卒業論文

OpenMP ハードウェア動作合成におけるコード生成手法の検討

氏名　　：　西川 諒

学籍番号　：　226007073-7

指導教員　：　山崎 勝弘 教授

提出日　　：　2011 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 ハードウェア動作合成システムにおいて、既存の2つのコードジェネレータによる生成回路と、手書きで記述した HDL による回路との比較、検討を行い、コードジェネレータの改良点の検討を行った。OpenMP ハードウェア動作合成システムは，PC クラスタや SMP クラスタを用いたアルゴリズム検証・評価を行うシミュレーション系，ハードウェアを自動的に合成するハードウェア動作合成系で構成される．シミュレーション系において，OpenMP を用いて対象のアルゴリズムを記述し，クラスタを用いた高速シミュレーションによって，アルゴリズムの正当性や並列化手法の妥当性の評価・検討，及び要求に対する改良を行う．ハードウェア動作合成系では，シミュレーション系から得られたプログラムを OpenMP の構文を利用しながらハードウェアに変換する．

本論文では、本システムを用いて素数判定、マンデルブロ集合の2つの OpenMP プログラムを動作合成し、生成された回路と手書き VerilogHDL 記述による回路と性能を比較した。また比較結果からコードジェネレータの課題を検討し、新しいコードジェネレータの提案を行っている。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討を行う。検討は、記述量、実行クロック数、回路規模、状態遷移数の4つの点で行った。

目次

図目次

表目次

付録

1. はじめに

近年の半導体の微細化技術の発展に伴い、LSI は高性能で多様な機能を実現している。いまや LSI は大規模計算器からパーソナルコンピュータ、さらには安価な玩具に至るまで様々な電子機器に搭載されており、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、LSI の回路規模や複雑さは著しく増加している. しかし一方で、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るため、短期間で高性能な LSI を設計する必要があり、設計規模の増大に生産設計能力の向上が追いつかないという状況が生じている。

生産設計能力を向上させる手段の1つに、動作合成があげられる。動作合成とは、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技術である。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[4]。

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っており、ハードウェアの重要概念である並列処理が記述できないという問題点がある。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、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、依然として熟練した設計者による手動での並列化に頼っており、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[11]。

本研究では、これらの問題を解決するために、並列処理の概念をもつプログラム言語である OpenMP を用いたハードウェア動作合成システムの検証を目的とする。OpenMP は、SMP(Symmetric Multiprocessing)環境における並列プログラミングの標準 API であり、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、DSM(Distributed shared memory)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もある。

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]。

検討方法は、本システムの既存の2つのコードジェネレータにより生成された回路と、手書きによる HDL 記述で生成された回路を比較することで行う。

2008年までに 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]、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[2]、システムとしての体系は一応完成した。しかし当時作成されたコードジェネレータでは、1状態で1演算のみを行うものであり、手書きによる HDL 記述で生成された回路と比較して1クロックあたりの計算量に課題があった。そこで2010年に、1状態で複数の演算を行うコードジェネレータが実装され、1クロックの計算量が改善された[6]。

本研究では、素数判定とマンデルブロ集合に対して、2つのコードジェネレータにより生成された回路と、手書き生成された回路の比較を行い、それぞれの利点、欠点を洗いだす。2つのコードジェネレータでの素数判定とマンデルブロ集合の評価は、2010年度に住井氏が行っているが[6]、昨年度までのマンデルブロ集合のではスレーブ部のみの評価に留まっており、回路全体での評価が行えていないため、全体の評価を行う。また評価は従来の記述量、実行クロック数、回路規模に加え、1クロックでの計算量に深く関係しており、今後の高速化の重要な要素になりうる状態遷移数も測定、評価を行っている。そしてより高性能なコードジェネレータを作成するべく問題点、改良点を提案、考察している。

本論文では、第 2 章において OpenMP ハードウェア動作合成システムの構成および中間表現についての解説とハードウェアモジュールの生成方法を示す。第 3 章では素数判定に対する動作合成の実験結果、第 4 章ではマンデルブロ集合に対する動作合成の実験結果を示す。第 5 章では実験結果からコードジェネレータの改良点を挙げ、新しいコードジェネレータの提案と考察を行う。

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。

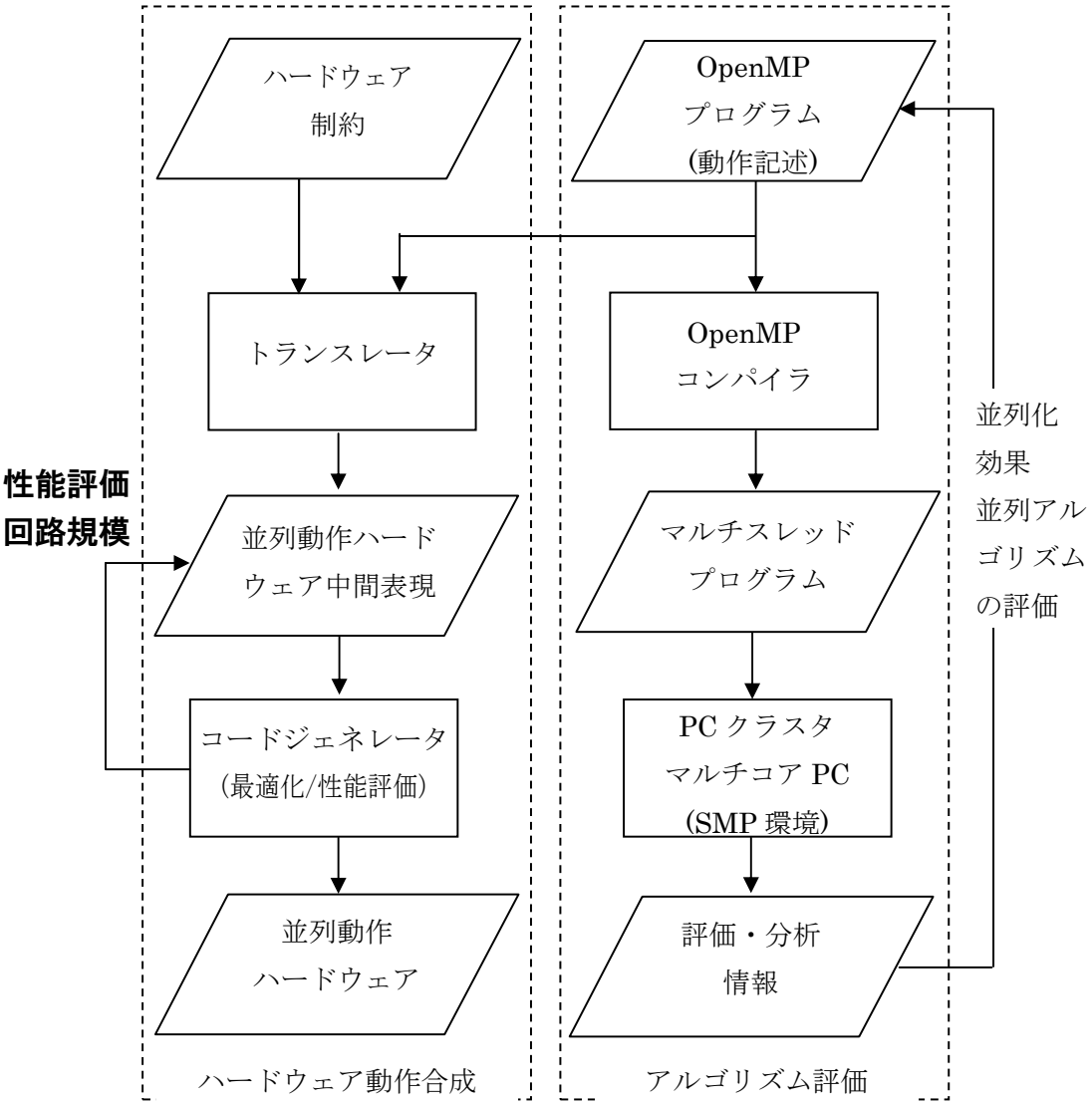


図 1：OpenMP を用いたハードウェア動作合成システム

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、本システムを用いて動作合成を行い、2つのコードジェネレータにより生成された回路と手書きにより生成された回路との比較を行っている[4]。

2.2 中間表現

コードジェネレータに入力される中間表現について説明する。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 プログラムをレジスタ転送方式である RTL の中間表現へと変換する。

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。

OpenMP を用いた 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加算と変数への代入を行っている。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 =(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、シンボル 4 で示される定数 0 を代入することを示している。次に while ループの条件式である#1 へ遷移して、条件式の判定を行う。ここでは無限ループの条件であるため、定数であるシンボルテーブルの 6 を参照し、真であることから#3 の状態へ遷移する。#3 では CFG の 8、9 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[4]。

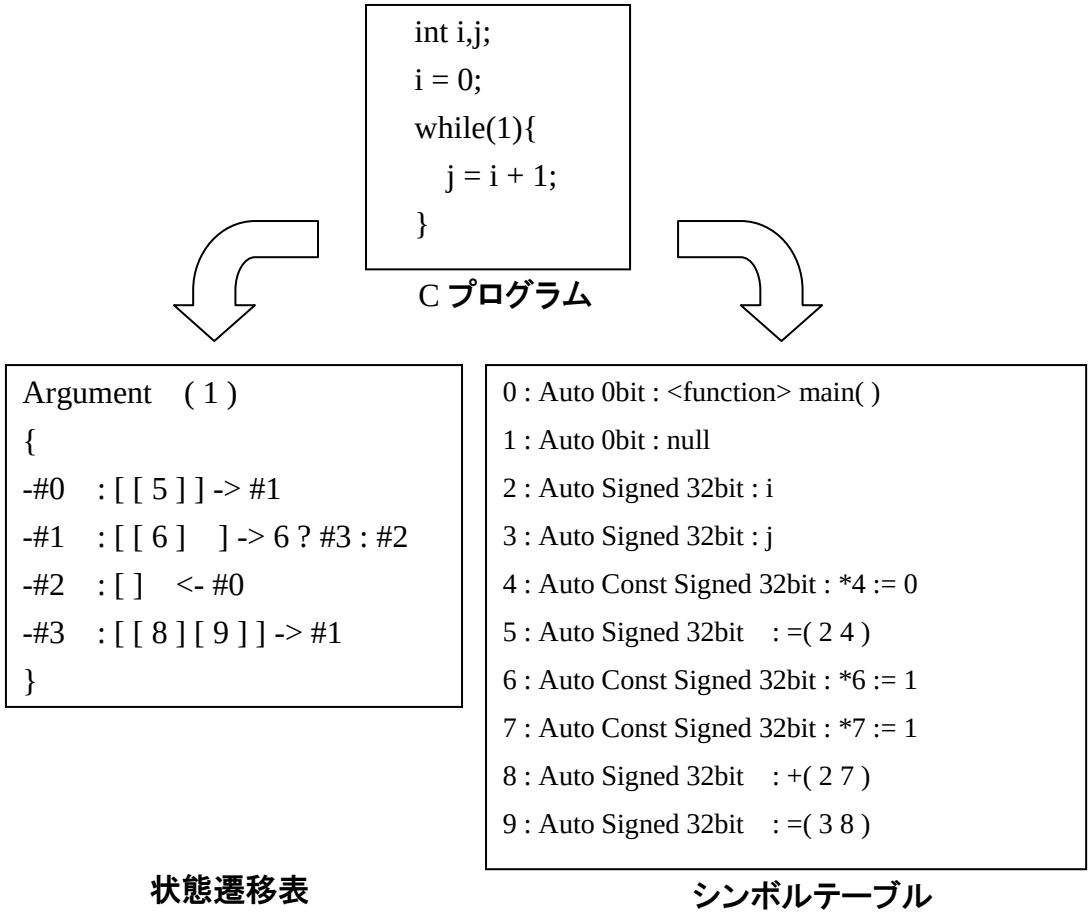


図 2:中間表現の例

2.3 中間表現からのハードウェア生成方法

中間表現からのコードジェネレータによるコード生成手法について説明する。
現在本システムには一状態一演算(S ジェネレータ)と一状態複数演算(M ジェネレータ)の2つのコードジェネレータが実装されており、それぞれにより、生成手法が違う。

2.3.1 一状態一演算ジェネレータ(S ジェネレータ)[2]

一状態一演算コードジェネレータ(S ジェネレータ)について説明する。生成されるモジュールは並列処理を行うスレーブ部であり、別にスレーブ部に値を振り分け、プログラム全体を制御するマスター部を作成しなければならない。モジュールには CurrentState というステートマシンが存在し、ステートマシンにしたがって処理を行う。CurrentState はシンボルテーブルの行数だけ存在し(図2:シンボルテーブルでは 10 つ)、処理別に状態遷移の条件を列挙している。また生成されるモ

ジュールは演算器部、代入部、状態遷移部からなる。

演算器部の生成はシンボルテーブルを参照し、各演算を行う演算器の動作を記述することで行われる。例として図 2 の中間表現をもとに変数 i と j に定数である 1 を加算する演算を、図 3 に示す。論理合成ツールでは基本的に verilog のソースコードで書かれた演算子は、演算子一つに対し演算器一つを論理合成する。そのため、同一モジュール内で同じ演算子の記述を一つだけにすることで、不必要な演算器が生成されないようにしている。また演算器コードでは、演算子の二つのオペランドに対応する部分を wire で宣言し、オペランドに対して assign 文を用いて演算器の動作を記述する。

assign 文の条件式の部分には現在の状態を表すステートマシン（CurrentState）に対し、演算が行われる状態遷移番号を列挙していく。他の演算器についても同様にコードを生成する。

```
wire [31:0] ADD1_RESULT;
wire [31:0] ADD1_A, ADD1_B;
assign ADD1_RESULT = ADD1_A + ADD1_B;
assign ADD1_A  =  (CurrentState==P_STAT8 )? i:
                  (CurrentState==P_STAT9) ? j;
assign ADD1_B  =  (CurrentState==P_STAT8 )?
ConstNum7:
                  (CurrentState==P_STAT9) ?
ConstNum7;
```

図 3:S ジェネレータによる演算器部生成の例

代入部の生成もシンボルテーブルから行われ、変数用のレジスタ、一時格納用のレジスタ、メモリのアドレスオフセット計算用のレジスタ、及びメモリからのデータ入出力の代入を行う。図 4 に S ジェネレータの代入部生成例を示す。図4の例では、図 2 の中間表現をもとに変数 i と定数との足し算の結果を j に代入する場合を示している。代入部はクロックに同期する always 文で記述されており、always 文の最初の if(!XRST)非同期リセットで、演算に利用するレジスタを初期化している。

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. 現在の状態遷移表の場所を表す CurrentState を case 文により参照し、各 CurrentState 内での演算を記述する。また状態に合わせて必要なレジスタへ代入が行われる。

代入式の左辺にはレジスタが、右辺にはレジスタ、wire、定数のいずれかが入る。また外部用ポートのレジスタも含まれるため、外部への制御用変数のレジスタの代入も行われる。外部からのモジュールのスタート信号の入力や外部への終了信号の出力、メモリや arbiter への read,write 信号の代入も同様である。

```
always @(posedge CLK or negedge XRSI)
begin
 if(!XRST)begin
   i <= 32'd0;
   ～省略～
 end else begin
   case(CurrentState)
     P_INIT    :  oEND <= 1'b0;
     P_STATE8  :  REG8 <= ADD1_RESULT;
     P_STATE9  :  j    <= REG8;
   ～省略～
   endcase
 end
end
```

図 4:S ジェネレータによる代入部の例(抜粋)

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記述することで生成する。図 2 の中間表現をもとに生成された状態遷移の例を図 5 に、状態遷移図を図6に示す。状態遷移部もクロックに同期する always 文で記述されており、always 文の最初の if(!XRST)は非同期リセットであり、演算に利用するレジスタを初期化している。例では、P_INIT で示される初期状態から図 2 中間表現にて#0 で示される最初の演算である[5]を行うため、P_STATE5 に遷移する。P_STATE5 では，次に示される状態分岐判定を行う[6]に移動するため P_STATE6 に遷移する。P_STATE6 では条件分岐を行うが、無限ループであるため、条件式が定数である ConstNum6 になっている。条件式が真であるため、#3 で示される演算である[8]と次の[9]の状態へ遷移した後に#1 のループの最初に遷移する。

```
always @(posedge CLK or negedge XRSI) begin
 if(!XRST) begin
   CurrentState <= P_INIT;
 end else begin
   case(CurrentState)
     P_INIT    :  if(iSTART==1'b1) CurrentState <= P_STATE5;
                  else CurrentState <= CurrentState;
     P_STATE5  :  CurrentState <= P_STATE6;
     P_STATE6  :  if( ConstNum6 )   CurrentState <= P_STATE8;
                  else              CurrentState <= P_END;
     P_STATE8  :  CurrentState <= P_STATE9;
     P_STATE9  :  CurrentState <= P_STATE6;
     P_END     :  CurrentState <= P_INIT;
   endcase
 end
end
```

図 5:S ジェネレータによる状態遷移部生成の例

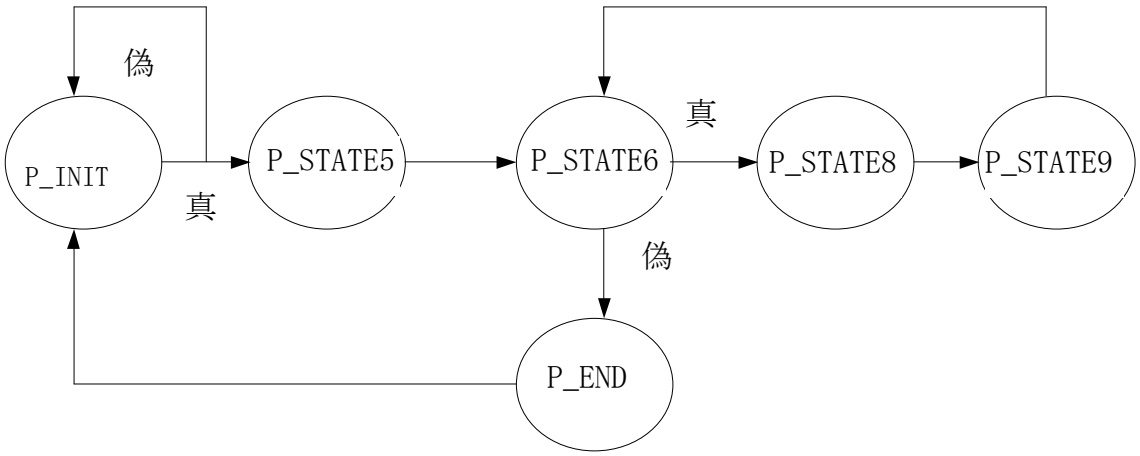


図 6:S ジェネレータによる状態遷移図

2.3.2 一状態複数演算(M ジェネレータ)[6]

一状態複数演算コードジェネレータ(M ジェネレータ)について説明する。生成されるモジュールは、各 slave に値を振り分け、プログラム全体を制御する master と並列時に master から値を受けて演算を行い、演算の終了を master に知らせる slave の 2 つがある。各モジュールには GlobalState、DomesticState という 2 つのステートマシンが存在し、それらにしたがって、処理を行う。GlobalState は状態遷移表の行数だけ存在し(図2:状態遷移表では#0-#3 の 4 つ)、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。また各モジュールは代入部、状態遷移部の 2 つのパートにより構成される。

代入部は状態遷移表とシンボルテーブルを参照して、実際の演算部分を記述することで生成される。図 2 の中間表現をもとに生成された代入部の例を図 7 に示す。代入部はクロックに同期する always 文で記述されており、always 文の最初の if(!XRST)は非同期リセットであり、演算に利用するレジスタを初期化している。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。現在の状態遷移表の場所を表す GlobalState を case 文により参照し、各 GlobalState 内での演算を記述する。また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調べる。図7の例の場合は変数 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。

```
always @ (posedge CLK or negedge XRST) begin
  if(!XRST) begin
    oEND <= 1'b0;
    DomesticState <= D_END;
～省略～
  end
else begin
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
```

図 7:M ジェネレータによる代入部の例(抜粋)

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記述することで生成する。図 2 の中間表現をもとに生成された状態遷移の例を図 8 に、状態遷移図を図 9 に示す。状態繊維部もクロックに同期する always 文で記述されており、always 文の最初の if(!XRST)は非同期リセットであり、演算に利用するレジスタを初期化している。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。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 8 の G_STATE2のように DomesticState が記述されているとき、代入部にした

がって処理が行われているので、GlobalState 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_STATE2 の状態を維持する。

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

図 8:M ジェネレータによる状態遷移部の例

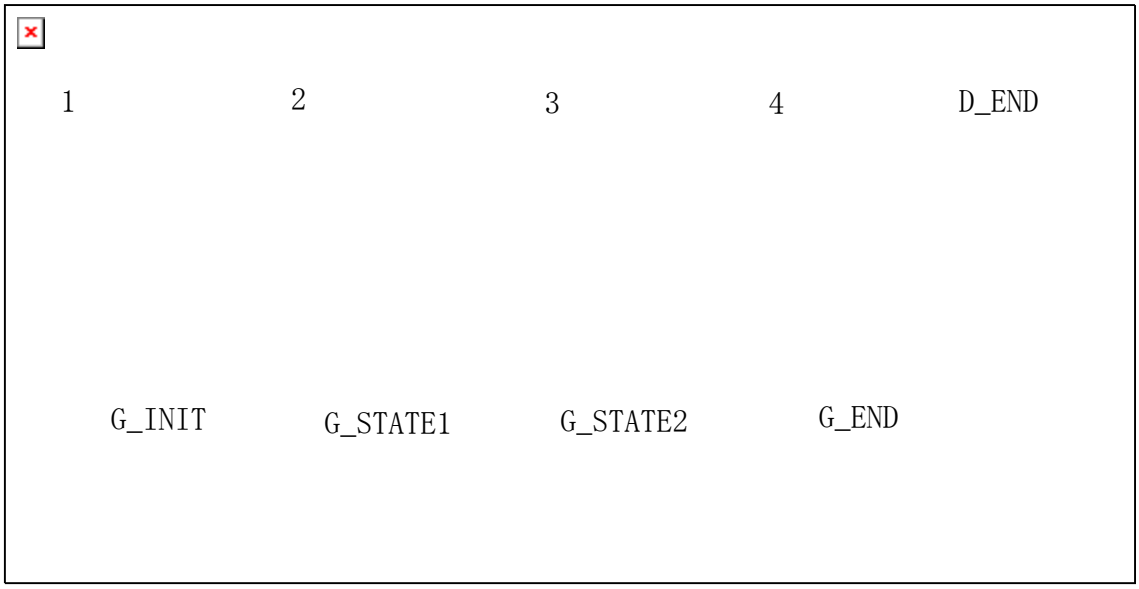


図 9:M ジェネレータによる状態遷移図

3. 素数判定に対するハードウェア動作合成

3.1　素数判定のアルゴリズム

本研究で扱う素数判定は、2つのコードジェネレータは剰余に対応していないため、減算を使用したアルゴリズムになっている。まず素数判定を行いたい値に対して、2から順に、1つの数字で繰り返し減算を行う。引けなくなるまで減算を繰り返した後、値が0になっていれば、減算を行った数字で割れたとし、oDATA に1を入れ、終了する。そして判定を行いたい値-1 までの減算を行い、それでも oDATA が0であれば素数とする。アルゴリズムのフローチャートを図 10 に示す。

素数判定プログラムでは、素数判定を行う値が素数でないと判断した時点でプログラムが終了してしまうので、プログラムが処理を最後まで適切に動いているか確認するために 100003 という素数を判定値に使用した。並列手法としては、i によるループ箇所を分割することにした。

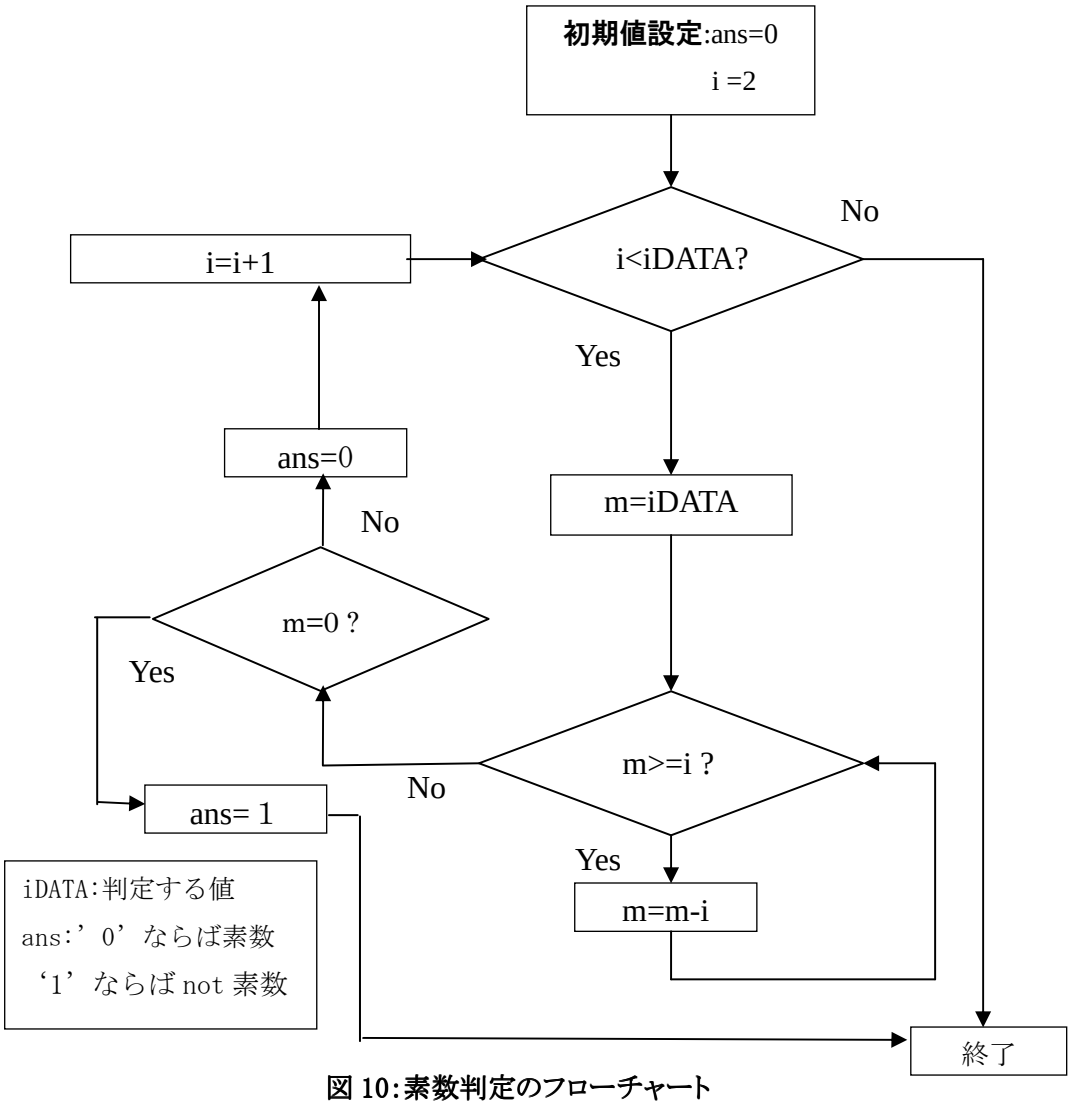


図 10:素数判定のフローチャート

3.2　手書きとシステムによる HDL 記述の生成

図 8 のフローチャートをもとに作成した OpenMP によるプログラムを図 11 に示す。
基本的にはアルゴリズム通りで判定する値を iDATA、素数かどうか判別する値を ans と置いている。
なお、判定する値 iDATA の値制御は HDL 上で行うため、OpenMP プログラムからは省いている。

```
void main(void){
int    iDATA,i,m,ans;
#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) ans=1;
       }
}
```

図 11:素数判定の OpenMP プログラム

次に図 11 のプログラムをもとに、トランスレータの生成した中間表現を図 12 に示す。中間表現は状態遷移表とシンボルテーブルの2つから構成されており、それぞれコードジェネレータにて、状態遷移部作成時に参照される。状態遷移表は22行、シンボルテーブルは11行であった。

```
----SemanticsAnalyze----
function 0 : main
0 : Auto 0bit : <function> main( )
1 : Auto 0bit : null
2 : Auto Const Signed 32bit : *2 := 100003
3 : Auto Signed 32bit : i
4 : Auto Signed 32bit : m
5 : Auto Signed 32bit : oDATA
6 : Auto Const Signed 32bit : *6 := 0
7 : Auto Signed 32bit   : =( 3 6 )
～～省略～～
20 : Auto Const Signed 32bit : *20 := 1
21 : Auto Signed 32bit   : =( 5 20 )
Argument （ 1 ）
{
--#0   : { /0 } -> #1
--#1   : [ ]  <- #0
}
/0 Parallel FOR (2) ( P[ 3 5 4 ]  )
-0:#0   : [ [ 7 ] ] -> #2
-0:#1   : [ ]  <- #0
-0:#2   : [ [ 8 ] ] -> 8 ? #3 : #1
～～省略～～
-0:#8   : [ [ 16 ] [ 17 ] ] -> #7
-0:#9   : [ [ 9 ] ] -> #2
-0:#10   : [ [ 21 ] ] -> #9
```

図 12:素数判定の中間表現(抜粋)

次に図 12 の中間表現をもとに手書き生成した回路、S ジェネレータの生成した回路、M ジェネレータの生成した回路と3つの回路の違いを述べる。なお、それぞれの生成した回路記述の全文は論文の最後に付録として添付するので参考されたい。

コードジェネレータによる生成回路の Verilog 記述は演算部(S ジェネレータのみ)、代入部、状態遷移部の3部からなる。最も大きな差は演算部が S ジェネレータにのみ実装されていることと、状態遷移の生成方部にある。手書きによる生成回路と、両ジェネレータの生成した回路の状態遷移数を表5に示す。また手書きによる生成回路の状態遷移図を図13に、また各ジェネレータによる生成回路の状態遷移図を図14に示す。

表 1:各生成回路の状態遷移数

	手書き	S ジェネレータ	M ジェネレータ
状態遷移数	9	23	13

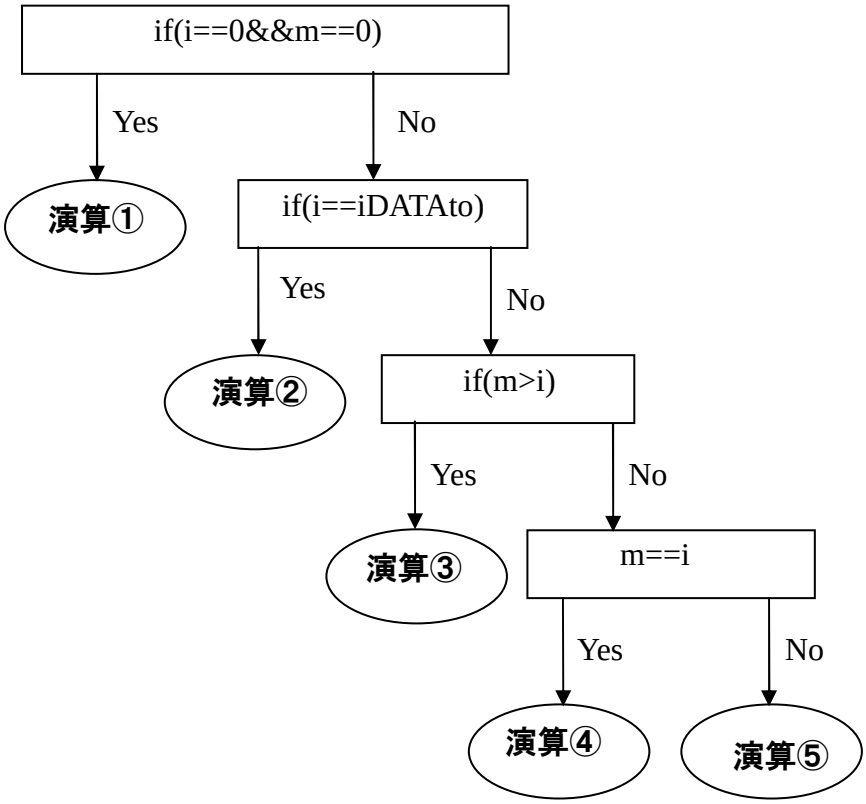
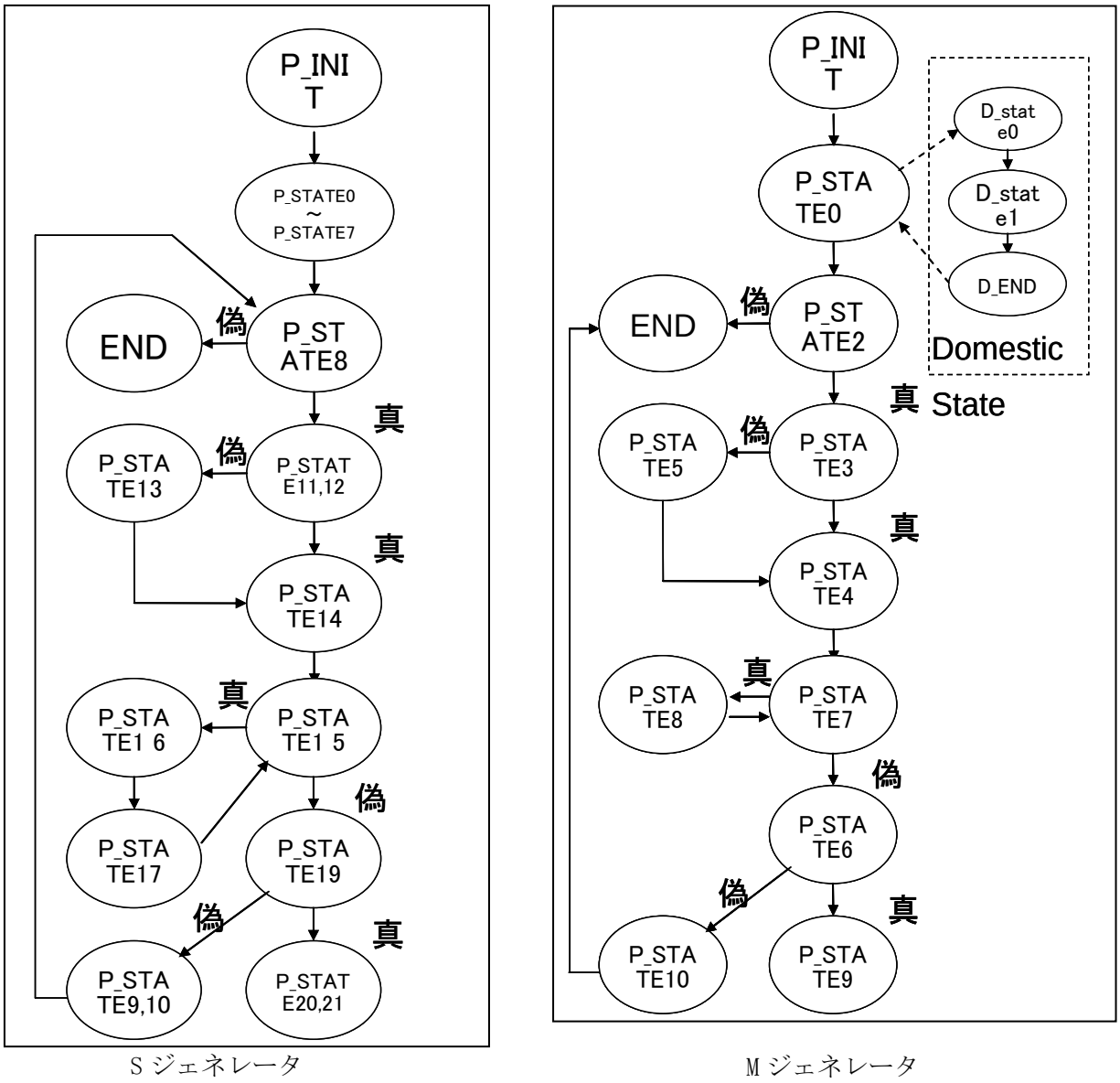


図 13:素数判定の手書き生成回路の状態遷移図



S ジェネレータ　　M ジェネレータ

図 14: 素数判定の両ジェネレータの生成した回路の状態遷移図

手書きよる回路生成では、コードジェネレータの様に代入部、状態遷移部等に分かれておらず、ステートマシンも存在しない。このため図13のようにシンプルな状態遷移になり、状態遷移数も最も少ない。S ジェネレータと M ジェネレータの一番の違いはステートマシンにある。S ジェネレータでは一状態で一演算しか行わないため、全ての演算が一状態ずつにあてがわれており、状態遷移数も多い。またステートマシンも CurrentState 一種類である。M ジェネレータでは一状態で複数の演算を行えるため、複数の演算が一状態ずつにあてがわれている。そのため状態遷移数も S ジェネレータより少なくなっている。またステートマシンも全体を制御する GlobalState と、データに依存関係があり、同時に複数演算を行えない際に演算を制御する DomesticState の 2 種類が実装されている。

3.3　手書きとシステムによる生成回路の比較

SMP 環境、および動作合成システムの実験環境を表2に示す。

表 2:実験環境

回路シミュレーション	PC 環境	Intel Core2 duo 3.66GHz Memory 4GB
	シミュレーションツール	ModelSim SE 6.3c
アルゴリズム評価	SMP 環境	Quad Xeon 3.0Ghz,Memory 4GB
	OpenMP コンパイラ	Intel コンパイラ 9.1.038
ハードウェア動作合成	論理合成ツール	Xilinx ISE 13

高性能計算研究室には Raptor、Diplo、Nycto の現在3種類の PC クラスタがある。今回は Nycto と呼ばれるクラスタを使用した。Nycto は CPU を8つ搭載したボード2つで構成されており、逐次のプログラムを最大16つのスレーブに分けて実行できる。今回は並列ハードウェアの最大ノードが4であるため、最大4スレーブで実行した。図 15 に PC クラスタ Nycto の構成図を示す。

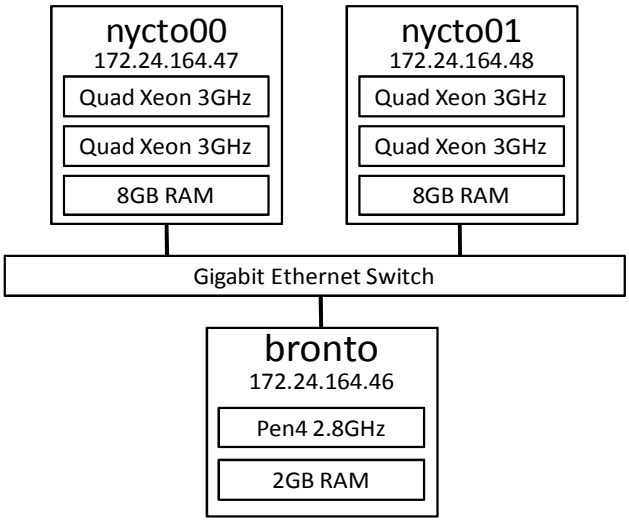


図 15:PC クラスタ Nycto の構成図

素数の判定には、プログラムが処理を最後まで適切に動いているか確認するために、100003 という素数を用いた。素数判定の OpenMP プログラムを SMP クラスタで実行した場合の時間と速度向上比を表3に示す。

表 3:素数判定の SMP クラスタでの実行速度

ノード数	実行時間[ms]	速度向上比
1	18.9	1.0
2	11.2	1.69
4	8.1	2.33

ノード数が 1 の場合は逐次実行を示している。ノード数が増えるに従い、実行時間が短くなることから、速度向上が得られている。このことから、素数判定は並列化に適していることがわかる。

次に記述量の比較を行う。余分な記述は回路規模の増大を招く場合をあるため、重要な要素である。また動作合成では生成後、手直しが必要な場合が多く、手直しのしやすさという点でも、重要な要素でもある。なお記述は for 文等ループ文を除いて、1行1文で行う。手書きと各ジェネレータの記述量の表を表4に示す。

表 4:各生成回路の記述量

	記述量(マスター部)	記述量(スレーブ部)
手書き	37	46
S ジェネレータ	36(手書き)	135
M ジェネレータ	109	149

記述量は手書きが一番少なく、M ジェネレータが一番多いという結果になった。これはシステムには手書き記述にはない、余分な記述があることが原因としてあげられる。なお S ジェネレータではマスター部は自動生成されないため、手書きで記述を行った。このため手書きと S ジェネレータではマスター部の記述量にほぼ差がなくなっている。次に回路規模と実行クロック数の表を表5に示す。

表 5:回路規模と実行クロック数

スレッド数		1	2	4
回路面積(slice 数)	手書き	113	225(1.99)	480(4.25)
	S ジェネレータ	139	288(2.07)	591(4.25)
	M ジェネレータ	116	228(1.97)	453(3.9)
実行クロック数	手書き	1066765	980102	899002
	S ジェネレータ	3350322	3026000	2760018
	M ジェネレータ	2780052	2499882	2450012

回路規模は逐次処理ハードウェアでは手書きによる生成回路が一番小さくなった。しかしノード4時の場合、M ジェネレータが一番コンパクトな回路になった。このことより、手書きよりも M ジェネレータのほうが、ノード数を増やした時の回路規模の増加が少ないことが考えられる。また S ジェネレータでは値を一時保存する演算値保持レジスタがある分、回路規模が大きくなっていると考えられる。

実行クロック数では手書きが圧倒的に少ないクロック数で動作することがわかった。これは1クロック数あたりの計算量に差があるためと考えられる。またノード数に対して、実行クロック数があまり減少しなかったのは、各スレーブの処理量の違いが原因だと考えられる。今回の素数判定では、入力値を 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。今回素数判定を行う値は 100003 で、もっとも低い i の値 2 の場合は 100003/2=50002 回減算を行う。しかし i=1000 のときは 100003/1000=101 回の減算でループが終了してしまい、i の値が低いほど遅延が生じてしまい、ノード数に対して実行クロック数があまり減少しなかった。

4. マンデルブロ集合に対するハードウェア動作合成

4.1　マンデルブロ集合のアルゴリズム

マンデルブロ集合とは $z_{n+1} = z_n^2 + c$、$z_0 = 0$ の式で定義される複素数列が　“n → ∞”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。アルゴリズムのフローチャートを図 16、図 17 に示す。

まずマスター部で初期値の計算、並列処理の分担を行い、それぞれの値をスレーブ部に入力する。そしてスレーブ部では図 17 のフローチャートにしたがって処理が行われる。計算が終了すると、スレーブ部はマスター部に値を返し、終了する。スレーブ部の制御や分割の範囲は oEND、SL_START、SL_END で行っている。

並列手法として i によるループ箇所を分割することにした。

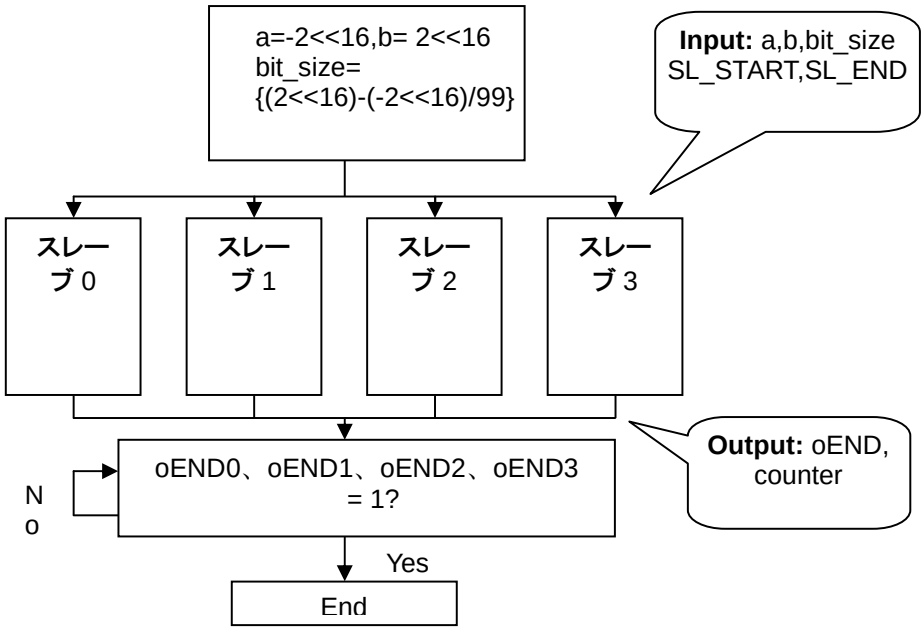


図 16:マンデルブロ集合:マスター部のフローチャート

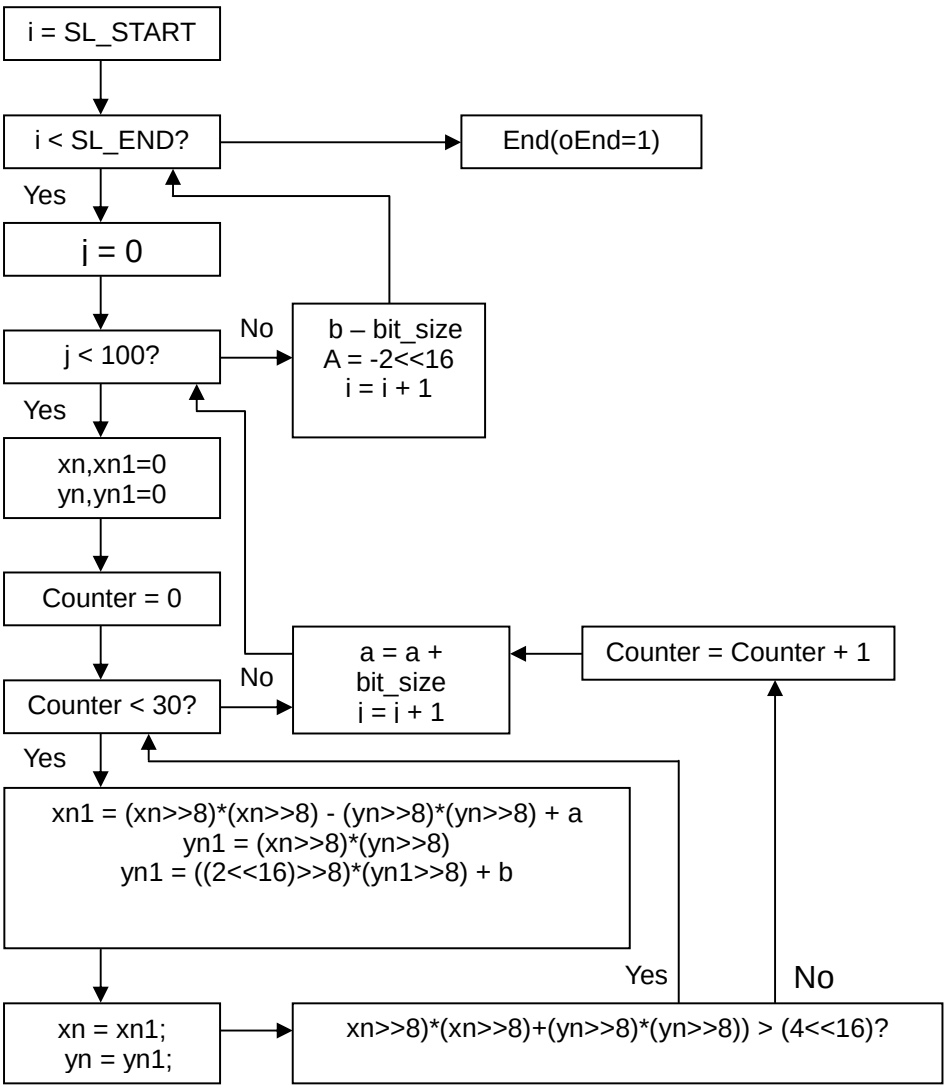


図 17:マンデルブロ集合:スレーブ部のフローチャート

4.2　手書きとシステムによる HDL 記述の生成

図 16、図 17 のアルゴリズムをもとに作成した OpenMP によるプログラムを図 18 に示す。

```
int main()
{
        int i, j;
        int a, b;
        int bit_size;
        int xn, yn,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 18:マンデルブロ集合の OpenMP プログラム

次に図 18 の OpenMP プログラムをもとにトランスレータの生成した中間表現のシンボルテーブルを図 19 に、中間表現を図 20 に示す。3.2 でも述べた通り、中間表現は状態遷移表とシンボルテーブルの2つから構成され、状態遷移部作成時に、それぞれコードジェネレータが参照する。状態遷移表は 115 行、シンボルテーブルは15 行であった。

```
Argument    ( )
{
--#0    : [ [ 12 ] [ 14 ] [ 15 ] [ 18 ] [ 19 ] [ 22 ] [ 24 ] [ 26 ] [ 27 ] [ 29 ] [ 30 ] ] -> #1
--#1    : { /0 } -> #2
--#2    : [ ]    <- #0
}
/0 Parallel FOR (2) ( )
-0:#0    : [ [ 32 ] ] -> #2
-0:#1    : [ ]    <- #0
-0:#2    : [ [ 34 ] ] -> 34 ? #3 : #1
-0:#3    : [ [ 39 ] ] -> #5
-0:#4    : [ [ 108 ] [ 109 ] [ 111 ] [ 113 ] [ 114 ] ] -> #14
-0:#5    : [ [ 41 ] ] -> 41 ? #6 : #4
-0:#6    : [ [ 46 ] [ 48 ] [ 50 ] [ 52 ] ] -> #7
-0:#7    : [ [ 54 ] ] -> #9
-0:#8    : [ [ 106 ] [ 107 ] ] -> #13
-0:#9    : [ [ 56 ] ] -> 56 ? #10 : #8
-0:#10    : [ [ 61 ] [ 63 ] [ 64 ] [ 66 ] [ 68 ] [ 69 ] [ 70 ] [ 71 ] [ 72 ] [ 74 ] [ 76 ] [ 77 ] [ 78 ]
[ 81 ] [ 83 ] [ 85 ] [ 86 ] [ 87 ] [ 88 ] [ 89 ] [ 90 ] ] -> #11
-0:#11    : [ [ 92 ] [ 94 ] [ 95 ] [ 97 ] [ 99 ] [ 100 ] [ 101 ] [ 104 ] [ 105 ] ] -> 105 ? #8 : #12
-0:#12    : [ [ 58 ] [ 59 ] ] -> #9
-0:#13    : [ [ 43 ] [ 44 ] ] -> #5
-0:#14    : [ [ 36 ] [ 37 ] ] -> #2
```

図 19:マンデルブロ集合のシンボルテーブル

```
0 : Auto Signed 32bit : <function> main( )
1 : Auto Signed 32bit : i
2 : Auto Signed 32bit : j
3 : Auto Signed 32bit : a
4 : Auto Signed 32bit : b
5 : Auto Signed 32bit : bit_size
6 : Auto Signed 32bit : xn
7 : Auto Signed 32bit : yn
8 : Auto Signed 32bit : xn1
9 : Auto Signed 32bit : yn1
10 : Auto Signed 32bit : counter
11 : Auto Const Signed 32bit : *11 := 2
12 : Auto Signed 32bit  : -( 11 )
13 : Auto Const Signed 32bit : *13 := 16
14 : Auto Signed 32bit  : <<( 12 13 )
15 : Auto Signed 32bit  : =( 3 14 )
16 : Auto Const Signed 32bit : *16 := 2
17 : Auto Const Signed 32bit : *17 := 16
18 : Auto Signed 32bit  : <<( 16 17 )
19 : Auto Signed 32bit  : =( 4 18 )
20 : Auto Const Signed 32bit : *20 := 2
21 : Auto Const Signed 32bit : *21 := 16
22 : Auto Signed 32bit  : <<( 20 21 )
23 : Auto Const Signed 32bit : *23 := 2
24 : Auto Signed 32bit  : -( 23 )
25 : Auto Const Signed 32bit : *25 := 16
26 : Auto Signed 32bit  : <<( 24 25 )
27 : Auto Signed 32bit  : -( 22 26 )
28 : Auto Const Signed 32bit : *28 := 99
29 : Auto Signed 32bit  : /( 27 28 )
30 : Auto Signed 32bit  : =( 5 29 )
～～～省略～～～
91 : Auto Const Signed 32bit : *91 := 8
92 : Auto Signed 32bit  : >>( 6 91 )
93 : Auto Const Signed 32bit : *93 := 8
94 : Auto Signed 32bit  : >>( 6 93 )
95 : Auto Signed 32bit  : *( 92 94 )
96 : Auto Const Signed 32bit : *96 := 8
97 : Auto Signed 32bit  : >>( 7 96 )
98 : Auto Const Signed 32bit : *98 := 8
99 : Auto Signed 32bit  : >>( 7 98 )
100 : Auto Signed 32bit  : *( 97 99 )
101 : Auto Signed 32bit  : +( 95 100 )
102 : Auto Const Signed 32bit : *102 := 4
103 : Auto Const Signed 32bit : *103 := 16
104 : Auto Signed 32bit  : <<( 102 103 )
105 : Auto Signed 32bit  : >( 101 104 )
106 : Auto Signed 32bit  : +( 3 5 )
107 : Auto Signed 32bit  : =( 3 106 )
108 : Auto Signed 32bit  : -( 4 5 )
109 : Auto Signed 32bit  : =( 4 108 )
110 : Auto Const Signed 32bit : *110 := 2
111 : Auto Signed 32bit  : -( 110 )
112 : Auto Const Signed 32bit : *112 := 16
113 : Auto Signed 32bit  : <<( 111 112 )
114 : Auto Signed 32bit  : =( 3 113 )
```

図 20:マンデルブロ集合の中間表現(抜粋)

次に図 19、図 20 の中間表現をもとに手書き生成した回路、S ジェネレータの生成した回路、M ジェネレータの生成した回路と3つの回路の違いを述べる。3.2 と同様に、それぞれの生成した回路記述の全文は論文の最後に付録として添付するので参考されたい。

マンデルブロ集合でも、コードジェネレータによる生成回路の Verilog 記述は演算部(S ジェネレータのみ)、代入部、状態遷移部の3部からなる。そして最も大きな差は演算部が S ジェネレータにのみ実装されていることと、状態遷移の生成方部にある。手書きによる生成回路と、両ジェネレータの生成した回路の状態遷移数を表6に示す。また手書きによる生成回路の状態遷移図を図 21 に、また各ジェネレータによる生成回路の状態遷移図を図 22 に示す。

表 6:マンデルブロ集合の各生成回路の状態遷移数

	手書き	S ジェネレータ	M ジェネレータ
状態遷移数	4	114	20

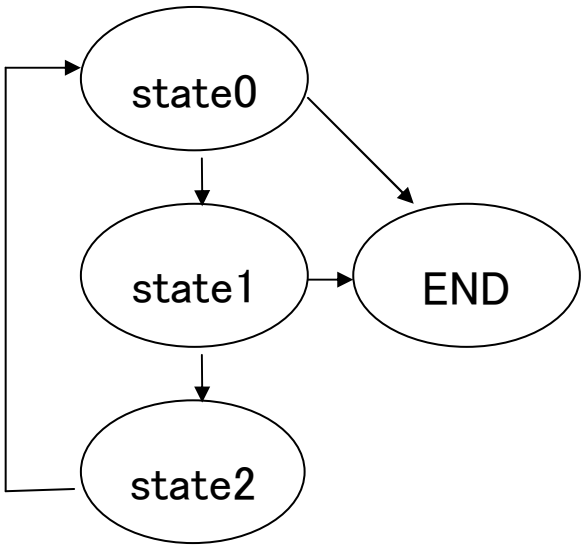
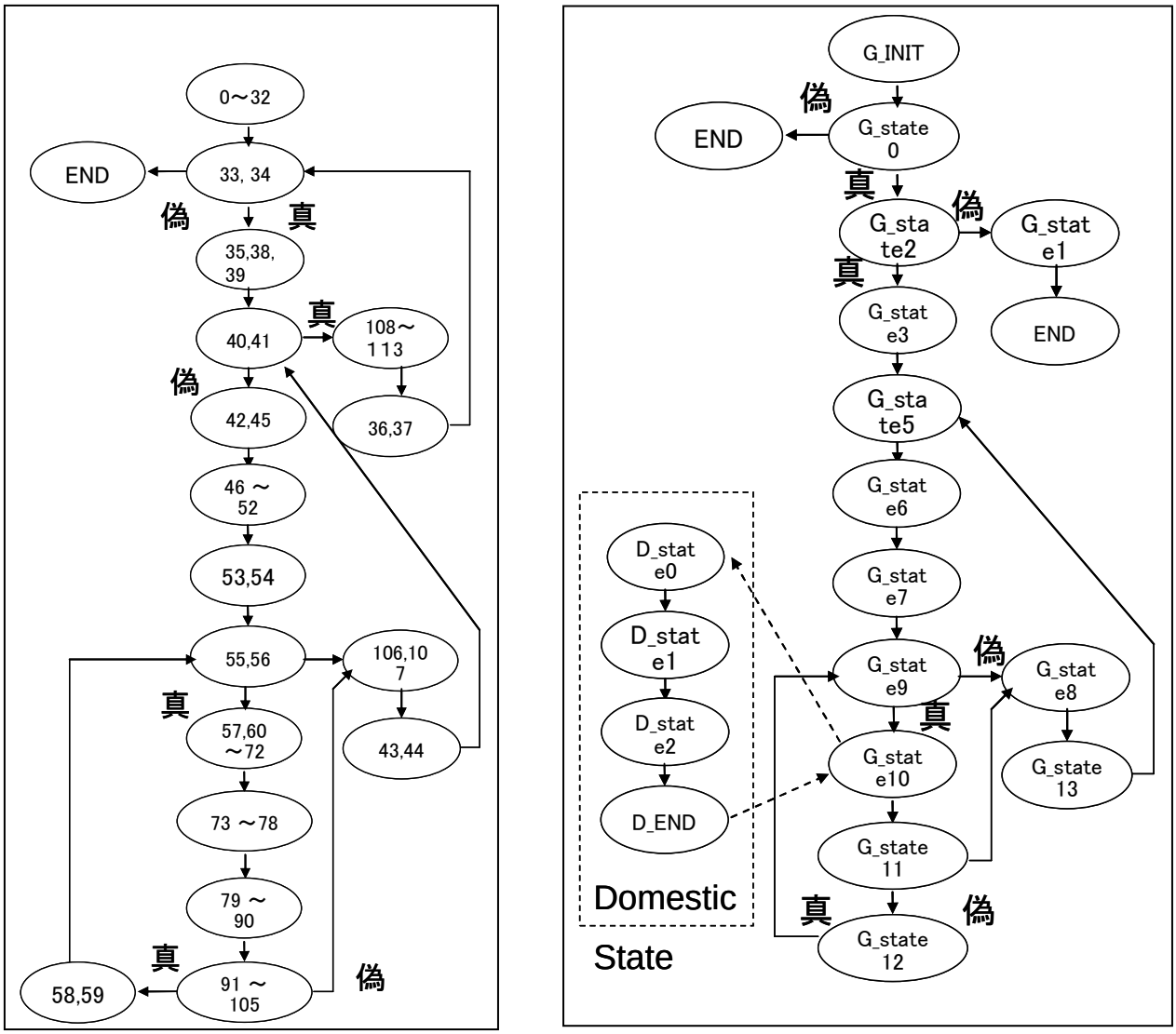


図 21:マンデルブロ集合の手書き生成回路の状態遷移図



S ジェネレータ　　M ジェネレータ

図 22: マンデルブロ集合の両ジェネレータの生成した回路の状態遷移図

4.3　手書きとシステムによる生成回路の比較

実験環境は3章に記した通りである。またマンデルブロ集合に用いた画像は 100×100 である。

表 7:マンデルブロ集合の SMP クラスタでの実行速度

ノード数	実行時間[ms]	速度向上比
1	3.824	1
2	2.766	1.38
4	2.048	1.87

ノード数が 1 の場合は逐次実行を示している。ノード数が増えるに従い、実行時間が短くなることから、速度向上が得られている。このことから、マンデルブロ集合も並列化に適していることがわかる。

次に記述量の比較を行う。3.3 でも示した通り、記述量は回路規模や手直しのしやすさという点において、重要な要素である。手書きと各ジェネレータの記述量の表を表8に示す。

表 8:マンデルブロ集合の各生成回路の記述量

	記述量(マスター部)	記述量(スレーブ部)
手書き	40	92
S ジェネレータ	40(手書き)	536
M ジェネレータ	106	212

記述量は手書きによる記述が両ジェネレータより少なく、優秀であるという結果になった。やはりシステムには手書き記述にはない、余分な記述があることが原因としてあげられる。4.2 で挙げた通り、状態遷移数の差が原因の一つにあげられる。両ジェネレータとも、状態遷移数に依存したステートマシンがパラメータとして実装されており、状態遷移数の増加に比例してパラメータも増加する。今回は手書きの状態遷移数4に対して、S ジェネレータが114、M ジェネレータが23と数倍～数百倍の違いがある。

次に回路規模と実行クロック数の比較を行う。表9に各生成回路の回路規模と実行クロック数を示す。

表 9:マンデルブロ集合の回路規模と実行クロック数

スレッド数		1	2	4
回路面積(slice 数)	手書き	858	1940	4629
	S ジェネレータ	813	1809	4247
	M ジェネレータ	753	1786	4277
実行クロック数	手書き	313662	156850	78490
	S ジェネレータ	411141	205600	102858
	M ジェネレータ	172713	71207	34970

回路規模は両ジェネレータにあまり差はなく、手書きによる生成回路が一番小さくなった。これはSジェネレータでは1クロックで1つの演算しか行わないため、演算器の数が少なくすんでいるためである。またSジェネレータを元に作成されたMジェネレータでは、複数の演算器を実装する変わりに演算値保持レジスタを使用してないため、Sジェネレータと同等の回路規模になっていると考えられる。

実行クロック数ではMジェネレータが一番短く、次に手書き、Sジェネレータという順になった。これは1クロック数あたりの計算量で、一状態複数演算を実装しているMジェネレータが手書きを上回ったためと考えられる。さらにあとにあげる通り、手書きとMジェネレータシステムには状態遷移数に差があるため、Mジェネレータの生成回路はさらに高速化が見込めると考えられる。一状態で一演算しか行えないSジェネレータでは、クロック数が一番多くかかってしまっている。

現時点でマンデルブロ集合においてMジェネレータがクロック数、回路規模において優秀であるが、状態遷移数を減らすことで1クロックあたりの計算量が増え、より高性能な回路を生成できると考えられる。

5. コード生成手法の検討

5.1　新しいコードジェネレータの提案

この章では既存の2つコードジェネレータの課題を延べ、新しいコードジェネレータの提案を行う。提案は回路の重要な要素である、実行速度と回路規模の2点から行う。

5.1.1 実行速度

実行速度において素数判定では手書き＞M ジェネレータ＞S ジェネレータ、マンデルブロ集合では M ジェネレータ＞手書き＞S ジェネレータの順に速度が速かった。これは1クロックの計算量の差によるものであると考えられる。特に S ジェネレータでは 1 クロックに1演算しか行わないため、素数判定、マンデルブロ集合両方の対象において一番実行クロック数が遅いという結果になった。

1クロックの計算量をあげるためには、状態遷移数を減らし演算の量を増やす必要がある。また演算の量を増やしても、演算器の数が足りなければ遅延が起きてしまい、かえって実行速度が遅くなってしまう。そのため、1 クロックの最大演算数に合わせた演算器の生成も求められる。図 23 に S ジェネレータをもとにした演算器部の改良例を示す。

```
wire signed [31:0]ADD1_RESULT;
wire signed [31:0]ADD1_A,ADD1_B;
assign ADD1_RESULT = ADD1_A +
ADD1_B;
assign ADD1_A =
(CurrentState==P_STATE2) ? counter :
(CurrentState==P_STATE3) ? i :
(CurrentState==P_STATE4) ? j :
assign ADD1_B =
(CurrentState==P_STATE2) ? x :
(CurrentState==P_STATE3) ? a :
(CurrentState==P_STATE4) ? b :
```

改良前

```
wire signed [31:0]ADD1_RESULT;
wire signed [31:0]ADD1_A,ADD1_B;
assign ADD1_RESULT = ADD1_A + ADD1_B;
assign ADD1_A =(CurrentState==P_STATE2) ? counter :
assign ADD1_B =(CurrentState==P_STATE2) ? x :
wire signed [31:0]ADD2_RESULT;
wire signed [31:0]ADD2_A,ADD2_B;
assign ADD1_RESULT = ADD2_A + ADD2_B;
assign ADD2_A =(CurrentState==P_STATE2) ? i :
assign ADD2_B =(CurrentState==P_STATE2) ? a :
wire signed [31:0]ADD3_RESULT;
wire signed [31:0]ADD3_A,ADD3_B;
assign ADD3_RESULT = ADD3_A + ADD3_B;
assign ADD3_A =(CurrentState==P_STATE2) ? j :
assign ADD3_B =(CurrentState==P_STATE2) ? b :
```

改良後

図 23:演算器部の改良例

改良例では counter+x、i+a、j+b の加算を行う加算器を生成している。改良前では加算器１つで３クロックかけて演算を行っているが、改良後は３つの加算器を生成し、１クロックで演算を行っている。上記の加算において、改良後は改良前の三倍の速度向上が見込め

る。なおMジェネレータでは演算器部は実装されておらず、配線やレジスタ数、演算奇数において工夫しにくいジェネレータになっている。

実行速度最速のコードジェネレータを生成するのであれば、1クロックの計算量を増やすべく状態遷移数を減らし、演算の量を増やす必要がある。また演算器の不足から、回路が遅延を起こさないよう、1クロックの最大演算数に合わせた演算器の生成も求められる。

5.1.2 回路規模

回路規模において、システムの生成する回路が手書きによる生成回路に劣る理由に、手書きにはない余分な記述がある点と、演算値保持レジスタ等、不必要な回路が生成されている点の2点があげられる。本システムは多様な生成回路に対応しなければならず、それにより生成を行いたい回路には不必要な回路を生成してしまう事がある。条件分岐をコードジェネレータに実装することで、生成を行う回路に不必要な回路の記述を行わないよう改良する必要がある。

また、マンデルブロ集合等、回路規模の大きな回路を生成する場合にシステムの方が手書きによる生成回路よりコンパクトな回路を生成する場合がある。Sジェネレータの場合は1状態に一演算のみを行うため、各演算器の個数が最小限に抑えられているためである。またSジェネレータをもとに改良されたMジェネレータでは、演算器は複数生成されるがSジェネレータでは生成されている演算値保持レジスタが生成されておらず、Sジェネレータと同等の回路規模に抑えることが出来ている。

回路規模最小のコードジェネレータを生成するのであれば、Sジェネレータと同様に各演算器の個数を最小限に抑え、Mジェネレータと同様に演算値保持レジスタを生成しない仕様にする必要がある。また条件分岐をコードジェネレータに実装することで、システムがゆえの不必要な回路生成を行わないよう改良する必要がある。

5.2 考察

5.1.1 において、実行速度と回路規模に特化した改良点や新しいコードジェネレータを提案した。しかし実際の回路設計においては実行速度と回路規模2つのトレードオフを考慮した回路を生成しなければならない。例えばレジスタ数や回路規模(slice 数)、最大クロック数をユーザーに入力してもらい、それらの範囲で回路を設計するシステムを実装する必要がある。

6. おわりに

本研究では、OpenMP ハードウェア動作合成システムに実装されている既存の2つのコードジェネレータによる生成回路と、手書きによる生成回路との性能比較を行った。そしてその結果から、既存の2つのコードジェネレータの抱える問題点を洗い出し、新しいコードジェネレータの提案を行った。比較は素数判定とマンデルブロ集合を用いて行った。また記述量、実行クロック数、回路規模、状態遷移数の4つの点で行った。

素数判定ではほぼすべての点において手書きがシステムを上回ったが、K ジェネレータがノード 4 時の回路規模のみ手書きを上回った。また両コードジェネレータ間では記述量を除いた全ての点において M ジェネレータが上回った。

マンデルブロ集合では記述量、状態遷移数では手書き＞M ジェネレータ＞S ジェネレータの順に優秀であった。また実行クロック数では M ジェネレータ＞手書き＞S ジェネレータ、回路規模では M ジェネレータ≒S ジェネレータ＞手書きという結果になった。

実行速度最速のコードジェネレータを生成するのであれば、状態遷移数を減らし演算の量を増やす必要がある。また 1 クロックの最大演算数に合わせた演算器の生成も求められる。

回路規模最小のコードジェネレータを生成するのであれば、各演算器の個数を最小限に抑え、演算値保持レジスタを生成しないコードジェネレータにする必要がある。またシステムがゆえの不必要な回路生成を行わないように条件分岐をコードジェネレータに実装する等、改良を行う必要がある。

謝辞

本研究の機会を与えてくださり、貴重な助言、ご指導を頂きました山崎勝弘教授に深く感謝いたします。また、度々相談に乗って頂き、貴重な助言を頂いた住井大輔氏に深く感謝いたします。

最後に高性能計算研究室の皆様に心より感謝いたします。

参考文献

[1] 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.

[2] 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.

[3] 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.

[4] 苅屋徹, “OpenMP ハードウェア動作合成システムの検証と評価(Ⅱ)”, 立命館理工学部電子情報デザイン学科卒業論文, 2009.

[5] 金森央樹, “OpenMP ハードウェア動作合成システムの検証と評価(Ⅰ)”, 立命館理工学部電子情報デザイン学科卒業論文, 2009.

[6] 住井大介, “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”, 立命館理工学部電子情報デザイン学科卒業論文, 2010.

[7] 小林優, “改訂・入門 Verilog HDL 記述”,CQ 出版, 2009.

[8] 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.

[9] 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.

[10] 半導体産業新聞編集部, “図解 半導体業界ハンドブック”,東洋経済新報社, 2008.

[11] 松田昭信,南谷崇,“高位合成手法を用いた C ベース設計による LSI 開発事例”, 情報処理学会第 67 回全国大会, p99-100, 2005.

[12] 井上諭, 近藤毅, 泉知論, 福井正博, “C 言語からの高位合成を用いたハードウエア最適化に関する一検討”, 情報処理学会研究報告, Vol.2005, No.102 pp.173-178. , 2005.

付録 a 素数判定の手書きと各コードジェネレータによる回路生成の HDL 記述

手書き(スレーブ部)

```
module sosu_tegaki40( iSTART, oEND, oDATA, iDATA, CLK, XRST);
 input iSTART ,CLK,XRST;
 output oEND;
 reg oEND;
 output[31:0] oDATA;
 reg [31:0] oDATA, i,m;
 input [31:0]iDATA;
wire [31:0] iDATAfor,iDATAto; //手書き修正
assign iDATAfor = 2;              //データ分割(手動)
assign iDATAto = iDATA/8;     //データ分割(手動)
always@(posedge CLK or negedge XRST)
if(!XRST)begin
      oEND <=1'b0;
      i     <= 1'b0;
      m     <= 1'b0;
      oDATA <=32'b0;
end else begin
      if(i==0&&m==0)begin
            i <= iDATAfor;
            m <= iDATA;
      end else if(i==iDATAto)begin
            oEND <=1'b1;
      end else begin
            if(m>i)begin
                  m <= m-i;
            end else if(m==i) begin
                  oDATA<=1;
                  oEND <=1'b1;
            end else if(!oDATA)begin
                  m <= iDATA;
                  i <= i+1;
            end
      end
end  endmodule
```

S ジェネレータ(スレーブ部)

```
module sosu_omp420( iSTART, oEND, oDATA, iDATA, CLK, XRST);
 input iSTART;
 output oEND;
 reg oEND;
 output[31:0] oDATA;
 reg [31:0] oDATA;
 input [31:0]iDATA;
 input CLK,XRST;
 reg [7:0]CurrentState;
 wire [31:0] iDATAfor,iDATAto; //手書き修正
assign iDATAfor = 0;              //データ分割(手動)
assign iDATAto = iDATA/8;     //データ分割(手動)
parameter   [31:0]ConstNum6 = 32'd0;
parameter   [31:0]ConstNum10 = 32'd0;
parameter   [31:0]ConstNum12 = 32'd2;
parameter   [31:0]ConstNum18 = 32'd0;
parameter   [31:0]ConstNum20 = 32'd1;
parameter P_INIT = 8'd0;
parameter P_END   = 8'd1;
parameter P_STATE0 = 8'd2;
parameter P_STATE1 = 8'd3;
parameter P_STATE2 = 8'd4;
parameter P_STATE3 = 8'd5;
parameter P_STATE4 = 8'd6;
parameter P_STATE5 = 8'd7;
parameter P_STATE6 = 8'd8;
parameter P_STATE7 = 8'd9;
parameter P_STATE8 = 8'd10;
parameter P_STATE9 = 8'd11;
parameter P_STATE10 = 8'd12;
parameter P_STATE11 = 8'd13;
parameter P_STATE12 = 8'd14;
parameter P_STATE13 = 8'd15;
parameter P_STATE14 = 8'd16;
parameter P_STATE15 = 8'd17;
```

```
parameter P_STATE16 = 8'd18;
parameter P_STATE17 = 8'd19;
parameter P_STATE18 = 8'd20;
parameter P_STATE19 = 8'd21;
parameter P_STATE20 = 8'd22;
parameter P_STATE21 = 8'd23;
reg   [31:0]i;
reg   [31:0]m;
reg   [31:0]REG8;
reg   [31:0]REG9;
reg   [31:0]REG15;
reg   [31:0]REG16;
//演算器部
wire [31:0]ADD1_RESULT;
wire [31:0]ADD1_A,ADD1_B;
assign ADD1_RESULT = ADD1_A + ADD1_B;
assign ADD1_A = (CurrentState==P_STATE9) ? i :i;
assign ADD1_B = (CurrentState==P_STATE9) ? 32'd1 :i;
wire [31:0]SUB1_RESULT;
wire [31:0]SUB1_A,SUB1_B;
assign SUB1_RESULT = SUB1_A - SUB1_B;
assign SUB1_A = (CurrentState==P_STATE16) ? m :m;
assign SUB1_B = (CurrentState==P_STATE16) ? i :i;
//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;
      P_STATE9 : i <= ADD1_RESULT;
      P_STATE13 : i <= ConstNum12;
      P_STATE14 : m <= iDATA;
      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;      //処理終了
            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  endmodule
```

M ジェネレータ(スレーブ部)

```
module sosu_slave( iSTART, oEND, oDATA, CLK, XRST, SL_START, SL_END );
input   iSTART;
output   oEND;
reg   oEND;
output  [31:0]  oDATA;
//wire  [31:0]  oDATA;
input   CLK;
input   XRST;
input [31:0]  SL_START;
input [31:0]  SL_END;
reg [7:0] GrobalState;
reg [7:0] DomesticState;
reg  [31:0]  i;
reg  [31:0]  m;
reg  [31:0]  oDATA;
parameter D_END = 8'd0;
parameter G_INIT = 8'd1;
parameter G_END = 8'd2;
parameter G_STATE0 = 8'd3;
parameter G_STATE1 = 8'd4;
parameter G_STATE2 = 8'd5;
parameter G_STATE3 = 8'd6;
parameter G_STATE4 = 8'd7;
parameter G_STATE5 = 8'd8;
parameter G_STATE6 = 8'd9;
parameter G_STATE7 = 8'd10;
parameter G_STATE8 = 8'd11;
parameter G_STATE9 = 8'd12;
parameter G_STATE10 = 8'd13;
//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;
      oDATA <= 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_STATE4 : begin
            m <= 100003;
        end
        G_STATE5 : begin
            i <= 2;
        end
        G_STATE8 : begin
            m <= m - i;
        end
        G_STATE9 : begin
            i <= i + 1;
        end
        G_STATE10 : begin
            oDATA <= 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 <= GrobalState;
      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_STATE1;
        end
      endcase
   end
end
endmodule
```

付録 b マンデルブロ集合の手書きと各コードジェネレータによる回路生成の HDL 記述

手書き(スレーブ部)

```
module mandel( iSTART, oEND, oADDR, oDATA, iDATA, oRD, oWR, iEN, iRUN, iSTALL, clk,
      rst, SL_START, SL_END, in_a, in_b, in_bit_size);
      output[31:0]oADDR,oDATA;
      reg [31:0]oADDR,oDATA;
      input [31:0]iDATA;
      output oRD, oWR;
      reg oRD,oWR;
      input iEN;
      input [31:0]iRUN;
      input iSTALL;
      input signed  [31:0] SL_START;
      wire signed  [31:0]  SL_START;
      reg signed [31:0]  in_SL_START;
      input        iSTART;
      output reg oEND;
      input signed  [31:0] in_a;
      wire signed  [31:0]  in_a;
      input signed  [31:0] in_b;
      wire signed  [31:0]  in_b;
      input signed  [31:0] in_bit_size;
      wire signed  [31:0]  in_bit_size;
      reg signed[31:0] a;
      reg signed[31:0] b;
      reg signed[31:0] bit_size;
      input        clk,rst;
      wire         clk,rst;
      reg     signed [31:0]  j;
      input signed [31:0]  SL_END;
      reg                  count;
      reg          signed [31:0] xn,yn, xn1,yn1;
      reg          [7:0] state;
always@(posedge clk or negedge rst) begin
      if(!rst) begin
          oEND=0;
```

```
        count=0;
        xn=0;
        yn=0;
        xn1=0;
        yn1=0;
        state=0;
        j=0;
        a=in_a;
        b=in_b;
        bit_size=in_bit_size;
        in_SL_START=SL_START;
    end else if (iSTART == 1 ) begin
        if( state == 0 ) begin
                if( in_SL_START<SL_END ) begin
                  state = 1;
           end else begin
                  state = 3;
           end
        end
                        if( state == 1 ) begin
                if(j<100) begin
                        if( count<30 ) begin
                                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) )
    begin
                                                state = 2;
                                           end
                                count = count+1;
                   end else begin
                   a = a + bit_size;
                   j = j+1;
                   xn=0;
```

```
                    yn=0;
                    xn1=0;
                    yn1=0;
                    end
                end else begin
                state = 2;
                end
        end
                        if(state == 2) begin
                b = b - bit_size;
                a = (-2)<<16;
                in_SL_START = in_SL_START + 1;
                state = 0;
        end
                        if( state == 3 ) begin
                oEND = 1;
        end
    end
end
endmodule
```

S ジェネレータ(スレーブ部)

```
module mandel( iSTART, oEND, oADDR, oDATA, iDATA, oRD, oWR, iEN, iRUN, iSTALL,
      CLK, XRST, SL_START, SL_END, in_a, in_b, in_bit_size);
 output[31:0]oADDR,oDATA;
 reg [31:0]oADDR,oDATA;
 input [31:0]iDATA;
 output oRD, oWR;
 reg oRD,oWR;
 input iEN;
 input [31:0]iRUN;
 input iSTALL;
 input iSTART;
 output oEND;
 reg oEND;
 input CLK,XRST;
 input signed [31:0]  SL_START;
 input signed [31:0]  SL_END;
 input signed  [31:0] in_a;
 wire signed  [31:0]  in_a;
 input signed  [31:0] in_b;
 wire signed  [31:0]  in_b;
 input signed  [31:0] in_bit_size;
 wire signed  [31:0]  in_bit_size;
 reg [7:0]CurrentState;
 reg signed  [31:0]  i;
 reg signed  [31:0]  j;
 reg signed  [31:0]  a;
 reg signed  [31:0]  b;
 reg signed  [31:0]  bit_size;
 reg signed  [31:0]  xn;
 reg signed  [31:0]  yn;
 reg signed  [31:0]  xn1;
 reg signed  [31:0]  yn1;
 reg signed  [31:0]  counter;
parameter  [31:0]ConstNum11 = 32'd2;
parameter  [31:0]ConstNum13 = 32'd16;
```

```
parameter   [31:0]ConstNum16 = 32'd2;
parameter   [31:0]ConstNum17 = 32'd16;
parameter   [31:0]ConstNum20 = 32'd2;
parameter   [31:0]ConstNum21 = 32'd16;
parameter   [31:0]ConstNum23 = 32'd2;
parameter   [31:0]ConstNum25 = 32'd16;
parameter   [31:0]ConstNum28 = 32'd99;
parameter   [31:0]ConstNum31 = 32'd0;
parameter   [31:0]ConstNum33 = 32'd25;
parameter   [31:0]ConstNum35 = 32'd1;
parameter   [31:0]ConstNum38 = 32'd0;
parameter   [31:0]ConstNum40 = 32'd100;
parameter   [31:0]ConstNum42 = 32'd1;
parameter   [31:0]ConstNum45 = 32'd0;
parameter   [31:0]ConstNum47 = 32'd0;
parameter   [31:0]ConstNum49 = 32'd0;
parameter   [31:0]ConstNum51 = 32'd0;
parameter   [31:0]ConstNum53 = 32'd0;
parameter   [31:0]ConstNum55 = 32'd30;
parameter   [31:0]ConstNum57 = 32'd1;
parameter   [31:0]ConstNum60 = 32'd8;
parameter   [31:0]ConstNum62 = 32'd8;
parameter   [31:0]ConstNum65 = 32'd8;
parameter   [31:0]ConstNum67 = 32'd8;
parameter   [31:0]ConstNum73 = 32'd8;
parameter   [31:0]ConstNum75 = 32'd8;
parameter   [31:0]ConstNum79 = 32'd2;
parameter   [31:0]ConstNum80 = 32'd16;
parameter   [31:0]ConstNum82 = 32'd8;
parameter   [31:0]ConstNum84 = 32'd8;
parameter   [31:0]ConstNum91 = 32'd8;
parameter   [31:0]ConstNum93 = 32'd8;
parameter   [31:0]ConstNum96 = 32'd8;
parameter   [31:0]ConstNum98 = 32'd8;
parameter   [31:0]ConstNum102 = 32'd4;
parameter   [31:0]ConstNum103 = 32'd16;
```

```
parameter   [31:0]ConstNum110 = 32'd2;
parameter   [31:0]ConstNum112 = 32'd16;
parameter P_INIT = 8'd0;
parameter P_END   = 8'd1;
parameter P_STATE0 = 8'd2;
parameter P_STATE1 = 8'd3;
parameter P_STATE2 = 8'd4;
parameter P_STATE3 = 8'd5;
parameter P_STATE4 = 8'd6;
parameter P_STATE5 = 8'd7;
parameter P_STATE6 = 8'd8;
parameter P_STATE7 = 8'd9;
parameter P_STATE8 = 8'd10;
parameter P_STATE9 = 8'd11;
parameter P_STATE10 = 8'd12;
parameter P_STATE11 = 8'd13;
parameter P_STATE12 = 8'd14;
parameter P_STATE13 = 8'd15;
parameter P_STATE14 = 8'd16;
parameter P_STATE15 = 8'd17;
parameter P_STATE16 = 8'd18;
parameter P_STATE17 = 8'd19;
parameter P_STATE18 = 8'd20;
parameter P_STATE19 = 8'd21;
parameter P_STATE20 = 8'd22;
parameter P_STATE21 = 8'd23;
parameter P_STATE22 = 8'd24;
parameter P_STATE23 = 8'd25;
parameter P_STATE24 = 8'd26;
parameter P_STATE25 = 8'd27;
parameter P_STATE26 = 8'd28;
parameter P_STATE27 = 8'd29;
parameter P_STATE28 = 8'd30;
parameter P_STATE29 = 8'd31;
parameter P_STATE30 = 8'd32;
parameter P_STATE31 = 8'd33;
```

```
parameter P_STATE32 = 8'd34;
parameter P_STATE33 = 8'd35;
parameter P_STATE34 = 8'd36;
parameter P_STATE35 = 8'd37;
parameter P_STATE36 = 8'd38;
parameter P_STATE37 = 8'd39;
parameter P_STATE38 = 8'd40;
parameter P_STATE39 = 8'd41;
parameter P_STATE40 = 8'd42;
parameter P_STATE41 = 8'd43;
parameter P_STATE42 = 8'd44;
parameter P_STATE43 = 8'd45;
parameter P_STATE44 = 8'd46;
parameter P_STATE45 = 8'd47;
parameter P_STATE46 = 8'd48;
parameter P_STATE47 = 8'd49;
parameter P_STATE48 = 8'd50;
parameter P_STATE49 = 8'd51;
parameter P_STATE50 = 8'd52;
parameter P_STATE51 = 8'd53;
parameter P_STATE52 = 8'd54;
parameter P_STATE53 = 8'd55;
parameter P_STATE54 = 8'd56;
parameter P_STATE55 = 8'd57;
parameter P_STATE56 = 8'd58;
parameter P_STATE57 = 8'd59;
parameter P_STATE58 = 8'd60;
parameter P_STATE59 = 8'd61;
parameter P_STATE60 = 8'd62;
parameter P_STATE61 = 8'd63;
parameter P_STATE62 = 8'd64;
parameter P_STATE63 = 8'd65;
parameter P_STATE64 = 8'd66;
parameter P_STATE65 = 8'd67;
parameter P_STATE66 = 8'd68;
parameter P_STATE67 = 8'd69;
```

```
parameter P_STATE68 = 8'd70;
parameter P_STATE69 = 8'd71;
parameter P_STATE70 = 8'd72;
parameter P_STATE71 = 8'd73;
parameter P_STATE72 = 8'd74;
parameter P_STATE73 = 8'd75;
parameter P_STATE74 = 8'd76;
parameter P_STATE75 = 8'd77;
parameter P_STATE76 = 8'd78;
parameter P_STATE77 = 8'd79;
parameter P_STATE78 = 8'd80;
parameter P_STATE79 = 8'd81;
parameter P_STATE80 = 8'd82;
parameter P_STATE81 = 8'd83;
parameter P_STATE82 = 8'd84;
parameter P_STATE83 = 8'd85;
parameter P_STATE84 = 8'd86;
parameter P_STATE85 = 8'd87;
parameter P_STATE86 = 8'd88;
parameter P_STATE87 = 8'd89;
parameter P_STATE88 = 8'd90;
parameter P_STATE89 = 8'd91;
parameter P_STATE90 = 8'd92;
parameter P_STATE91 = 8'd93;
parameter P_STATE92 = 8'd94;
parameter P_STATE93 = 8'd95;
parameter P_STATE94 = 8'd96;
parameter P_STATE95 = 8'd97;
parameter P_STATE96 = 8'd98;
parameter P_STATE97 = 8'd99;
parameter P_STATE98 = 8'd100;
parameter P_STATE99 = 8'd101;
parameter P_STATE100 = 8'd102;
parameter P_STATE101 = 8'd103;
parameter P_STATE102 = 8'd104;
parameter P_STATE103 = 8'd105;
```

```
parameter P_STATE104 = 8'd106;
parameter P_STATE105 = 8'd107;
parameter P_STATE106 = 8'd108;
parameter P_STATE107 = 8'd109;
parameter P_STATE108 = 8'd110;
parameter P_STATE109 = 8'd111;
parameter P_STATE110 = 8'd112;
parameter P_STATE111 = 8'd113;
parameter P_STATE112 = 8'd114;
parameter P_STATE113 = 8'd115;
parameter P_STATE114 = 8'd116;
reg   [31:0]REG0;
reg   [31:0]REG12;
reg   [31:0]REG14;
reg   [31:0]REG18;
reg   [31:0]REG22;
reg   [31:0]REG24;
reg   [31:0]REG26;
reg   [31:0]REG27;
reg   [31:0]REG29;
reg   [31:0]REG34;
reg   [31:0]REG36;
reg   [31:0]REG41;
reg   [31:0]REG43;
reg   [31:0]REG56;
reg   [31:0]REG58;
reg   [31:0]REG61;
reg   [31:0]REG63;
reg   [31:0]REG64;
reg   [31:0]REG66;
reg   [31:0]REG68;
reg   [31:0]REG69;
reg   [31:0]REG70;
reg   [31:0]REG71;
reg   [31:0]REG74;
reg   [31:0]REG76;
```

```
reg   [31:0]REG77;
reg   [31:0]REG81;
reg   [31:0]REG83;
reg   [31:0]REG85;
reg   [31:0]REG86;
reg   [31:0]REG87;
reg   [31:0]REG92;
reg   [31:0]REG94;
reg   [31:0]REG95;
reg   [31:0]REG97;
reg   [31:0]REG99;
reg   [31:0]REG100;
reg   [31:0]REG101;
reg   [31:0]REG104;
reg   [31:0]REG105;
reg   [31:0]REG106;
reg   [31:0]REG108;
reg   [31:0]REG111;
reg   [31:0]REG113;
//演算器部
wire [31:0]ADD1_RESULT;
wire [31:0]ADD1_A,ADD1_B;
assign ADD1_RESULT = ADD1_A + ADD1_B;
assign ADD1_A = (CurrentState==P_STATE36) ? i :
(CurrentState==P_STATE43) ? j :
(CurrentState==P_STATE58) ? counter :
(CurrentState==P_STATE71) ? REG70 :
(CurrentState==P_STATE87) ? REG86 :
(CurrentState==P_STATE101) ? REG95 :
(CurrentState==P_STATE106) ? a :
a;
assign ADD1_B = (CurrentState==P_STATE36) ? ConstNum35 :
(CurrentState==P_STATE43) ? ConstNum42 :
(CurrentState==P_STATE58) ? ConstNum57 :
(CurrentState==P_STATE71) ? a :
(CurrentState==P_STATE87) ? b :
```

```
(CurrentState==P_STATE101) ? REG100 :
(CurrentState==P_STATE106) ? bit_size :
bit_size;
wire [31:0]SUB1_RESULT;
wire [31:0]SUB1_A,SUB1_B;
assign SUB1_RESULT = SUB1_A - SUB1_B;
assign SUB1_A = (CurrentState==P_STATE12) ? 32'd0 :
(CurrentState==P_STATE24) ? 32'd0 :
(CurrentState==P_STATE27) ? REG22 :
(CurrentState==P_STATE70) ? REG64 :
(CurrentState==P_STATE108) ? b :
(CurrentState==P_STATE111) ? 32'd0 :
b;
assign SUB1_B = (CurrentState==P_STATE12) ? ConstNum11 :
(CurrentState==P_STATE24) ? ConstNum23 :
(CurrentState==P_STATE27) ? REG26 :
(CurrentState==P_STATE70) ? REG69 :
(CurrentState==P_STATE108) ? bit_size :
(CurrentState==P_STATE111) ? ConstNum110 :
ConstNum110;
wire [31:0]MUL1_RESULT;
wire [31:0]MUL1_A,MUL1_B;
assign MUL1_RESULT = MUL1_A * MUL1_B;
assign MUL1_A = (CurrentState==P_STATE64) ? REG61 :
(CurrentState==P_STATE69) ? REG66 :
(CurrentState==P_STATE77) ? REG74 :
(CurrentState==P_STATE86) ? REG83 :
(CurrentState==P_STATE95) ? REG92 :
(CurrentState==P_STATE100) ? REG97 :
REG97;
assign MUL1_B = (CurrentState==P_STATE64) ? REG63 :
(CurrentState==P_STATE69) ? REG68 :
(CurrentState==P_STATE77) ? REG76 :
(CurrentState==P_STATE86) ? REG85 :
(CurrentState==P_STATE95) ? REG94 :
(CurrentState==P_STATE100) ? REG99 :
```

```
REG99;
wire [31:0]DIV1_RESULT;
wire [31:0]DIV1_A,DIV1_B;
assign DIV1_RESULT = DIV1_A / DIV1_B;
assign DIV1_A = (CurrentState==P_STATE29) ? REG27 :
REG27;
assign DIV1_B = (CurrentState==P_STATE29) ? ConstNum28 :
ConstNum28;
wire signed [31:0]RSFT1_RESULT;
wire signed [31:0]RSFT1_A,RSFT1_B;
assign RSFT1_RESULT = RSFT1_A >>> RSFT1_B;
assign RSFT1_A = (CurrentState==P_STATE61) ? xn :
(CurrentState==P_STATE63) ? xn :
(CurrentState==P_STATE66) ? yn :
(CurrentState==P_STATE68) ? yn :
(CurrentState==P_STATE74) ? xn :
(CurrentState==P_STATE76) ? yn :
(CurrentState==P_STATE83) ? REG81 :
(CurrentState==P_STATE85) ? yn :
(CurrentState==P_STATE92) ? xn :
(CurrentState==P_STATE94) ? xn :
(CurrentState==P_STATE97) ? yn :
(CurrentState==P_STATE99) ? yn :
yn ;
assign RSFT1_B = (CurrentState==P_STATE61) ? ConstNum60 :
(CurrentState==P_STATE63) ? ConstNum62 :
(CurrentState==P_STATE66) ? ConstNum65 :
(CurrentState==P_STATE68) ? ConstNum67 :
(CurrentState==P_STATE74) ? ConstNum73 :
(CurrentState==P_STATE76) ? ConstNum75 :
(CurrentState==P_STATE83) ? ConstNum82 :
(CurrentState==P_STATE85) ? ConstNum84 :
(CurrentState==P_STATE92) ? ConstNum91 :
(CurrentState==P_STATE94) ? ConstNum93 :
(CurrentState==P_STATE97) ? ConstNum96 :
(CurrentState==P_STATE99) ? ConstNum98 :
```

```
ConstNum98 ;
wire signed [31:0]LSFT1_RESULT;
wire signed [31:0]LSFT1_A,LSFT1_B;
assign LSFT1_RESULT = LSFT1_A <<< LSFT1_B;
assign LSFT1_A = (CurrentState==P_STATE14) ? REG12 :
(CurrentState==P_STATE18) ? ConstNum16 :
(CurrentState==P_STATE22) ? ConstNum20 :
(CurrentState==P_STATE26) ? REG24 :
(CurrentState==P_STATE81) ? ConstNum79 :
(CurrentState==P_STATE104) ? ConstNum102 :
(CurrentState==P_STATE113) ? REG111 :
REG111;
assign LSFT1_B = (CurrentState==P_STATE14) ? ConstNum13 :
(CurrentState==P_STATE18) ? ConstNum17 :
(CurrentState==P_STATE22) ? ConstNum21 :
(CurrentState==P_STATE26) ? ConstNum25 :
(CurrentState==P_STATE81) ? ConstNum80 :
(CurrentState==P_STATE104) ? ConstNum103 :
(CurrentState==P_STATE113) ? ConstNum112 :
ConstNum112;
//代入部
always @ (posedge CLK or negedge XRST) begin
 if(!XRST) begin
    oEND <= 1'b0;
    i <= 32'd0;
    j <= 32'd0;
    a <= 32'd0;
    b <= 32'd0;
    bit_size <= 32'd0;
    xn <= 32'd0;
    yn <= 32'd0;
    xn1 <= 32'd0;
    yn1 <= 32'd0;
    counter <= 32'd0;
    REG0 <= 32'd0;
    REG12 <= 32'd0;
```

```
REG14 <= 32'd0;
REG18 <= 32'd0;
REG22 <= 32'd0;
REG24 <= 32'd0;
REG26 <= 32'd0;
REG27 <= 32'd0;
REG29 <= 32'd0;
REG34 <= 32'd0;
REG36 <= 32'd0;
REG41 <= 32'd0;
REG43 <= 32'd0;
REG56 <= 32'd0;
REG58 <= 32'd0;
REG61 <= 32'd0;
REG63 <= 32'd0;
REG64 <= 32'd0;
REG66 <= 32'd0;
REG68 <= 32'd0;
REG69 <= 32'd0;
REG70 <= 32'd0;
REG71 <= 32'd0;
REG74 <= 32'd0;
REG76 <= 32'd0;
REG77 <= 32'd0;
REG81 <= 32'd0;
REG83 <= 32'd0;
REG85 <= 32'd0;
REG86 <= 32'd0;
REG87 <= 32'd0;
REG92 <= 32'd0;
REG94 <= 32'd0;
REG95 <= 32'd0;
REG97 <= 32'd0;
REG99 <= 32'd0;
REG100 <= 32'd0;
REG101 <= 32'd0;
```

```
    REG104 <= 32'd0;
    REG105 <= 32'd0;
    REG106 <= 32'd0;
    REG108 <= 32'd0;
    REG111 <= 32'd0;
    REG113 <= 32'd0;
end else begin
  case(CurrentState)
     P_INIT : oEND <= 1'b0;
     P_END   : oEND <= 1'b1;
     P_STATE12 : REG12 <= SUB1_RESULT;
     P_STATE14 : REG14 <= LSFT1_RESULT;
     P_STATE18 : REG18 <= LSFT1_RESULT;
     P_STATE22 : REG22 <= LSFT1_RESULT;
     P_STATE24 : REG24 <= SUB1_RESULT;
     P_STATE26 : REG26 <= LSFT1_RESULT;
     P_STATE27 : REG27 <= SUB1_RESULT;
     P_STATE29 : a <= in_a;
     P_STATE30 : b <= in_b;
     P_STATE31 : bit_size <= in_bit_size;
     P_STATE32 : i <= SL_START;
     P_STATE36 : REG36 <= ADD1_RESULT;
     P_STATE37 : i <= REG36;
     P_STATE39 : j <= ConstNum38;
     P_STATE43 : REG43 <= ADD1_RESULT;
     P_STATE44 : j <= REG43;
     P_STATE46 : xn <= ConstNum45;
     P_STATE48 : yn <= ConstNum47;
     P_STATE50 : xn1 <= ConstNum49;
     P_STATE52 : yn1 <= ConstNum51;
     P_STATE54 : counter <= ConstNum53;
     P_STATE58 : REG58 <= ADD1_RESULT;
     P_STATE59 : counter <= REG58;
     P_STATE61 : REG61 <= RSFT1_RESULT;
     P_STATE63 : REG63 <= RSFT1_RESULT;
     P_STATE64 : REG64 <= MUL1_RESULT;
```

```
    P_STATE66 : REG66 <= RSFT1_RESULT;
    P_STATE68 : REG68 <= RSFT1_RESULT;
    P_STATE69 : REG69 <= MUL1_RESULT;
    P_STATE70 : REG70 <= SUB1_RESULT;
    P_STATE71 : REG71 <= ADD1_RESULT;
    P_STATE72 : xn1 <= REG71;
    P_STATE74 : REG74 <= RSFT1_RESULT;
    P_STATE76 : REG76 <= RSFT1_RESULT;
    P_STATE77 : REG77 <= MUL1_RESULT;
    P_STATE78 : yn1 <= REG77 ;
    P_STATE81 : REG81 <= LSFT1_RESULT;
    P_STATE83 : REG83 <= RSFT1_RESULT;
    P_STATE85 : REG85 <= RSFT1_RESULT;
    P_STATE86 : REG86 <= MUL1_RESULT;
    P_STATE87 : REG87 <= ADD1_RESULT;
    P_STATE88 : yn1 <= REG87;
    P_STATE89 : xn <= xn1;
    P_STATE90 : yn <= yn1;
    P_STATE92 : REG92 <= RSFT1_RESULT;
    P_STATE94 : REG94 <= RSFT1_RESULT;
    P_STATE95 : REG95 <= MUL1_RESULT;
    P_STATE97 : REG97 <= RSFT1_RESULT;
    P_STATE99 : REG99 <= LSFT1_RESULT;
    P_STATE100 : REG100 <= MUL1_RESULT;
    P_STATE101 : REG101 <= ADD1_RESULT;
    P_STATE104 : REG104 <= LSFT1_RESULT;
    P_STATE106 : REG106 <= ADD1_RESULT;
    P_STATE107 : a <= REG106;
    P_STATE108 : REG108 <= SUB1_RESULT;
    P_STATE109 : b <= REG108;
    P_STATE111 : REG111 <= SUB1_RESULT;
    P_STATE113 : REG113 <= LSFT1_RESULT;
    P_STATE114 : a <= REG113 ;
    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_INIT     : if(iSTART==1'b1) CurrentState <= P_STATE29;
                     else CurrentState <= CurrentState;
        P_END      : CurrentState <= CurrentState;
        P_STATE29: CurrentState <= P_STATE30;
        P_STATE30: CurrentState <= P_STATE31;
        P_STATE31: CurrentState <= P_STATE32;
        P_STATE32: CurrentState <= P_STATE34;
        P_STATE34: if(i<SL_END) CurrentState <= P_STATE39;
                     else CurrentState <= P_END;
        P_STATE36: CurrentState <= P_STATE37;
        P_STATE37: CurrentState <= P_STATE34;
        P_STATE39: CurrentState <= P_STATE41;
        P_STATE41: if(j<ConstNum40) CurrentState <= P_STATE46;
                     else CurrentState <= P_STATE108;
        P_STATE43: CurrentState <= P_STATE44;
        P_STATE44: CurrentState <= P_STATE41;
        P_STATE46: CurrentState <= P_STATE48;
        P_STATE48: CurrentState <= P_STATE50;
        P_STATE50: CurrentState <= P_STATE52;
        P_STATE52: CurrentState <= P_STATE54;
        P_STATE54: CurrentState <= P_STATE56;
        P_STATE56: if(counter<ConstNum55) CurrentState <= P_STATE61;
                     else CurrentState <= P_STATE106;
        P_STATE58: CurrentState <= P_STATE59;
        P_STATE59: CurrentState <= P_STATE56;
        P_STATE61: CurrentState <= P_STATE63;
        P_STATE63: CurrentState <= P_STATE64;
        P_STATE64: CurrentState <= P_STATE66;
        P_STATE66: CurrentState <= P_STATE68;
        P_STATE68: CurrentState <= P_STATE69;
```

```
            P_STATE69: CurrentState <= P_STATE70;
            P_STATE70: CurrentState <= P_STATE71;
            P_STATE71: CurrentState <= P_STATE72;
            P_STATE72: CurrentState <= P_STATE74;
            P_STATE74: CurrentState <= P_STATE76;
            P_STATE76: CurrentState <= P_STATE77;
            P_STATE77: CurrentState <= P_STATE78;
            P_STATE78: CurrentState <= P_STATE81;
            P_STATE81: CurrentState <= P_STATE83;
            P_STATE83: CurrentState <= P_STATE85;
            P_STATE85: CurrentState <= P_STATE86;
            P_STATE86: CurrentState <= P_STATE87;
            P_STATE87: CurrentState <= P_STATE88;
            P_STATE88: CurrentState <= P_STATE89;
            P_STATE89: CurrentState <= P_STATE90;
            P_STATE90: CurrentState <= P_STATE92;
            P_STATE92: CurrentState <= P_STATE94;
            P_STATE94: CurrentState <= P_STATE95;
            P_STATE95: CurrentState <= P_STATE97;
            P_STATE97: CurrentState <= P_STATE99;
            P_STATE99: CurrentState <= P_STATE100;
            P_STATE100: CurrentState <= P_STATE101;
            P_STATE101: CurrentState <= P_STATE104;
            P_STATE104: CurrentState <= P_STATE105;
            P_STATE105: if(REG101>REG104) CurrentState <= P_STATE106;
                        else CurrentState <= P_STATE58;
            P_STATE106: CurrentState <= P_STATE107;
            P_STATE107: CurrentState <= P_STATE43;
            P_STATE108: CurrentState <= P_STATE109;
            P_STATE109: CurrentState <= P_STATE111;
            P_STATE111: CurrentState <= P_STATE113;
            P_STATE113: CurrentState <= P_STATE114;
            P_STATE114: CurrentState <= P_STATE36;
            default : CurrentState <= CurrentState;
        endcase
end endmodule
```

M ジェネレータ(スレーブ部)

```
//input,wire,reg に signed を追加し、<<を<<<に、>>を>>>に書き換えました
module mandel_slave( iSTART, oEND, oDATA, CLK, XRST, SL_START, SL_END, in_a/*,
      out_a*/, in_b/*, out_b*/, in_bit_size/*, out_bit_size*/ );
input   iSTART;
output   oEND;
reg   oEND;
output signed  [31:0]  oDATA;
wire signed  [31:0]  oDATA;
input   CLK;
input   XRST;
input signed [31:0]  SL_START;
input signed [31:0]  SL_END;
input signed  [31:0]  in_a;
wire signed  [31:0]  in_a;
input signed  [31:0]  in_b;
wire signed  [31:0]  in_b;
input signed  [31:0]  in_bit_size;
wire signed  [31:0]  in_bit_size;
reg [7:0] GrobalState;
reg [7:0] DomesticState;
reg signed  [31:0]  i;
reg signed  [31:0]  j;
reg signed  [31:0]  b;
reg signed  [31:0]  bit_size;
reg signed  [31:0]  a;
reg signed  [31:0]  xn;
reg signed  [31:0]  yn;
reg signed  [31:0]  xn1;
reg signed  [31:0]  yn1;
reg signed  [31:0]  counter;
parameter D_END = 8'd0;
parameter G_INIT = 8'd1;
parameter G_END = 8'd2;
parameter G_STATE0 = 8'd3;
parameter G_STATE1 = 8'd4;
```

```
parameter G_STATE2 = 8'd5;
parameter G_STATE3 = 8'd6;
parameter G_STATE4 = 8'd7;
parameter G_STATE5 = 8'd8;
parameter G_STATE6 = 8'd9;
parameter G_STATE7 = 8'd10;
parameter G_STATE8 = 8'd11;
parameter G_STATE9 = 8'd12;
parameter G_STATE10 = 8'd13;
parameter G_STATE11 = 8'd14;
parameter G_STATE12 = 8'd15;
parameter G_STATE13 = 8'd16;
parameter G_STATE14 = 8'd17;
//代入部
always @ (posedge CLK or negedge XRST) begin
  if(!XRST) begin
    oEND <= 1'b0;
    DomesticState <= D_END;
    i <= 32'b0;
    j <= 32'b0;
    b <= 32'b0;
    bit_size <= 32'b0;
    a <= 32'b0;
    xn <= 32'b0;
    yn <= 32'b0;
    xn1 <= 32'b0;
    yn1 <= 32'b0;
    counter <= 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;
                a <= in_a;
                b <= in_b;
                bit_size <= in_bit_size;
end
G_STATE3 : begin
        j <= 0;
end
G_STATE4 : begin
        b <= b - bit_size;
        a <=   -2 <<< 16;
end
G_STATE6 : begin
        xn <= 0;
        yn <= 0;
        xn1 <= 0;
        yn1 <= 0;
end
G_STATE7 : begin
        counter <= 0;
end
G_STATE8 : begin
        a <= a + bit_size;
end
G_STATE10 : begin
    if(DomesticState == 8'd1) begin
        DomesticState <= 8'd2;
        xn1 <= (xn >>> 8) * (xn >>> 8) - (yn >>> 8) * (yn >>> 8) + a;
        yn1 <= (xn >>> 8) * (yn >>> 8);
    end
    else if(DomesticState == 8'd2) begin
        DomesticState <= 8'd3;
        yn1 <= ((2 <<< 16) >>> 8) * (yn1 >>> 8) + b;
        xn <= xn1;
    end
```

```
            else if(DomesticState == 8'd3) begin
               DomesticState <= 8'd4;
               yn <= yn1;
            end
            else DomesticState <= D_END;
         end
         G_STATE12 : begin
               counter <= counter + 1;
         end
         G_STATE13 : begin
               j <= j + 1;
         end
         G_STATE14 : begin
               i <= i + 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 <= GrobalState;
         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_STATE5;
    end
    G_STATE4 : begin
        GrobalState <= G_STATE14;
    end
    G_STATE5 : begin
      if(j < 100) begin
        GrobalState <= G_STATE6;
      end
      else GrobalState <= G_STATE4;
    end
    G_STATE6 : begin
        GrobalState <= G_STATE7;
    end
    G_STATE7 : begin
        GrobalState <= G_STATE9;
    end
    G_STATE8 : begin
        GrobalState <= G_STATE13;
    end
    G_STATE9 : begin
      if(counter < 30) begin
        GrobalState <= G_STATE10;
      end
```

```
            else GrobalState <= G_STATE8;
        end
        G_STATE10 : begin
            if(DomesticState == D_END) begin
                GrobalState <= G_STATE11;
            end
            else GrobalState <= GrobalState;
        end
        G_STATE11 : begin
            if((xn >>> 8) * (xn >>> 8) + (yn >>> 8) * (yn >>> 8) > (4 <<< 16)) begin
                GrobalState <= G_STATE8;
            end
            else GrobalState <= G_STATE12;
        end
        G_STATE12 : begin
                GrobalState <= G_STATE9;
        end
        G_STATE13 : begin
                GrobalState <= G_STATE5;
        end
        G_STATE14 : begin
                GrobalState <= G_STATE2;
        end
      endcase
   end
end
endmodule
```