高位合成を用いたFPGAの開発

高村　政孝

FPGA（Field Programmable Gate Array）の大規模化に伴い、C言語やC++言語といった高級プログラミング言語を用いて、FPGAの回路設計を行う高位合成が近年注目されている。高位合成は、設計の抽象度を高めることで設計効率を向上させることのみならず、設計した回路の検証やデバッグの効率を高めることもできる。本稿では、高位合成の利点と開発ツール、並びに文章のパターンマッチングを高速に行う回路を、高位合成を用いて設計した事例について紹介する。

高位合成の利点

一般に、FPGAの回路は、HDL（Hardware Description Language：ハードウェア記述言語）と呼ばれる論理回路を設計するための言語を用いて機能を設計し、それを論理合成ツールに入力して生成する。しかし、HDLを用いた設計には、ソフトウェア設計にはない難しさが存在する。具体的には、

・処理をクロック単位で記述しなければならない
・動作の周波数や、信号が到達するタイミングを意識しなければならない
・検証やデバッグが難しい

などがある。[1)]

FPGAの規模と動作クロックは年々向上しており、それに伴い、1クロックで処理する内容や、タイミングの合わせこみの設計も難しくなってきている。また、RTL（Register Transfer Level：レジスタ転送レベル）の検証やデバッグは、シミュレーターと呼ばれる、回路を模擬動作するツールを用いて行われるが、シミュレーターは動作が非常に遅いため、多数の入力パターンに対する検証に数日、或いは数週間の時間がかかってしまうことがある。更には、ソフトウェアの統合開発環境のように、直感的で手軽なデバッグを行う機能も用意されていない。

そこで、これらの難しさを払拭し、あたかもソフトウェアを設計するようにFPGAの回路を設計する手法として注目されているのが、高位合成である。

高位合成は、回路の機能をHDLではなく、C言語やC++言語といった高級プログラミング言語を用いて記述する。それらの記述を高位合成ツール入力してHDLに変換し、出力させることで、FPGAの回路を生成する。

高位合成には、プログラミング言語を用いて記述するという特徴から、以下の利点が存在する。

・HDLよりも高い抽象度で設計することができる
・プログラミング言語の段階で、検証やデバッグを行うことができる

例えば、プログラミング言語はクロックに関する記述を行う必要がないため、設計者はクロック周波数や信号が到達するタイミングを細かく意識せずに、回路の機能を設計することができる（どのクロックでどのような処理を行うかは、高位合成ツールが自動的に判断して決めてくれる）。また、機能の検証やデバッグも変数の参照やステップ実行といった、ソフトウェアのデバッグ機能をそのまま用いることができるため、作業効率を高めることができる。

高位合成ツール

高位合成ツールは、FPGAツールベンダー各社が、様々なものを提供している。主要なものをまとめたものを、**表1**に示す。

表1　主要な高位合成ツール

FPGAツールベンダー名	高位合成ツール名
ザイリンクス株式会社	Vivado® HLS*1)
日本シノプシス合同会社	Synphony C Compiler
日本電気株式会社	CyberWorkBench®*2)
カリプト・デザイン・システムズ株式会社	Catapult C Synthesis
日本ケイデンス・デザイン・システムズ	C-to-Silicon Compiler

*1）Vivado® は、ザイリンクス株式会社の登録商標です。　*2）CyberWorkBench® は、日本電気株式会社の登録商標です。

本稿では、ザイリンクス株式会社が提供しているVivado® HLSを高位合成ツールとして用いた開発の基本的な流れと、文章のパターンマッチングを高速に行う回路を、本ツールを用いて設計を行った事例について紹介する。

開発の基本的な流れ

Vivado® HLSを用いた開発の、基本的な流れを示したものを、**図 1**に示す。

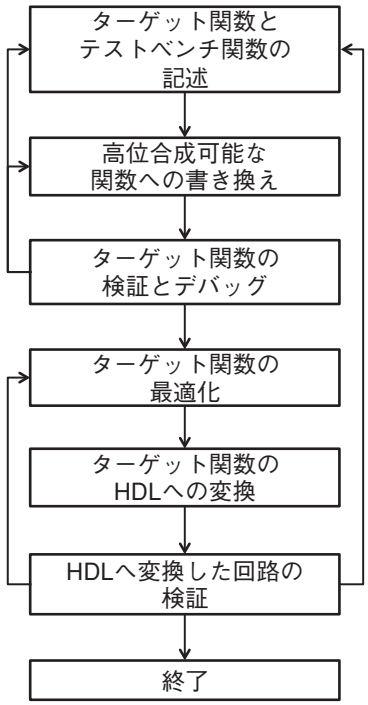

図 1　開発の基本的な流れ

まず、ターゲット関数（回路として生成しようとしている関数）と、その関数に対するテストベンチ関数（ターゲット関数の検証を行うための関数）を記述する。

次に、ターゲット関数を高位合成可能な関数に書き換える。例えば、Vivado® HLSはC言語のmalloc()関数や、C++言語のnew演算子のような、メモリ資源を動的に確保する機能を高位合成することはできないため、これらを静的なメモリ資源に確保するよう書き換える必要がある。

次に、ターゲット関数に対して、テストベンチ関数を用いて十分な検証とデバッグを行う。なお、ここまでの手順は、必ずしもVivado® HLS上で行う必要はなく、別のソフトウェア開発ツールを利用しても良い。

次に、最適化指示子を使用して、ターゲット関数の最適化を行う。[2] 高位合成の制御は、プログラムのループや変数に対して、最適化指示子を指定することで行われる。最適化指示子により、ループを展開して1クロックで複数回のループ処理を実行するよう指示したり、変数にどのFPGA資源を割り当てるかを指示したりすることができる。

最後に、ターゲット関数とテストベンチ関数を高位合成してHDLに変換した後、その回路の検証を行う。最適化指示子の指定の仕方や、使用している標準ライブラリの精度の違いなどによって、高位合成の前と後で、ターゲット関数の出力結果が異なることがあるので、合成された後の検証も行っておく必要がある。検証を行う回路は、テストベンチ関数の記述から自動的に生成されるため、別途設計を行う必要はない。

検証の中で問題が発見された場合は、前の手順に戻り、その対策を実施した後再検証を行う。

パターンマッチングを高速に行う回路の設計事例

本章では、文章のパターンマッチング（キーワード検索）を高速に行う回路を、高位合成を用いて設計した事例を紹介する。また、その中で具体的なC言語コードを提示し、最適化指示子の指定により、生成される回路がどのように変化するかも併せて紹介する。

（1）パターンマッチング回路の設計手法

文章のパターンマッチングは、本文と検索パターンの文字を先頭から1文字ずつ比較していき、全ての文字が一致したかどうかを結果として返すことにより実現できる。そのため、実行速度を無視すれば、割合簡単なHDL記述で実現することができる。

しかし、高速に実行する場合には、1つのクロックで複数のパターンや複数の文字をまとめて比較する必要がある。そのため、高速にしようとすればするほど、クロック単位の処理の記述が煩雑になってしまう。

そこで本事例では、パターンマッチングの処理をC言語で簡単に記述し、それに最適化指示子を指定することで、複雑な記述をすることなしに高速な回路を生成するという手法をとっている。

（2）パターンマッチング処理の具体例

本手法の具体的な例として、文章のパターンマッチングを行う回路の構成要素である「本文の1文字と、パターンに含まれる1文字を比較する」処理をC言語で記述したプログラムを以下に示す。

```
typedef unsigned char Uint8_t

#define LEN_PAT (32) // パターンの長さ
#define NUM_PAT (16) // パターンの種類

void subPatternMatch(Uint8_t textBuf,
  Uint8_t patBuf[NUM_PAT][LEN_PAT],
  Uint8_t resultBuf[NUM_PAT][LEN_PAT])
{
  int i,j;
  LOOP_OUTER:
  for(j = 0; j < NUM_PAT; j++)
  {
    LOOP_INNER:
    for(i = 0; i < LEN_PAT; i++)
    { // 本文1文字と、あるパターンの1文字を比較
      resultBuf[j][i] = textBuf ^ patBuf[j][i];
    }
  }
}
```

これは、本文の1文字が変数textBufに、長さ32文字の16種類のパターンが配列patBufにそれぞれ入っており、本文1文字と全てのパターンの全ての文字のEXOR（Exclusive OR:排他論理和）を取った結果を、配列resultBufに代入するという処理を行っている。よって、resultBufの要素の値が0ならば、パターンのその場所の1文字と本文の1文字が一致したことになる。

本プログラムは、既に高位合成可能な関数になっているため、そのまま高位合成を行うことができる。そのまま高位合成を行った場合の結果を、**表 2**の「指定なし」に示す。

ここでレイテンシーとは、回路に入力を開始してから出力が完了するまでのクロック数を表す。また、FF使用数とLUT使用数は、FPGAの資源として存在するフリップフロップ（FF）と、ルックアップテーブル（LUT）をどれだけ使用したかをそれぞれ表す。

表 2　生成された回路のレイテンシーと FPGA 使用量

最適化指示子の指定	レイテンシー	FF 使用数	LUT 使用数
指定なし	1057	42	55
ループの展開のみ	256	257	3796
ループの展開と配列の分割	0	0	4096

最適化指示子を何も指定しなかった場合、Vivado® HLSは概ねC言語の1ステップを1クロックで処理する回路に変換する。具体的には、

ステップ１：パターンの1文字の読み込み
ステップ２：パターン1文字と本文1文字のEXORと結果の書き込み

の2ステップを、32×16 = 512回繰り返す回路に変換する。よって、レイテンシーは1057 = 2 ×512 + 33となる（33には、変数の初期化やループの終了判定といったその他の処理が含まれる）。

（3）最適化指示子を用いた高速化

本プログラムに最適化指示子を用いることで処理を高速化し、レイテンシーを下げることができる。例えば、EXORを取る処理は、1クロックで1回行うよりも、複数回をまとめて1クロックで処理したほうが効率がよい。この最適化を模式的に示したものを、**図 2**に示す。

このようにループを展開するよう、最適化指示子を指定する。具体的には、以下のように指定する。

```
set_directive_unroll “subPatternMatch/LOOP_INNER”
set_directive_unroll “subPatternMatch/LOOP_OUTER”
```

この指定で高位合成を行ったときの結果を、**表 2**の「ループの展開のみ」に示す。最適化指示子を何も指定しなかったときと比較して、レイテンシーがおよそ1/4に短縮され、約4倍高速な回路になっていることが分かる。その一方で、EXORの処理をまとめて行うようになったため、FFとLUTの使用量は増加している。

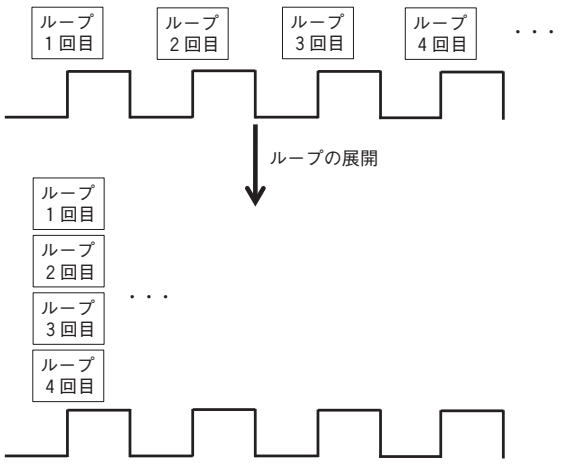

図 2　ループの展開

ここで、EXORの処理は本文1文字と、全てのパターンの全ての1文字（計512文字）とまとめて取ることもできるため、1クロックで処理を行う回路を作ることも可能なはずであるが、ループを展開しただけではそのようにならない。これは、本文やパターンの文字を読み込むインターフェースの部分が最適化されていないためである。

そこで今度は、ターゲット関数の引数に対しても、最適化指示子を指定する。具体的には、以下の指定を追加する。

```
set_directive_array_partition –type complete dim 0 “subPatternMatch” patBuf
set_directive_array_partition –type complete dim 0 “subPatternMatch” resultBuf
```

これは、ターゲット関数の引数である配列の要素を1個1個に分割し、個々の要素に同時に読み込みや書き込みができるようにする指定を表す。

この指定で高位合成を行った場合の結果を、**表 2**の「ループの展開と配列の分割」に示す。レイテンシーとFFが0になっているのは、最適化指示子の指定により、これまでは順序回路だったものが組み合わせ論理回路として最適化され、LUTのみで構成される回路に変換されたためである。

このように、最適化指示子を指定することで、元のプログラムは変更せずに高速な回路を作成することができる。本事例では、1個当たり32文字で構成される8,192個のパターンと、1個当たり140文字で構成される本文を、1秒間に13,000本文の速度でパターンマッチングを行う回路を、わずか100行のC言語プログラムで実現することができた。なお、本プログラムをHDLに変換した後の行数は、約40,000行となった。

まとめ

本稿では、従来の設計手法の難しさを克服する高位合成の紹介を、具体的な事例を交えて行った。この高位合成は、FPGAの回路設計や検証、デバッグ等を効率化し、生産性を向上させる手助けとなってくれる。

ただし、高速に動作する回路を生成するにはアルゴリズム（ソフトウェア面）とFPGAのアーキテクチャー（ハードウェア面）の両方を理解している必要がある。

OKIアイディエスは、高位合成を通じてハードウェアとソフトウェアの協調設計を推進し、高品質な回路を短期間で設計する事を可能とした。 ◆◆

■参考文献

1）三好健文：ソフトウェアをハードウェア化する、ディジタル・デザイン・テクノロジ、NO.15、pp.14-27、2012年11月1日
2)ザイリンクス株式会社：Vidado Design Suiteユーザーガイド 高位合成、
http://japan.xilinx.com/support/documentation/sw_manuals_j/xilinx2014_3/ug902-vivado-high-level-synthesis.pdf （2015年1月29日）

●筆者紹介

高村政孝：Masataka Takamura. OKIアイディエス 事業統括部 開発第一部