Ullmann のアルゴリズムの
ハードウェアによる実装に関する研究

知識情報工学専攻
学籍番号 963715 齋藤秀充

指導教官 市川周一

目 次

1　はじめに

部分グラフ同型判定問題 (Subgraph Isomorphism Problem) は画像認識におけるシーン解析や化学情報における構造活性推測等，様々な分野に応用が可能である．しかし部分グラフ同型判定問題は NP 完全であることが知られており，実用的な時間で解くことは非常に困難である [1]．このような問題に対して，効率の良いアルゴリズムが見つからない場合は専用回路を用いて高速化することにより実用的な時間で解けるようになることが期待できる [2].

Ullmann のアルゴリズム [3] は，代表的な部分グラフ同型判定アルゴリズムの 1 つである．Ullmann によれば，このアルゴリズムはハードウェアに実装することが可能だが，必要な資源量が膨大になってしまう．しかも，Ullmann のアルゴリズムをハードウェアに実装した場合の具体的な回路規模の評価は行われていない．そこで本研究では，Ullmann のアルゴリズムのハードウェアへの実装方法について検討し，実用可能な回路規模での実装を目指す.

2　部分グラフ同型判定問題

この章では，まず最初にグラフ理論に関する用語を説明し，その後に部分グラフ同型判定問題について説明する．

2.1　グラフ理論に関する用語

グラフ (graph)G を $G = (V, E)$ と定義する．$V = \{v_1, v_2, \cdots, v_p\}$ は有限個の頂点 (vertex) からなる集合である．$p = |V|$ は頂点数である．$E \subseteq V \times V (= \{e_1, e_2, \cdots e_q\})$ は頂点対の有限集合で，頂点対 $e_i = (v_{i_1}, v_{i_2})$ を辺 (edge) と呼ぶ．$q = |E|$ は辺数である．頂点 v_i の次数 (degree) とは頂点 v_i から出ている辺の本数，すなわち v_i と隣接している頂点の個数のことである．枝の方向，すなわち頂点対の順序を考慮するものを有向グラフ (directed graph)，考慮しないものを無向グラフ (undirected graph) と呼び，本研究では無向グラフを扱う．全ての頂点対の間に辺を持つ無向グラフを完全グラフ (complete graph) と呼ぶ．完全グラフの辺数 q_c は $q_c = p(p-1)/2$ となる．

次に辺密度 (edge density) を定義する．グラフ G の辺密度とは G の辺数 q の，頂点数 p に対する完全グラフの辺数 q_c に対する比のことである．すなわち，辺密度 ed は $ed = q/q_c$ となる．

グラフのデータ表現には隣接行列 (adjacency matrix) を用いる．隣接行列は頂点間の接続関係を表した $p \times p$ の行列である．グラフ G の隣接行列を $A = [a_{ij}]$ とすると，グラフ G が無向グラフである時の隣接行列の各要素 a_{ij} は (1) 式で表せる．

$$a_{ij} = \begin{cases} 1 & (v_i, v_j) \in E \ \ or \ \ (v_j, v_i) \in E \\ 0 & (v_i, v_j) \notin E \ \ and \ \ (v_j, v_i) \notin E \end{cases} \qquad (1 \leq i, j \leq p) \tag{1}$$

(1) 式からわかるように，無向グラフの隣接行列は対称行列になり隣接行列の対角要素は'0' になる．また，完全グラフの隣接行列は対角要素以外全て'1' になる．各頂点の次数は隣接行列の各行について'1' の数を数えることで求めることができる．

2.2　部分グラフ同型判定問題とは

部分グラフ同型判定問題とは，グラフ G_α，G_β が与えられた時 G_α が G_β の部分グラフと同型であるか否か判定する問題である．グラフ $G_1 = (V_1, E_1)$ と $G_2 = (V_2, E_2)$ が同型であるとは，G_1 と G_2 の頂点集合 V_1，V_2 の間に 1 対 1 の対応関係があり，かつ E_1，E_2 の間に 1 対 1 の対応があることを言う．

2.3 部分グラフ同型判定アルゴリズム

部分グラフ同型問題を解くには，G_α と G_β の頂点間の写像を全て数え上げ，各写像すなわち G_β の部分グラフ $G' = (V', E')$ について V_α と V'，E_α と E' の間に 1 対 1 の対応関係がある (同型である) か否か調べる．

一般的な部分グラフ同型判定アルゴリズムでは頂点間の写像を根付木 (探索木) で表現し，深さ優先探索で探索木を巡回することにより全ての写像を数え上げる．図 1に $V_\alpha = \{1, 2, 3\}$ $(p_\alpha = 3)$，$V_\beta = \{1', 2', 3', 4'\}$ $(p_\beta = 4)$ の場合の探索木を示す．深さ優先探索では探索木中のある終端節点に行った後，途中まで木を遡り別の終端節点に行くという手順で探索木の巡回を行う．探索木の深さは p_α となり，各深さは G_α の頂点番号に相当する．探索木中の各節点に割り当てられた番号は G_β の頂点番号に相当する．よって各節点は深さ d での G_α 中の頂点 v_{α_d} から G_β の任意の頂点への写像に対応する．探索木の終端節点は G_α の全ての頂点に対する写像であり，これは大きさ p_α の G_β の部分頂点集合である．この部分頂点集合を含む G_β の部分グラフ G' が存在する場合，G_α と G' について V_α と V'，E_α と E' の間に 1 対 1 の対応関係があることを示す．これは部分グラフ同型の定義そのものである．

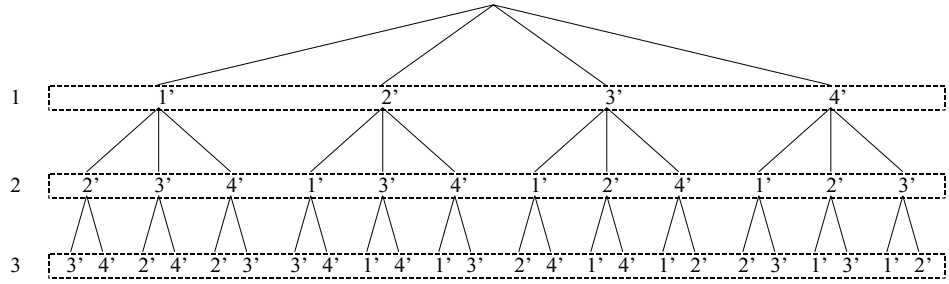


図 1: 探索木

探索木の終端節点の数は ${}_{p_\beta}P_{p_\alpha}$ 個となり，これは探索木全体を巡回すると探索空間が膨大になってしまうことを示している．そこであらかじめ部分グラフ同型になる可能性のない探索枝を切り捨て，探索空間の削減を行う必要がある．Ullmann のアルゴリズムでは Refinement Procedure という方法により探索空間の削減を行う．

3 Ullmann のアルゴリズム

本研究では代表的な部分グラフ同型判定アルゴリズムの 1 つである Ullmann のアルゴリズムを扱う．Ullmann のアルゴリズムは探索木を巡回する “探索木巡回アルゴリズム” と探索空間を削減する “Refinement Procedure” の 2 つにより構成される．以下にアルゴリズムの説明を行う．

3.1 探索木巡回アルゴリズム

まず，部分グラフ同型判定を行う 2 つの入力グラフを $G_\alpha = (V_\alpha, E_\alpha)$，$G_\beta = (V_\beta, E_\beta)$ とする．G_α と G_β の頂点数，辺数をそれぞれ p_α，q_α と p_β，q_β とする．

探索木巡回アルゴリズムは，深さ優先探索により探索木を巡回するアルゴリズムである．Ullmann のアルゴリズムでは要素数 $p_\alpha \times p_\beta$ の行列 $M = [m_{ij}]$ を定義する．M の値は探索木の各内部節点 (写像) に対応し，$m_{ij} = 1$ は G_α の頂点 v_{α_i} から G_β の頂点 v_{β_j} への写像において部分グラフ同型になる可能性があることを示す．探索木の終端節点での M を M' とすると，M' は以下のような性質を持つ．

- M' の各行は 1 個だけ’1’ を含み，残りは’0’ になる．

- M' の各列は最大 1 個だけ'1' を含む.

以上の性質は，G_α の各頂点が G_β の異なる頂点に写像されることを示す.

行列 M の初期値 $M^0 = [m^0_{ij}]$ は (2) 式のように決定される.

$$m^0_{ij} = \begin{cases} 1 & deg(v_{\alpha_i}) \leq deg(v_{\beta_j}) \\ 0 & otherwise \end{cases} \qquad (1 \leq i, j \leq p) \tag{2}$$

$deg(v_{\alpha_i})$ は G_α 中の i 番目の頂点 v_{α_i} の次数を，$deg(v_{\beta_j})$ は G_β 中の j 番目の頂点 v_{β_j} の次数を示す．部分グラフ同型になるためには，G_α と G_β の部分グラフの次数が一致する必要がある．そのため G_α の頂点で G_β の任意の頂点より次数が多いものはこの条件に合わないためあらかじめ初期値 m^0_{ij} を 0 にする．また，探索木の終端節点 M' において $m'_{ij} = 1$ になるための必要条件は $m^0_{ij} = 1$ になることである．よって探索木巡回アルゴリズムは $m_{ij} = 1$ に対応する探索木中の節点を巡回する．終端節点でない場合 (深さ $d < p_\alpha$) の M の値は (3) 式のようになる．ここで M_i は M の i 行目を取り出して得られる行ベクトルである.

$$M_i = \begin{cases} 1 \text{ 個だけ'1'，残りは'0'} & 0 \leq i \leq d \\ M^0_i & d+1 \leq i \leq p_\alpha \end{cases} \tag{3}$$

以上に示す性質より，探索木巡回アルゴリズムは $m_{dk} = 1$ となる k を見つけ，m_{dk} 以外を'0' にすることにより深さ d における頂点の写像を決定する．図 2に探索木巡回アルゴリズムを示す.

Step 1. $M := M^0$; $d := 1$; $H_1 := 0$;
for all $i := 1, \cdots, p_\beta$ set $F_i := 0$;
refine M; if exit FAIL then terminate algorithm;
Step 2. If there is no value of j such that $m_{dj} = 1$ and $f_j = 0$ then go to step 7;
$M_d := M$;
$k := 0$;
Step 3. $k := k + 1$;
if $m_{dk} = 0$ or $f_k = 1$ then go to step 3;
for all $j \neq k$ set $m_{dj} := 0$;
refine M; if exit FAIL then go to step 5;
Step 4. If $d < p_\alpha$ then go to step 6 else give output to indicate that an isomorphism has been found;
Step 5. If there is no $j > k$ such that $M_d(d, j) = 1$ and $f_j = 0$ then go to step 7;
$M := M_d$;
go to step 3;
Step 6. $H_d := k$; $F_k := 1$; $d := d + 1$;
go to step 2;
Step 7. If $d = 1$ then terminate algorithm;
$d := d - 1$; $M := M_d$; $k := H_d$; $F_k := 0$;
go to step 5;

図 2: 探索木巡回アルゴリズム

図 2で，p_β ビットの 2 進ベクトル $\{F_1, \cdots, F_i, \cdots, F_{p_\beta}\}$ は現時点でどの列が使用済みであるかを記録する．$F_i = 1$ は i 番目の列が使用済みであることを示す．ベクトル $\{H_1, \cdots, H_i, \cdots, H_{p_\alpha}\}$ は探索木の各深さで選択されている列を記録する．$H_d = k$ は深さ d で k 列目が選択されていることを示す．":=" は代入記号で $d := d+1$ は $d+1$ を d に代入することを示す．特に $M := M_d$ のように行列について書かれている場合は，行列全体をコピーすることを示す．各深さにおける M の値は探索木を下るときに M_d に格納され，探索木を遡る時に M_d から読み出される．Refine M で Refinement Procedure を実行し探索空間の削減を行う.

3.2 Refinement Procedure

Ullmann のアルゴリズムでは探索木の内部節点で Refinement Procedure という “同型可能性判定” を行う．Refinement Procedure により部分グラフ同型になる可能性がないと判定された場合，その節点以下の部分木は切り捨てられ探索空間が削減される．Refinement Procedure によるオーバーヘッドは増えるが，探索空間削減の効果により実行時間は劇的に減少する．図 3に Refinement Procedure を示す．

Step 1. elim := 0;
$i := 1$;
Step 2. $k := 1$;
sc := $2^{(p_\alpha - 1)}$;
$h := 1$;
Step 3. if sc & $A_i = 0$ then go to step 4;
$\mathrm{lst}_k := h$;
$k := k + 1$;
Step 4. sc := sc $\times 2^{-1}$;
$h := h + 1$;
if $k \neq \deg_i + 1$ then go to step 3;
Step 5. $j := 1$;
sc := $2^{(p_\beta - 1)}$;
Step 6. if M_i & sc $= 0$ then go to step 9;
$h := 1$;
Step 7. $x := \mathrm{lst}_h$;
if M_x & $B_j = 0$ then go to step 8;
$h := h + 1$;
if $h \neq \deg_i + 1$ then go to step 7 else go to step 9;
Step 8. $M_i := M_i$ & NOT sc;
elim := elim +1;
Step 9. sc := sc $\times 2^{-1}$;
$j := j + 1$;
if $j \neq p_\beta + 1$ then go to step 6;
Step 10. if $M_i = 0$ then go to FAIL exit;
$i := i + 1$;
if $i \neq p_\alpha = 1$ then go to step 2;
if elim $\neq 0$ then go to step 1;
go to SUCCEED exit;

図 3: Refinement Procedure

図 3で h，i，j，k，x は整数である．elim は Refinement Procedure により’1’ から’0’ に変わった M 中の要素の数を示す．A，B はそれぞれグラフ G_α，G_β の隣接行列である．A_i は行列 A の i 行目を取り出して得られる行ベクトルである．$\deg_i$ はグラフ G_α の i 番目の頂点の次数である．lst は G_α の i 番目の頂点と隣接している頂点番号のリストである．sc は 1 ビットだけ’1’ を含む 2 進ベクトルで，行ベクトル中の’1’ が現れる位置の探索に使用する．&はビットごとの AND 演算を示す (例: 1100 & 1010 = 1000)．NOT は全てのビットの NOT 演算を示す (例: NOT 1100 = 0011)．

次に Refinement Procedure による同型可能性判定の判定条件について述べる．

3.2.1 同型可能性判定における判定条件

Refinement Procedure は部分グラフ同型に必要な条件が満たされているかどうか調べ，条件が満たされずかつ $m_{ij} = 1$ である時に $m_{ij} = 0$ にして部分木の切り捨てを行う．

V_α 中の i 番目の頂点を v_{α_i}，V_β 中の j 番目の頂点を v_{β_j} とする．$\{v_{\alpha_1}, \cdots, v_{\alpha_x}, \cdots, v_{\alpha_\gamma}\}$ は G_α 中で v_{α_i}

と隣接している頂点集合とする．部分グラフ同型の定義より，v_{α_i} が v_{β_j} に対応しているならそれぞれの $x = 1, \cdots, \gamma$ について v_{α_x} に対応しているような v_{β_j} と隣接する V_β 中の頂点 v_{β_y} が存在していなければならない．この場合 $\{v_{\alpha_x}, v_{\beta_y}\}$ に対応する M の要素 m_{xy} は'1' になる．すなわち (4) 式に示す条件を満たせばよい．

$$\underset{1 \le x \le p_\alpha}{(\forall x)} \ ((a_{ix} = 1) \Rightarrow \underset{1 \le x \le p_\beta}{(\exists y)} \ (m_{xy} \cdot b_{yj} = 1)) \tag{4}$$

Refinement Procedure は $m_{ij} = 1$ となる i，j について (4) 式を満たすかどうかチェックし，もし満たさない場合は $m_{ij} = 1$ から $m_{ij} = 0$ にする．

Refinement Procedure は同型の可能性がない節点に対応する M 中の'1' の値を'0' にすることで探索空間の削減を行う．これを M が変化しなくなるまで続け，M が変化しなくなった場合，SUCCEED exit を行う．また M のいずれかの行の要素が全て'0' になった場合，FAIL exit を行う．探索木巡回アルゴリズムの Step 1 での Refinement Procedure が FAIL exit になった場合，部分グラフ同型が見つからないことを示す．よってアルゴリズムを停止する．探索木の終端節点において Refinement Procedure が SUCCEED exit になった場合，V_α と V_β の間で 1 対 1 の対応が見つかり，かつ部分グラフ同型の条件が満たされたことを示す．

また m_{ij} を修正する際，i，j の順序に関する制約条件は存在しないため，Refinement Procedure はハードウェア化により並列に実行することが可能である．以下に Refinement Procedure のハードウェア化について説明する．

3.2.2 Refinement Procedure のハードウェア化

Ullmann[3] によれば，Refinement Procedure はハードウェアへの実装が可能である．以下に Ullmann による Refinement Procedure のハードウェア化の提案について示す．

Refinement Procedure は (4) 式を満たさずかつ $m_{ij} = 1$ であるものについて m_{ij} を 1 から 0 に修正する．(4) 式は以下のように変形することができる．ここで，$1 \le i \le p_\alpha$，$1 \le j \le p_\beta$，$1 \le x \le p_\alpha$ とする．

1. A，B，M を真偽値 ($1 \to$ True，$0 \to$ False) の行列とする．ここで，$p_\alpha \times p_\beta$ 個の要素からなる行列 $R = [r_{xj}]$ を以下のように定義する．

$$r_{xj} = (\exists y)(m_{xy} \cdot b_{yj}) \tag{5}$$

2. (4) 式は (5) を用いて以下のように変形可能である．

$$(\forall x)(\bar{a}_{ix} \vee r_{xj}) \tag{6}$$

式の変形手順を以下に示す．

$$\begin{aligned}
& (\forall x)((a_{ix} = 1) \Rightarrow (\exists y)(m_{xy} \cdot b_{yj} = 1)) && //(4) \text{ 式} \\
\Leftrightarrow \ & (\forall x)(a_{ix} \Rightarrow (\exists y)(m_{xy} \cdot b_{yj})) && //\text{Integer} \to \text{Boolean} \\
\Leftrightarrow \ & (\forall x)(a_{ix} \Rightarrow r_{xj}) && //(5) \text{ 式} \\
\Leftrightarrow \ & (\forall x)(\bar{a}_{ix} \vee r_{xj}) && //(a \Rightarrow b) \Leftrightarrow (\bar{a} \vee b)
\end{aligned}$$

3. (4) 式が満たされる場合 (6) 式の値は'True' に，満たされない場合'False' になる．(4) 式が満たされずかつ $m_{ij} =$ 'True' の場合に $m_{ij} =$ 'False' にするには，(6) 式より (7) 式を導く．

$$m_{ij} = m_{ij} \cdot (\forall x)(\bar{a}_{ix} \vee r_{xj}) \tag{7}$$

(5)，(7) 式は AND，OR 等の論理ゲートを用いた組合せ回路でハードウェアによる実装が可能である．具体的には (5) 式では各 x，j について $m_{xy} \cdot b_{yj}$ の計算で 2 入力 AND ゲートを p_β 個，$(\exists y)(m_{xy} \cdot b_{yj})$ の計算で 2 入力 OR ゲートを $(p_\beta - 1)$ 個使用する．(7) 式では各 i について $\bar{a}_{ix}$ の計算で NOT ゲートを 1 個，各 i，j について $\bar{a}_{ix} \vee r_{xj}$ の計算で 2 入力 OR ゲートを p_α 個，$(\forall x)(\bar{a}_{ix} \vee r_{xj})$ の計算で 2 入力 AND ゲートを $(p_\alpha - 1)$ 個，$m_{ij} \cdot (\forall x)(\bar{a}_{ix} \vee r_{xj})$ の計算で 2 入力 AND ゲートを 1 個使用する．ハードウェア化により Refinement Procedure は並列に実行することが可能であるが，論理ゲートに要するハードウェアの使用量が $O(p_\alpha p_\beta^2)$ と見積もられるため，大量のハードウェアを使用しないと Refinement Procedure が実装できないことが予想できる．

本研究では，まず Ullmann の提案に従い Refinement Procedure を組合せ回路で実装する．しかし，組合せ回路で実装するとハードウェアの量が $O(p_\alpha p_\beta^2)$ になり，ハードウェアへの実装が困難である．そこで，回路規模を縮小するために Refinement Procedure を順序回路により実装する方法を提案する．順序回路で実装すると回路規模が小さくなるが実行時間が長くなってしまう．しかし，順序回路するのは Refinement Procedure だけで，全体の実行時間は探索木巡回アルゴリズムの実行時間が影響するため，逐次化によりどの程度遅くなるかを確かめる必要がある．

本研究では実装対象のデバイスとして，試作用 LSI の一種である FPGA(Field Programmable Gate Array) を採用する．FPGA として Lucent 社の OR2CxxA FPGA[4] への実装を想定した場合について，提案した実装方法について回路規模，動作周波数を見積り，ハードウェアに実装できるか否か調べる．そして，FPGA に実装した場合の実行時間の予測値を測定し，実装方法の評価を行う．

4 設計

4.1 Ullmann の提案手法によるハードウェア実装方法

この節では Ullmann の提案手法 [3] によるハードウェアの実装方法について述べる．実装した回路は，探索木巡回アルゴリズムと Refinement Procedure の 2 つにより構成される．探索木巡回アルゴリズムは図 2を基に実装し，Refinement Procedure は Ullmann の提案に従い，組合せ回路により実装する．以下に各アルゴリズムを実現する回路について説明する．

4.1.1 探索木巡回アルゴリズム

図 4に探索木巡回アルゴリズムの概念図を示す．探索木巡回アルゴリズムは以下に示す回路により構成される．

- priority encoder

  Step 3 で k の値を求める時，M の d 行目の中で最初に “未使用かつ’1’ になる” 列を求めればよい．これはプライオリティエンコーダを用いて実装できる．

- decoder1

  デコーダ 1 は k の値を入力として上位 k ビット目が’1’ でその他の値が’0’ であるような 2 進ベクトル (例: $k = 3$ の場合，“ 0 0 1 0 0 $\cdots$ 0 ”) を生成するものである．このベクトルの値は F の修正 (Step 6, 7)，M の d 行目の修正 (Step 3) に使用する．

- decoder2

  デコーダ 2 は k の値を入力として上位 k ビット目まで全て’0’ で $k+1$ ビット目以降は全て’1’ であるような 2 進ベクトル (例: $k = 3$ の場合，“ 0 0 0 1 1 $\cdots$ 1 ”) を生成する．Step 5 の if 文では $k+1$

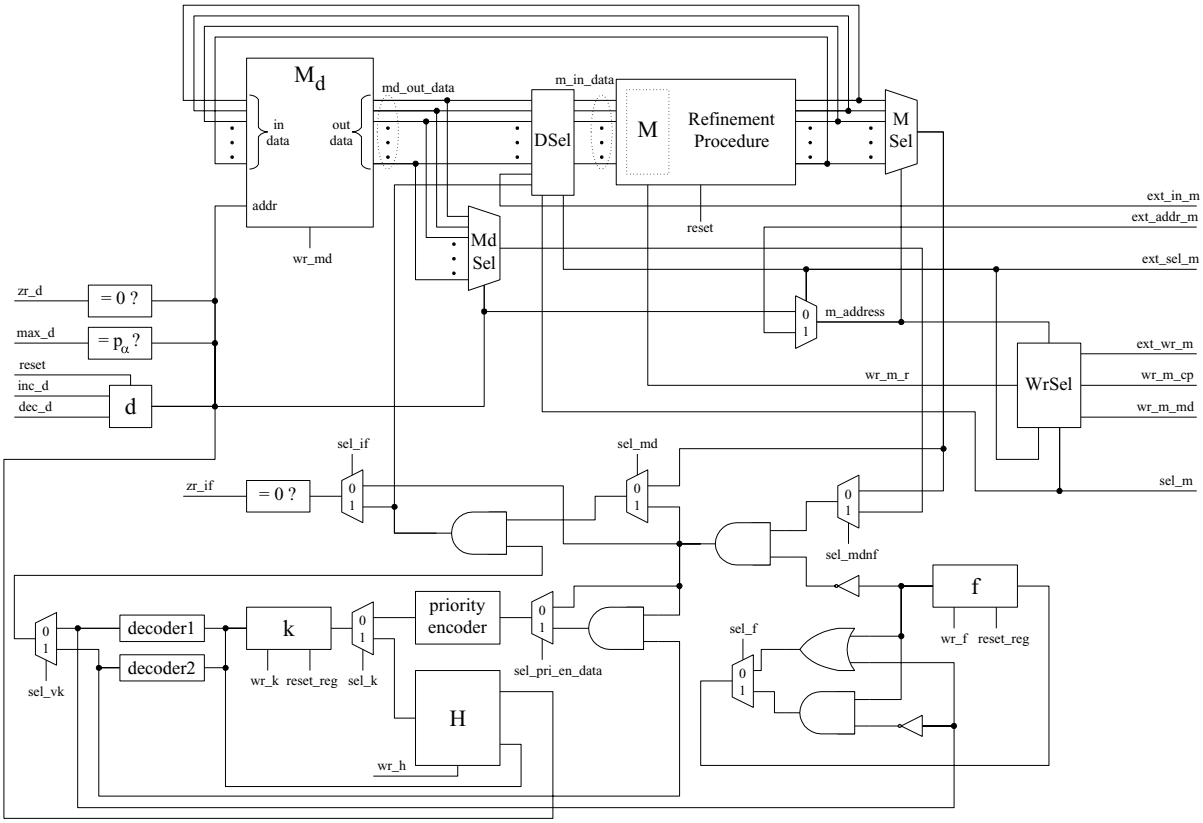


図 4: 探索木巡回アルゴリズムのデータパス

ビット目以降について評価するため，デコーダ 2 によって生成された 2 進ベクトルと AND 演算を行うことで上位 k ビットを無視することができる．

- M_d

  M_d は探索木のそれぞれの深さでの M を格納するメモリである．それぞれの深さの M は違う値になるため，全ての深さ ($d_{max} = p_\alpha$) について M を格納する必要がある．M と M_d の間の読み書きは高速化のため，全行同時に行うことにする．M のデータ幅は p_β, 行数が p_α であることから，M_d のハードウェア使用量は $O(p_\alpha^2 p_\beta)$ となる．

- H

  H は探索木のそれぞれの深さで選択されている列の番号 (k) を格納する．k のデータ幅は $\log p_\beta$ ビットになるため，H のハードウェア使用量は $O(p_\alpha \log p_\beta)$ となる．

- k

  プライオリティエンコーダの出力である k の値は M, M_d の値により変化するため，Step 3, 7 で k の値が修正された時点で k の値を保存しておく必要がある．k の格納にはレジスタを用いる．k のハードウェア量は $O(\log p_\beta)$ である．

- F

  p_β ビットの 2 進ベクトル F の値は，Step 6, 7 で F が修正された時点でその値を保存する．F のハードウェア量は $O(p_\beta)$ である．

- MSel, MdSel

  Step 2, 3 では M の行ベクトルを使用するために，行ベクトルを取り出すためにマルチプレクサ (図 4: MSel) を用いる．また Step 5 で M_d の出力から 1 行取り出すためにもマルチプレクサ (図 4: MdSel) を用いる．それぞれのマルチプレクサのデータ幅は p_β，入力数は p_α となるため，ハードウェア使用量は $O(p_\alpha p_\beta)$ となる．

- カウンタ (d)

  d の値は Step 6 でインクリメントされ，Step 7 でデクリメントされる．これはアップダウンカウンタを用いる．

- 比較器

  Step 4 で $d = p_\alpha$ のチェックを行う際，比較器を用いる．比較対象はカウンタ (d) の出力と p_α の値である．比較器の出力は，$d = p_\alpha$ であるときに 0, そうでない時に 1 となる．

- ゼロチェック回路

  ゼロチェック回路は $d = 0$ のチェック (Step 7)，if 文の判定 (Step 2，5) で使用する．回路の出力は，$d = 0$ である時に 0, そうでない時に 1 となる．

- DSel

  M の値を更新する際，以下の 3 通りについて行われる．

  - d 行目を更新 (Step 3)
  - M_d の値を全ての行について同時に更新 (Step 5, 7)
  - M の初期値を 1 行ずつ入力し更新 (アルゴリズム開始前)

  M の入力データは複数のマルチプレクサにより必要なものが選択された後に M に入力される．この複数のマルチプレクサをまとめて DSel とする．図 5(a) に DSel の概要図を示す.

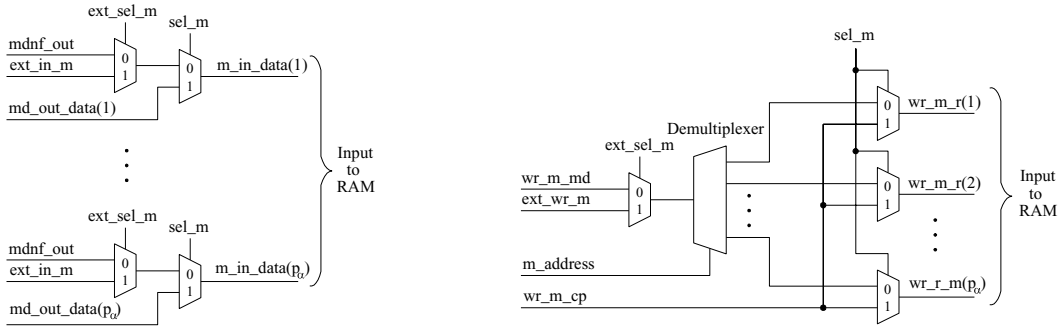
- WrSel

  DSel の項でも述べたように，M の更新には 3 通りの方法がある．その際，M への入力データと共に M の各行に対する Write Enable 信号も変化する．各行に対する Write Enable 信号の値は書き込みたい行のアドレスを入力とするようなデマルチプレクサと複数のマルチプレクサより決定される．これらをまとめて WrSel とする．図 5(b) に WrSel の概要図を示す.

図 6に探索木巡回アルゴリズムの制御ユニットの状態遷移図を示す．start_ull, finish_ref, result_ref, zr_if, max_d, zr_d は状態遷移のために必要な信号である．start_ull は Ullmann のアルゴリズムの開始信号で，start_ull='1' になるとアルゴリズムの実行を開始する．finish_ull は Refinement Procedure の終了を検出するための信号で，Refinement Procedure が終了すると finish_ref='1' になる．result_ref は Refinement Procedure の終了方法を検出するための信号で，SUCCEED exit の時 result_ref='1' に，FAIL exit の時 result_ref='0' になる．zr_if は STEP2, 5 の if 文の結果に相当するもので，zr_if='1' の時 “条件 = 真 (True)” に，zr_if='0' の時 “条件 = 偽 (False)” になったことを示す．max_d は $d = p_\alpha - 1$ すなわち探索木の終端節点かどうかを示す信号で，max_d='0' の時 $d = p_\alpha$ であることを示す．zr_d は $d = 0$ かどうかを示す信号で，zr_d='0' の時 $d = 0$ であることを示す．各状態の動作を以下に示す.

- INIT

  INIT は初期状態である．start_ull='1' である間は INIT のままである．start_ref='1' になると STEP1 に移り，アルゴリズムの実行を開始する．



(a)DSel (b)WrSel

図 5: 部分回路 (DSel，WrSel)

- STEP1

  STEP1 ではカウンタ，レジスタに対し Reset 信号を入力し，d, k, F の初期化する．

- START_REF1, START_REF2

  START_REF1, START_REF2 では Refinement Procedure の開始信号を出力し，Refinement Procedure の実行を開始する．START_REF1 は Step 1 での，START_REF2 は Step 3 での Refinement Procedure にそれぞれ対応する．

- RUN_REF1

  RUN_REF1 では Step 1 での Refinement Procedure で finish_ref='1' になるまで待ち，Refinement Procedure が終了し finish_ref='1' になると result_ref の値により次の状態を決定する．FAIL exit(result_ref='0') の時は TERMINATE に移りアルゴリズムを停止する．SUCCEED exit(result_ref='1') の時は STEP2 に移る．

- STEP2

  STEP2 では Step 2 での if 文の条件判定を行う．条件 = 真 (zr_if='1') の時は MD_COPY に移り M の値を M_d にコピーする．条件 = 偽 (zr_if='0') の時は STEP7_1 に移る．

- MD_COPY

  MD_COPY では Step 2 での $M_d := M$ に当たる処理を行う．M のコピーは全行について並列に行われるため 1 クロックで済む．

- STEP3_F2_1, STEP3_F5_1

  Step 3 に行く方法は 2 通りあり，Step 2 から来る場合 (STEP3_F2_1(2)) と Step 5 から来る場合 (STEP3_F5_1(2)) がある．別々の状態に分けたのは k の修正方法が Step 2 から来る場合と Step 5 から来る場合で違うからである．STEP3_F2(5)_1 は Step 3 中の第 1 ステップで，k の値の更新を行う．

- STEP3_F2_2, STEP3_F5_2

  STEP3_F2_2, STEP3_F5_2 は Step 3 中の第 2 ステップで，M の d 行目の修正 (上位 k ビット目を'1' にし，残りを'0' にする) を行う．

- RUN_REF2

図 6: 制御ユニット状態遷移図 (探索木巡回アルゴリズム)

RUN_REF2 では Step 3 での Refinement Procedure で finish_ref='1' になるまで待ち，Refinement Procedure が終了し finish_ref='1' になると result_ref の値により次の状態を決定する．FAIL exit(result_ref='0') の時は STEP5_F34(Step 5) に移り，SUCCEED exit(result_ref='1') の時は STEP4(Step 4) に移る．

- STEP4

  STEP4 では Step 3 の Refinement Procedure が SUCCEED exit になった時，d の値をチェックする．$d = p_\alpha$(max_d='0') の時は終端節点の Refinement Procedure が SUCCEED exit になったことを示し，OUTPUT に移り部分グラフ同型が検出されたことを出力する．$d \neq p_\alpha$(max_d='1') の時は STEP6 に移る．

- OUTPUT

  OUTPUT では同型検出数を示すカウンタをインクリメントし，同型を発見したことを出力する．

- STEP5_F34

Step 5 に行く方法は 2 通りあり，Step 3 または Step 4 から来る場合 (STEP5_F34) と，Step 7 から来る場合 (STEP5_F7) がある．別々の状態に分けたのは，Step 7 から来た場合は $M := M_d$ に当たる処理を Step 7 ですでに行っているため，同じことを Step 5 で行う必要がないからである．STEP5_F34 では Step 5 での if 文の条件判定を行い，条件 = 真 (zr_if='1') の場合は M_COPY_5 に移り M_d のコピーを行う．条件 = 偽 (zr_if='0') の場合は STEP7_1 に移る．

- STEP5_F7

  STEP5_F7 は Step 7 から Step 5 に来たことを示し，Step 5 での if 文の条件判定を行う．条件 = 真 (zr_if='1') の場合は BEFORE_STEP3 に移る．条件 = 偽 (zr_if='0') の場合は STEP7_1 に移る．

- M_COPY_5

  M_COPY_5 では Step 5 での $M := M_d$ に当たる処理を行う．M_d のコピーは全行について並列に行われるため 1 クロックで済む．

- BEFORE_STEP3

  Step 5 から Step 3 に来た場合，Step 2 から来た場合とプライオリティエンコーダへの入力が違う．BEFORE_STEP3 ではプライオリティエンコーダへの入力を変更するための信号を出力する．

- STEP6

  STEP6 では d をインクリメントし，1 つ下の深さを探索する．同時に F, H の修正も行う．

- STEP7_1

  STEP7_1 では $d = 0$ かどうかチェックする．$d = 0$(zr_d='0') の時は TERMINATE に移り，アルゴリズムを停止する．$d \neq 0$(zr_d='1') の時は STEP7_2 に移る．

- STEP7_2

  STEP7_2 は d の値をデクリメントする．他の値の修正は d がデクリメントした後に行う必要があるため，あらかじめ d をデクリメントする必要がある．

- STEP7_3

  STEP7_3 は H から上の深さでの k の値を取り出し，k の値を更新する．

- M_COPY_7

  M_COPY_7 では Step 7 での $M := M_d$ に当たる処理を行う．M_d のコピーは全行について並列に行われるため 1 クロックで済む．また，Step 7 での F の修正も同時に行う．

- TERMINATE

  TERMINATE ではアルゴリズムの終了信号を出力し，アルゴリズムを停止する．

4.1.2 Refinement Procedure

Ullmann の提案 [3] に従い，Refinement Procedure を組合せ回路で実装する．以下，組合せ回路による実装方法を comb と呼ぶ．Refinement Procedure を前節の通りに実装すると図 7に示す回路 (以下，sub_comb と呼ぶ) を $p_\alpha p_\beta$ 個並べることで実装することが可能である．sub_comb のハード ウェア量は $O(p_\alpha p_\beta)$ である．

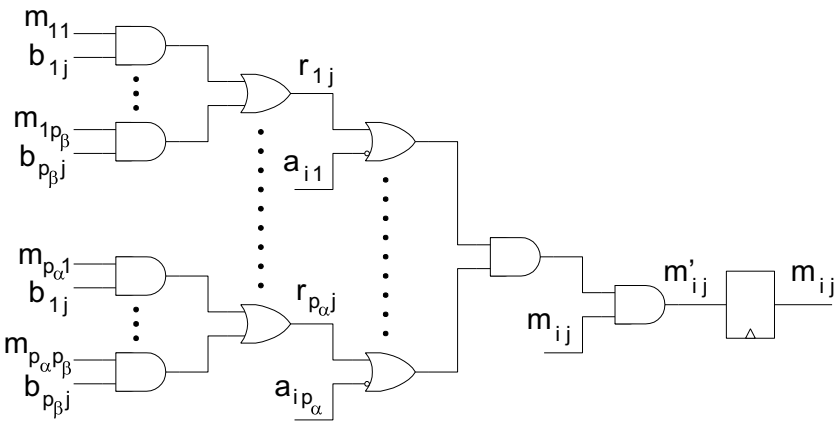


図 7: Refinement Procedure の部分回路

Refinement Procedure の全体図は図 8のようになる．A，B，M の全ての値を同時に読み出し，組合せ回路により M の修正を全ての要素について同時に行う．FAIL exit，SUCCEED exit のための判定用信号も組合せ回路により実装し，M の修正と同時に判定用信号を生成させる．

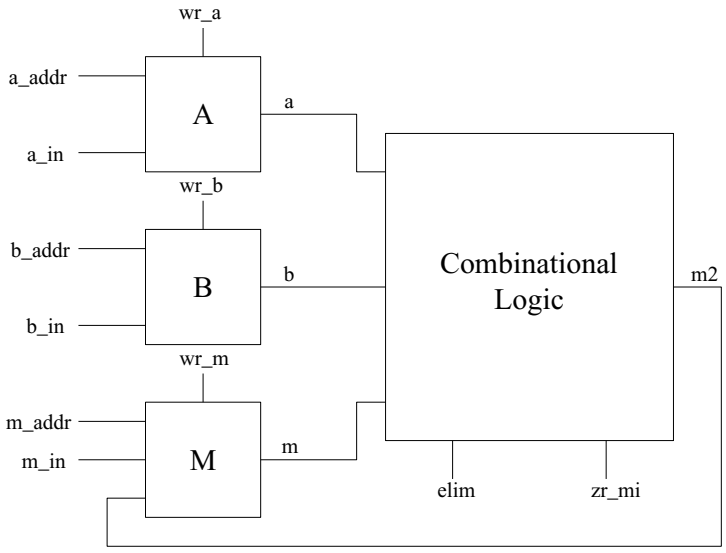


図 8: Refinement Procedure の概念図 (全体)

図 9に comb の制御ユニットの状態遷移図を示す．start_ref，zr_mi，elim は状態遷移のために必要な信号である．start_ref は Refinement Procedure の開始信号で，start_ref='1' になると Refinement Procedure

を開始する．zr_mi は M のいずれかの行が 0 になっているかどうかを示す信号で，zr_mi='0' の時にいずれかの行が 0 になっていることを示す．elim は M が Refinement Procedure によって修正されたかどうかを示す信号で，elim='1' の時に M の修正があったことを示す．各状態の動作を以下に示す．

- INIT

  INIT は初期状態である．start_ref='0' である間は INIT のままである．start_ref='1' になると CHECK_STATE に移り，Refinement Procedure の実行を開始する．

- CHECK_STATE

  CHECK_STATE は，zr_mi と elim の値をチェックし，値の組み合わせにより次の状態が決まる．まず zr_mi の値をチェックし，zr_mi='0' なら，elim の値に関係なく FAIL_EXIT に移る．zr_mi='1' かつ elim='1' の間は CHECK_STATE を実行し続ける．zr_mi='1' かつ elim='0' なら M がこれ以上修正されないことを示すので SUCCEED_EXIT に移る．

  CHECK_STATE では M の修正も同時に行う．M が修正されるとすぐに次の世代の M が組合せ回路により計算される．

- FAIL_EXIT

  FAIL exit のことである．この状態になると Refinement Procedure を終了し，INIT に戻る．

- SUCCEED_EXIT

  SUCCEED exit のことである．この状態になると Refinement Procedure を終了し，INIT に戻る．

制御ユニットの動作より，comb の動作は以下のように書ける．

1. M の修正とチェック行う．
2. FAIL exit もしくは SUCCEED exit になると Refinement Procedure を終了する．そうでなければ再び M の修正とチェックを行い，これを繰り返す．

なお，信号 (start_ref，zr_mi，elim)，状態 (INIT，FAIL_EXIT，SUCCEED_EXIT) は Refinement Procedure の実装案に関係なく同じであるため，今後示す実装案では説明を省略する．

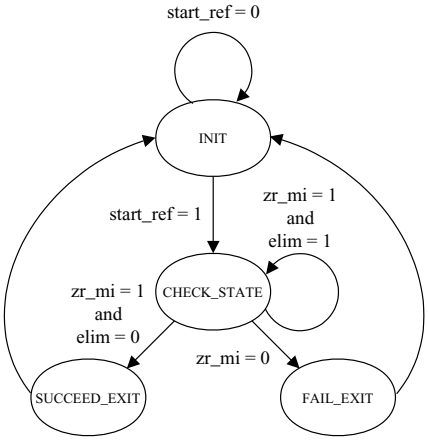


図 9: 制御ユニット状態遷移図 (comb)

A，B，M の全ての値を同時に読み書きするためには，レジスタもしくはフリップフロップを並べて実装しなければならない．そのため A, B, M のハードウェア使用量はそれぞれ $O(p_\alpha^2)$, $O(p_\beta^2)$, $O(p_\alpha p_\beta)$ となる．

図 10に A，B，M の詳細図を示す．Refinement Procedure を行っている最中は全要素を同時に読み出せばいいのだが，(1) 探索木巡回アルゴリズムで 1 行ずつ書き込みを行う時 (M)，(2) アルゴリズムを開始させる前に初期値を入力する時 (A, B, M) の 2 つの場合は，アドレスを指定して 1 行ずつ読み書きするため，レジスタの他に周辺回路が必要になる．具体的には書き込む行に対する書き込み信号をアドレスを入力とするデマルチプレクサにより決定する．M に関しては全ての行について同時に更新する必要があるため，全ての行に同じ信号 (wr_m_ref) を入力し，デマルチプレクサの出力とマルチプレクサで切り替えるようにする．

また，M をレジスタで実装するのは Refinement Procedure で使うためだけではなく，探索木巡回アルゴリズム中で M_d に対する入出力を全行について同時に行うためである．

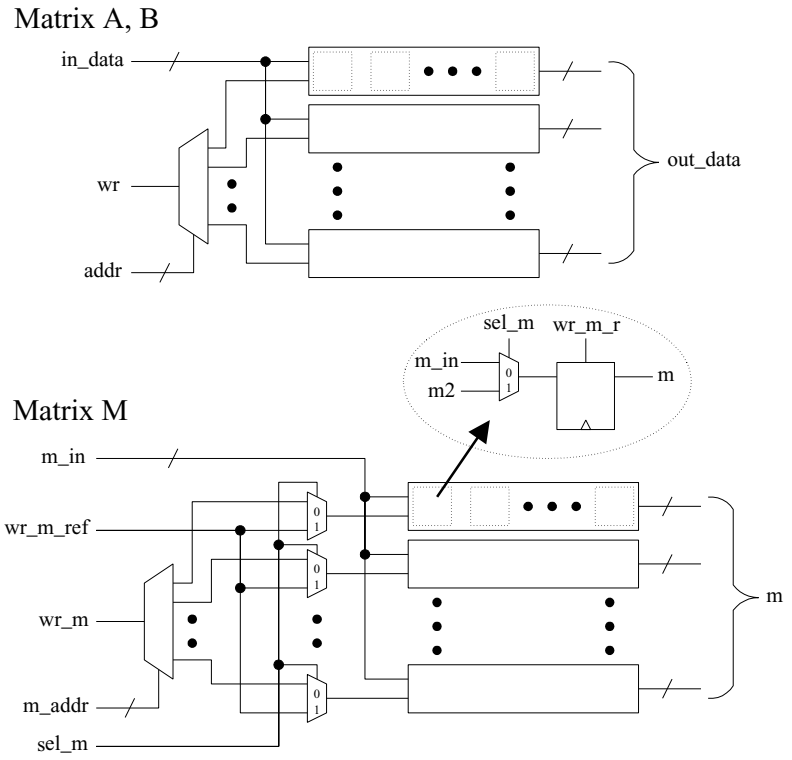


図 10: メモリ A，B，M の概念図

Refinement Procedure を組合せ回路で実装すると，Refinement Procedure の部分だけでハードウェア量が $O(p_\alpha p_\beta^2)$ になることがわかる．ここでハードウェア量が 4 乗オーダーにならないのは，r_{xj} を求める部分が $i(1 \leq i \leq p_\alpha)$ について共通なためである．

組合せ回路による実装は Refinement Procedure による M の修正を全要素について並列に実行させるため，論理ゲートを大量に使用する．そこで，Refinement Procedure を全て並列で行うのではなく，順序回路により処理の一部を逐次処理することにより，実行時間が長くなる代わりにハードウェアの使用量を抑えることができる．次節では本研究で提案した Refinement Procedure を順序回路により実装する方法を述べる．

4.2 順序回路による実装方法

この節では Ullmann の提案手法を基に導き出した順序回路による Refinement Procedure の実装方法について説明する．なお，探索木巡回アルゴリズムは comb と共通であるため，Refinement Procedure についてのみ説明を行う．

sub_comb の個数を限定したもの

最初に示す実装案は，comb で使用した図 7の sub_comb を利用するものである．comb では sub_comb を $p_\alpha p_\beta$ 個並べていたのに対し，これから示す 3 つの実装案では sub_comb を使用する個数を減らし，その分逐次処理を行うことによりハードウェア量を減らすことを目的とする．

4.2.1 実装案 seq_i

組合せ回路部・逐次処理

実装案 seq_i では sub_comb を p_β 個並べ，m_{ij} を j について並列に修正，すなわち M を行単位で修正する．全ての行について修正するためには i をループ変数として p_α 回繰り返し実行を行う．seq_i の組合せ回路部のハードウェア使用量は $O(p_\alpha p_\beta^2)$ となる．

終了判定信号

Refinement Procedure の FAIL exit の検出は各行を修正するたびに行い，FAIL exit になった時点で Refinement Procedure の実行を止める．elim の値のチェックは p_α 回の繰り返しが終わった後に行う．なお，elim は 1 以上になれば同じなので各行の elim 用の出力信号について OR 演算を行い，elim のチェックはその値が'0' か'1' かどうかを見ればよい．

マルチプレクサ

seq_i では，A の i 行目, M の i 行目を取り出すマルチプレクサが必要となる．

制御ユニット

図 11に seq_i の制御ユニットの状態遷移図を示す．start_ref，zr_mi，elim，max_i は状態遷移のために必要な信号である．max_i は i に関するループが p_α 回実行されたかどうかを判定する信号で，max_i='0' の時に $i = p_\alpha$ になったことを示す．各状態の動作を以下に示す．

- RESET_REF_I

  i，elim の値をリセットする．

- CHECK_MI

  M の i 行目の値をチェックし，0 になったら FAIL_EXIT に移る．そうでなかったら CHECK_I に移り i の値をチェックする．M のチェックと同時に M の修正と i のインクリメントも行う．

- CHECK_I

  i の値をチェックし，p_α と等しくなっていたら CHECK_ELIM に移る．そうでなかったら CHECK_MI に移り，次の行について M のチェック，修正を行う．

- CHECK_ELIM

  elim の値をチェックする．elim='0' なら SUCCEED_EXIT に移る．そうでなかったら RESET_REF_I に移り，ループを再実行する．

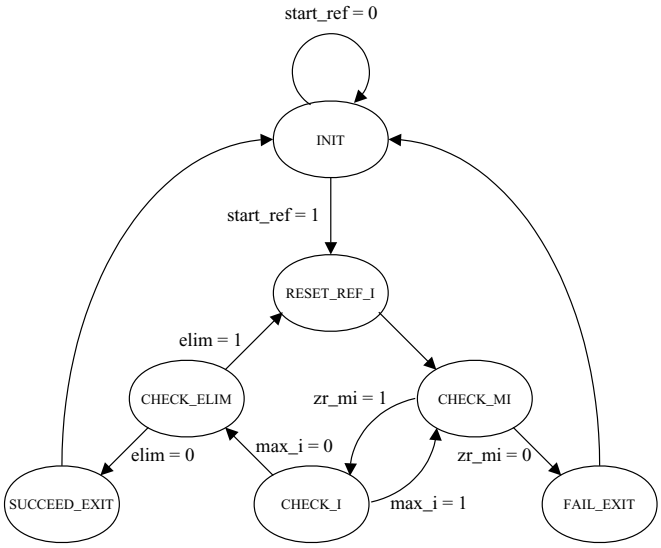


図 11: 制御ユニット状態遷移図 (seq_i)

4.2.2 実装案 seq_i_j

組合せ回路部・逐次処理

実装案 seq_i_j は sub_comb を 1 つだけ使用し，m_{ij} を i, j について更新，すなわち M を要素単位で修正する．実際には j の繰り返しでは修正後の M の i 行目の値をレジスタに格納しておき，i 行目全てが修正された時点で行単位で M を更新するようになっている．全ての要素について修正するためにはループ変数 i，j を用いて $p_\alpha p_\beta$ 回繰り返し実行を行う．seq_i_j の組合せ回路部のハードウェア使用量は $O(p_\alpha p_\beta)$ となる．しかし，全体のオーダーはメモリ B の回路規模が効いてくるために $O(p_\beta^2)$ となる．

終了判定信号

FAIL exit の検出は各 j について p_β 回実行させた後に行い，FAIL exit になった時点で Refinement Procedure の実行を止める．この時，m_{ij} の値の AND 演算を行い，その値を保存しておくことで p_β 回のループが終わった後に zr_mi の値が求まるようにする．elim の値のチェックは $p_\alpha p_\beta$ 回の繰り返しが終わった後に行う．seq_i_j も seq_i と同様に elim 用の出力信号の OR 演算を行い，その出力を見ればよい．

マルチプレクサ

seq_i_j では，A の i 行目，B の j 行目，M の i 行目を取り出すマルチプレクサが必要となる．また，M の i 行目から j 個めの要素を取り出すマルチプレクサも必要になる．

制御ユニット

図 12に seq_i_j の制御ユニットの状態遷移図を示す．start_ref，zr_mi，elim，max_i，max_j は状態遷移のために必要な信号である．max_i は seq_i と同じ働きをする．max_j は j に関するループが p_β 回実行されたかどうかを判定する信号で，max_j='0' の時に $j = p_\beta$ になったことを示す．各状態の動作を以下に示す．状態 (RESET_REF_I，CHECK_ELIM) は seq_i と同じ動作を行うため，説明を省略する．

- RESET_REF_I

  i，elim の値をリセットする．

- REST_REF_J

  j，zr_mi の値をリセットする．

- MODIFY_MIJ

  m_{ij} の値を更新し，レジスタに格納する．同時に elim と zr_mi の値の更新，j のインクリメントを行う．

- CHECK_J

  j の値をチェックし，p_β と等しくなっていたら CHECK_MI に移る．そうでなかったら MODIFY_MIJ に移り，次の要素について M の修正を行う．

- CHECK_MI

  M の i 行目の値をチェックし，0 になったら FAIL_EXIT に移る．そうでなかったら CHECK_I に移り i の値をチェックする．M のチェックと同時に M の修正と i のインクリメントも行う．

- CHECK_I

  i の値をチェックし，p_α と等しくなっていたら CHECK_ELIM に移る．そうでなかったら RESET_REF_J に移り，次の行について M の修正を行う．

4.2.3 実装案 seq_j

組合せ回路部・逐次処理

実装案 seq_j では sub_comb を p_α 個並べ，m_{ij} を i について並列に修正，すなわち M を列単位で修正する．全ての列について修正するためには j をループ変数として p_β 回繰り返し実行を行う．seq_j の組合せ回路部のハードウェア使用量は $O(p_\alpha p_\beta)$ となる．3 乗オーダーにならないのは，図 7の r_{xj} 以下は i について共通で，r_{xj} OR $\bar{a}_{ix}$ 以上の部分も $O(p_\alpha^2)$ で済むからである．

終了判定信号

FAIL exit の検出は全ての列を修正した後に行う．列を修正する際に各行の値をフリップフロップに記憶しておき，列を修正するたびにフリップフロップの値との OR 演算を行う．M のいずれかの行が 0 になる場合，全ての列を修正した時点でフリップフロップの出力が'0' になるため，全てのフリップフロップの出力をチェックし，いずれかの出力が'0' になれば FAIL exit するようにすればよい．

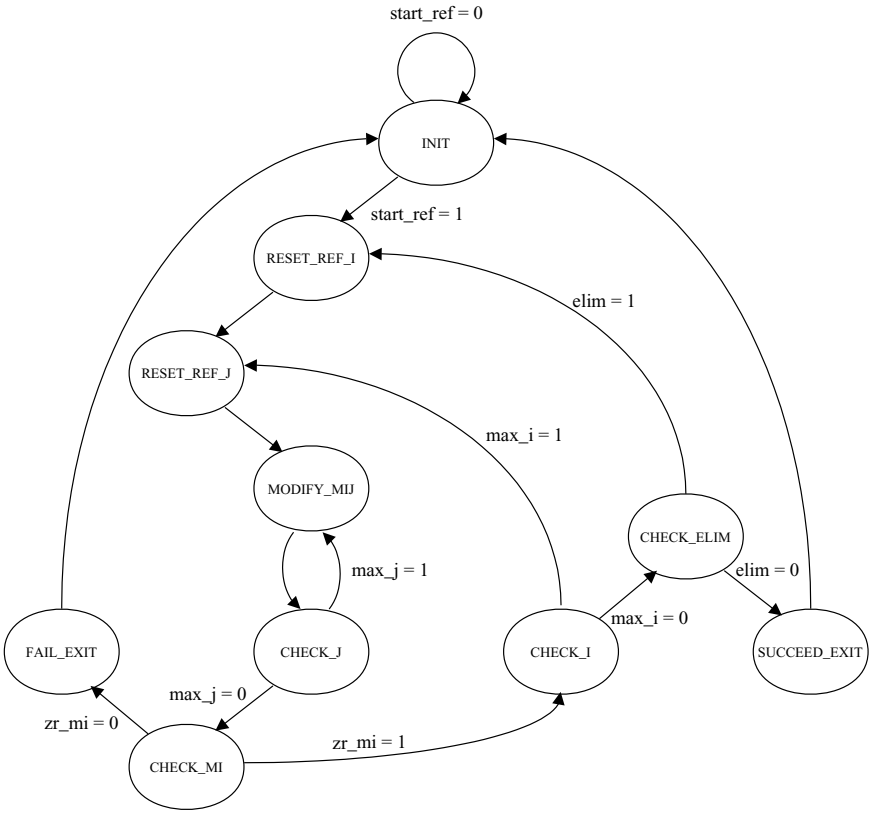


図 12: 制御ユニット状態遷移図 (seq_i_j)

マルチプレクサ

seq_j では，B の j 行目，M の j 列目を取り出すためのマルチプレクサが必要になる．また，各 m_{ij} に対応するフリップフロップについて，Write Enable 信号を選択するためのマルチプレクサが必要になる．

制御ユニット

図 13に seq_j の制御ユニットの状態遷移図を示す．start_ref, zr_mi, elim, max_j は状態遷移のために必要な信号である．max_j は seq_i_j と同じ働きをする．各状態の動作を以下に示す．

- RESET_REF_J

  j, elim の値をリセットする．

- MODIFY_M

  M を列単位で修正する．同時に j のインクリメントも行う．

- CHECK_J

  j の値をチェックし，p_β と等しくなっていたら CHECK_STATE に移る．そうでなかったら MODIFY_M に移り，M の次の列の修正を行う．

- CHECK_STATE

CHECK_STATE は，zr_mi と elim の値をチェックし，値の組み合わせにより次の状態が決まる．まず zr_mi の値をチェックし，zr_mi='0' なら，elim の値に関係なく FAIL_EXIT に移る．zr_mi='1' かつ elim='0' なら M がこれ以上修正されないことを示すので SUCCEED_EXIT に移る．zr_mi='1' かつ elim='1' なら RESET_REF_J に移り，ループを再実行する．

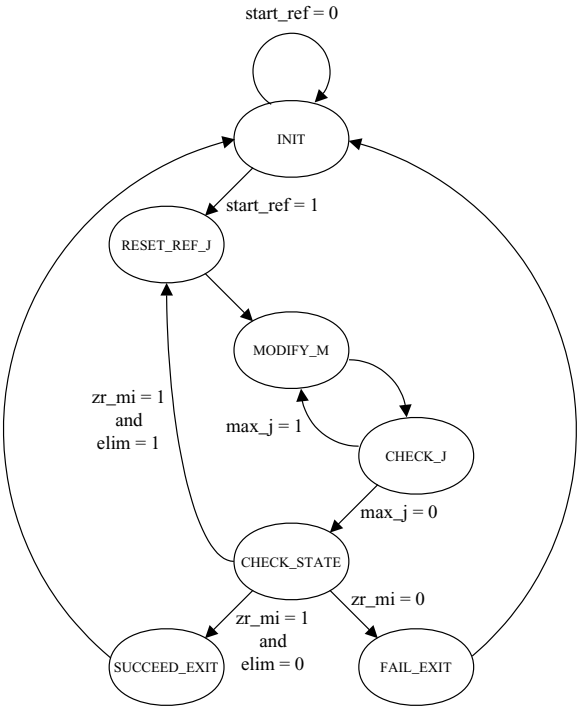


図 13: 制御ユニット状態遷移図 (seq_j)

sub_comb の性質を利用したもの

次に示す実装案は図 14に示す sub_comb 中の破線部に着目する．破線部は p_α 入力 AND ゲートである．n 入力 AND ゲートは 1 つでも'0' が入力された時点で AND 演算の結果が'0' に決定される．これから示す 3 つの実装案はこの性質を利用し，x をループ変数とした順序回路によりハードウェア量を減らすことを目的とする．

4.2.4 実装案 seq_x

組合せ回路部・逐次処理

実装案 seq_x では図 15に示す部分回路 (sub_comb_x と呼ぶ) を $p_\alpha p_\beta$ 回並べることにより，M の全要素を同時に更新する．x をループ変数にし p_α 回繰り返すことにより M の修正を行う．常に AND の結果を M に反映させているために，FAIL exit になる場合は p_α 回繰り返す前に Refinement Procedure を終了できる可能性があるため，seq_i と比べて高速に実行できることが期待できる．seq_x の組合せ回路部のハードウェア量は $O(p_\beta^2)$ となる．3 乗オーダーにならないのは，sub_comb_x のハードウェア量は $O(p_\beta)$ で，sub_comb_x の r_{xj} までの部分は列についてのみ並べれば良いため，$O(p_\alpha p_\beta + p_\beta^2)$ となり，$p_\alpha \leq p_\beta$ より全体のオーダーは $O(p_\beta^2)$ に抑えられるからである．

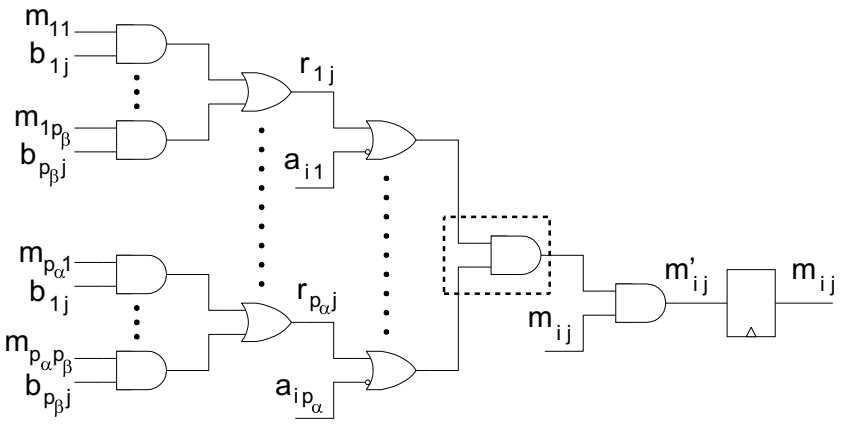


図 14: 利用する sub_comb の性質

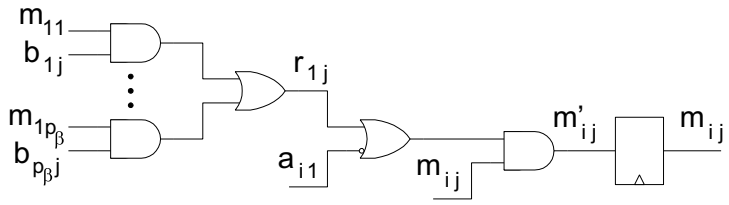


図 15: sub_comb_x

終了判定信号

FAIL exit の検出用回路は comb と同じものになる．elim の値は x 回繰返した後に判定するため，comb と同じ回路で生成された elim の値に対し OR 演算を行い，結果をフリップフロップに保持し，フリップフロップの出力を OR ゲートの入力とする．

マルチプレクサ

seq_x では，A の x 行目，M の x 行目を取り出すためのマルチプレクサが必要になる．

制御ユニット

図 16に seq_x の制御ユニットの状態遷移図を示す．start_ref，zr_mi，elim，max_x は状態遷移のために必要な信号である．max_x は x に関するループが p_α 回実行されたかどうかを判定する信号で，max_x='0' の時に $x = p_\alpha$ になったことを示す．各状態の動作を以下に示す．

- RESET_REF_X

  x，elim の値をリセットする．

- CHECK_M

  M の値をチェックし，M のいずれかの行が 0 になったら FAIL_EXIT に移る．そうでなかったら CHECK_X に移り x の値をチェックする．M のチェックと同時に M の修正と x のインクリメントも行う．

- CHECK_X

  x の値をチェックし，p_α と等しくなっていたら CHECK_ELIM に移る．そうでなかったら CHECK_M に移り，再び M の修正とチェックを行う．

- CHECK_ELIM

  elim の値をチェックする．elim='0' なら SUCCEED_EXIT に移る．そうでなかったら RESET_REF_X に移り，ループを再実行する．

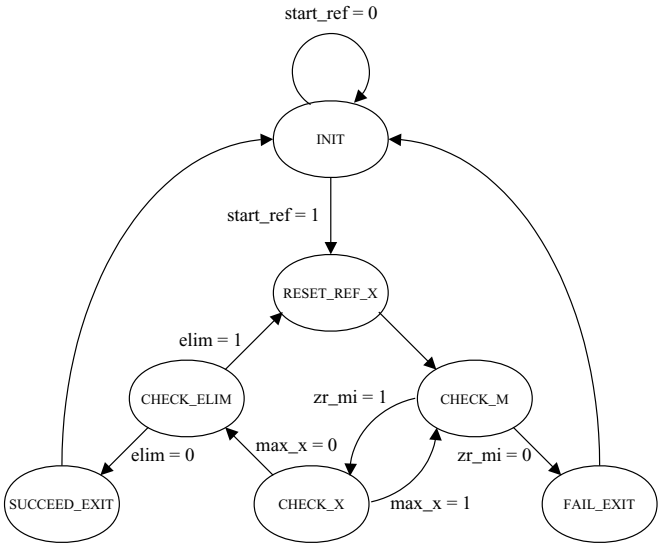


図 16: 制御ユニット状態遷移図 (seq_x)

4.2.5 実装案 seq_i_x

組合せ回路部・逐次処理

実装案 seq_i_x は sub_comb_x を p_β 個並べ，M を行単位で修正する．M の修正は seq_x のように x による繰り返し毎に行い，FAIL exit になる場合は x を p_α 回繰り返す前に Refinement Procedure を終了する．最大繰り返し回数は i による繰り返しと x による繰り返しにより p_α^2 回となる．seq_i_x の組合せ回路部のハードウェア量は $O(p_\beta^2)$ となる．

終了判定信号

seq_i_x については，FAIL exit, elim の検出は seq_i と同じ回路を使用する．

マルチプレクサ

seq_i_x では，A の i 行目，M の x 行目を取り出すためのマルチプレクサが必要となる．また，A の i 行目から x 個めの要素を取り出すマルチプレクサも必要になる．

制御ユニット

図 17に seq_i_x の制御ユニットの状態遷移図を示す．start_ref，zr_mi，elim，max_i，max_x は状態遷移のために必要な信号である．max_i は seq_row_i と同じ働きをする．max_x は seq_all_x と同じ働きをする．各状態の動作を以下に示す．状態 (RESET_REF_I，CHECK_ELIM) は seq_i と，状態 (RESET_REF_X) は seq_x と同じ動作を行うため，説明を省略する．

- RESET_REF_X

  x の値をリセットする．

- CHECK_MI

  M の i 行目のチェックを行い，0 になったら FAIL_EXIT に移る．そうでなかったら CHECK_X に移り x の値をチェックする．M のチェックと同時に M の修正と i のインクリメントも行う．

- CHECK_X

  x の値をチェックし，p_α と等しくなっていたら CHECK_I に移り，i のチェックを行う．そうでなかったら CHECK_MI に移り，再び M の修正とチェックを行う．

- CHECK_I

  i の値をチェックし，$p_\alpha - 1$ と等しくなっていたら CHECK_ELIM に移る．そうでなかったら RESET_REF_X に移り，x のループを再実行する．また，seq_i_x は i のチェックと i のインクリメントを同時に行う．

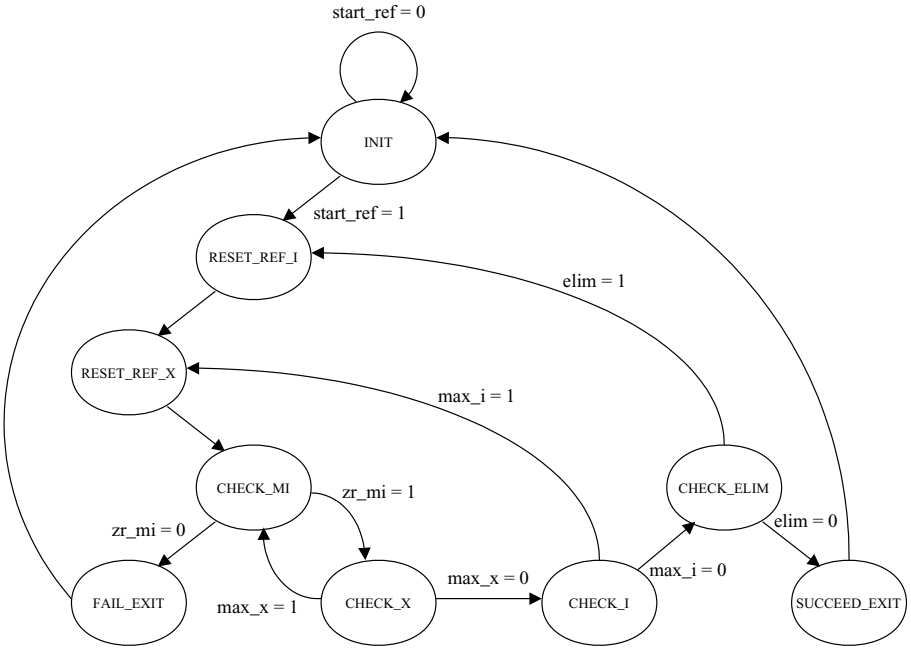


図 17: 制御ユニット状態遷移図 (seq_i_x)

4.2.6 実装案 seq_j_x

組合せ回路部・逐次処理

実装案 seq_j_x は sub_comb_x を p_α 個並べ，M を列単位で修正する．M の修正は seq_x のように x による繰り返し毎に行うが，FAIL exit は行について判定するため，seq_j と同様に全ての列を判定した後に行う．よって，FAIL exit による実行時間短縮は行われない．繰り返し回数は j による繰り返しと x による繰り返しにより $p_\alpha p_\beta$ 回となる．seq_j_x の組合せ回路部のハードウェア量は $O(p_\beta^2)$ となる．しかし，全体のオーダーはメモリ B の回路規模が効いてくるために $O(p_\beta^2)$ になる．

終了判定信号

seq_j_x については，FAIL exit, elim の検出は seq_j と同じ回路を使用する．

マルチプレクサ

seq_j_x では，A の x 行目, B の j 行目，M の x 行目を取り出すためのマルチプレクサが必要になる．また，各 m_{ij} の対応するフリップフロップについて，Write Enable 信号を選択するためのマルチプレクサが必要になる．

制御ユニット

図 18に seq_j_x の制御ユニットの状態遷移図を示す．start_ref, zr_mi, elim, max_j, max_x は状態遷移のために必要な信号である．max_j は seq_j と，max_x は seq_x と同じ働きをする．各状態の動作を以下に示す．状態 (RESET_REF_J, CHECK_STATE) は seq_j と，状態 (RESET_REF_X) は seq_x と同じ働きをするため，説明を省略する．

- MODIFY_M

  M を列単位で修正する．同時に x のインクリメントも行う．

- CHECK_X

  x の値をチェックし，p_α と等しくなっていたら CHECK_J に移り，j のチェックを行う．そうでなかったら MODIFY_M に移り，再び M の修正を行う．

- CHECK_J

  j の値をチェックし，$p_\beta - 1$ と等しくなっていたら CHECK_STATE に移る．そうでなかったら RESET_REF_X に移り，x のループを再実行する．また，seq_j_x は j のチェックと j のインクリメントを同時に行う．

5 実装

この節では Ullmann のアルゴリズムを実装する方法を述べる．本研究では Lucent 社 OR2CxxA FPGA への実装を想定しているため，OR2CxxA FPGA のアーキテクチャを最大限に利用した実装を行う．

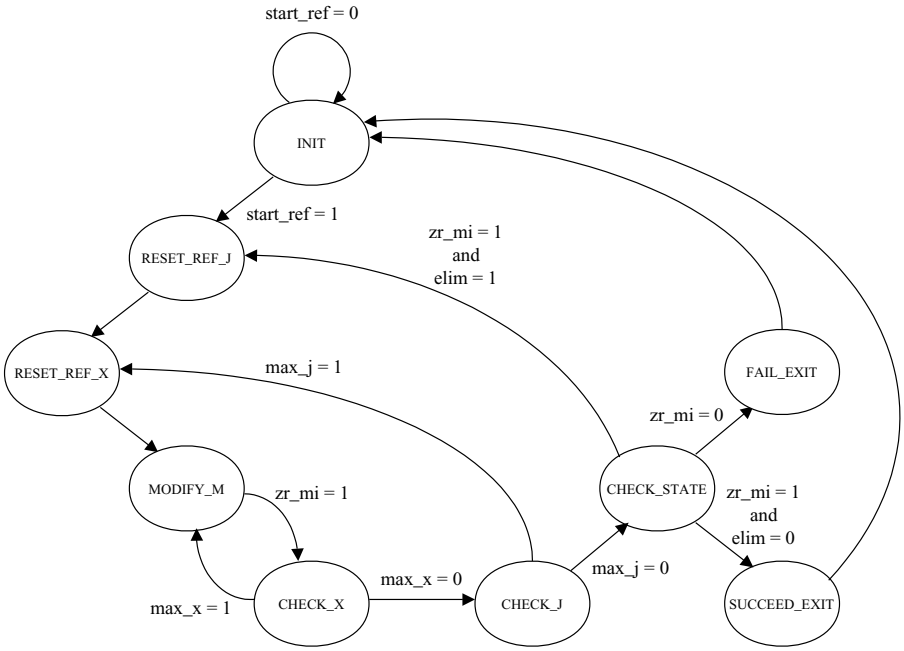


図 18: 制御ユニット状態遷移図 (seq_j_x)

5.1 OR2C シリーズ FPGA

ここでは OR2C シリーズ FPGA のアーキテクチャについて説明する．OR2C シリーズ FPGA[4] は PIC(Programmable Input/Output Cell) と PLC(Programmable Logic Cell) の 2 つのセル要素から構成され，PLC の周りに PIC が配置されている．

PLC は PFU(Programmable Fuction Unit) と配線リソースから構成されている．PFU は論理機能を実現させるために使用され，様々な機能を持つ．PFU は OR2C シリーズ FPGA のリソース基本単位として使用される．1 つの PFU 内には 4 つの 64 ビット LUT(Look Up Table) と 4 つのラッチ/フリップフロップ (FF) を含み，LUT 内で組合せ論理を，ラッチ/FF 内で順序論理を実行する．LUT は SRAM(Static RAM) だが，RAM/ROM としても利用可能である．また，各 PLC 内には 8 つの 3 ステート・バッファがあり，これらは PFU の外部にある．

LUT は以下の 3 つのモードを持ち，その中の 1 つで動作するようにコンフィギュレーションできる．

- 組合せ論理モード
- リップル・モード
- メモリ・モード

組合せ論理モードでは 64 ビット LUT を使用してブール関数を表現する．組合せ論理は 1PFU について 6 入力による 1 つの機能, 5 入力による 2 つの機能, 4 入力による 4 つの機能のいずれかと，5 入力による 2 つの機能および c0(PFU の外部入力の 1 つ) に基づいた 3 つの特殊機能を実行するために LUT を使用できる．

LUT をリップル・モードで動作させると，4 ビット加算, 減算, 加算/減算, カウンタなどの標準演算機能が実現できる．OR2CxxA FPGA では新たに 2 つの新規モード (4×1 乗算と 4 ビット・コンパレータ (Comparator)) が実現できる．これらの新規モードを使用すると，論理機能が高密度化され，高速化されるという利点がある．

LUT をメモリ・モードで動作させる場合，2 つの 16 × 2 ビット非同期メモリが実現できる．OR2CxxA FPGA ではこれらに加えて 16 × 4 ビット同期シングル・ポート・メモリ (SSPM) と 16 × 2 ビット同期デュアル・ポート・メモリ (SDPM) が実現可能である．

5.2 実装方法

OR2C シリーズ FPGA の実装手順は以下のようになる．

1. 回路記述
2. 論理合成
3. テクノロジ・マッピング
4. 配置配線
5. ビットファイル生成
6. FPGA にダウンロード，実行

本研究は FPGA への実装に適した実装方法について検討するため，FPGA へ実装した場合の回路規模の見積りを行うだけで，配置配線以降の作業は行わない．

5.2.1 回路記述

回路記述はハードウェア記述言語である VHDL により行う．回路記述の後に行う論理合成では，

- Verilog, VHDL のいずれかのハードウェア記述言語による回路記述
- ViewLogic 等による回路図入力による回路記述

のいずれかを行う必要があるのだが，VHDL 言語による回路記述が一般的であり，VHDL 言語自体の文法もわかりやすいため，本研究での回路記述は VHDL 言語を採用した．

VHDL 言語による回路記述は大きく分かれて 2 つの方法がある．1 つは機能記述による方法で，この場合は FPGA 別に用意されているテクノロジ・ライブラリを用いて論理合成をすることにより，機能記述が FPGA に特化したライブラリに変換される．もう 1 つは FPGA に特化したライブラリを直接記述する方法で，論理合成では生成されないものについてはライブラリによる回路記述を行うことで動作速度の向上と回路規模の縮小が期待できる．

5.2.2 論理合成

論理合成は Synopsys 社の論理合成ツールである Design Compiler を使用する．論理合成の際には OR2C シリーズ FPGA 用のテクノロジ・ライブラリを用いて論理合成を行う．論理合成の結果，組合せ論理は LUT に，記憶素子はフリップフロップに，カウンタは適切な機能記述によりカウンタのマクロに変換される．合成した回路は EDIF ネットリストで保存する．

5.2.3 テクノロジ・マッピング

テクノロジ・マッピングは，論理合成により合成した回路 (EDIF ネットリスト) を実際の FPGA に写像することである．テクノロジ・マッピングにより FPGA に実装した場合の PFU の使用量を知ることができる．

5.2.4 FPGA に適した実装

論理ゲート

AND, OR 等の論理ゲートは機能記述を行い，論理合成により最適な論理ゲートの組み合わせを生成させる．本研究で実装する Refinement Procedure で用いる組合せ回路は機能記述により実装する．

また，メモリ，マルチプレクサについては ORCA の Macro Library による回路記述を行う．

メモリ

メモリの実装は 2 通りある．1 つはフリップフロップもしくはレジスタを使用する方法である．この方法は Refinement Procedure で，A, B, M を同時に読み出す場合である．また，探索木巡回アルゴリズムで k, F の値を格納するのにもレジスタを用いる．

2 つ目は 16×4 ビットシングル・ポート・メモリ (RCF16X4) を直接記述することにより，LUT をメモリ・モードで動作させる方法である．この方法は Refinement Procedure で A, B を 1 行ずつ取り出す場合に用いる．また，探索木巡回アルゴリズムで M_d や H を実装するのにもこの方法を用いる．M_d は M の各行について 1 つの RAM を使用する．LUT によりメモリを実装する利点は，フリップフロップで実装する場合に比べて回路規模が小さくて済む．OR2C シリーズ FPGA では 1PFU あたり 4 つのフリップフロップしか実装できないため，フリップフロップによる実装では，1PFU あたり 4 ビットしか保持することができない．それに対し，LUT で実装すると 1PFU あたり 16×4 ビットの値を保持することができる．また，A や B から 1 行取り出すときに用いるマルチプレクサが不要になるため，その分回路規模が縮小できる．メモリが LUT で実装できる部分については積極的にライブラリによる記述を行うことで，回路規模を縮小することができる．表 1に各実装方法の Refinement Procedure 中の A, B, M のメモリの実装方法を示す．ここで，FF はフリップフロップ，LUT は LUT による実装のことを示す．

表 1: Refinement Procedure のメモリの実装方法

実装方法	A	B	M
comb	FF	FF	FF
seq_i	LUT	FF	FF
seq_i_j	LUT	LUT	FF
seq_j	FF	LUT	FF
seq_x	LUT	FF	FF
seq_i_x	LUT	FF	FF
seq_j_x	LUT	LUT	FF

マルチプレクサ

マルチプレクサの実装には 3 ステートバッファを用いる．これは PLC 内にある 8 つの 3 ステート・バッファを利用する．マルチプレクサは Macro Library による記述も可能だが，この場合は論理合成により LUT に変換されてしまうため，膨大な量の PFU を消費してしまうという欠点がある．3 ステート・バッファを用いることにより，リソース消費量は劇的に減少することが期待できる．よって本研究ではマルチプレクサを 3 ステート・バッファを用いて実装する．

6 評価

組合せ回路と提案手法によって実装した回路について，回路規模，実行時間について評価を行う．以下に測定方法，測定結果について述べる．

6.1 測定方法

6.1.1 回路規模

(a) 面積レポート

Synopsys Design Compiler では論理合成した回路について report_area コマンドを実行することにより，面積レポートを出力することができる．本研究では OR2C シリーズ FPGA 用のテクノロジ・ライブラリを使用しており，面積レポートの値は，ライブラリのデータシート中に載っている値と同じものになる．ただし，面積レポートは PFU 数ではなく，あくまでも仮想的な値であるため，主に回路全体に対するユニット別の内訳を調べるために用いる．あらかじめ回路の内訳を知ることにより，回路規模の上でネックになっている部分を見つけることができる．

(b) PFU 使用量

回路規模の評価には PFU の使用量を用いる．PFU の使用量はテクノロジ・マッピングにより求めることができる．テクノロジ・マッピングはオプションによりさまざまな条件を付加することが可能である．表 2に使用したオプションを示す．表 2で “サイズがオーバーしても NCD ファイルを生成する” を “Yes” にするのは，実装方法によっては指定した FPGA に収まらない場合にどの程度オーバーしたのかを知るためである．NCD ファイルはテクノロジ・マッピングにより生成されるファイルで，配置配線をする際に利用する．

表 2: テクノロジ・マッピングのオプション

ターゲット・デバイス	OR2C40A(900PFU)
パッケージ	S208
Speed Grade	4
サイズがオーバーしても NCD ファイルを生成する	Yes

6.1.2 動作周波数

テクノロジ・マッピングを行った後，トレースを実行することにより，ゲート遅延を積算した動作可能周波数を求めることができる．この動作可能周波数は配線遅延が含まれていないもので，PFU 数の上限も無視しているため，実際に実装した場合の動作周波数とは違うが，動作可能周波数の 1 つの上限と考えて良い．本研究ではテクノロジ・マッピングの後のトレースにより得られた動作可能周波数を実装方法の評価に用いる．

6.1.3 実行時間

ハードウェア・シミュレータ

ハードウェアの実行時間は，部分グラフ同型判定の実行クロック数を求め，テクノロジ・マッピングにより求めた動作周波数を用いて実行時間の予測値を求める．実行クロック数を求めるには Synopsys 社の VSS シミュレータを用いて記述した VHDL 記述を用いてシミュレーションを行う方法が挙げられる．しかし，VSS を用いると 1 つの入力グラフに対するシミュレーションの所要時間が $(p_\alpha,\ p_\beta)$=(15, 15) のグラフで数 100 秒かかってしまう．ソフトウェアと同様に 100 パターン分の平均を求めることを考えると，100 パターン分のデータを測定するのに短くて半日，長くて数週間かかることが予想される．

そこで，ハードウェアの動作を再現すると共に実行クロック数を求めるようなハードウェアのシミュレータを C 言語で実装し，実行時間を求めることを考えた．comb の場合で，入力グラフが $(p_\alpha,\ p_\beta)$=(15, 15), $(ed_\alpha,\ ed_\beta)$=(0.2, 0.2) の場合について VSS と C 言語で実装したハードウェア，シミュレータの所要時間を測定した．VSS の場合，実行クロック数が比較的少ないデータについて実行したところ，約 3 分かかった．一方，ハードウェア，シミュレータで同じ入力グラフについて実行してみたところ，1 秒未満だった．正確な時間が測定できないため，同じ $p_\alpha, p_\beta, ed_\alpha, ed_\beta$ について，100 パターン分実行してみたところ，14 秒で終わった．これより 1 パターン分の平均値の概算は 0.14 秒となり，VSS の約 1000 倍早いことがわかる．本研究では，実行クロック数を求めるのに所要時間の短いハードウェア・シミュレータを用いることにする．

実行時間の予測値

各実装方法について Ullmann のアルゴリズムのハードウェア・シミュレータを C 言語で実装し，実行クロック数を求め，動作周波数と共に実行時間の予測値を求めた．実行クロック数を C_{run}，動作可能周波数を F_{run}[Hz] とした場合，実行時間の予測値 $T_{estimate}$[sec] は以下の式で求めることができる．

$$T_{estimate} = \frac{C_{run}}{F_{run}} \tag{8}$$

クロック数の見積り

ハードウェア・シミュレータで用いたクロック数の見積りを以下に示す.

(1) 探索木巡回アルゴリズム

図 19に探索木巡回アルゴリズムのクロック数の見積りを示す.

Step 1.	$M := M^0$; $d := 1$; $H_1 := 0$;
	for all $i := 1, \cdots, p_\beta$ set $F_i := 0$; [ここまでの初期値設定: 1]
	refine M; [1+"Refinement Procedure のクロック数"]
	if (FAIL exit [1]) { terminate algorithm; [1] }
Step 2.	If (there is no value of j such that $m_{dj} = 1$ and $f_j = 0$ [1]) { go to step 7; }
	$M_d := M$; [1]
	$k := 0$;
Step 3.	$k := k + 1$;
	if $m_{dk} = 0$ or $f_k = 1$ then go to step 3; [if 文の条件が満たされなかった場合: 1]
	for all $j \neq k$ set $m_{dj} := 0$; [1]
	refine M; [1+"Refinement Procedure のクロック数"]
	if (FAIL exit [1]) { go to step 5; }
Step 4.	If ($d < p_\alpha$ [1]) { go to step 6; }
	else { output "an isomorphism has been found"; [1] }
Step 5.	If (there is no $j > k$ such that $M_d(d, j) = 1$ and $f_j = 0$ [1]) { go to step 7; }
	$M := M_d$; [Step 3 または Step 4 から Step 5 に来た場合: 1]
	go to step 3; [1]
Step 6.	$H_d := k$; $F_k := 1$; $d := d + 1$; [1]
	go to step 2;
Step 7.	If ($d = 1$ [1]) { terminate algorithm; [1] }
	$d := d - 1$; [1]
	$k := H_d$; [1]
	$F_k := 0$; [1]
	$M := M_d$; [$F_k := 0$ と同時]
	go to step 5;

図 19: 探索木巡回アルゴリズムのクロック数見積り

(2) Refinement Procedure(comb)

図 20に comb により実装した場合の Refinement Procedure のクロック数の見積りを示す.

```
while(1) {
   M を計算; [1]
   if ((∃i)(M_i = 0) [M の計算と同時]) {
      FAIL exit; [1]
   } else if (M が修正された [M の計算と同時]) {
      SUCCEED exit; [1]
   }
}
```

図 20: Refimement Procedure のクロック数見積り (comb)

(3) Refinement Procedure(seq_i)

図 21に seq_i で実装した場合の Refinement Procedure のクロック数の見積りを示す.

```
while(1) {
   i := 0; elim := 0; [1]
   do {
      M_i を計算; [1]
      if (M_i が修正された [M_i の計算と同時]) { elim := 1; }
      if (M_i = 0 [M_i の計算と同時]) { FAIL exit; [1] }
      i++;
   } while (i ≠ p_α [1]);
   if (elim = 0 [1]) { SUCCEED exit; [1] }
}
```

図 21: Refimement Procedure のクロック数見積り (seq_i)

(4) Refinement Procedure(seq_i_j)

図 22に seq_i_j で実装した場合の Refinement Procedure のクロック数の見積りを示す.

```
while(1) {
  i := 0; elim := 0; [1]
  do {
    j := 0; [1]
    do {
      m_ij を計算; [1]
      if (m_ij が修正された [m_ij の計算と同時]) { elim := 1; }
      j++;
    } while (j ≠ p_β [1]);
    M_i を更新; [1]
    if (M_i = 0 [M_i の更新と同時]) { FAIL exit; [1] }
    i++;
  } while (i ≠ p_α [1]);
  if (elim = 0 [1]) { SUCCEED exit; [1] }
}
```

図 22: Refimement Procedure のクロック数見積り (seq_i_j)

(5) Refinement Procedure(seq_j)

図 23に seq_j で実装した場合の Refinement Procedure のクロック数の見積りを示す.

```
while(1) {
  j := 0; elim := 0; [1]
  do {
    M の j 列目を計算; [1]
    if (M が修正された [M の計算と同時]) { elim := 1; }
    j++;
  } while (j ≠ p_β [1]);
  if ((∃i)(M_i = 0) [1]) {
    FAIL exit; [1]
  } else if (elim = 0 [M_i のチェックと同時]) {
    SUCCEED exit; [1]
  }
}
```

図 23: Refimement Procedure のクロック数見積り (seq_j)

(6) Refinement Procedure(seq_x)

図 24に seq_x で実装した場合の Refinement Procedure のクロック数の見積りを示す.

```
while(1) {
   x := 0; elim := 0; [1]
   do {
      M を計算; [1]
      if (M が修正された [M の計算と同時]) { elim := 1; }
      if ((∃i)(M_i = 0) [M の計算と同時]) { FAIL exit; [1] }
      x++;
   } while (x ≠ p_α [1]);
   if (elim = 0 [1]) { SUCCEED exit; [1] }
}
```

図 24: Refimement Procedure のクロック数見積り (seq_x)

(7) Refinement Procedure(seq_i_x)

図 25に seq_i_x で実装した場合の Refinement Procedure のクロック数の見積りを示す.

```
while(1) {
   i := 0; elim := 0; [1]
   do {
      x := 0; [1]
      do {
         M_i を計算; [1]
         if (M_i が修正された [M_i の計算と同時]) { elim := 1; }
         if (M_i = 0 [M_i の計算と同時]) { FAIL exit; [1] }
         x++;
      } while (x ≠ p_α [1]);
      i++;
   } while (i ≠ p_α [1]);
   if (elim = 0 [1]) { SUCCEED exit; [1] }
}
```

図 25: Refimement Procedure のクロック数見積り (seq_i_x)

(8) Refinement Procedure(seq_j_x)

図 26に seq_j_x で実装した場合の Refinement Procedure のクロック数の見積りを示す.

```
while(1) {
  j := 0; elim := 0; [1]
  do {
    x := 0; [1]
    do {
      M の j 列目を計算; [1]
      if (M が修正された [M の計算と同時]) { elim := 1; }
      x++;
    } while (x ≠ p_α [1]);
    j++;
  } while (j ≠ p_β [1]);
  if ((∃i)(M_i = 0) [1]) {
    FAIL exit; [1]
  } else if (elim = 0 [M_i のチェックと同時]) {
    SUCCEED exit; [1]
  }
}
```

図 26: Refimement Procedure のクロック数見積り (seq_j_x)

入力データについて [6]

Ullmann のアルゴリズムの実行時間は，ソフトウェア, ハードウェア共に入力グラフの頂点数，辺数に依存するため，本研究ではグラフ G_α, G_β の頂点数 p_α, p_β，辺密度 ed_α, ed_β を変えながら測定を行う．データの生成は p_α, p_β のそれぞれの組合せについて連結グラフを 100 パターン生成する．そして，それぞれのパターンに対する実行時間の平均値を求めた．

ed_α, ed_β の組合せには以下の 4 通りが考えられる．

- G_α, G_β 共に薄い場合
- G_α が薄く，G_β が濃い場合
- G_α, G_β 共に濃い場合
- G_α が濃く，G_β が薄い場合

それぞれの組合せについて，濃いグラフの時は $ed_\alpha(ed_\beta) = 0.4$ とし，薄いグラフの時は $ed_\alpha(ed_\beta) = 0.2$ とする．すると，$(ed_\alpha,\ ed_\beta)$ の組合せは $(ed_\alpha,\ ed_\beta)$=(0.2, 0.2), (0.2, 0.4), (0.4, 0.4), (0.4, 0.2) の 4 通りに決まる．

測定可能な p_α, p_β の組合せは ed_α, ed_β の値によって決まる．辺密度の定義より，辺密度 ed は以下の式で計算できる．

$$ed = \frac{2q}{p(p-1)} \tag{9}$$

辺数 q の値は頂点数 p に対して $q \geq p-1$ となるため，ed の値は

$$ed_{min} \geq \frac{2}{p} \tag{10}$$

となる．(10) 式より，$ed = 0.4$ の場合の p の最小値は 5，$ed = 0.2$ の場合の p の最小値は 10 になることがわかる．次に部分グラフ同型の定義より，p_α と p_β には以下の関係が成り立つ必要がある．

$$p_\alpha \leq p_\beta \tag{11}$$

また，q_α, q_β については以下の関係が成り立つ必要がある．

$$q_\alpha < q_\beta \tag{12}$$

これらの制約条件により，p_α, p_β の組合せは表 3のようになる．

表 3: p_α, p_β の組合せ

$(ed_\alpha,\ ed_\beta)$	p_α	p_β
(0.2, 0.2)	10	11〜15
	11	12〜15
	12	13〜15
	13	14〜15
	14	15

$(ed_\alpha,\ ed_\beta)$	p_α	p_β
(0.4, 0.4)	5	6〜15
	6	7〜15
	7	8〜15
	8	9〜15
	9	10〜15
	10	11〜15
	11	12〜15
	12	13〜15
	13	14〜15
	14	15

$(ed_\alpha,\ ed_\beta)$	p_α	p_β
(0.2, 0.4)	10	10〜15
	11	11〜15
	12	12〜15
	13	13〜15
	14	14〜15
	15	15
(0.4, 0.2)	5	10〜15
	6	10〜15
	7	10〜15
	8	11〜15
	9	13〜15
	10	14〜15

AT 積

実装方法の評価には回路規模と実行時間を用いる．しかし，回路規模と実行時間は相反する性質で，回路規模が小さくなるほど実行時間が長くなり，回路規模が大きくなるほど実行時間が短くなる．本研究で提案した実装方法では，実行時間，回路規模共に異なるため，実装方法を評価するには明確な評価基準を決める必要がある．

そこで，実装方法の評価には AT 積を用いる．AT 積は回路面積 (Area) と実行時間 (Time) の積で表されるものである．相反する性質について積を求めることで，コストパフォーマンスの指標として表すことができる．よって，AT 積が小さい回路ほどコストパフォーマンスの良い回路であると言える．

また，一般に順序回路の AT 積は組合せ回路よりも悪くなると言われている．それは，順序回路にすることにより，制御ユニットによる実行時間，回路規模のオーバーヘッドが生じ，それにより AT 積が増えてしまうからである．しかし，本研究では Refinement Procedure を順序回路化している．そのため，探索木巡回アルゴリズムの回路規模，実行時間の影響により，全体の AT 積の傾向が変わる可能性がある．

本研究では回路面積に PFU の使用量を，実行時間に p_α, p_β のそれぞれの組合せでの実行時間の総和を用いる．また，前に述べたように実行時間は入力グラフに依存するため，ed_α, ed_β のそれぞれの値について個別に AT 積を求めることにする．

7　測定結果

7.1　面積レポート

本研究では，それぞれの実装方法について，入力グラフのサイズが $(p_\alpha,\ p_\beta)$=(7, 7), (15, 15) まで扱える回路を設計した．Refinement Procedure と回路全体について report_area コマンドを実行し面積を求めた後，それらの差を取ることで探索木巡回アルゴリズムの部分の面積を求めた．表 4に (7, 7) の回路，表 5に (15, 15) の回路の面積レポートの結果を示す．

表 4: $(p_\alpha,\ p_\beta)$=(7, 7) の回路面積

ユニット	comb[3]	seq_i	seq_i_j	seq_j	seq_x	seq_i_x	seq_j_x
ハードウェア量	$O(p_\alpha p_\beta^2)$	$O(p_\alpha p_\beta^2)$	$O(p_\alpha p_\beta)$	$O(p_\alpha p_\beta)$	$O(p_\beta^2)$	$O(p_\beta^2)$	$O(p_\beta)$
全体	363.33	337.18	304.88	394.42	314.70	324.80	403.53
探索木	194.70	194.71	194.70	195.62	194.70	194.70	195.66
Refinement	168.63	142.47	110.18	198.80	120.00	130.10	207.87

表 5: $(p_\alpha,\ p_\beta)$=(15, 15) の回路面積

ユニット	comb[3]	seq_i	seq_i_j	seq_j	seq_x	seq_i_x	seq_j_x
ハードウェア量	$O(p_\alpha p_\beta^2)$	$O(p_\alpha p_\beta^2)$	$O(p_\alpha p_\beta)$	$O(p_\alpha p_\beta)$	$O(p_\beta^2)$	$O(p_\beta^2)$	$O(p_\beta)$
全体	1771.05	1402.15	1000.55	1450.48	1086.20	1109.78	1463.58
探索木	597.83	597.82	597.83	598.75	597.83	597.83	598.75
Refinement	1173.22	804.33	402.72	851.73	488.37	511.95	864.83

(7, 7) の結果を見ると全体の面積について差がほとんどない．これは，3 ステートバッファ(TIBUF) の面積 (0.37) が加算されているからである．Refinement Procedure について見ると，(7, 7) の回路は論理ゲートの量がそれほど多くないのに加え，LUT の面積が 4 入力のもので 1 個あたり 0.17 であるため，それほど大きくなっていない．それに対してマルチプレクサを多く使用している探索木巡回アルゴリズム部の面積は Refinement Procedure より大きくなっていることから，面積レポートの値については 3 ステートバッファによる影響が大きいと言える．

(15, 15) の結果は実装方法によってある程度面積に差が出てきている．これは，Refinement Procedure 内の論理ゲートの占める面積が多くなるためで，Refinement Procedure の面積を見ると，実装方法によって差が出てきていることがわかる．探索木巡回アルゴリズム部は (7, 7) の場合と同様にマルチプレクサにより面積が多くなっている．

通常，面積レポートにより回路の内訳の概算ができるのだが，今回のように 3 ステートバッファを用いた場合は 3 ステートバッファの占める割合が多くなってしまうために，実際に実装した場合の内訳とは大幅に異なることが予想できる．

7.2　PFU 使用量，動作周波数

(7, 7), (15, 15) の回路について，表 2に示すマッピング・オプションを指定して，テクノロジ・マッピングを行い，PFU の使用量と動作周波数を求めた．図 6に (7, 7) の回路の，図 7に (15, 15) の回路の結果を

示す.

表 6: マッピング，トレース結果 ($(p_\alpha,\ p_\beta)$=(7, 7))

実装方法	PFU	Freqency[MHz]
comb[3]	387	32.66
seq_i	289	34.20
seq_i_j	202	34.20
seq_j	218	34.20
seq_x	218	31.04
seq_i_x	203	34.20
seq_j_x	185	34.20

表 7: マッピング，トレース結果 ($(p_\alpha,\ p_\beta)$=(15, 15))

実装方法	PFU	Freqency[MHz]
comb[3]	2754	22.47
seq_i	1770	27.62
seq_i_j	467	34.22
seq_j	583	27.94
seq_x	671	23.08
seq_i_x	529	33.98
seq_j_x	387	33.98

comb, seq_x のように，ユニットを $p_\alpha p_\beta$ 個並べたものは動作周波数が低くなっている．これは，論理ゲートによる遅延によるものであると考えられる．(7, 7) の回路については論理ゲートの個数が少ないために，ほとんどの実装方法では動作周波数が同じになっている．この場合は，探索木巡回アルゴリズム内の遅延が大きくなっているからであると考えられる．一方，(15, 15) の回路は実装方法により動作周波数に差が出ている．これは，論理ゲートの個数が増えたため，Refinement Procedure 内の遅延が増大したからであると考えられる．

PFU 使用量については，report_area の結果と傾向が異なり，実装方法により差が出てきている．これは，5 章に示したように 3 ステートバッファが PFU 数とは関係ないため，report_area で考慮されていた 3 ステートバッファの面積が PFU 数に反映されてないことが原因として挙げられる．本研究では，全ての実装方法についてマルチプレクサを 3 ステートバッファとデコーダを用いて実装しているため，このような差が出た．

また，seq_j と seq_x の回路規模が 1 乗オーダーの繰返し回数にも関わらず小さくなっている．seq_i は Refinement Procedure のオーダーから予想できる通り，他の順序回路に比べて PFU 数が多くなっている．最も回路規模の少ないのは seq_j_x である．seq_j_x は Refinement Procedure の論理ゲートが少ない上に，A, B を RAM で実装したことにより回路規模が縮小している．

なお，PFU 使用量についての詳細は 8 章で説明する．

7.3 実行時間，AT 積

測定した実行時間と PFU 数により AT 積を求めた．AT 積はそれぞれの $(ed_\alpha,\ ed_\beta)$ によって別々に求め，実行時間の合計に対する AT 積も求めた．表 8に $(ed_\alpha,\ ed_\beta)$=(0.2, 0.2), (0.4, 0.4) の場合の実行時間と AT 積を，表 9に $(ed_\alpha,\ ed_\beta)$=(0.2, 0.4), (0.4, 0.2) の場合の実行時間と AT 積を，表 10に実行時間の合計値とそれに対する AT 積を示す.

表 8: 各実装方法の実行時間，AT 積 (1)

$(ed_\alpha,\ ed_\beta)$	(0.2, 0.2)		(0.4, 0.4)	
実装方法	Time[sec]	AT 積	Time[sec]	AT 積
comb[3]	1.78×10^{-2}	4.89×10^{1}	3.89×10^{-2}	1.07×10^{2}
seq_i	5.16×10^{-2}	9.13×10^{1}	1.04×10^{-1}	1.84×10^{2}
seq_i_j	4.73×10^{-1}	2.21×10^{2}	8.88×10^{-1}	4.15×10^{2}
seq_j	6.36×10^{-2}	3.71×10^{1}	1.41×10^{-1}	8.23×10^{1}
seq_x	5.83×10^{-2}	3.91×10^{1}	1.16×10^{-1}	7.78×10^{1}
seq_i_x	3.68×10^{-1}	1.95×10^{2}	6.31×10^{-1}	3.34×10^{2}
seq_j_x	4.91×10^{-1}	1.90×10^{2}	9.37×10^{-1}	3.62×10^{2}

表 9: 各実装方法の実行時間，AT 積 (2)

$(ed_\alpha,\ ed_\beta)$	(0.2, 0.4)		(0.4, 0.2)	
実装方法	Time[sec]	AT 積	Time[sec]	AT 積
comb[3]	1.86×10^{1}	5.14×10^{4}	5.93×10^{-4}	1.63×10^{0}
seq_i	5.99×10^{1}	1.06×10^{5}	1.20×10^{-3}	2.12×10^{0}
seq_i_j	5.75×10^{2}	2.68×10^{5}	8.60×10^{-3}	4.02×10^{0}
seq_j	6.59×10^{1}	3.84×10^{4}	2.11×10^{-3}	1.23×10^{0}
seq_x	6.85×10^{1}	4.59×10^{4}	1.36×10^{-3}	9.11×10^{-1}
seq_i_x	5.12×10^{2}	2.71×10^{5}	4.35×10^{-3}	2.30×10^{0}
seq_j_x	5.85×10^{2}	2.26×10^{5}	9.67×10^{-3}	3.74×10^{0}

表 10: 各実装方法の実行時間，AT 積 (3)

$(ed_\alpha,\ ed_\beta)$	all	
実装方法	Time[sec]	AT 積
comb[3]	1.87×10^{1}	5.15×10^{4}
seq_i	6.00×10^{1}	1.06×10^{5}
seq_i_j	5.76×10^{2}	2.69×10^{5}
seq_j	6.61×10^{1}	3.85×10^{4}
seq_x	6.86×10^{1}	4.61×10^{4}
seq_i_x	5.13×10^{2}	2.71×10^{5}
seq_j_x	5.86×10^{2}	2.27×10^{5}

seq_j と seq_x については，$(ed_\alpha,\ ed_\beta)$ の値により AT 積の傾向が変わっている．FAIL exit の効果がそれほどないと思われる (0.2, 0.2) と (0.2, 0.4) では seq_j の方が AT 積が良く，FAIL exit の効果がある程度出ると思われる (0.4, 0.4) と (0.4, 0.2) は seq_x の方が AT 積が良くなっている．seq_j と seq_j_x の AT 積が良くなるのは，PFU 使用量が少ないためであると言える．seq_x については，動作周波数が seq_j に比べて低いことも AT 積が悪くなる原因として挙げられる．

その他については，$(ed_\alpha,\ ed_\beta)$ が変わっても AT 積の傾向が変わらないことがわかる．

7.4 回路規模と実行時間の関係

次に，回路規模と実行時間の傾向をわかりやすくさせるために，横軸を回路規模，縦軸を実行時間として，グラフ上のそれぞれの実装方法に対応する位置にプロットした．図 27に $(ed_\alpha,\ ed_\beta)$=(0.2, 0.2) の，図 28に (0.4, 0.4) の，図 29に (0.2, 0.4) の，図 30に (0.4, 0.2) の，図 31に実行時間の合計に関する回路規模と実行時間の関係を示す．

それぞれの図上に引いてある直線は，組合せ回路 (comb) の AT 積を基準とした AT 積=一定の直線である．この直線より上にあるものは comb の実行時間が比較対象と同じだった場合の回路規模が比較対象より小さいことを示し，言い換えると比較対象の AT 積が comb より悪いことを示す．直線より下にあるものは comb の実行時間が比較対象と同じだった場合の回路規模が比較対照より大きいことを示し，言い換えると比較対象の AT 積が comb より良いことを示す．これらの図からも，seq_j と seq_x の AT 積が comb より良いことが言える．他の実装方法の AT 積についても，この図を見ることで傾向を容易に知ることができる．

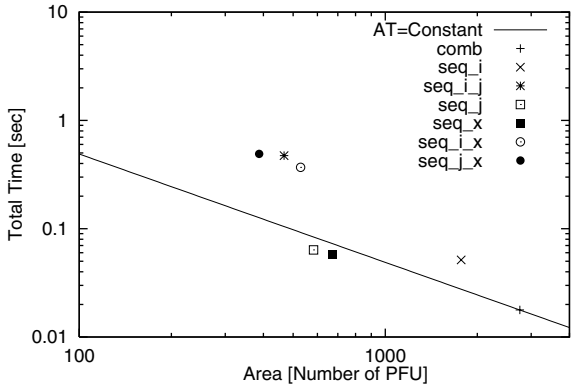


図 27: 各実装方法の実行時間，面積 ($ed_\alpha = 20$, $ed_\beta = 20$)

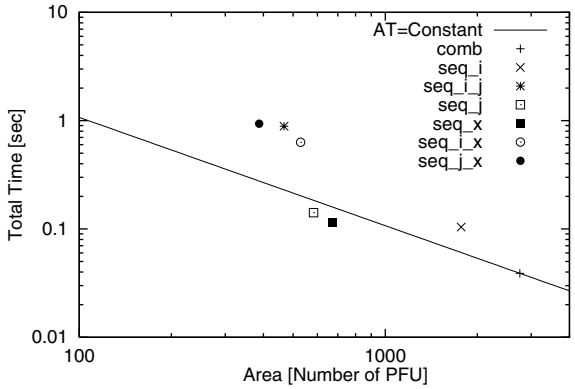


図 28: 各実装方法の実行時間，面積 ($ed_\alpha = 40$, $ed_\beta = 40$)

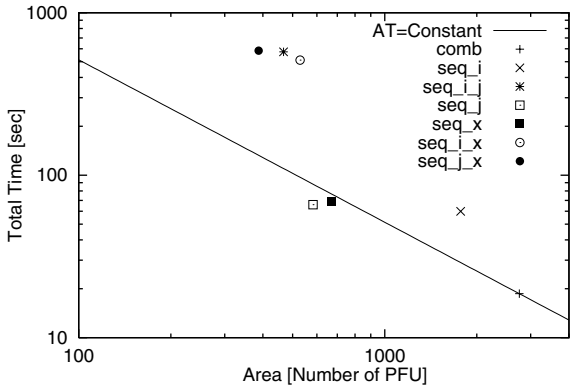


図 29: 各実装方法の実行時間，面積 ($ed_\alpha = 20$, $ed_\beta = 40$)

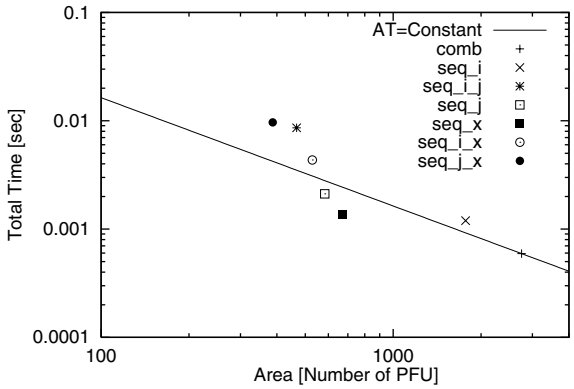


図 30: 各実装方法の実行時間，面積 ($ed_\alpha = 40$, $ed_\beta = 20$)

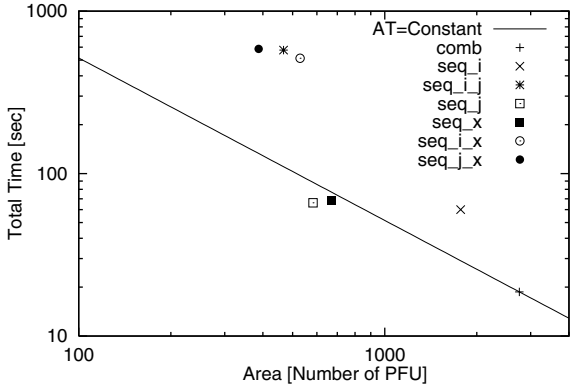


図 31: 各実装方法の実行時間，面積 (合計)

8 分析

本研究で提案した順序回路での実装方法の一部は，組合せ回路よりも良い AT 積を得ることができた．これは順序回路の AT 積が組合せ回路より悪くなるという予想に反する．そこで本節では，回路規模，実行時間の分析を行い，測定結果の裏づけを行う．

8.1 実行時間

8.1.1 FAIL exit の効果

順序回路による実装では，繰返しの途中で FAIL exit することにより実行時間が短縮することが期待される．FAIL exit の効果がある実装方法は seq_i, seq_i_j, seq_x, seq_i_x である．しかし，実際にはどの程度短縮するかどうかはっきり示していないため，FAIL exit しない場合についてデータを測定し，FAIL exit の効果を調べた．FAIL exit の効果は実行クロック数に影響し，動作周波数には無関係なため，実行クロック数の傾向を測定した．表 11〜13に (ed_α, ed_β)=(0.2, 0.2), (0.4, 0.4), (0.4, 0.2) についての FAIL exit する場合の実行クロック数, FAIL exit しない場合の実行クロック数, FAIL exit による実行クロック数の減少率を示す.

表 11: FAIL exit による実行時間減少, (0.2, 0.2)

実装方法	FAIL なし	FAIL あり	減少率 [%]
seq_i	1.43×10^6	1.42×10^6	0.17
seq_i_j	1.62×10^7	1.62×10^7	0.17
seq_x	1.35×10^6	1.35×10^6	0.32
seq_i_x	1.25×10^7	1.25×10^7	0.22

表 12: FAIL exit による実行時間減少, (0.4, 0.4)

実装方法	FAIL なし	FAIL あり	減少率 [%]
seq_i	2.89×10^6	2.87×10^6	0.75
seq_i_j	3.06×10^7	3.04×10^7	0.78
seq_x	2.70×10^6	2.68×10^6	1.19
seq_i_x	2.17×10^7	2.14×10^7	1.01

表 13: FAIL exit による実行時間減少, (0.4, 0.2)

実装方法	FAIL なし	FAIL あり	減少率 [%]
seq_i	3.34×10^4	3.30×10^4	1.09
seq_i_j	2.98×10^5	2.94×10^5	0.17
seq_x	3.18×10^4	3.13×10^4	0.32
seq_i_x	1.51×10^5	1.48×10^5	0.22

これらの表により，FAIL exit の効果は多くて 1%で，FAIL exit の効果はほとんど見られないことがわかる．また，x による逐次処理を行った方が FAIL exit の効果があることがわかる．これは，M の全要素

が同時に修正されることにより，修正結果が次の M の修正に反映されやすいからであると考えられる．

8.1.2 Refinement Procedure の実行時間

組合せ回路 (comb) の実行時間が順序回路に比べてそれほど差が出なかった原因として，Refinement Procedure の実行時間が全体の実行時間に対して短いことが挙げられる．組合せ回路で Refinement Procedure を実装すると，Refinement Procedure そのものの実行時間が極端に短くなるために，探索木巡回アルゴリズムの実行時間の影響が全体の実行時間に大きく影響することが予想できる．そこで，それぞれの実装方法について Refinement Procedure のみの実行時間を測定した．表 14〜17に (ed_α, ed_β)=(0.2, 0.2), (0.4, 0.4), (0.4, 0.2), 合計について，全体の実行クロック数，Refinement Procedure の実行クロック数，Refinement Procedure のクロック数が全体に占める割合，comb の Refinement Procedure のクロック数との比を，表 18〜21に (ed_α, ed_β)=(0.2, 0.2), (0.4, 0.4), (0.4, 0.2), 合計について，全体の実行時間，Refinement Procedure の実行時間，Refinement Procedure の実行時間が全体に占める割合，comb の Refinement Procedure の実行時間との比を示す．

表 14: Refinement Procedure の実行クロック数, (0.2, 0.2)

実装方法	全体	Refine	[%]	Rate
comb	3.99×10^5	7.64×10^4	19.15	1.00
seq_i	1.42×10^6	1.10×10^6	77.35	14.42
seq_i_j	1.62×10^7	1.59×10^7	98.01	207.45
seq_j	1.78×10^6	1.46×10^6	81.86	19.05
seq_x	1.35×10^6	1.02×10^6	76.05	13.40
seq_i_x	1.25×10^7	1.22×10^7	97.42	159.46
seq_j_x	1.67×10^7	1.63×10^7	98.06	213.97

表 15: Refinement Procedure の実行クロック数, (0.4, 0.4)

実装方法	全体	Refine	[%]	Rate
comb	8.74×10^5	1.80×10^5	20.57	1.00
seq_i	2.87×10^6	2.18×10^6	75.82	12.11
seq_i_j	3.04×10^7	2.97×10^7	97.71	165.02
seq_j	3.95×10^6	3.25×10^6	82.40	18.07
seq_x	2.68×10^6	1.98×10^6	74.05	11.02
seq_i_x	2.14×10^7	2.08×10^7	96.76	115.37
seq_j_x	3.18×10^7	3.11×10^7	97.82	173.04

実行クロック数を見ると，comb の Refinement Procedure の実行クロック数は全体に対して少ない割合であることがわかる．Refinement Procedure について，comb と他の実装方法と実行クロック数を比べると 10〜100 倍以上高速に実行できることがわかる．comb の次に小さいのは seq_x, seq_i と続く．seq_x の方が速いのは，前述のように FAIL exit の効率が良いのと，M を全要素同時に修正するために M が効率よく修正され，早く SUCCEED exit するのではないかと考えられる．seq_i_j, seq_i_x で seq_i_x が速いのは，FAIL exit の効率が良いのと，i のループに要するクロック数が seq_i_j の方が 1 クロック多くなってい

表 16: Refinement Procedure の実行クロック数, (0.4, 0.2)

実装方法	全体	Refine	[%]	Rate
comb	1.33×10^4	2.72×10^3	20.39	1.00
seq_i	3.30×10^4	2.24×10^4	67.87	8.25
seq_i_j	2.94×10^5	2.84×10^5	96.39	104.41
seq_j	5.90×10^4	4.84×10^4	82.02	17.81
seq_x	3.13×10^4	2.07×10^4	66.15	7.63
seq_i_x	1.48×10^5	1.37×10^5	92.82	50.48
seq_j_x	3.28×10^5	3.18×10^5	96.77	117.00

表 17: Refinement Procedure の実行クロック数, 全体

実装方法	全体	Refine	[%]	Rate
comb	1.29×10^6	2.59×10^5	20.13	1.00
seq_i	4.33×10^6	3.30×10^6	76.26	12.75
seq_i_j	4.68×10^7	4.58×10^7	97.81	176.90
seq_j	5.78×10^6	4.76×10^6	82.23	18.36
seq_x	4.05×10^6	3.03×10^6	74.65	11.69
seq_i_x	3.41×10^7	3.31×10^7	96.99	127.70
seq_j_x	4.88×10^7	4.78×10^7	97.89	184.52

表 18: Refinement Procedure の実行時間, (0.2, 0.2)

実装方法	全体 [sec]	Refine[sec]	[%]	Rate
comb	1.78×10^{-2}	3.40×10^{-3}	19.15	1.00
seq_i	5.16×10^{-2}	3.99×10^{-2}	77.35	11.73
seq_i_j	4.73×10^{-1}	4.63×10^{-1}	98.01	136.19
seq_j	6.36×10^{-2}	5.21×10^{-2}	81.86	15.32
seq_x	5.83×10^{-2}	4.44×10^{-2}	76.05	13.05
seq_i_x	3.68×10^{-1}	3.59×10^{-1}	97.42	105.43
seq_j_x	4.91×10^{-1}	4.81×10^{-1}	98.06	141.48

表 19: Refinement Procedure の実行時間, (0.4, 0.4)

実装方法	全体 [sec]	Refine[sec]	[%]	Rate
comb	3.89×10^{-2}	8.01×10^{-3}	20.57	1.00
seq_i	1.04×10^{-1}	7.89×10^{-2}	75.82	9.85
seq_i_j	8.88×10^{-1}	8.67×10^{-1}	97.71	108.33
seq_j	1.41×10^{-1}	1.16×10^{-1}	82.40	14.53
seq_x	1.16×10^{-1}	8.59×10^{-2}	74.05	10.72
seq_i_x	6.31×10^{-1}	6.11×10^{-1}	96.76	76.29
seq_j_x	9.37×10^{-1}	9.16×10^{-1}	97.82	114.41

表 20: Refinement Procedure の実行時間, (0.4, 0.2)

実装方法	全体 [sec]	Refine[sec]	[%]	Rate
comb	5.93×10^{-4}	1.21×10^{-4}	20.39	1.00
seq_i	1.20×10^{-3}	8.11×10^{-4}	67.87	6.71
seq_i_j	8.60×10^{-3}	8.29×10^{-3}	96.39	68.54
seq_j	2.11×10^{-3}	1.73×10^{-3}	82.02	14.32
seq_x	1.36×10^{-3}	8.98×10^{-4}	66.15	7.42
seq_i_x	4.35×10^{-3}	4.04×10^{-3}	92.82	33.38
seq_j_x	9.67×10^{-3}	9.36×10^{-3}	96.77	77.36

表 21: Refinement Procedure の実行時間, 全体

実装方法	全体 [sec]	Refine[sec]	[%]	Rate
comb	5.73×10^{-2}	1.15×10^{-2}	20.13	1.00
seq_i	1.57×10^{-1}	1.20×10^{-1}	76.26	10.37
seq_i_j	1.37×10^{0}	1.34×10^{0}	97.81	116.13
seq_j	2.07×10^{-1}	1.70×10^{-1}	82.23	14.76
seq_x	1.76×10^{-1}	1.31×10^{-1}	74.65	11.37
seq_i_x	1.00×10^{0}	9.73×10^{-1}	96.99	84.43
seq_j_x	1.44×10^{0}	1.41×10^{0}	97.89	122.01

るためである．また，seq_j, seq_j_x のように M を列単位で修正する場合は実行クロック数が多くなってしまうことがわかる．これは，FAIL exit により実行時間が短縮しないことと，列単位で修正すると M の修正が効率よく行われないのではないかと考えられる．例えば $r_{xj} = (\exists y)(m_{xy} \cdot b_{yj})$ の式では M の x 行目を取り出して評価するため，M を列単位で修正した場合は修正結果が反映されにくいと考えられる．M の修正が効率よく行われないと，SUCCEED exit するのが遅くなってしまう可能性がある．

実行クロック数と動作周波数により求めた実行時間の予測値もほとんどクロック数と同じ傾向になる．しかし，seq_i と seq_x の大小関係が逆転し，seq_i の方が実行時間が短くなっている．これは，seq_i の動作周波数が seq_x よりも速いためである．Refinement Procedure の実行時間だけを比べると，comb が他に対して 10〜100 倍以上速いことがわかる．それにも関わらず，全体の実行時間が他の実装方法に対してそれほど差が出ていない．このことより，comb の AT 積が良くならないのは，実行時間の影響が大部分を占めていると考えられる．

8.2 回路規模

8.2.1 面積レポート

まず，report_area で求めた回路面積を用いた AT 積を表 22, 23に示す．report_area による面積レポートは 3 ステートバッファも面積に考慮されているため，データにより一般性を持たせることができる．

表 22: 面積レポートの結果による AT 積 (1)

実装方法	回路面積	AT 積, (0.2, 0.2)	AT 積, (0.4, 0.4)
comb	1771.05	3.15×10^1	6.89×10^1
seq_i	1402.15	7.23×10^1	1.4×10^2
seq_i_j	1000.55	4.73×10^2	8.88×10^2
seq_j	1450.48	9.23×10^1	2.05×10^2
seq_x	1086.20	6.34×10^1	1.26×10^2
seq_i_x	1109.78	4.08×10^2	7.01×10^2
seq_j_x	1463.58	7.18×10^2	1.37×10^3

表 23: 面積レポートの結果による AT 積 (2)

実装方法	回路面積	AT 積, (0.2, 0.4)	AT 積, (0.4, 0.2)	AT 積, 合計
comb	1771.05	3.30×10^4	1.05×10^0	3.31×10^4
seq_i	1402.15	8.40×10^4	1.68×10^0	8.42×10^4
seq_i_j	1000.55	5.75×10^5	8.60×10^0	5.76×10^5
seq_j	1450.48	9.56×10^4	3.06×10^0	9.59×10^4
seq_x	1086.20	7.44×10^4	1.48×10^0	7.45×10^4
seq_i_x	1109.78	5.68×10^5	4.83×10^0	5.69×10^5
seq_j_x	1463.58	8.56×10^5	1.41×10^1	8.58×10^5

これらの表より，面積レポートの結果から求めた AT 積では comb が最も良い結果になることがわかる．次に表 24, 25に制御ユニット, メモリ, マルチプレクサの内訳を含めた面積レポートの結果を示す．

表 24: 面積レポートの結果 (探索木巡回アルゴリズム)

実装方法	全体	探索木	制御	メモリ	MUX
comb	1771.05	597.83	15.02	33.00	544.15
seq_i	1402.15	597.82	15.02	33.00	544.15
seq_i_j	1000.55	597.83	15.02	33.00	544.15
seq_j	1450.48	598.75	15.02	33.00	545.08
seq_x	1086.20	597.82	15.02	33.00	544.15
seq_i_x	1109.78	597.83	15.02	33.00	544.15
seq_j_x	1463.58	598.75	15.02	33.00	545.08

表 25: 面積レポートの結果 (Refinement Procedure)

実装方法	全体	Refine	制御	メモリ	MUX
comb	1771.05	1173.22	1.67	84.38	222.00
seq_i	1402.15	804.33	4.12	56.25	312.70
seq_i_j	1000.55	402.72	5.93	34.00	324.65
seq_j	1450.48	851.73	4.65	56.25	716.00
seq_x	1086.20	488.38	4.12	56.25	312.70
seq_i_x	1109.78	511.95	4.77	56.25	407.95
seq_j_x	1463.58	864.83	5.35	56.25	806.70

面積レポートより，探索木巡回アルゴリズム部の回路面積が大きいために，全体の面積差があまり見られないことが言える．これは，3 ステートバッファの面積 (0.375) が影響したせいである．3 ステートバッファはテクノロジ・マッピングされると PFU の外側に配置されるため，PFU 数に影響しない．よって，PFU 数の傾向について分析する場合，3 ステートバッファの面積を差し引いたものについて行う必要がある．3 ステートバッファはマルチプレクサのみで用いられているので，マルチプレクサの数を求めれば 3 ステートバッファの面積も求めることができる．表 26に使用するマルチプレクサの入力ポート数, データ幅, 3 ステートバッファの個数を，表 27に (p_α, p_β)=(15, 15) の回路の Refinement Procedure 内の，表 28に探索木巡回アルゴリズム内のマルチプレクサの数を示す．

次に，表 26の 3 ステートバッファ数と表 27, 28のマルチプレクサの個数を用いて，Refinement Procedure と探索木巡回アルゴリズム中の 3 ステートバッファの個数を求めた．その値と 3 ステートバッファの面積 (0.375) の積を面積レポートの結果から差し引いたものを表 29に示す．

3 ステートバッファの面積を差し引いたことで，実装方法によりある程度差が出るようになったが，いくつかの実装方法については PFU 数の傾向とは異なっている．これには Refinement Procedure 内の論理ゲート, マルチプレクサのデコーダ部で用いられている LUT の数と，メモリを実装するためのフリップフロップと RAM の数が影響してくる．LUT については，4 入力 LUT の場合，1 つあたり 0.17 と換算されているが，実際は 1 つの PFU に 4 つしか 4 入力 LUT を実装できないため，0.25 と換算した方が正しい結果が得られる可能性がある．また，5 入力 LUT の場合，1 つあたり 0.25 と換算されているが，実際には 1 つの PFU に 2 つしか 5LUT を実装できないため，0.5 と換算と換算した方が正しい結果が得られる可能性がある．メモリについては，5 章で述べたように実装方法によりメモリの実装方法が異なるため，PFU 数にも影響が出ると考えられる．

まずは LUT の使用量に関する検討を行う．

表 26: マルチプレクサの詳細

MUX	ポート数	データ幅	3 ステート
15×1	15	1	15
15×15	15	15	$225(15^2)$
15×16	15	16	$240(15 \cdot 16)$
2×1	2	1	2
2×4	2	4	8
2×15	2	15	30
2×16	2	16	32

表 27: Refinement Procedure のマルチプレクサ数

実装方法	15×1	15×15	2×1	2×4	2×15
comb	0	0	15	0	15
seq_i	0	1	15	1	15
seq_i_j	1	1	15	2	15
seq_j	0	1	$451(15^2 \times 2 + 1)$	1	15
seq_x	0	1	15	1	15
seq_i_x	1	2	15	1	15
seq_j_x	0	2	$451(15^2 \times 2 + 1)$	2	15

表 28: 探索木巡回アルゴリズムのマルチプレクサ数

実装方法	15×16	2×1	2×4	2×16
seq_j, seq_j_x	2	17	1	23
それ以外	2	16	1	23

表 29: 回路面積 (3 ステートバッファの面積を無視したもの)

実装方法	全体	探索木	Refine
comb	1120.05	126.83	993.22
seq_i	663.78	126.82	536.955
seq_i_j	253.55	126.83	126.72
seq_j	384.36	127.00	257.36
seq_x	347.83	126.82	221.01
seq_i_x	281.405	126.83	154.58
seq_j_x	310.08	127.00	183.08

8.2.2 LUT 使用量

M 修正部

Refinement Procedure 内で使用されている論理ゲートの使用量は実装方法により大幅に異なる．表 30 に論理ゲートの使用量を示す.

表 30: Refinement Procedure の論理ゲート使用量

実装方法	$(\exists y)(m_{xy} \cdot b_{yj})$	$\bar{a}_{ix} \vee r_{xj}$	$(\forall x)$	$m_{ij} \cdot (\cdots)$
ゲート数	$O(p_\beta)$	$O(1)$	$O(p_\alpha)$	$O(1)$
comb	$p_\alpha p_\beta$	$p_\alpha^2 p_\beta$	$p_\alpha p_\beta$	$p_\alpha p_\beta$
seq_i	$p_\alpha p_\beta$	$p_\alpha p_\beta$	p_β	p_β
seq_i_j	p_α	p_α	1	1
seq_j	p_α	p_α^2	p_α	p_α
seq_x	p_β	$p_\alpha p_\beta$	0	$p_\alpha p_\beta$
seq_i_x	p_β	p_β	0	p_β
seq_j_x	1	p_α	0	p_α

ゲート量をオーダーであらわすと表 30のようになるが，実際に論理合成を行うと，論理合成により論理ゲートは最適な LUT の組合せに変換される．表 31, 32に，(p_α, p_β)=(15, 15) の回路について，M の計算を行う組合せ回路を構成するコンポーネントと構成 LUT 数, コンポーネント使用量を示す．ここで，LUT4 は 4 入力 LUT, LUT5 は 5 入力 LUT のことを示す.

表 31: 組合せ回路部のコンポーネント数 (1)

実装方法	$(\exists y)(m_{xy} \cdot b_{yj})$	$(\forall x)(\bar{a}_{ix} \vee r_{xj})$	$m_{ij} \cdot (\cdots)$
LUT4	7	7	1
LUT5	2	2	0
comb	225	225	225
seq_i	225	15	15
seq_i_j	15	1	1
seq_j	15	15	15

表 32: 組合せ回路部のコンポーネント数 (2)

実装方法	$(\exists y)(m_{xy} \cdot b_{yj})$	$m_{ij} \cdot (\bar{a}_{ix} \vee r_{xj})$
LUT4	7	1
LUT5	2	0
seq_x	15	225
seq_i_x	15	15
seq_j_x	1	15

seq_j, seq_x は j または x を逐次処理にしたため，r_{xj} を求める部分が共有化され，この部分の回路規模が comb に比べて $1/p_\alpha$(seq_x), $1/p_\beta$(seq_j) になることがわかる．seq_j については sub_comb の並べる数を減らすことにより，他の部分も comb に比べて $1/p_\beta$ になっていることがわかる．seq_x, seq_i_x, seq_j_x については，2 入力 AND と 2 入力 OR を 1 つの LUT で実現するため，回路規模がさらに縮小している．

これらの表より各 LUT の使用量を調べた．表 33に各実装方法の LUT 使用量を示す.

表 33: M 修正部分の LUT 数

実装方法	LUT4	LUT5
comb	3375	900
seq_i	1695	480
seq_i_j	113	32
seq_j	225	60
seq_x	330	30
seq_i_x	120	30
seq_j_x	22	2

表 33より, M の修正部分に関しては同時に修正する要素数が同じものでも，x による逐次化により効率的に LUT の数を減らすことができることがわかる.

終了判定信号生成部

組合せ回路を用いて実装しているのは M の修正部分だけではなく，Refinement Procedure の終了判定 (SUCCEED exit または FAIL exit) を行うための信号を生成する部分もそうである．SUCCEED exit は elim という信号を生成してその値を判定する．FAIL exit は zr_mi という信号を生成してその値を判定する．表 34に elim 生成回路の, 表 35に zr_mi 生成回路の LUT 使用量を示す．ここで，LUT6 は 6 入力 LUT のことである．

表 34: elim 生成回路の LUT 数

実装方法	LUT4	LUT5	LUT6
comb	107	32	0
seq_i	16	3	0
seq_i_j	1	0	0
seq_j	10	0	1
seq_x	10	4	0
seq_i_x	8	2	0
seq_j_x	8	2	0

これらの表により，実装方法により違いはあるが，順序回路については M の修正部に比べて差が少ないことがわかる.

表 35: zr_mi 生成回路の LUT 数

実装方法	LUT4	LUT5
comb	47	32
seq_i	2	2
seq_i_j	1	0
seq_j	32	2
seq_x	47	32
seq_i_x	2	2
seq_j_x	32	2

マルチプレクサ

他に LUT 使用量に差が出る部分として，マルチプレクサが挙げられる．マルチプレクサは前述の通り実装方法により使用量に差が出る．表 36にそれぞれのマルチプレクサで用いられているデコーダと LUT 使用量を示す．表 37に表 27と表 36により求めた Refinement Procedure 内の LUT の使用量を示す．

表 36: マルチプレクサ中のデコーダと LUT 使用量

MUX	デコーダ	LUT4
15×1	4 to 15 デコーダ	15
15×15	4 to 15 デコーダ	15
15×16	4 to 15 デコーダ	15
2×1	NOT ゲート	1
2×4	NOT ゲート	1
2×15	NOT ゲート	1
2×16	NOT ゲート	1

表 37: マルチプレクサによる LUT 使用量

実装方法	LUT4
探索木 (seq_j, seq_j_x)	419
探索木 (それ以外)	418
comb	240
seq_i	259
seq_i_j	278
seq_j	695
seq_x	259
seq_i_x	289
seq_j_x	714

表 37により seq_j と seq_j_x の LUT 数が多くなることがわかる．これは，Refinement Procedure では M の修正を列単位で行っているのに対し，探索木巡回アルゴリズムでは行単位で修正を行ってため，各フリップフロップへの入力信号に対するマルチプレクサが必要になるからである．

制御ユニット

制御ユニットは状態から各制御信号を生成する組合せ回路と状態を保持するフリップフロップにより構成される．よって，制御ユニット内の LUT 数は制御変数の数に依存すると言える．表 38に各実装方法の Refinement Procedure の制御ユニット, 探索木巡回アルゴリズムの制御ユニットに用いられる LUT の使用量を示す．

表 38: 制御ユニットの LUT 数

実装方法	LUT4	LUT5	LUT6
探索木	28	26	1
comb	6	0	0
seq_i	10	2	1
seq_i_j	16	2	1
seq_j	13	3	0
seq_x	10	2	1
seq_i_x	13	3	0
seq_j_x	12	5	0

表 38より，制御ユニットの LUT 数も M の修正部に比べてあまり差がないことがわかる．

以上の表により，Refinement Procedure 内で使用している総 LUT 数を求める．表 39に LUT 数と，LUT 数より求めた PFU 数の予測値を示す．なお，ここでは述べてないが，イネーブル付デマルチプレクサ (LUT4×16, LUT5×2) の個数 (2 または 1) も含んだ値を示す．予測値とは，LUT4×4 で 1PFU, LUT5×2 で 1PFU, LUT6×1 で 1PFU とし [4]，端数は切り上げてカウントした．

表 39: Refinement Procedure の総 LUT 数, PFU 数見積り

実装方法	LUT4	LUT5	LUT6	PFU
comb	3807	968	0	1436
seq_i	2014	491	1	751
seq_i_j	442	38	1	131
seq_j	1007	73	0	288
seq_x	770	98	1	243
seq_i_x	464	41	0	137
seq_j_x	804	13	0	216

seq_j と seq_j_x については，表 39の結果から求めた PFU 数の見積り値と，表 7に示す回路全体の PFU 数とで違う傾向を示している．

LUT 以外で PFU 数に差が出る要因としては，メモリの実装方法 (RAM またはフリップフロップ) による差が考えられる．次はメモリの内訳について説明する．

8.2.3 メモリ構成

表 40に各実装方法のメモリの構成，使用量を示す．メモリを実装するには，フリップフロップで実装する場合と RAM を用いて実装する場合があるので，使用量としてフリップフロップと RAM の個数を示す．表 40で，M_2 は i 行目を一時的に格納しておくものである．

表 40: メモリの構成

実装方法	フリップフロップ数	フリップフロップ内訳	RAM 数	RAM 内訳
comb	675	$15^2 \times 3(A,\ B,\ M)$	0	−
seq_i	450	$15^2 \times 2(B,\ M)$	1	A
seq_i_j	240	$15^2 \times 1(M)$+$15(M_2)$	2	A, B
seq_j	450	$15^2 \times 2(A,\ M)$	1	B
seq_x	450	$15^2 \times 2(B,\ M)$	1	A
seq_i_x	450	$15^2 \times 2(B,\ M)$	1	A
seq_j_x	225	$15^2 \times 1(M)$	2	A, B

この表により，seq_i_x は seq_i_j より多量のフリップフロップを使用していることがわかる．OR2C FPGA では 1 つの PFU あたりに 4 つのフリップフロップしか実装できないため [4]，seq_i_x はフリップフロップにより seq_i_j より PFU 数が増加してしまうことがわかる．

8.2.4 Refinement Procedrue の PFU 使用量・内訳 (測定値)

PFU 使用量

上記の見積値が正しいかどうか確かめるために，Refinement Procedure についてテクノロジ・マッピングを行い，Refinement Procedure の PFU 使用量を求めた．表 41に回路全体と Refinement Procedure の PFU 数，Refinement Procedure の回路全体に対する割合 ([%]) を示す．

表 41: Refinement Procedure の PFU 使用量

実装方法	PFU(全体)	PFU(Refinement Procedure)	Ratio [%]
comb	2754	2537	92.12
seq_i	1770	1342	75.82
seq_i_j	467	205	43.90
seq_j	583	313	53.69
seq_x	671	381	56.78
seq_i_x	529	253	47.83
seq_j_x	387	145	37.47

表 41で，単純に全体の PFU 数と Refinement Procedure の PFU 数の差を取り，その値を探索木巡回アルゴリズムの PFU 使用量とすると，探索木巡回アルゴリズムの PFU 使用量は実装方法により差が出ることがわかる．これは，探索木巡回アルゴリズムと Refinement Procedure で別々の LUT, フリップフロップが 1 つの PFU を共有している可能性があるからではないかと考えられる．しかし，それでも Refinement Procedure の PFU 使用量が全体の PFU 使用量に影響を与えていることには変わりない．このことを確

かめるために，Refinement Procedure の PFU 使用量について comb との比 (Rate_{refine}) を取り，その値と comb の Refinement Procedure が全体に対する割合 ($\mathrm{Ratio}_{comb}[\%]$) を用いて，全体の PFU 使用量の comb に対する比の予測値 ($\mathrm{Rate}_{estimate}$) を以下の式より求めた．

$$\mathrm{Rate}_{estimate} = \mathrm{Ratio}_{comb} \times \mathrm{Rate}_{refine} + (1 - \mathrm{Ratio}_{comb}) \tag{13}$$

式 (13) より求めた PFU 使用量の比を表 42に示す．

表 42: Refinement Procedure の PFU 使用量の比

	全体			Refinement Procedure		
実装方法	PFU	$\mathrm{Rate}_{measure}$	$\mathrm{Rate}_{estimate}$	PFU	Ratio [%]	Rate_{refine}
comb	2754	1.00	1.00	2537	92.1	1.00
seq_i	1770	0.64	0.57	1342	75.8	0.53
seq_i_j	467	0.17	0.15	205	43.9	0.08
seq_j	583	0.21	0.19	313	53.7	0.12
seq_x	671	0.24	0.22	381	56.8	0.15
seq_i_x	529	0.19	0.17	253	47.8	0.10
seq_j_x	387	0.14	0.13	145	37.5	0.06

この表より，PFU 使用量の比の予測値は実測値と同じ傾向を示すことがわかる．これより，Refinement Procedure の回路規模の傾向から回路全体の傾向を予測可能であることがわかる．

LUT, フリップフロップ使用量

テクノロジ・マッピングのレポートファイルから LUT やフリップフロップの使用量を知ることができる．表 43に Refinement Procedure 内の LUT, フリップフロップの使用量を示す．

表 43: Refinement Procedure の LUT, フリップフロップ使用量

実装方法	QLUT(comb)	QLUT(Ripple)	HLUT	HLUT(LUT6)	FF
comb	3548	0	994	0	680
seq_i	1759	8	513	2	466
seq_i_j	185	16	60	2	267
seq_j	314	8	95	2	483
seq_x	526	8	110	2	466
seq_i_x	211	16	65	0	473
seq_j_x	120	16	27	0	265

表 43で，QLUT(comb) は組合せ論理モードとして実装された LUT のことである．QLUT(Ripple) はリップル・モードとして実装された LUT のことで，カウンタ, 比較器に使用される．リップル・モードでは 1 つの機能あたり 4 つの LUT，すなわち 1 つの PFU が必要となる．HLUT は 5 入力 LUT またはメモリ・モードとして実装された LUT のことで，メモリ・モードの場合は 16×4 ビット・シングル・ポート・

メモリ 1 つあたり 2 つの HLUT が必要になる．HLUT(LUT6) は HLUT を 2 つ用いることにより 6 入力 LUT として実装された LUT のことである．

この表の値と，表 39の LUT 数が異なっているのは，以下の理由からである．

- 表 39の値は，カウンタ, 比較器, 16×4 ビットシングル・ポート・メモリの LUT 数が換算されていない．
- 2 入力マルチプレクサで用いられている LUT(NOT ゲート) は，余分な LUT としてテクノロジ・マッピングにより取り除かれている．

フリップフロップについては，表 40の値は，カウンタ，制御ユニット等で用いられているフリップフロップ数が含まれていないため，表 43の値と誤差が生じる．

これらの相違点について，LUT, フリップフロップの使用量の見積値を補正した．表 44に補正後の見積値を示す．表 40の RAM は，16×4 ビットシングル・ポート・メモリを 4 つ使用するため，1 つの RAM について HLUT を 8 つ使用するものとする．

表 44: Refinement Procedure の LUT, フリップフロップ使用量 (見積値)

実装方法	QLUT(comb)	QLUT(Ripple)	HLUT	HLUT(LUT6)	FF
comb	3567	0	968	0	680
seq_i	1770	8	499	2	466
seq_i_j	194	16	54	2	267
seq_j	327	8	77	2	483
seq_x	526	8	106	2	466
seq_i_x	220	16	49	0	473
seq_j_x	120	16	29	0	265

実際の結果と見積値がほぼ近い値になっている．QLUT(comb) と HLUT の数が表 43と若干異なっているが，LUT 数の傾向は変わらないことがわかる．また，2 入力マルチプレクサの LUT 数が無視されることから，QLUT の数は M の修正部の LUT 使用量が大きく影響していると考えられる．次に表 44より PFU 数の予測値を算出した．算出方法は QLUT×4 で 1PFU，HLUT×2 で 1PFU，FF×4 で 1PFU である [4].

表 45: Refinement Procedure の PFU 使用量 (予測値)

実装方法	PFU(実測値)	PFU(予測値)	誤差
comb	2537	1546	991
seq_i	1342	813	529
seq_i_j	205	148	57
seq_j	313	245	68
seq_x	381	305	76
seq_i_x	253	202	51
seq_j_x	145	115	30

表より，PFU 数の予測値が大きくなるほど，実測値との差が大きくなることがわかる．これは，PFU への割り当て方法の差によるものであると考えられる．次にマッピングの内訳について検討する．

PFU 中の LUT, フリップフロップの内訳

PFU 使用量の予測値と実測値が大幅に異なっているのは，1 つの PFU 内の LUT, フリップフロップをどの PFU についても全て使い切るとは限らないからであると考えられる．このことを確かめるために，テクノロジ・マッピングのレポートファイルより，PFU 内の LUT, フリップフロップの内訳を調べた．表 46 に comb の，表 47に seq_i の，表 48に seq_i_j の，表 49に seq_j の，表 50に seq_x の，表 51に seq_i_x の，表 52に seq_j_x の PFU 中の LUT, フリップフロップの内訳を示す．

表 46: PFU 中の LUT, フリップフロップの内訳 (comb)

LUT4	LUT5	LUT6	FF	個数
0	2	0	0	393
0	2	0	1	45
0	2	0	3	5
1	0	0	0	476
1	0	0	1	1
1	0	0	4	18
1	1	0	0	63
2	0	0	0	1126
2	0	0	1	180
2	0	0	3	25
2	1	0	0	19
3	0	0	0	84
3	0	0	2	1
4	0	0	0	12
0	0	0	1	2
0	0	0	4	72

表 47: PFU 中の LUT, フリップフロップの内訳 (seq_i)

LUT4	LUT5	LUT6	FF	個数
0	0	1	1	1
0	1	0	0	5
0	1	0	4	5
0	2	0	0	232
0	2	0	3	3
1	0	0	0	243
1	0	0	1	1
1	0	0	2	1
1	0	0	4	22
1	1	0	0	5
1	1	0	1	1
2	0	0	0	693
2	0	0	3	27
2	1	0	0	3
2	1	0	2	1
2	1	0	3	2
3	0	0	0	3
4	0	0	0	7
4	0	0	1	1
0	0	0	1	1
0	0	0	4	63

表 48: PFU 中の LUT, フリップフロップの内訳 (seq_i_j)

LUT4	LUT5	LUT6	FF	個数
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	0	4	1
0	2	0	0	15
1	0	0	0	19
1	0	0	1	2
1	0	0	2	2
1	0	0	3	1
1	0	0	4	2
1	1	0	0	3
1	1	0	1	2
2	0	0	0	41
2	0	0	1	1
2	0	0	3	15
2	1	0	2	1
3	0	0	1	2
4	0	0	0	9
0	0	0	1	16
0	0	0	4	42

表 49: PFU 中の LUT, フリップフロップの内訳 (seq_j)

LUT4	LUT5	LUT6	FF	個数
0	0	1	0	1
0	1	0	0	2
0	1	0	4	1
0	2	0	0	22
0	2	0	1	1
0	2	0	3	7
1	0	0	0	38
1	0	0	1	2
1	0	0	4	6
1	1	0	0	2
1	1	0	2	1
1	1	0	3	1
2	0	0	0	80
2	0	0	1	1
2	0	0	3	23
2	1	0	0	2
3	0	0	0	5
4	0	0	0	7
4	0	0	3	1
4	0	0	4	3
0	0	0	1	1
0	0	0	4	84

表 50: PFU 中の LUT, フリップフロップの内訳 (seq_x)

LUT4	LUT5	LUT6	FF	個数
0	0	1	1	1
0	1	0	4	1
0	2	0	0	15
0	2	0	3	14
1	0	0	0	40
1	0	0	1	1
1	0	0	2	1
1	0	0	4	4
1	1	0	0	18
1	1	0	1	1
1	1	0	3	2
2	0	0	0	65
2	0	0	3	14
2	1	0	0	16
2	1	0	2	1
2	1	0	3	2
3	0	0	0	46
4	0	0	0	30
4	0	0	1	1
0	0	0	1	1
0	0	0	4	85

表 51: PFU 中の LUT, フリップフロップの内訳 (seq_i_x)

LUT4	LUT5	LUT6	FF	個数
0	2	0	0	17
0	2	0	2	1
0	2	0	3	1
1	0	0	0	19
1	0	0	1	1
1	0	0	2	2
1	0	0	3	2
1	0	0	4	3
1	1	0	0	1
1	1	0	1	2
2	0	0	0	37
2	0	0	3	14
3	0	0	0	5
4	0	0	0	12
4	0	0	1	2
0	0	0	1	1
0	0	0	3	15
0	0	0	4	87

表 52: PFU 中の LUT, フリップフロップの内訳 (seq_j_x)

LUT4	LUT5	LUT6	FF	個数
0	2	0	0	1
0	2	0	2	2
1	0	0	0	8
1	0	0	1	2
1	0	0	3	1
1	1	0	0	3
2	0	0	0	8
2	1	0	0	3
2	1	0	1	1
3	0	0	0	5
4	0	0	0	11
4	0	0	0	1
4	0	0	0	1
4	0	0	0	3
0	0	0	1	6
0	0	0	3	15
0	0	0	4	45

これらの表より，どの実装方法についても 1PFU あたり LUT4 を 1 個または 2 個実装してあるものが多い．総 LUT 数が多いほどこのような PFU が多くなっていることから，PFU 数の予測値と実測値に差が出るのではないかと考えられる．LUT5 については 1PFU あたり 2 個配置されているものが多いことから，LUT4 の PFU へのマッピングが PFU 数に影響していると考えられる．

また，総 PFU 数が少ないものほどフリップフロップだけを含むものの割合が多くなることもわかる．フリップフロップだけを含むものについては 1PFU あたり 3〜4 個含むものがほとんどであるため，フリップフロップだけを含む PFU の割合が小さくなるほど PFU 数の実測値と予測値の差が大きくなるのではないかと考えられる．

8.3 AT 積

以上の分析に基づいて AT 積の傾向について述べる．実行時間については，comb の Refinement Procedure の全体に占める割合が 2 割しかないため，他の実装方法に対する全体の速度向上は Refinement Procedure の速度向上の約 1/5〜1/4 になってしまうことがわかる．例えば，seq_j について例を挙げると，Refinement Procedure については comb の方が約 15 倍速いのに対して，全体の実行時間については約 4 倍しか速くなっていない．また，comb の速度向上が減っているのは，動作周波数が低いのも原因として挙げられる．

一方，回路規模については，comb の Refinement Procedure の全体に占める割合が 9 割であるため，他の実装方法の comb に対する Refinement Procedure の回路規模縮小率がほとんどそのまま回路全体の縮小率に反映されることがわかる．例えば，seq_j について例を挙げると，Refinement Procedure については約 8 分の 1 になっており，回路全体についても 5 分の 1 と，実行時間による影響に比べて Refinement Procedure の縮小率が少ない．

seq_j, seq_x については，Refinement Procedure 内の論理ゲート使用量が M の修正部で comb に比べて約 10 分の 1 に減少している．これがその他の部分の影響により回路規模の縮小率が 5 分の 1 まで減ってしまうが，comb の速度向上が探索木巡回アルゴリズムの実行時間の影響により 12〜15 倍から 4〜5 倍に減るため，AT 積は comb に比べて小さくなる．

一方，繰り返し回数が 2 乗オーダーのものについては，comb の速度向上が減っても，論理ゲート量が 2

乗オーダーであるためにそれほど論理ゲート量が減少しない．加えて，M の修正部以外の回路規模の影響により全体の回路規模があまり縮小しないため，comb に比べて AT 積が悪くなると考えられる．seq_i_j や seq_i_x がその例である．

また，論理ゲート量が 1 乗オーダーであるものは，M の修正部がいくら減っても，それ以外の部分の回路規模の影響により全体の回路規模はそれほど縮小しないため，comb より AT 積が悪くなる．seq_j_x について例を挙げると，M の修正部は comb の約 1/15 になっているにも関わらず，Refinement Procedure 全体に対して 1〜2 割しかない．さらに，Refinement Procedure 自体は全体の 37%しかないため，全体の回路規模は comb に比べて 1/8 しか縮小していない．

Refinement Procedure だけについて AT 積を取れば間違いなく comb が最も良い AT 積を得るが，全体の AT 積を見ると，comb の速度向上比の低下と，Refinement Procedure の性質を利用して回路規模の縮小により，seq_j と seq_x が comb に比べて良くなるという結果になった．

9 おわりに

Ullmann の提案手法 [3] に従い，Refinement Procedure を組合せ回路により実装すると，回路規模が膨大になるため FPGA への実装は困難である．具体的には，$(p_\alpha,\ p_\beta)$=(15, 15) の回路でも 100K ゲート未満の FPGA に収まらないことがわかる．しかし，他社の 100 万ゲート級 FPGA を用いれば，(15, 15) の回路だけでなく，(15, 15) より大きいグラフを扱える回路も実装できる可能性がある．100 万ゲート級 FPGA の例としては，ALTERA 社の EP20K1000E(標準ゲート数: 100 万ゲート), EP20K1500E(標準ゲート数: 150 万ゲート)[5] が挙げられる．しかし，組合せ回路のハードウェア量は 3 乗オーダーであるため，技術の進歩により LSI の集積度が上がったとしてもそれほど大規模なグラフを扱うことはできないであろう．

一方，本研究では，Refinement Procedure を順序回路により実装することを試みた．順序回路による実装方法では，Ullmann のアルゴリズムの性質を利用したいくつかの実装方法を提案した．その結果，組合せ回路に比べて回路規模が小さくなり，実用的な回路規模を得ることができた．さらに，順序回路による実装は Ullmann の提案，すなわち組合せ回路による実装より良い AT 積を得ることができた．実装方法を工夫することにより良い AT 積が得られたことにより，Ullmann のアルゴリズムが優れたアルゴリズムであることを裏付けている．よって，Ullmann のアルゴリズムの性質についてさらに検討することにより，seq_x より AT 積の良い実装方法が見つかる可能性は十分にある．

また，別の FPGA に実装する場合は，アーキテクチャの違いにより本研究で得られた結果と異なる可能性があるが，その FPGA のアーキテクチャに適した回路記述をすることにより，さらに良い結果が得られる可能性もある．順序回路は組合せ回路に比べて実行時間が長くなってしまうが，回路規模が抑えられるため，複数ユニットによる実装を行うことにより実行速度を増加させることが可能である．しかも，順序回路は組合せ回路より良い AT 積が得られたため，組合せ回路より実行速度が良くなる可能性がある．

しかし本研究では，Ullmann のアルゴリズムを FPGA に実装した場合の回路規模，実行時間の “見積り” を行っただけである．そこで FPGA への実装を試みて，

- 本研究の見積り結果は配線を無視したものである．実際に実装する時は配置配線を行う必要があり，配線資源の不足により実装できない可能性もあるため，本当に実装できるかを確かめる．
- 配線を考慮することにより動作周波数が下がる可能性がある．そこで，実装により本研究で得られた性能がどの程度再現できるかを評価する．

の 2 つを行う必要がある．

参考文献

[1] Michael R. Garey, David S. Johnson, “Computers and intractability : a guide to the theory of NP-completeness,” W. H. Freeman, San Francisco, 1979.

[2] D. A. Buell, J. M. Arnold, and W. J. Kleinfelder, “Splash 2: FPGAs in a Custom Computing Machine,” IEEE Computer Society, 1996.

[3] J. R. Ullmann, “An Algorithm for Subgraph Isomorphism,” J. ACM, Vol. 23, No. 1, pp. 31-42(1976).

[4] ルーセント・テクノロジー半導体販売株式会社, “ORCA^{TM} OR2CxxA と OR2TxxA シリーズ・フィールド・プログラマブル・ゲートアレイ,” 1997 年 10 月.

[5] 日本アルテラ株式会社, “APEX 20K プログラマブル・ロジック・デバイス・ファミリ,” ver. 2.02J, 1999 年 8 月.

[6] ラターナセンタン・ウドーン, “部分グラフ同型判定アルゴリズムの FPGA を用いた実装,” 豊橋技術科学大学 修士論文, 2000.

謝辞

最後まで根気よく御指導してくださった市川周一先生にこの場を借りて御礼申し上げます．また，2 年間という短い間でしたが，楽しい時や苦しい時を共に過ごしてきたラターナセンタン・ウドーン君を始めとする市川研究室のみなさんに感謝いたします．ラターナセンタン・ウドーン君には，本研究で作成したプログラムの基となる Ullmann のアルゴリズムのソフトウェアを提供してもらいました．この場を借りて御礼申し上げます．最後に，高専に 5 年間通わせてくれた上に，4 年間にわたって大学に通わせてくれた両親に心から感謝いたします．