C-011

動的再構成可能ハードウェアの設計における JHDL 利用手法の提案
A Proposal of Using JHDL for Dynamically Reconfigurable Hardware Design

古島 直道 [†]　渡邊 誠也 [†]　名古屋 彰 [†]
Naomichi Furushima　Nobuya Watanabe　Akira Nagoya

1. はじめに

情報通信分野の発展により，SoC (System-on-a-Chip) などに搭載する専用ハードウェアの高性能化，低消費電力化の要望が高まり，その実装面積や設計コストの増大が問題となっている．その解決法の 1 つとして，再構成可能ハードウェア [1] の使用への注目が高まっている．また，処理の実行中にも頻繁に機能を変更可能なハードウェアである動的再構成可能ハードウェア [2][3] の研究が盛んに行われており，一部は商用製品に搭載されるなどし，実用化の段階にある．一方で，デバイス技術の進展を見越しながら，各種のアプリケーションに適したアーキテクチャや効果的な再構成手法，処理手法を今後も追求していく必要がある．そのためには，再構成可能ハードウェアにおけるアプリケーション開発を効率的に行える記述言語や，抽象度の高いレベルでの性能評価を可能とする設計環境を用意することが重要である．

ここで，特定の再構成可能ハードウェア向けに最適化された記述言語，設計環境では，設計資産の再利用などが難しくなるため，様々な再構成可能ハードウェアへ対応可能な記述言語，設計環境を用意することが有効である．このような設計環境となり得る候補として Brigham Young University（以下，BYU と略す）により開発，配布されている JHDL（Just-another Hardware Description Language）[4] がある．JHDL は Java のクラスライブラリとして実現されており，Java 言語を用いて回路の設計を行うことが可能である．Java は動的にクラスをロードするなどの機能を容易に扱えるため，実行時に回路の構成情報を生成しロードするといった再構成手法の表現にも容易に対応可能である．しかし，現在のところ，JHDL は FPGA（Field Programmable Gate Array）の設計を主な対象としており，動的再構成への対応については，部分再構成への対応の提案に留まっている．

そこで，本稿では，自律再構成方式なども含めた各種再構成可能ハードウェア向けの設計環境の実現を目的として，各種再構成可能ハードウェアを JHDL の枠組みで記述する手法を提案し，JHDL が目的とする記述言語となり得ることを示す．また，JHDL の処理系に追加すべき機能およびその実現手法について提案する．

2. 再構成可能ハードウェアの概要

再構成可能ハードウェアについて，再構成方式，構成要素の粒度，再構成を行う領域の単位，構成の変更方法といった観点から分類し，それぞれの概要を述べる．

再構成方式

再構成可能ハードウェアの再構成方式は大きく**静的再構成方式**と**動的再構成方式**に分類される．静的再構成方式では，一度構成を変更すると一連の処理が終了するまで構成を変更しない．一方，動的再構成方式は，処理の実行中にも構成を変更することができる．多くの場合，再構成の制御は外部に用意された制御用プロセッサや専用の回路によって行われる．

動的再構成方式の中でも，再構成可能ハードウェア自身が構成情報を自己参照，自己改変する手法を**自己再構成**という．自己再構成では処理の実行中に構成情報を自己参照，自己改変することを可能とすることで，処理を行いながら性能を向上させるなどの効果が期待できる．さらに，ハードウェアが自己再構成を繰り返しながら実行中の環境の変化へ適応してゆく手法を**自律再構成**という．

再構成要素の粒度

再構成可能ハードウェアは構成要素の粒度により**細粒度構成**と**粗粒度構成**に分類される．細粒度構成は，LUT (Look-Up Table) やマルチプレクサなどを基本の構成要素とし，粗粒度構成は ALU などを基本の構成要素とする．

一度に再構成を行う領域の単位

動的再構成を行う際に一度に再構成可能な領域の単位によって**部分再構成**と**全体再構成**に分類される．部分再構成ではあらかじめ決められた領域でのみ再構成を可能とする場合や，使用者が任意の領域を指定して再構成可能とする場合がある．

構成の変更手法

動的再構成における構成の変更手法としては，各構成要素が複数コンテキスト分の構成データ用のメモリを持ち，参照するメモリのアドレスを再構成の際に一斉に切り替える**マルチコンテキスト方式**と，構成データをチップ内のメモリに格納しておき，再構成の際にそのメモリから各構成要素に構成データを転送する**命令/構成データ配送方式**がある．

3. JHDL の概要

JHDL は FPGA の設計を主な対象とした設計環境であり，BYU により開発，配布されている．JHDL は Java のクラスライブラリとして提供されており，拡張されていない Java 言語を用いての回路設計が可能である．

JHDL の処理系は CUI，GUI を用いたクロックサイクル精度のシミュレーション機能や，一部の FPGA デバイス向けのネットリスト生成機能などを有している．なお，JHDL では構造記述または動作記述を用いた回路記述が可能である．ただし，現在のところ動作記述により記述された回路はシミュレーションのみ可能であり，動作記述からのネットリスト生成機能は提供されていない．

図 1 に JHDL による回路設計の流れを示し，以下で概

[†]岡山大学大学院自然科学研究科, Graduate School of Natural Science and Technology, Okayama University

Copyright © 2011 by Information Processing Society of Japan and
The Instiute of Electronics, Information and Communication Engineers
All rights reserved.

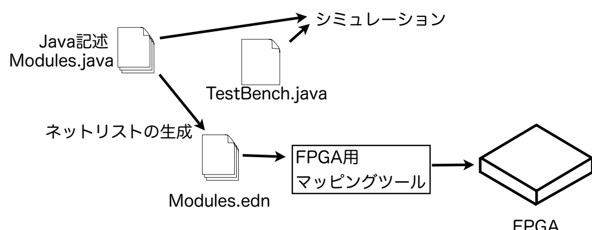

図 1: JHDL による回路設計の流れ

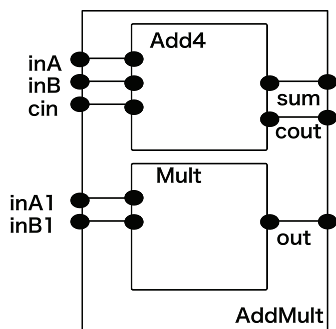

図 2: AddMult のブロック図

要を述べる．

3.1 JHDL による回路記述

図 2 に示す AddMult という名前のモジュールを構造記述により記述した例を図 3 に示し，JHDL を用いた回路記述の概要を述べる．

JHDL では 1 つのモジュールを 1 つのクラスとして記述するため，図 3 では AddMult というクラスを記述している．

クラスの記述ではまず，フィールドにモジュールの入出力ポートを記述する．次に，コンストラクタの記述を行う．コンストラクタの引数はそのモジュールを使用する親モジュールおよびモジュールに接続されるワイヤのリストなどである．親モジュールの情報は設計した回路の階層構造を構築するために使用される．回路の階層構造は木構造のデータとして処理系内部で保持され，シミュレーションやネットリストの生成に使用される．引数として渡されたワイヤは connect メソッドにより入出力ポートとの関連付けが行われる．そして，構造記述の場合はコンストラクタに具体的な回路の記述を行う．図 3 の例では 17 行目および 18 行目にて使用するモジュールのコンストラクタを呼び出しており，これにより回路の階層構造が構築される．動作記述を行う場合は回路の動作をメソッドに記述する．例えば，クロックに同期して動作する回路の場合は clock メソッドを，組み合わせ回路の場合は propagate メソッドを実装することになっている．

3.2 JHDL による回路のシミュレーション

BYU により提供されているシミュレータはクロックサイクル精度でのシミュレーションを行う．クロックはユーザが定義しない場合には JHDL の処理系が用意す

```
1   public class AddMult extends Logic {
2     public static CellInterface[]
3       cell_interface = {
4         in("inA", 4), in("inB", 4),
5         in("cin", 1), out("sum", 4),
6         out("cout", 1), out("inA1", 4), ...
7       };
8
9     public AddMult(Node parent, Wire a,
10                   Wire b, Wire cin,
11                   Wire sum, Wire cout,
12                   Wire a1, ... ) {
13      super(parent);
14      connect("inA", a);
15      connect("inB", b);
16       ...
17      new Add4(this, a, b, cin, sum, cout);
18      new Mult(this, a1, ... );
19    }
20  }
```

図 3: JHDL による回路の記述例

```
1   public class TbAddMult extends Logic
2       implements ProgrammaticTestBench {
3     Wire inA, inB, ...;
4     public static void main(String args[]) {
5       HWSystem hws = new HWSystem();
6       TbAddMult = TbAddMult(hws);
7       tb.execute();
8     }
9     public TbAddMult(Node parent) {
10      super(parent);
11      ...
12    }
13
14    public void execute() {
15      new AddMult(this, inA, inB, ...);
16      getSystem().reset();
17      getSystem().cycle(1);
18      ...
19    }
20    ...
21  }
```

図 4: JHDL によるテストベンチの記述例

るデフォルトのクロック信号が使用される．また，複数のクロック信号を使用する回路のシミュレーションも可能である．

CUI を用いたシミュレーションの手順は以下のようになる．まず，テストベンチを Java 言語によって記述し，記述したテストベンチを Java コンパイラによりコンパイルする．そして Java の実行環境で実行する．

図 3 で示した AddMult のテストを行うテストベンチの記述例を図 4 に示し，以下で詳細を述べる．

図 4 のように，テストベンチのクラスには主に main メソッド，コンストラクタ，execute メソッドを記述する．3 行目ではテスト対象のモジュールへ接続するテスト信号用のワイヤを宣言している．14 行目から 19 行目の execute メソッドではテスト対象のモジュールに対するシミュレーションの具体的な処理を動作記述によって記述している．動作記述の部分には，回路をリセットする，サイクル時間を進める，ワイヤから値を取得しコンソールに表示する，などを記述する．

3.3 BYU が提案する部分再構成の記述手法

JHDL では再構成可能ハードウェア自体を静的な回路として **3.1** で述べた方法によって記述し，そのシミュレーションを行うことは可能である．

Copyright © 2011 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.

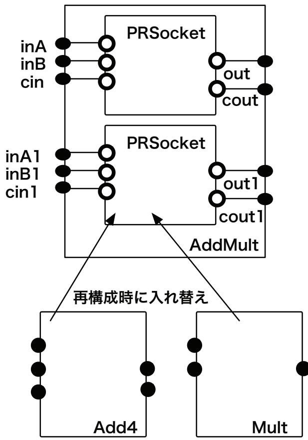

図 5: PRSocket を用いた部分再構成の例

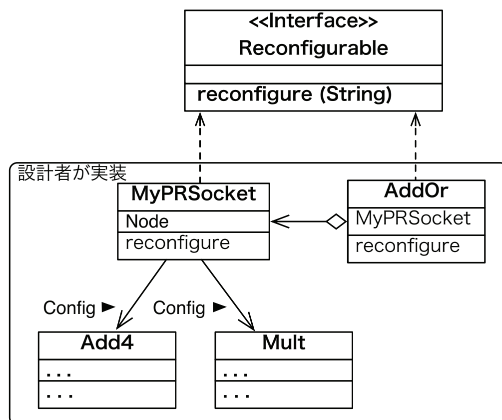

図 6: 再構成を行うモジュールを含む設計の例

より抽象度の高いレベルでの記述法として，文献 [5][6] では，動的に部分再構成を行う回路の JHDL による記述手法が提案されている．この提案は PRSocket（文献 [5] では PRSocket，文献 [6] では Reconfigurable と表現されているが，ここでは PRSocket に統一する）というクラスを導入するものである．

PRSocket を用いた動的部分再構成のイメージを図 5 に示す．図 5 は，部分再構成によって PRSocket の内部に Add4 または Mult を適宜構成する例である．PRSocket は入出力ポートおよび PRSocket 内に構成するモジュールを格納するための変数のみを持つモジュールであり，再構成が行われる領域を表す．ただし，現在のところ部分再構成を行う回路の一貫した具体的な JHDL による記述手法の提示および部分再構成へ対応するために必要となる処理系の機能の具体的な提示はされていない．

4. 各種再構成可能ハードウェアに対応可能な設計環境の提案

本研究では，**2.** で述べた各種再構成方式を対象とし，それらの設計およびシミュレーションを抽象度の高いレベルで実現することを目的とする．抽象度の高いレベルという表現について，ここでは再構成を行う契機や再構成時の動作を動作記述により明確に記述可能であることとする．しかし，**3.** で示した現状のままの JHDL の記述手法および処理系では，各種の再構成可能ハードウェアを抽象度の高いレベルで設計しシミュレーションすることは困難である．そこで，まず，既存の JHDL の記述モデルを基に，各種再構成可能ハードウェアを記述するための手法を提案する．そして，提案する記述手法を実現するために追加すべき処理系の機能およびその実現手法を提案する．

```
 1  public class MyPRSocket extends Logic
 2      implements Reconfigurable {
 3    public static CellInterface[]
 4      cell_interface = {
 5        in("a", 4), in("b", 4),
 6        in("cin", 1), out("out", 8),
 7        out("cout", 1)
 8    };
 9
10    Wire i_a, i_b, i_cin, o_out, o_cout;
11    Node n; //構成されるモジュールを格納する変数
12    public MyPRSocket(Node parent, Wire a, Wire b,
13                      Wire cin, Wire out,
14                      Wire cout) {
15      super(parent);
16      connect("a", a);      connect("b", b);
17      connect("cin", cin);  connect("out", out);
18      connect("cout", cout);
19      i_a    = a;   i_b   = b;
20      i_cin  = cin; o_out = out;
21      o_cout = cout;
22      n = null;
23    }
24    public void reconfigure(String moduleName) {
25      if (n != null)
26        n.delete();
27      if (moduleName.equals("Add4"))
28        setReconfigEvent(this, n, "Add4",
29                         i_a, i_b, ...);
30      else if (moduleName.equals("Mult")) {
31        setReconfigEvent(this, n, "Mult",
32                         i_a, i_b, o_out);
33        i_cin = gnd(); o_cout = gnd();
34      }
35    }
36  }
```

図 7: PRSocket の実装例

4.1 再構成可能ハードウェアの記述手法の提案

BYU が提案する PRSocket の概念を基に，各種再構成方式に対応する具体的な記述手法を提案する．

4.1.1 各種再構成方式に共通する記述手法

図 6 から図 8 に提案する記述手法を用いて設計したモジュールのクラス図および主なクラスの記述例を示す．この例は図 5 に示した部分再構成を行う回路中の PRSocket を MyPRSocket として実装した例となっている．

Reconfigurable インタフェース

PRSocket 等，そのモジュールが再構成を行うモジュールであることを明確にするため，また保守性等の観点から Reconfigurable インタフェースを導入する．Reconfigurable インタフェースを実装するクラスは reconfigure メソッドを実装する必要がある．reconfigure メソッドにはどの領域にどのモジュールを構成するかなどの再構成時の具体的な操作を記述する．

Reconfigurable インタフェースでは reconfigure メソッドの引数を 1 つの文字列としているが，図 8 に示す記述

Copyright © 2011 by Information Processing Society of Japan and
The Instiute of Electronics, Information and Communication Engineers
All rights reserved.

```
1       ...
2       MyPRSocket sock0, sock1;
3       public AddMult(Node parent, Wire a,
4                      Wire b, Wire cin, ...) {
5         super(parent);
6         connect("inA", a);
7         connect("inB", b);
8         ...
9         sock0 = new MyPRSocket(this, a, b,
10                               cin, out, cout);
11        sock1 = new MyPRSocket(this, a1, ...);
12      }
13      public void reconfigure(String moduleName) {
14        reconfigure(moduleName, 0);
15      }
16      public void reconfigure(String moduleName,
17                              int sel) {
18        if (sel == 0) {
19          sock0.reconfigure(moduleName);
20        } else if (sel == 1) {
21          sock1.reconfigure(moduleName);
22        }
23      }
24    }
```

図 8: 部分再構成を行うモジュール記述の一部

```
1   public void execute() {
2     MyPRSocket sock = new MyPRSocket(this, inA,
3                                      inB, ...);
4     sock.reconfigure("Add4");
5     ...  //シミュレーションの動作記述
6   }
```

図 9: 静的再構成を行うテストベンチの記述例

のように設計者が reconfigure メソッドをオーバーロードすることで，柔軟な設計が可能である．

PRSocket の記述手法

3.3 で述べたように，PRSocket は再構成が行われる領域を表すためのクラスであり，入出力ポートの違いなどにより設計者が必要とする種類だけ実装する必要がある．図 7 に示した PRSocket の実装の一例を用い，以下で詳細を述べる．

PRSocket は再構成を行うための領域を表すものであるため，12 行目からのコンストラクタではワイヤの接続以外の具体的な回路の記述を行っていない．

MyPRSocket は Reconfigurable インタフェースを実装するクラスであり，そのため，24 行目から reconfigure メソッドを実装している．reconfigure メソッドでは，27 行目および 30 行目において，引数として渡された文字列を基にどのモジュールを構成するかを決定している．28 行目および 31 行目では setReconfigEvent メソッドを用いて再構成を行うイベントをシミュレータに設定している．new 演算子を用いて直接モジュールを構成する記述も考えられるが，再構成に要する時間等のシミュレーションを考慮した場合，再構成が終了したという通知を行うなどの仕組みが必要となるため，このようなメソッドの導入が必要である．

4.1.2 各種再構成方式に対応する記述手法

静的再構成

静的再構成，つまり処理の実行前に構成を変更しておき，一連の処理が終了するまで再構成を行わない場合の記述手法について，図 9 に例を示す．図 9 はテストベンチ記述の一部であり，その記述の中で 1 つの PRSocket

```
1   public void execute() {
2     MyPRSocket sock = new MyPRSocket(this, inA,
3                                      inB, ...);
4     sock.reconfigure("Add4");
5     ...  //シミュレーションの動作記述
6     sock.reconfigure("Mult");
7     ...  //シミュレーションの動作記述
8     ...
9   }
```

図 10: 動的再構成を行うテストベンチの記述例

を生成し，その PRSocket を再構成デバイス全体と見立てている．PRSocket へは処理に必要なモジュールを 1 度のみ構成し，シミュレーション中には PRSocket への構成を行わないよう記述する．

テストベンチ内で PRSocket を用いずにモジュールを静的な回路として記述する場合とほとんど同様であるが，再構成に要する時間や再構成領域のリソース量がモジュールの構成に十分であるかを考慮するなどを考える場合，上記のような記述が必要となる．

動的再構成

動的再構成では処理の実行中に必要に応じて構成を変更する．再構成の制御を再構成可能ハードウェアの外部で行うことを想定する場合には，図 10 のように再構成の制御をテストベンチに記述することで表現が可能となる．

自己，自律再構成では，構成情報を自己参照，自己改変するため，テストベンチに再構成の制御を記述するだけではなく，設計するモジュール内に再構成の制御を行う記述を行う必要がある．また，自己，自律再構成では任意の領域が変更される可能性があるため，事前に再構成を行う部分を PRSocket の使用によって明確にしておくことは困難である．以上を踏まえた上で自己，自律再構成の記述手法として以下を提案する．まず，自己参照，自己改変を行うモジュールを記述する．そのモジュールは，条件によって任意の PRSocket の削除を行い，また条件によって新たなモジュールの記述を生成し，必要な場所に PRSocket を生成し，その中に生成したモジュールの構成などを行う．テストベンチでは PRSocket を用意し，その内部に自己参照，自己改変を行う初期回路を構成する記述を行う．これにより，PRSocket およびその内部に構成されるモジュールを初期回路が必要時に動的に生成，削除してゆくことの表現が可能となる．なお，リスト等のデータ構造を用いることで任意の PRSocket の削除または生成といった PRSocket の管理を記述可能である．また，動的に生成されたモジュールに対応するクラスのロードは，Java のリフレクション機能および動的なクラスのロード機能を使用することで実現可能である．

細粒度，粗粒度構成要素の表現手法

細粒度構成や粗粒度構成といった構成要素の粒度や構成要素の物理的な配置を意識したシミュレーションは，PRSocket を構成要素として見立て，複数個並べる記述を行うことで実現可能である．PRSocket の記述では入出力ポートのビット幅を想定する粒度に合わせ，また構成要素の物理的な位置を特定するために適切に識別子を付けるなどで表現できる．

Copyright © 2011 by Information Processing Society of Japan and
The Instiute of Electronics, Information and Communication Engineers
All rights reserved.

部分再構成，全体再構成の表現手法

部分再構成では再構成が行われる領域が単数とは限らない．そこで，図 8 に示した記述例のように，モジュール内に複数の PRSocket を用意し，reconfigure メソッドにて適切にどの PRSocket に対して再構成の指示を与えるかを記述する．また，テストベンチにて複数の PRSocket を生成し，そのそれぞれについて再構成を行うといった記述方法も可能である．

全体再構成は，部分再構成の特殊な例として考える．つまり，部分再構成を行う領域を再構成可能ハードウェアの全領域とする．図 10 に示した例のように，テストベンチ内に 1 つの PRSocket を生成し，順次その PRSocket の内部にモジュールを構成する記述を行う．

マルチコンテキスト，命令/構成データ配送方式の表現手法

再構成が行われている領域は，その間動作を停止する必要があり，また，その領域に接続されているモジュールは再構成が終了するまで待ち合わせを行う必要がある．再構成に要する待ち合わせ時間を考慮したシミュレーションは，setReconfigEvent メソッドを用いて再構成が終了する時間をシミュレータに設定することで実現可能である．

再構成に要する時間は，PRSocket などのモジュールの記述に直接記述することも考えられるが，様々な値を用いて評価を行う場合には値を変更する度に再コンパイルを行う必要があり，設計の効率が低下してしまう．そこで，再構成に要する時間を別途ファイル等に記述しておき，シミュレーション時にそのファイルを読み込むといった機能を処理系に加えることで，再コンパイルなしに効率の良い設計評価が可能となる．また，構成するモジュールの規模などによって再構成に要する時間が異なる場合を想定し，再構成の時間コストを算出するメソッドの設計者による実装を可能とする．

4.2 処理系に実装すべき機能およびその実現手法

4.1 では，各種再構成方式の JHDL による具体的な記述手法の提案を述べた．しかし，BYU が提供する現状の JHDL の処理系では，提案した記述手法に対応したシミュレーションを実現することは出来ない．そこで，JHDL の処理系を **4.1** で提案した記述手法へ対応させるために実装すべき具体的な機能および，JHDL による設計をより効率的に行うために必要な機能の拡張について，それらの実現手法とともに提案する．

4.2.1 動的再構成への対応

PRSocket を用いて動的に再構成を行う場合，処理系が保持している回路の階層構造を表現する木構造の操作が必要となる．具体的な操作としては，PRSocket 以下のノードの追加と削除が必要となる．現状の JHDL の処理系にはノードの追加や削除を行うためのメソッド等は用意されているが，これらを用いてシミュレーション中に回路構成を変更した場合，正しくシミュレーションを継続することが出来ない．これは，値を伝搬すべき信号のリストおよびそのスケジュールの更新が正しく行われていないためであり，適切に再スケジューリングを行うなどの機能の実装が必要である．

再構成に要する時間を考慮する場合には，再構成の終了を通知するなどの機能が必要となる．現状の JHDL では，複数サイクルを要するような処理の動作記述や，n サイクル後に指定された動作を行うなどのシミュレーションは不可能である．そのため，イベントドリブン方式のシミュレータを新規に実装する必要がある．また，複数サイクルを要する処理の動作記述を可能とするため，新たなメソッド，例えば behavior といった名前のメソッドを導入し，そのメソッドに設計者が複数サイクルを要する処理を記述できるようにする．

4.2.2 再構成時におけるレジスタやメモリの内容の保持

一連の処理を複数の構成に分割し，パイプラインを時分割で処理する手法などでは，再構成可能ハードウェアが構成要素として持つレジスタやメモリの値を再構成後も保持する必要がある．

JHDL のシミュレーション環境では，レジスタやメモリなどは実際の再構成可能ハードウェアのように静的なモジュールとして存在するのではなく，実行環境のメモリ上で生成，削除されるオブジェクトとして表現される．そのため，再構成の前後で PRSocket に構成されるモジュール間において，正しくレジスタやメモリといった記憶素子の内容を保持する機能が必要となる．

実現方法として，まず，モジュールの記述において，再構成の前後で値を保持すべきレジスタやメモリに識別子と値を保持するか否かのフラグを設定可能とする．処理系へは，再構成時に値の保存が必要なレジスタやメモリの値を保存し，再構成後に再構成前と同一の識別子を持つレジスタやメモリに値を復元する機能を実装する．

4.2.3 実装面積，実行時間，消費電力などの見積もり

回路設計において評価の指標となる実装面積や実行時間，消費電力を正確に見積もることは重要である．

実装面積，実行時間の見積もりは，JHDL のライブラリとして提供されている基本部品のクラスや設計者が記述したモジュールのクラスにそのモジュールの実装面積や遅延時間といった情報を格納するフィールドを持たせ，その合計を計算することで実現可能である．また，消費電力についてはシミュレーション時に信号が変化した回数などを記録し，その数値を基に計算可能である．なお，各諸量を変更させながら何度もシミュレーションを行う場合には，それらの値をモジュールの記述に直接含めてしまうとシミュレーションを行うごとに再コンパイルが必要となるため，別途用意したパラメータ設定用のファイルの利用などが必要となる．

ここで，再構成を行う場合においては，実装するモジュールのリソース使用量が再構成可能な領域のリソース量以内に収める必要があるため，リソース量の把握は必須となる．再構成可能ハードウェアでは構成要素の粒度によって，LUT や ALU などといった構成要素の使用個数をリソース量の目安とする．設計者が設計時にこれらのリソース量を正確に見積もることは困難であるため，回路の合成機能を充実させるなどが重要となる．

4.2.4 動作合成の実現

抽象度が高いレベルでのモジュールの設計，シミュレーションは設計効率を高めるために必須である．

Copyright © 2011 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.

表 1: JHDL への拡張機能等のまとめ

	主な記述手法	主に必要となる処理系の機能
各種再構成方式への対応	PRSocket クラスの実装 reconfigure メソッドの実装	Reconfigurable インタフェースの導入 再構成に伴うシミュレータの再スケジューリング
マルチコンテキスト， 命令/構成データ配送方式への対応	外部ファイルへ再構成に要する 時間を記述	setReconfigEvent メソッドの実装
複数サイクルを要する動作記述	behavior メソッドの実装	シミュレータをイベントドリブン方式として実装
再構成時の記憶素子の値の保存	記憶素子への適切な命名 値を保持するか否かのフラグの使用	再構成の処理時に値を保存し再構成後に 適切なレジスタなどへ復元する機能
諸量の評価	外部ファイルへ各種諸量を記述	外部ファイルからの各種諸量の取得 各諸量の適切な算出
動作合成	動作記述によるモジュール記述	動作合成機能

現在リリースされている JHDL では，動作記述により設計されたモジュールはシミュレーションのみ可能である．すなわち，実際のハードウェアへの実装までを考慮して設計を行う場合には構造記述による設計が必要となり，設計する回路の大規模化にともない設計者への負担が増加する．そこで，動作記述から Verilog HDL や VHDL などといった市販の合成ツールで合成可能なハードウェア記述へ自動的に変換するような動作合成機能を充実させることで，JHDL からの動作合成が実現できる．なお，JHDL は構造記述により記述されたモジュールと動作記述により記述されたモジュールを混在させることが可能であるため，動作合成による最適化が難しい部分は構造記述を使用するなど柔軟な設計が可能である．

動作合成機能の実現については C 言語や C++言語といった言語を基にしたシステム設計言語からの動作合成と同様の技術が利用できる．

5. JHDL 利用の正当性に関する考察

表 1 は **4.** で提案した各事項をまとめたものである．

4.1 で提案した記述手法により，**2.** で述べた各種再構成方式に対応する設計を行うことが可能となる．したがって，本研究の目的である各種再構成可能ハードウェア向けの設計環境の実現において，JHDL を記述言語として使用することが可能である．ただし，実用のためには JHDL の処理系に対し，**4.2** で提案した機能を実装することが必要である．特に，自己，自律再構成については，シミュレーションの実行中に生成されたクラスをロードするといった機能が必須となるが，Java に標準で提供されているリフレクションやクラスローダの機能を使用することで比較的容易に実現可能である．ここで，C++言語を用いて同様の機能を実現する場合を考えると，共有ライブラリやダイナミックリンクライブラリを使用する必要がある．そのため，プラットフォームごとに実装の変更が必要となる可能性があり，機能の実装の容易さといった点からも Java 言語の使用は妥当であると言える．

以上より結論として，動作記述を含む高い抽象度レベルの記述能力を有し，記述した回路のシミュレーションおよび性能評価を可能とし，さらに，動作合成機能などにより実際のハードウェアへの実装までを可能とする各種再構成可能ハードウェアを対象とした設計環境の実現が JHDL の利用により可能である．

6. おわりに

本稿では，FPGA の設計を主な対象とした設計環境である JHDL について，各種再構成可能ハードウェアの設計に利用するための具体的な記述手法および処理系に追加すべき機能とその実現手法を明確にした．

今後は，本稿で提案した機能などの実装を進め，提案した記述手法の記述性，シミュレーション時間の定量的な評価などを行いながら，有用な設計環境を実現し，その設計環境を用いて動的再構成可能ハードウェアのアーキテクチャやアプリケーションの処理手法に関する研究を進める予定である．

謝辞

本研究の一部は，科学研究費補助金 (課題番号 21500055) による．

参考文献

[1] 末吉 敏則，天野 英晴，リコンフィギャラブルシステム，オーム社，2005.

[2] 木村 真人，“C ベースプログラマブル HW コア「STP エンジン」の現状と展望，” 電子情報通信学会技術研究報告，RECONF2008–48，pp. 51 – 56，2008.

[3] 天野 英晴，“動的リコンフィギャラブルシステムの最近の動向，” 電子情報通信学会誌，vol. 91，pp. 478 – 483，2008.

[4] B. Hutchings and P. Bellows, “A CAD Suite for High-Performance FPGA Design,” *Proc. FCCM 1999*, pp. 12 – 24, 1999.

[5] P. Bellows and B. Hutchings, “JHDL – An HDL for Reconfigurable Systems,” *Proc. FCCM 1998*, pp. 175 – 184, 1998.

[6] P. Bellows and B. Hutchings, “Designing Run-Time Reconfigurable System with JHDL,” *Journal of VLSI Signal Processing*, vol. 28, pp. 29 – 45, 2001.

Copyright © 2011 by Information Processing Society of Japan and
The Institute of Electronics, Information and Communication Engineers
All rights reserved.