FPGA向け高位合成言語としてのJavaの活用手法の検討

三 好 健 文[†] 船 田 悟 史[††]

複雑なアルゴリズムを簡単にFPGA上に実装できるように，Javaプログラムからの高位合成を検討する．ハードウェアを表現するための特殊な文法や，アノテーション，ライブラリを追加しないJavaで記述されたプログラムをそのまま高位合成言語として用いることで，設計コストの削減ができる．また同じプログラムがソフトウェアとしても実行できることからアルゴリズムレベルでの検証コストの削減もできる．

本稿では，まず，Javaを高位合成言語として利用する動機について述べる．次に，開発したJavaからHDLに変換するコンパイラJavaRockの設計方針とソフトウェア階層について述べ，コード変換規約について説明する．また，FPGAを活用する上で重要な並列性をJavaでどう記述するかについて示す．最後に，生成したHDLから合成した回路のリソース使用量と処理性能の点から評価した結果を示し，Javaを高位合成言語として利用することの有効性について示す．

A High-level Synthesis for FPGA from Java

Takefumi Miyoshi[†] and Satoshi Funada [††]

It is considered that Java is employed as a high-level synthesis(HLS) Language in order to implement complex algorithms onto FPGA. Cost for hardware design is reduced due to using pure java program without additional special syntax, annotations, and libraries as a HLS. In addition, it also reduces verification cost of the algorithm, since same program is executable as a software on JVM.

In this paper, the motivation to employ Java as a high-level synthesis language for FPGA is described, firstly. Then, the design of JavaRock which is a compiler from Java into HDL and the rules to compile are presented. In addition it is described how to implement parallelized modules running on FPGA by Java. Finally, evaluation results of generated hardware by JavaRock in terms of the usage of hardware resources and the performance are shown. The results show that Java is a dominant candidate of a high-level synthesis language for FPGA.

1. はじめに

Field Programmable Gate Array(FPGA)は，プログラム可能なハードウェアデバイスであり，ユーザが自由にハードウェアロジックをその上に構築できる．実行したい処理中の並列性を活用することで，プロセッサによるソフトウェア処理に比べ，低消費電力で高い処理能力を得ることができる．

FPGAの性能を効率良く活用するためには，一般に，VHDLやVerilogを用いたRegister Transfer Level(RTL)の設計が行われている．しかしながら，アルゴリズムとして複雑な処理のRTL記述は繁雑で手間がかかり，時にはバグの温床となる．そのため，RTLより高い抽象度でのハードウェア設計を可能にする高位合成言語が求められている．高位合成言語には，設計の繁雑さを解消しながら，FPGAのパフォーマンスを引き出すことが要求される．

高水準言語にハードウェア設計のために必要な拡張を加えることで，RTLより高い抽象度のハードウェア設計を可能にする高位合成言語として，PHDL[1)]，RHDL，JHDL[2)]，DSL-Based[3)]などの研究があり，SystemC[4)]，ImpulseC[5)]，Handel-C[6)]，BachC[7)]，MaxCompiler[8)]など実用化されているものも数多く存在する．これらの高位設計言語では，元となる高水準言語に特別な型やクラスライブラリ，文法などを追加することでハードウェアの要素やハードウェア上での処理の振舞いを表現できるように拡張が加えられている．また，FPGAで効率良く処理を実行させるために処理内のデータ並列性とパイプライン並列性を抽出するための，アノテーションやコンパイラへの指示子が導入されている．これらの高位合成言

† 電気通信大学大学院情報システム学研究科
Graduate School of Information Systems, The University of Electro-Communications
†† 株式会社イーツリーズ・ジャパン
e-trees.Japan, Inc.

語を用いることで，高水準言語のもつ有用な機能を利用した簡単なHDL設計で，FPGAを効率良く活用できる．

しかしながら，これらの研究あるいはツールでは，高水準言語の機能を用いて記述された部分とハードウェア化の対象となる部分のコードを混在させることができない．また，ハードウェアとして最適化するように指示した箇所のコードはソフトウェアとして実行することができない．そのため，処理の検証には，それぞれの言語に向けて開発されたシミュレータかRTLでの検証が必要となる．すなわち，単にソフトウェアをプログラムとして実行した場合に比べて検証時間が長大し，また検証に使える手段が限られコストが大きい．従って，これらの高位合成言語では設計の繁雑さを完全に解決できていない．

プログラム言語を拡張することなく，既存の言語で記述されたプログラムからハードウェアへの合成を可能にする手法として，CyberWorkBench[9] やLegUp[10] がある．これらは，プログラミング言語Cで記述されたプログラムからハードウェアの合成を可能にする．入力されるプログラムは，元来Cで記述されているため，一般的なCコンパイラでコンパイルすればコンピュータの上でソフトウェアとして動作させることができる．これらは，新たなプログラミング習得の手間なしでプログラムをハードウェアとして実装でき，また，ソフトウェアとしてアルゴリズムレベルでのデバッグが可能になるため，開発の繁雑さを解消している．

しかしながら，Cで記述されたプログラムを対象とするため，ポインタをどのように合成処理系で取り扱うかという課題が生じる．また，FPGAでは，性能向上の実現のために処理を並列に実行することが一般的なアプローチであるが，Cの言語仕様には並列処理をどのように扱うかという規定がない．そのため，スレッドレベルやタスクレベルの並列性をプログラマ自身が簡単に記述できるように，それぞれの処理系で独自にサポートしなければならない．そのため，各処理系に合わせて，プログラマが，ソフトウェアの記述方法と，そのハードウェアでの実現方式を意識してプログラミングしなければならず，プログラミングのコストが大きいという課題は以前として残る．

特定の高水準言語をベースとするのではなく，新しく設計された高位合成言語もある．Bluespec System Verilog[11](BSV)は，ハードウェア設計に関数型言語の特徴を取り入れ，高階関数による強力な抽象的な設計手法と，強力な型システムによる頑健な設計を可能にする新しいハードウェア記述言語である．BSVでは，これらの高水準言語としての機能をすべてハードウェア化することができる．しかし，関数型言語とHDLという二種類のパラダイムを融合したこの言語は，修得コストが大きい．また高速なシミュレータを持つものの，ソフトウェアとして処理を実行する場合に比べると，デバッグに使える手法の幅が少なく，アルゴリズムレベルのデバッグは簡単ではない．

以上を踏まえ，広く普及しているJavaプログラムからハードウェアを合成することを考える．Javaプログラムをソフトウェアとしてコンピュータ上で実行してアルゴリズムレベルのデバッグを行い，その十分にデバッグしたプログラムをハードウェア化することで，RTL設計以降でのアルゴリズムレベルのデバッグが不要になる．従って，新たなプログラミング習得の手間なく，ソフトウェアとしてアルゴリズムレベルでのデバッグが可能になることで，開発の繁雑さが解消される．また，Javaの場合，Cのように明示的にポインタを扱うことはなく，また，並列処理を記述可能なThreadを言語仕様として含むため，プログラミング言語CからハードウェアをCyberWorkBenchやLegUpに残った課題も解決できる．

しかしながら，ハードウェア設計に必要となる，クロックの取り扱い手法や，データの保持されるタイミング，細粒度での処理同期や，パイプライン/データ並列性を活用するプログラムはJavaでは記述できない．そのため，FPGAの性能を十分に活用することは難しい．しかし，Javaで記述された処理を実行するためのプロセッサ機構を構成するのではなく，直接的にハードウェア化することで，フェッチやデコードなどのソフトウェア処理のための機構が省略され，演算密度が高くなる．このことによって，FPGAを用いることでの処理の高速化や消費電力の削減などの効果を得られることが期待できる．また，HDLで独自に記述したFPGAのために最適化されたモジュールをJavaから簡単に呼び出せる仕組みが提供できれば，ボトルネックとなる部分的な処理をユーザがHDLで記述することで性能向上を図ることができるようになる．これらによって，Javaによるハードウェア設計で，FPGAの性能を引き出すという課題にも，ある程度の解決が期待できる．

実際に，FPGAが使用されたアプリケーションの実装に目を向けてみると，処理の本体でないI/O処理や複雑な逐次処理部分に対しては，補助的なプロセッサの利用や愚直に実装されたステートマシンで処理されているケースが多数みられる．このようなケースでは，

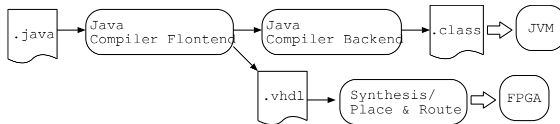

図 1 コンパイルフロー

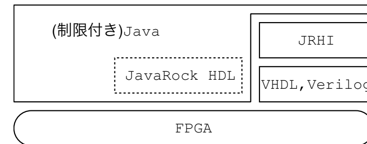

図 2 JavaRock におけるソフトウェア階層

クロックを意識して高度に最適化したハードウェアと，高位合成言語から生成されたハードウェアとの間に有意な性能差がみられないと考えられる．逆に，補助的なプロセッサを用いる場合には，それらプロセッサ用のコードやツールセットの整備と保持しなければならず，開発コストが増加する．すなわち，多少の性能低下によるデメリットは，開発コストを削減できるというメリットで上回ると考えられる．

本論文では，第 2 節で Java からハードウェアを設計するためのコンパイラ JavaRock の設計について述べる．JavaRock は，ハードウェアを生成するための中間言語として，Java から VHDL を生成する．第 3 節では，この Java から VHDL コードを生成するためのコード変換規約について述べ，第 4 節では，FPGA を有効に活用する上で重要なハードウェア上の並列性を Java でどう記述するかについて述べる．第 5 節では，JavaRock の評価として生成したハードウェアを合成したときのリソース使用量および最高動作周波数を示し，またケーススタディによって，Java によるハードウェア開発の利点について考察する．

2. JavaRock の設計

Java で記述されたプログラムを FPGA 上にそのままハードウェア化するコンパイラ JavaRock の設計について述べる．JavaRock は，Java で記述されたプログラムを VHDL に変換し，一般的な合成・配置配線ツールを用いて変換した VHDL から FPGA 上のリソースにマッピングすることで，プログラムをハードウェア化する．**図 1** にコンパイルフローを示す．

第 1 節で述べたように，開発の繁雑さの軽減および検証の効率の向上のために，Java への言語拡張は行なわずにハードウェア設計に適用することが JavaRock の基本的な設計方針である．その一方で，Java の言語仕様の範囲内では，ハードウェアの設計に必要なクロックや信号のビット幅といったプリミティブな部分の記述をすることができない．これを解決するために，Java と RTL のインターフェイスを提供する．

図 2 に JavaRock におけるソフトウェア階層を示す．Java で記述したプログラムを VHDL に変換し FPGA 上のハードウェアにする他，VHDL あるいは Verilog と Java とのインターフェイスである JavaRock Hardware Interface(JRHI) を介した FPGA 上のモジュールへのアクセスが可能である．加えて，Java による HDL 設計を可能にする JavaRock HDL によるハードウェア設計もサポートする．それぞれの詳細について次の各小節で述べる．

2.1 対象とする Java の言語機能

JavaRock では，Java で記述されたプログラムをハードウェアに変換する．しかし，Java では多くの動的な振舞いをサポートする仕組みがあり，それらをすべて FPGA 上にハードウェア化するのは困難であり，また現実的ではない．なぜならば，それらをハードウェア化するためにはヒープやスタックを構成し，その上で JVM 相当のハードウェア機構を実現する必要があり，FPGA を用いる意味がない程度に性能が低下すると考えられるからである．

JavaRock では，ハードウェア化できない Java の動的な振舞いは使用できないよう制限を加える．まずは，Java の言語機能のうち，プリミティブの変数とその演算，プリミティブ変数の配列，制御構造，`final` で宣言したインスタンスの生成と，インスタンスのメソッド呼び出しなどをサポートする．一方，まだサポートしない機能としては，動的なインスタンス生成，インスタンスの共有，例外機構などがある．

2.2 JavaRock Hardware Interface

Java では，クロックや信号のビット幅といったハードウェア設計のプリミティブな部分の記述をすることができない．しかしながら，実際に FPGA 上で動作するハードウェアを開発する場合には，FPGA と FPGA 外部のデータのやりとりや，細かいタイミング制御が必要不可欠である．また，高性能化あるいは既存資産の活用を目的として RTL で設計した HDL のコードを活用したいという要求もある．

そこで，JavaRock では，VHDL あるいは Verilog によって RTL で記述されたモジュールを Java と結合するためのインターフェース JavaRock Hardware Interface(JRHI) を提供する．これは，Java で記述できないプログラムを C/C++で記述し JNI で結合することと同様である．

HDL で記述したモジュールを使用する場合には，そ

表 1 定義したアノテーション (抜粋)

名前	役割
javarockhdl	クラス内の JavaRock HDL アノテーションを有効にする
raw	if や switch などの条件文を HDL に直接変換する
auto	電源投入直後にメソッドを自動的に実行する
width	変数のビット幅を指定する

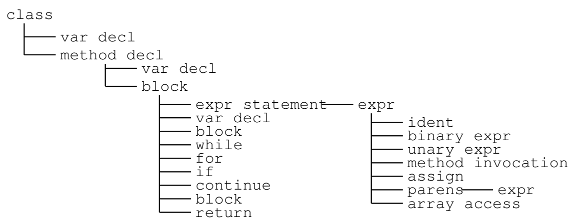

図 3 Java の構文木の一部

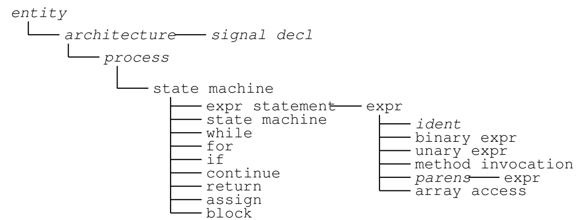

図 4 JavaRock で扱う VHDL の構文木 (抜粋)

```
1: public class sum{
2:  public int sum(int[] a){
3:    int sum = 0;
4:    for(int i = 0; i < a.length; i++){
5:      sum += a[i];
6:    }
7:    return sum;
8:  }
9: }
```

図 5 与えられた配列の合計値を求める Java のプログラム

のモジュールにアクセスする Java プログラムをコンパイルできるように，JRHI として，その HDL ライブラリの入出力ポートを変数として持つラッパークラスを Java で記述する．HDL で処理されたモジュールを利用するクラスは，このラッパークラスに宣言された変数を介してモジュールを制御する．これにより，Java で記述されたプログラムを只の Java プログラムとしてコンパイルすることと，変換後の HDL で実際にそのモジュールを利用することとの両方を実現できる．

2.3 JavaRock HDL

JavaRock では，基本的に只の Java を FPGA 上のハードウェアにマッピングするか，JRHI を介して RTL で記述したモジュールにアクセスする．しかしながら，JHDL などの高水準言語をベースとした HDL の研究同様，みかけ上 Java で記述することのメリットを活かしつつ，ハードウェア設計のプリミティブの記述が可能であると便利なことも多い．

そのため，いくつかのアノテーションを定義した JavaRock HDL を提供する．定義したアノテーションの一部を**表 1** に示す． ここで JavaRock HDL は，あくまで利便性のために定義したものであり，必ずしも必要ではないことに注意されたい．

3. Java から VHDL へのコード変換規約

JavaRock では，構文木レベルで Java を VHDL に変換する．Java の構文木の一部を図 3 に示す．JavaRock では，**図 3** に示した Java の構文木を**図 4** に示す VHDL の構文木へと変換する． 中の斜体で表示している要素は，基本的には直接 VHDL の構文木の要素を用いて表現する．しかしながら，与えられた Java のクラスを VHDL エンティティに，クラス内で定義されたメソッドはそのエンティティのプロセスに，と類似する要素に変換できるが，Java の制御構文の多くは直接 VHDL に変換することはできない．そのため Java のプログラムと同様のデータ処理を実現するためのハードウェアアーキテクチャを作る必要がある．

本節では，まず Java プログラムに相当する基本的なハードウェアアーキテクチャの概要について述べ，次に，Java の個別の構文要素を VHDL に変換する手法を，**図 5** に示すプログラムを例に説明する．このプログラムコードは，与えられた配列の合計値を求めるメソッドを持つクラスである．これは一般的な Java プログラムであり，Java コンパイラでコンパイルすれば，JVM で実行できる．

3.1 ハードウェアアーキテクチャ概要

図 6 に図 5 のプログラムから変換される VHDL エンティティの概観図を示す．ここで，sum という名前の VHDL のエンティティ(外側の箱) と sum という名前のプロセス (内側の箱) が，それぞれ，sum クラスと sum メソッドに対応している．複数のメソッドが定義されている場合には，それぞれ複数のプロセスに変換する．メソッドは基本的に同期順序回路のプロセスに変換し，クロックとリセット信号を外部から供給する．メソッドの仮引数はエンティティへの入力ポートとして，メソッドの返り値は出力ポートとして定義する．この例では，メソッドの仮引数である int 型の配列 `a` に `input_port_sum_a_{we,wdata,wadd,length}`の 4 ポートが対応し，メソッドの返り値に `output_port_sum` が対応している．入力ポートの `sum_method_busy` と出力ポートの `sum_method_req` は，メソッド呼び出し

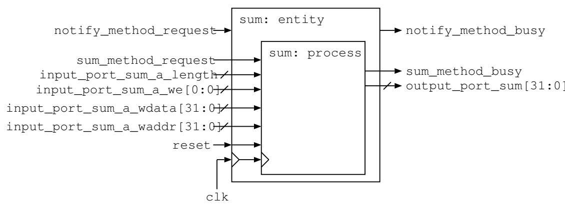

図 6 図 5 から生成されるハードウェアアーキテクチャの外観

```
1: case conv_integer(sum_method_state) is
2:   ...
3:  when 2 =>
4:    sum_0 <= conv_std_logic_vector(0, 31-0+1);
5:    sum_method_state <= sum_method_state + 1;
6:  when 3 =>
7:   ...
```

図 7 文のステートマシンへの変換例

を制御するための信号線である．メソッド呼び出しに関するコード変換規約は，第 3.5 節で詳しく説明する．

3.2 逐次処理のステートマシン化

Java で記述されたプログラムは，一般には JVM で逐次的に実行される．一方，VHDL では逐次的に実行させる処理を直接書き下すことができない．そこで，JavaRock では Java の文をすべてステートマシンに変換する．

例えば，図 5 のプログラム 3 行目の，`sum` に 0 を代入する文は，**図 7** に示すようなステートマシンの 1 ステートに変換する．ここで，メソッドの逐次的な処理が `sum_method_state` と命名されたカウンタによるステートマシンに変換されている．このステートでは，Java の文に相当する処理を実行 (4 行目) 後に，カウンタをインクリメントして (5 行目)，次のステートすなわち次の文の処理の実行に遷移する．

ここで，Java の `sum` が VHDL では `sum_0` という信号名に変換されていることに注意されたい．この`_0` という接尾辞は変数のスコープに応じて JavaRock により付与されている．

3.3 制御構造

JavaRock では，Java の `if` 文，`while` 文，`for` 文をサポートしている．これらの制御文は，メソッド全体のステートマシンに内包されるサブステートマシンに変換する．これは，コンパイル時の繁雑なステートマシンのカウンタ処理を省略でき，また，ステートマシンのカウンタの加算器を小さくできることで，信号遅延を小さくする効果が期待できる．

例として，図 5 のプログラム 4-6 行目の，`for` ループに相当する VHDL コードを**図 8** に示す．`state_counter_sum_1` をカウンタとする 3-20 行目のステートマシンが `for` ループに相当する．ここで，`i_1` は for 文のカウンタ変数，`input_port_sum_a_length` は配列の length フィールドに相当する変数である．

```
1: case conv_integer(sum_method_state) is
2:  ...
3:   when 3 =>
4:     case conv_integer(state_counter_sum_1) is
5:       when 0 =>
6:         i_1 <= conv_std_logic_vector(0, 31-0+1);
7:         state_counter_sum_1 <= state_counter_sum_1 + 1;
8:       when 1 =>
9:         if (conv_integer(i_1) < input_port_sum_a_length) then
10:          state_counter_sum_1 <= state_counter_sum_1 + 1;
11:        else
12:          sum_method_state <= sum_method_state + 1;
13:          state_counter_sum_1 <= (others => '0');
14:        end if;
15:      when 2 =>
16:           ...
17:      when 3 =>
18:          i_1 <= conv_std_logic_vector(conv_integer(i_1 + 1), 31-0+1);
19:          state_counter_sum_1 <= conv_std_logic_vector(1, 32);
20:      when others => state_counter_sum_1 <= (others => '0');
21:    end case;
22:  when 4 =>
23:  ...
```

図 8 for ループの変換例

このステートマシンは，カウンタ変数の初期化 (4-6 行目)，継続条件の判定 (7-13 行目)，処理本体 (15 行目，コードは省略)，および，継続条件の判定に戻る (16-18 行目)，の 4 ステートを持つ．継続条件の判定が偽になった場合，`state_counter_sum_1` をクリアし，一つ外側のステートマシンである `sum_method_state` をインクリメントすることで，メソッド全体のステートマシンを次のステートに遷移させる．

同様に `if` や `while` でもサブステートマシンを構成する．ただし，`while` の場合には初期化のための状態はない．また，JavaRock では `for` や `while` 内での `break` と `continue` をサポートしている．`break` は `for`/`while` のステートマシンのカウンタのクリアと外側のステートマシンのカウンタのインクリメントで実現する．また，`continue` は，`for`/`while` のステートマシンのカウンタを継続条件判定ステートに設定することで実現する．

3.4 配 列

Java プログラム中に記述された `boolean`, `int`, `char` などのプリミティブの変数は直接 VHDL のシグナル変数に変換する．しかしながら，Java で記述された配列を VHDL の配列に直接変換すると，FPGA のレジスタを大量に使用してしまう可能性がある．そこで，配列には，FPGA に内蔵された BlockRAM を使う．

図 5 のプログラム 5 行目の，配列の値を sum に足し合わせる部分の変換後の VHDL コードを**図 9** に示す．なお，この 2-18 行目のコードは図 8 中の 16 行目で省略されたコードに相当する． ここで，配列 `a` は，**図 10** に示すコードでインスタンシエーションされた BlockRAM に変換する．すなわち，配列 `a` へのアク

```
 1: when 2 =>
 2:   case conv_integer(state_counter_sum_2) is
 3:     when 0 =>
 4:       case conv_integer(array_index_operation_state_counter_3) is
 5:         when 0 =>
 6:           param_input_port_sum_a_raddr <=
                       conv_std_logic_vector(conv_integer(i_1), 11-1-0+1);
 7:           array_index_operation_state_counter_3 <=
                                 array_index_operation_state_counter_3 + 1;
 8:         when 1 =>
 9:           array_index_operation_state_counter_3 <= (others => '0');
10:            state_counter_sum_2 <= state_counter_sum_2 + 1;
11:          when others => array_index_operation_state_counter_3 <=
                                                         (others => '0');
12:        end case;
13:      when 1 =>
14:        sum_0 <= conv_std_logic_vector(
              conv_integer(sum_0 + param_input_port_sum_a_rdata), 31-0+1);
15:        state_counter_sum_1 <= state_counter_sum_1 + 1;
16:        state_counter_sum_2 <= (others => '0');
17:      when others => state_counter_sum_2 <= (others => '0');
18:   end case;
19: when 3 =>
20: ...
```

図 9 配列にアクセスするコードの変換例

```
 1: U_param_input_port_sum_a : simpledualportram
 2: generic map(
 3:   DEPTH => param_input_port_sum_a_DEPTH,
 4:   WIDTH => param_input_port_sum_a_WIDTH
 5: )
 6: port map(
 7:   clk => clk,
 8:   we => param_input_port_sum_a_we,
 9:   raddr => param_input_port_sum_a_raddr,
10:   rdata => param_input_port_sum_a_rdata,
11:   waddr => param_input_port_sum_a_waddr,
12:   wdata => param_input_port_sum_a_wdata
13: );
```

図 10 BlockRAM のインスタンシエーション

セスは param_input_port_sum_ という接頭辞を付けた信号群の操作に変換される．

BlockRAM から値を読み出すためには，読み出す前にアドレスを指定しなければならない．そのため，state_counter_sum_2 をカウンタとするサブステートマシンを構成する．アドレスを指定するステートがさらに 2 ステートに分割されているのは，Xilinx の BlockRAM のデータ出力がバッファされて 1 クロック遅延するのに対応するためである．

3.5 メソッド呼び出し/終了の制御

メソッドの呼び出しと終了に関するコード変換規約について説明する．VHDL では，サブプログラムを記述可能な procedure あるいは function がある．しかしながら，インスタンス間でのメソッド呼び出しの実現，メソッド間でのインスタンス内でのリソース共有のための排他制御を考慮にいれたメソッド呼び出しの実現のためには，procedure や function をそのまま使用することはできない．そのため JavaRock では**図 11** に示す独自のメソッド呼び出し規約を導入する．

この呼び出し手順では，まず，callee 側は各メソッドに対応して生成される "メソッド名_method_req" 信号をアサートし，caller 側にメソッド呼び出しをリクエストする．caller 側では，"メソッド名_method_req" が

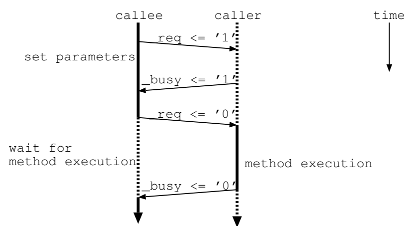

図 11 メソッド呼び出しと終了の制御方式

```
 1: case conv_integer(sum_method_state) is
 2:   when 0 =>
 3:     if(sum_method_request = '1') then
 4:       sum_method_busy <= '1';
 5:       sum_method_state <= sum_method_state + 1;
 6:     else
 7:       sum_method_busy <= '0';
 8:     end if;
 9:   when 1 =>
10:      if(sum_method_request = '0') then
11:        sum_method_state <= sum_method_state + 1;
12:      end if;
13:   when 2 =>
14:     ...
15:   when 4 =>
16:    output_port_sum <= conv_std_logic_vector(conv_integer(sum_0), 31-0+1);
17:    sum_method_state <= (others => '0');
15:   when 5 =>
16:     sum_method_busy <= '0';
17:     sum_method_state <= (others => '0');
18:   when others => sum_method_state <= (others => '0');
19: end case;
```

図 12 caller 側でのメソッドの実行制御コード

アサートされると，メソッドの実行状態に入り，"メソッド名_busy" 信号をアサートする．caller 側では "メソッド名_method_busy" 信号がアサートされたらメソッドへの実引数を caller の仮引数に対応した入力ポートにセットし，最後に "メソッド名_method_req" をディアサートする．caller は，"メソッド名_method_req" がディアサートされたら処理を開始する．

図 5 に示した sum メソッドから生成したメソッドの実行制御に関する HDL コードを**図 12** に示す．ステート 0 で callee からのリクエストを待ち，ステート 1 で実行開始を待機する．ステート 2 でメソッドの処理を行い，ステート 4 で返り値を出力ポートに代入する．最後にステート 5 でメソッドの実行を終了し，ステート 0 に遷移することで再び呼び出されるのを待つ．ステート 1 で待機している間に callee 側は実引数をセット，すなわち入力ポートに値を代入する．

メソッド間の引数の受け渡しは，一般にソフトウェアではスタックが用いられる．しかし，FPGA 上でスタックを作るためには，レジスタを多数利用するか BlockRAM を用いるしかないが，いずれにしても潤沢に利用できるわけではない．そのため，プリミティブな値の受け渡しには，単に signal で値を受け渡すこととする．配列の受け渡しに signal を使用するのはリソース使用量の観点で現実的ではないため，caller 側に

引数を受けとるための BlockRAM を用意する．第 3.4 節に示した BlockRAM`U_param_input_port_sum_a` は引数 `a` として与えられた配列の要素をコピーするための BlockRAM である．

なお，再帰呼び出しを実現する方法については将来的な課題として残る．また，配列は本来参照渡しすべきであるが，現在の実装では値渡しになっている．

4. Java による並列性の活用手法

FPGA では，複数の処理を同時並列で処理可能なハードウェアモジュールを自由に設計することができる．高性能な汎用プロセッサと比べ高い処理能力を実現するためには，この並列性を活用することが必要であり，高位合成言語では，並列性を活用したハードウェアの設計ができることが求められる．

JavaRock は，Java でのプログラム記述能力の範囲を逸脱することなく，ハードウェア設計に Java を活用することを目指している．そのため，文法の拡張やアノテーションの追加による並列性の記述は望まれない．

ここで，ビットレベルおよび命令レベルの細粒度並列性を Java で記述することは困難であるため明示的にプログラマに記述させることはしない．ビットレベルの並列性は HDL の合成・配置配線レベルで適用され，また，基本ブロック内での命令レベルの細粒度並列性はコンパイラによる最適化でサポートすることとする．このことにより，プログラマは意図することなく，それらのレベルでの並列化の効果を享受できる．一方で，タスクレベルでのデータ並列，パイプライン並列性の抽出は，現状のコンパイラによる自動化が難しい．しかしながら，粗・中粒度のデータ並列処理，パイプライン並列処理は，Java の Thread や wait-notify を利用することで表現可能である．従って，JavaRock ではハードウェア上で並列に動作する粗・中粒度のモジュールを Java のこれらの機能を使って実現することとする．

4.1 Thread による並列処理

FPGA 上では，処理を行うことを同時並行的に動作させることが必要不可欠である．この同時並行的に動作するモジュールを Java で表現するために，`Thread` を用いる．

Java では，`Thread` クラスを継承したクラスのインスタンスの `start` を呼び出すことで，新しいスレッドを生成することができる．新しいスレッドでは，そのインスタンスの `run` メソッドが呼び出され，その処理は，呼び出し元の処理とは並列に実行される．

JavaRock では，基本的には，Java のメソッド呼び出しは，第 3.5 節で述べたように，呼び出したメソッドが終了するまで呼び出し側の処理が停止するように変換する．しかし，`Thread` を継承したクラスの `start` は，同インスタンスの `run_method_request` をアサートした後，メソッドの実行完了を待たずに直ちに処理を戻し，呼び出し側に返る．こうすることで，Java のスレッド機構同様のパラダイムを HDL でも利用できる．

```
1: case conv_integer(wait_method_state) is
...
2:   when 2 =>
3:     if (d_notify_flag = '0' and notify_flag = '1') then
4:       wait_method_state <= wait_method_state + 1;
5:     else
6:       d_notify_flag <= notify_flag;
7:     end if;
8:   when 3 =>
...
```

図 13 wait でのループ

```
1: when 2 =>
2:   notify_flag <= '1';
3:   notify_method_state <= notify_method_state + 1;
4: when 3 =>
5:   notify_flag <= '0';
6:   notify_method_state <= notify_method_state + 1;
...
```

図 14 notify でのフラグの操作

4.2 wait-notify によるパイプライン処理

データ並列性と同様にパイプライン並列性を活用することは FPGA を効率良く使う上で重要である．タスクレベルのパイプライン並列性は Producer-Consumer パタンとして抽象化され，Java では，Object クラスの持つメソッドである wait と notify を使って実装できる[12]．従って，wait と notify をハードウェア上の回路にマッピングすることを考える．

wait-notify は，オブジェクトのロックを考えなければ，ロック変数の操作とその変化に対するビジーウェイトで表現できる．`notify_flag` をロック変数とし，`d_notify_flag` を `notify_flag` を 1 クロック遅延させた信号だとするとき，**図 13** に示す HDL コード片で `notify_flag` が立ち上がるタイミングまで待つループとして wait メソッドを実装できる． 一方で notify メソッドは，**図 14** に示すフラグ操作で実装できる．

現状の JavaRock では，インスタンスを共有できず，また，第 3.5 節で述べたように，関数を呼び出す callee は，呼び出された caller が処理をしている間他の処理ができない．ゆえに，図 13 と図 14 に示した単純なビジーウェイトで wait-notify を実装できている．

5. 評　　価

第 1 節で述べたように，JavaRock 開発の主な動機は，生成できるハードウェアの質ではなく，開発効率

表 2 XC6VLX240T-1 のリソース量の諸元

項目	個数
#. of Slice Registers	301,440
#. of Slice LUTs	150,720
#. of Slices	37,680
#. of BRAM (32KB)	416
#. of DSP48	768

表 3 リソース使用量の評価

	レジスタ数	LUT 数	Block RAM
sieve	106	246	16
sort	268	408	1
connect6	2990	4469	2

の向上である．しかしながら，汎用のプロセッサではなく FPGA で処理を実行する目的の一つは，汎用のプロセッサでは得られない高い処理能力を得るためである．そのため，JavaRock の評価として，他の高位合成言語が生成するハードウェアの質と比較することには重点を置かないものの，生成した回路が実用的なリソース使用量に抑えられること及び，ソフトウェアの処理能力に勝るとも劣らないことが重要である．

本稿では，エラストテネスのふるいによる素数判定 (sieve) とバブルソート (sort) および，FPT デザインコンペティションの connect6[13] のソースコード C++ を Java に変換したものを用いて，JavaRock が生成するハードウェアの性能を，リソース使用量と処理性能の観点で評価し，Java を高位合成言語として用いることの有用性を示す．実装の対象とする FPGA として Xilinx 社の表 2 に示す Virtex-6 XC6VLX240T[14] を用い， 合成と配置配線には，Linux 版の XilinxISE 13.1 と付属に XST コンパイラを使用する．また，connect6 の設計と VGA へのグラフィクスを描画するハードウェアの設計をケーススタディとしてソフトウェアスタックを利用したデバッグ手法について述べる．

5.1 リソース使用量

生成されたハードウェアを合成した結果得られたリソース使用量を**表 3** に示す．ここで，sort は 512 個の int 型の要素からなる配列を，sieve は 65536 までの数の中の素数を判定するハードウェアである． それぞれの最高動作周波数は，269.179MHz，230.153MHz，および 152.707MHz であった．

JavaRock ではリソース量を削減するための最適化，たとえばデータやステートマシンの状態を表現するための各 signal を本当に必要なビットだけに削減するなどを一切行なっていない．しかしながら，表 3 の結果によると，ソフトウェアで宣言したリソース量よりも少ないリソース量だけがハードウェアにマッピングされており，既存の合成・配置配線ツールによって，RTL での最適化が適用されていることがわかる．XC6VLX240T のレジスタ数と LUT 数は，それぞれ 301,440 と 150,720 であるから，大規模な FPGA に対しては，リーズナブルなリソース使用量で所望の回路が生成できていると言える．

表 4 比較対象の計算機

CPU	Core i7 2.93GHz
メモリ	16GB 1333MHz DDR3
OS	MacOSX 10.6.8
JVM	1.6.0_26 64-Bit Server VM

5.2 処 理 性 能

次に計算機上でソフトウェアとして sieve と sort を実行した場合と生成したハードウェアでの処理速度を比較した結果を示す．ここで，sort の対象とするデータは降順すなわちまったくの逆順で並べたものを入力して与える．JavaRock でハードウェア化可能なプログラムは純粋に Java のプログラムであるから，そのまま計算機上の JVM で実行することができる．比較対象の計算機の諸元は**表 4** に示す通りである．FPGA での処理性能は，Xilinx ISE13.1 付属の ISim で評価した．

評価の結果 200MHz で FPGA が動作するとした場合，FPGA で sieve と sort の処理にかかる時間はそれぞれ，7.3m 秒と 9.8m 秒と求められた．一方で，計算機上で実行した場合は，それぞれ 10.2m 秒と 7.6m 秒であった．つまり，これらのプログラムの実行においては，Java から生成された HDL による FPGA が 200MHz で動作する場合には Core i7 2.93GHz に匹敵する処理性能を実現できた，と言える．

5.3 ケーススタディ

次に，connect6 の設計と VGA へのグラフィクス描画ハードウェアの設計をケーススタディとして，JavaRock を用いたハードウェア設計におけるソフトウェアスタックを利用したデバッグ手法について述べる．ただし，実装の都合上，VGA へのグラフィクス描画ハードウェアの設計は，Altera Cyclone® IV EP4CE115 を搭載した評価キットである DE2-115[15] を対象とした．

5.3.1 connect6 のハードウェア化

開発にあたって，まず，Web サイト[13] で公開されている C++で記述された connect6 のソースコードを機械的に Java に置換した．対象としたのは，connect6 のカーネルである，`connetc6.cpp` 内の各格子点の評価関数を求める `calulatepoints` とすべての格子点の評価値を計算し手を選択する `connect6ai` の二つの関数である．この二つの関数をそれぞれ，`Calculator.java`

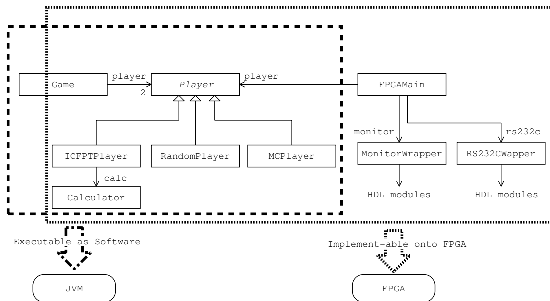

図 15 connect6 のデバッグで用いたクラスのクラス図

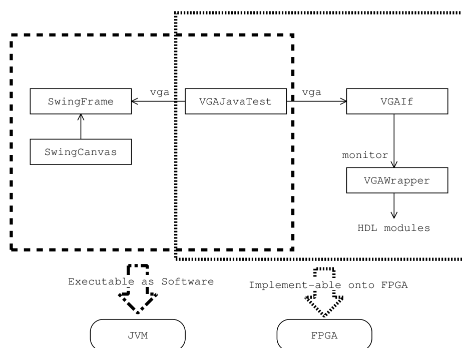

図 16 グラフィクス描画ハードウェア設計のためのクラス構造

と `ICFTPPlayer.java` として Java で実装する．

図 15 に，Java で実装した connect6 をデバッグする際に利用したクラス群のクラス図を示す．`ICFPTPlayer` と `Calculator` からなる ICFPT で提示された connect6 プレイヤは，他の幾つかのプレーヤ実装と共に `Player` インターフェイスを実装するクラスである．このクラスのインスタンスは，`Game` と `FPGAGame` で共有される．`Game` は JVM 上で動作させるときのメインルーチンであり，`FPGAGame` は FPGA 上に実装するときのトップエンティティに相当するクラスである．`Game` を含むクラス群は JVM で直接実行することができ，`FPGAGame` を含むクラス群は FPGA に実装可能である． このように，ソフトウェアとしてのデバッグ環境を構築することで，簡単かつ効率的にデバッグできる．

5.3.2 VGA グラフィクス描画ハードウェア

VGA グラフィクス描画ハードウェアを設計することを考える．描画ルーチン，たとえば四角形を色を変えながら描画するルーチンをハードウェア化するとする．**図 16** にデバッグのために設計したクラス階層を示す．

ハードウェア化すべき，描画ルーチンを記述した VGAJavaTest から，実際の描画対象として Swing で

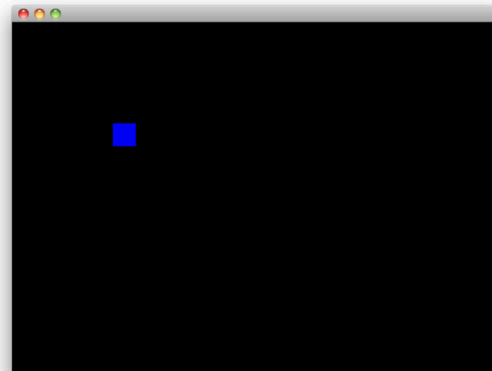

図 17 VGA グラフィクス描画ハードウェアをソフトウェアでデバッグ

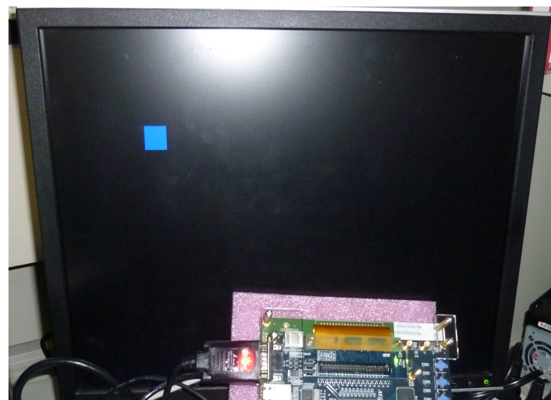

図 18 VGA グラフィクス描画ハードウェア

作成した描画対象である `SwingFrame` をインスタンシエーションすることで**図 17** に示すようにソフトウェアとして実行結果を確認できる． 一方でハードウェアモジュールを JRHI でラップした `VGAIf` をインスタンシエーションすることで，**図 18** に示すように実ハードウェアとして実行結果を確認できる．

今回は説明のために，ハードウェア生成時とソフトウェアスタック上のクラスで異なるクラス名を用いた．しかし，同じクラス名でソフトウェアあるいはハードウェアのモジュールを設計しておき，コンパイル時および実行時のクラスパスでそれぞれを呼び出すようにすることで，ソースコードはまったく変更することなく，どちらの環境も利用できるようにすることもできる．

6. まとめ

JavaRock は，複雑なアプリケーションを FPGA で処理するための開発をサポートする高位合成言語の設

計課題のうち，設計の困難さと動作検証の困難さを解決するために，Java プログラムからハードウェアへの合成を実現することを目指している．JavaRock で取り扱い可能な Java プログラムは，JVM 上でソフトウェアとして実行することも，VHDL に変換して FPGA 上のロジックとして実装することも可能であり，プログラマビリティの向上と機能検証の面で有用である．

本稿では JavaRock のコード生成規約について説明し，また，素数判定とバブルソートのプログラムをベンチマークとして，JavaRock の生成するコードの質について，リソース使用量および処理性能の評価結果を示した．評価の結果，リソース使用量に関しては，既存の合成ツールの最適化により比較的小規模に収まることが確認できた．処理性能に関しては，Core i7(2.93GHz) 上で動作する JVM 上での実行と 200MHz で動作する FPGA 上の回路でほぼ同程度の処理性能を実現できることを確認した．また，ケーススタディによって，JavaRock が可能にするソフトウェアスタックを用いたデバッグ手法について述べた．これらより，Java プログラムを FPGA 向けの高位合成言語として活用することが，生産性の観点および生成されたハードウェアロジックの有効性の観点での有用性が示された．

今後の課題としては，まだサポートできていない Java の機能のサポートが挙げられる．なお，JavaRock は，`http://javarock.sourceforge.net/` で公開している．

謝　　辞

本研究の一部は科研費（#23700054），論理回路中の疎な関係にある部分回路群の抽出と活用手法の研究，を受けたものである．ここに記して謝意を表す．

参 考 文 献

1) Ali Mashtizadeh. PHDL : a Python Hardware Design framework. Thesis (M. Eng.)–Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2007.

2) P. Bellows and B. Hutchings. Jhdl - an hdl for reconfigurable systems. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, FCCM '98, pp. 175–, Washington, DC, USA, 1998. IEEE Computer Society.

3) Jocelyn Serot, Francois Berry, and Sameer Ahmed. Implementing stream-processing applications on fpgas: A dsl-based approach. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pp. 130 –137, sept. 2011.

4) SystemC. `http://www.systemc.org/home/`.

5) David Pellerin and Scott Thibault. *Practical FPGA Programming in C*. Prentice Hall Press, Upper Saddle River, NJ, USA, first edition, 2005.

6) Handel-C Synthesis Methodology. `http://www.mentor.com/products/fpga/handel-c/`.

7) 西田浩一, Kay Andrew, 山田晃久, 神戸尚志, 野村俊夫．ハードウェアコンパイラ Bach．情報処理学会研究報告. 設計自動化研究会報告, Vol. 97, No. 103, pp. 167–172, 1997-10-28.

8) Maxeler Technologies. MaxCompiler White Paper. `http://www.maxeler.com/content/frontpage/`.

9) 日本電気株式会社．CyberWorkBench. `http://www.nec.co.jp/soft/cwb/`.

10) Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. "legup: high-level synthesis for fpga-based processor/accelerator systems". In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, pp. 33–36, New York, NY, USA, 2011. ACM.

11) Bluespec. `http://www.bluespec.com/`.

12) 結城浩. Java 言語で学ぶデザインパターン入門【マルチスレッド編】, pp. 163–196. ソフトバンク クリエイティブ株式会社, 増補改訂版, 2006.

13) FPT 2011 Design Competition – Connect6. `http://www.eecg.toronto.edu/~janders/FPT_2011_competition/`.

14) Virtex-6 ファミリ. `http://japan.xilinx.com/products/silicon-devices/fpga/virtex-6/`.

15) DE2–115 Development and Education Board. `http://www.altera.com/education/univ/materials/boards/de2-115/unv-de2-115-board.html`.