卒業論文

OpenMP ハードウェア動作合成のためのコード生成手法の改良

氏名	:	住井 大介
学籍番号	:	2260060060-9
指導教員	:	山崎 勝弘 教授
提出日	:	2010 年 2 月 18 日

立命館大学　理工学部　電子情報デザイン学科

内容梗概

本研究では OpenMP を用いたハードウェア動作合成システムにおいて、中間表現から VerilogHDL を生成するコードジェネレータの改良を行い、改良前のコードジェネレータが生成する回路と、改良後のコードジェネレータが生成する回路との性能検証を、3 つの OpenMP プログラムを用いて行った。

本論文では、改良したコードジェネレータによって一状態複数演算が可能となった回路と、一状態一演算しか行うことのできない以前の回路との性能検証を行う。検証方法として、N の総和、素数判定、そしてマンデルブロ集合の 3 つの OpenMP プログラムを使用した。ハードウェア動作合成システムのトランスレータを用いて OpenMP プログラムから中間表現を生成し、その中間表現から 2 つのコードジェネレータを用いて 2 種類のハードウェアモジュールを生成することで検証を行う。検証は、回路規模、実行クロック数、最大動作周波数の 3 つを行った。検証結果により、改良後のコードジェネレータにより生成された複数演算可能な回路が、ほとんど全てにおいて性能が上回っていたことが確認できた。

目次

図目次

表目次

付録

1. はじめに

近年、大規模計算機からパーソナルコンピュータ、安価な玩具に至るまで様々な電子機器にLSIが搭載されるようになり、LSIは電子機器において新しい機能やサービスを実現する最も重要な要素となっている。日々高まるユーザの要求を実現するため、電子機器に搭載されるLSIにはさらに高い処理性能、多彩な機能、高い信頼性、低い消費電力などが要求されており、LSIの回路規模や複雑さは著しく増加している。しかし、多様な製品にLSIを供給する多品種少量生産の現代においては製品の開発サイクルが短縮され、短期間で高性能なLSIを設計する必要があり、設計規模の増大に設計能力が追いつかないという状況が生じ、設計生産性の危機が問題となっている。

設計生産性の向上が可能な技術として、LSIの回路の動作をC言語などのプログラミング言語を用いてより抽象的に記述し、LSIの回路構造を動作の記述から自動合成する動作合成技術が多数提案されている。動作合成では回路記述を抽象化することで回路設計が容易に行えることに加え、最適化により抽象的な記述からASICやFPGAなど実装環境に適した回路を生成することで性能のさらなる向上が見込める。またHDLなどを用いたRTL(Register Transfer Level)の回路検証と比べ、C言語などのプログラミング言語を用いた検証では、同一の機能の検証速度が1万～100万倍以上も高速であり、検証期間の短縮が可能である。このような多くの利点から動作合成技術は商用ツールとしても多数販売され、実際の製品開発に適用され始めている。

実際に高性能な回路を実現するためには、処理の並列化が重要な要素となる。しかし現在の動作合成技術ではC言語など既存の逐次処理を実行モデルとして扱う。このような言語を入力とした動作合成技術では、問題に対する並列化手法の有効性の推定や、設計者の意図した並列動作回路を自動で生成することは難しい。実際の高位合成技術において、Spec CやHandel Cなどの言語では、並列化の制御をRTLでの動作を考慮して設計者が記述する必要がある。またその他の多くの高位合成系では最適化による並列化では、演算レベルの最適化が主であり、大規模な並列化は設計者が責任を持って記述しなければならない。

そのため主な並列化部位の選定や並列動作回路の設計は、熟練した設計者による手動での並列化に依然として頼っており、動作合成手法を導入しているにも関わらず、設計者の負担が非常に大きくなっている。また並列化したハードウェアの検証においては主にRTLのシミュレータなどを用いるため、機能、及び性能の検証が設計後期になりコストが増大するという問題もある。

本研究では、我々の研究室においてすでに設計されている既存のハードウェア合成システムに改良を加え、一状態で複数演算を可能にしたときの性能の検証を目的とする。

検証方法は、本システムにより生成されたHDL回路と、そこから依存性のない値を手書きで複数演算に修正したHDL回路を比較することで行う。

OpenMPは、共有メモリ(SMP)環境における並列プログラミングの標準APIである。OpenMP

は既存の逐次プログラムに対し、並列部を示す指示文を追加することにより並列化を行うことが可能である。また、OpenMP はマルチスレッドでの実行を行う際に、異なるスレッド間で同一のデータを同じアドレスで参照できるので、分散メモリ(DMP)環境用の MPI や PVM で要求される明示的なメッセージ・パッシングを記述する必要がないといった利点がある。

OpenMP の並列動作を容易に記述が可能であり抽象度が高いという利点を生かし、本研究で提案する動作合成システムでは、並列動作回路の動作記述に OpenMP を用いる。並列プログラミング言語をハードウェアの設計に用いることで、並列動作の記述や分析、SMP 環境を用いて設計の早期における検証・評価を容易にし、ハードウェアの動作合成における設計者の負担を軽減することが可能である。[1][3]

昨年度までに OpenMP で書かれたプログラムから中間表現までの出力が可能なトランスレータ[1][3]、中間表現から並列化された HDL を出力するコードジェネレータが実装されており[2][4]、システムとしての一応の完成は満たされている。本研究では、コードジェネレータを現状の一状態一演算の生成手法から、一状態複数演算を可能にした生成手法に改良し、N の総和プログラム、素数判定プログラム、そしてマンデルブロ集合の 3 つのプログラムで比較を行い、本システムにおいての複数演算の必要性を検証する。検証方法は、実行クロック数、最大動作周波数、回路規模の 3 つを測定した。

本論文では、第 2 章において OpenMP を用いたハードウェア動作合成システムの構成、OpenMP から中間表現に変換するトランスレータ、そして中間表現から HDL 言語に変換するコードジェネレータの生成方法を示す。第 3 章では中間コードを現在の一状態一演算から一状態複数演算の HDL 言語を生成する手法について述べ、第 4 章では 4 つのサンプルプログラムの検証の実験結果を示す。第 5 章ではコードジェネレータの改良点を挙げ、性能評価を行った。

2. OpenMP ハードウェア動作合成システム

2.1 ハードウェア動作合成システムの構成

ハードウェア動作合成システムの構成を図 1 に示す。本研究で提案するハードウェア動作合成システムは、並列化の検証・評価を行うアルゴリズム評価系と動作合成を行うハードウェア動作合成系で構成される。

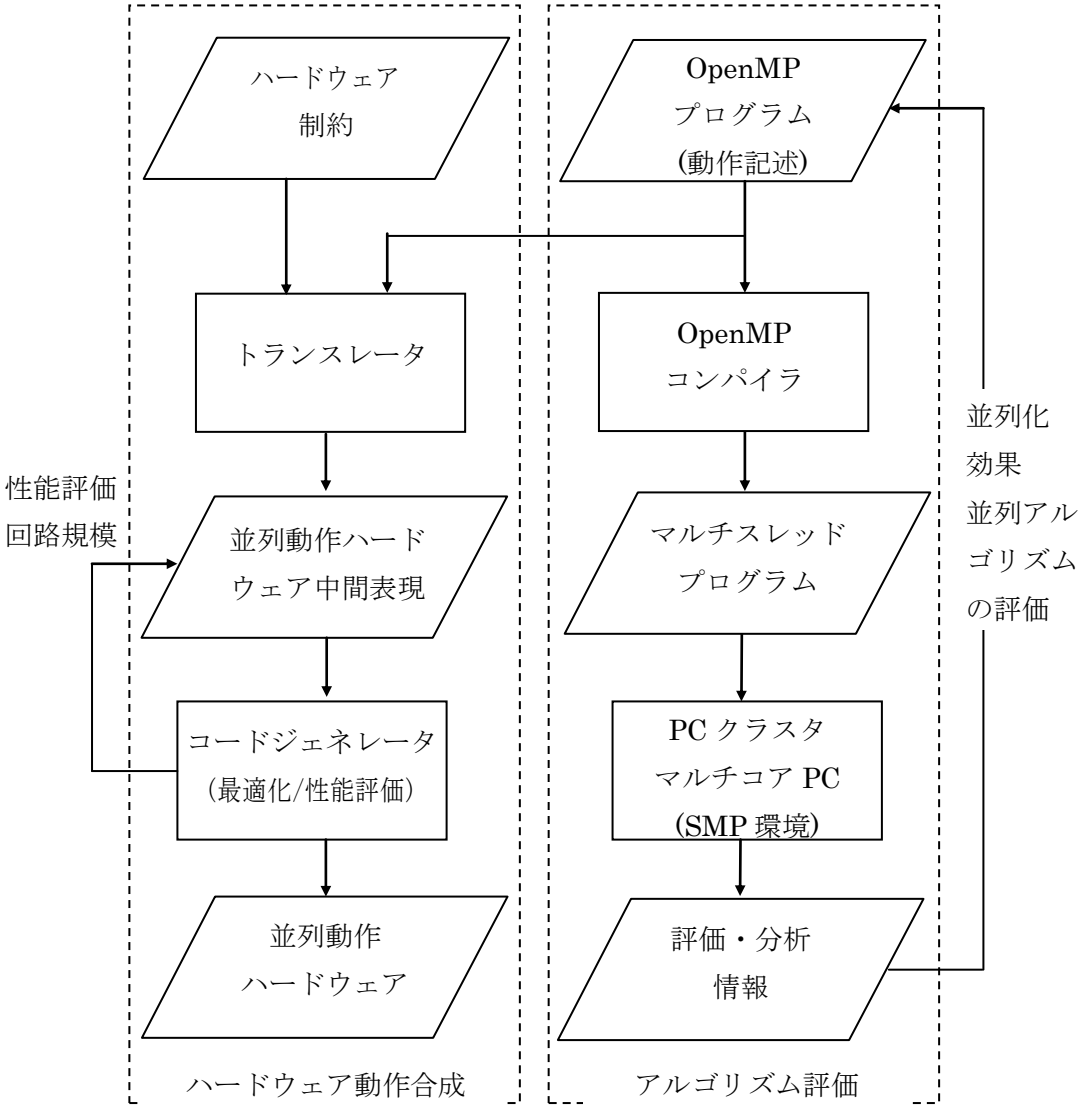


図 1：OpenMP を用いたハードウェア動作合成システム

アルゴリズム評価系では、動作記述として書かれた OpenMP プログラムを、OpenMP コンパイラによってマルチスレッドプログラムに変換し、PC クラスタやマルチコア PC などの SMP 環境によってアルゴリズムの検証と並列化の評価を行う。すなわち、プロセッサ数を変化させて実行時間を計測し、速度向上を算出して並列化の効果を明らかにする。並列化アル

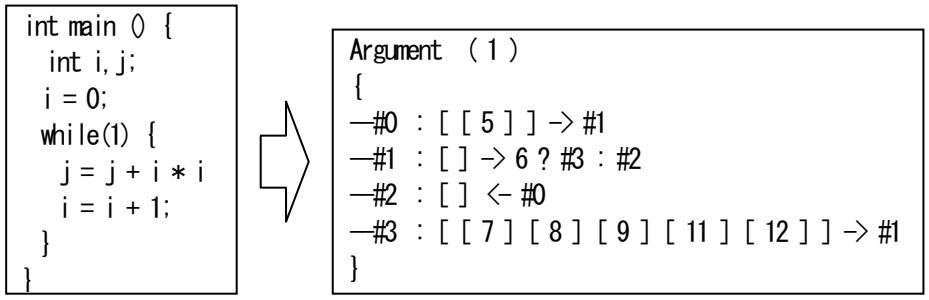
ゴリズムの評価・検証を行ない、分析結果を用いて OpenMP プログラムを改善する。SMP 環境により、高速なソフトウェアシミュレーションを行うことが出来るため、検証時間の短縮と並列化アルゴリズムの評価を設計の早期に行うことが可能である。ハードウェア動作合成系では、アルゴリズム評価系の検証後、得られた OpenMP のソースコードの動作合成を行う。トランスレータを通して中間表現に変換した後、コードジェネレータで並列動作ハードウェアを生成する。トランスレータで出力される中間コードには、OpenMP で指定された並列化情報が含まれており、コードジェネレータではそれらを用いて最適化を行い、並列動作ハードウェアを生成する。

本システムにおける C 言語から中間表現への変換を行うトランスレータ 、また中間コードからコード生成を行うコードジェネレータはすでに実装されている。本研究では、OpenMP 上では逐次処理とされている演算の依存関係を求め、一状態で複数の演算が行える回路を生成するためにコードジェネレータの改良を行い、以前のコードジェネレータが生成する回路と、今回作成したコードジェネレータが生成する回路の比較を行っている。

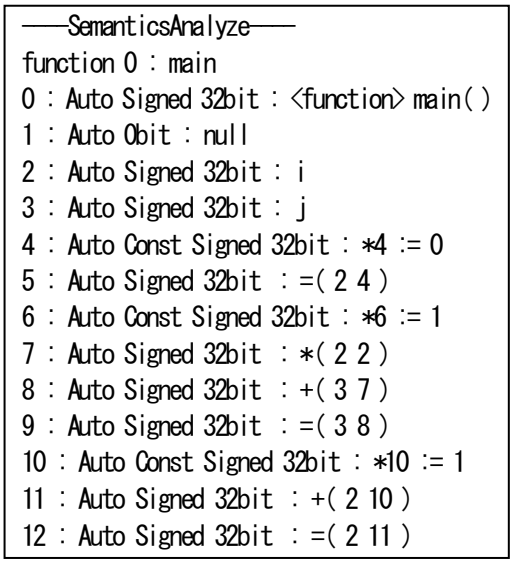
2.2 OpenMP から中間表現への変換

OpenMP から中間表現への変換手法について説明する。ハードウェア動作合成系におけるトランスレータは、動作記述である OpenMP プログラムを中間表現へと変換する。
トランスレータが生成するレジスタ転送方式である RTL 中間表現を用いてハードウェアの生成を行う。RTL の中間表現では処理を表すシンボルテーブルと制御情報を表す状態遷移表が出力され、両方を合わせてコントロールフローグラフ(CFG)を表す。シンボルテーブルは演算される変数や処理、代入先を示しており、状態遷移表によって次に遷移する状態が示される。

C 言語コードを中間コードのシンボルテーブルと状態遷移表に変換した例を図 2 に示す。サンプルのＣ言語コードは単純な while の無限ループとループ内で 1 ずつ加算される i の二乗の総和を j に格納している。状態遷移表の#0 で示される状態から、最初に CFG の 5 で示される定数の代入を表す”： =(2 4)”の処理が行われる。”： =(2 4)”ではシンボル 2 で示される変数 i に対し、シンボル 4 で示される定数 0 の代入を示している。次に while 文の条件式である#1 へ遷移し、条件式の判定を行い分岐する。ここでは無限ループの条件であるため、定数の 1 が格納されているシンボルテーブルが参照され、真であることから #3 の状態へ遷移する。#3 では CFG の 7-12 に該当する加算と代入の演算を行った後、状態をループの先頭に当たる#1 へ遷移する。



(a) C プログラム　　(b) 状態遷移表



(c) シンボルテーブル

図 2：中間表現の例

2.3　中間表現からのハードウェアの生成方法

中間表現からハードウェアを生成するコートジェネレータについて説明する。生成されるハードウェアモジュールは、スレッドに値を振り分けプログラムを制御する master と、並列時に master から値を受けて演算を行い、演算の修了を master に知らせる slave の 2 つがある。各モジュールには GrobalState、DomesticState という 2 つのステートマシンが存在し、それを用いて順序立てた処理を行う。GrobalState は状態遷移表の行数存在（図 4 の状態遷移では#0-#3 の 4 つ）し、処理別に状態遷移の条件を列挙している。DomesticState は状態遷移表の各行内の”[ ]”で囲まれた状態番号の逐次処理を依存関係の有無に分け、依存のない処理は同時に、ある処理では状態を分けて処理する。コードジェネレータより生成されるハードウェアモジュールは代入部、状態遷移部の 2 つのパートにより処理が行われる。

2.3.1 代入部

代入部の生成は状態遷移表を基に、シンボルテーブルを参照して実際に演算を実行するハードウェアモジュールを生成する。代入部の例を図 3 (a) に示す。図 3 の例は図 2 の中間表現より生成された代入部である。代入部はクロックに同期する always 文で記述する。always 文の最初の if(!XRST)は非同期リセットであり、この部分で演算に利用するレジスタの初期化を記述する。実際の代入部に当たるのは else の中の case 文によって示されている部分である。case 文の参照を現在状態遷移表の場所を表す GrobalState にし、各 GrobalState 内での演算を記述する。図 2 (a) のように GrobalState 内に i<=i+1, j<=j+i 複数の演算が存在している場合、それぞれの演算に依存関係がないかを調べる。例の場合は変数 i が依存関係にあるので、同時には演算を行えない。よって DomesticState を用いて if 文でそれぞれの演算の状態を分ける。もし依存関係がない場合、DomesticState は記述されない。

2.3.2 状態遷移部

状態遷移部の生成は状態遷移表とシンボルテーブルを用いて、分岐時の処理や全体の処理の流れを生成する。状態遷移部の例を図 3 (b) に示す。図 3 の例は図 2 の中間表現より生成された状態遷移部である。例で示す状態遷移部 (b) は、#0 から始まり、”->”で示す次の状態に遷移する。 always 文を用いて、現在の状態を表すレジスタの初期化と case 文を用いた条件による代入を行う。case 文では現在の状態と遷移先を記述していく。分岐条件の場合は、case 文の中で if 文を挿入し実現する。中間コードでは、状態遷移の分岐は高々 2 つである。分岐における if 文の条件式には外部からの入力信号や内部変数のレジスタなどが条件に合わせて列挙される。図 3（b）の G_STATE3 のように DomesticState が記述されているとき、代入部で順序立てた処理が行われているので、GrobalState は DomesticState<=DomesticState とし、代入部で DomesticState <= D_END が入力されるまで if 文を使い G_STATE3 の状態を維持する。

```
case(GrobalState)
G_INIT : oEND <= 1'b0;
G_END  : oEND <= 1'b1;
G_STATE0 : begin
    i <= 0;
end
G_STATE1 : begin
end
G_STATE2 : begin
  DomesticState <= 8'd1;
end
G_STATE3 : begin
  if(DomesticState == 8'd1) begin
    j <= j + i * i;
    DomesticState <= 8'd2;
  end
  else if(DomesticState == 8'd2) begin
    i <= i + 1;
    DomesticState <= 8'd3;
  end
  else if(DomesticState == 8'd3) begin
    DomesticState <= 8'd4;
  end
  else DomesticState <= D_END;
end
default : oEND <= 1'b0;
endcase
```

（a）代入部

```
case(GrobalState)
  G_INIT : begin
    if(iSTART == 1'b1) begin
      GrobalState <= G_STATE0;
    end
    else GrobalState <= GrobalState;
  end
  G_END : begin
    GrobalState <= G_INIT;
  end
  G_STATE0 : begin
     GrobalState <= G_STATE1;
  end
  G_STATE1 : begin
    GrobalState <= G_STATE3;
  end
  G_STATE2 : begin
    GrobalState <= G_END;
  end
  G_STATE3 : begin
    if(DomesticState == D_END) begin
      GrobalState <= G_STATE1;
    end
    else GrobalState <= GrobalState;
  end
endcase
```

（b）状態遷移部

図 3：代入部と状態遷移部の例

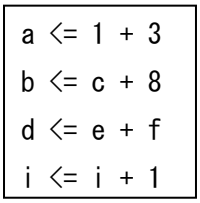
3. 演算方式の検討（仮）中間コードからのコード生成手法

3.1　一状態一演算によるコード生成

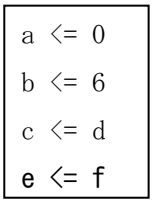
改良前のコードジェネレータシステムでは、中間表現からハードウェアモジュールを生成する際、一状態一演算の処理しか実現していない。一状態一演算は、1 クロックあたり 1 つの処理しか行わないので、最高でも各演算器を 1 つずつ用意すればモジュールを実現することができる。そのため回路の小規模化を実現することができるという利点がある。しかしそのシステムには 2 つ問題が存在する。

1 つ目は、ただの依存関係のない代入文でも逐次に処理してしまうことである。例として図 4(a)(b)の様な 2 種類の処理を示す。(a) では、一状態一演算のモジュールのとき、“+”を 1 処理あたり 1 回しか使用しないので、回路には加算器を 1 つ用意するだけでよい。しかし一状態複数演算のプログラムでは“+”を同時に 4 回使用するため、加算器 4 つ用意する必要がある。実行クロック数は減少させることができるが、回路規模に大きな差が生じることになる。

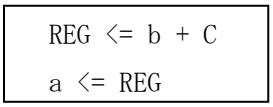
しかし (b) のように演算器を通さずに直接値を代入する場合、演算器を用いる必要がない。そのため一状態一演算の利点がなくなり、一状態複数演算をおこなうモジュールが、回路規模は変わらず、高い処理速度を実現できる。



(a)一状態一演算の利点



(b) 一状態一演算の改良すべき点



(c) 一時格納レジスタ

図 4：改良前のコードジェネレータの問題点

2 つ目の問題は、“=”と“+”が別の処理だと認識されていることである。例えば a=b+c といった式があるとすると、既存のプログラムではこの演算の処理を図 4 (c) のように 2 回の処理に分けて行う。本来ならば 1 状態で可能な演算だが 2 つの演算と認識している。このため、不必要な一時格納レジスタが発生して回路規模を大きくなり、また処理速度も敗戦距離が延びることで回路の遅延が発生する。

3.2　一状態複数演算によるコード生成

上記した通り、現在の一状態一演算のシステムでは回路規模や高速化を図る上でいくつか問題が生じる。そのため、一状態複数演算の処理が求められる。一状態複数演算モジュールは、分岐のような状態遷移が起こるまでの演算処理を、できるだけ少ない状態数で処理しようという考えである。例えば、図４（a）や（b）のような処理の場合、一状態で全ての処理を行え、システムの高速化が期待できる。しかし複数の演算を同時に処理した場合、演算結果が変わる場合がある。例として図５を挙げる。

図 5 では、簡単な代入と演算処理を行っている。左の 5 つの演算を同時に行った場合、a<=0 と f<=a=3 そして b<=3 と g<=b*c に依存関係が発生するため、f と g の結果が正しく表示されなくなってしまう。

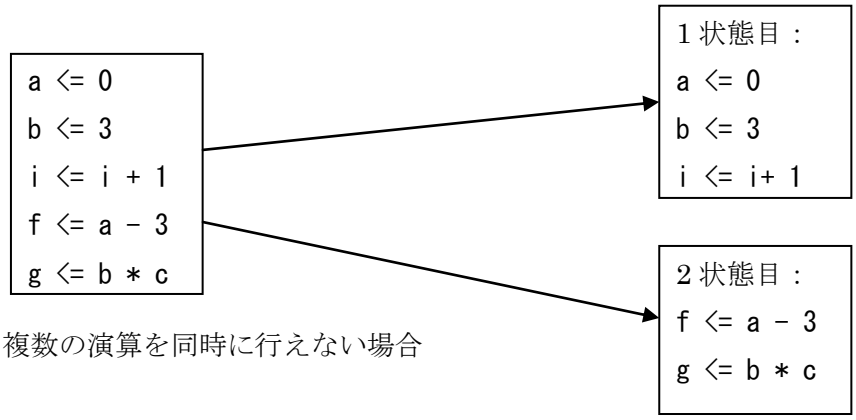


図 5：複数演算の依存関係

図５の矢印の先にある２つの処理は､5 つの処理の依存関係を考慮して分けたものである。1 状態目では、3 つの処理が依存関係を含んでいないので同時に処理が可能である。そして 2 状態目に f,g を求める演算を行うことで正しい結果を処理することができる。この依存関係を、DomesticState を用いて状態分けしている。

4. 複数演算生成の例題による検証

本章では、OpenMP で書かれた N の総和、素数判定、マンデルブロ集合の 3 つのプログラムをトランスレータにかけ中間表現を生成し、その中間表現を既存の一状態一演算を生成するコードジェネレータと改良した一状態複数演算を生成するコードジェネレータのシステムの両方で動作合成を行い、ハードウェアモジュールを生成する。生成されたモジュールの性能評価を行い、一状態複数演算システムの評価を行う。N の総和、素数判定の 2 つのプログラムでは、並列化を行うループ内に 1 つしか演算がなく、複数演算を行えない。しかし、図 4（ｃ）で述べたような問題を、複数演算を生成するコードジェネレータでは解決しているので、単純な生成の性能比較を行うことができる。マンデルブロ集合の検証では、ループ内に複数の演算が存在するので複数演算を行え、一状態との性能比較を行うことができた。なお一状態一演算の各モジュールは昨年度に刈屋、金森の 2 名によって検証されており[5][6]、そのモジュールを参考に今回の検証を行った。

動作合成システムの実験環境を表 1 に示す。

表 1：実験環境

ハードウェア動作合成	論理合成ツール	Xilinx ISE 9.1i
回路シミュレーション	PC 環境	Intel Core2 duo 2.40GHz,Memory 2GB
	シミュレーションツール	ModelSim SE 6.3c

4.1 Nの総和プログラム

4.1.1 Nの総和のハードウェアモジュール生成

本研究では、1 から 100 までの総和の計算を行った。研究に用いた N の総和の OpenMP 記述を図 6 に示す。アルゴリズムはループカウンタ i を 1 ずつ加算し、ループ内で n が i の値を加算する単純な for 文で、i のループを分割し並列化を実現している。

```
int main(void)
{
   int i;
   int n;
   n = 0;
   #pragma omp parallel for private(i)
   for( i=0 ; i<=100 ; i++ )
   {
      n += i;
   }
}
```

図 6：N の総和の OpenMP 記述

2つのコードジェネレータにより生成されたVerilogHDL記述の相違点が現れている代入部を図7（a）（b）に示す。

```
if(!XRST) begin
  oEND <= 1'b0;
  REG2 <= 32'd0;
  REG3 <= 32'd0;
  REG9 <= 32'd0;
end else begin
  case(CurrentState)
    P_INIT : oEND <= 1'b0;
    P_END  : oEND <= 1'b1;
    P_STATE5 : REG2 <= ConstNum4;
    P_STATE8 : begin
      REG2 <= ADD1_RESULT;
      oADDR <= REG2;
      end
    P_STATE9 : REG9 <= ADD1_RESULT;
    P_STATE10 : begin
      REG3 <= REG9;
      oDATA <= REG3;
      end
    default : oEND <= 1'b0;
  endcase
end
```

（a）一状態一演算

```
if(!XRST) begin
  oEND <= 1'b0;
  i <= 32'b0;
  n <= 32'b0;
end
else begin
  case(GrobalState)
    G_INIT : oEND <= 1'b0;
    G_END  : oEND <= 1'b1;
    G_STATE0 : begin
        i <= SL_START;
    end
    G_STATE3 : begin
        n <= n + i;
    end
    G_STATE4 : begin
        i <= i + 1;
    end
    default : oEND <= 1'b0;
  endcase
end
```

（b）一状態複数演算

図 7：N の総和の代入部の HDL 記述

それぞれのモジュールには相違点は 3 つ存在する。まず 1 つ目はレジスタの数である。どちらも代入部のリセット信号（XRST）時に使用される全てのレジスタを 0 とするので、リセット時のレジスタ数を見れば図 7 の（a）は 3 つ、（b）は 2 つと（b）の方が少ない。2 つ目は演算数である。（a）では 7 つの状態（case の数）で 7 つ全てに処理が行われる。それに対し（b）の状態数は 6 つと 1 つ少ない。3 つ目は状態遷移数の違いである。図 8 に N の総和の状態遷移の HDL 記述を示し、また図 9 に図 8 の状態遷移を示す。（a）では 1 ループ辺りに行われる状態数が 4 である。それに対して（b）は 3 で同処理を行うことができる。

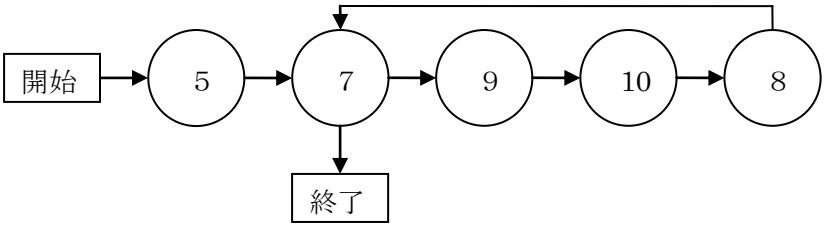
```
case(CurrentState)
  P_STATE5: CurrentState <= P_STATE7;
  P_INIT   : if(iSTART==1'b1) CurrentState <= P_STATE5;
             else CurrentState <= CurrentState;
  P_END    : CurrentState <= P_INIT;
  P_STATE7: if(REG2<ConstNum6)
              CurrentState <= P_STATE9;
            else CurrentState <= P_END;
  P_STATE9: CurrentState <= P_STATE10;
  P_STATE10: CurrentState <= P_STATE8;
  P_STATE8: CurrentState <= P_STATE7;
  default : CurrentState <= CurrentState;
endcase
```

（a） 一状態一演算

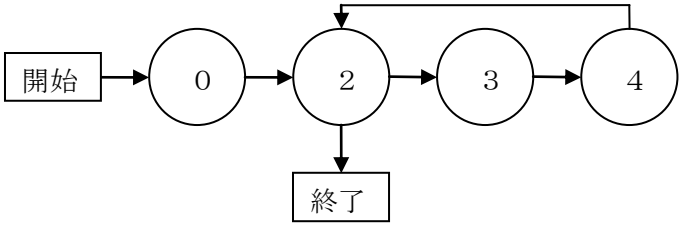
```
case(GrobalState)
  G_INIT : begin
    if(iSTART == 1'b1) begin
      GrobalState <= G_STATE0;
    end
    else GrobalState <= GrobalState;
  end
  G_END : begin
    GrobalState <= G_INIT;
  end
  G_STATE0 : begin
      GrobalState <= G_STATE2;
  end
  G_STATE1 : begin
    GrobalState <= G_END;
  end
  G_STATE2 : begin
    if(i <= SL_END) begin
      GrobalState <= G_STATE3;
    end
    else GrobalState <= G_STATE1;
  end
  G_STATE3 : begin
      GrobalState <= G_STATE4;
  end
  G_STATE4 : begin
      GrobalState <= G_STATE2;
  end
endcase
```

（b） 一状態複数演算

図 8：N の総和の状態遷移部の HDL 記述



（a）一状態一演算回路の状態遷移



（b） 一状態複数演算回路の状態遷

図 9：N の総和の状態遷移

4.1.2　N の総和プログラムの回路検証

検証方法として、2 つの生成回路の実行クロック数とスレッドごとのクロック数の変化、回路規模とスレッドごとの回路面積の変化、そして最大動作周波数の測定を行った。一状態一演算のコードジェネレータより生成された回路を“一演算”、一状態複数演算のコードジェネレータより生成された回路を“複数演算”とし、表 2 に測定結果を示す。

表 2：N の総和の測定結果

スレッド数		1	2	4
回路規模(Slices)	一演算	101	181	383
	複数演算	65	143	298
回路面積比	一演算	1.00	1.79	3.79
	複数演算	1.00	2.20	4.58
実行クロック数(cloks)	一演算	407	207	107
	複数演算	310	160	85
クロック減少比	一演算	1.00	1.97	3.80
	複数演算	1.00	1.94	3.65
最大動作周波数（MHz）	一演算	130.056	130.531	130.056
	複数演算	149.276	149.276	149.276

測定結果より、全て回路規模は複数演算の回路がよい数値を出している。しかし回路面積比を見てみると、一演算の回路の方が好ましい数値を示している。これは、配線の問題が考えられる。複数演算の回路は 4 スレッドのループの開始値と終了値を、制御部により分割してそれぞれのスレッドに送っている。しかし一演算の回路は 4 スレッドにそれぞれループの開始値と終了値を手書きで入力するため、制御部からスレッド部への配線が必要ない。このため回路規模が小さくなっていると考えられる。

次に実行クロック数については、複数演算の回路の方がよい結果を示している。またクロック減少比については、ほぼ同様の減少比と示している。実行クロック数の差は、4.1.1 で述べたように 1 ループあたりのクロック数が一演算の回路のほうが 1 つ多いからだと考えられる。

最大動作周波数は、ほぼスレッド数に関係なく複数演算の回路が上回る結果となった。この理由としては一演算の特性が関係していると考えられる。一演算の回路は、N の総和プログラムで計算される n += i と i++という 2 つの加算で同じ加算器を使用している。そのため加算器までの配線の長さ分速度が低下した。また、スレッド数を変えても変化が見られないのは、全てのスレッドで同じ処理を行っているため、分割しても最も遅い演算の速度に変化はないからである。

4.2　素数判定プログラム

4.2.1　素数判定のハードウェアモジュール生成

本研究では、剰余に一状態一演算のコードジェネレータが対応していないため、減算を代用することで同様の結果を求めるプログラムを用いて検証を行った。素数判定プログラムでは、素数でない値と判定された時点で処理が終了するので、全てのループ処理を最後まで実行して適切な実験結果を得るために、100003 という素数を用いて実験を行った。実際に用いた OpenMP 記述を図 9 に示す。

```
void main(void)
{
   Int iDATA,i,m,oDATA;

   #pragma omp parallel for private(i,ans,m)
   for(i=0;i<iDATA;i++)
   {
      if(i==0) i=2;
      for(m=iDATA;m>=i;m=m-i)
      {
      }
      if(m==0) oDATA=1;
   }
}
```

図 10：素数判定の OpenMP 記述

2 つのコードジェネレータにより生成されたハードウェアモジュールを付録 a,b に示す。相違点は N の総和のプログラムと同様で、レジスタ数が一演算の回路では 8 つ、複数演算の回路では 4 つであり、複数演算の回路の方が小規模な回路を見込める。図 10 に複数演算のコードジェネレータが生成した状態遷移を示す。並列化を行っている部分は図 10 中の点線で囲まれている部分である。複数演算の回路の状態数は 9 であるが、一演算の回路では 10 であり、複数演算の回路の実行クロック数が少ないことが見込める。

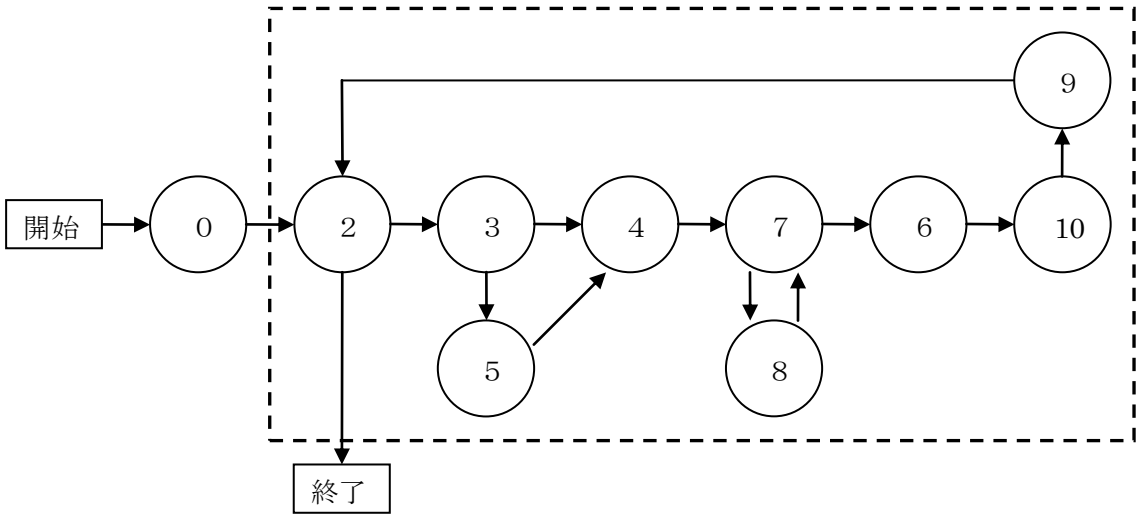


図 11：素数判定の複数演算回路の状態遷移

4.2.2　素数判定プログラムの回路検証

検証方法は 4.1.2 と同様で、2 つの生成回路の実行クロック数とスレッドごとのクロック数の変化、回路規模とスレッドごとの回路面積の変化、最大動作周波数の測定を行った。一状態一演算のコードジェネレータより生成された回路を“一演算”、一状態複数演算のコードジェネレータより生成された回路を“複数演算”とし、表 3 に測定結果を示す。

表 3：素数判定の測定結果

スレッド数		1	2	4
回路規模(Slices)	一演算	141	290	626
	複数演算	119	239	480
回路面積比	一演算	1.00	2.06	4.44
	複数演算	1.00	2.01	4.03
実行クロック数(cloks)	一演算	3350295	3025275	2759904
	複数演算	2733552	2564648	2456306
クロック減少比	一演算	1.00	1.11	1.21
	複数演算	1.00	1.07	1.11
最大動作周波数(MHz)	一演算	137.344	134.571	134.571
	複数演算	146.649	146.628	145.264

測定結果より、回路規模、実行クロック数、最大動作周波数と全ての測定結果において複数演算を生成するコードジェネレータが高い数値を出している。

素数判定では、スレッド数を増やしてもクロック減少比はあまり向上しなかった。この理由は 1 ループあたりの演算数の違いである。今回の素数判定では、入力値をループカウンタ i で減算し続け、0 にならないときは素数、0 になったときは素数でない。という判定方法をとっている。今回入力値は 100003 で、もっとも低い i の値の 2 は 100003/2=50002 回減算を行う。i=1000 のときは 100003/1000=101 回の減算でループが終了する。このため i の低い数値を演算するスレッドほど遅延が生じてしまい理想的なスレッド数に対する減少比を得られなかった。

複数演算の最大動作周波数は、スレッド数の増加と共にわずかに低くなっている。これは、スレッド数が増加することによる配線の増加が原因で発生する遅延だと考えられる。

4.3 マンデルブロ集合

4.3.1 マンデルブロ集合のハードウェアモジュール生成

マンデルブロ集合とは、漸化式 $z_{n+1} = z_n^2 + c$, $z_0 = 0$ で定義される複素数列 $\{z_n\}_{n \in N}$ が、$n \rightarrow \infty$ の極限で無限大に発散しないという条件を満たす複素数 c 全体が作る集合のことである。本研究では 100×100 の画像を用いて検証を行った。マンデルブロ集合の OpenMP 記述を図 11 に示す。

```
int main()
{
   int i, j;
   int a, b;
   int bit_size;
   int xn, yn;
   int xn1, yn1;
   int counter;

   a = -2<<16;
   b =  2<<16;
   bit_size = ((2<<16)-((-2)<<16))/(99);

   #pragma omp parallel for
   for( i=0 ; i<100; i=i+1 )
   {
      for( j=0 ; j<100 ; j=j+1 )
      {
         xn=0;
         yn=0;
         xn1=0;
         yn1=0;
         for( counter=0 ; counter<30 ; counter=counter+1 )
         {
            xn1 = (xn>>8)*(xn>>8) - (yn>>8)*(yn>>8) + a;
            yn1 = (xn>>8)*(yn>>8);
            yn1 = ((2<<16)>>8)*(yn1>>8) + b;

            xn = xn1;
            yn = yn1;
            if( ((xn>>8)*(xn>>8)+(yn>>8)*(yn>>8)) > (4<<16) )
            {
               break;
            }
         }
         a = a + bit_size;
      }
      b = b - bit_size;
      a = (-2)<<16;
   }
}
```

図 12：マンデルブロ集合の OpenMP 記述

一状態一演算を生成するコードジェネレータでは、図 11 中の太枠に囲われている部分のみ生成されていたので、同じ条件の下比較を行った。2 つのコードジェネレータにより生成されたハードウェアモジュールを付録 c,d に示す。

マンデルブロ集合の演算は、1 つのループ内で複数の演算が行われているため、一状態一演算と一状態複数演算のコードジェネレータの比較として最適である。図 12 に複数演算のコードジェネレータが生成した状態遷移を示す。上下 2 つの点線で囲まれた部分は、上が DomesticState、下が GrobalState とそれぞれ違うステートマシンが働いている。GrobalState は 4.1N の総和や 4.2 素数判定で用いられているステートマシンで、2 から 5 までの 3 つの処理をループしている。DomesticState は、GrobalState の 1 状態の中に複数の演算があり、さらに依存関係を持っていて順序立てて処理を行わなければならないときに生成されるステートマシンである。図 12 では GrobalState が 3 になったときに DomesticState が 1 に進み、DomesticState が end になるまで GrobalState は 3 の状態を維持する。そして DomesticState に end が代入されたとき GrobalState は 4 に進む仕組みとなっている。複数演算回路の 1 ループあたりの状態数は、GrobalState が 4、DomesticState が 5 なので、合計 9 となる。一状態一演算の回路が 1 ループに必要な状態数が 33 なので、動作周波数が同じであった場合、ループだけの処理で比較すると 3 倍以上高速な処理が行える。その他の相違点は、他と同様でレジスタの数の違いがあり、複数演算の回路の方が少ないレジスタで演算が行える。実際にそれぞれのコードジェネレータから生成されたモジュールは付録 c,d に示す。

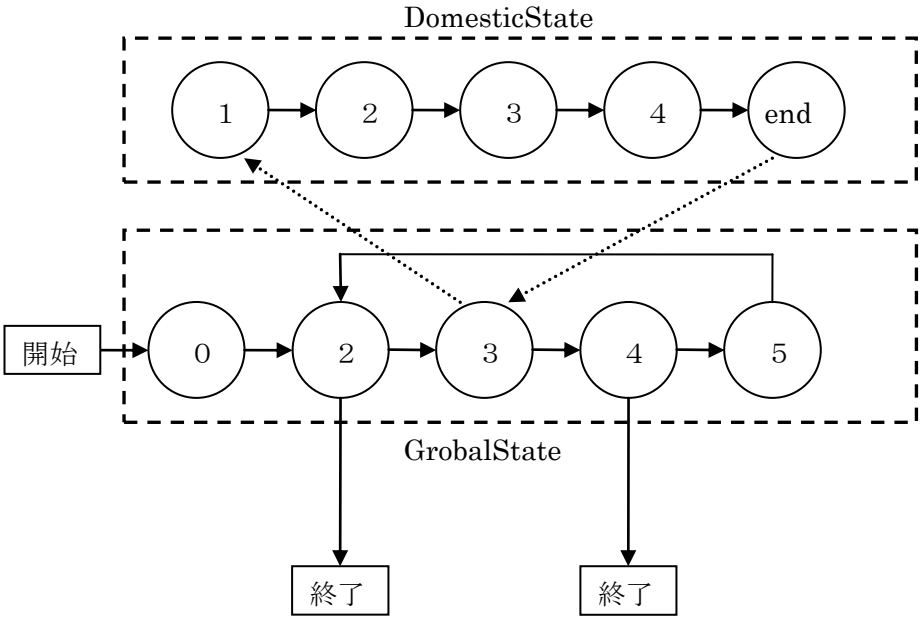


図 13：マンデルブロ集合の複数演算回路の状態遷移

4.3.2　マンデルブロ集合の回路検証

検証方法は 4.1.2 と同様で、2 つの生成回路の実行クロック数とスレッドごとのクロック数の変化、回路規模とスレッドごとの回路面積の変化、最大動作周波数の測定を行った。一状態一演算のコードジェネレータより生成された回路を“一演算”、一状態複数演算のコードジェネレータより生成された回路を“複数演算”とし、表 4 に測定結果を示す。

表 4：マンデルブロ集合の測定結果

スレッド数		1	2	4
回路規模(Slices)	一演算	892	1787	3574
	複数演算	855	1711	3437
回路面積比	一演算	1.00	2.00	4.01
	複数演算	1.00	2.00	4.02
実行クロック数(cloks)	一演算	919047	461430	386610
	複数演算	166361	101150	61935
クロック減少比	一演算	1.00	1.99	2.38
	複数演算	1.00	1.64	2.69
最大動作周波数(MHz)	一演算	44.140	44.195	44.195
	複数演算	48.544	48.544	48.544

測定結果より、回路規模はほとんど同じ値を示した。これは、一演算の回路では演算結果を格納するレジスタ数が増加と、複数演算の回路では複数の演算を同時に行うために用いられた演算器数の増加が似たような結果を示したことが理由として考えられる。その結果、どちらの回路もスレッド数の増加に比例して回路面積比も変化した。

実行クロック数は、複数演算回路の測定結果の方が約 5 倍少ない結果となった。これは 4.3.1 で前述したように、1 ループあたりの状態数の違いによるものである。複数演算の利点が生かされるプログラムの検証であったので、検証結果に大きな差がついた。

それぞれスレッド数に対する実行クロック数の変化は、スレッドの増加に比例しない結果が得られた。これは、プログラムのループ回数が一定に決まっていないことが問題に挙げられる。図 11 の OpenMP より、for 文の中にループを中断する “break” 存在するので、一度のループで処理される回数が不規則となる。そのため、並列化を行っても期待通りのクロック減少比を得ることができなかった。

最大動作周波数は、これまでのプログラムの検証と比べるとどちらも非常に低い数値を出した。この理由としては回路規模の増加が考えられる。回路規模が増加することにより一本一本の配線が長くなり遅延に繋がる。また、一演算の回路ではレジスタ数が多いために配線が増え、複数演算の回路では一度に多項の複数演算を行なうために配線が増えたた

めに遅延が発生した。

5. 複数演算を生成するコードジェネレータ

5.1 改良点

今回コードジェネレータを新たに改良するにあたり、3 つ目的を持って取り組んだ。

1 つ目は、今までのコードジェネレータに不足している点を補うことである。第 3 章で述べた通り、一状態一演算を生成するコードジェネレータには“不必要な逐次処理”と“一時格納レジスタ”という 2 点が、回路の高速化や回路規模などネックとなっていた。

2 つ目は、制御部の生成である。去年度の先輩の研究[5, 6]を参考にしていると各スレッドを束ねる役割を果たす制御部は全て手動で生成されていた。そのためスレッド自体を自動生成しても制御部の作成に時間がかかるので、当初の目的である設計時間の短縮が行えない。今回作成したコードジェネレータでは、制御部（master）とスレッド部（slave）の 2 つのモジュールを生成し、必要であれば自動で 2 つのモジュールの値の受け渡しを行えるように設計した。

3 つ目は、生成されたモジュールの見易さを考慮した。今までのコードジェネレータから生成される回路は行数が長いという問題もあるが、それを差し引いても理解しにくいモジュールであった。OpenMP 動作合成システムは、状態遷移、値の代入、そして演算を行うことが可能だが、未だ表現できない変換もいくつか存在し、手書きを加える必要がある。そのときに理解できない HDL 言語では、プログラムを完成させるまでに無駄な時間がかかってしまう。今回作成したコードジェネレータでは、できるだけ手書きに近い回路を生成できるように心がけた。

5.2 考察

今回コードジェネレータの改良を行い、ほぼ全ての検証で改良前よりよい結果を得られた。しかしこの OpenMP 動作合成システムは、トランスレータにもコートジェネレータにもまだまだ改良の余地がある。

トランスレータについては、まだまだ対応していない表現が多いことがあげられる。例えば符号なしの変数を定義するときに用いる“unsigned”や変数のビット幅を倍にする“long”、並列処理の結果を統合するときなどに用いるリダクション演算などが挙げられる。これらは OpenMP で記述されていると中間表現にエラーが生じるか、記述されなくなる。中間表現で記述していないものを VerilogHDL で表現することは不可能なので、手書きをせずによりスムーズに動作合成を行うために、はコードジェネレータの改良が必要不可欠である。

コードジェネレータの問題は 2 つあり、1 つ目は回路の最適化である。今回改良したことにより、依存関係を求めて複数の演算同時に行うことが可能になった。しかしそのために余分な演算器が増えたり、処理時間の長い演算にあわせて全体の周波数を低くしたりする

問題が発生するので、実行クロック数が減少しても結果的に処理が遅い回路を作ることがある。

2 つ目は多次元配列の問題である。OpenMP は仮想的なアルゴリズムだけを考えればいいので、多次元配列が可能である。しかし回路構造を考えたとき、多次元配列は表現困難なため、自動生成が難しい。

今後コードジェネレータを改良する場合、一状態一演算と一状態と複数演算の最適化が求められる。場合に合わせて複数演算すべきか一演算すべきかを判定するシステムが必要である。

6. おわりに

本研究ではハードウェア動作合成システムにおいて、コードジェネレータの改良と、3 つの OpenMP プログラムを用いた新しいコードジェネレータの性能検証を以前のコードジェネレータを比較して行った。

新しいコードジェネレータで生成した回路の方が、ほぼ全ての検証においてよい結果を出した。それは、無駄なレジスタを省いたことによる回路の小規模化と、一状態で複数演算を可能にしたことによる高速化が理由に挙げられる。

今後の課題としては、より多くのプログラムで検証し、対応していない記述をなくすこと。演算ごとに一状態あたりの処理を分けるか同時に行うかを判断し、最適化を行うこと。それによって、回路の高速化と回路規模の縮小を追及することである。

謝辞

本研究の機会を与えてくださり、貴重な助言、ご指導をいただきました山崎勝弘教授に深く感謝いたします。また、本動作合成システムを立ち上げ、貴重な助言を頂いた中谷嵩之氏、松崎裕樹氏に深く感謝いたします。

参考文献

[1] 中谷嵩之，松崎裕樹，山崎勝弘：OpenMP によるハードウェア動作合成システムの設計と検証，第７回情報科学技術フォーラム論文集 FIT2007，C-006，2007.

[2] 松崎裕樹，中谷嵩之，山崎勝弘：OpenMP によるハードウェア動作合成システム：コードジェネレータの実装と画像処理による評価，第６回情報科学技術フォーラム論文集 FIT2008，C-008，2008.

[3] 中谷嵩之：OpenMP によるハードウェア動作合成システムの設計と検証”，立命館大学大学院理工学研究科修士論文，2006.

[4] 松崎裕樹：OpenMP によるハードウェア動作合成システム:コードジェネレータの実装と画像処理による評価，立命館大学院理工学研究科修士論文，2008.

[5] 金森央樹：OpenMP ハードウェア動作合成システムの検証（Ⅰ），立命館大学理工学部卒業論文，2009.

[6] 苅屋徹：OpenMP ハードウェア動作合成システムの検証（Ⅱ），立命館大学理工学部卒業論文，2009.

[7] デビット･トーマス，アンドリュー・ハント：達人プログラマーズガイドプログラミング Ruby，ピアソン・エデュケーション，2001.

[8] 青木峰郎，後藤裕蔵，高橋征義：Ruby レシピブック 268 の技，ソフトバンクパブリッシング，2004.

[9] 並木秀明：ディジタル回路と Verilog HDL，技術評論社，2008.

[10]小林優：入門 Verilog HDL 記述，CQ 出版社，1996.

付録 a　一状態一演算の素数判定回路（代入部、状態遷移部のみ）

＜代入部＞

```
always @ (posedge CLK or negedge XRST) begin

 if(!XRST) begin
    oEND <= 1'b0;
    i <= 32'd0;
    m <= 32'd0;
    oDATA <= 32'd0;
    REG8 <= 32'd0;
    REG9 <= 32'd0;
    REG15 <= 32'd0;
    REG16 <= 32'd0;
 end else begin
  case(CurrentState)
     P_INIT : oEND <= 1'b0;
     P_END   : oEND <= 1'b1;
     P_STATE7 : i <= iDATAfor;          //データ分割したいところ
     P_STATE9 : i <= ADD1_RESULT;
     P_STATE13 : i <= ConstNum12;
     P_STATE14 : m <= iDATA;              //要修正
     P_STATE16 : REG16 <= SUB1_RESULT;
     P_STATE17 : m <= REG16;
     P_STATE21 : oDATA <= ConstNum20;
     default : oEND <= 1'b0;
  endcase
 end
end
```

＜状態遷移部＞

```
always @(posedge CLK or negedge XRST) begin
  if(!XRST)
     CurrentState <= P_INIT;
  else
      case(CurrentState)
          P_STATE7: CurrentState <= P_STATE8;
          P_INIT     : if(iSTART==1'b1) CurrentState <= P_STATE7;
                       else CurrentState <= CurrentState;
          P_END      : CurrentState <= CurrentState;
          P_STATE8: if(i<iDATAto) CurrentState <= P_STATE11;
                       else CurrentState <= P_END;
          P_STATE11: if(i==ConstNum10) CurrentState <= P_STATE13;
                       else CurrentState <= P_STATE14;
          P_STATE14: CurrentState <= P_STATE15;
          P_STATE13: CurrentState <= P_STATE14;
          P_STATE19: if(m==ConstNum18) CurrentState <= P_STATE21;
                       else CurrentState <= P_STATE9;
          P_STATE15: if(m>=i) CurrentState <= P_STATE16;
                       else CurrentState <= P_STATE19;
          P_STATE16: CurrentState <= P_STATE17;
          P_STATE17: CurrentState <= P_STATE15;
```

```
            P_STATE9: CurrentState <= P_STATE8;
            P_STATE21: CurrentState <= P_STATE9;
            default : CurrentState <= CurrentState;
        endcase
end
```

付録 b　一状態複数演算の素数判定回路（スレーブの代入部、状態遷移部のみ）

＜代入部＞

```
always @ (posedge CLK or negedge XRST) begin

    if(!XRST) begin
      oEND <= 1'b0;
      DomesticState <= D_END;
      i <= 32'b0;
      m <= 32'b0;
      o <= 32'b0;
    end

    else if(DomesticState == D_END) begin
      DomesticState <= DomesticState + 1;
    end

    else begin
      case(GrobalState)
        G_INIT : oEND <= 1'b0;
        G_END   : oEND <= 1'b1;
        G_STATE0 : begin
              i <= SL_START;
        end
        G_STATE1 : begin
           DomesticState <= 8'd1;
        end
        G_STATE2 : begin
        end
        G_STATE3 : begin
        end
        G_STATE4 : begin
              m <= SL_END;
        end
        G_STATE5 : begin
              i <= 2;
        end
        G_STATE6 : begin
        end
        G_STATE7 : begin
        end
        G_STATE8 : begin
              m <= m - i;
```

```
        end
        G_STATE9 : begin
            i <= i + 1;
        end
        G_STATE10 : begin
            o <= 1;
        end
        default : oEND <= 1'b0;
      endcase
    end
end

＜状態遷移部＞
always @(posedge CLK or negedge XRST) begin
    if(!XRST) begin
      GrobalState <= G_INIT;
    end
    else begin
      case(GrobalState)
        G_INIT : begin
          if(iSTART == 1'b1) begin
            GrobalState <= G_STATE0;
          end
          else GrobalState <= GrobalState;
        end
        G_END : begin
          GrobalState <= G_INIT;
        end
        G_STATE0 : begin
            GrobalState <= G_STATE2;
        end
        G_STATE1 : begin
          GrobalState <= G_END;
        end
        G_STATE2 : begin
          if(i < SL_END) begin
            GrobalState <= G_STATE3;
          end
          else GrobalState <= G_STATE1;
        end
        G_STATE3 : begin
          if(i == 0) begin
            GrobalState <= G_STATE5;
          end
          else GrobalState <= G_STATE4;
        end
        G_STATE4 : begin
            GrobalState <= G_STATE7;
        end
        G_STATE5 : begin
            GrobalState <= G_STATE4;
        end
```

```
        G_STATE6 : begin
          if(m == 0) begin
            GrobalState <= G_STATE10;
          end
          else GrobalState <= G_STATE9;
        end
        G_STATE7 : begin
          if(m > i) begin
            GrobalState <= G_STATE8;
          end
          else GrobalState <= G_STATE6;
        end
        G_STATE8 : begin
            GrobalState <= G_STATE7;
        end
        G_STATE9 : begin
            GrobalState <= G_STATE2;
        end
        G_STATE10 : begin
            GrobalState <= G_STATE9;
        end
      endcase
    end
end
```

付録 c　一状態一演算のマンデルブロ集合回路（代入部、状態遷移部のみ）

＜代入部＞

```
always @ (posedge CLK or negedge XRST) begin

  if(!XRST) begin
     oEND <= 1'b0;
     xn <= 32'd0;
     yn <= 32'd0;
     xn1 <= 32'd0;
     yn1 <= 32'd0;
     counter <= 32'd0;
     REG0 <= 32'd0;
     REG19 <= 32'd0;
     REG21 <= 32'd0;
     REG24 <= 32'd0;
     REG26 <= 32'd0;
     REG27 <= 32'd0;
     REG29 <= 32'd0;
     REG31 <= 32'd0;
     REG32 <= 32'd0;
     REG33 <= 32'd0;
     REG34 <= 32'd0;
     REG37 <= 32'd0;
     REG39 <= 32'd0;
```

```
    REG40 <= 32'd0;
    REG44 <= 32'd0;
    REG46 <= 32'd0;
    REG48 <= 32'd0;
    REG49 <= 32'd0;
    REG50 <= 32'd0;
    REG55 <= 32'd0;
    REG57 <= 32'd0;
    REG58 <= 32'd0;
    REG60 <= 32'd0;
    REG62 <= 32'd0;
    REG63 <= 32'd0;
    REG64 <= 32'd0;
    REG67 <= 32'd0;
    REG68 <= 32'd0;
  end else begin
   case(CurrentState)
      P_INIT : oEND <= 1'b0;
      P_END   : begin
                  oEND <= 1'b1;
                  oDATA <= counter;
                  end
      P_STATE9 : xn <= ConstNum8;
      P_STATE11 : yn <= ConstNum10;
      P_STATE13 : xn1 <= ConstNum12;
      P_STATE15 : yn1 <= ConstNum14;
      P_STATE17 : counter <= ConstNum16;
      P_STATE21 : REG21 <= ADD1_RESULT;
      P_STATE22 : counter <= REG21;
      P_STATE24 : REG24 <= RSFT1_RESULT;
      P_STATE26 : REG26 <= RSFT1_RESULT;
      P_STATE27 : REG27 <= MUL1_RESULT;
      P_STATE29 : REG29 <= RSFT1_RESULT;
      P_STATE31 : REG31 <= RSFT1_RESULT;
      P_STATE32 : REG32 <= MUL1_RESULT;
      P_STATE33 : REG33 <= SUB1_RESULT;
      P_STATE34 : REG34 <= ADD1_RESULT;
      P_STATE35 : xn1 <= REG34;
      P_STATE37 : REG37 <= RSFT1_RESULT;
      P_STATE39 : REG39 <= RSFT1_RESULT;
      P_STATE40 : REG40 <= MUL1_RESULT;
      P_STATE41 : yn1 <= REG40;
      P_STATE44 : REG44 <= LSFT1_RESULT;
      P_STATE46 : REG46 <= RSFT1_RESULT;
      P_STATE48 : REG48 <= RSFT1_RESULT;
      P_STATE49 : REG49 <= MUL1_RESULT;
      P_STATE50 : REG50 <= ADD1_RESULT;
      P_STATE51 : yn1 <= REG50;
      P_STATE52 : xn <= xn1;
      P_STATE53 : yn <= yn1;
      P_STATE55 : REG55 <= RSFT1_RESULT;
      P_STATE57 : REG57 <= RSFT1_RESULT;
```

```
    P_STATE58 : REG58 <= MUL1_RESULT;
    P_STATE60 : REG60 <= RSFT1_RESULT;
    P_STATE62 : REG62 <= RSFT1_RESULT;
    P_STATE63 : REG63 <= MUL1_RESULT;
    P_STATE64 : REG64 <= ADD1_RESULT;
    P_STATE67 : REG67 <= LSFT1_RESULT;
    default : oEND <= 1'b0;
  endcase
 end
end
```

＜状態遷移部＞

```
always @(posedge CLK or negedge XRST) begin
  if(!XRST)
    CurrentState <= P_INIT;
  else
      case(CurrentState)
          P_STATE17: CurrentState <= P_STATE19;
          P_INIT     : if(iSTART==1'b1) CurrentState <= P_STATE17;
                       else CurrentState <= CurrentState;
          P_END      : CurrentState <= P_INIT;
          P_STATE19: if(counter<ConstNum18) CurrentState <= P_STATE24;
                       else CurrentState <= P_END;
          P_STATE24: CurrentState <= P_STATE26;
          P_STATE26: CurrentState <= P_STATE27;
          P_STATE27: CurrentState <= P_STATE29;
          P_STATE29: CurrentState <= P_STATE31;
          P_STATE31: CurrentState <= P_STATE32;
          P_STATE32: CurrentState <= P_STATE33;
          P_STATE33: CurrentState <= P_STATE34;
          P_STATE34: CurrentState <= P_STATE35;
          P_STATE35: CurrentState <= P_STATE37;
          P_STATE37: CurrentState <= P_STATE39;
          P_STATE39: CurrentState <= P_STATE40;
          P_STATE40: CurrentState <= P_STATE41;
          P_STATE41: CurrentState <= P_STATE44;
          P_STATE44: CurrentState <= P_STATE46;
          P_STATE46: CurrentState <= P_STATE48;
          P_STATE48: CurrentState <= P_STATE49;
          P_STATE49: CurrentState <= P_STATE50;
          P_STATE50: CurrentState <= P_STATE51;
          P_STATE51: CurrentState <= P_STATE52;
          P_STATE52: CurrentState <= P_STATE53;
          P_STATE53: CurrentState <= P_STATE55;          //
          P_STATE55: CurrentState <= P_STATE57;
          P_STATE57: CurrentState <= P_STATE58;
          P_STATE58: CurrentState <= P_STATE60;
          P_STATE60: CurrentState <= P_STATE62;
          P_STATE62: CurrentState <= P_STATE63;
          P_STATE63: CurrentState <= P_STATE64;
```

```
            P_STATE64: CurrentState <= P_STATE67;
            P_STATE67: CurrentState <= P_STATE68;
            P_STATE68: if(REG64>REG67) CurrentState <= P_END;
                       else CurrentState <= P_STATE21;
            P_STATE21: CurrentState <= P_STATE22;
            P_STATE22: CurrentState <= P_STATE19;
            default : CurrentState <= CurrentState;
        endcase
end
```

付録 d　一状態複数演算のマンデルブロ集合回路（スレーブ部の代入部状態遷移部のみ）

＜代入部＞

```
always @ (posedge CLK or negedge XRST) begin

  if(!XRST) begin
    oEND <= 1'b0;
    DomesticState <= D_END;
    counter <= 32'b0;
    xn <= 32'b0;
    yn <= 32'b0;
    xn1 <= 32'b0;
    yn1 <= 32'b0;
  end
  else if(DomesticState == D_END) begin
    DomesticState <= DomesticState + 1;
  end
  else begin
    case(GrobalState)
      G_INIT : oEND <= 1'b0;
      G_END   : begin
                  oEND <= 1'b1;
                  oDATA <= counter;
                end
      G_STATE0 : begin
          counter <= 0;
      end
      G_STATE1 : begin
        DomesticState <= 8'd1;
      end
      G_STATE2 : begin
      end
      G_STATE3 : begin
        if(DomesticState == 8'd1) begin
          xn1 <= (xn >> 8) * (xn >> 8) - (yn >> 8) * (yn >> 8) + a;
          yn1 <= (xn >> 8) * yn >> 8;
          DomesticState <= 8'd2;
        end
        else if(DomesticState == 8'd2) begin
          yn1 <= ((2 << 16) >> 8) * (yn1 >> 8) + b;
```

```
            xn <= xn1;
            DomesticState <= 8'd3;
          end
          else if(DomesticState == 8'd3) begin
            yn <= yn1;
            DomesticState <= 8'd4;
          end
          else if(DomesticState == 8'd4) begin
            DomesticState <= 8'd5;
          end
          else DomesticState <= D_END;
        end
        G_STATE4 : begin
        end
        G_STATE5 : begin
            counter <= counter + 1;
        end
        default : oEND <= 1'b0;
      endcase
    end
end

＜状態遷移部＞
always @(posedge CLK or negedge XRST) begin
    if(!XRST) begin
      GrobalState <= G_INIT;
    end
    else begin
      case(GrobalState)
        G_INIT : begin
          if(iSTART == 1'b1) begin
            GrobalState <= G_STATE0;
          end
          else GrobalState <= GrobalState;
        end
        G_END : begin
          GrobalState <= G_INIT;
        end
        G_STATE0 : begin
            GrobalState <= G_STATE2;
        end
        G_STATE1 : begin
          GrobalState <= G_END;
        end
        G_STATE2 : begin
          if(counter < 30) begin
            GrobalState <= G_STATE3;
          end
          else GrobalState <= G_STATE1;
        end
        G_STATE3 : begin
          if(DomesticState == D_END) begin
```

```
            GrobalState <= G_STATE4;
          end
          else GrobalState <= GrobalState;
        end
        G_STATE4 : begin
          if( ((xn >> 8) * (xn >> 8) + (yn >> 8) * (yn >> 8)) > (4 << 16) ) begin
            GrobalState <= G_STATE1;
          end
          else GrobalState <= G_STATE5;
        end
        G_STATE5 : begin
            GrobalState <= G_STATE2;
        end
      endcase
    end
end
```