# 遅延ばらつき許容量を調整できるRDRアーキテクチャ向け高位合成手法

萩尾 勇太 † 柳澤 政生 †† 戸川 望 †

† 早稲田大学大学院基幹理工学研究科情報理工学専攻
†† 早稲田大学大学院基幹理工学研究科電子光システム学専攻

LSI の微細加工技術の進歩により，配線遅延の拡大や製造時の遅延ばらつきによるタイミング違反が問題となっている．とりわけ配線遅延がゲート遅延と比較して相対的に増加しており，高位合成段階でいかに配線遅延を取り扱うかが鍵となる．また，製造時の遅延ばらつきに対応するために，従来は過剰なマージンの挿入，統計的静的遅延解析などが適用されてきたが，近年は性能低下しない手法としてチップ製造後の回路チューニングが提案されている．このような背景に基づき，本稿では遅延ばらつきの許容量を調整できる RDR アーキテクチャ向け高位合成手法を提案する．遅延ばらつきによるタイミング違反が発生しない場合と発生した場合の 2 通りのスケジューリング，バインディングを想定することで，配線遅延の拡大と製造時の遅延ばらつきに対応した高位合成を実現する．入力としてばらつき率を与えることで，ばらつきの許容量を調整する．また，RDR アーキテクチャの空き領域を利用しここに演算器を追加することで，遅延ばらつきによるタイミング違反が発生した場合でも実行時間の最小化を図る．さらに，2 通りのスケジューリング，バインディング結果に類似化という考えを導入することでチップ面積を最小化する．計算機実験により，提案手法は従来手法と比較して遅延ばらつき発生時の実行時間を最大 32.3% 削減できることを確認した．

# A High-Level Synthesis Algorithm with Delay Variation Tolerance Adjustment for RDR Architectures

Yuta HAGIO† Masao YANAGISAWA†† Nozomu TOGAWA†

† Dept. of Computer Science and Engineering, Waseda University.
†† Dept. of Electronic and Photonic Systems, Waseda University.

As device feature size drops, interconnection delays often exceed gate delays. We have to incorporate interconnection delays even in high-level synthesis. Using RDR architectures is one of the effective solutions to this problem. At the same time, process and delay variation also becomes a serious problem which may result in several timing errors. How to deal with this problem is another key issue in high-level synthesis. In this paper, we propose a high-level synthesis algorithm with delay variation tolerance adjustment for RDR architectures. We first obtain a *non-delayed* scheduling/binding result and a *delayed* scheduling/binding result independently. When we obtain a *delayed* scheduling/binding result, we use *variation rate*. By adding several extra functional units to *vacant* RDR islands, we have a delayed scheduling/binding result so that its latency cannot be increased compared with the non-delayed one. After that, we *similarize* the two scheduling/binding results by repeatedly modifying their results. We can finally realize non-delayed and delayed scheduling/binding results simultaneously on RDR architecture with almost no area/performance overheads and we can select either one of them depending on post-silicon delay variation. Experimental results show that our algorithm successfully reduces delayed scheduling/binding latency by up to 32.3% compared with the conventional approach.

## 1 はじめに

半導体の微細加工技術の発達により，チップ面積の削減や消費電力の低減，回路性能の向上などが実現されている．その一方、配線遅延の拡大や製造時の遅延ばらつきによるタイミング違反が深刻な問題となっており，対応が求められている．

また，LSI 設計において，高位合成は時間短縮と設計コスト削減の面から重要な技術となっている．高位合成は抽象度の高い動作記述から，レジスタやクロックによる同期などのハードウェア特有の概念を意識した RTL 記述を自動合成する．

とりわけ配線遅延がゲート遅延と比較して相対的に増加している今日の LSI 設計では，高位合成の段階でいかに配線遅延を扱うかが鍵となっている．配線遅延を考慮した高位合成として，RDR (Regular Distributed Register) アーキテクチャを対象とした高位合成手法が報告されている [2]．RDR アーキテクチャはチップを規則的な 2 次元配列の島に分割し，それぞれの島の内部に LCC (Local Computational Cluster) と呼ばれる演算器群とローカルレジスタ，FSM (Finite State Machine) を持つ．$2 \times 3$ の島を持つ RDR アーキテクチャの例を図 1 に示す．RDR アーキテクチャでは演算器とレジスタを隣接して配置し，それぞれの島の内部では 1 コントロールステップ以内でデータ転送できるように設計されている．島間のデータ通信では，レジスタ間通信により複数クロックサイクルかけてデータ転送する．また，島が規則的に配置されているため，高位レベルで配線遅延を予測できる．

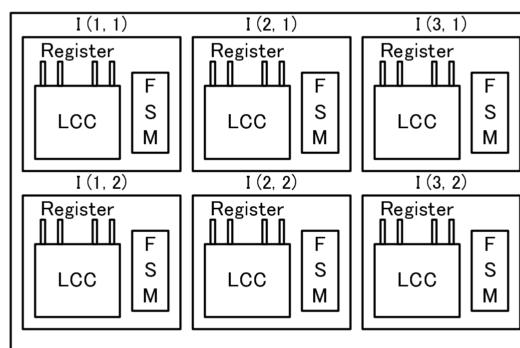



図 1: $2 \times 3$ の島を持つ RDR アーキテクチャ．

一方，製造時の遅延ばらつきによるタイミング違反も問題となっており対応が求められている．この問題に対応するために，従来は過剰なマージンの挿入，統計的静的遅延解析 [1] などが適用されてきた．しかし，このような手法を適用した場合，回路性能の低下は避けられない．近年，性能低下を起こさない手法としてチップ製造後の回路チューニングが提案されている [8]．この手法を用いることで，入力として与えられた機能修正信号をもとに，製造後に動作を変更できる．

以上の議論のもと，我々は配線遅延の拡大と製造時

©2014 Information Processing Society of Japan

の遅延ばらつきの双方に対応した高位合成技術として，製造後遅延調整機能を持つ RDR アーキテクチャ向け高位合成手法を提案した [4]．配線長の増大にしたがって遅延ばらつきの発生率が増大するとの仮定のもと，配線長が最大である配線の遅延ばらつきを考慮し，遅延ばらつきによるタイミング違反が発生しない場合と発生した場合の 2 通りのスケジューリング，バインディングを想定することで，配線遅延の拡大と製造時の遅延ばらつきの双方に対応した高位合成を実現した．また，面積の利用効率が悪いという RDR アーキテクチャの欠点を逆手に取り，空き領域に新たな演算器を追加することで遅延ばらつきによるタイミング違反が発生した場合でも実行時間の最小化を図り，2 通りのスケジューリング，バインディング結果に類似化という考えを導入することでチップ面積を最小化した．しかし，最長の配線以外のばらつきは考慮しておらず，演算器の遅延ばらつきも考慮していなかった．また，遅延ばらつきを考慮する配線においても，一括して 1 コントロールステップ多く見積もるという手法であった．各演算およびデータ転送において，遅延ばらつきを許容できる量は異なるため，一括で扱うのは非効率である．

本稿では，遅延ばらつき許容量を調整できる RDR アーキテクチャ向け高位合成手法を提案する．提案手法では，ばらつき率を入力として与えることで，演算器や配線，レジスタ，MUX の遅延見積もりを変動させ，これら全ての遅延ばらつきに対応できる．また，空き領域への新たな演算器の追加と 2 通りのスケジューリング，バインディング結果の類似化も導入することで，遅延ばらつき発生時のレイテンシおよびチップ面積を最小化する．提案手法を計算機上に実装し，従来手法と比較した結果，遅延ばらつき発生時の総コントロールステップ数を最大 32.3% 削減できることを確認した．

本稿は以下のように構成される．2 章では遅延ばらつき許容量を調整できる RDR アーキテクチャ向け高位合成問題を定義する．3 章では遅延ばらつき許容量を調整できる RDR アーキテクチャ向け高位合成手法を提案する．4 章では実験結果から提案手法の有効性を示す．5 章では結論と今後の研究課題を提示する．

## 2 問題の定式化

動作記述の中間形式のグラフ表現として，DFG (Data Flow Graph) を利用する．DFG $G = (V, E)$ は有向グラフで表現される．$V$ は演算ノードの集合から成り，$E$ はデータフローエッジの集合から成る．本稿では説明を簡単にするため DGF を入力として用いるが，CDFG (Control-Data Flow Graph) でも同一の議論ができる．

### 2.1 RDR アーキテクチャ

RDR アーキテクチャは，自然数 $N, M$ に対してチップを $N \times M$ の 2 次元配列の島に分割する．それぞれの島は正方形である．$1 \leq x \leq N$, $1 \leq y \leq M$ に対して，座標 $(x, y)$ 上の島を $I(x, y)$ と表現する．演算器 $fu$ はいずれかの島に配置され，遅延時間を $d_{fu}$ とする．また，レジスタおよび MUX の遅延時間をそれぞれ $d_R$，$d_M$ とする．すべての島 $I(x, y)$ はそれぞれローカルレジスタ $R(I(x, y))$ を持つ．特定の 2 つの島 $i_1 = I(x_1, y_1)$, $i_2 = I(x_2, y_2)$ 間の距離は一定であり，配線遅延が距離の 2 乗に比例する [5, 7] ことから $i_1$, $i_2$ 間の配線遅延時間 $D_c(i_1, i_2)$ を

$$D_c(i_1, i_2) = C_d \times (|x_1 - x_2| + |y_1 - y_2|)^2$$

とする．なお，$C_d$ は配線遅延係数である．

島 $i_1 = I(x_1, y_1)$ に配置された演算器の 1 つを $fu_1$ とする．$fu_1$ が演算を実行し，その結果を島 $i_2 = I(x_2, y_2)$ に配置されたローカルレジスタ $R(i_2)$ に格納することを考える．クロック周期を $T_{clk}$ とし，$d_{fu_1} < T_{clk}$ であると仮定する．

$$T_{clk} \geq D_c(i_1, i_2) + d_{fu_1} + d_R + d_M$$

のとき，$fu_1$ の演算実行と同じコントロールステップで $R(i_2)$ へデータを格納する．一方，

$$T_{clk} < D_c(i_1, i_2) + d_{fu_1} + d_R + d_M$$

のとき，最初のコントロールステップでは $fu_1$ の実行結果を島 $i_1$ のレジスタ $R(i_1)$ に格納する．そして，次のコントロールステップ以降，$\lceil \frac{D_c(i_1,i_2)}{T_{clk}} \rceil$ ステップかけて島 $i_1$ のレジスタ $R(i_1)$ から島 $i_2$ のレジスタ $R(i_2)$ へデータを転送する．$d_{fu_1} \geq T_{clk}$ の場合も同様に，配線遅延時間が大きい場合にはデータ転送のためだけにクロックサイクルを使用する．

RDR アーキテクチャのすべての島は容量制約 $C$ を持つ．演算器 $fu$ の配置に必要な容量コストを $c_{fu}$ とする．島 $i = I(x, y)$ に配置されている演算器の集合を $Fu(i)$ とすると，任意の島 $i$ について

$$C \geq \sum_{fu \in Fu(i)} c_{fu}$$

が成り立たなければならない．したがって，ある演算器 $fu_i$ について，

$$c_{fu_i} \leq C - \sum_{fu \in Fu(i)} c_{fu}$$

であれば新たに演算器 $fu_i$ を島 $i$ に配置できる．

### 2.2 データ転送表

RDR アーキテクチャを対象としたスケジューリングでは，各演算器の間のデータ転送時間をデータ転送表と呼ばれる 2 次元配列により管理する．今，島 $i_1$ に配置された演算器 $fu_1$ の演算結果を島 $i_2$ に配置されたローカルレジスタ $R(i_2)$ に格納することを考える．$T_{clk} \geq D_c(i_1, i_2) + d_{fu_1} + d_R + d_M$ となる場合，前述の通り演算実行と同じコントロールステップで $R(i_2)$ にデータを格納する．一方，$T_{clk} < D_c(i_1, i_2) + d_{fu_1} + d_R + d_M$ となる場合，データ転送のためだけにクロックサイクルを使用する．その際，データ転送には $\lceil \frac{D_c(i_i,i_2)}{T_{clk}} \rceil$ ステップかかるので，この値をデータ転送表に格納し，スケジューリングを行うときに参照する．したがって，演算器および配線の遅延見積もりを変化させてスケジューリング，バインディングを行う場合，データ転送表の生成方法を変更することで実現できる．

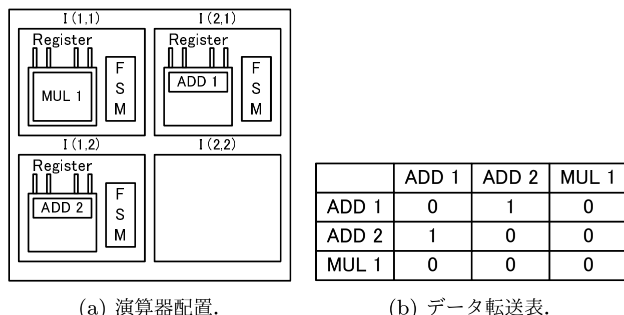
図 2a に示す演算器配置に対するデータ転送表を図 2b に示す．ADD1 と ADD2 の間のデータ転送は 1 コントロールステップとなっており，$I(0, 1)$ と $I(1, 0)$ の間のデータ転送に 1 コントロールステップかかる．

### 2.3 RDR アーキテクチャの製造時遅延ばらつき

提案手法では，合成フローにばらつき率 $v$ $(v > 1.0)$ を入力として与える．本稿で扱う製造時遅延ばらつきは，ばらつき率 $v$ を用いて以下のように定義する．

**定義 1.** 本稿で扱う製造時遅延ばらつきとは，演算器や配線，レジスタ，*MUX* において製造時に遅延が発生し，想定されていた遅延時間の $v$ 倍の遅延が発生することである． □

©2014 Information Processing Society of Japan




| | ADD 1 | ADD 2 | MUL 1 |
|---|---|---|---|
| ADD 1 | 0 | 1 | 0 |
| ADD 2 | 1 | 0 | 0 |
| MUL 1 | 0 | 0 | 0 |

(a) 演算器配置.　　(b) データ転送表.

図 2: データ転送表.

以上の定義のもと，以下に遅延ばらつき許容量を調整できる RDR アーキテクチャ向け高位合成問題を定義する．

**定義 2.** 遅延ばらつき許容量を調整できる *RDR* アーキテクチャ向け高位合成問題とは，*DFG* $G = (V, E)$, *RDR* アーキテクチャの島数 $N \times M$, 島の容量制約 $C$, 演算器ライブラリ, クロック周期制約 $T_{clk}$，ばらつき率 $v$ が与えられたとき，製造時遅延ばらつき発生時も動作し，なおかつ総コントロールステップ数を最小化するよう $G$ をスケジューリング・バインディングし，演算器を各島に割り当てることである． □

## 3　遅延ばらつき許容量を調整できる RDR アーキテクチャ向け高位合成手法

本章では遅延ばらつき許容量を調整できる RDR アーキテクチャ向け高位合成手法を提案する．3.1 節で提案手法を動機付け，3.2 節以降で遅延ばらつき許容量を調整できる RDR アーキテクチャ向け高位合成手法を提案する．

### 3.1　動機付け

製造時の遅延ばらつきに対応する上で重要なことは以下の 2 点である．

1. 従来手法である MCAS [2] を利用することで遅延ばらつき未考慮時のレイテンシは最小化できる．同様に遅延ばらつき考慮時にもレイテンシの最小化を図る．
2. もし遅延ばらつきが発生しない場合と発生した場合の 2 通りのスケジューリング，バインディングを行った場合，コントローラ面積や MUX の数が増大すると考えられる．したがって，2 通りのスケジューリング，バインディング結果を可能な限り類似させ，資源を共有させる必要がある．

RDR アーキテクチャ向け高位合成はスケジューリング，バインディング，アロケーション，フロアプランから構成される．以下に，遅延ばらつき許容量調整の考慮を組み込むステップの選択方針を示す．

- **スケジューリング/バインディング**
既存の RDR アーキテクチャ向け高位合成手法 MCAS [2] はレイテンシを最小化する手法であるため，遅延ばらつき未発生時，発生時のそれぞれのスケジューリング/バインディングについて各々独立して MCAS を用いる．また，遅延ばらつき許容量を変化させるため，ばらつき率を入力として与え，遅延ばらつき考慮時のスケジューリング，バインディングにおいて用いる．この後，遅延ばらつき未考慮時と考慮時の 2 通りのスケジューリング，バインディング結果を類似させることで (これを類似化と呼ぶ)，これら 2 つを同時に RDR アーキテクチャ上に実現した場合でもコントローラの規模の増大および MUX 数の増大を防ぐことができる．したがって，2 通りのスケジューリング，バインディングをした後に，類似化させるために再度スケジューリング，バインディングを行う (これをリスケジューリング，リバインディングと呼ぶ)．なおリスケジューリング，リバインディングについては，文献 [4] と同様のアルゴリズムを用いる．

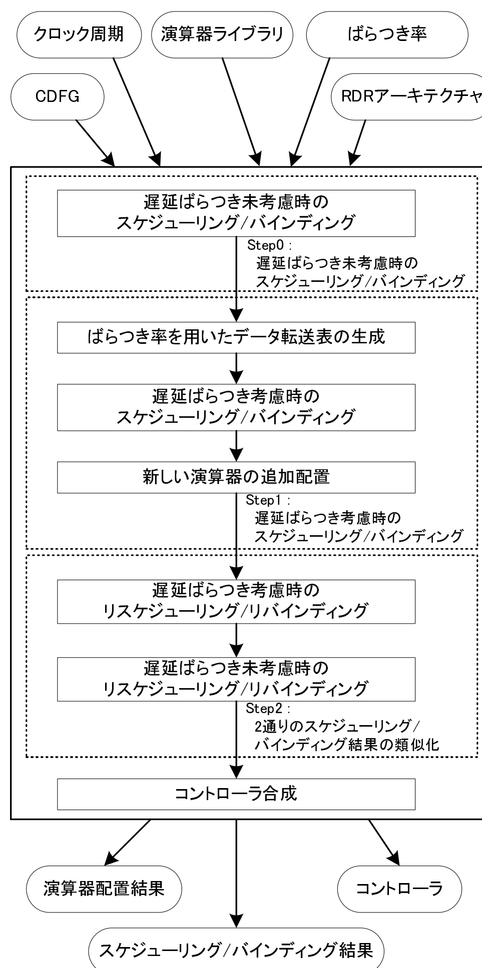



図 3: 合成フロー.

- **フロアプラン/アロケーション**
フロアプランを変更することによりレイテンシの増加を招く恐れがあるが，RDR アーキテクチャは規格化された島をベースとするため必ずしも全ての島に演算器が容量いっぱい配置されているわけでなく，適当な島に演算器を追加することにより遅延ばらつき発生時の性能劣化を防ぐことができる．したがって，与えられた演算器配置自体は変更しないが，空き領域に演算器を追加配置するものとする．

### 3.2　合成フロー

3.1 節をもとに，遅延ばらつき許容量を調整できる RDR アーキテクチャ向け高位合成手法を提案する．提案手法の合成フローを図 3 に示す．

合成フローへの入力は DFG とクロック周期，RDR アーキテクチャ，演算器ライブラリ，ばらつき率である．まず，MCAS [2] で遅延ばらつきを考慮しない場合

©2014 Information Processing Society of Japan

のスケジューリング，バインディングを行う (Step0)．続いて，ばらつき率を用いてデータ転送表を生成し，そのデータ転送表を元に遅延ばらつきを考慮したスケジューリング，バインディングを行う (Step1)．この際，各島の空き領域を計算し，追加の演算器を配置する．その後，2 通りのスケジューリング，バインディング結果の類似化を行う (Step2)．これにより，チップ面積を最小化する．最後にコントローラを合成し，最適な演算器配置およびスケジューリング，バインディング結果，コントローラを出力する．

### 3.3 遅延ばらつき考慮時のスケジューリング，バインディング (Step1)

Step1 では RDR アーキテクチャ上の演算器の配置および遅延ばらつき未考慮時のスケジューリング，バインディング結果を元に，遅延ばらつき考慮時のスケジューリング，バインディングを行う．Step1 のアルゴリズムを以下に示す．

(1.1) ばらつき率を用いたデータ転送表を生成する．
(1.2) 遅延ばらつき考慮時のスケジューリング，バインディングを行う．
(1.3) 空き領域に演算器を追加する．

(1.1) ではばらつき率を用いたデータ転送表を生成する．入力されたばらつき率 $v$ を元に遅延ばらつきを考慮したデータ転送表を生成することで，遅延ばらつき発生時でも動作するスケジューリング，バインディングを実現する．

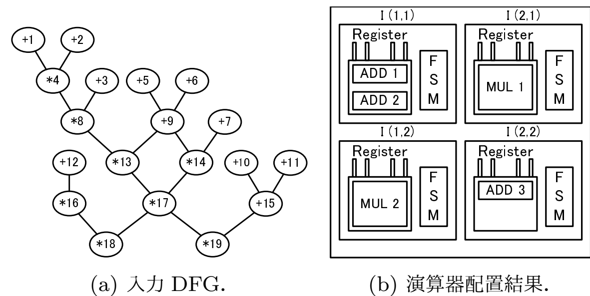
2 章で説明したとおり，遅延ばらつき未考慮時のデータ転送表を生成する際には $T_{clk}$ と $D_c(i_1, i_2) + d_{fu_1} + d_R + d_M$ を比較し，1 コントロールステップ内で別の島へのデータ転送を行うか否かを決定する．一方，遅延ばらつき考慮時のデータ転送表を生成する際には，$T_{clk}$ と $\{D_c(i_1, i_2) + d_{fu_1} + d_R + d_M\} \times v$ を比較する．これにより，遅延ばらつき未考慮時の $v$ 倍の遅延までを許容するスケジューリング，バインディングを実現するためのデータ転送表を生成できる．

(1.2) では遅延ばらつき未考慮時のスケジューリング，バインディングの結果得られた RDR アーキテクチャ上の演算器配置のもと，遅延ばらつき考慮時のスケジューリング，バインディングを行う．この際，(1.1) にて生成した遅延ばらつきを考慮したデータ転送表を利用する．スケジューリングアルゴリズムは，MCAS を用いる．これにより，遅延ばらつき未考慮時の $v$ 倍の遅延までを許容するスケジューリング，バインディング結果が得られる．

(1.3) では遅延ばらつき考慮時のレイテンシを最小化するため，空き領域に演算器の追加を行う．RDR アーキテクチャは全ての島の面積が等しいという性質上空き領域が発生する．この空き領域に演算器を追加することで遅延ばらつき考慮時の性能の向上を図る．

まず，入力されている DFG を走査し，追加の候補となる演算器を取得する．さらに，追加の候補となる演算器の面積も取得する．次に，各島の空き領域面積を計算する．その後，取得した演算器の候補から一つを選択し，いずれかの島に仮配置する．演算器を仮配置した後，データ転送表を再構築し，スケジューリング，バインディングを行い，総コントロールステップ数を求める．この操作を全ての演算器および島に対して総当たりで実行し，総コントロールステップ数を最小にする候補の演算器 1 つとこれを配置する島を求める．そのときの演算器の種類および配置する島を記憶する．

最後に，演算器追加前の総コントロールステップ数 $CS1$ と追加後の総コントロールステップ数 $CS2$ を比較する．$CS1 \leq CS2$ の場合，演算器を追加しても性能が向上しないので演算器を追加しない．$CS1 > CS2$ の場合，演算器を追加することでレイテンシが向上するので，記憶している演算器を記憶している島に配置する．演算器を追加した場合，さらに演算器を追加できないか確認する．演算器を追加しない場合，アルゴリズムを終了し，演算器配置を確定する．




(a) 入力 DFG.　(b) 演算器配置結果.

図 4: 入力 DFG と演算器配置結果.

| | ADD 1 | ADD 2 | ADD 3 | MUL 1 | MUL 2 |
|---|---|---|---|---|---|
| ADD 1 | 0 | 0 | 0 | 0 | 0 |
| ADD 2 | 0 | 0 | 0 | 0 | 0 |
| ADD 3 | 0 | 0 | 0 | 0 | 0 |
| MUL 1 | 0 | 0 | 0 | 0 | 0 |
| MUL 2 | 0 | 0 | 0 | 0 | 0 |

(a) 遅延ばらつき未考慮時.

| | ADD 1 | ADD 2 | ADD 3 | MUL 1 | MUL 2 |
|---|---|---|---|---|---|
| ADD 1 | 0 | 0 | 0 | 0 | 0 |
| ADD 2 | 0 | 0 | 0 | 0 | 0 |
| ADD 3 | 0 | 0 | 0 | 0 | 0 |
| MUL 1 | 1 | 1 | 1 | 0 | 1 |
| MUL 2 | 1 | 1 | 1 | 1 | 0 |

(b) 遅延ばらつき考慮時.

図 5: データ転送表.

**例 1.** 図 4a の DFG を 2 × 2 の RDR アーキテクチャ上にスケジューリング，バインディングする場合を考える．このとき，ばらつき率 $v = 1.5$ として入力したと仮定する．したがって，遅延ばらつき考慮時のスケジューリング，バインディング結果は，遅延ばらつき未考慮時の 1.5 倍の遅延まで許容できる結果となる．

加算器の遅延を $1ns$，乗算器の遅延を $2ns$，隣接した島同士の配線遅延を $0.1ns$，対角の島同士の配線遅延を $0.4ns$ とする．また，クロック周期を $3ns$ とする．

Step0 において，遅延ばらつき未考慮時のスケジューリング，バインディングを行う．この場合の演算器配置は図 4b となり，データ転送表は図 5a，スケジューリング，バインディング結果は図 6a となる．この例の場合，全ての演算器間の通信時間は 0 クロックサイクルとなり，データ転送表にも全て 0 を格納する．

Step1 において，遅延ばらつき考慮時のスケジューリング，バインディングを行う．まず，入力されたばらつき率を用いて，遅延ばらつき考慮時のデータ転送表を生成する．ばらつき率が 1.5 の場合を考えているため遅延ばらつき考慮時では，加算器の遅延を $1.5ns$，乗算器の遅延を $3ns$，隣接した島同士の配線遅延を $0.15ns$，対角の島同士の配線遅延を $0.6ns$ として見積もる．したがって，乗算実行後に別の島へとデータ転送する場合，1 コントロールステップ内で演算とデータ転送を行えず，遅延ばらつき考慮時のデータ転送表は図 5b となる．これを用いて行った遅延ばらつき考慮時のスケジューリング，バインディング結果を図 6b に示す．

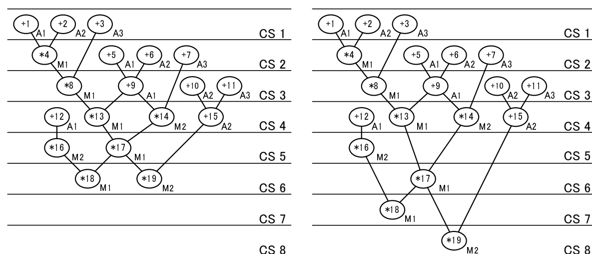
さらに，図 4b の演算器配置に演算器を追加した結果を図 7a に示す．また，その際に再構築した遅延ばらつき考慮時のデータ転送表を図 7c に示す．そして，それを用いて行った遅延ばらつき考慮時のスケジューリング，バインディング結果を図 7b に示す． □

### 3.4 2 通りのスケジューリング，バインディング結果の類似化 (Step2)

Step2 では遅延ばらつき未考慮時と考慮時のスケジューリング，バインディング結果の類似化を行う．類似化を行うことで，レジスタおよび演算器に与える制御信号が共有され，コントローラ面積および MUX 数の削減が期待できる．

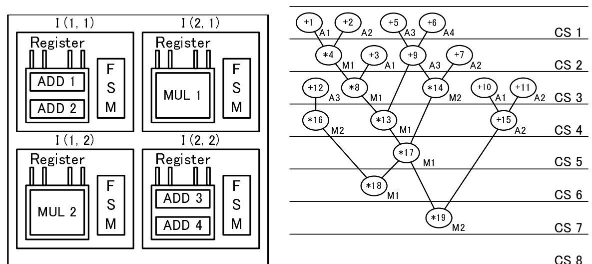
DFG $G = (V, E)$ において，ノード $v \in V$ を考える．遅延ばらつき未考慮時におけるノード $v$ を実行す

©2014 Information Processing Society of Japan




(a) 遅延ばらつき未考慮時. (b) 遅延ばらつき考慮時.

図 6: スケジューリング，バインディング結果.




(a) 演算器配置. (b) スケジューリング結果 (遅延ばらつき考慮時).

| | ADD 1 | ADD 2 | ADD 3 | ADD4 | MUL 1 | MUL 2 |
|---|---|---|---|---|---|---|
| ADD 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| ADD 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| ADD 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| ADD 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| MUL 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| MUL 2 | 1 | 1 | 1 | 1 | 1 | 0 |

(c) データ転送表.

図 7: 演算器追加後の様子.

るコントロールステップを $S(v)$，実行する演算器を $B(v)$ とする．一方，遅延ばらつき考慮時におけるノード $v$ の演算を実行するコントロールステップを $S_D(v)$，実行する演算器を $B_D(v)$ とする．したがって，スケジューリング，バインディング結果を類似化するとは，$S(v) = S_D(v)$ かつ $B(v) = B_D(v)$ となるノード $v$ の数を最大化することである．

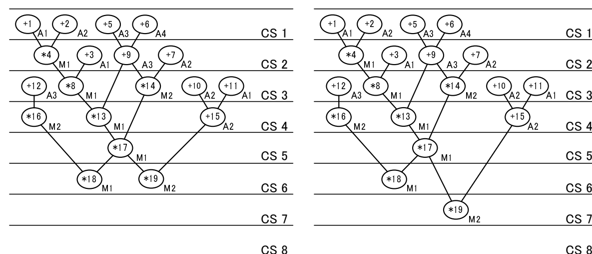
類似化のアルゴリズムは文献 [4] と同様のものを用いる．Step2 のアルゴリズムの概要を以下に示す．

(2.1) 遅延ばらつき考慮時のリスケジューリング，リバインディング．

(2.2) 遅延ばらつき未考慮時のリスケジューリング，リバインディング．

まず，遅延ばらつき未考慮時の総コントロールステップ数 $CS1$ と遅延ばらつき考慮時の総コントロールステップ数 $CS2$ を比較する．もし $CS1 = CS2$ の場合，遅延ばらつき考慮時のスケジューリング，バインディング結果のみを採用すれば良い．これにより，回路規模の増大を防ぎつつ遅延ばらつきへの耐性を向上させることができる．$CS1 < CS2$ の場合，2 通りのスケジューリング，バインディング結果を 1 つにまとめることはできない．したがって類似化を行い，回路規模の増大を抑制する．

**例 2.** 図 6a の遅延ばらつき未考慮時のスケジューリング，バインディング結果と図 7b の遅延ばらつき考慮時のスケジューリング，バインディング結果をリスケジューリング，リバインディングした場合を考える．今回の場合，遅延ばらつき未考慮時の総コントロールステップ数は 6 ステップ，遅延ばらつき考慮時の総コントロールステップ数は 7 ステップとなっている．したがって，2 通りのスケジューリング，バインディング結果を 1 つにまとめることはできず，類似化によって可能な限り類似したスケジューリング，バインディング結果に変更する．遅延ばらつき未考慮時のリスケジューリング，リバインディング結果を図 8a，遅延ばらつき考慮時ののリスケジューリング，リバインディング結果を図 8b に示す．2 通りのスケジューリング，バインディング結果が類似化されたことが確認できる． □




(a) 遅延ばらつき未考慮時. (b) 遅延ばらつき考慮時.

図 8: リスケジューリング，リバインディング結果.

以上の手順により，最適な演算器配置およびスケジューリング，バインディング解が得られる．最後に入力信号によって遅延ばらつき未考慮時および考慮時のスケジューリング，バインディング結果を切り替えられるようにコントローラを合成する．演算器配置とスケジューリング，バインディング結果，コントローラを出力してアルゴリズムを終了する．

## 4 計算機実験結果

提案手法を C++ 言語を用いて計算機上に実装した．計算機実験環境は，CPU が AMD Quad-Core Opteron 2360 SE 2.5$GHz$，メモリ容量が 16$GB$ である．対象アプリケーションとして DCT (演算ノード数 48，島 $2 \times 2$)，EWF (演算ノード数 102，島 $2 \times 2$)，FIR (演算ノード数 75，島 $2 \times 3$) を用いた．クロック周期は 2.0$ns$ とする．各演算器は 90$nm$ テクノロジノードのもとで 16$bit$ 幅と仮定する．各島の容量制約を 2，加算器の容量コストを 1，乗算器の容量コストを 2 とする．クロック配線遅延係数を $C_d = 1ns$ とする [7]．論理合成には Synopsys 社の Design Compiler を利用した．

各アプリケーションに対して以下の 4 つの手法を適用した．

MCAS: 従来の RDR アーキテクチャ向けの高位合成手法である MCAS [2] を適用する．遅延ばらつき未考慮時のデータ転送表のみを利用するため遅延ばらつきが大きくなった場合は対応できず，タイミングエラーが発生する．

Modified MCAS (1): 遅延ばらつきを考慮する上で最も単純な手法は，遅延のマージンを挿入することである．したがって，遅延ばらつき考慮時のデータ転送表を利用してスケジューリング，バインディングを行えば良い．この手法でもスケジューリング，バインディングには MCAS を利用する．今回はばらつき率 $v = 1.5$ としたときの遅延ばらつき考慮時のデータ転送表を利用した．したがって，実際の遅延ばらつきが未考慮時の 1.5 倍以下の場合は正常に動作する．

Modified MCAS (2): もう一つの単純な手法は，エラー検出可能なフリップフロップ [3, 6] を利用する手法である．RDR アーキテクチャ上のローカルレジスタをエラー検出が可能なものに置き換え，タイミングエラーが検出された場合は 1 コントロールステップ待つことでエラー訂正を行う．

©2014 Information Processing Society of Japan

表 1: 計算機実験結果.

| App. | 島数 | 演算器 | アルゴリズム | ばらつきの倍率毎の総 CS 数 | | | 追加した | 最大島面積 | CPU 時間 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | 1.0 倍 | ∼1.2 倍 | ∼1.5 倍 | 演算器 | $[\mu m^2]$ | $[sec]$ |
| DCT | $2 \times 2$ | ADD×4 MUL×2 | MCAS | 19 | NA | NA | - | 6932 | 100.54 |
| | | | Modified MCAS (1) [2] | 28 | 28 | 28 | - | 7311 | 100.17 |
| | | | Modified MCAS (2) [3, 6] | 19 | 23 | 30 | - | - | - |
| | | | 提案手法 ($v = 1.2$) | 19 | 19 | NA | - | 6480 | 97.73 |
| | | | 提案手法 ($v = 1.5$) | 19 | 27 | 27 | - | 7408 | 103.95 |
| EWF3 | $2 \times 2$ | ADD×3 MUL×2 | MCAS | 63 | NA | NA | - | 6480 | 108.56 |
| | | | Modified MCAS (1) [2] | 82 | 82 | 82 | - | 6407 | 109.84 |
| | | | Modified MCAS (2) [3, 6] | 63 | 78 | 96 | - | - | - |
| | | | 提案手法 ($v = 1.2$) | 63 | 65 | NA | ADD×1 | 6777 | 113.76 |
| | | | 提案手法 ($v = 1.5$) | 63 | 65 | 65 | - | 5969 | 114.03 |
| FIR | $2 \times 3$ | ADD×3 MUL×3 | MCAS | 35 | 35 | NA | - | 6587 | 127.42 |
| | | | Modified MCAS (1) [2] | 51 | 51 | 51 | - | 6966 | 123.31 |
| | | | Modified MCAS (2) [3, 6] | 35 | 35 | 56 | - | - | - |
| | | | 提案手法 ($v = 1.2$) | 35 | 35 | NA | - | 6587 | 127.51 |
| | | | 提案手法 ($v = 1.5$) | 35 | 35 | 45 | MUL×1 | 5417 | 129.80 |

この場合は，遅延ばらつき未考慮時のデータ転送表を利用して MCAS によりスケジューリング，バインディングを行い，遅延ばらつき発生時にタイミングエラーが検出される箇所に 1 コントロールステップ多く割り当てることで総コントロールステップ数を計算した．

**提案手法**: 提案手法を適用する．ばらつき率 $v = 1.2$ と $v = 1.5$ の 2 通りの場合を求めた．

比較項目としては，総コントロールステップ数，最大島面積，CPU 時間とした．遅延ばらつきは未考慮時の 1.2 倍と 1.5 倍のときを求めた．また，RDR アーキテクチャは全ての島が同じ大きさとなっているため，最大島面積の大きさが直接チップ面積に影響することから最大島面積によって比較している．

実験結果を表 1 に示す．提案手法は従来手法である Modified MCAS (2) と比較して，遅延ばらつき発生時の総コントロールステップ数を最大 32.3% 削減できることを確認した．したがって，提案手法は半導体の微細化に伴う配線遅延の拡大および製造時遅延ばらつきに有効な手法であると言える．

また，MCAS と比較して最大島面積や CPU 時間には大きな変化は見られなかった．RDR アーキテクチャの場合，特定の島に演算器やレジスタ，MUX が集まるとチップ全体の面積が大きくなり，複数の島に演算器やレジスタ，MUX が分散するとチップ全体の面積は小さくなる．実験結果より，提案手法を適用することで製造後調整機能を付加した影響よりも，レジスタや MUX の各島への割り当ての方がチップ面積に影響を与えていると考えられる．

## 5 おわりに

本稿では遅延ばらつき許容量を調整できる RDR アーキテクチャ向け高位合成手法を提案した．半導体の微細化技術の向上により配線遅延および製造時遅延ばらつきが問題となっており，提案手法では RDR アーキテクチャの従来の高位合成手法に遅延ばらつき許容量の調整可能な製造後遅延調整機能を組み込むことによって問題を解決している．また，入力されたばらつき率に応じて様々な大きさの遅延ばらつきに対応することができる．計算機実験の結果から提案手法は従来手法に比べ遅延ばらつき発生時の実行時間を最大 32.3% 削減できることを確認した．

今後の研究課題は，総コントロールステップ数の増大を防ぎつつ，よりロバストな設計を行える手法を考案することである．

## 謝辞

本研究は独立行政法人新エネルギー・産業技術総合開発機構 (NEDO) の先導的産業技術創出事業の支援を受けて行われた．

## 参考文献

[1] C. S. Amin, N. Menezes, K. Killpack, F. Dartu, U. Choudhury, N. Hakim, and Y. I. Ismail, “Statistical static timing analysis: how simple can we get?,” in *Proc. of the 42nd Annual Design Automation Conference*, pp. 652–657, 2005.

[2] J. Cong, Y. Fan, X. Yang, and Z. Zhang, “Architecture and synthesis for multi-cycle communication,” in *Proc. of 2003 International Symposium on Physical Design*, pp. 190–196, 2003.

[3] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, “Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation,” in *Proc. of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 7–18, 2003.

[4] Y. Hagio, M. Yanagisawa, and N. Togawa, “A delay-variation-aware high-level synthesis algorithm for RDR architectures,” *IPSJ Trans. on System LSI Design Methodology*, vol. 7, 2014.

[5] K. Kawamura, M. Yanagisawa, and N. Togawa , “A thermal-aware high-level synthesis algorithm for RDR architectures through binding and allocation,” *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E96-A, no. 1, pp. 312–321, 2013.

[6] Y. Shi, H. Igarashi, N. Togawa, and M. Yanagisawa, “Suspicious timing error prediction with in-cycle clock gating,” in *Proc. of 14th International Symposium on Quality Electronic Design*, pp. 335–340, 2013.

[7] S. Tanaka, M. Yanagisawa, T. Ohtsuki, and N. Togawa, “A fault-secure high-level synthesis algorithm for RDR architectures,” *IPSJ Trans. on System LSI Design Methodology*, vol. 4, pp. 150–165, 2011.

[8] H. Yoshida and M. Fujita, “An energy-efficient patchable accelerator for post-silicon engineering changes,” in *Proc. of the 9th International Conference on Hardware/Software Codesign and System Synthesis*, pp. 13–20, 2011.

©2014 Information Processing Society of Japan