

タイミングを考慮したフロアプランニングと動作合成の統合化の一手法

山崎 晋哉[†] 若林 真一[†]

[†] 広島大学大学院工学研究科

〒 739-8527 広島県東広島市鏡山一丁目 4 番 1 号

E-mail: †{shinya,wakaba}@bsys.hiroshima-u.ac.jp

あらまし 半導体技術がナノメートルテクノロジー領域に入り、フィジカル設計における配線工程が回路のパフォーマンスを決定する上で最も支配的な要因となってきた。このため、設計の初期段階からモジュール間の接続関係やフロアプランを考えた設計手法が必要となってきた。本研究では条件分岐を含むコントロールデータフローグラフ (CDFG) に対し、条件分岐の排他性を利用したリソースの有効利用を行い、スケジューリング、アロケーション、バインディング等の動作合成中に最終的に生成されるであろう回路のレイアウトを予測しながら動作合成を行う手法を提案する。提案手法では、より正確な配線遅延見積もりを行うことで、膨大な設計空間の中からチップ面積と動作周波数の設計制約を満たす空間だけを効率的に探索することを可能にする。

キーワード 動作合成, フロアプランニング, スケジューリング, バインディング, タイミング制約

An Integrated Method of Timing-Driven Floorplanning and Behavioral Synthesis

Shinya YAMASAKI[†] and Shin'ichi WAKABAYASHI[†]

[†] Graduate School of Engineering, Hiroshima University

4-1, Kagamiyama 1 chome, Higashi-Hiroshima, Hiroshima, 739-8527 JAPAN

E-mail: †{shinya,wakaba}@bsys.hiroshima-u.ac.jp

Abstract In deep submicron semiconductor technology, interconnection becomes the most dominant factor of system performance. Thus in the early step of VLSI design, the design technique considering interconnection among modules as well as floorplanning is needed. In this paper, we focus on a control data flow graph (CDFG) including conditional branches, and effectively handle resource sharing based on exclusiveness of conditional branches. We propose a behavioral synthesis method consisting of scheduling, allocation, and binding, with the prediction of a final layout. The proposed method searches only the space which satisfies design constraints of chip area and clock frequency in a huge design space by performing a more exact wiring delay estimation.

Key words Behavioral Synthesis, Floorplanning, Scheduling, Binding, Timing Constraint

1. ま え が き

LSI 製造プロセスの微細化、回路の大規模化に伴い、LSI チップにおいてシステム全体の遅延に占める配線遅延の割合がますます増大している。このため、LSI 設計において従来の配線工程だけの配線最適化は困難になり、配置配線を同時に考慮した手法や、論理設計からレイアウトを考慮した手法などが数多く研究されている。また一方で、設計階層の最も上位に位置する動作合成に関する研究もさかに行われており、C 言語等を入力として与え RTL 記述を自動生成する設計技術も実用段階に達しつつある。しかし、従来の動作合成システムでは、レイ

アウト工程を十分に考慮しておらず、演算器のリソース数、コントロールステップ数を最小化することが最適化の目的であり、この結果を基に、演算を演算器へ、演算データをレジスタへそれぞれ割り当てるバインディング工程へ続き、演算器とレジスタの接続関係を生成した後、レイアウトを行う逐次的な改良が行われる。しかし、このようなアーキテクチャ合成、論理合成、レイアウト設計を逐次的に行う従来の設計手法ではタイミング収束を達成することが困難である。この理由は、フィジカル設計が行われるまで正確な配線遅延情報が見積もれ無いため、動作合成や論理合成では不正確な配線情報を使うか、もしくは配線情報無しで処理を実行するためである [9]。

近年では、配線遅延が回路のパフォーマンスに多大な影響を与えるため、設計の初期段階から面積、遅延、電力消費のようなレイアウト情報を見積もる必要がある。しかし、これらの情報を得るためにはかなり詳細なレイアウト情報を持たないとより高い精度での見積もりは不可能である。このため、近年多くの研究者がフロアプランニングとスケジューリングやバインディング等の動作合成を同時に考慮する手法の研究を行っている[3],[4],[8]。

3-D[3]は始めにスケジューリングとバインディングを行って、その後フロアプランニングを実行し配線遅延を減少させるように改良を行う。BITNET[5]はフィジカル情報に基づくバインディングを考慮しているが配線遅延を考慮しておらず、SMB[4]は、Functional Unit(FU,以降は演算器と呼ぶ)のバインディングとフロアプランニングを同時に実行することでクリティカルパス遅延を最小化している。

これらの手法はいずれも条件分岐の無いデータバスを対象としており、制御フローの含まれるデータバスは扱うことが出来ない。本研究では、入力として与えられた条件分岐を含むコントロールデータフローグラフに対し、条件分岐の排他性を利用したリソースの有効利用を行い、動作合成の主なステップであるスケジューリング、アロケーション、バインディング等の動作合成中に最終的に生成されるであろう回路のレイアウトを予測する。同時に、より正確な配線遅延見積もりを行うことで、膨大な設計空間の中からチップ面積と動作周波数の設計制約を満たす空間だけを効率的に探索することを可能にする。

2. 準 備

2.1 動作合成

動作合成とは、データフローグラフ(DFG)のような動作記述から演算器、レジスタ、接続関係等から構成されるRTL記述を合成する工程である。DFGの各頂点は演算(加算、乗算等)を表しており、辺は変数を表している。このとき、与えられたDFGと演算器やレジスタのリソース資源制約、実行時間制約が与えられるとこれらを最小とするRTL記述を出力する。

動作合成は主にスケジューリング、アロケーション、バインディングの3つの処理からなっている。以下簡単に各手続きを説明する。

2.1.1 スケジューリング

スケジューリングとは、動作記述中の演算の実行順序と、演算を実行するタイミング(コントロールステップ)を決定する処理であり、これにより、全体のステップ数と必要なハードウェアリソースが決定されるため、最終的に生成されるLSIチップの性能に大きな影響を与える。演算のスケジューリングは、大きく次の2つに分類される。

【資源制約型スケジューリング】：演算器の種類と最大利用可能数が与えられた場合、その制約内で実行ステップ数を最小とする。

【時間制約型スケジューリング】：最大実行ステップ数が与えられた場合、その制約内で演算器数を最小とする。

これらのスケジューリングに対し、資源制約が無いと仮定して

各演算を出来るだけ早く開始するASAP(As Soon As Possible)スケジューリングと、同様に各演算を出来るだけ遅く開始するALAP(As Late As Possible)スケジューリングがある。

2.1.2 アロケーション

アロケーションとは、データバスの実現に必要なハードウェアユニットの数と種類を決定する処理で、算術演算のための演算器、変数を保持するためのレジスタやメモリ、配線のためのバスやマルチプレクサなどの数と種類を決定する。

2.1.3 バインディング

バインディングとは、ハードウェア面積の最小化を目的として、動作記述で示された演算、演算データ、データ転送要求を、それぞれ利用可能な演算器、レジスタ、データ転送路に割り当てる処理で、一般にスケジューリング結果に基づいて実行する。

演算が実行されている期間とデータを保持する期間をそれぞれ演算のライフタイム、データのライフタイムと呼ぶ。各演算器とレジスタは同時に一つの演算かデータしか扱えないため、同一ステップに二つ以上の演算、データを共有することは出来ない。また、逆にライフタイムの異なる演算、データに関しては同一の演算器、レジスタを使用することが出来る。

2.2 条件ベクタ(CV)

提案手法では、制御フローを含むデータバスを効率良く扱うために、文献[11]で提案された条件ベクタ(CV)を用いる。この手法は、入れ子になった条件分岐に対し効率的な制御シーケンスを合成することが目的で、このCVを用いることで、オペレーション間の相互排他性を検出し、これにより優れたリソースの共有を実現している。また、コントロールデータフローグラフ(CDFG)にCVの概念を組み込むことで、条件分岐ノードを含むグラフに対しても、条件分岐を含まないデータフローグラフと同様の方法で扱うことが出来る。

基本となるCVは、分岐に対して異なる葉(ノード)ベクトルのビット列を定義する。つまり全ての葉(ノード)は唯一の状態が定義される。例を図1に示す。(a)は、始めにxが求められ、その後if文の条件式(c1,c2)によってyが求められるという動作記述を示している。

(b)は、(a)のBCV(Basic Condition Vector)に基づいた制御木を示しており、例えば(b)のn2とn5はそれぞれ排他的な関係があり、if構文c1によって、どちらかのブランチが実行される(入れ子に入る)かが判断されている。また、葉で無いノードは、それらの子の論理和で求められる(例、n5 = n3 ∪ n4)。

(c)は、(a)のECV(Extended Condition Vector)に基づいた制御木を示している。このECVは、BCVをCDFGノードに対して拡張されたものであり、ECVはそのノードがどこで実行される必要があるかの条件を示している。(a)の例では、式(3)の場合だとxを計算する必要が無く、(c)のように、xを実行する必要がある場合と無い場合に分けて考えられる。この考え方が、効率的なスケジューリングを導く。

また、図2の例では、(a)の動作記述を1つずつの加算器、減算器、比較器で実行すると仮定した場合、条件演算から逐次的に実行すると仮定すると最低5ステップ必要であるが、(b)に示すように分岐内演算を先に行うと同じ演算器制約で3ステッ

```

x = a+b-c+d;      --- (1)  BCV  [1,1,1]  ECV  [1,0,1]
if (c1)           y = x+1; --- (2)  [1,0,0]  [1,0,0]
else if (c2)      y = e+1; --- (3)  [0,1,0]  [0,1,0]
else              y = x-1; --- (4)  [0,0,1]  [0,0,1]

```

(a) 動作記述

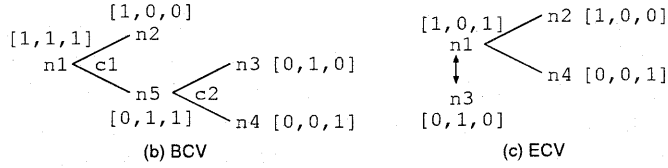


図1 if構文による条件ベクタ (CV) の例

ブで実行が可能であることを示している。この時の (b) の各演算ノードの右側に記す CV より、Step1 で実行される '+' 演算は常に実行され (CV[1,1]), Step3 の '+' 演算は排他になり加算器を共有できる (CV[1,0][0,1]) ことを示している。

このように、CV を利用すると、各ステップのベクトルの和を計算することでそのステップに割り当て済みの演算器数の判定や演算の分割による分岐内の演算の数を自由に変更するなどが容易に行え、効果的なスケジューリングを行うことが出来る。詳細は [11] を参照されたい。

2.3 問題の定式化

動作合成においてフィジカル設計の影響を検討するために、目的とするハードウェアアーキテクチャを定義する必要がある。本稿では、マルチプレクサベースのアーキテクチャを対象としており、マルチサイクルオペレーションを考慮している。

以下に問題の定式化を行う。

入力

- コントロールデータフローグラフ (CDFG)

入力として有向グラフであるコントロールデータフローグラフ $G = (V, E)$ を与える。 $V = \{v_i \mid i = 1, 2, \dots, n\}$ はノードの集合、 $E = \{(v_i, v_j) \mid i, j = 1, 2, \dots, m, i \neq j\}$ は各ノードの接続関係の集合をそれぞれ表している。また、ノード集合 V は、加算や乗算等を表す演算ノードの集合 V_O 、定数や変数等を表すデータノードの集合 V_D 、データの流れの制御を表す構造ノードの集合 V_C を含んでおり、 $V = V_O \cup V_D \cup V_C$ とする。

- オペレーション集合

オペレーション (以降では、演算器) は l 種類存在し、各演算器の利用可能な最大数を k_i ($i = 1, 2, \dots, l$) とする。この時、演算ノードの演算器への割り当てを以下のように定義する。 $\rho: V_O \rightarrow \{1, 2, \dots, l\}$ 。また、各演算器の面積と遅延情報はライブラリ内に持っている。

- システム情報

最大クロック周期

出力

- スケジューリング結果

演算ノード V_O をステップ S にスケジューリングする。 $\delta: V_O \rightarrow S$

- バインディング結果

演算ノード V_O とデータノード V_D をそれぞれ演算器とレジスタに割り当てる。 $\rho: V_O \rightarrow FU$ 。 $\sigma: V_D \rightarrow R$ 。

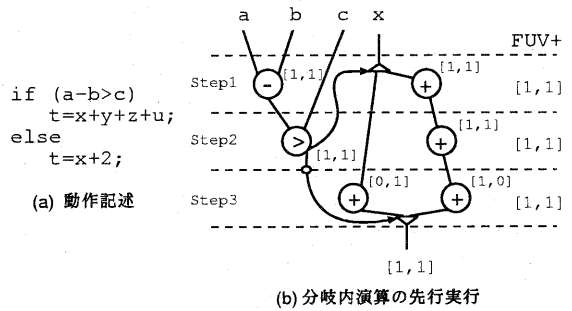


図2 コントロール・データフローグラフ (CDFG) の例

• レイアウト

演算器とレジスタを二次元平面上に配置する。

制約

S_{max} : 最大ステップ数制約

$K = \{k_i \mid i = 1, 2, \dots, l\}$: 使用可能な演算器の個数

A_{max} : チップ面積制約

T_{const} : バス遅延制約

3. 提案アルゴリズム

提案手法は、入力として与えられた C 言語プログラムを基に、スケジューリングを行った CDFG を生成し、バス長やレジスタのライフタイムの比較的大きいクリティカルなパスにつながる演算をブロックとして形成し、優先してステップの割り当てやレイアウトを行う。残された演算ノードもリソース制約が許す限りブロックを生成し、ブロック内部の仮想レイアウトを行う。最後に生成した全てのブロックのレイアウトを行い、タイミングや面積制約を満たす解を生成するまで逐次改良を行う。

提案手法の詳細アルゴリズムを以下に示す。

【準備】 CDFG を生成。

C 言語等の入力プログラムから CDFG を生成する。

【step1】 レイアウト素子のおおまかな見積もり。

生成した CDFG とリソース制約より、演算 (演算器数のリソース制約)、変数 (レジスタ数)、接続関係 (配線) からこの入力プログラムから予想されるおおよそのレイアウトの大きさを見積もる。後で対象としているブロックがどれほど重要 (優先度が

高い)かを判別するのに使用する。

【step2】条件ベクタ (CV) の設定。

与えられた CDFG に対して、各演算に CV を設定する。これにより、図 1 に示すような条件分岐での各オペレーションの排他性を検出する。また、条件分岐を含まない DFG を入力データとする場合、CV の設定を行わず【step3】へ進む。

【step3】スケジューリング。

(1) 演算器リソース制約の基で、各演算の ASAP, ALAP 値の差であるライフタイムを用いてステップ数最適化のリストスケジューリングを行う。

(2) 入力が条件分岐を含む CDFG である時、【step2】で設定した条件ベクタを効率的に使用し条件分岐による排他性や図 2 に示すような先行実行演算の変更を用いることでスケジュールを改良する。

ここでのスケジューリングは最長バスの検出が主な目的であり、【step4】以降の工程でスケジューリングの改良の自由度を残すためにも、必要以上の改良を行わない。また、提案手法では乗算器と加算器のそれぞれの実行ステップ数を考慮して、マルチサイクルに対応したスケジューリングを行う。

【step4】バスの選択とブロックの生成。

入力が CDFG である場合、任意のレベルの条件分岐内を部分 DFG と考える。

(1) 部分 DFG か、もしくは入力 DFG に対して最長バスを検出し、このバスに含まれる演算を優先的に有効なステップに割り当てる。この時、バスに含まれる全ての演算を必要最小数の演算器に割り当てることとする。ここで、選択したバスの全ての演算を含む集合をブロックと呼ぶ。最長バスの例を図 3(a) に示し、対応するスケジューリングを表 1(a) に示す。

(2) ブロック内のあるステップで未使用の演算器があれば、ブロック内の演算に接続する演算をそのステップに割り当てブロックを拡大する。例えば、表 1(a) のスケジュール結果は、(b) のように改良される。対応するブロックを図 3(b) に示す。

(1) の工程より、ステップ数最小が見込まれ、(2) より、演算間のバスを出来るだけ寸断すること無く演算器やレジスタの効率的な利用が可能となる。さらに、ブロックに追加するバスの選択にあたって、複数の出力先を持ち、かつ比較的長いデータのライフタイムを必要とするバスを寸断することなく、一つのレジスタに値を保持するようにブロックを作ることで、レジスタ数の減少が見込まれ、面積の最小化につながると考えられる。

【step5】ブロック内部の仮想レイアウト。

(1) ブロック内部に含まれる演算器の種類とスケジューリング結果を基に、各演算データのライフタイムを用いてレジスタバインディングを行う。

(2) 演算器、レジスタ、演算器-レジスタ間のバスよりブロック内の仮想的なレイアウトを行う。この時、全てのバスにバス遅延制約を与える。もしレイアウトを想定して実現不可能な場合は、

● 選択したブロック内部の演算ステップの再割り当てとレジスタバインディングによるライフタイムの変更を行い、再度仮想レイアウトを行う。

● 最大ステップ数制約範囲内でステップ数の増加を許してブロック内を再スケジューリングし、レジスタバインディングと仮想レイアウトを行う。

● ブロック内部だけの変更では制約を満たすレイアウトが不可能であれば、全体のグラフを対象としてオペレーションの複製やステップの再割り当てなどを行って再スケジューリングし、【step4】に戻りバスを選択しブロックを生成する。

(3) また、選択したバスを基にブロック内部の仮想レイアウトが確定すると、対応するスケジューリング・バインディング結果を元の全体のグラフにフィードバックし、それまでに生成したクリティカルなブロックに影響を及ぼさない範囲で再スケジューリングを行い、【step4】に戻りバスを選択しブロックを生成する。

【step6】全体のレイアウトの完成。

出来上がった複数個のブロックと、ブロック生成によって寸断したバスを基に全体のレイアウトを行う。生成されたレイアウトの例を図 4 に示す。このレイアウトでは図 3(b) のスケジュール結果を基に最長バスを含む部分ブロックを先に形成し、出来上がった 2 つのブロックを遅延制約を満たすように配置している。

【step7】レイアウトの最適化。

【step6】で実現したレイアウトに対して、ステップ数、リソース数、タイミング制約など各制約を満たしているかをチェックし、また、異なるブロック間の演算器やレジスタの各ステップでの使用状況を確認することで、さらなる効率的なパッキングを行う。

4. あとがき

本稿では条件分岐を含む CDFG に対し、条件分岐の排他性を利用したリソースの有効利用を行い、スケジューリング、バインディング等の動作合成中に最終的に生成されるであろう回路のレイアウトを予測しながら動作合成を行う手法を提案した。これにより、設計の初期段階からチップ面積と動作周波数の設計空間を満たす空間だけを効率的に探索することを可能にする。

現在、C 言語により提案手法の実装を進めており、部分ブロックの生成まで完了している。今後レイアウトを考慮したブロック内部とブロック全体のレイアウトの逐次改良を実現し、完成次第、ワークステーション上でシミュレーション実験を行う予定である。

文 献

- [1] D.Gajski, N.Dutt, A.Wu and S.Lin, "High-Level Synthesis: Introduction to Chip and System Design," Kluwer Academic Publishers, 1992.
- [2] P.Michel, U.Lauther and P.Duzy, "The Synthesis Approach to Digital System Design," Kluwer Academic Publishers, 1992.
- [3] J.Weng and A.C.Parker, "3D scheduling: High-Level synthesis with floorplanning," Proc. of 28th ACM/IEEE Design Automation Conference, pp.668-673, 1991.
- [4] Y.Fang and D.F.Wong, "Simultaneous functional-unit binding and floorplanning," Proc. International Conference on Computer Aided Design, pp.317-321, 1994.

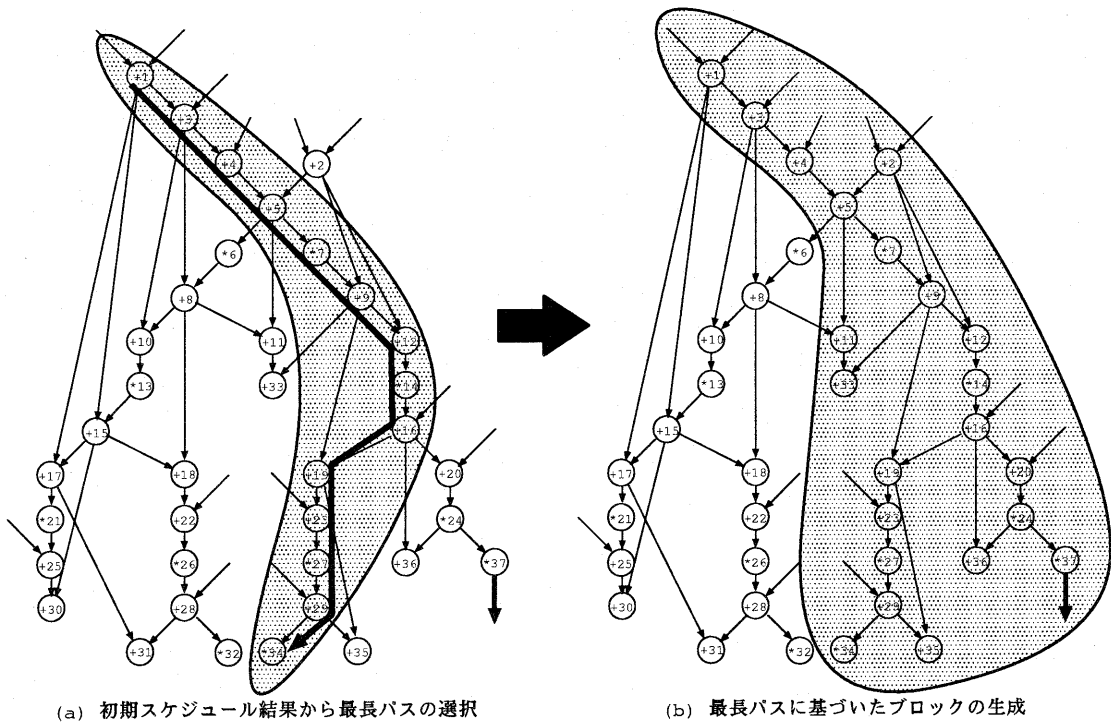


図3 最長パスを基にしたブロックの形成 (入力 DFG は Elliptical Wave Filter)

step	+1	+2	*1	*2
1	Op ₁	Op ₂		
2	Op ₃			
3	Op ₄			
4	Op ₅			
5		Op ₇	Op ₆	Op ₇
6		Op ₉	Op ₆	Op ₇
7	Op ₈	Op ₉		
8	Op ₁₀	Op ₁₂		
9	Op ₁₁		Op ₁₃	Op ₁₄
10	Op ₃₃		Op ₁₃	Op ₁₄
11	Op ₁₅	Op ₁₆		
12	Op ₁₈	Op ₁₉		
13	Op ₁₇	Op ₂₀		
14	Op ₂₂	Op ₂₃	Op ₂₁	Op ₂₄
15			Op ₂₁	Op ₂₄
16	Op ₂₅	Op ₃₆	Op ₂₆	Op ₂₇
17			Op ₂₆	Op ₂₇
18	Op ₂₈	Op ₂₉	Op ₃₇	
19	Op ₃₀	Op ₃₁	Op ₃₇	Op ₃₂
20	Op ₃₅		Op ₃₄	Op ₃₂
21			Op ₃₄	
22				

(a) 初期スケジューリング結果

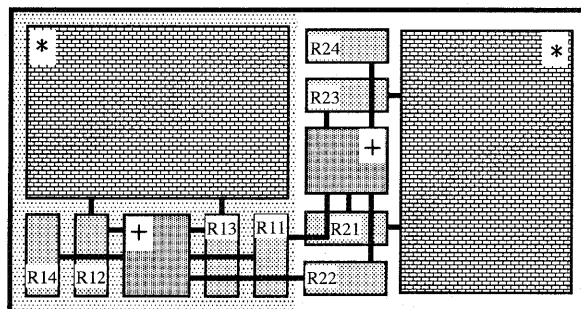
step	+1	*1	+2	*2
1	Op ₁			
2	Op ₃			
3	Op ₄			
4	Op ₂			
5	Op ₅			
6		Op ₇		Op ₆
7		Op ₇		Op ₆
8	Op ₉		Op ₈	
9	Op ₁₂		Op ₁₀	
10	Op ₁₁	Op ₁₄		Op ₁₃
11	Op ₃₃	Op ₁₄		Op ₁₃
12	Op ₁₆		Op ₁₅	
13	Op ₁₉		Op ₁₈	
14	Op ₂₀		Op ₁₇	
15	Op ₂₃	Op ₂₄	Op ₂₂	Op ₂₁
16		Op ₂₄		Op ₂₁
17	Op ₃₆	Op ₂₇	Op ₂₅	Op ₂₆
18		Op ₂₇		Op ₂₆
19	Op ₂₉	Op ₃₇	Op ₂₈	
20		Op ₃₇	Op ₃₀	Op ₃₂
21	Op ₃₅	Op ₃₄	Op ₃₁	Op ₃₂
22		Op ₃₄		

(b) 最長パスに基づいたブロック生成結果

表1 スケジューリング結果 (制約: 最大ステップが22, 加算器, 乗算器が2つずつ)

block 1	
R11	1, 2, 3, 4, 16
R12	5, 7, 12, 14, 23, 27
R13	11, 20, 24, 29
R14	9, 19

block 2	
R21	6, 10, 13, 22, 26, 28
R22	8, 21, 25
R23	15
R24	17



(a) バインディング結果

(b) レイアウト結果

図4 図3(b)のスケジューリング結果を基にしたレイアウトの例

- [5] A.Mujumdar, R.Jain and K.Saluja, "Incorporating performance and testability constraints during binding in high-level synthesis," IEEE Transactions of Computer Aided Design of Integrated Circuits and System, Vol.15, No.10, October pp.1212-1225, 1996.
- [6] M.Xu and F.J.Kurdahi, "Layout-driven high level synthesis for FPGA based architectures," Proc. Design, Automation and Test in Europe, pp.446-450, 1998.
- [7] S.Tarafdar, M.Leeser and Z.Yin, "Integrating floorplanning in data-transfer based high-level synthesis," Proc. International Conference on Computer Aided Design, pp.412-417, 1998.
- [8] W.E.Dougherty and D.E.Thomas, "Unifying behavioral synthesis and physical design," Proc. of 37th ACM/IEEE Design Automation Conference, pp.756-761, 2000.
- [9] Daehong Kim, Jinyong Jung, Sunghyun Lee, Jinhwan Jeon and Kiyoun Choi, "Behavior-to-placed RTL synthesis with performance-driven placement," Proc. International Conference on Computer Aided Design, pp.320-325, 2001.
- [10] R.Camposano, "Path-based scheduling for synthesis," IEEE Transactions of Computer Aided Design of Integrated Circuits and System, Vol.10, No.1, January pp.85-93, 1991.
- [11] K.Wakabayashi and T.Yoshimura, "A resource sharing and control synthesis method for conditional branches," Proc. International Conference on Computer Aided Design, pp.62-65, 1989.
- [12] K.Wakabayashi and H.Tanaka, "Global scheduling independent of control dependencies based on condition vectors," Proc. of 29th ACM/IEEE Design Automation Conference, pp.112-115, 1992.