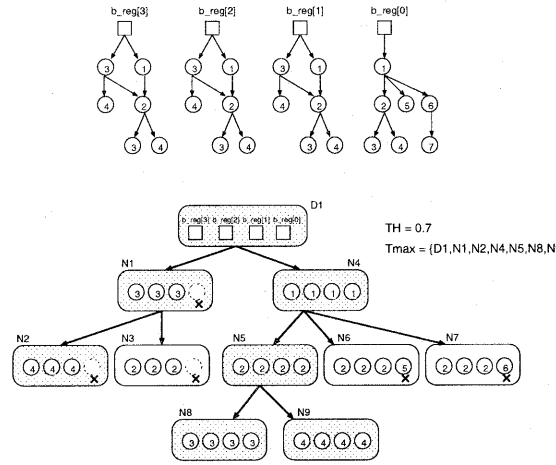

図5　同論理インスタンス集合をノードとする探索木

また、**RegExtract**() は各同ラベルデータパスエレメント集合について **PathTrace**() を実行した後、同ラベルデータパスエレメント集合を根とし各同論理インスタンス集合をノードとする探索木の中で最も多くインスタンスが含まれる部分木 (T_{max}) を選び、それよりデータパスグループを決定する。図5の場合は、$T_{max} = \{D1, N1, N2, N4, N5, N8, N9\}$ となる。このとき得られるデータパスグループは $S_0 =$ {b_reg[0],1,2,3,4}, $S_1 =$ {b_reg[1],1,2,3,4,3,4}, $S_2 =$ {b_reg[2],1,2,3,4,3,4}, $S_3 =$ {b_reg[3],1,2,3,4,3,4} となる。全てのデータパスグループが決定するまで以上のことを繰り返す。

3.3　ビットスライスグループの決定

各ビットを計算する部分回路を表すビットスライスグループ B を決定する。k ビット目のビットスライスグループ B_k はビット番号が k のデータパスエレメントを含むデータパスグループの集合とする。

表1 GF(2^8) 上の除算器のビットスライスグループ

BSG	DPG(# DataPathElement + # Instance)															
bit	m_reg	b_reg	v_reg	p_reg	u_reg	d_reg	a_reg	y	x	rslt	rslt_reg	g	st	s	CLK	RST
0	7	2	4	2	2	1	14	1	1	1	1	1	3	3	1	6
1	7	5	4	3	2	1	2	1	1	1	1	1	42	-	-	-
2	7	5	4	3	2	1	2	1	1	1	1	1	-	-	-	-
3	7	5	4	3	2	2	2	1	1	1	1	1	-	-	-	-
4	7	5	4	3	2	2	2	1	1	1	1	1	-	-	-	-
5	7	5	4	3	2	2	2	1	1	1	1	1	-	-	-	-
6	7	5	4	3	2	2	2	1	1	1	1	1	-	-	-	-
7	7	5	4	3	2	2	2	1	1	1	1	1	-	-	-	-
8	5	2	-	1	-	2	-	-	-	-	-	-	-	-	-	-

4. 実験結果

ビットスライス構造抽出のためのアルゴリズムをC++で実装し、配列型乗算器と $GF(2^8)$ 上の除算器[7]に対して適用した。[7]の除算器は互除法に基づく除算アルゴリズムに基づいている。配列型乗算器のデータパスグループの抽出結果を図6に、$GF(2^8)$ 上の除算器のビットスライスグループの抽出結果を表1に示す。

図6の上段は配列型乗算器のネットリストであり、下段は各ビットのデータパスグループである。従来の手法[2]では不十分であった規則的な構造の抽出が提案手法によって可能となった。表1の各行は $GF(2^8)$ 上の除算器の各ビットのビットスライスグループに属するデータパスグループに含まれる要素数(データパスエレメント数(=1)+インスタンス数)である。最上位や最下位ビットなどで数が他のビットと異なっていることがあるが、それは元々この回路ではそれらのビットの回路構造が他のビットの構造と異なるからである。ビット1のビットスライスグループ中のデータパスグループstに含まれるインスタンス数が他と大きく異なるが、これはレジスタstが回路全体の制御に関わっているためである。

5. まとめ

算術演算回路のビットスライス構造を反映するレイアウトのためのビットスライス構造抽出手法を提案した。提案手法は算術演算回路のネットリストに現れる規則的な構造やHDL記述の情報を利用してビットスライス構造を抽出する。また、ネットリストを入力とするため現在主流の設計フローに柔軟に組み込むことができる。実験により従来の一般のデータパス回路を対象とした手法では十分ではないこともあった配列型乗算器や互除法に基づく除算器からビットスライス構造の抽出に成功した。

今後の課題として、抽出できたビットスライス構造をレイアウトに反映させるためのセル配置手法の開発が挙げられる。

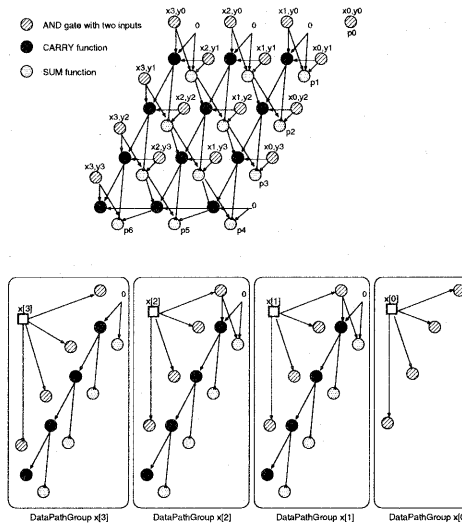

図6 4×4配列型乗算器のデータパスグループ

文献

[1] 高木直史 “乗算回路のアルゴリズム -高速計算と回路の規則性の両立-” 情報処理 Vol.37, No.2, pp.174-180, Feb.1996

[2] A.Chowdhary, S.Kale, P.Saripella, N.Sehgal, and R.Gupta “A general approach for regularity extraction in datapath circuits”, ICCAD98, pp.332-339, Nov.1998

[3] S.Hassoun and C.McCreary ”Regularity extraction via clanbased structural circuitdecomposition”, Computer-Aided Design, ICCAD99, pp.414-418, Nov.1999

[4] S.R.Arikati, and R.Varadarajan “A signature based approach to regularity extraction”, ICCAD97, pp.542-545, Nov.1997

[5] T.T.Ye, and G.D.Micheli “Data Path Placement with Regularity”, ICCAD2000, pp.264-270, Nov.2000

[6] M.D.Ercegovac and T.Lang “Division and Square Root Digit-Recurrence Algorithms and Implementations” Kluwer Academic Publishers, 1994

[7] 渡辺恭章, 高木直史, 高木一義 “Steinのアルゴリズムに基づく有限体上の除算アルゴリズム” VLD2001-59, 電子情報通信学会技術研究報告, pp.95-102, Jun.2001