

回路分割機構を備えた高位合成ツールによる検証回路の生成

松田 和也¹ 三好 健文² 竹本 正志¹ 船田 悟史² 中條 拓伯¹

概要：近年、従来の回路設計に用いられてきた HDL に替わり、高位合成ツールの活用に注目が集まっている。しかし、複雑なアルゴリズムをハードウェア化する際に、合成回路が大規模化する場合やシミュレーション時間が膨大となる場合がある。そこで、複数 FPGA に対する分割実装が用いられるが、FPGA の回路規模や I/O ブロック数による制約が問題となり、検証環境の構築は容易ではない。本研究では、高位合成ツールの合成回路を部分回路に分割し、回路検証用のラッパーを生成することで、部分回路単位での検証を可能とする。高位合成ツールを用いて、FFT を実行するプログラムを合成し、回路分割機構により分割した。各部分回路は、シミュレーションおよび FPGA 上で動作検証を行い、正常に動作することを確認した。

キーワード：高位合成, 回路分割, 回路検証

Generation of the Verification Circuit by High-Level Synthesis Tool in a Circuit Partitioning mechanism

KAZUYA MATSUDA¹ TAKEFUMI MIYOSHI² MASASHI TAKEMOTO¹ SATOSHI FUNADA²
HIRONORI NAKAJO¹

Abstract: In recent years, a high-level synthesis tool has been attracted in designing hardware circuits instead of conventional HDL. However, there exist two issues to implement a complex algorithm into hardware, which brings growing scale of a synthesized circuit and time for simulation. Therefore, though partitioning a circuit into multiple FPGAs is currently put into practical use, there are two constraints in implementation; the scale and the number of I/O blocks in an FPGA. Thus it is difficult to build a verification environment. In this study, we partition a circuit synthesized by a high-level synthesis tool into some reduced circuits. Moreover, the small circuits are equipped with self-verification function with generating a wrapper for each circuit verification. An FFT circuit which is generated by a high-level synthesis tool is partitioned by our proposed circuit partitioning mechanism. We verify the partitioned circuits in RTL simulation as well as implementation on an FPGA in order to confirm our targeted circuits are correctly operated.

Keywords: High Level Synthesize, HLS, Circuit Partitioning, Circuit Verification

1. はじめに

近年の集積回路技術の進歩により、LSI に搭載可能な回路規模は増大し、設計開発は複雑化している。それに伴い、高位合成ツールによる回路設計が広まり始め、従来のハードウェア記述言語 (HDL) による回路設計よりも高い抽象レベルで、複雑なアルゴリズムのハードウェア化が可能となった。現在、C 言語や Java 言語をベースとしたものをはじめ、様々な高位合成ツールが存在する。C 言語を

ベースとするツールとしては、SystemC[1], ImpulseC[2], CyberWorkBench[3] や LegUp[4] が挙げられ、Java 言語ベースには、JHDL[5] や Lime[6], JavaRock[7] がある。

高位合成ツールには、プログラミング言語に対する拡張を行わず、既存の C 言語や Java 言語で記述されたプログラムを入力とするものもあり、C 言語をベースとする LegUp[4], Java 言語をベースとする JavaRock[7] などが該当する。これらのツールでは、対象となる入力プログラムをソフトウェアとして動作させることで、アルゴリズムレベルでの検証を可能とするため、高位合成ツールにより合成された回路の検証コストは削減できる。また、C 言語や Java 言語が活用できるため、これらの言語ユーザによる

¹ 東京農工大学
Tokyo University of Agriculture and Technology
² 株式会社イーツリーズ・ジャパン
e-trees.Japan, Inc.

ハードウェア設計に門戸を開くこととなる。

ASIC 開発において、合成回路が仕様を満たすことを確認するためには、シミュレーションを行い、回路動作を検証する必要がある。しかしながら、高位合成ツールによる合成回路に対するシミュレーション時間が膨大となる場合がある。シミュレータによる検証では、各論理ゲートに対するシミュレートが必要であるため、複雑なアルゴリズムをハードウェア化する場合、そのシミュレーション時間は無視できないほど大きくなる。ASIC 開発において、シミュレーションによる回路検証の他に、FPGA を用いた回路検証があるが、FPGA の実装容量の制約のために、1 つの FPGA に格納不可能となる可能性がある。この場合、複数 FPGA に対する分割実装が行われるが、分割回路間の信号線数が FPGA の I/O ブロック数を超えてしまうという問題が起きている。

これまで、FPGA の I/O ブロック数制約を満たすために、分割回路間の信号線数を最小化するための手法 [8][9] や、1 つの I/O ブロックを複数の信号で共有する手法 [10] が提案されてきた。しかしながら、現状の高位合成ツールには、合成回路を分割し、複数 FPGA に実装するための機能はほとんど見当たらない。そのため、FPGA を用いた回路検証を行うために、これらの手法を手動で適用しなければならない。

高位合成ツールの普及により、C ユーザや Java ユーザによるハードウェア設計が身近になった今日において、これらのユーザが、実現したいアルゴリズムを効率良く設計して実装できるようにし、複数 FPGA にまたがるような大規模回路に対しても容易に検証できるような環境を提供することが望ましい。そこで、本研究では、高級言語による記述の段階で回路分割を考慮し、分割回路を検証するための機構を提案する。分割回路を検証することにより、実装したアルゴリズムにおいて、ボトルネックとなる処理を探索することが可能となる。

高位合成ツールに回路分割機構を備えることで、合成回路に対する分割点の探索が自動的に行われ、分割回路を検証するための回路が自動生成される。検証用回路を合成することで、ユーザによる検証環境の構築に要する手間は簡略化される。

本論文では、Java ベースの高位合成ツールである JavaRock を対象とし、JavaRock の合成回路を分割することで、検証用回路を生成する回路分割機構を実装し、評価する。2 章において、関連研究について述べ、3 章で高位合成ツールの合成回路に対する回路分割機構の処理について述べる。4 章では、回路分割機構の概要を述べ、5 章で評価を行う。続く 6 章で考察を行った後、本論文のまとめを行う。

2. 関連研究

2.1 高位合成ツールに対する回路分割

高位合成ツールによる回路分割手法については研究途上にあり、現在、CyberWorkBench に対する回路分割手法 [11][12] および JavaRock に対する回路分割手法 [13] が提案されている。文献 [11][12] は、ハードウェアオフローディングを目的とし、大規模回路の分割に焦点が当てられている。一方、文献 [13] では、汎用的な用途としての回路分割手法が提案されている。

文献 [11] では、C 言語で記述されたアプリケーションプログラムを BDL 形式に変更し、設定ファイルとともに分割ツールに渡される。分割ツールは、プログラムのモジュールや関数をブロックとし、設定ファイルで与えた条件を満たすように分割を行う。FPGA の回路規模制約を満たし、分割回路の性能が最も高くなる部分において、ソースコードは分割される。分割されたソースコードは CyberWorkBench に対する入力となり、分割済みの回路が合成される。

文献 [12] は、文献 [11] の回路分割手法におけるストリーム処理プログラムへの適用について述べられている。データパス回路表示ツールによる CyberWorkBench の合成回路に対する解析を実施し、有効な分割点候補を特定した後、入力ソースコードの分割を行う。

なお、文献 [11][12] では、FPGA 間の通信部分を手動で設計する必要があるため、FPGA 間の通信インタフェースへの対応は将来的な課題として挙げられている。

文献 [13] では、Java プログラムにおけるクラスをベースに回路分割を行い、FPGA の回路規模制約を満たした上で、I/O ブロック数の削減が試みられている。この手法では、JavaRock による合成回路における BlockRAM の利用に着目し、同一 BlockRAM にアクセスするモジュールを同一 FPGA に格納するという戦略がとられている。しかしながら、複数 FPGA に分割実装した回路の動作に関する評価は行われていない。

以上のように、文献 [11][12][13] は、高位合成ツールの合成回路に対する回路分割手法の提案を行ってはいるものの、分割した回路が実際にどのように動作するかについての検証は行われていない。その検証のための端的な方法として FPGA 上で分割回路を動作させることが挙げられるが、複数 FPGA を装備したシステムや、そのシステム上での FPGA 間の接続や通信方式にも大きく影響を受けるため、そのインタフェースを含めた開発は設計者にとって大きな負担となる。そのため、複数 FPGA に対する分割実装にはハードウェアに関する知識が不可欠であり、高位合成ツールの普及を入り口として、ハードウェア開発に携わる C ユーザや Java ユーザに対するハードルは高くなるも

のと考えられる。

この問題を解決するためには、分割回路に対して汎用的なインタフェースを供給し、その分割回路単位での回路検証を可能とすることが有効となると考えた。そこで、本研究では、以下の条件を満たす回路分割機構を実装した。

- (1) 高位合成ツールによる合成回路に対する自動分割機能を持つこと
- (2) FPGA に対して分割回路を汎用的に実装できること

条件 (1) に関しては、文献 [11][12][13] においても実現されている。しかしながら、条件 (2) には対応しておらず、この条件を満たすためには、複数 FPGA が相互通信を行い、回路全体を動作させる必要がある。そこで、本研究では、条件 (2) を満たすために、文献 [11][12][13] とは異なるアプローチを採用する。つまり、高位合成ツールが合成した回路の動作に着目し、実行状態にある回路のみを取り出すことで、その時点における合成回路の動作を再現することを考える。このとき、各分割回路に対する回路検証用のラッパーを生成し、分割回路間の信号を隠蔽することで、I/O ブロック数制約を満たす。また、各分割回路は、ラッパーに内包されるため、FPGA 上において分割回路を独立して動作させることが可能である。以上により、C ユーザや Java ユーザに対して回路検証を容易なものとし、複雑なアルゴリズムや大規模計算のハードウェア化により増大化した回路に対しても対応できる開発環境を提供できるものと考えられる。

また、近年では、FPGA の部分再構成に注目が集まっている。提案手法により分割された回路に対して部分再構性を適用し、1 つの FPGA 上での回路検証を可能とすることは、将来的な課題として挙げられる。

2.2 高位合成ツール JavaRock

Java ベースの高位合成ツールである JavaRock[14] は、Java 言語で記述されたプログラムを合成し、VHDL コードに変換するツールであり、Java 言語に対する拡張を行うことなく、ハードウェア設計を可能にするという設計方針のもとに開発が進められている。そのため、JavaRock の習得や活用に対するコストは小さく、Java ユーザに対するハードウェア開発は容易なものとなる。

JavaRock に入力される Java プログラムは、JVM (Java Virtual Machine) 上においてソフトウェアとしても動作可能であるため、ソフトウェア実行によるアルゴリズムレベルでのデバックが可能である。

JavaRock では、ソフトウェアとしての動作をハードウェアにトレースするために、Java プログラムにおける逐次処理をステートマシンで実現する。これにより、メソッド全体の処理を管理するステートマシンが生成され、ステートマシンの状態遷移により、処理対象となる Java コードが切り替わる。if 文や while 文をはじめとした制御構文に対

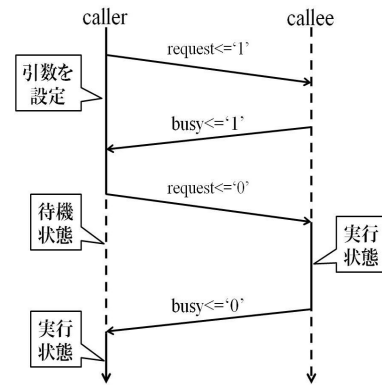


図 1 メソッド呼び出し

Fig. 1 Call of the method

してもステートマシンによる制御が行われ、Java プログラムの処理が実現される。

Java プログラムにおけるメソッド呼び出しに対しては、Java 言語のメソッドを VHDL の process 文に変換することで対応する。各メソッドに対して生成されるリクエスト信号やビジー信号を利用することで、実行制御が行われ、メソッドが呼び出される。メソッドを呼び出すための具体的な手順は以下のとおりである (図 1)。

- (1) caller 側はリクエスト信号をアサートし、callee 側のメソッド呼び出しをリクエスト
- (2) callee 側はリクエスト信号の確認後、ビジー信号をアサート
- (3) caller 側はビジー信号の確認後、メソッドの引数をセットし、リクエスト信号をネゲート
- (4) callee 側はリクエスト信号の確認後、メソッドを実行
- (5) callee 側は処理終了後、ビジー信号をネゲートし、戻り値を出力

また、JavaRock には、配列宣言された変数を BlockRAM に割り当てるなど、特別なクラスや型、文法を導入することなく、ハードウェア設計を行うための工夫がなされている。現在、JavaRock を活用した研究としては、ハードウェアとソフトウェアの協調設計 [15] や Reconfigurable Android 上での活用 [16]、ロボット制御システムの設計 [17] などが挙げられる。しかしながら、今後複雑なアルゴリズムや CFD (Computational Fluid Dynamics) に代表される高性能計算 (HPC) への利用をターゲットにすると、合成回路の増大化は避けられず、FPGA で実装するには回路分割機構が必要不可欠なものになっていくものと考えられる。

3. JavaRock の合成回路に対する分割処理

JavaRock は、合成回路に対する回路分割をサポートしていないため、合成回路が大規模化する場合やシミュレーション時間が膨大となる場合に問題となる。そこで、本研究では、JavaRock によるハードウェア開発によるコスト

をさらに削減するために、回路分割機構を JavaRock に付与することを考える。

3.1 Java プログラムの細分化

提案手法を適用するためには、Java プログラムを解析し、実行状態にあるモジュールを把握する必要がある。JavaRock の合成回路における実行モジュールの切り替えは、インスタンスに対するメソッド呼び出し時に起こる。そこで、インスタンスの呼び出し地点を分割点として回路分割を行うことが考えられる。しかしながら、大規模な Java プログラムにおいて複数のインスタンスが利用されることは一般的なことであるため、単純にインスタンスの呼び出し地点で回路を分割すると、分割数の増大を招くことになる。また、各分割回路が FPGA に実装可能な回路規模に収まるとは限らない。

そこで、提案手法では以下の 3 点を分割点候補とし、Java プログラムの細分化を行う。

- (1) インスタンスに対するメソッド呼び出し
- (2) 同一クラス内におけるメソッド呼び出し
- (3) メソッド内部のステートメント (for 文, while 文, if 文, switch 文)

分割点候補 (1) は実行モジュールの切り替え地点である。JavaRock の生成回路では、メソッド呼び出し時に実行制御が行われ、モジュール間で通信を行う。この通信時において、実行速度が低速なモジュールがボトルネックとなり、全体の動作速度が低速化する。したがって、各モジュールの動作速度を考慮することが望ましい。

分割点候補 (2) は分割点候補 (1) の場合と同様である。同一モジュール内においてメソッド間の通信が行われ、実行速度が低速なメソッドによりモジュール自体の動作速度が低速化する問題に対する。

文献 [11][12] では、プログラムのモジュールや関数をブロックとして回路分割を行うため、ブロックの回路規模が単一の FPGA に格納できないほど大きくなる場合には対処できない。この点は、クラスをベースに回路分割を行う文献 [13] も同様である。提案手法では、分割点候補 (3) でメソッド内部を分割し、FPGA に格納できる規模に制限する。

3.2 検証領域の判別

次に、3 つの分割点候補により細分化されたプログラムをブロック (以下では、分割ブロックと呼ぶ) として考え、統合アルゴリズムに適用する。ここでは、分割ブロックの統合結果に対する回路規模を閾値 (ユーザが指定した回路規模) 以下に制限する。

まず、ステートメントにより区切られた分割ブロックである分割点候補 (3) に対する処理を行う。このとき、処理の対象となる分割ブロックは、同一クラスに所属するもの

であり、回路規模が最小となる分割ブロックが統合対象とされる。ある統合対象と連続する分割ブロックに対して、統合した場合の回路規模を算出し、回路規模が閾値以下に収まる場合は、統合処理を行う。各クラスの分割ブロックに対する統合の余地がなくなるまで処理は継続する。

続いて、メソッド呼び出しにより区切られた分割点候補 (1) および分割点候補 (2) の分割ブロックを処理対象とする。ここでは、以下の手順に従い、統合処理を行う。

- Step1: メソッドの呼び出し元および呼び出し先の分割ブロック ($A = a_1, \dots, a_n$) をまとめる
- Step2: 回路規模が最小となる分割ブロック (a_i) を探索する
- Step3: 呼び出し元 (呼び出し先) と統合した場合の回路規模 (a'_i) を算出する
- Step4: a'_i が閾値以下に収まる場合は、統合処理を行う。閾値を上回る場合は、 A から a_i を除く
- Step5: $n(A) > 1$ である場合は、Step2 へ移行する。条件が満たされない場合は、処理を終了する

回路分割では、回路規模だけでなく、I/O ブロック数を考慮することが一般的である。しかしながら、提案手法ではラッパーを自動生成することにより、部分回路単体での検証を可能とするため、回路検証を行うために回路全体を動作させる必要はない。したがって、分割回路を実装した複数 FPGA 間の通信は行われず、I/O ブロック数に対する考慮が不要となる。自動生成されるラッパーの詳細に関しては次節で説明する。

3.3 回路検証用のラッパー

提案手法では、時系列に沿った回路分割を行うため、各部分回路での動作は、本来の Java プログラムの動作と一致する必要がある。本来の Java プログラムは連続しているため、プログラム内での変数情報は共有される。つまり、ある部分回路での処理結果が、次の部分回路の処理に反映される必要がある。

この部分回路間のデータ依存に対するために、ラッパーを自動生成し、分割点前後の変数情報に対する整合性を保証する。生成するラッパーの概念図を図 2 に示す。ラッパーは、初期値入力部と変数値比較部により構成される。各部位の役割は以下のとおりである。

初期値入力部: 各部分回路に対する初期値を入力する。

変数値比較部: 各部分回路に対する処理が終了した時点の変数値が正しいことを確認する。各変数値が Java コンパイル実行後の値と一致する場合は true、不一致の場合は false を出力する。

ラッパーを活用することで、部分回路単位での検証が可能となる。そのため、複数 FPGA 間での通信は不要とな

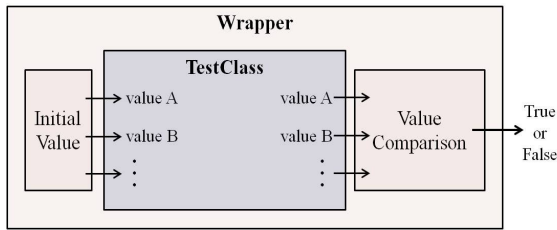


図 2 ラッパーの概念図

Fig. 2 Conceptual diagram of the wrapper

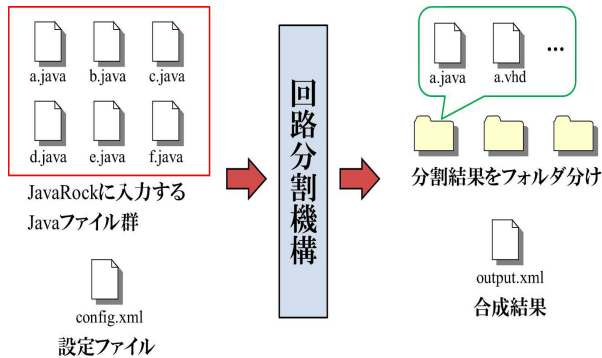


図 3 回路分割機構の全体図

Fig. 3 Overall view of the circuit partitioning mechanism

り、I/O ブロック数による制約から解放される。通信インタフェースを実装する手間はかからないため、大規模回路のための検証環境の構築は容易となる。

4. 検証を考慮した回路分割機構

本節で、実装および評価を行った回路分割機構の概要を説明する。回路分割機構の全体図を図 3 に示す。実装した回路分割機構では入力ファイルとして、Java ファイル群と設定ファイルを用いる。

設定ファイルでは、使用可能な FPGA デバイス名や FPGA のリソース使用率、回路分割機構の動作モードを指定する。FPGA のリソース使用量が大きくなると、論理合成時間の長大化および動作周波数の低速化を招く。そこで、回路分割機構では、分割後の各部分回路に対する FPGA のリソース使用量をユーザが任意に指定したリソース使用率以下に制限する。Java ファイルには、アプリケーションプログラムが記述されており、JavaRock に適用可能である。したがって、この Java ファイルを JavaRock で合成することで、VHDL ファイルを取得できる。また、回路分割機構は、合成モードと検証モードを備える。合成モードでは、入力された Java ファイルを JavaRock に適用し、JavaRock 本来の動作を行う。一方、検証モードでは、JavaRock が合成した回路を部分回路に分割し、検証用回路を生成する。

回路分割機構は、入力された Java ファイルと設定ファイルを基に回路分割を行う。分割後、実行順に番号付けを行

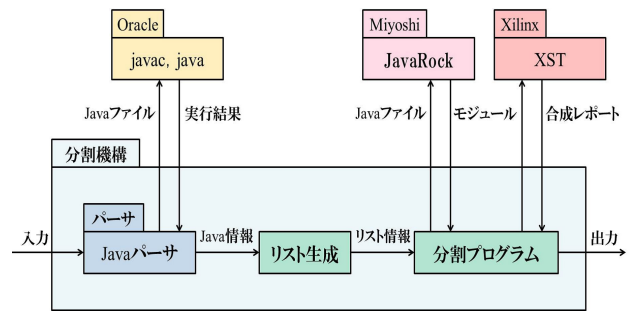


図 4 回路分割機構の構成

Fig. 4 Configuration of the circuit partitioning mechanism

い、Java ファイルと VHDL ファイルをフォルダ分けして出力する。この時、Java ファイルに記述されるのは、その時点において実行される分割ソースコードである。VHDL ファイルは、この Java ファイルを JavaRock で合成した際の出力である。

回路分割機構の構成を図 4 に示す。はじめに、入力として与えられた Java ファイルをパーサで解析する。提案手法において、各分割ブロックにおける変数情報の取得は不可欠である。しかしながら、ある時点における変数情報を Java ソースコードから見積もることは困難である。そこで、パーサでは各分割ブロックに対する Java コンパイルを実行し、該当プログラムに対する実行後の変数情報の取得を行う。ここでは、Oracle 社が提供する Java の開発環境 [18] における javac および java コマンドを使用する。パーサは解析を行った後、Java の情報をリスト生成部に渡す。

リスト生成部において、Java の情報は整理される。時系列に沿ってリストを生成し、各分割ブロックにおける変数の初期値と終了値を設定する。作成されたリストは、次の分割プログラムに渡される。

分割プログラムにおいて回路分割が行われるが、Java の情報から JavaRock が生成する回路の規模を見積もることは困難である。そこで、HDL に変換するために JavaRock で合成後、論理合成ツールを実行する。回路分割機構では、Xilinx 社の XST (Xilinx Synthesis Technology) [19] を呼び出す。XST は論理合成結果をレポートとして出力し、レポートには、レジスタ数や LUT 数、BlockRAM 数などの回路規模に関する情報が記載される。このレポートから回路規模の情報を取得し、回路分割を行う。

5. 評価

5.1 評価環境

本実験で用いる FPGA は Xilinx 社の FPGA である XC3S1200E である。JavaRock で合成した回路に対する回路分割を行い、回路規模を XC3S1200E のリソース以下に制限する。XC3S1200E のリソース量は表 1 に示す。実験

表 1 XC3S1200E のリソース量

Table 1 Amount of resources in XC3S1200E

	個数
スライス数	8672
レジスタ数	17344
LUT 数	17344
Block RAM 数	28

表 2 評価用回路のリソース使用量

Table 2 Amount of resources in the target circuit for evaluation

	評価用回路	FPGA リソース	利用率
スライス数	8981	8672	103%
レジスタ数	5885	17344	33%
LUT 数	16941	17344	97%

表 3 回路規模を 75%以下に制限した場合のリソース量

Table 3 Amount of resources in the case of the limitation on 75% or less

	スライス数	レジスタ数	LUT 数
partition1	5995	3737	11932
partition2	2515	2123	5165

に用いた PC のスペックは, Intel Core i5-750 Processor 2.66GHz, メインメモリは 4GB, OS は Windows7 である. 論理合成には XilinxISE14.4 を使用し, シミュレーションには ISim14.4 を使用した.

実装した回路分割機構の評価を行うため, FFT を実行するプログラムを作成した. FFT プログラムを JavaRock で合成した場合における, 評価用回路に対する FPGA のリソース使用量は表 2 に示す. スライスの利用率が 100%を超えているため, 1 つの FPGA に格納することは不可能である. この回路に対して回路分割を行い, 回路規模を 75%以下に制限する場合と, 回路規模が 50%以下に制限する場合の二通りの実験を行った.

5.2 評価結果

評価用回路に対して, 閾値を 75%および 50%に設定し, 回路分割を行った結果が図 5 と図 6 である. また, 各分割結果を表 3 と表 4 に示す. 元となった Java プログラムの実行順に回路が分割され, 各部分回路の回路規模は閾値以下に制限される. 各部分回路に対する統合処理をさらに実行した場合, 閾値で設定された回路規模を上回る構成となっている.

次に, 各分割結果に対する動作をシミュレーションおよび実機上で確認する. ここでは, 閾値を 50%に設定した際の分割結果である partition1 のシミュレーションおよび実機上での動作を図 7 と図 8 に示す. partition1 は, 2 つのモジュールにより構成されており, 処理が終了した時点の変数値比較がラッパーにより行われる. ラッパーに内

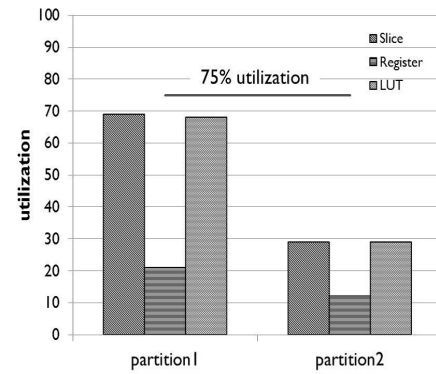


図 5 回路規模を 75%以下に制限した場合の利用率

Fig. 5 Utilization in the case of the limitation on 75% or less

表 4 回路規模を 50%以下に制限した場合のリソース量

Table 4 Amount of resources in the case of the limitation on 50% or less

	スライス数	レジスタ数	LUT 数
partition1	2489	1985	4951
partition2	3261	2375	6475
partition3	2515	2123	5165

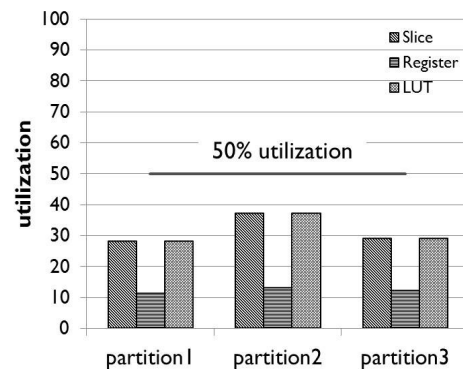


図 6 回路規模を 50%に制限した場合の利用率

Fig. 6 Utilization in the case of the limitation on 50% or less

包される各モジュールの検証が行われ, 図 7 中 (1) において true が出力された. また, ラッパーが出力する信号を XC3S1200E の LED に接続し, 実機上での動作確認を行った結果, 図 8 中 (2) において LED が点灯した. したがって, partition1 に該当する回路が正常に動作することが確認された.

partition1 以外の分割結果に対しても同様にシミュレーションおよび実機上で動作検証を行った結果, 評価用回路全体が正常に動作することが確認された.

6. 考察

6.1 回路規模の増減

以上の結果から, 入力された Java プログラムを合成した回路に対する回路分割が行われるといえるが, 分割後の回

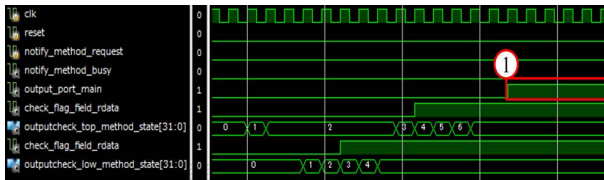


図 7 シミュレーション上での動作
Fig. 7 Operation in the simulation

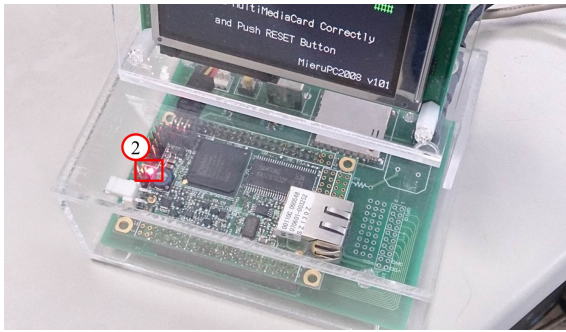


図 8 実機上での動作
Fig. 8 Operation in the actual machine

路規模は分割前の回路規模とは異なる値をとる。分割後の回路規模が変化した要因として、以下の3点が考えられる。

- (1) ラッパーの実装
- (2) 回路の非共有
- (3) 実行制御機構の削減

1点目は、ラッパーが分割前の回路には存在しないモジュールであることに起因する。また、内包されるモジュールの複雑化に伴い、ラッパー自体が複雑化するため、リソースの消費は大きなものとなる。2点目は、論理合成ツールによる最適化が十分に行われないことによる。本来ならば、同一機能を有する回路に対しては、リソース共有が行われる。しかしながら、分割後の回路が異なる集合に属した場合、リソース共有は不可能となり、同一機能の回路を複数用意することとなる。そのため、回路規模は増大する。一方、3点目は、分割回路における実行モジュールの切り替えが削減されたことを意味し、リソースは削減される。JavaRockの合成回路における実行制御機構は、リクエスト信号やビジー信号を利用したポーリングにより実現されるため、リソースの消費は大きなものとなる。提案手法では、回路の実行状態を考慮した回路分割を行うことで、実行制御機構は最小限に抑えられる。

以上のように、分割前の回路と比較して回路規模は変化しているが、シミュレータによるシミュレーションは実行可能である。FPGAに実装する場合においても、分割後の回路はより多くのFPGAのリソースを扱うことが可能であるため、回路規模の増大による影響は小さいと考えられる。

シミュレーションを行うことで、回路の動作を検証することが可能であるが、実機上で回路が正常に動作することを保証できない。そこで、FPGAに対する分割回路の実装が行われるが、I/Oブロック数制約を考慮する必要がある。回路全体を動作させることは容易ではない。提案手法では、各部分回路はラッパーに内包されるため、独立してFPGAに実装可能である。したがって、分割回路全体を動作させる必要はなく、通信インタフェースを実装する手間はかからない。そのため、部分回路の検証は容易なものとなる。また、LEDやChipscopeを活用し、ラッパーの出力信号を観測することで、部分回路に対するデバックは簡単化される。

6.2 部分回路への分割

FPGAを用いた回路検証において、回路の動作に不具合が発生した場合、問題箇所を特定することは困難である。FPGA上で動作する信号を可視化できないため、シミュレーションを用いて回路の動作を確認する必要がある。しかしながら、大規模回路のシミュレーションに要する時間は膨大であり、実用的ではない。

一方、提案手法では、部分回路単位での検証が可能であり、動作不良を起こしている回路の切り分けが容易である。ラッパーの出力信号をFPGAのLEDに接続することで、部分回路レベルでの可視化が可能である。また、提案手法では、FPGAのリソース使用率による分割が可能であるため、不具合のある回路をさらに細分化し、問題箇所を特定できる。したがって、細分化した部分回路に対するシミュレーションを行うだけでよく、回路検証に要する時間は短縮されることが考えられる。

また、部分回路への分割により、ボトルネックとなる処理を探索できる。該当部分に対するHDLへの書き換えやJavaソースコードのアルゴリズムの変更をユーザは選択でき、FPGAの処理能力を生かすことが可能である。

6.3 分割後の動作周波数

提案手法により分割された回路に対する動作周波数を以下の表5と表6に示す。その結果、分割前の回路との比較において、分割後の回路における動作周波数が向上することが観測される。分割前の回路は、論理合成ツールの最適化によりリソースが共有されるため、回路構成が複雑である。一方、分割後の回路は、割り当てられたリソースの占有が可能である。また、JavaRockの合成回路における実行制御機構が削減されるため、回路構成は単純化する。以上により、クロック周期が削減されたと考えられる。

7. まとめ

本論文では、高位合成ツールJavaRockの合成回路の検証を容易にする回路分割機構について述べた。細分化した

表 5 回路規模を 75%以下に制限した場合の動作周波数

Table 5 Operation frequency in the case of the limitation on 75% or less

	default	partition1	partition2
動作周波数	49MHz	71MHz	86MHz

表 6 回路規模を 50%以下に制限した場合の動作周波数

Table 6 Operation frequency in the case of the limitation on 50% or less

	default	partition1	partition2	partition3
動作周波数	49MHz	78MHz	74MHz	86MHz

Java プログラムを JavaRock により合成し、ラッパーを自動生成することで、部分回路単位での検証を可能とした。実験では、FFT プログラムを対象とし、指定した FPGA のリソース使用量が 50%および 75%に収まるように回路を分割した。また、各分割結果に対して、シミュレーションおよび実機上で正常に動作することを確認した。

今後の課題としては、提案手法を他の高位合成ツールに適用し、有効性を確認することが挙げられる。提案手法を適用可能な高位合成ツールの条件として、以下の 2 点が考えられる。

- (1) 入力プログラムをソフトウェアとして動作可能
- (2) 高位合成ツールの合成回路に対する実行制御が存在

条件を満たす高位合成ツールとして、JavaRock-Thrash[20] が挙げられる。JavaRock-Thrash とは Java 言語をベースとする高位合成ツールであり、Java ソースコードから Verilog-HDL の生成を行う。また、リストスケジューリングやレフトエッジ法によるレジスタバインディングを導入するなど、回路の省資源化や高速化が図られている。現在、JavaRock-Thrash に提案手法を適用することを考えている。

謝辞 本研究の一部は、文部科学省特別経費「持続可能社会にむけた知的情報空間技術の創出」及び JSPS 科研費基盤研究 (C) 25330067 による支援を得た。ここに記して感謝する。

参考文献

- [1] SystemC: <http://www.systemc-japan.com/>.
- [2] Impulse accelerated technology: <http://www.impulseaccelerated.com/>.
- [3] CyberWorkBench: <http://jpn.nec.com/cyberworkbench/>.
- [4] Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., Brown, S. and Czajkowski, T.: LegUp: high-level synthesis for FPGA-based processor/accelerator systems, *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 33–36 (2011).
- [5] JHDL: <http://www.jhdl.org/>.
- [6] Auerbach, J., Bacon, D. F., Cheng, P. and Rabbah, R.: Lime: a Java-compatible and synthesizable language for heterogeneous architectures, *Proceedings of the ACM in-*

- ternational conference on Object oriented programming systems languages and applications*, pp. 89–108 (2010).
- [7] JavaRock: <http://javarock.sourceforge.net/>.
- [8] Kernighan, B. W. and Lin, S.: An Efficient Heuristic Procedure for Partitioning Graphs, *Bell System Technical Journal*, Vol. 49, No. 2, pp. 291–308 (1970).
- [9] Fiduccia, C. M. and Mattheyses, R. M.: A linear-time heuristic for improving network partitions, *DAC '82 Proceedings of the 19th Design Automation Conference*, pp. 175–181 (1982).
- [10] Babb, J., Tessier, R. and Agarwal, A.: Virtual wires: overcoming pin limitations in FPGA-based logic emulators, in *Proc. of IEEE Workshop on FPGA-based Custom Computing Machines*, pp. 142–151 (1993).
- [11] Kugami, D., Miyajima, T. and Amano, H.: A Circuit Division Method for High-Level Synthesis on Multi-FPGA Systems, *Proceedings of the 2013 27th International Conference on Advanced Information Networking and Applications Workshops*, pp. 156–161 (2013).
- [12] 國上太旗, 宮島敬明, 天野英晴: 高位合成を用いたストリーム処理におけるマルチ FPGA システム向け回路分割手法の提案, 電子情報通信学会技術研究報告, Vol. 113, No. 324, pp. 53–58 (2013).
- [13] 松田和也, 三好健文, 船田悟史, 中條拓伯: 高位合成ツール JavaRock の回路分割システム, 組込みシステムシンポジウム 2013 論文集, Vol. 2013, pp. 49–54 (2013).
- [14] 三好健文, 船田悟史: FPGA 向け高位合成言語としての Java の活用手法の検討, 第 53 回プログラミング・シンポジウム予稿集, Vol. 2012, pp. 59–68 (2012).
- [15] 三好健文, 船田悟史: JavaRock を用いた HW/SW の協調設計の検討, 電子情報通信学会技術研究報告 AI, Vol. 112, pp. 119–124 (2012).
- [16] 榎戸健二, 三好健文, 小池恵介, 船田悟史, 藤波香織, 中條拓伯: 高位合成系 JavaRock による Reconfigurable Android におけるハードウェア・アクセラレーション, 情報処理学会論文誌「組込みシステム工学」特集号, Vol. 55, No. 2, pp. 1027–1036 (2014).
- [17] 植竹大地, 大川猛, 三好健文, 横田隆史, 大津金光: 高位合成ツール JavaRock による倒立振り子制御処理の高速化, 電子情報通信学会技術研究報告 Reconfigurable Systems, Vol. 113, pp. 55–60 (2013).
- [18] Oracle: <http://www.oracle.com/>.
- [19] Xilinx: <http://japan.xilinx.com/>.
- [20] 小池恵介, 三好健文, 船田悟史, 中條拓伯: Java 言語ベース高位合成ツール JavaRock-Thrash の開発, 組込みシステムシンポジウム 2013 論文集, Vol. 2013, pp. 41–48 (2013).