社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2006－SLDM－127 (13)
2006／11／28

束データ方式による非同期式回路の動作合成手法の提案

濱田 尚宏† 小西 隆夫†† 齋藤 寛†† 米田 友洋††† 南谷 崇††††

† 会津大学コンピュータ理工学研究科 コンピュータシステム学専攻
†† 会津大学コンピュータ理工学部 コンピュータハードウェア学科
††† 国立情報学研究所
†††† 東京大学 先端科学技術研究センター

あらまし 本稿では，制約のあるC言語で記述されたアプリケーションの動作仕様から，束データ方式による非同期式論理回路を自動合成するための手法を提案する．ケーススタディとしてMPEG2のidctrow関数を合成し，提案手法を評価する．

Proposal of a Behavioral Synthesis Method for Asynchronous Circuits in Budled-data Implementation

Naohiro HAMADA†, Takao KONISHI††, Hiroshi SAITO††, Tomohiro YONEDA†††,
and Takashi NANYA††††

† Dept. of Computer Systems, The Graduate School of The University of Aizu
†† Dept. of Computer Hardware, The University of Aizu
††† National Institute of Informatics
†††† Research Center for Advanced Science and Technology, The University of Tokyo

Abstract In this paper, we propose a behavioral synthesis method for asynchronous circuits in bundled data-implementation which synthesizes an asynchronous logic circuit from a behavioral description written in a restricted C language. We evaluate the proposed method through a case study where the idctrow function in MPEG2 is synthesized.

1. はじめに

LSIの微細化に伴い，配線遅延のばらつきや消費電力の増加といった問題が顕著となってきている．回路全体をクロック信号によって制御する同期式回路では，クロックスキューによる同期の失敗，クロックツリーにかかる消費電力などの問題が今後ますます顕著となる．一方，ローカルなハンドシェーク信号によって回路を制御する非同期式回路では，クロック信号に纏わる上記のような問題がない．また，低消費電力，低電磁放射などの利点が挙げられる．しかしながら，非同期式回路では，ハザードのない回路が要求されるなど設計が困難であり，かつCADツールなどによる設計支援も非常に限られている．

本稿では，制限のあるC言語で記述されたアプリケーションの動作仕様から束データ方式による非同期式回路を動作合成する手法を提案する．束データ方式では，演算の実行時間は使用するリソースの最大遅延時間に依存するが，同期式回路のように演算の全てがクロックサイクル時間に縛られることはない．また，使用されるリソースのほとんどは，同期式回路のために設計されたものと同じである．そのため，二線方式などによる他の非同期式回路実装と比べ，回路面積が少なく，設計も容易である．

提案手法は，従来の動作合成手法[1]と同様に，スケジューリング，リソースアロケーション，制御回路合成を中心に，動作仕様に対応したRTL構造を生成する．規模の大きなアプリケーションにも対応できるように，従来の同期式回路で使われるスケジューリング手法を改良し，また，制御回路合成も状態空間からの論理合成ではなく，Q楽子とよばれる単純な制御のマッピングとした．

2. Data Flow Graph

Data Flow Graph(DFG)とは，アプリケーションにおけるデータフローを表したグラフであり，以下のように定義される．本稿では，DFGを提案手法の入力として用いる．

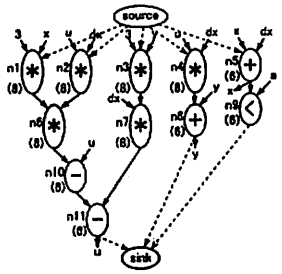

図 1 Data Flow Graph

$G = \langle N, E \rangle$

N, E はそれぞれ，ノードの集合，有向エッジの集合を表している．ノード $n_i \in N(i = 1, \ldots, m)$ は，演算の種類と遅延時間とでラベル付けされている．また，N には，参照ノードとして演算の開始と終了を表す source ノードと sink ノードが含まれる．演算は，あらかじめ用意されているリソースライブラリに含まれているリソースを使って実行される．なお，各リソースは面積と遅延時間をパラメータとして，リソースライブラリに登録されていると仮定する．ノードの遅延時間は，演算を実行するリソースの最大遅延時間に相当する．この値は，初期的に各演算に特定のリソースを割り当てることで得られる．

エッジ $e_{n_i,n_j} \in E$ は，ノード n_i からノード n_j への依存関係を表している．

図 1 に，ラベル付けされた DFG を示す．

3. 束データ方式による非同期式回路

束データ方式とは，非同期式回路におけるデータエンコーディング手法の一つである．束データ方式では，1 ビットのデータが 1 本の信号線に対応する．回路の制御には，ローカルなハンドシェーク信号を使用する．

本稿で扱う束データ方式による非同期式回路は，図 2 のようにデータパス回路と制御回路とで構成される．データパス回路は，各演算を実行する演算器，演算結果を保持するレジスタ，演算器とレジスタを共有するためのマルチプレクサで構成される．制御回路は，演算のスケジューリング結果より定められた回路の各状態毎に，データパス回路上のリソースを以下のように制御する．まず，状態 s の開始を表わす要求信号 req_s を生成する．その後，状態の終了を表わす応答信号 ack_s を待つ．束データ方式による実装では，リソースの最大遅延時間に相当する遅延素子を要求信号 req_s の信号線上に配置し，応答信号 ack_s を生成する．この応答信号 ack_s は，レジスタへの書き込み許可信号 enb_s ともなる．マルチプレクサや演算器は，制御信号 $sel_{s,t}(t = 1, \ldots, l)$ によって制御される．

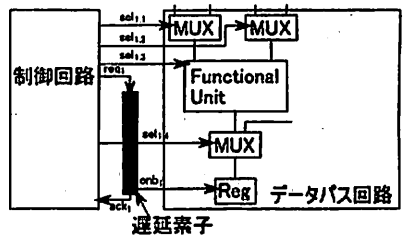

図 2 束データ方式による非同期式回路

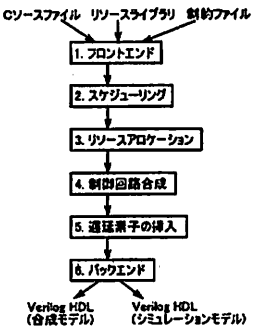

図 3 合成フロー

4. 束データ方式による非同期式回路の動作合成

4.1 合成フロー

提案手法の流れを図 3 に示す．提案手法の入力は，C 言語で記述されたアプリケーションとリソースライブラリと制約ファイルである．まず，フロントエンド部では，アプリケーションの動作記述より，DFG を生成する．その後，DFG に対して，スケジューリング，リソースアロケーション，制御回路合成を行う．制御回路の合成後，タイミング解析をもとに，遅延素子を挿入する．最後にバックエンド部で，合成された回路にたいする Verilog HDL 記述とシミュレーション用の Verilog HDL 記述を生成する．

4.2 入　　力

提案手法では，入力として C 言語で記述されたアプリケーションを用いるが，現在のところ以下の記述のみに制限される．

- 変数宣言
- 代入文
- 関数呼び出し

また，今後，以下を扱えるように，提案手法を拡張する予定である．

- 分岐
- ループ
- 三項演算

仮にもし与えられた入力記述中に，提案手法で扱えない記述が存在した場合，フロントエンドでは，それらを無視して DFG を生成する．

リソースライブラリは，面積と遅延でパラメータ化されたリソースから構成される．また，これらのリソースに対応した Verilog 記述も含まれる．制約ファイルは，動作合成時の設計制約である時間制約が含まれる．リソースライブラリのパラメータと制約ファイルは，XML 記述で与える．

4.3 フロントエンド

C 言語から中間表現である DFG を生成するフロントエンドとして，COINS [7] を利用する．COINS は，C 言語で記述され

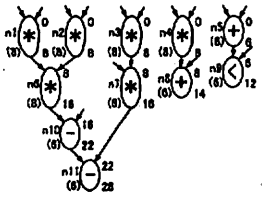

図 4 ASAP スケジュール

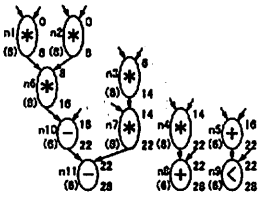

図 5 ALAP スケジュール

図 6 直前に終える演算からの開始時間の概算

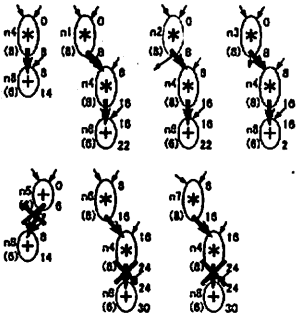

図 7 直前に終える二つの演算からの開始時間の概算

たアプリケーションを解析し，独自の中間表現である HIR(High level Intermediate Representation) を生成する．提案手法では，COINS によって生成された中間表現より，アプリケーションの構造を解析し，DFG を XML 記述で生成する．

4.4 スケジューリング

スケジューリングとは，各演算の開始時間を決定することである．提案手法では，Paulin らが同期式回路のために提案した Force-Directed Scheduling (FDS) アルゴリズム [2] を，東データ方式による非同期式回路に拡張した Asynchronous FDS (AFDS) アルゴリズム [3] を用いる．

FDS アルゴリズムは，与えられた時間制約の下，使用されるリソースの数が最適となるようにスケジューリングを行う時間制約アルゴリズムである．FDS アルゴリズムでは，時間制約を一定の時間間隔をもったコントロールステップ (以下ステップと呼ぶ) に分割し，リソースの使用量が最適となるように各演算をステップに割り当てる．ここで，ステップが一定の時間間隔を持つのは，クロックサイクルを想定しているためである．

一方，非同期式回路を対象とした場合，クロック信号が存在しないため，必ずしもステップを一定間隔にする必要はない．非同期式回路において演算は，前の演算の終了によって実行開始となるので，演算の開始時間を概算し，概算された開始時間を用いて，ステップを決めたほうが計算効率がよい．FDS アルゴリズムの計算時間は，ステップの 2 乗に比例するからである [3]．以下では，AFDS アルゴリズムの概要を説明する．

AFDS アルゴリズムでは，ステップの計算を行うに先だって，As Soon As Possible (ASAP) と As Late As Possible (ALAP) スケジュールを計算し，各演算の mobility を算出する．ASAP スケジュールでは，演算が実行可能となったら，直ちに演算が開始される．一方，ALAP スケジュールでは，与えられた時間制約の下，できるだけ遅い時間に演算が開始される．ASAP スケジュールによる開始時間と ALAP スケジュールによる開始時間の差が，各演算の mobility である．図 4 と図 5 に図 1 の DFG に対する ASAP スケジュールと ALAP スケジュールを示す．

各演算の mobility の算出後，ステップの計算を行う．ここで，ノード n_i で表わされた演算の直前に終わる演算を，依存関係があり直前に終了する演算と依存関係がなくリソースの共有が可能な演算の二種類とする．直前に終わる演算の終了時間が，ノード n_i で表わされた演算のひとつの開始時間候補となる．直前に終わる演算の終了時間が複数存在することもあるので，ノード n_i の開始時間候補の集合を S_i とする．同様に，直前に終わる演算についても，開始時間候補を求めていく．このようにして開始時間を概算すると，source ノードまで辿っていく可能性がある．しかしながら，多くの場合，source ノードまでたどったとしても，新たな開始時間候補を得られない可能性がある．従って，その演算から辿るノードを途中で打ち切ったとしても，スケジュールを行うのに十分な開始時間候補を得ることが可能であり，またそのほうが現実的でもある．最終的に，各演算についての開始時間候補の集合 S_i に対して和集合をとり，和集合の各要素をステップ $c_s(s = 0 \ldots k)$ とする．

ノードの開始時間の概算を，図 1 のノード n_8 を例に説明する．ノード n_8 の開始時間を概算するために，まず直前に終える可能性のあるノードから開始時間候補の集合 S_8 を計算する．ノード n_8 の直前に終わる可能性のあるノードは，ノード n_4 とノード n_5 である．従って，ノード n_4 とノード n_5 の ASAP における終了時間が，ノード n_8 の開始時間候補となる．しかしながら，ノード n_5 の ASAP における終了時間は，ノード n_8 の ASAP における開始時間よりも早いため (mobility の範囲外) 開始時間候補とはならない．従って，この時点でのノード n_8 の開始時間候補の集合 S_8 は $\{8\}$ である．図 6 に，その様子を示す．次に，ノード n_8 の直前に終わるノード (ノード n_4 とノード n_5) に対しても，同様に開始時間候補の集合を計算し，それらからノード n_8 の開始時間候補の集合 S_8 を計算する (図 7)．先ほどと同様に，ノード n_8 の mobility の範囲外となるものは，開始時間候補とはならない．この結果，S_8 は $\{8, 16\}$ となる．

最終的に，直前に終えるノードを 4 つまでたどった際に得られた各ノードの開始時間候補の集合に対する和集合は $\{0, 6, 8, 12, 14, 16, 18, 20, 22\}$ となるので，9 個のステップが生成される．なお，FDS アルゴリズムと違い，ステップ間隔は一定ではない．

ステップの計算後，各ノードに対して，演算が実行されるステップの範囲を表した Time frame を以下のように計算する．

$ALAP$ における終了時間 $-$ $ASAP$ における開始時間

次に，Time frame を用いて，各ノード n_i が，その開始時間候補の一つであるステップ c_s にスケジュールされる確率を求める．その後，Distribution Graph(DG) と呼ばれる，各ステッ

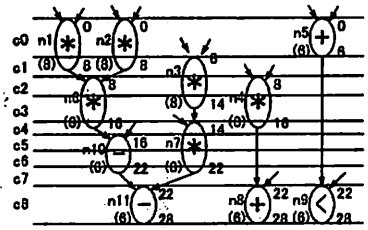

図 8　図 1 のスケジューリング結果

プ毎に利用されるリソース数の見積もりを表したグラフを計算する．

DG の計算後，各ノードの開始時間候補にたいし，Self-force と呼ばれるコスト関数を計算する．Self-force は，ノード n_i が，その開始時間候補の一つであるステップ c_s にスケジュールされたとき，リソースの利用が全体でどれだけバランス良くなったかを表す．Self-force の値が正のときには，必要となるリソース数が増加することを示し，負のときには，必要となるリソース数が減少することを示す．

一方，あるノード n_i を，その開始時間候補の一つであるステップ c_s にスケジュールした場合，その直前のノード，または直後のノードのスケジュールが一意に決まる場合がある．この場合，直前のノード，または直後のノードがスケジュールされる開始時間候補にたいする Self-force をノード n_i の Self-force に加算する．

Self-force の計算後，Self-force が最小となるノード n_i を当該ステップ c_s にスケジュールする．以上の計算を，全ての演算がスケジュールされるまで繰り返し行う．

図 8 は，図 1 の DFG に対して，AFDS アルゴリズムを適用したときのスケジューリング結果である．

4.5　リソースアロケーション

リソースアロケーション (以下アロケーションと呼ぶ) では，各演算に対して演算器を，各変数に対してレジスタを割り当てる．ここで変数 $v_a(a = 0, \ldots, u)$ とは，各演算の出力と外部入力を指す．提案手法では，始めに各演算に対して演算器を割り当て，ついで各変数に対してレジスタを割り当てる．マルチプレクサは，演算器とレジスタの割り当て後，対応するノードから入力情報を収集し，その情報を基に割り当てを行う．

アロケーションに先立って，各演算と各変数に対して，ライフタイムを計算する．演算のライフタイムは，演算の実行開始時間から終了時間までを指し，変数のライフタイムは，その変数に値が書き込まれてからその変数が最後に使用されるまでの時間を指す．

提案手法ではアロケーションに，Clique Partitioning アルゴリズムを用いる．Clique Partitioning アルゴリズムでは，まず始めに，Compatibility Graph(CG) と呼ばれるグラフを生成する．演算器の割り当ての場合，初期的に，DFG のノードを CG のノードとする．一方，レジスタの割り当ての場合，初期的に，DFG のノードと外部入力を CG のノードとする．ノードを生成した後，CG のノード間にエッジを挿入する．演算器

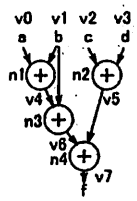

図 9　DFG

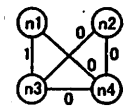

図 10　図 9 の演算に対する CG

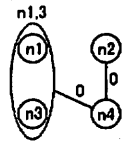

図 11　一つ目の clique 作成後の CG

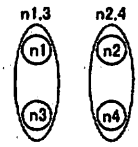

図 12　最終的に得られた CG

の割り当ての場合，次の条件のいづれかを満たすとき，ノード n_i とノード n_j の間にエッジが挿入される．

- ノード n_i に対する演算とノード n_j に対する演算が排他的に実行される場合
- ノード n_i に対する演算とノード n_j に対する演算が同じ演算器を使用する場合

一方，レジスタの割り当ての場合，次の条件を満たすとき，エッジが挿入される．

- 変数 v_a と変数 v_b のライフタイムに重なりがない場合

以下の操作は，演算器割り当てとレジスタ割り当てで共通である．

エッジの挿入後，各エッジに重みを付加する．この重みは，共通隣接ノードとエッジの両端のノードに共通する入出力の数から計算する．エッジ $e_{i,j}$ の両端のノード n_i とノード n_j に対し，ノード k とのエッジ $e_{k,i}$ とエッジ $e_{k,j}$ が存在する場合，ノード k をノード n_i とノード n_j の共通隣接ノードという．また，エッジ $e_{i,j}$ の両端のノード n_i とノード n_j に共通する入出力の数を計算する．これらより，ノード n_i とノード n_j の共通隣接ノード数を共通する入出力の数で乗算した結果をエッジ $e_{i,j}$ の重みとする．エッジの重みが大きい場合，必要となるマルチプレクサの数が少なくなる可能性がある．

次に，CG の中で，最も重みの大きいエッジを探索する．探索後，得られたエッジ $e_{i,j}$ の両端のノード n_i とノード n_j の共通隣接ノードの集合を求める．その後，ノード n_i とノード n_j に繋がっているエッジ全てとノード n_i とノード n_j をグラフから削除する．エッジとノードを削除した後，ノード n_i とノード n_j を一つのノード $n_{i,j}$ として CG に挿入し，共通隣接ノードの集合に含まれるノードと新たに作成したノード $n_{i,j}$ の間にエッジを挿入する．なお，新たに得られたノードを clique と呼ぶ．その後，新たに作られたエッジの重みを計算する．最終的に，CG 上にエッジがなくなるまで上記の操作を繰り返す．

ここで，図 9 を例に，Clique Partitioning アルゴリズムを説明する．まず，図 10 に，図 9 に対応する CG を示す．この例では，ノード n_1 とノード n_2 は同時に演算を行うため，ノード n_1 とノード n_2 の間にはエッジが挿入されない．一方，ノー

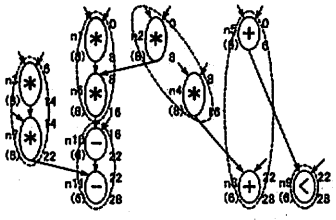

図 13　図 1 に対する演算器の割り当て結果

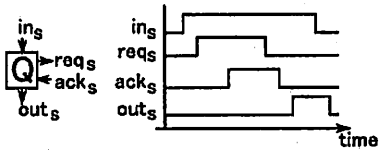

図 14　Q 素　子

ド n_1 とノード n_3，ノード n_1 とノード n_4，ノード n_2 とノード n_3，ノード n_2 とノード n_4，ノード n_3 とノード n_4 の間にはそれぞれエッジが挿入される．次に，各エッジについて共通隣接ノードを求める．$e_{1,3}$ では，ノード n_1 とノード n_3 はそれぞれノード n_4 と接続されている．よって，$e_{1,3}$ の共通隣接ノードは 1 である．同様に各エッジ毎に共通隣接ノードを求める．次に，二つのノードに共通する入出力の数を求める．これは，図 9 より簡単に求められる．ここまでの結果より，図 10. のようなエッジの重みが計算される．重みの計算後，重みが最大となっているエッジを探索する．この例では，$e_{1,3}$ の重みが最大となっているので，$e_{1,3}$ が選ばれる．次に，$e_{1,3}$ の共通隣接ノードを求め，ノード n_1 とノード n_3 に接続されている全てのエッジを削除し，ノード n_1 とノード n_3 も削除する．新たにノード $n_{1,3}$ というノードを追加し，先ほど求めた共通隣接ノードとの間にエッジを挿入し，エッジの重みを再計算する．この時点での CG を図 11 に示す．同様の処理をグラフ上からエッジが無くなるまで続ける．図 12 に最終的に得られるグラフを示す．

最終的に得られた CG から，ノードの部分集合が得られる．各部分集合に対して，一つの演算器やレジスタが割り当てられる．割り当て後，演算器やレジスタに割り当てられた演算，変数との対応を調べることで，各演算器と各レジスタの入力に必要なマルチプレクサを計算する．

演算器，レジスタ，マルチプレクサのアロケーション後，入力となる C 言語で記述されたアプリケーションに対応したデータパス回路が生成される．図 13 は，図 1 の DFG にたいして，演算器を割り当てた結果を表わす．図 13 で破線に囲まれたノードは同じ演算器へと割り当てられる．

4.6　制御回路合成

制御回路合成では，スケジューリングとアロケーションのあと得られたデータパス回路を制御する回路を生成する．まず，スケジューリングによって得られたステップより，以下の条件を満たすものを制御回路における状態 s とする．

- 開始可能となる演算が存在

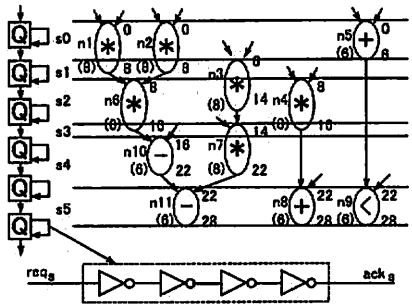

図 15　Q 素子の割り当てと遅延素子

表 1　リソースパラメータ

Name	Area (# of Slices)	Delay (ns)
add	8	6
sub	8	6
shft	2	4
mult	10	8

次に，各状態 s に対して，一つの Q 素子を割り当てる．こうした方針を採用することによって，状態数が多くなるようなアプリケーションでも，短時間で制御回路を合成することが可能となる．

Q 素子の入出力と動作は，以下の通りである．入力信号は，状態の開始を表す in_s 信号とデータパス回路側からの演算の終了を表わす ack_s 信号であり，出力信号は，データパス回路に対する要求信号となる req_s 信号と状態の終了を表す out_s 信号である．また，Q 素子の動作は以下の通りである．はじめに，前の状態の終了とともに，in_s が 1 となる．次にデータパス回路を動作させるために，要求信号 req_s を 1 とする．データパス回路から応答信号 ack_s が 1 として戻ってきたら，要求信号 req_s を 0 とする．最後に，応答信号 ack_s が 0 として戻ってきたら，状態の終了を表す out_s 信号を 1 とする．図 14 に，Q 素子のインターフェースと動作を示す．

制御回路の生成後，各状態で実行される演算を制御するために，要求信号 req_s より，マルチプレクサや演算器の制御信号 $sel_{s,t}$ を生成する．

4.7　遅延素子挿入

割り当てられた演算器，レジスタ，マルチプレクサの遅延時間より，各状態に必要となる時間を概算する．その後，概算された値を基に，各状態毎に遅延素子を図 15 のような論理ゲートのチェーンとして生成する．なお，レイアウト後に判明する配線遅延の影響を考慮して，概算された値に幾分かのマージンを加える．

遅延素子の入力は req_s で，出力は ack_s である．なお，ack_s よりレジスタへの書き込み信号 enb_s が生成される．レジスタへのデータの書込みは，enb_s が 0 のときに行われる．

4.8　出　　力

提案手法の出力は，合成された回路とその回路に対応したシミュレーションモデルである．これらは，Verilog HDL 記述で出力される．そうすることで，バックエンドツールやシミュ

表 2 動作合成の結果

時間制約 (ns)	ノード数	状態数	add	sub	shft	mult	reg	スケジューリング時間 (s)	アロケーション時間 (s)	その他 (s)	合成時間 (s)
60	61	16	5 (4)	4 (4)	6 (6)	5 (5)	29 (28)	0.390	257.234	3.266	260.890
90	61	20	4 (4)	3 (3)	6 (6)	4 (4)	29 (27)	0.594	268.656	3.141	272.391
120	61	24	4 (4)	3 (3)	6 (6)	2 (2)	27 (26)	0.860	256.453	3.015	260.328

レーションツールとの親和性を図ることができる．なお，合成された回路の中では，Q 素子と遅延素子の部分でフィードバックループを形成している．フィードバックループが入ると，論理シミュレータは，出力を生成するときにどの時点でのフィードバックの値を使ってよいか認識できなくなってしまい，ただしいシミュレーション結果を表示することができない．そのため，シミュレーションモデルでは，フィードバックループに一定の遅延が付加されている．

4.9 機能検証

出力されたシミュレーションモデルと Verilog HDL 用の論理シミュレータを用い，合成された回路の機能検証を行うことが可能である．

5. ケーススタディ

提案手法を用いて，MPEG2 の idctrow 関数を合成し，合成された回路について評価を行う．idctrow 関数には条件分岐が存在するが，今現在の提案手法では条件分岐が扱えないため，その部分を削除し回路を合成した．提案手法は Java を用いて実装され，Pentium4 プロセッサ (3GHz) 一つと 1GB のメモリを搭載した Windows マシンを用いて合成を行った．

表 1 にケーススタディで用いたリソースの各面積，遅延時間を示す．これらの値は，Verilog HDL で各リソースの動作を記述し，Xilinx 社製の ISE Service Pack [5] を用いて論理合成を行った結果得られた値である．

表 2 に，時間制約を変化させた場合の，合成された回路で使用される演算器数，レジスタ数，スケジューリング時間，アロケーション時間，その他合成にかかった時間を示す．また，リソース数のうち括弧で囲まれた数字は，最小リソース数を示している．

次に，合成された回路に対して，Verilog HDL でテストパターンを記述し，Mentor Graphics 社の ModelSim [6] を用いてシミュレーションを行った．動作の正しさを確認するために，Verilog HDL で用いたテストパターンを C 言語で記載し，C 言語レベルでシミュレーションを行ったものと結果を照合した．

提案手法を用いることによって，idctrow のような実設計で利用される例を，数分で束データ方式による非同期式回路として実現することができる．しかしながら，合成時間を見れば分かるが，アロケーションで多大な時間を消費している．これは，エッジの重みの計算に原因があると考えられる．また，アロケーションのなかでも，特にレジスタを割り当てるときに多大な時間を要するのがわかる．これは，レジスタに対して CG を生成したときに，CG の規模が大きくなってしまうためである．このため，今後の課題として，エッジの重みの計算とレジスタ割り当てを改善することが考えられる．

合成された回路は，利用しているアルゴリズムの性質より，最適解というよりはむしろ，それに近い値となっている．実際，ケーススタディでは，加算器とレジスタは，それぞれ最小数より一つ多く割り当てられてしまった．今後，提案手法の評価を通じて，提案手法の改善を図る．

6. ま と め

本稿では，制限された C 言語より，束データ方式による非同期式回路を自動合成する手法を提案した．提案手法を用いることによって，アプリケーションに応じた非同期式回路を容易に得ることが可能となる．今後の課題として，言語上の制約の緩和に伴う動作合成手法の変更，パイプライン化された回路の動作合成手法の提案を行う．また，アロケーションの高速化を検討する．

謝 辞

本研究は，文部科学省科学技術研究費補助金 若手研究 (B)(課題番号 18700047) の研究助成による．

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. G. Paulin and J. P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," IEEE Transactions on Computer-Aided Design, vol.8, no.6, pp.661–679, June 1989.

[3] 齋藤，米田，南谷，"Force-Directed Scheduling 手法の非同期式回路への適用と評価"，DA シンポジウム，pp.37–42, 2005.

[4] 齋藤，米田，南谷，"Integer Linear Programming を用いた束データ方式による非同期式回路のスケジューリング"，DA シンポジウム，pp.43–48, 2006.

[5] XILINX Inc, ISE ServicePack 8.1i http://www.xilinx.com

[6] Mentor Graphics Corp, ModelSim http://www.mentor.com

[7] COINS-project, A compiler infra structure http://www.coins-project.org