

算術演算回路のデバッグ支援技術

久保 賢生[†] 藤田 昌宏[†]

[†] 東京大学大学院工学系研究科 〒113-8656 東京都文京区本郷 7-3-1

E-mail: [†]{masao,fujita}@cad.t.u-tokyo.ac.jp

あらまし 集積回路の大規模化・複雑化にともない、設計時間が長期化し、検証・デバッグに費す時間が支配的になってきている。設計者は従来、製品用にスタンダードセルを、試作用にFPGAを用いてきたが、DSM時代に突入しこちらの実装でも回路性能はレイアウトに大きく依存するようになってきている。そのため、設計誤り、仕様変更等による再設計が大変難しくなり、元の回路と同等の性能を出すために、デバッグでは最小の回路変更をすることが重要になっている。本稿では、その要求をみたす、論理設計での算術演算回路を対象としたデバッグ技術について紹介する。デバッグは、設計誤り部分の回路を抽出する部分と正しい回路へ置き換える部分の二つからなり、抽出部分について実験によりその性能を評価した。

キーワード 算術演算回路、乗算回路、デバッグ、検証

Debug Methodology for Arithmetic Circuits

Masao KUBO[†] and Masahiro FUJITA[†]

[†] Graduate School of Engineering, The University of Tokyo Hongo 7-3-1, Bunkyo-ku, Tokyo, 113-8656 Japan

E-mail: [†]{masao,fujita}@cad.t.u-tokyo.ac.jp

Abstract As VLSI design becomes larger and takes up much longer time, verification and debugging of its logic design become the dominating part of total design period. As circuit performances are very sensitive to layout designs in DSM, minimal change of circuit structures is important when debugging. In this paper, we give a debug methodology targeting arithmetic circuits. It modifies circuits locally and speeds up the total time for redesign with the two steps of extracting the erroneous parts and replacing by correct circuits.

Key words Arithmetic circuits, Multipliers, Debugging, Verification

1. はじめに

近年、集積回路設計の大規模化、複雑化により、設計期間が以前よりも長期化してきている。そのため、チップの集積度と一定期間に設計できる規模とは開く一方であり、設計の分野では利用可能な資源を使いきることができない問題に直面している。そして、設計の正しさを確認する検証・設計を修正するDebugは、設計時間の7、8割を占めていると言われており、設計期間を短縮するため、この検証・Debugを効率化することがますます重要になってきている。

集積回路のうち、算術演算回路、特に乗算回路は、多種の回路実装法が存在するために回路構造が大きく異なり、長年、検証しにくい問題として知られている。また、一般に算術演算回路は遅延や面積が大きくなりがちで、設計者は多くの時間を費して回路最適化を行うため、その回路に設計誤りが見つかった際の再設計は設計者の大きな負担となっている。それ故に、回

路全体にわたるRTL記述からの再合成を避け、一度費した労力を無駄にしないように、可能なかぎり一度設計した回路に近い回路を再合成し仕様を実現することが重要である。最近では、インテル・ペンティアムの浮動小数点除算演算のバグもあり、算術演算回路の検証技術への関心が高まってきており、検証による設計誤り発見後のDebug技術にも関心が及んでいる。

本研究では、論理回路設計段階において、算術演算回路、特に乗算回路を検証して設計誤りを確認した際の計算機によるDebug手法を提案する。ここでは、修正をできるだけ小さなものにするにより、元の回路の性能に近い回路を導きだし、設計の高速化を目指す。処理の流れは図1に示すとおり、回路中の設計誤り部分とその正しい論理の抽出と、設計誤りを正しい論理に置き換えの二つの部分からなる。はじめに、誤設計と検証された回路と論理的に正しい回路を比較していくことにより、設計誤り部分を抽出する。ついで、それを仕様をみたす最小の回路変更を導きだし、回路の置き換えを行う。デバッグ手

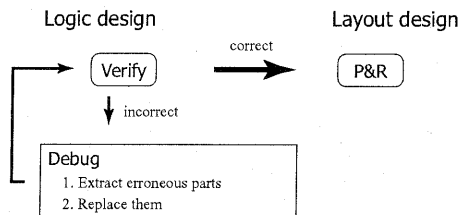


図 1 Position of debugging

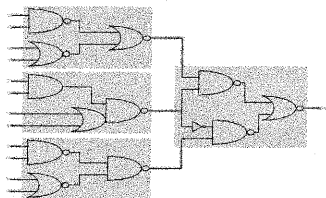


図 2 LUTs mapped on standard logics

法は、スタンダードセルを用いた回路、FPGA を用いた回路両方へ応用させることを考えている。FPGA は LUT (Look Up Table) を基本構造に構成されているが、その回路は、図 2 にあるように、論理レベルでの最適化が行われた後に LUT によりマッピングされたものであると考えることができる。つまり、回路のトップモジュールでは LUT だけが見えているが、一つ下のモジュールでは標準的な論理ゲートを用いて表されていると考えることができる。そのため、回路中の設計誤りの抽出部分ではスタンダードセルによる回路と同様に扱い、回路の正しい論理への置き換え部分において LUT を意識した処理を行う。

本稿は、第 2 節で取り組むべき問題を整理し、第 3 節、第 4 節で提案手法の抽出部分、置き換え部分について、細かく説明を加え、第 5 節で実験結果を示し、第 6 節でまとめと今後の予定について触れる。

2. 算術乗算回路の回路修正問題

算術演算回路とは、加算や乗算、あるいはもう少し複雑な $a \times b + c$ などを計算する回路のことである。このとき、 $a \times b + c$ などの複合算術演算回路は、乗算回路 $a \times b$ の部分積に、余分な部分積 c が加わった回路として実現でき、基本構造は乗算回路であると考えられる。そのため、現時点では $a \times b$ で表される乗算を Debug の対象とする。

一般的に、 n ビットの乗算回路 $f = a \times b$ は、図 3 のように二つの部分に分けられる。一つは、入力 $a[j], b[j]$ から、 $n \times n$ 個の考えられる入力 a と b の論理積を計算して、部分積 $P_{ij} = a[j] \cdot b[j]$ を求める回路である。もう一つは、求めた部分積をそれぞれの桁ごとに加算を行う回路である。この後半部分の回路は、例えば、4 ビットの乗算回路に対して、図 4 のように部分積 P_{ij} を用いて構成することができる。

乗算回路への回路修正は、回路設計後に回路の設計誤りが検証ツールにより発見されたときに必要となる。本稿では、乗算回路の仕様である“ n ビットの乗算回路 $f = a \times b$ ”に対して、設計誤りを含んだ論理回路の Debug を考える。前提条件として、

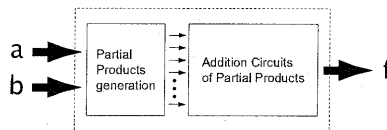


図 3 Basic structure

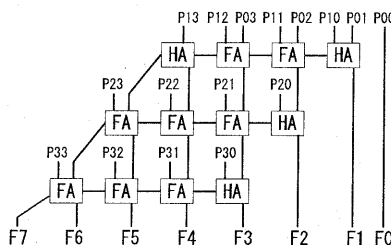


図 4 4bit multiplier

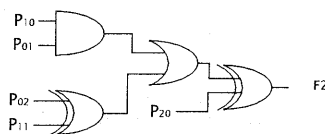


図 5 Circuit for 3rd output with a error

n ビットの乗算回路を設計し、その i 桁目の出力が設計誤りを含む最小の桁であり (多くの場合、 $i(< 2n)$ 桁目に設計誤りがある場合、それ以降の $j(> i)$ 桁目の出力にも、設計誤りが含まれる)、論理回路上の信号線が、外部入力 a, b 、外部出力 f のそれぞれ何桁目を表すのか分かるものとする。その時で、 i 桁目の出力に関する回路を診断し、設計誤り部分を特定し、それを可能な限り最小の回路変更で、正しい論理回路に置き換えることを試みる。

例えば、4 ビット乗算器を設計した際に、3 桁目の出力に設計誤りがあり、その回路図が図 5 であるとする。このとき、3 桁目の正しい論理は、部分積 P_{01}, P_{10} の AND により生成されるキャリーと、部分積 P_{02}, P_{11}, P_{20} の XOR となる。これを部分積を用いて論理関数で表すと、 $F_3 = (P_{01} \cdot P_{10}) \oplus P_{02} \oplus P_{11} \oplus P_{20}$ である。これから、図 5 を見直すと OR ゲートが設計誤り部分であると特定でき、その部分を XOR ゲートに置き換えてやることが Debug となり、この自動化が本研究の目標である。

Debug では、設計誤りを探すのに回路の間違い探しをするため、その意味で検証と通ずる点がある。そして、乗算回路は、その検証を行いにくい回路として知られている。それは、部分積の計算順序、ビット表現や符号化方法により、加算器をアレイ構造に並べたもの、キャリーセーブアダプターをアレイ構造に並べたアレイ型、ツリー構造に並べたワレストリー型、そして、ブースエンコーディングを用いる実装など、多くの回路構成方法があることによる。更に、全加算器の実装法や入力となる部分積や桁あげ (キャリー) の選択に自由度があり、回路の構成法のみで回路が一意に定まると言うことも問題を難しくさせている。しかしながら、興味深いことに、いづれの構成法でも、基本構造は、半加算器、全加算器を規則的に並べたものであり、

この特徴を Debug に利用してゆく。

3. 設計誤りの抽出

3.1 関連する乗算回路の検証技術

設計誤りを抽出するために、設計誤りを含む回路と論理的に正しい乗算回路を等価性検証していき、等価と判定できなかった部分を設計誤りを含む部分として抽出する。ここでは、その際、応用可能な検証技術を紹介する。

一般に、検証ツールは比較する二つの回路の内部等価点を探し、見つかった内部等価点を入力とみなして、次なる内部等価点を探し、最終的に外部出力の等価性判定を行う。しかし、乗算回路に対しては、乗算回路を論理表現する際に表現が爆発してしまうこと、多種の回路構成方法があるために、二つの回路の内部等価点が見つかりにくいことより、単純に一般の検証技術を応用することは難しい。その弱点を克服するため、[1] では、回路を半加算器のネットワークで対応づけし、その表現上で等価性検証をおこなっている。また、[2], [3] では、算術演算回路の特徴から、数学的に回路構造から満たすべき条件を導きだし、それを帰納的に検証している。本研究では、[1] の乗算回路を半加算器のネットワークで表現する手法を利用する。また、[2] で提案される検証手法は、回路内部の操作はないため、等価点を見つけるために利用することはできないが、算術演算回路全体をみて、どの算術演算部分で誤りがあるか特定するのに利用可能である。

3.2 提案する Debug 技術

本節では、本研究で提案する乗算回路の Debug 技術を紹介する。処理のフローは以下に示す通りで、入力設計誤りを含む回路で、出力が回路の置き換えを行う次処理へ渡す、回路とその満たすべき論理である。3 番目のステップである、誤った回路を半加算器/全加算器 (HA/FA) のネットワークに対応づける部分がこの処理の主要な部分で、1 番目、2 番目のステップは、この処理の負担を軽減させるために試みられる。

- (1) 潜在的に対象範囲となる回路の抽出
- (2) 桁ごとの計算部分への回路の分解
- (3) 半加算器、全加算器のネットワークへの対応づけ

これを以下の小節で例をもって説明を加えてゆく。そして、ここでは、のように、4 ビットの乗算回路において、5 桁目 (図 4 の F_4) の出力が最小桁の設計誤りを含む出力である状況を想定して、例を示して行く。

3.2.1 潜在的に対象範囲となる回路の抽出

注目する対象を小さくするために、まず全体から設計誤りを含む可能性のある回路の抽出を行う。乗算回路の i 桁目の出力が最小桁の設計誤りを含む出力であった場合、その設計誤りは i 桁目の出力を構成する部分にあるため、 i 桁目の出力の TFI 部分が単純に設計誤りを含むと疑われる。ここではさらに、 i 桁目の出力が最小桁の設計誤りであることを考慮して、TFO の中に含まれる出力のうち i 桁目の出力が最小桁の出力となっているゲートの集合による部分回路が潜在的に設計誤りを含む部分として抽出できる。実際には、回路の論理を追いながら取り扱うため、部分積を意識した単位で抽出をすることになる。

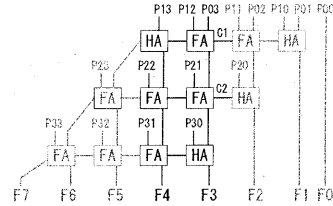


図 6 Ideal extraction

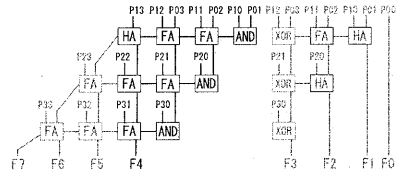


図 7 Extraction in bad case

理想的には、乗算回路は、図 4 のように、回路の j 桁目の出力は、 $j-1$ 桁目の出力で計算するのに用いる $j-2$ 桁目からのキャリーと $j-1$ 桁目の部分積から j 桁目へのキャリーを生成し、更にそれと j 桁目の部分積から出力を計算することが望まれる。図 6 も同じ状況であり、5 桁目が設計誤りを含んでいるとき、4 桁目の出力に設計誤りがないため、3 桁目からのキャリー $C1, C2$ の伝播値は正しい論理値を伝播していると考え、設計誤りを含む部分の対象範囲は黒線で表した部分となる。しかし、しばしば、回路の高速化のため、図 7 のように 5 桁目の計算に使うキャリーを、4 桁目以下の出力計算で用いた回路とは別に用意することがあり、この場合は対象範囲は、黒線部の 5 桁目の出力を構成する回路全体となる。このように i 桁目の出力計算回路に対し、 $i-1$ 桁目以下の出力計算回路との関係より、回路抽出を行う。

3.2.2 桁ごとの計算部分への回路の分解

本手法では、HA/FA のネットワークへの対応づけを行うが、そのとき、回路が小さいほど対応づけが行いやすい。そこで、回路の構造から桁ごとの計算部分へ分解可能かを調べ、可能ならそれを分解し取り扱う。前のステップでは、最終ステップの HA/FA への対応づけの処理全体で取り扱う大きさを決めるのに対し、このステップでは、HA/FA への対応づけ一回一回での処理で取り扱う大きさを決めている。乗算回路は論理関数表現をする際に爆発しやすい性質をもっているため、一度の処理で取り扱う必要のある回路が小さいことが望まれており、その意味で、この分解は前の抽出より重要なステップである。

分解可能か否かの判定は、各桁の部分積を元に回路構造的に部分積、キャリーらしき信号線を見つけたし、更に j 桁目への下位桁からのキャリー、 j 桁目の部分積、 j 桁目から上位桁へのキャリーについて、回路上の数と理論的な数とを比較して行う。もし、部分積、キャリーらしき信号線が見つけ出せない場合は、分解不可能として最終ステップに進むことになる。 j 桁目への下位桁からのキャリー、 j 桁目の部分積の理論的な数は以下のように与えられる。また、 j 桁目からの上位桁へのキャリーは、(1) 式について、 $j+1$ 桁目への下位桁からのキャリー

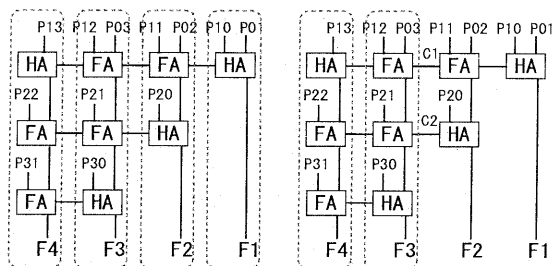


図 8 Decomposition by digit

図 9 Ideal decomposition

として考えた値になり、同様を求めることができる。

$$\# \text{ of carries} = \begin{cases} 0 & (j=1) \\ -|j - (n + \frac{3}{2})| + n - \frac{1}{2} & (j \geq 2) \end{cases} \quad (1)$$

$$\# \text{ of partial products} = -|j - n| + n \quad (2)$$

まず、設計誤りの i 桁目の計算部分に注目し、上記の数の信号線からなっており、かつ、キャリーが $i-1$ 桁目の計算部分から計算されているならば、回路を分解する。更に、その下位桁の計算部分に対し、信号線の数を確認し、繰り返し分解を行ってゆく(図 8)。回路が、利用できるキャリーすべてを共有するよう理想的に設計されていれば、図 9 にあるように二つの桁のみを注目すればよく、更には 5 桁目の計算部分は部分積とキャリーが XOR により回路を作っていることを確認するだけでよい。分解ができない場合は、いくつかの桁にまたがった大きな計算部分として扱うことになるが、実際に Design Compiler/Design Ware を用いて、4 ビット乗算器にクロックと面積により制約を加え合成し、それを手動で回路診断したところ、すべての回路が上記に示した方法で分解できるものであった。この性質は、ビット数が 4 ビットであったことによるかもしれないが、[1] においても同様の性質が見られるとあり、比較的多くがこの性質をもっていると思われる。

3.2.3 半加算器、全加算器のネットワークへの対応づけ

先のステップで分割可能であれば分割された回路に対し、そうでなければ抽出された回路全体に対し、半加算器/全加算器(HA/FA)のネットワークで対応づけを行っていく。実際には分割できない回路が存在するかも知れないが、その例がわからず問題として捉えることができないため、ここでは分割された回路に対する方法のみについて触れる。また、[1] では半加算器で対応をとっていたが、3 入力 AND など半加算器 1 のみでは対応がとれないことがあり、半加算器だけでなく全加算器も用いて対応づけを行う。そして、全加算器は半加算器 3 つで対応づけが可能であることを利用し、全加算器は柔軟に 3 入力 AND などの全加算器の論理、あるいは、半加算器 3 つによる全加算器の論理に対応するものとする。

以下に HA/FA のネットワークへ対応づける流れを示す。

- (1) 対応づけられる HA/FA のネットワークの用意
- (2) 検証ツールを用いて、入力側から HA/FA のネットワークへの対応づけ

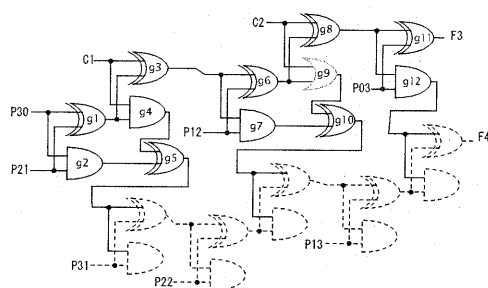


図 10 Target circuit computing 4th output

(3) 設計誤りにぶつかった後、出力側から HA/FA のネットワークへの対応づけ

まず、修正すべき回路に対して、論理的に正しい HA/FA ネットワークをいくつか手元に用意する。これらの回路は、後に与えられた設計誤りを含む回路と比較されるものである。回路構成は単純なアレイ型で、半加算器が各桁の最も入力側/最も出力側にあるか、各桁間の配線の仕方により、いくつかの種類を用意している。乗算回路は部分積を足しあげてゆく部分があるが、その部分積の順番は修正すべき回路と同じまたは似たものになるよう配慮している。これにより、手元の回路と修正すべき回路の内部等価点を見つけやすくなっている。

次に、手元の回路と修正すべき回路を既存の検証ツールにより内部等価点を見つけて対応づける。検証ツールは乗算回路用に特別な機能を持たせているわけではないが、部分積の計算順が近い回路の検証をしているため、通常の乗算回路の等価性検証よりロバストなものになっている。

検証ツールが終了し内部等価点を見つけられなくなったら、ついで、出力側からの対応づけに移る。この過程では、単純に半加算器/全加算器の論理があるか否かをしらべ、手元の回路と対応づけをしていく。この処理で、どのゲートを入力として論理を調べるかが、半加算器/全加算器の論理の存在を調べるのに影響し、最も重要となる。現実装では、入力側からの対応づけで検証ツールの見つけた内部等価点への依存度を見て入力を選んでいく。最後に、出力側からの対応づけでも対応づけが行えなくなったところが実際の設計誤り部分と考えられ、その部分の回路と手元の回路から導き出される満たすべき論理が導き出される。

この過程を、設計誤りが 5 桁目に含まれている例で紹介する。簡単のため、論理ゲートを半加算器の成分 XOR や AND で表現した図 10 を考えてみよう。このとき、回路は 4、5 桁目計算部分に分解でき、これは、キャリーが $C1, C2$ で、5 桁目へのキャリーを $g5, g10, g12$ で生成している 4 桁目の計算部分である。また、論理ゲート $g9$ が設計誤りで、本来 AND である必要がある。この回路に対し、手元の HA/FA のネットワークは、HA の位置が最も入力側か最も出力側かで図 11 にあるように 2 種類考えられる。そして、この回路に対し、入力側から HA/FA のネットワークへの対応づけを行っていく。検証ツールにより、論理関数を見ていくと、 $g3, g5$ で全加算器の論理

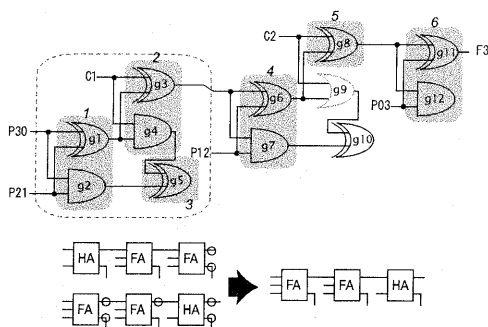


図 11 Forward mapping

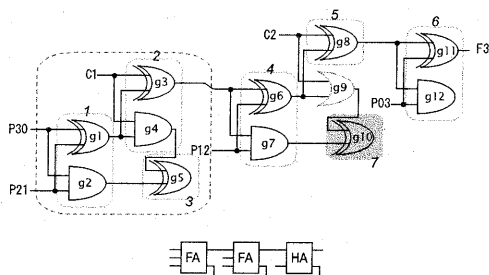


図 12 Backward mapping

SUM、CARRY、 g_8 で全加算器のSUM、そして、 g_{11} と g_{12} で、半加算器のSUMとCARRYの論理が見つかり、それらは手元のネットワークと対応づけされる。同時に、 g_9 と g_{10} において対応づけできない部分が見つかり検証ツールはストップする。ここで、検証中に見つかった内部等価点は、上の手元のネットワークで2つ、下の手元のネットワークで5つとれているので、与えられた回路の候補は、下側のネットワーク(図11)にしばらくされる。次に、対応づけできなかった部分を設計誤り部分とみなし、出力側からの対応づけに移る。出力側より回路の論理関数を調べる(図12)と、 g_{10} でXORが見つかり、それ以上マッピングできるところがないために、 g_9 が設計誤り部分として特定される。同時に手元のネットワークから、修正すべき論理としてANDを抜き出し、抽出は終了する。現時点では、手元のネットワークは半加算器、全加算器の精度で構成しているため、修正すべき論理は半加算器、全加算器単位となる。

上記では、Debugに関して図3の部分積の加算部分のみに焦点をあてて話をしてきたが、その場合、正しい設計を設計誤りとみなしてしまう可能性があり、部分積生成部分も含めて、回路を見る必要がある。例えば、図13のような仕様と実際にD.C./D.W.で設計された回路を見てみる。ここで、部分積のみに注目し、仕様の論理関数を調べると、 $(P_{02} \cdot P_{11}) \oplus P_{03} \oplus P_{12} = \overline{(P_{02} \cdot P_{11})} \cdot P_{03} \oplus P_{12} + (P_{02} \cdot P_{11}) \cdot P_{03} \oplus P_{12}$ となり、回路は $\overline{(P_{02} \cdot P_{11})} \cdot P_{03} \oplus P_{12} + (P_{02} \cdot P_{11}) \cdot (P_{03} \cdot P_{12})$ となる。このとき、部分積の入力を $P_{02} = 1, P_{11} = 1, P_{03} = 0, P_{12} = 0$ とすると、仕様は"1"で、回路は"0"となる。これは一見、設計誤りのように見えるが、実は、 $P_{02} = P_{11} = 1 \Rightarrow a_2 = a_1 = b_0 = b_1 = 1 \Rightarrow P_{12} = 1$ より、上記のような $P_{12} = 0$ 組合せにな

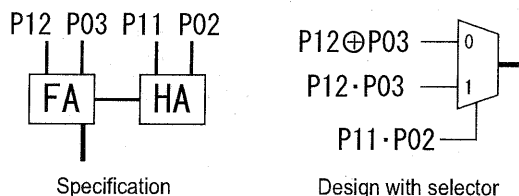


図 13 Design with Selector

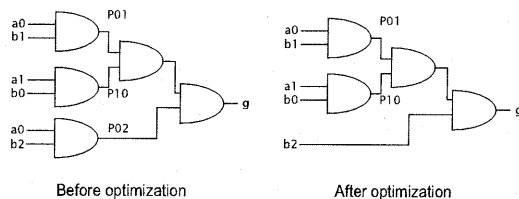


図 14 Elimination of AND

ることではない。つまり、これは、設計誤りではなく、回路最適化がかかっていたのである。このようなことを避けるため、回路を見る範囲を広げ、特に入力に近い部分では、回路を外部的入力 a, b で見る必要がある。

また、上記の説明では、部分積を元にした回路を見ていたが、実際の回路では部分積同士のANDをとるような計算で最適化されることがある。例えば、図14左側の回路を実装する際、入力 a_0 は、回路に2つ入力としてはいつているため、図14右側のように最適化が可能となる。そして、この場合、回路は P_{01}, P_{10}, P_{02} のANDを計算していることを理解できるようにする必要がある。提案手法では、回路前半部分に対して検証ツールを用いることにより上に示す例に対応している。

4. 設計誤り部分の正しい論理への置き換え

抽出過程で導き出された回路中の設計誤り部分を、満たすべき論理を持つような論理ゲートに置き換えることがDebugの最終ステップである。回路の性能はレイアウトに大きく依存するため、この置き換えでは構造的変化の小さい修正を行い、元の回路と同等の性能をもった回路を導き出すことが重要となる。そして、再配置、再配線の負荷を低減するため、スタンダードセルを対象としたもの、FPGAを対象にしたものがそれぞれ研究されてきた。スタンダードセルに対しては、論理ゲートをそのまま利用して再配線のみを行うことにより、論理を修正する技術[4],[5]、FPGAに対しては、配置・配線はそのまま、LUTの論理関数を変更して、論理を修正する技術[6]が提案されている。以下で、簡単に説明を加える。

スタンダードセルを対象として、[4],[5]は、対象の論理ゲート間の再配線を考え、局所的な再配線により元の回路と同等の性能を実現している。その後、レイアウトを得るには、再配置は必要なく、再配線のみを局所的に行うだけでよい。実際の処理は、ループを作らないように配線間に接続/非接続の意味をもつ2値変数を与え、回路が仕様を満たすような2値の真理値を求めることをしている。

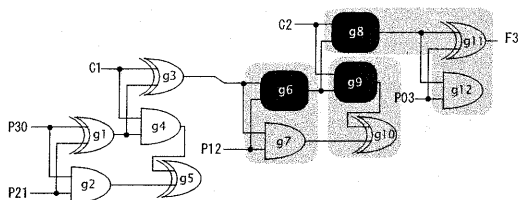


図 15 Consider at LUT level

対象が FPGA であっても、回路抽出の部分では論理的処理しか行わなかったため通常のゲートのように扱っていたが、回路の置き換えにはレイアウトレベルの処理が必要となり、LUT 単位で扱う必要がある。つまり、図 15 において、抽出を行う論理レベルでは論理ゲート一個一個をみればよかったが、回路の置き換えを行うレイアウトレベルではグレーで囲われた LUT 部分を基本ブロックとして考える必要がある。もし、設計誤り部分が 1 つの LUT 内に納まっているならば、単純に LUT の論理を書き換えればよいが、図 15 のように、黒い論理ゲートが設計誤りでいくつかの LUT にまたがる場合は、最小の回路修正を探すことになる。[6] では、回路の配置・配線はそのままに、LUT の論理を変えることのみで論理修正を実現している。このため、レイアウトを得るのに、再配置、再配線は必要なく、性能が変わることはない。

これらが応用できる回路の範囲は大きくはないが、本研究での利用はスタンダードセルの 10 個以下の論理ゲート、数個の LUT と想定されるため、これらの技術は利用可能であると考えられる。

5. 実験結果と考察

提案手法の設計誤り抽出部分について、16 ビットのアレイ構造の乗算回路である C6288 を対象として、実験を行った。実験は、まず回路中からひとつ設計誤りを含ませる出力を無作為に選ぶ。次いで、その出力の TFI からひとつ論理ゲート無作為に選び、そのゲートの論理を反転、あるいは論理の変更を行い、設計誤りを含む回路を作成した。ここで、C6288 からこのように作った回路は、選んだ設計誤りを含む出力より下の桁においても論理的に誤った出力が存在する。第 3 節で説明した通り、最も下位桁の設計誤りを含む出力に注目して、その TFI に対して回路修正するのが効率的であるが、対象によっては最悪図 7 のように出力以下の回路全部が調べる範囲になりえるため、ここでは最悪な場合を想定し実験を行った。また、提案手法の実装はまだ初期段階であり、無作為に選ぶ設計誤りを含む桁は 15 桁目以下とした。このように作った回路に対し、Linux 上の 700MHz の CPU と 512MB のメモリの PC 上で、設計誤り部分を見つけたし回路修正を行い、確認のため修正した回路の検証を試みた。

表 1 は、論理の反転、論理の変更により設計誤りを一個挿入した回路に対し、20 回のこの実験を行った平均の CPU 時間である。一列目は設計誤りの種類で、二列目、三列目は入力側からの対応づけ、出力側からの対応づけ、四列目は回路修正後の回

表 1 Rectification time(sec)

error type	forward	backward	verification
invert	3.17	0.04	2.34
change	2.71	0.04	2.55

路への検証時間を示している。実験において、すべての回路で設計誤り部分を特定、修正すべき論理が導かれた。また、同様に設計誤りを二個挿入した回路でも実験を行った。このとき、いくつかの回路で設計誤りを発見、修正することができた。設計誤りの論理ゲートが互いに依存しない場合、設計誤りを修正できたと思われる。一方が他方に依存する場合は、不可能ではないが、回路修正は大変難しくなるものと思われる。

上記の結果は、予備的な実験結果であるが、基本的な構造の乗算回路に対して、正常に動作することが確認できた。また、取り扱った回路が非常に簡単であったこともあるが、同時に数個の設計誤りを修正できることも示した。更に、今回は利用しなかった、潜在的に対象範囲となる回路の抽出、桁ごとの計算部分への回路の分解を行うことよりデバッグの効率化が期待でき、大きな回路、最適化のかかった回路、また別の実装による回路を取り扱える可能性がある。

6. まとめと今後の予定

本稿では、性能上ボトルネックになる算術演算回路、特に乗算回路に焦点をあてた Debug 技術を提案した。これは設計した回路の変更を最小限にするため、再配置、再配線の負担を軽減し、結果的に設計時間の短縮を可能にしている。第 3 節、第 4 節で、回路の設計誤り部分の抽出、その置き換え技術について紹介した。第 5 節では、設計誤り抽出部分について予備的ではあるが、実際に回路抽出可能であることを示した。

今後は、まず提案手法の実装を向上させ、最適化されたアレイ型乗算器、他の実装法による乗算器、更に一般の算術演算回路と対象の回路を一般化をして行く予定である。

文 献

- [1] Dominik Stoffel, Wolfgang Kunz. "Verification of Integer Multipliers on the Arithmetic Bit Level," Proc. IEEE Intl. Conf. on Computer-Aided Design (ICCAD-01), pp.183-189, November 2001.
- [2] Ying-Tsai Chang, Kwang-Ting Cheng "Induction-based Gate-level Verification of Multipliers," Proc. IEEE Intl. Conf. on Computer-Aided Design (ICCAD-01), pp.190-193, November 2001.
- [3] Masahiro Fujita. "Verification of arithmetic circuits by comparing two similar circuits," Proc. Intl. Conf. on Computer Aided Verification (CAV-96), pp.159-168, 1996.
- [4] Yuji Kukimoto, Masahiro Fujita, Robert K. Brayton. "A Redesign Technique for Combinational Circuits Based on Gate Reconnection," Proc. IEEE Intl. Conf. on Computer-Aided Design (ICCAD-94), pp.632-637, November 1994.
- [5] Masao Kubo, Masahiro Fujita "A Redesign Method Based on Evaluating Quantified Boolean Formulae," International Workshop on Logic & Synthesis, June 2001
- [6] Yuji Kukimoto, Masahiro Fujita. "Rectification Method for Lookup-Table Type FPGA's" Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD-92), pp.54-61, November 1992.