Java 言語ベース高位合成ツール JavaRock-Thrash の開発

小池　恵介 †　三好 健文 ††　船田　悟史 ††
中條　拓伯 †

近年 LSI 設計が複雑化するにつれて，高位合成技術に注目が集まっている．しかし，高位合成ツールによる開発は，HDL による開発と比べ，回路の処理速度やリソースの面で大きく劣る場合がある．そこで，本研究では，処理速度向上や回路規模削減のための機能を備えた Java ベースの高位合成ツール JavaRock-Thrash を開発する．本ツールを用いて，FIR フィルタと FFT を行う回路を記述して，論理合成，配置配線を行い IP コアと比較したところ，回路規模において 1.6〜4.0 倍に増えたものの，処理速度の低下は約 1.7 倍に抑えられた．この結果から，本ツールは，ハードウェア・アクセラレーションの研究やプロトタイピングなどの様々な分野に適用できると考えられる．

Development of JavaRock-Thrash : a High Level Synthesis tool based on Java language

Keisuke Koike ,† Takefumi Miyoshi ,†† Satoshi Funada ††
and Hironori Nakajo †

In recent years, as designing an LSI is getting more complex, high level synthesis has been expected to be a useful designing tool. However, running frequency and occupied resource of a circuit designed with a high level synthesizer are sometimes inferior to those designed with an HDL. In this research, we have been developing a high level synthesizer called JavaRock-Thrash with mechanisms for speedup and reducing circuit scale based on designing with a Java language. In designing circuits of an FIR filter and an FFT with our tool, comparing with designing with IP cores, though circuit sizes are 1.6 - 4.0 times as large as ones of IP cores, execution time is limited up to 1.7 times slower. From this results, our tool can be expected to be applied for hardware acceleration or prototyping of LSI design.

1. はじめに

近年，LSI 設計の複雑化に伴い，高位合成技術に注目が集まっている．高位合成技術とは，RTL よりも高い抽象度で回路の動作を記述し，ハードウェア設計を可能とする技術である．高位合成技術の主な利点は，その高い記述性による設計期間の短縮や可読性の向上である．また，高位合成ツールに対し，利用するリソース量の指定ができる場合，規模の異なる FPGA への移植が容易になるという利点もある．

高位合成ツールの適用場面として，FPGA や ASIC 開発，およびその開発過程でのプロトタイピングなどがあげられる．特に，ソフト/ハード開発において，ハードウェアのプロトタイプを短期間で開発できれば，ソフトウェアの開発も実機を用いて効率的に行えるため，高位合成ツールを利用するメリットは大きい．また，その他の適用場面としては，FPGA アクセラレーションの研究などがあげられる．具体的には，高位合成ツールを用いて，HDL での記述が困難なアルゴリズムをハードウェア化したり，FPGA クラスタの性能評価用アプリケーションを迅速に開発するといったことが可能になる．

しかし，高位合成ツールによって作成された回路は，手書きの HDL から作成されたものと比べ，回路規模や処理速度の面で大きく劣る場合がある．Stream-C[5] では，手書きの HDL と比べて，回路規模が 3 倍に増え，動作周波数が 2 倍以上遅くなったという結果が示されている．さらに，以下にあげるような，高位合成ツールの習得コストも無視できない．

(1) 高位合成ツールの開発言語を習得する手間
(2) シミュレータなどの開発環境に慣れ親しむ手間
(3) 高性能な回路を設計するためのコツやコーディング方法を理解する手間

(3) に関して，高位合成ツールを用いて高速な回路を

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

設計するためには，ハードウェア化を意識した記述が必要となる場合が多く，そのためには，ツール固有の機能や特徴を理解しなければならない．これについて，文献 7) では，高位合成ツールに入力するコードに対し，ループ展開やアルゴリズムの工夫をした方が，何も施さない場合と比べ，実行時間が短くなるという結果が示されている．

前述したように，高位合成ツールの適用場面は多岐にわたり，特にプロトタイピングや研究用途では，回路の性能を追求するよりも短期間で回路が作成できることが求められる．その際，問題となるのは，動作速度の低下や回路規模の増加よりも習得コストにあげた (1) と (2) である．この問題に対し，JavaRock[12)13)] のアプローチは，Java 言語を開発言語とし，その文法や型には手を加えず，VHDL のソースコードに変換する方針を採っている．これにより，Java の開発経験があれば，文法を覚える必要はなく，さらに，回路の開発および動作確認に一般的な Java の開発環境が使える．勿論，Java からハードウェアへの変換規則やハードウェア化に伴う記述の制限などを覚える必要はあるが，このことは抽象度の高い言語からハードウェアを作成する以上仕方のないことである．

本研究では，文法や型を拡張しない Java 言語を開発言語とする JavaRock の設計方針を受け継いだ上で，Java のソースから Verilog HDL のコードに変換し，さらに，処理速度向上や回路規模削減のための機能を備えた高位合成ツール JavaRock-Thrash を開発する．JavaRock-Thrash は，習得コストを抑えるだけでなく，高速に動作する回路や低リソースを目指した回路も設計できるため，回路の性能が要求される場合にも適用できる．

2. 関 連 研 究

2.1 既存の高位合成ツール

現在，商用，研究用を含め多数の高位合成ツールが開発され，利用されている．Java 言語ベースのものとしては，JavaRock[12)13)] や MaxCompiler[2)], Liquid Metal[3)], Sea Cucumber[10)] などがある．本節では，Java 言語ベースの高位合成ツールの特徴を述べ，JavaRock-Thrash との違いについて説明する．

第 1 章で述べたように，JavaRock と JavaRock-Thrash は，ともに開発言語である Java の型や文法を拡張しないという方針をとっている．この 2 つの高位合成ツールの違いとして，JavaRock が VHDL を出力するのに対し，JavaRock-Thrash は，Verilog HDL を出力するという点があげられる．しかし，最も大きな違いは，JavaRock が演算レベルの並列性を抽出しないのに対して，JavaRock-Thrash では Data Flow Graph （DFG）を用いて演算レベルの並列性を抽出するということである．JavaRock では依存関係の無いいくつかの文が 1 ステートに割り当てられるため，1 つの文に多くの演算を記述するとその処理からクリティカルパスが構築される可能性がある．一方，JavaRock-Thrash では，文を複数のステートに分割するため，1 つの文に多数の演算を記述してもクリティカルパスが構築されない．

MaxCompiler[2)] と Liquid Metal[3)] はストリーム計算を高速に実行するためのプログラミングモデルを提供している．しかし，Java は本来ストリーム処理を記述できる言語ではないため，この 2 つの高位合成ツールでは，ストリーム計算用の型や文法の拡張を行っている．また，これらを拡張すると，一般的な Java の実行環境で実行できなくなるため，専用のシミュレータ上で動作確認をしなければならない．一方，JavaRock-Thrash は，型や文法の拡張を行わないため，専用のシミュレータも不要である．したがって，これらの高位合成ツールと比べ，JavaRock-Thrash は習得コストが低い．

しかし，ストリーム計算は，パイプライン処理が可能で ハードウェアでの実装に向いているため，高位合成ツールには，これを高速に処理する手段が必要となる．そこで，JavaRock-Thrash は，ストリーム計算を高速に処理する手段として，ループ展開機能を備えており，これについては 5.2 節で紹介する．

Sea Cucumber[10)] は JavaRock と同じく，型や文法の拡張を行わず，一般的な JVM で回路の動作確認を行う方針をとっている．また，Java スレッドを用いて粗粒度並列性を記述し，高位合成ツールが細粒度並列性を抽出するという方針も，JavaRock-Thrash と同じものである．JavaRock-Thrash との相違点の一つは，入出力ファイルである．Sea Cucumber が Java のクラスファイルを入力し，ネットリストが出力されるのに対し，JavaRock-Thrash では，Java のソースファイルを入力し，Verilog HDL ファイルが出力される．出力ファイルがネットリストであることに対して，文献 10) において，高性能な論理合成ツールが HDL ファイルを最適化した場合と比べると，回路品質が低下すると述べられている．また，JavaRock-Thrash が new 演算子でオブジェクトを生成し，サブモジュールを定義するのに対し，Sea Cucumber にはそのような機能がない．しかし，Java のクラスの包含関係を用いて Verilog HDL のモジュールの包含関係を記述す

るという手法は，プログラマにとって分かりやすく有効であると考えられる．

2.2 スケジューリング，バインディングアルゴリズム

高位合成ツールの性能を決める最も重要な処理がスケジューリングとバインディングである．スケジューリングとは，DFG の各演算の実行サイクルを決定する処理である．スケジューリングアルゴリズムは，時間制約を与えるものと演算器数などのリソース制約を与えるものに大別される．時間制約を与えるものには，整数線形計画法（ILP）を解くもの[9] や Force-Directed Scheduling[8] がある．一方，リソース制約を与えるものには，ASAP[1] や，リストスケジューリング[6] などがある．このうち，JavaRock-Thrash では，リストスケジューリングを採用した．その理由は，回路の省資源化のためには，リソース制約を与える必要があり，また，リストスケジューリングは，ASAP よりも実行サイクル数の小さな解が得やすいからである．

バインディングとは，DFG の演算や変数に対し，演算器やレジスタといった実際のハードウェアコンポーネントを割り当てる処理である．レジスタバインディングのアルゴリズムには，クリーク分割問題を解くもの[1] や 2 部グラフの最適マッチング問題を解くもの[11]，レフトエッジ法[1] などが存在する．このうち，レフトエッジ法は多項式時間で，必要となる最小のレジスタ数を求めることができる．しかし，1 つのレジスタを複数のステートで共有すると，たくさんのマルチプレクサが必要となり，回路規模の増加や動作周波数の低下を招く．そこで，JavaRock-Thrash では，レジスタの最大シェアリングステート数を指定できるようにした上で，レジスタバインディングにレフトエッジ法を採用した．これにより，FPGA 上のレジスタと LUT をバランスよく利用することができる．

3. JavaRock-Thrash のコンパイルフロー

図 1 に JavaRock-Thrash のコンパイルフローを示す．JavaRock-Thrash は，Java のソースファイルと config ファイルを入力ファイルとし，Java のクラスファイルと Verilog HDL ファイルを出力する．config ファイルは，回路を生成する際の設定を記述するファイルで，xml 形式となっている．設定可能な項目としては，各演算 IP のパラメータや各演算のチェイニングの有効/無効などがある．クラスファイルは JVM 上で実行可能であるため，生成された回路の動作確認に利用できる．また，Verilog HDL ファイルは，ハードウェア化可能な形式で記述されているため，論理シミュレーションや論理合成に利用できる．

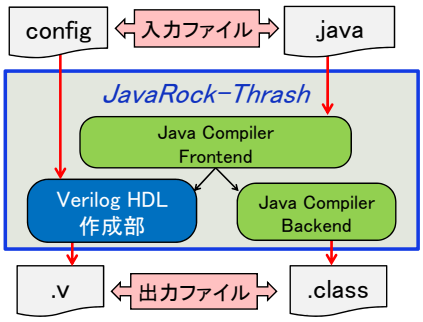

図 1 JavaRock-Thrash のコンパイルフロー
Fig. 1 compilation flow of java

```
1  class TopClass{
2
3    int numX;
4    final int[] arrayX = new int[1024];
5    final SubClass sub = new SubClass();
6  }
7
8  class SubClass{
9
10   @JRThrashExtToTop
11   short numA;
12
13   @JRThrashReadOnlyPort
14   double numB;
15 }
```

図 2 メンバ変数変換例（Java ソースコード）
Fig. 2 An example of converting member variables (Java source code)

図 1 の中の Verilog HDL 作成部が本研究で開発した部分である．Java コンパイラのフロントエンドとバックエンドは OpenJDK のコードを利用しており，Verilog HDL 作成部はこのフロントエンドが作成するする構文木をもとに DFG, CFG（Control Flow Graph）を作成し，スケジューリングやバインディングなどを行った後，Verilog HDL コードを生成する．

4. JavaRock-Thrash のコード変換規約

本章では，JavaRock-Thrash が Java ソース中に記述したメソッドや変数をどのようなハードウェアコンポーネントに変換するかについて説明する．

4.1 メンバ変数の変換

JavaRock-Thrash は，クラスごとに Verilog HDL のモジュールを作成するため，図 2 の TopClass と SubClass は図 3 の TopClass モジュールと SubClass モジュールに変換される．また，JavaRock-Thrash では，実行時にヒープを確保するような Java の動的な

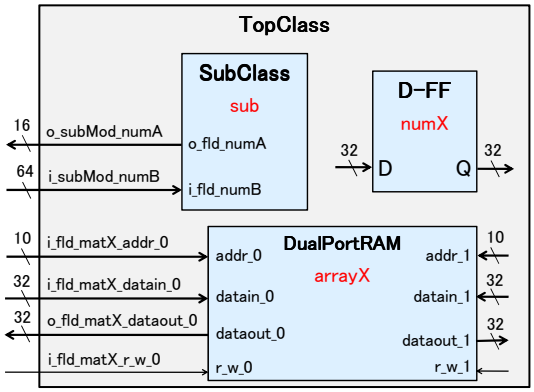

図 3 メンバ変数変換例（ブロック図）
Fig. 3 An example of converting member variables (block diagram)

```
1  class ClassX{
2
3    int average(int dataNum, int[] matX){
4           ・・・
5    }
6  }
```

図 4 メソッドとパラメータリストの変換例（Java ソースコード）
Fig. 4 An example of converting method and parameter list (Java source code)

振る舞いをサポートしていないため，クラスや配列のインスタンス化は，final をつけたメンバ変数として記述する．以下に，メンバ変数として宣言したプリミティブ型変数，配列，オブジェクトの変換規則を記す．

- プリミティブ型変数 → レジスタ
- 配列 → デュアルポート RAM（FPGA の内部 RAM を利用）
- オブジェクト → サブモジュール

図 2 の SubClass に示したアノテーションについて説明する．JRThrashExtToTop アノテーションは，プリミティブ型変数から作成されるレジスタの出力や，配列から作成される RAM の制御信号をトップモジュールの I/O ポートに接続することができる．JRThrashReadOnlyPort アノテーションは，プリミティブ型変数を入力ポート化し，トップモジュールの I/O ポートに接続することができる．

4.2 メソッドおよびパラメータリストの変換

JavaRock-Thrash では，各メソッドごとに処理の開始を命令する入力ポート（メソッド名_req）とメソッドが処理中かどうかを示す出力ポート（メソッド名_busy）

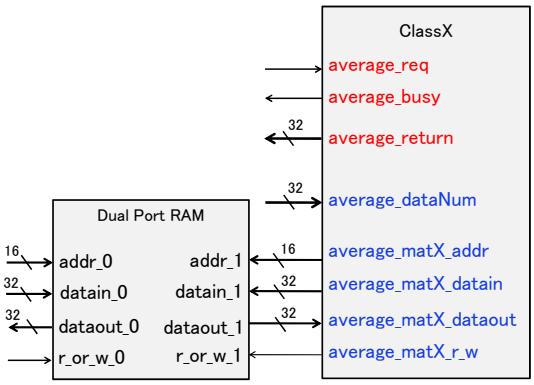

図 5 メソッドとパラメータリストの変換例（ブロック図）
Fig. 5 An example of converting method and parameter list (block diagram)

が作成される．また，メソッドの戻り値がプリミティブ型の場合，戻り値の出力ポート（メソッド名_return）が作成される．図 4 の average メソッドから作成されるポートを図 5 に赤字で示す．

パラメータリストに引数を宣言すると，それぞれの引数に対応した入出力信号が作成される．引数がプリミティブ型の場合，入力ポートが宣言され，メソッドの開始時にこのポートの値がモジュール内部のレジスタに格納される．一方，引数が配列型の場合，モジュール外部の RAM 制御用の 4 つの入出力ポートが宣言される．図 4 のパラメータリストから作成されるポートを図 5 に青字で示す．また，配列型の引数から作成されたポートとモジュール外部の RAM との接続関係を図 5 に示す．

5. 高速化のための機能

JavaRock-Thrash では，並列処理を記述する仕組みとして Java スレッドとループ展開をサポートしている．Java スレッドを用いると，複数のモジュールが並列に動作するため，空間的並列性が向上する．一方，ループ展開を用いると，パイプライン処理が構築されるため，時間的並列性が向上する．

また，並列処理のほかに，既存の RTL の利用も高速化に対し有効である．JavaRock-Thrash では，算術演算やキャスト演算などの演算子に対し，対応する演算 IP を自動的に割り当てるため，これらの IP コアは Java の演算子を記述するだけで利用可能である．しかし，対応する演算子が存在しない IP コアを利用したい場合もある．例として，文献 7) では乱数生成器を RTL 化し，それを高位合成ツールが利用することで高速化を行っている．これに対し，本ツールではメソッドの演算器化をサポートしている．

```
1  class TopClass{
2
3    final SubClass subA = new SubClass();
4    final SubClass subB = new SubClass();
5
6    void startThreads(){
7
8      //スレッドの処理開始
9      subA.start();
10     subB.start();
11
12     try{
13       //スレッドの終了待ち
14       subA.join();
15       subB.join();
16     }
17     catch(Exception e){}
18   }
19 }
20
21 class SubClass extends Thread{
22
23   public void run(){
24         ...
25   }
26 }
```

図 6 Java スレッド記述例
Fig. 6 A description example of Java Thread

5.1 スレッドを用いた空間的並列性の記述

図 6 に Java スレッドを用いた並列処理の記述例を示す．まず，TopClass が SubClass のオブジェクト subA, subB を宣言することで，TopClass モジュール内部に 2 つの SubClass モジュールがインスタンス化される．さらに，Thread クラスを継承した SubClass の start メソッドを呼び出すことで，SubClass の run メソッドが実行される．通常，メソッド呼び出しを行うと，呼び出し元は呼び出したメソッドの終了を待つために処理を停止するが，スレッドの start メソッドを呼び出した場合，呼び出し元はその終了を待たずに処理を続行する．したがって，図 6 の 10 行目以降，2 つの SubClass モジュールが同時に処理を行う．また，スレッドの終了を待つ際は，スレッドの join メソッドを呼び出す．現在，synchronized による排他制御や wait，notify の機能はないが，今後実装する予定である．

5.2 ループ展開を用いた時間的並列性の記述

図 7 にループ展開を用いた並列処理の記述例を示す．図 7 のメソッド multStream に記された JRThrashUnroll がループ展開を指定するアノテーションである．アノテーションにより，展開対象となったループは，JavaRock-Thrash の内部で図 8 のように変換される．

```
1  @JRThrashUnroll(
2     unrollType = JRThrash.pipeline
3     unrollNum = 4,
4     loopVariableName = "i")
5
6  public void multStream(){
7
8     for(int i=0; i<128; ++i)
9       c[i] = a[i] * b[i];
10 }
```

図 7 ループ展開記述例
Fig. 7 A description example of loop unrolling

```
1  public void multStream(){
2
3     for(int i=0; i<128; i+=4){
4       c[i]   = a[i]   * b[i];
5       c[i+1] = a[i+1] * b[i+1];
6       c[i+2] = a[i+2] * b[i+2];
7       c[i+3] = a[i+3] * b[i+3];
8     }
9  }
```

図 8 展開後のループ
Fig. 8 Loop after unrolling

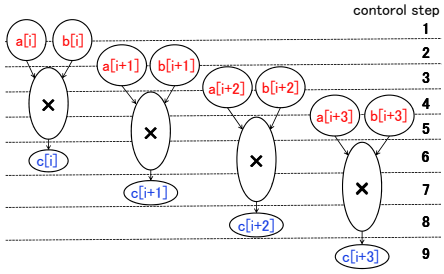

図 9 展開されたループのスケジューリング例
Fig. 9 An example of scheduling the unrolled loop

ループ展開アノテーションには，unrollNum，loopVariableName，unrollType の 3 つの引数がある．unrollNum は展開する式の個数を指定する引数で，loopVariableName は展開するループのループ変数を指定する引数である．unrollType は，式の展開方法とイテレーション間の依存関係の有無を指定する引数である．

図 8 を毎クロック演算可能な乗算器を用いてスケジューリングすると，図 9 のようになる．図 9 では，コントロールステップが 3〜6 の間，乗算器に毎クロック配列 a と b の値が入力されている．このように，for ループがパイプライン化されることで，演算密度を向上させることができる．

5.3 メソッドの演算器化

図 10 にメソッドの演算器化の記述例を示す．図 10

```
1  @JRThrashConvertedIntoIPcore(
2     availableNum = 45, latency = 0,
3     throughput = 1, outputPName = "p",
4          ...
5  )
6  int MAC(short a, short b, int c){
7     return (int)(a*b+c);
8  }
9
10 public void macStream(){
11
12    for(int i=0; i<128; ++i)
13      w[i] = MAC(x[i], y[i], z[i]);
14 }
```

図 10 メソッドの演算器化記述例
Fig. 10 A description example of defining a computing unit with a method

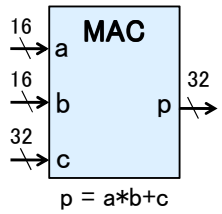

図 11 MAC メソッドから定義される演算器
Fig. 11 A computing unit defined with a MAC method

のメソッド MAC についている JRThrashConverted-IntoIPcore がメソッドの演算器化を行うアノテーションである．このアノテーションの引数には，演算 IP の利用個数やレイテンシ，スループットなどを設定するものがあり，組み込む IP コアのインタフェースや処理性能を細かく指定できる．

図 10 のメソッド MAC は，図 11 のようなポートを持つ演算 IP として扱われ，JavaRock-Thrash の出力するモジュール内部にインスタンス化される．そして，この演算 IP を利用する際は，メソッド macStream にあるように MAC を呼び出せばよい．また，IP コアの動作を演算器化するメソッドの処理部分に記述することで，ソフトウェアシミュレーションも可能である．

6. 省資源化のための機能

JavaRock-Thrash では，config ファイルの設定により，回路規模の削減が可能である．config ファイルで設定可能な項目のうち，回路規模に影響を与えるものを以下に示す．

- 各演算器の最大利用個数
- 各演算器のポートビット数
- 各演算器のスループット，レイテンシ
- プリミティブ型のビット幅
- レジスタの最大シェアリングステート数

各演算を実行する演算器数を制限し，複数のステートで共有することで，DSP ブロックのような数の少ないリソースを有効活用できる．しかし，演算器数を少なくしすぎると，共有するステート数が多くなり，マルチプレクサの数が増加する．したがって，演算器数は，全体の回路規模が小さくなるように設定しなければならない．

各演算 IP のポートビット幅やスループット，レイテンシを変更することで，演算 IP の回路規模を小さくできる．例えば，演算 IP のポートビット幅を処理に必要な最低限のビット幅に制限すれば，小さな演算 IP を利用することができる．しかし，計算する値が演算 IP のポートビット幅を超えている場合，処理結果がソフトウェアシミュレーションと一致しなくなるため注意が必要である．

プリミティブ型のビット幅を小さくすると，プリミティブ型変数から作成されるレジスタや配列から作成される RAM のビット幅が小さくなる．したがって，フリップフロップや RAM ブロックのリソースが削減できる．しかし，格納される値がレジスタや RAM のビット幅を超えている場合，処理結果がソフトウェアシミュレーションと一致しなくなるため注意が必要である．

第 2.2 小節でも述べたように，JavaRock-Thrash では，レジスタバインディングにレフトエッジ法を用いている．レフトエッジ法は必要となるレジスタ数を最少化することができるが，レジスタのシェアリングステート数が増えるとマルチプレクサの数が増加するため，レジスタをシェアリングする最大ステート数を設定できるようにしている．

7. 評　　価

JavaRock-Thrash が生成した回路の性能を調べるため，FIR フィルタと FFT について IP コアとの比較を行った．JavaRock-Thrash が生成した回路と Xil-

表 1 FIR フィルタの特徴
Table 1 Features of the Fir Filter

データ型	16bit 固定小数点数
タップ数	323
乗算器数	2 個
Filter Architecture (IP コア)	Transpose Multiply Accumulate

inx ISE 13.4 の Core Generator によって作成した回路をそれぞれ論理合成し，Virtex-5（xc5vlx50t）上に配置配線した．比較する項目は，レジスタ数，LUT 数，スライス数，DSP ブロック数，動作周波数，処理サイクル数，処理時間の 7 つである．

FIR フィルタの特徴を表 1 に示す．FIR フィルタは，乗算器の数によってスループットが変化するため，JavaRock-Thrash と IP コアで 利用する乗算器数を 2 個に統一した．また，処理するデータは 2048 点の 16 ビット固定小数点数とし，全データを処理するのにかかったサイクル数をシミュレーション上で計測した．

FFT の特徴を表 2 に示す．FFT のアルゴリズムは，IP コア，JavaRock-Thrash ともに 基数 2 の Cooley-Tukey アルゴリズムを用いた．IP コアのアーキテクチャは 基数 2 の Cooley-Tukey アルゴリズムを低リソースで実行する Radix-2 Lite を Core Generator 上で選択した．また，処理するデータは，1024 点の単精度浮動小数点数とし，全データを処理するのにかかったサイクル数をシミュレーション上で計測した．

7.1 結　　果

FIR フィルタの比較結果を表 3 に示す．FIR フィルタは，JRT のスライス数が IP コアの約 4.0 倍となっているものの，処理時間は 1.69 倍の増加に抑えられた．同様に FFT の比較結果を表 4 に示す．FFT は，JRT のスライス数が IP コアの約 1.6 倍になっているのに対し，処理時間は 1.67 倍の低下に抑えられている．

8. 考　　察

以上の結果から，FIR フィルタのスライス数の増加率が 4.0 倍であることを除けば，JavaRock-Thrash の生成する回路は，IP コアと比べてもさほど劣っておらず，ハードウェア・アクセラレーションの研究やプロトタイピングなどに十分利用可能であるといえる．

8.1 回路規模の増加

JRT の LUT の利用数が IP コアの 4.7 倍となった理由について考える．JavaRock-Thrash は，処理をいくつものステートに分けて実行するため，データパスが複雑になりやすい．特に，ループ展開を行った場合，パイプライン構築のために多くのステートが定義され，結果として LUT の増加が発生する．一方，表 3 において，JRT のレジスタ数は IP コアの約 2.7 倍で，LUT の増加率より小さい．これは，レジスタシェアリングにより，レジスタ数が削減できたためであると考えられる．以上より，レジスタの最大シェアリング数を制限し，レジスタと LUT の利用率のバランスがとれた回路を生成することが重要であるといえる．

表 2　FFT の特徴
Table 2　Features of the FFT

データ型	単精度浮動小数点
ポイント数	1024
アルゴリズム	radix-2/Cooley-Tukey
Implementation Options(IP コア)	radix-2 Lite

8.2 処理時間の増加

処理時間が増加する原因は，処理サイクル数の増加と動作周波数の低下の 2 つである．動作周波数が低下した原因は，回路規模が増えたことによる遅延の増加である．処理サイクル数が増加した原因は，処理に直接関係の無いステート（for 文の比較部など）に遷移するためである．これらのステートに遷移すると，パイプライン処理が止まるため，スループットが低下する．スループットの低下を避けるためには，ループの展開数を大きくすればよいが，展開数を大きくすると回路規模が増え，動作周波数の低下につながる．つまり，処理サイクル数と動作周波数のバランスを考慮してループの展開数を決める必要がある．

8.3 FIR フィルタと FFT の回路規模の増加率

表 4 と表 3 の JRT の LUT の増加率に注目すると，FIR フィルタの LUT の増加率は，4.7 倍であるのに対し，FFT の LUT の増加率は 2.9 倍に収まっている．これは，手書きで記述した回路のデータパスが，FFT の方が FIR フィルタより複雑であるからだと考えられる．つまり，FIR フィルタはデータパスが単純であるため，処理を複数のステートに分割する JavaRock-Thrash との回路規模の差が大きくなったと考えられる．逆に，データパスが複雑な回路のほうが，回路規模の差が小さくなるため，JavaRock-Thrash で生成するのに向いていると言える．

8.4 手書きの HDL から作成した回路との比較

人手で記述した HDL から作成した回路と JavaRock-Thrash が生成する回路の比較を行う．文献 4) では，64 点の FFT を基数 2 の Cooley-Tukey アルゴリズムを用いて FPGA（XC3S5000）上に実装している．結果は，スライス数が約 1000 で，動作周波数が約 125MHz である．表 4 の結果と比べると，JavaRock-Thrash のスライス数は文献 4) の 1.4 倍であり，動作周波数は 1.6 倍となっている．文献 4) で用いた FPGA は，本研究で用いた FPGA より古いタイプではあるが，文献 4) で処理するデータ数が 64 点であることを考慮すれば，JavaRock-Thrash の生成した回路が手書きの HDL より回路規模および動作周波数の両面においても，さほど劣っていないと言える．

表 3 IP コアと JavaRock-Thrash が生成した回路の比較（FIR フィルタ）
Table 3 Comparison of the IP core and the circuit generated JavaRock-Thrash（FIR Filter）

	reg	lut	slice	DSP	動作周波数 [MHz]	処理サイクル数	処理時間 [ms]
IP コア	94	84	34	2	290	331782	1.14
JRT	257	394	139	2	224	432137	1.93

表 4 IP コアと JavaRock-Thrash が生成した回路の比較（FFT）
Table 4 Comparison of the IP core and the circuit generated JavaRock-Thrash（FFT）

	reg	lut	slice	DSP	動作周波数 [MHz]	処理サイクル数	処理時間 [μs]
IP コア	2471	1551	890	8	256	13419	52.4
JRT	3101	4514	1412	8	201	17648	87.8

9. ま と め

本稿では，JavaRock-Thrash の設計方針を述べた上で，Java ソースコードの変換規約や高速化，省資源化に関する機能について紹介した．また，FIR フィルタと FFT について IP コアとの比較を行い，データパスが複雑な回路であれば，回路規模，処理速度ともにさほど劣らないという結果を得た．

今後の課題としては，回路規模削減と処理速度向上のためのさらなる最適化や，複雑な機能を要する RTL の組み込みなどがあげられる．

参 考 文 献

1) High-level synthesis. http://www.ida.liu.se/petel/SysSyn/lect3.frm.pdf.
2) Maxeler technologies. maxcompiler.
3) Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: a java-compatible and synthesizable language for heterogeneous architectures. *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, Vol. 45, No. 10, pp. 89–108, October 2010.
4) Arman Chahardahcherik, Yousef S. Kavian, and Otto Strobel andRidha Rejeb. Implementing fft algorithms on fpga. *IJCSNS International Journal of Computer Science and Network S 148 ecurity*, Vol. 11, pp. 148–156, 2011.
5) Maya B. Gokhale, Janice M. Stone, Jeff Arnold, and Mirek Kalinowski. Stream-oriented fpga computing in the streams-c high level language. *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 49–56, 2000.
6) M. J. M. Heijligers and J. A. G. Jess. High-level synthesis scheduling and allocation using genetic algorithms based on constructive topological scheduling techniques. *Proceedings of the ASP-DAC95/CHDL95/VLSI95. Asia and South Pacific Design Automation Conference. IFIP International conference on Computer Hardware Description Languages and their Applications. IFIP International Conference on Very Large Scale Integration*, pp. 56–61, 1995.
7) Jamshaid Sarwar Malik, Paolo Palazzari, and Ahmed Hemani. Effort, resources, and abstraction vs performance in high-level synthesis: finding new answers to an old question. *SIGARCH Comput. Archit. News*, Vol. 40, No. 5, pp. 64–69, March 2012.
8) Pierre G. Paulin and John P. Knight. Force-directed scheduling for the behavioral synthesis of asics. *IEEE Trans. on CAD of Integrated Circuits and Systems*, pp. 661–679, 1989.
9) S. Tosun, O. Ozturk, N. Mansouri, E. Arvas, M. Kandemir, Y. Xie, and W-L. Hung. An ilp formulation for reliability-oriented high-level synthesis. *Proceedings of the 6th International Symposium on Quality of Electronic Design*, pp. 364–369, 2005.
10) Justin L. Tripp, Preston A. Jackson, and Brad Hutchings. Sea cucumber: A synthesizing compiler for fpgas. *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pp. 875–885, 2002.
11) Chu yi Huang, Yen shen Chen, Youn long Lin, and Yuchin Hsu. Data path allocation based on bipartite weighted matching. *Matching, Proceedings of the IEEE Design Automation Conference*, pp. 499–504, 1990.
12) 健文三好, 悟史船田. 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, pp. 59–68, 2012.
13) 健文三好, 悟史船田. Javarock を用いた hw/sw 協調設計の検討. 電子情報通信学会技術研究報告. AI, 人工知能と知識処理, Vol. 112, No. 70, pp. 119–124, 2012.