Synthesijer 2.0 ユーザー ガイド

2015 年 1 月 5 日
第 0.1 版

Synthesijer は Java ベースの高位合成処理系です．ソフトウェアとして実行可能な Java で書かれたプログラムから，同様の動作を行う VHDL あるいは Verilog HDL コードを生成します．生成した VHDL あるいは Verilog HDL のコードは FPGA ベンダの提供する開発環境を使って FPGA 上のハードウェア情報に変換できます．

表 1　改訂履歴

日付	版	項番	修正内容
2015/1/5	0.1	-	暫定版

目次

表目次

図目次

第 1 章

Java ベースの高位合成処理系 Synthesijer とは

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，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ます．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ます．

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ます．しかし，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ります．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ます．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ます．

Synthesijer は，Java プログラムからそのままハードウェアを合成する高位合成処理系です．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ります．新たなプログラミング習得の手間は必要なく，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ます．

多くの高位合成処理系は C 言語をベースとしている中，Synthesijer は Java を選択しました．Java を選択した理由は広く普及しているからということに加えて，並列処理を記述可能な Thread を言語仕様として含むため，ソフトウェアアプログラマにとって自然な形でハードウェアでも並列性を活用できます．また，C では明示的に処理系がポインタを扱う必要があり，どのように合成処理系で取り扱うかという課題がありますが，Java の場合ポインタは処理系内部に隠蔽されているため，ソフトウェアで意識する必要がないというメリットがあります．

しかし，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 では記述できません．そのため，Java だけで FPGA の性能を十分に活用することは簡単ではありません．

とはいえ，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，演算密度は高くなります．そのため，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が期待できます．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を提供することで，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ます．

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ます．このようなケースでは，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ます．補助的なプロセッサを用いると，プロセッサ用のコードやツールセットの整備と保守が必要になり，開発コストが増加してしまいます．すなわち，多少の性能低下によるデメリットがあっても，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ます．

1.1 Synthesijer を用いた開発フロー概要

1. コンパイル済みのクラスファイルをアーカイブした Synthesijer の JAR ファイルをダウンロードします[*1].
2. 馴染みの環境で HW 化する Java プログラムを書きます．開発環境で (1) でダウンロードした JAR ファイルへのクラスパスを追加しておくとアノテーションや組み込みライブラリを使用する場合に補完がきいて便利です.
3. Java プログラムを Synthesijer でコンパイルし，VHDL または Verilog HDL を作成します.
4. できあがった HDL を，いつもの合成・配置配線ツールで FPGA 用の bit ファイルに変更します．必要に応じてピン配置などの制約が必要になるでしょう.

1.2 Synthesijer の使用方法

Synthesijer はコマンドラインプログラムです．ホームページからダウンロードした JAR ファイルの名前を `synthesijer-YYYYMMDD.jar` とします.

1.2.1 Windows の場合の実行方法

Windows の場合 DOS プロンプトを使って Synthesijer を実行します．ダウンロードした JAR と合成対象のソースコードがコンパイルされるクラスパスを指定して実行します．合成対象の Java ソースコードはすべて引数で指定する必要があります.

```
java -cp synthesijer-YYYYMMDD.jar;. synthesijer.Main コンパイル対象のファイル
```

図 1.1 Windows での実行

この例では，カレントディレクトリのみをクラスパスとして追加しています．合成が成功すると，Java のソースコードに対応した HDL コードおよびコード生成時の中間データがカレントディレクトリに出力されます.

Cygwin を使って実行することもできます．ただし，クラスパス区切り文字の;(セミコロン) が Shell のコマンド終了に相当してしまいますので\"(ダブルクオーテーション) で囲む必要があります.

```
java -cp "synthesijer-YYYYMMDD.jar;." synthesijer.Main [オプション] コンパイル対象
```

図 1.2 Windows での実行

また Cygwin 上での絶対パスは Java プログラムでは解決されないので注意が必要です．相対パスで指定するか Windows のファイルシステム上のパスを指定する必要があります.

[*1] http://synthesijer.sourceforge.net からダウンロードできます

1.2.2 Unix 系 OS の場合の実行方法

PC-Unix 系，Linux，MacOSX などの Unix 系 OS の場合はターミナルを使って Synthesijer を実行します．ダウンロードした JAR と合成対象のソースコードがコンパイルされるクラスパスを指定して実行します．合成対象の Java ソースコードはすべて引数で指定する必要があります．

```
java -cp synthesijer-YYYYMMDD.jar:. synthesijer.Main [オプション] コンパイル対象
```

図 1.3 Unix 系 OS での実行

この例では，カレントディレクトリのみをクラスパスとして追加しています．合成が成功すると，Java のソースコードに対応した HDL コードおよびコード生成時の中間データがカレントディレクトリに出力されます．

1.2.3 オプション

表 1.1 は Synthesijer で指定可能なオプションです．

表 1.1 オプション 一覧

オプション	内容
-h	ヘルプを表示します
–help	ヘルプを表示します
–vhdl	VHDL ファイルを生成します
–verilog	Verilog HDL ファイルを生成します
–no-optimize	最適化機構をオフにする

`--vhdl` も`--verilog` のどちらのオプションも指定しなかった場合には，VHDL ファイルを生成します．両方同時に指定することもできます．両方指定した場合には，VHDL ファイルと Verilog HDL の両方を生成します．

1.2.4 生成されるファイル

Synthesijer では HDL ファイルの他に生成時の中間情報のファイルを同時に生成します．たとえば，Foo.java に対して生成されるファイルは表 1.2 の通りです．

表 1.2 生成されるファイル 一覧

ファイル	内容
Foo.class	class ファイル．JVM で実行できる．
Foo.vhd	生成された VHDL ファイル
Foo.v	生成された Verilog HDL ファイル
Foo_scheduler_*.txt	中間状態のスケジューラのテキスト出力
Foo_scheduler_*.dot	中間状態のスケジューラの状態遷移グラフ (Graphviz の DOT 形式)

1.3 クイックスタートガイド

例題を通じて Synthesijer を使った開発フローを紹介します．ホームページからダウンロードした JAR ファイルの名前を synthesijer-YYYYMMDD.jar としています．実行する時には，ダウンロードした実際のファイル名と読み代えてください．

図 1.4 にサンプルコードを示します．

```
public class Test{
    public boolean flag;
    private int count;

    public void run(){
        while(true){
            count++;
            if(count > 5000000){
                count = 0;
                flag = !flag;
            }
         }
    }
}
```

図 1.4 サンプルコード Test.java

この Java プログラムの run メソッドが呼び出されると，インスタンス変数の count をインクリメントし続けます．ただし，count が 5000000 を越えた時点で 0 にリセットすると同時にインスタンス変数の flag の真偽を反転します．Java では public クラスはクラス名と同じ名前のファイルに保存しなければいけません．サンプルコードは Test.java という名前で保存しましょう．

プログラムを Synthesijer を使ってコンパイルしましょう．ダウンロードした JAR ファイルへのクラスパスを指定して，図 1.5 のように synthesijer.Main を実行します．引数にコンパイル対象のファイル，この例では Test.java を指定します．

```
java -cp synthesijer-YYYYMMDD.jar synthesijer.Main Test.java
```

図 1.5 Synthesijer を使って合成する

Synthesijer はデフォルトでは VHDL ファイルを生成します．Verilog HDL コードを生成したい場合には，図 1.6 のように--verilog オプションを付けて実行します．

```
java -cp synthesijer-YYYYMMDD.jar synthesijer.Main --verilog Test.java
```

図 1.6 Synthesijer を使って Verilog HDL ファイルを生成する

ちなみに，--verilog と--vhdl オプションを両方付けて実行すると VHDL と Verilog HDL のコードを同時に生成します．

生成された VHDL コードのエンティティを図 1.7 に示します．

```
entity Test is
  port (
    clk      : in  std_logic;
    reset    : in  std_logic;
    flag_out : out std_logic;
    flag_in  : in  std_logic;
    flag_we  : in  std_logic;
    run_req  : in  std_logic;
    run_busy : out std_logic
  );
end Test;
```

図 1.7 生成された VHDL コードのエンティティ

clk，reset はクロックとリセット信号です．Synthesijer で生成されるモジュールは clk に同期する回路になります．reset は正論理のリセット信号です．flag_{in,out,we}は，元のソースコード (図 1.4) の public 変数である flag に対応する読み書き用のポートです．最後の run_{req,busy}が run メソッド呼び出し用の制御ポートです．なお，private 変数である count は外からアクセスされるべき変数ではないのでポートとして生成されません．

第 2 章

HDL モジュールの利用方法

第 3 章

生成モデル

Java に対応したハードウェアの生成モデルについて述べます.

3.1 サポートされない言語機能

Synthesijer はすべての Java プログラムを受理できるわけではありません．現在のバージョンでは，基本的に動的なインスタンス生成機能は使用できません．また，catch 節の帯域脱出，再帰呼び出しは利用できません．Thread 以外の継承，インターフェースとその実装 (implement) は未サポートです.

3.2 クラス

クラスは Java プログラム同様，ハードウェアとしてもモジュール (VHDL なら entity，Verilog HDL なら module) に対応します．`public` なメソッドやフィールド変数はポートとしてモジュールの外に引き出されます.

3.2.1 名前

パッケージとクラス名から成る絶対クラス名がモジュールの名前になります．Java でパッケージの区切りに使用される “.” は VHDL および Verilog HDL で名前に使用できないため “_” に置換されます.

3.2.2 クロックとリセット

一つのクラスは，単一のクロックに同期するハードウェアモジュールとして生成されます．クロック入力用のポートは `clk` です．また正論理の 1bit のリセット入力 `reset` を一つ持ちます.

3.2.3 入出力ポート

クラス内で定義された `public` のメソッドおよびフィールド変数に対して，対応する入出力ポートを生成します.

メソッドに対応するポート

メソッドに対しては，実行制御用のフラグ，返り値，引数に対応した入出力ポートが生成されます．メソッドの詳細については，節 3.3 を参照してください.

■実行制御用のポート　実行を開始するためのフラグ (入力ポート) と実行中を示すフラグ (出力ポート) が生成されます．それぞれメソッド名に接尾詞`_req`，`_busy` を付した名前がつけられます.

■返り値用のポート　プリミティブ型の変数の返り値を持つメソッドの場合には，その返り値に相当する出力ポートが，接尾詞_return を付けた名前で生成されます．void 型の返り値を持つメソッドに対しては何も生成されません．プリミティブ型以外の返り値のメソッドは Synthesijer では使用できません．

■引数用のポート　メソッドの引数に相当する入力ポートを生成します．メソッドの引数にはプリミティブ型の変数のみが使用できます．ポート名には，メソッド名と変数名を “_”(アンダースコア) でつないだ文字列がつけられます．

フィールド変数に対応するポート

プリミティブ型およびプリミティブ型の配列型のフィールド変数はモジュール外部から読み書きできるように対応するポートが生成されます．

■プリミティブ型の変数に対応するポート　プリミティブ型のフィールド変数に対しては，外部からの書き込み用の入力ポートと，読み出し用の出力ポートの組を生成します．それぞれ，変数名に接尾詞_in，_out をつけた名前がつけられます．それぞれのポートは型に相当するビット幅で定義されます．また，書き込みを示すフラグ用の入力ポートを生成します．変数名に_we という接尾詞をつけた名前になります．外部からの書き込み用の入力ポート_in と_we，読み出し用の出力ポート_out の組を生成します．

フィールド変数へのクラス外からの書き込みは，クラス内で定義されたメソッドの処理の書き込みと同時になる可能性があります．書き込みが同時になった場合，メソッドの処理の書き込みが優先され，クラス外からの書き込みは無効になります．

■配列型の変数　配列のフィールド変数に対しては，アドレスとデータおよび制御信号から成る 1 セットのメモリポートを引き出します．ポートの名前は変数名に対して次の接尾詞をつけたものが使用されます．

_address　アドレスに相当する入力ポート．32bit.
_din　書き込み用の入力ポート．型に対応したビット幅.
_dout　読み出し用の出力ポート．型に対応したビット幅.
_length　配列の長さの出力ポート．32bit.
_we　書き込みフラグ．32bit.
_oe　読み出しフラグ．32bit.

配列の取り扱いについての詳細は節 3.7 を参照してください．

3.3　メソッド

メソッドは呼出し制御構造付きの処理単位として生成されます．引数にはプリミティブ型の変数のみが使用可能です．返り値の型はプリミティブ型または void のみが使用できます．

3.3.1　呼び出し規約

メソッド呼び出しは，実行開始用のフラグ (リクエストフラグ) と，実行状態を示すフラグ (ビジーフラグ) で管理されます．リクエストフラグとビジーフラグは，それぞれメソッド名に接尾詞_req，_busy を付けた名前の 1bit の正論理の信号です．

リクエストフラグを’1’ にアサートすることでメソッドの処理を開始できます．メソッドの処理を開始して完了するまでの間ビジーフラグが’1’ にアサートされます．ビジーフラグが’1’ にアサートされているときはメソッド処理中で

す．リクエストフラグをアサートしてはいけません．

3.3.2 返り値

メソッド名に接尾詞_returnを付けた名前で定義される信号です．ビジーフラグが'0' の時，直前に実行した結果を取り出すことができます．

3.3.3 状態遷移機械

リセット終了後，アイドル状態に遷移します．アイドル状態に遷移するまでの間，ビジーフラグは'1' にアサートされます．

3.4 演算子

整数の +，-など軽量な演算は HDL の演算子にそのままマッピングされます．掛け算，割り算，浮動小数点数演算は予め用意したインスタンスを使って演算します．

3.5 制御構造

Java の if，while，for，および switch 文をサポートしています．これらの制御構造は状態遷移機械として実装されます．VHDL や Verilog HDL の if や case などに直接的に対応づけられるわけではありません．

3.6 データ型と変数

プリミティブ型は，型のビット幅分の変数になります．Java の型と VHDL および Verilog HDL での型の対応を表 3.1 に示します．プリミティブ型の変数はフィールド変数あるいはメソッドのローカル変数として使用することができます．

表 3.1 Java と VHDL および Verilog HDL の型の対応

Java の型	VHDL での型	Verilog HDL での型
boolean	std_logic	reg
byte	singed(7 downto 0)	reg signed[7:0]
char	std_logic_vector(15 downto 0)	reg [15:0]
short	singed(15 downto 0)	reg signed[15:0]
int	singed(31 downto 0)	reg signed[31:0]
long	singed(63 downto 0)	reg signed[63:0]
float	std_logic_vector(31 downto 0)	reg [31:0]
double	std_logic_vector(63 downto 0)	reg [63:0]

また，定義したクラス型のインスタンス変数を使用できます．インスタンス変数はクラスのフィールド変数として private かつ final で宣言する必要があります．すなわち，メソッド内の変数としてインスタンス変数を使うことはできません．

3.7　配列

プリミティブ型の配列をサポートしています．配列はインスタンス変数として宣言する必要があります．言い換えると，メソッド内の変数として宣言することはできません．配列は指定した型とサイズに応じた幅と深さのデュアルポートブロック RAM に変換されます．クラス内のメソッドからのアクセスと，クラス外からのアクセスで独立したポートを利用します．

3.8　アノテーション

Synthesijer に特殊な指示を与えるために，変数やメソッドにアノテーションを付けることができます．使用可能なアノテーションを表 3.2 に示します．

表 3.2　アノテーション

アノテーション	対象	内容
synthesijerhdl	クラス	Synthesijer コードとしてアノテーションの使用を許可する
auto	メソッド	呼び出されることなくリセット開始直後に処理を開始する．
unsynthesizable	メソッド	合成の対象としない

3.9　並列化

Thread クラスを継承化することで並列処理を記述できます．次のメソッドが追加されます．

start　スレッド処理の実行を開始する．
join　スレッド処理の終了を待つ．
yield　他のスレッド処理を起動する (ハードウェア的にはなにもしない，ダミー)

Thread 化を継承したクラスに対しては `start` メソッドを呼び出すと，run メソッドが呼び出されます．`start` メソッドの呼び出しはブロックされません．

第 4 章

Synthesijer リファレンスガイド

付録 A

その他のリソース