7-19

演算器順序深度削減指向テスト容易化バインディング法

日大生産工(院)　○長 孝昭　　日大生産工　細川 利典

1. はじめに

近年，半導体技術の進展に伴い，大規模集積回路（Large Scale Integration: LSI）の回路規模や複雑度が飛躍的に増大し，いくつかの問題が発生している．その一つとして，LSI 設計の生産性に関するものが挙げられる．現在，レジスタ転送レベル（Register Transfer Level: RTL）での LSI 設計が主流であるが，LSI の大規模化により，設計が非常に困難になってきている．LSI 設計が困難になると，LSI の設計生産性の低下や，設計コストの増加につながってしまう．これを回避するため，より高位レベルでの設計が提案されてきた[1]．現在，RTL より高位レベルである動作レベルで設計された回路を RTL に変換する技術である動作合成[2]が注目を浴びている．近年，RTL 回路の面積や性能を最適化するための動作合成アルゴリズム[3, 4, 5, 6]が数多く提案され，動作合成による LSI 設計が一部で実用化されている．

LSI 設計に関する問題に解決策が見出されている一方，LSI テストに関する問題も存在する．設計，製造された LSI は，不良品であるか否かのテストを行い，良品であると判定された LSI のみを出荷しなければならない．現在，LSI のテスト容易化設計として，高い故障検出効率を得るテストパターンを容易に生成することができるフルスキャン設計[7]が主流となっている．しかしながら，フルスキャン設計を施した回路は，LSI の記憶素子（レジスタ）にスキャンチェインを挿入することによる面積オーバーヘッドや，テスト長の増大が問題となる．よって，フルスキャン設計を施さず，テスタビリティを向上させる手法への期待が高まっている．通常，フルスキャン設計を施していない LSI に対し，自動テストパターン生成ツール（Automatic Test Pattern Generator: ATPG）を用いてテストパターンを生成し，高い故障検出効率を得ることは困難である．フルスキャン設計を施さず，テスタビリティを向上させる方法には，動作合成時に，テスト容易化を考慮に入れたアルゴリズムとして，外部入力から外部出力までのレジスタの段数（順序深度）を削減することにより，テスト容易化を実現する手法が提案されている[8]．本論文では，[8]の手法をベースに，故障検出効率およびテスト生成時間に着目し，さらなるテスタビリティ向上手法を提案する．

2. 動作合成

動作合成とは，C 言語や SystemC[9]などのプログラミング言語でハードウェアの動作を表現した動作記述を，VHDL[10]や Verilog-HDL[11]といったハードウェア記述言語（Hardware Description Language: HDL）によって記述された RTL 回路記述に変換するものであり，その処理内容は大きくわけると 4 つのフェーズにわかれる[2]．各フェーズはそれぞれ，グラフ生成，スケジューリング，バインディング，RTL 回路記述生成となっており（図 1），入力として動作記述ファイルおよび使用可能な演算器やレジスタ（リソース）の数を入力する必要がある．グラフ生成というフェーズでは，与えられた動作記述を変換し，グラフで表現する．グラフには，CDFG（Control Data Flow Graph）[2]や ADD（Assignment Decision Diagram）[11]などが存在する．本論文では，グラフとして一般的に用いられている CDFG を用いて説明する．CDFG における頂点は演算操作を、辺は変数を表す．スケジューリングというフェーズでは，各演算操作の依存関係を保ちつつ，リソース制約の条件の下で各時刻に演算操作を割当てる．また，リソースの制約が与えられていない場合，スケジューリングのフェーズで，CDFG に現れる演算操作を実現するためのリソース数を決める必要がある．バインディングというフェーズでは，スケジューリングされた DFG（Scheduled Data Flow Graph: SDFG）を基に，各演算操作や変数に具体的な演算器やレジスタを割当てる．RTL 回路記述生成では，結線を実現した RTL データパスと制御信号を生成するコントローラ（Finite State Machine: FSM）を生成する．なお，本論文で扱う例は，説明の簡略化のため，CDFG ではなく DFG（Data Flow Graph）を用いる．

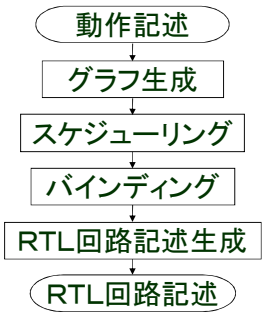

図 1. 動作合成の流れ

2.1. スケジューリング

スケジューリングには，可能な限り早い時刻に演算操作を割当てるスケジューリング手法（As Soon As Possible: ASAP）[13]，可能な限り遅い時刻に演算操作を割当てるスケジューリング手法（As Late As Possible: ALAP）[13]が最も基本的な手法として提案されている．ASAP および ALAP では，使用リソース数を考慮せず，演算操作の実行時刻しか考慮していない．よって，無駄な面積オーバーヘッドが存在している可能性がある．そこで，面積効率のよい SDFG を得るアルゴリズムとして，List スケジューリング[13]や

A binding method for testability based on resource sequential depths reduction

Takaaki CHO, Toshinori HOSOKAWA

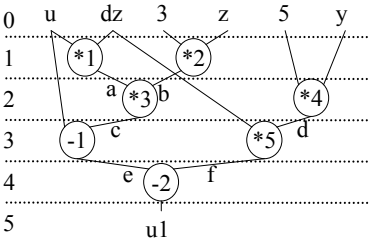

図 2. SDFG

FDS（Force Directed Scheduling）[6, 13]といったヒューリスティックなスケジューリングが提案されている．本論文では，FDS を用いている．

2.2. バインディング

バインディングは，実際に使用するリソースへ演算操作や変数を割当てるフェーズであるため，生成される RTL 回路の面積やテスタビリティに影響を与えやすい．よって，手法次第で順序深度やマルチプレクサ数などを削減し，面積の削減やテスタビリティの向上を図ることが可能である．なお，ここで扱う順序深度とは，外部入力から外部出力までに最低限通らなければいけないレジスタの段数のことを言う.本論文では，テスタビリティの向上を目的としているため，面積最小化を保ちつつテスタビリティの向上を図るバインディングを行う必要がある．ここで，テスト容易化アルゴリズムの一つである順序深度削減を指向したバインディングについて述べる．順序深度の削減を実現するには，入力変数が割当てられたレジスタ（入力レジスタ）や，出力変数が割当てられたレジスタ（出力レジスタ）に，可能な限り内部変数を割当て，入力レジスタから出力レジスタまでの順序深度を削減するという指針でバインディングを行う[8]．図 2 の SDFG に対し，テスタビリティを考慮せずに，使用リソース数最小を実現するレフトエッジアルゴリズム（Left-Edge Algorithm: LEA）[2] でバインディングを行った場合（図 3），レジスタに割当てられた変数はそれぞれ R1＝{u, e, u1}，R2＝{dz, f}，R3＝{z, a, c}，R4＝{y, d}，R5＝{b} となり，演算器はそれぞれ減算器（sub）1＝{-1, -2}，乗算器（mlt）1＝{*1, *3, *5}，mlt2＝{*2, *4} となる．順序深度は，【外部入力→外部出力 : 順序深度】のように記述するものとする．よって，順序深度は u→u1 : 1，dz→u1 : 2，z→u1 : 2，y→u1 : 3 となる．また，順序深度削減指向のバインディング[8]を行った場合（図 4），レジスタに割当てられた変数はそれぞれ R1＝{a, c, f, u1}，R2＝{u, e}，R3＝{dz}，R4＝{z, b}，R5＝{y, d} となり，演算器はそれぞれ sub1＝{-1, -2}，mlt1＝{*1, *4, *5}，mlt2＝{*2 *3} となる．よって，順序深度は u→u1 : 2，dz→u1 : 2，z→u1 : 2，y→u1 : 2 となる．結果，この例の結果では，順序深度削減指向バインディングを適用したことにより，順序深度の最大値が 3 から 2 に削減できた．このように，バインディングによって順序深度を削減することが可能であり，その結果テスタビリティが向上することが報告されている[8].

3. テスト容易化バインディング

本論文では，順序深度削減指向バインディング[8]

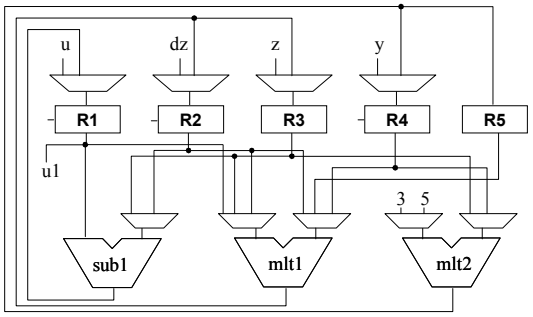

図 3. LEA 適用データパス

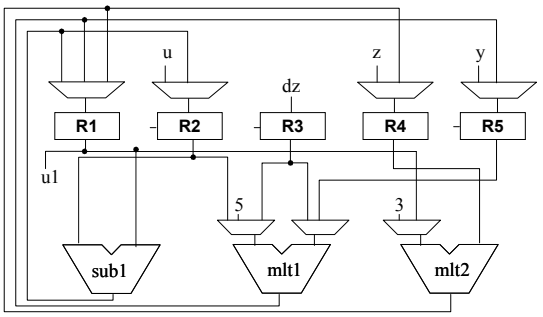

図 4. 順序深度削減法適用データパス

をベースに，さらなるテスタビリティ向上を目的とする．本論文では，以下の 2 つの戦略に着目した改善を行う．戦略 1 として，演算器順序深度という新たなテスタビリティの評価尺度を提案する．戦略 2 として演算器順序深度削減のためのバインディング手法を提案する．

3.1. 戦略 1

従来は順序深度にのみ着目し，それを削減することをテスタビリティ向上のための指針としてバインディングを行うものであった[8]．しかしながら，順序深度のみ削減するという評価尺度では外部入力から外部出力までのレジスタの段数のみを考慮しており，どの演算器を経路としたレジスタの段数なのかが考慮されていない．これでは，順序深度計算時に経路として扱われなかった演算器が存在してしまう．その結果，その演算器に関してはテスタビリティが考慮されていないため，順序深度が少ない回路でも，テスト困難な演算器が存在している可能性がでてきてしまう．そこで，各演算器に着目した新たな評価尺度として演算器順序深度を提案する．演算器順序深度とは，外部入力から演算器の各入力までのレジスタの最小段数と演算器の出力から外部出力までのレジスタの最小段数と定義し，演算器 M＝（α，β，γ）と表現する．αは，外部入力から演算器 M の左入力までのレジスタの最小段数を，βは，外部入力から演算器 M の右入力までのレジスタの最小段数を，γは演算器 M から外部出力までのレジスタの最小段数を表す.

図 5 に演算器順序深度削減指向バインディングにおける演算器バインディングのアルゴリズムを示す．2 行目では，各演算操作における順序深度（演算操作順序深度）を計算する．演算操作順序深度とは，外部入力から演算操作の各入力までの経路に存在する変数の最小数と演算操作の出力から外部出力までの経路に存在する変数の最小数と定義し，演算器順序深度と同様に表す．ここで，外部入力が定数であった場合，値の

```
1: ResourceBinding{
2:   Initialization_OperationSequentialDepth;
3:   while(O != φ){
4:     op = Choice_Operation(O);
5:     ResourceSharing(op);
6:     Update_OperationSequentialDepth;
7:     Update_ResourceSequentialDepth;
8:   }
9: }
```

図 5. 演算器バインディングアルゴリズム

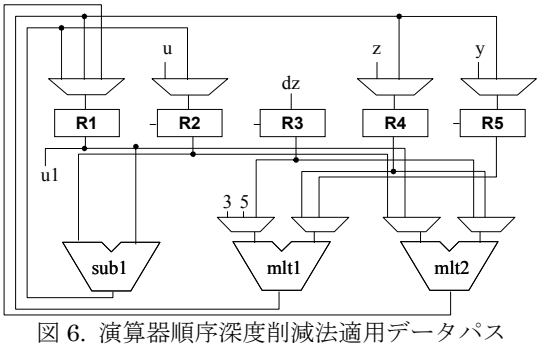

図 6. 演算器順序深度削減法適用データパス

設定が困難な入力と考え，演算操作順序深度の値を∞に設定する．3 行目から 8 行目までのループでは，演算器へ未割当ての演算操作集合 O が空になるまで処理を繰り返す．4 行目にて，O から演算操作を一つ選択する（op）．なお，選択される演算操作 op は実行時刻の遅い演算操作から優先的に選択する．5 行目では，op を演算器に割当てる．割当てる演算器は，演算操作が割当てられていない演算器を優先して選択する．また，すでに演算操作が割当てられている演算器しかない場合は，演算操作順序深度が低い演算操作が割当てられている演算器，すなわち演算器順序深度が低い演算器を優先して選択する．6 行目では，op を演算器に割当てることによって変化した演算操作順序深度を更新する．7 行目では，演算操作順序深度更新に伴い変化した，演算器順序深度の更新を行う．演算器バインディングが終了した後，レジスタバインディングを行う．レジスタバインディングには，順序深度削減指向バインディングベースのアルゴリズムを用いるが，紙面の都合で割愛する．

以上のアルゴリズムを適用し，この評価尺度の効果を，図 2 に示す SDFG を例に説明する．順序深度のみを考慮したバインディング結果は，2.2 章で示した通り，R1＝｛a, c, f, u1｝，R2＝｛u, e｝，R3＝｛dz｝，R4＝｛z, b｝，R5＝｛y, d｝となり，演算器はそれぞれ sub1＝｛-1, -2｝，mlt1＝｛*1, *4, *5｝，mlt2＝｛*2 *3｝となる．ここで，mlt2 について着目する．mlt2 において，外部入力から左入力までのレジスタの最小段数は 2, 外部入力から右入力までのレジスタの最小段数は 1, 出力から外部出力までのレジスタの最小段数は 1 となり，演算器順序深度は mlt2＝(2, 1, 1)と表現する．よって，演算器順序深度は sub1＝(1, 2, 1)，mlt1＝(1, 1, 1)，mlt2＝(2, 1, 1)となる（図 3）．同様の SDFG に演算器順序深度削減バインディングを適用する．結果は，レジスタに割当てられた変数がそれぞれ R1＝｛a, c, f, u1｝，R2＝｛u, e｝，R3＝｛dz｝，R4＝｛z, b｝，R5＝

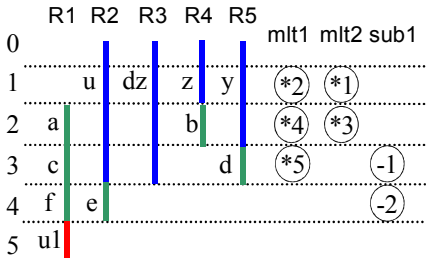

図 7. ライフタイム

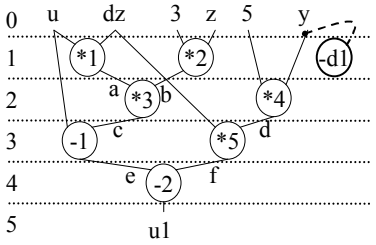

図 8. ダミー演算操作挿入 SDFG

｛y, d｝となり，演算器はそれぞれ sub1＝｛-1, -2｝，mlt1＝｛*2, *4, *5｝，mlt2＝｛*1 *3｝となる．よって，演算器順序深度は sub1＝(1, 2, 1)，mlt1＝(1, 1, 1)，mlt2＝(1, 1, 1)となる（図 3.1.1）．この例では，mlt2 の演算器順序深度を削減することができたため，乗算器の故障が検出しやすくなり，テスタビリティの向上につながると考えられる．

3.2. 戦略 2

バインディングした結果，各レジスタおよび各演算器に空き時間が存在している可能性が考えられる．この空き時間を利用し，さらなる演算器順序深度削減を狙い，テスタビリティの向上を図る手法として[14]の特徴であるダミー演算操作の概念を取り入れ提案する．この機能の追加による効果を，戦略 1 と同様，図 2 に示す SDFG を例に説明する．

まず，この SDFG に対し，戦略 1 で示した演算器順序深度削減法を適用する．その結果，3.1 章で示した通り，レジスタに割当てられた変数がそれぞれ R1＝｛a, c, f, u1｝，R2＝｛u, e｝，R3＝｛dz｝，R4＝｛z, b｝，R5＝｛y, d｝，演算器はそれぞれ sub1＝｛-1, -2｝，mlt1＝｛*2, *4, *5｝，mlt2＝｛*1 *3｝，つまり,演算器順序深度は sub1＝(1, 2, 1), mlt1＝(1, 1, 1), mlt2＝(1, 1, 1)となる（図 5）．この時点で，mlt1，mlt2 は演算器順序深度がともに(1, 1, 1)と最小になっており，改善の余地はないが，sub1 については右入力のレジスタの段数が 2 となっており，改善の可能性があるという点に着目する．

ここで，各レジスタおよび各演算器の使用時刻（ライフタイム）を調べ（図 7），各レジスタおよび各演算器の空き時間をもとめる．sub1 は，｛-1, -2｝が割当てられているため，時刻 3，4 で利用されている．つまり，時刻 1，2 が sub1 の空き時間となる．そこで，時刻 1 に擬似的な演算操作（ダミー演算操作）として’-d1’を挿入する（図 8）．このダミー演算操作は，sub1 に割当てられ，sub1＝｛-1, -2, -d1｝となる．ダミー演算操作の右入力にあたる変数を，外部入力に割当てることで，sub1 の演算器順序深度の削減を図る．ここ

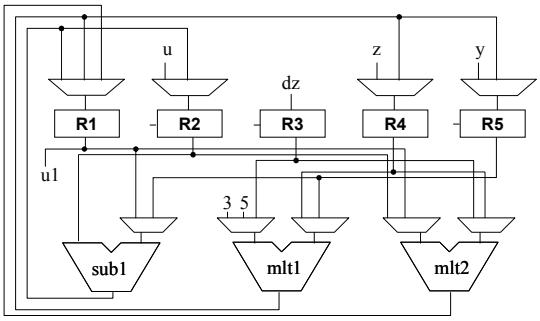

図 9. ダミー演算操作挿入法適用データパス

で，ダミー演算操作の右入力に y を割当てることとする．その結果，生成されるデータパスは（図 9）となり，演算器順序深度は sub1＝(1, 1, 1), mlt1＝(1, 1, 1), mlt2＝(1, 1, 1)となる．図 8 の例では，ダミー演算を用いることにより，追加する 2 入力マルチプレクサの面積オーバーヘッドの増加はあるが，演算器順序深度の削減に成功したため，テスタビリティの向上が期待できる．

4. 実験結果

本論文の 2 章で記載した SDFG（図 2）に対し，各手法を適用し，RTL 回路（データパスのビット幅 : 8, 32）を生成し，それぞれ TetraMAX を用い実験を行った（表 1，2）．なお，本論文で使用したモジュールを単体でテストした結果，故障検出効率は 100%であった．本実験では，テスタビリティを考慮しないバインディングアルゴリズムとして“LEA”（Left Edge Algorithm）[15], テスタビリティを考慮したバインディングアルゴリズムの“従来法”として順序深度削減指向バインディング法[8]，提案手法として“戦略 1”，“戦略 2”を比較する．

実験の結果, 戦略 1 , 戦略 2 どちらも, 故障検出効率, テスト生成時間ともに，LEA，従来法を上回る結果となった．32 ビット回路の打ち切り故障に着目すると，従来法は乗算器の内部に存在するのに対し，戦略 1，戦略 2 では，演算器の内部には存在しなかった．このことから，演算器順序深度を削減することにより，演算器のテスト容易化に成功したといえる．

5. おわりに

本論文では，テスト容易化動作合成法として，バインディング時における新たな評価尺度の提案，その評価尺度に基づくバインディング手法を提案した．今後の予定として，対象回路の増加および拡大を実現し，回路による効果の違いを解析し，バインディングアルゴリズムの改善を行うことが挙げられる．

参考文献

1) 藤原　秀雄: ディジタルシステムの設計とテスト, 工学図書株式会社, 2004
2) Daniel D . Gajski: High-Level Synthesis , Kluwer Academic Pub, 1992
3) M.C.McFarland, A.C.Parker, R.Camposano: The high-level synthesis of digital systems, Proc. IEEE, 301-318, 1990
4) R.Camposano and W.H.Wolf: High-Level VLSI Synthesis, Kluwer Academic Publishers, 1991
5) Z. Peng and K. Kuchcinski: Automated transformation of algorithms into register-transfer level implementations, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pages 150{166, 1994
6) P. G. Paulin and J. P. Knight: Forced-directed scheduling for the behavioral synthesis of ASIC's. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 8:661-678, June 1989
7) H.Fujiwara: Logic Testing and Design for Testability, The MIT Press, 1985
8) Tien-Chien Lee, Wayne H. Wolf, Niraj K. Jha, John M. Acken: Behavioral Synthesis for Easy Testability in Data Path Allocation, ICCD, 1992
9) IEEE Standard 1666: Open SystemC Language Reference Manual, IEEE, 2005
10) IEEE Standard 1076: VHDL Language Reference Manual, IEEE, 1987.
11) IEEE Standard 1076: Verilog Language Reference Manual, IEEE, 2001
12) Viraphol Chaiyakul, Daniel D.Gajski: Assignment Decision Diagram for High-Level Synthesis, Dept. of Information and Computer Science, Technical Report #92-103, December 12, 1992
13) Giovanni De Micheli, SYNTHESIS AND OPTIMIZATION OF DIGITAL CIRCUITS, McGraw-Hill,inc, 1994
14) Sying-Jyan Wang and Tung-Hua Yeh: High-Level Test Synthesis for Delay Fault Testability, EDAA, 2007
15) F.J.Kurdahi and A.C.Parker: “REAL: A program for register allocation”, In Proc. Design Automation Conf. Computer-Aided Design Automation Conf., pages 210-215, Miami Beach, June 1987.

表 1.　実験結果（8 ビット）

手法	打ち切り故障	テスト不能故障	総故障数	故障検出率	故障検出効率	テスト生成時間
LEA	36	203	3790	93.69%	99.05%	1044.58
従来法	14	38	3554	98.54%	99.60%	478.90
戦略1	8	47	3672	99.78%	99.78%	457.22
戦略2	0	48	3790	98.73%	100.00%	128.82

表 2.　実験結果（32 ビット）

手法	打ち切り故障	テスト不能故障	総故障数	故障検出率	故障検出効率	テスト生成時間
LEA	3687	4254	40938	80.60%	89.87%	1969070.38
従来法	36	182	40030	99.44%	99.90%	60401.60
戦略1	11	240	40484	99.38%	99.97%	24031.11
戦略2	4	239	40938	99.41%	99.99%	19616.10