内容梗概

本研究では、既存のコードジェネレータの抱えるデータ依存計算が不十分という課題を解決すべく、セグメント単位でのデータ依存計算を行う新たなコードジェネレータを作成し、評価を行なった。分岐するまでの連続したステートをセグメントとしてとらえ、セグメント内で各演算の依存関係を計算する事で、十分なデータ依存計算を実現する。データ依存計算を十分に行う事で、状態遷移数を削減し、実行時間の短縮を実現する。また増加した一状態での演算量に見合う演算器を実装すべく、演算器部を作成する。評価方法は、マンデルブロ集合、バイトニックソート、基数ソートに対して、既存の２つのコードジェネレータと新コードジェネレータを使用して回路を生成し、比較を行う。評価を行う回路パラメータは実行クロック数、実行時間、演算器数、回路規模である。

目次

図目次

表目次

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。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、生産設計能力の向上が求められている。

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 System C や specC、Cyber Work Bench など、実用的なツールとして、実際の製品開発に適用され始めている。[14]

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。実際に使用されている Spec C や SystemC などのツールでも、並列化制御のサポートを行うなど、並列ハードウェアの自動生成に向けて進んでいるが、別途スレッド作成や扱う変数の受け渡しを記述するなど、設計者が責任を持って記述しなければならない。[13]

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][13]。

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]。

本研究では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。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]。しかし依然として、データ依存の計算が不十分という課題がある。

この課題を解決するために、本研究ではデータ依存計算を十分に行える新しいコードジェネレータの作成、評価を行う。新コードジェネレータでは、分岐するまでの連続したステートをセグメントとしてとらえ、セグメント内で各演算の依存関係を計算する事で、十分なデータ依存計算を実現する。データ依存計算を十分に行う事で、状態遷移数を削減し、実行時間の短縮を実現する。また増加した一状態での演算量に見合う演算器を実装すべく、演算器部を作成する。評価方法は、マンデルブロ集合、バイトニックソート、基数ソートに対して、既存の２つのコードジェネレータと新コードジェネレータを使用して回路を生成し、比較を行う。評価を行う回路パラメータは実行クロック数、実行時間、演算器数、回路規模である。

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解説する。第3章ではハードウェアモジュールの生成方法と新コードジェネレータの改良点について述べる。第4章ではマンデルブロ集合、バイオニックソート、基数ソートに対する動作合成の実験結果を示す。第5章では生成回路の評価と考察を行う。

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。

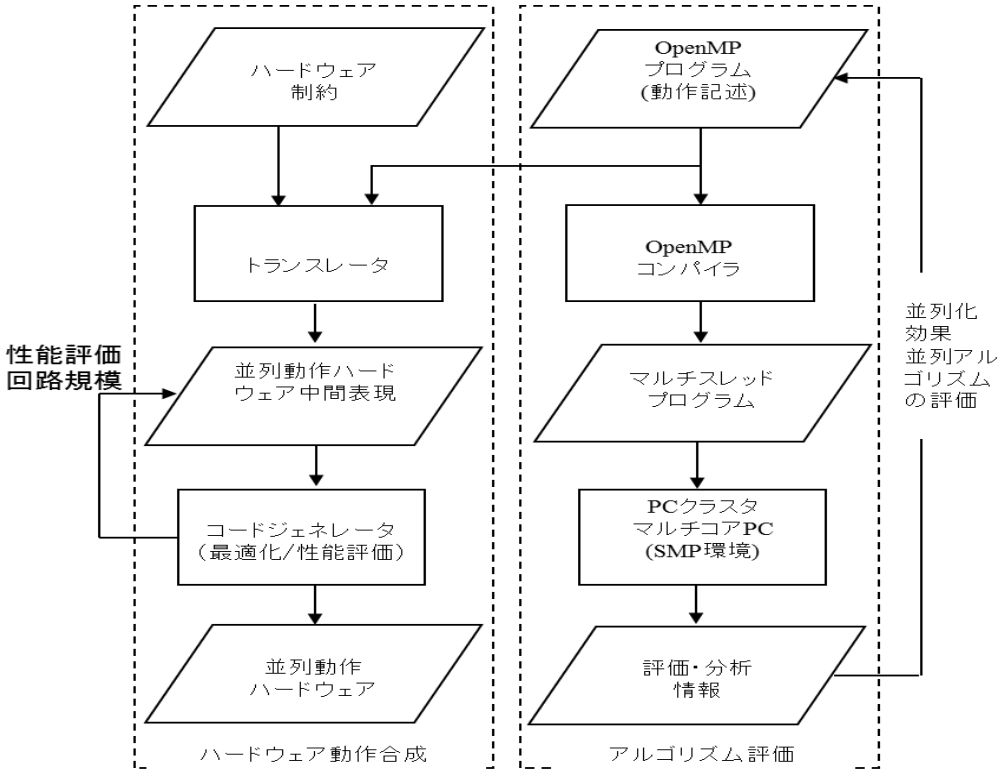


図 1　OpenMP を用いたハードウェア動作合成システム

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でき、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 、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。コードジェネレータには、一状態一演算を実装したSジェネレータと一状態複数演算を実装したMジェネレータの２種類あるが、依然としてデータ依存計算が不十分という課題がある。

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言語コードは比較でも使用したマンデルブロ集合である。状態遷移表の#0で示される状態から、最初にCFGの32で示される =(1 31)の処理が行われる。 =(1 31)ではシンボル1 で示される変数iに、シンボル31で示される定数0を代入することを示している。次にforループの条件式(終了条件)である#2へ遷移して、条件式の判定を行う。ここではi<100?が条件であり、条件が真であるなら#3に、偽であれば#1に遷移する。同様に、#3ではCFGの39に該当する代入を行った後、#5へ遷移、#5ではCFGの39に該当する条件分岐を行う等、終了条件を満たすまで、実行を続ける。

図 2　マンデルブロ集合における中間表現の例

3. コードジェネレータの改良点

Ｌジェネレータの改良点は２点あり、状態遷移数の削減と演算器部の作成である。状態遷移数を削減し、データ依存のない演算を同時に行うことで、少ないステートマシンで実行を行い、実行時間を短縮する。また状態遷移数を削減することで１状態の演算量が増えるため、演算器部を作成し、処理に必要な量の演算器を実装する。

3.1　従来のコード生成手法

従来のコードジェネレータによる中間表現からのコード生成手法について説明する。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。

3.1.1　状態遷移部

状態遷移部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。モジュールにはＳジェネレータではCurrentState、Ｍジェネレータでは GlobalState と DomesticState 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。図 2 に示したマンデルブロ集合の中間表現をもとに、Ｓジェネレータで生成した状態遷移部を図 3 に、Ｍジェネレータで生成した状態遷移部を図 4 に示す。

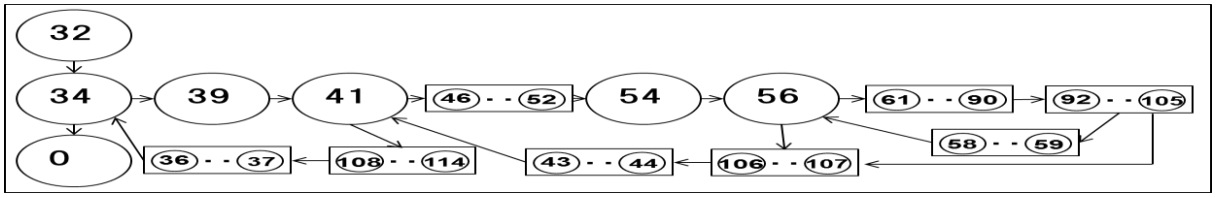


図 3　Ｓジェネレータによるマンデルブロ集合の状態遷移

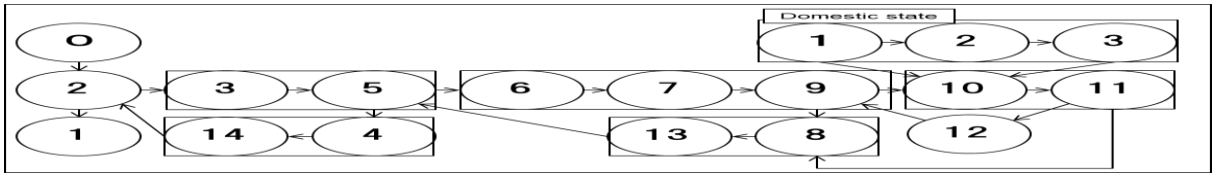


図 4　Ｍジェネレータによるマンデルブロ集合の状態遷移

CurrentState はシンボルテーブルの行数だけ（図 2：シンボルテーブルでは 114 個）、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-#14 の 14 個）、処理別に状態遷移の条件を列挙している。また 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。

3.1.2 演算器部

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 5 に示す。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_STATE36 )? i:
(CurrentState==P_STATE43) ? j;
assign ADD1_B  =  (CurrentState==P_STATE43 )? ConstNum36:
(CurrentState==P_STATE43) ? ConstNum43;
0
```

図 5　S ジェネレータによる演算器部生成例

3.1.3 代入部

代入部の生成もシンボルテーブルから行われる。図 6 (a)に S ジェネレータの、 図 6 (b)にMジェネレータの代入部生成例を示す。

Ｓジェネレータでは変数用のレジスタ、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。Mジェネレータでは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。

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 または、GrobalState を case 文により参照し、各ステート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。

またMジェネレータでは、一状態複数演算を実装しており、同ステートに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 6 (b)の例の場合は変数 yn1 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;
   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;
   c <= 0;
   REG0 <= 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_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
```

(a)　S ジェネレータ

```
always @ (posedge CLK or negedge XRST) 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_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_STATE14 : begin
          i <= i + 1;
      end
      G_STATE15 : begin
          c <= c + 1;
      end
      default : oEND <= 1'b0;
    endcase
  end
```

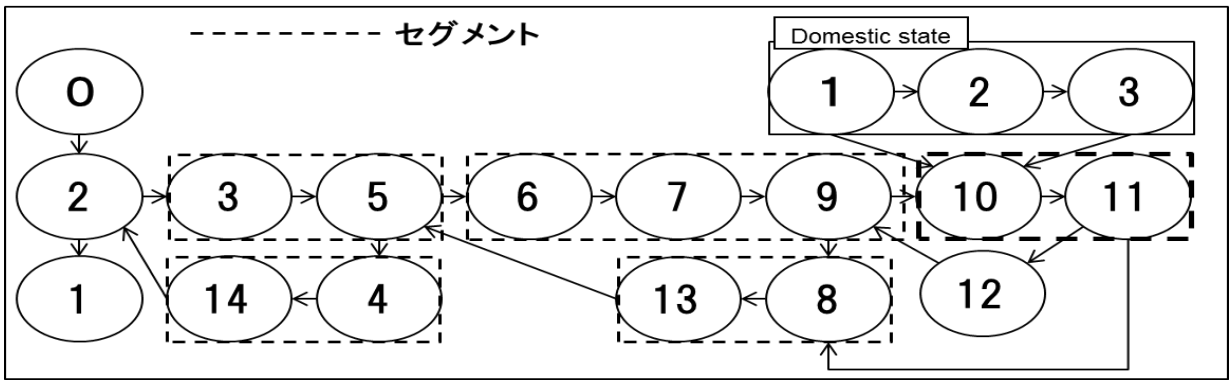
(b)　M ジェネレータ

図 6　マンデルブロ集合の代入部

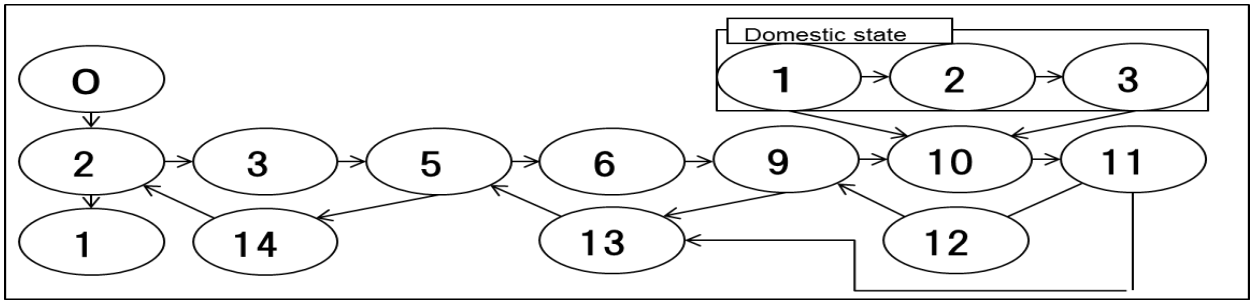
3.2　状態遷移数の削減

Ｓジェネレータでは、１状態で１演算のみを行うため、データ依存の計算は行われていない。ステートマシンは中間表現に記述されているシンボルテーブルの行数だけ作成される。Mジェネレータは１状態複数演算を実装しており、データ依存の計算を行っている。中間表現に記述されている状態遷移表の行数だけ、先にステートマシンを作成し、各処理を割り当てる。同ステートマシン内でデータ依存があれば、別途ステートマシンを作成する。しかし、これでは必要最低数のステートマシンが状態遷移表の行数以下の場合、データ依存のない計算が別々のステートマシンに残ってしまい、データ依存の計算が十分に行えない。

Ｌジェネレータでは、分岐するまでの連続した状態遷移を１つのセグメントという単位でとらえ、データ依存を計算している。セグメント単位に状態遷移数を削減後、セグメント内の依存関係を計算し、ステートマシンを作成してから各演算を割り振っている。図 7 にマンデルブロ集合におけるMジェネレータの状態遷移をもとにした、セグメント単位の状態遷移数削減例を示す。図 7(a)は状態遷移数削減前で、Mジェネレータの状態遷移である。図 7(b)は状態遷移数削減後で、Ｌジェネレータの状態遷移と同じである。



(a)　状態遷移数削減前



(b)　状態遷移数削減後

図 7　データ依存解析による状態遷移数の削減

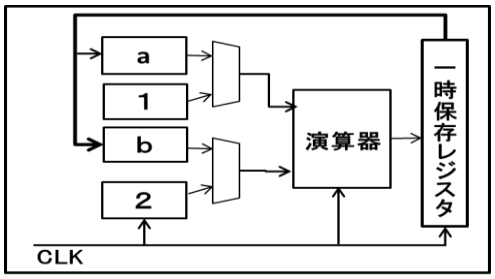
図７において、状態２、４、１４を１セグメントとして扱っている。状態４ではｂ = ｂ - bit_size、 ａ = -2 <<< 16、状態１４ではｉ = ｉ + １という演算を行っており、各演算にデータ依存がない為、同時に実行を行う事ができる。しかし状態２ではｉ = ｉ + １という演算を行っており、状態１４とデータ依存があるため、同時に実行できない。同様の判定を全セグメントに行い、状態遷移数を削減する。

3.3 演算器部の作成

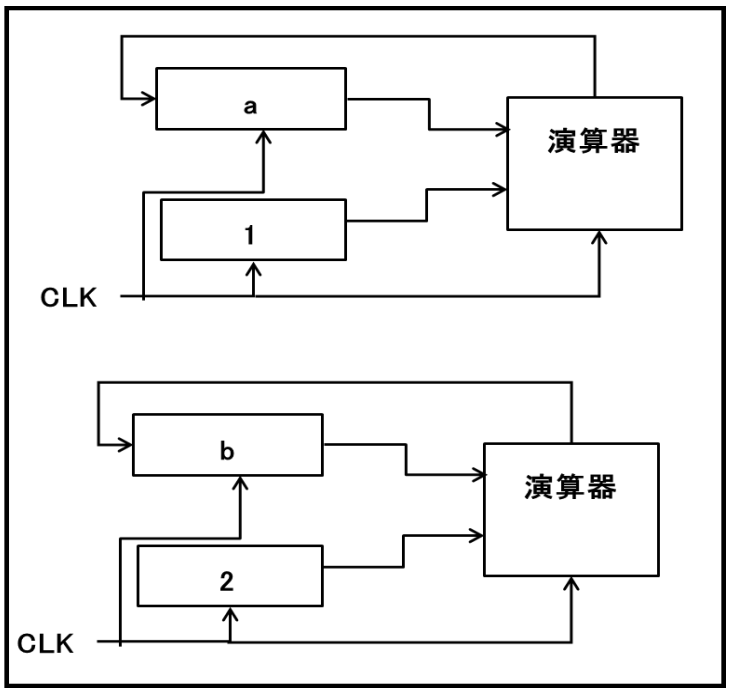
Ｌジェネレータでは状態遷移数を削減し、実行時間の短縮を目指した、状態遷移数を削減することで１状態の演算量が増えたため、演算器部を作成し、処理に必要な量の演算器を実装した。また図 8 (b)に示すように、従来のコードジェネレータでは演算後の値を保持する、一時保存レジスタがあったが、Ｌジェネレータでは演算後の値を直接書き込むことで、回路規模を抑え、遅延時間を少なくしている。

```
case(GlobalState)
G_STATE1.2: begin
a <= a + 1;
b <= b + 2;
end
```

(a)　一状態の演算例



(b)　従来の演算器



(c)　Ｌジェネレータでの演算器

図 8　演算器の生成

3.4 Ｌジェネレータの構成

Ｌジェネレータの構成を図 9 に示す。

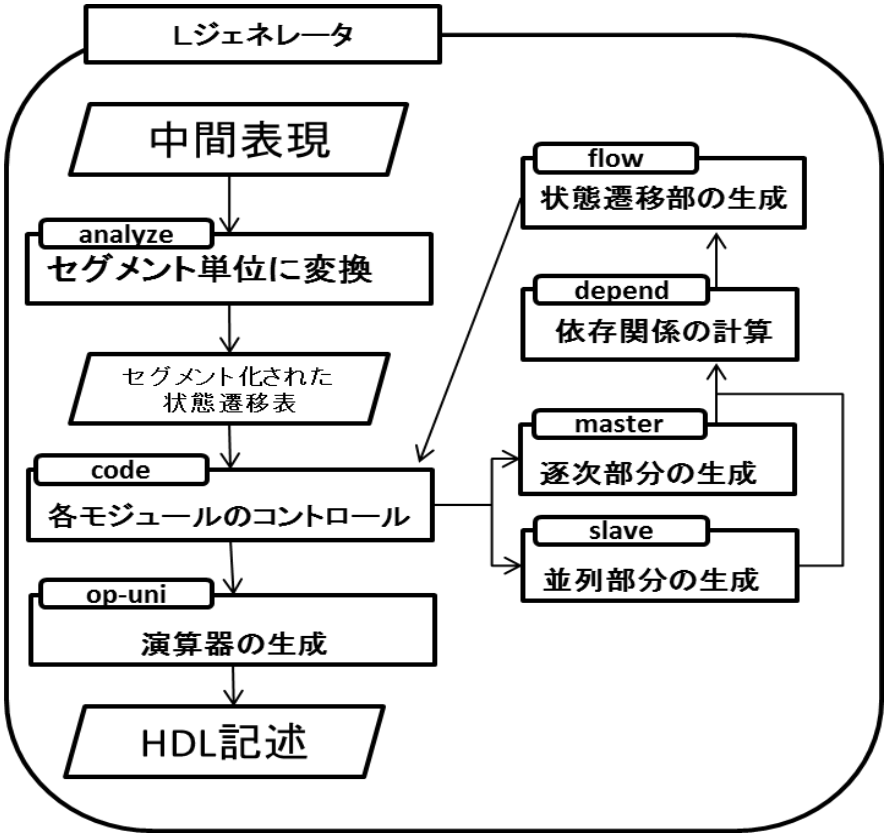


図 9　Ｌジェネレータの構成

状態遷移数の削減は、analyze モジュールにより、状態遷移表をセグメント単位に変換後、depennd モジュールによりデータ依存を計算し。ステートマシンを作成する事で行う。演算器の実装はステートマシン、代入部の作成後、動作に必要な演算器を計算し、実装する。以下にモジュールの説明を示す。

analyze：中間表現を読み込み、状態遷移表をセグメント単位に変換する。**code**：analyze、depend から受け取った値をもとに、master、slave に各要素、値を振り分ける。
master：code から受け取った値を基に depend を呼び出し、マスター部を記述する。
slave：code から受け取った値を基に depend を呼び出し、スレーブ部を記述する。
depend：セグメント内の依存関係を調べ、全体の状態数、各状態での処理を決定する。
flow：depend から全体の状態数、各状態での処理を受け取り、状態遷移の順番を決め、ステートマシンを記述する。
op-uni：記述されたコードをもとに、ハードウェアで使用する全ての演算器を作成する。

4. 各例題に対するハードウェア自動生成

4.1 マンデルブロ集合

4.1.1 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。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。

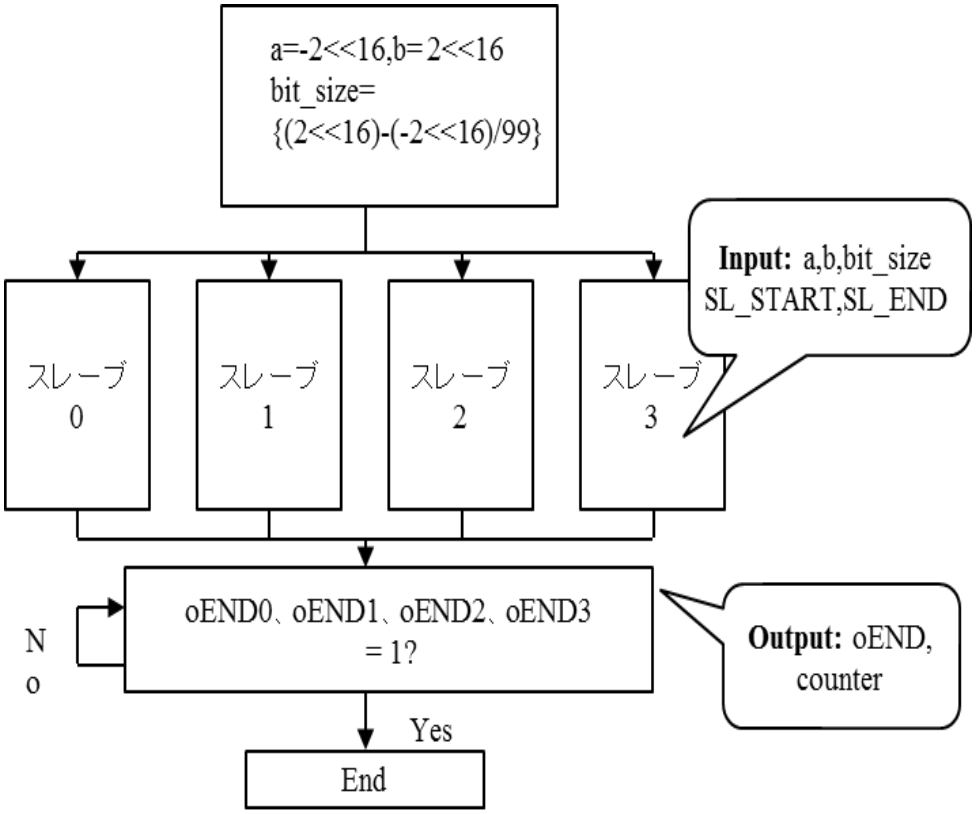


図 10　マンデルブロ集合：マスター部のフローチャート

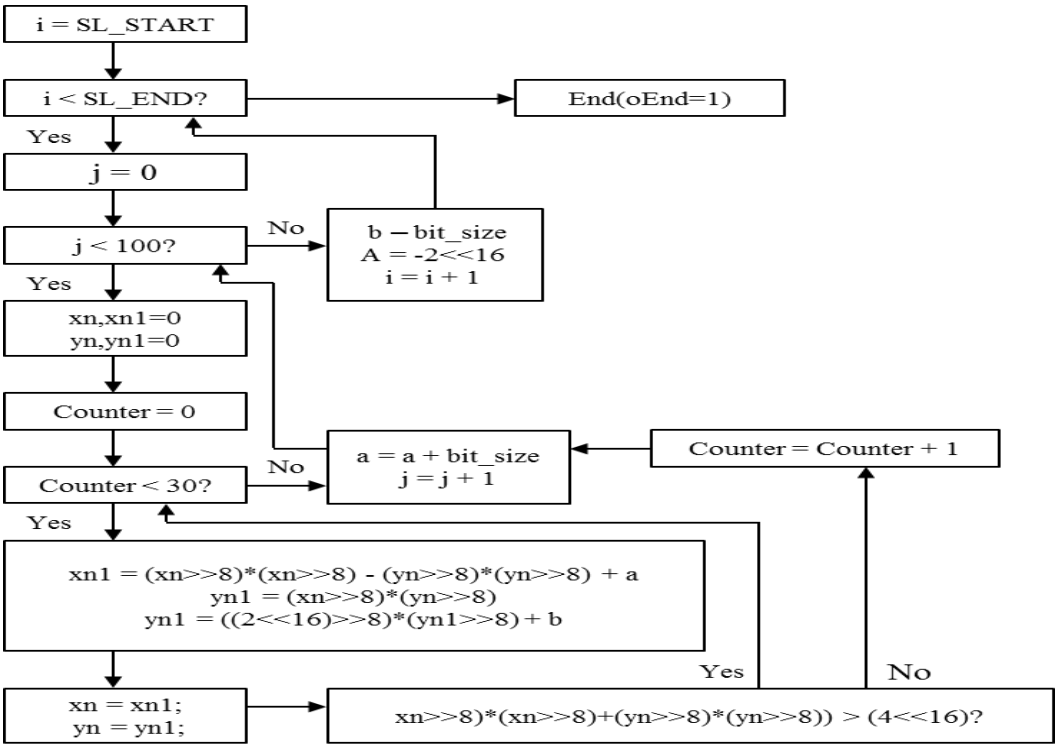


図 11　マンデルブロ集合：スレーブ部のフローチャート

4.1.2　各生成回路の比較

検証方法として、各コードジェネレータの生成回路の回路規模と実行クロック数、最大動作周波数、実行時間の測定を行った。表 1 に結果を示す

表 1　マンデルブロ集合の回路規模と実行時間

コードジェネレータ	S	M	L
演算器数	7	16	16
回路規模(Slice)	4505	1117	1101
実行クロック数(clocks)	788,476	197,018	168,764
クロック減少比(%)	-78.6	-14.4	0
最大動作周波数(MHz)	92.257	93.292	93.292
実行時間(ms)	8.55	2.11	1.80
実行時間減少比(%)	-79	-14.7	0

演算器数を比較すると S ジェネレータが一番少なく、M、L ジェネレータが同数となった。演算器部を実装したのにもかかわらず、演算器数が変わらなかった原因は、ISE シミュレータによって論理合成時に最適化が行われている為だと考えられる。しかし演算器部を作成した事により、生成される演算器を操作できるというメリットはあると考える。例え

ば演算器数の制約を与え、回路規模を小さくする、あるいは生成したい演算器の構成を指定する事が出来ると考える。

回路規模を比較するとMジェネレータ、Ｌジェネレータが同程度となり、Ｓジェネレータはその約４倍となった。Ｓジェネレータの回路規模が大きくなったのは、一時値保存レジスタを使用している点とステートマシンを多く使用するため、配線が多くなる点が理由として挙げられる。ＬジェネレータがMジェネレータより若干小規模になった理由としては、演算器数が同じであるが、ステートマシンが少ない事が効いていると考える。

実行クロック数を比較すると、マンデルブロ集合ではＬジェネレータの実行クロック数を、Ｓジェネレータと比べて約 79.6%、Ｍジェネレータと比べて約 14.4%削減できた。この理由は、依存関係のない演算を同時に行うことで、状態遷移数を減らし、少ないステートマシンで実行できたためである。

また最大動作周波数ではＳジェネレータが１番低く、ＭジェネレータとＬジェネレータは同じとなった。最大動作周波数は１クロックの処理内容や回路遅延に関連する。Ｓジェネレータの動作周波数が低くなった原因は、回路規模増大により、配線が多く、長くなったために回路遅延が起こったと考えられる。ＭジェネレータとＬジェネレータが同じ周波数となったのは回路規模が同程度であり、ボトルネックとなる処理が同じであったためと考えられる。

実行時間を比較すると、マンデルブロ集合ではＬジェネレータが、Ｓジェネレータと比べて約 79%、Ｍジェネレータと比べて約 14.7%削減できた。最大動作周波数では、ＭジェネレータとＬジェネレータは同じ周波数であったが、実行クロック数でＬジェネレータが優秀なため、実行時間はＬジェネレータの方が短くなったと考えられる。

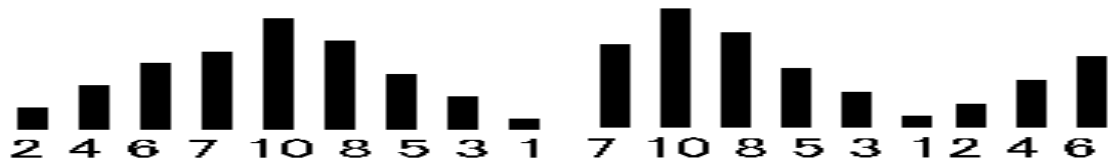
4.2　バイトニックソート

4.2.1　バイトニックソートのアルゴリズム

列<X1,X2,X3,...,X2n>が次の条件のうち、いずれかをみたすとき、バイトニック列であるといえる。

条件(1):X1≦X2...Xj,...,≧Ｘ2n(1≦j≦2)

条件(2):列を巡回シフトする事により、条件(1)を満たす列



条件(1)　　条件(2)

図 12　バイトニック列

また二つの基本性質がある。

<a1,a2,...a2n>がバイトニック列であり、部分列<d1,d2,...,dn>,<e1,e2,...,en>が

di = min<a1,an+1>, ei = max(ai,an+1), 1≦i≦n のとき

(1)<d1,d2,...,dn>,<e1,e2,...,en>はバイトニック列である。

(2)max(d1,d2,...,dn)≦min(e1,e2,...,en)である。

図 13 にバイトニックソートの並列化アルゴリズムを示す。並列化は基本性質(2)を使用して行う。まず各スレーブに値を渡して昇順、降順と交互にソートする。そしてソートできた列を昇順と降順で組み合わせて複数のバイトニック列をつくる。列を値の大きいもの、小さいものに分け、再びソート、バイトニック列作成を行う。これを全スレーブのデータを組み合わせるまで行い、最後に各スレーブで昇順ソートを行うとバイトニックソートが完了する。基本性質から、各組合せについて小さい値、大きい値がわけられ、それを繰り返す事でソートを実現している。

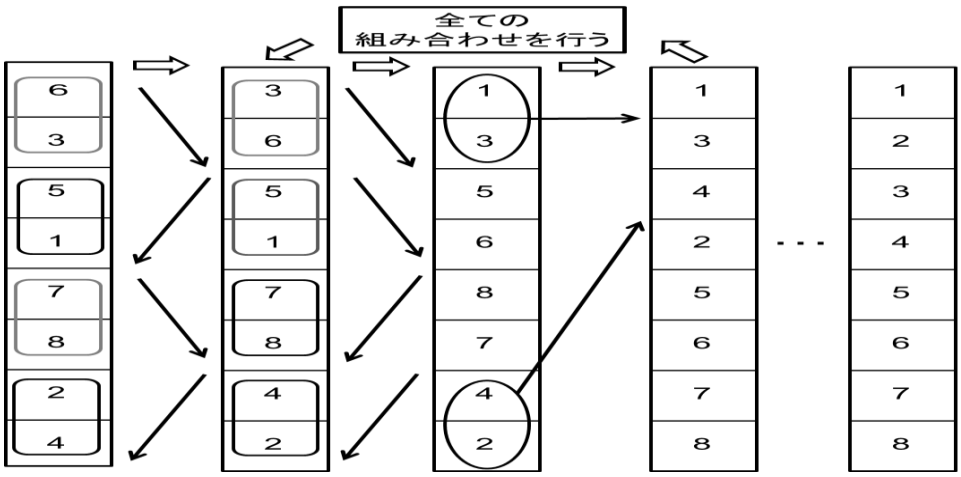


図 13　バイトニックソートの並列化

4.2.2　各生成回路の比較

検証方法は 4.1.2 と同様に、各コードジェネレータの生成回路の回路規模と実行クロック数、最大動作周波数、実行時間の測定を行う。表 2 に結果を示す

表 2　バイトニックソートの回路規模と実行時間

コードジェネレータ	S	M	L
演算器数	7	11	11
回路規模(Slices)	14613	14048	14067
実行クロック数(clocks)	2234	1618	1040
クロック減少比(%)	-53.5	-35.8	0
最大動作周波数(MHz)	149.813	144.950	149.809
実行時間(ns)	15	11.2	6.94
実行時間減少比(%)	-53.8	-38.1	0

演算器数を比較すると S ジェネレータが一番少なく、M、L ジェネレータが同数となった。演算器部を実装したのにもかかわらず、演算器数が変わらなかった原因は、4.1.11 と同様に ISE シミュレータによる最適化が原因だと考えられる。

回路規模を比較すると、Sジェネレータが少し大きく、Mジェネレータ、Lジェネレータが同程度となった。Sジェネレータの回路規模が大きくなった原因は、4.1.2 と同様に、一時値保存レジスタを使用している点とステートマシンを多く使用する点があげられる。しかしバイトニックソートでは、マスター部とスレーブ部で値のやり取りがあるため、メモリアクセス制御の記述を書き加えている。そのため、差があまり出なかったと考える。

実行クロック数を比較すると、マンデルブロ集合ではLジェネレータの実行クロック数を、S ジェネレータと比べて約 53%、M ジェネレータと比べて約 36%削減できた。依存関係のない演算を同時に行うことで、状態遷移数を減らし、少ないステートマシンで実行できたためである。

最大動作周波数ではM ジェネレータが 1 番低く、S ジェネレータ、L ジェネレータという順になった。Mジェネレータの動作周波数が低くなったのは、状態遷移の中で、なんらかの処理がボトルネックになっていると考える。

実行時間を比較すると、マンデルブロ集合ではL ジェネレータが、S ジェネレータと比べて約 54%、M ジェネレータと比べて約 38%削減できた。実行クロック数でL ジェネレータが優秀なため、実行時間が短くなったと考えられる。

4.3 Radix（基数）ソート

4.3.1 Radix（基数）のアルゴリズム

Radix sort はデータの種類が有限で、最大値・最小値がはっきりしている必要があるが、高速かつ安定ソートである。 基数（radix）とは、10 進数の 10、16 進数の 16 のように、桁上がりの基準になる数である。基数ソートでは、基数の数だけバケツを用意し、値ごとに 1 個のバケツを対応づける。元のデータ列を桁ごとに走査し、各データを対応するバケツに入れる。

```
170, 45, 75, 90, 2, 24, 802, 66
```

という数列を 1 の位についてソートすると（1 の位が 0、1・・9 のものをまとめていく）

```
170, 90, 2, 802, 24, 45, 75, 66
```

となる。さらに、10 の位についてソートすると、

```
2, 802, 24, 45, 66, 170, 75, 90
```

となる。最後に、100 の位についてソートすると、

```
2, 24, 45, 66, 75, 90, 170, 802
```

となり、ソートが完了する。並列手法としては、各ノードに基数を割り当てる事で並列化を行う。基数毎に一致する値を探すため。並列化できる。例えばノード 2 の場合、0～4 と 5～9 に基数を分け、ソートを行う。図 14 に Radix sort の並列化手法を示す。

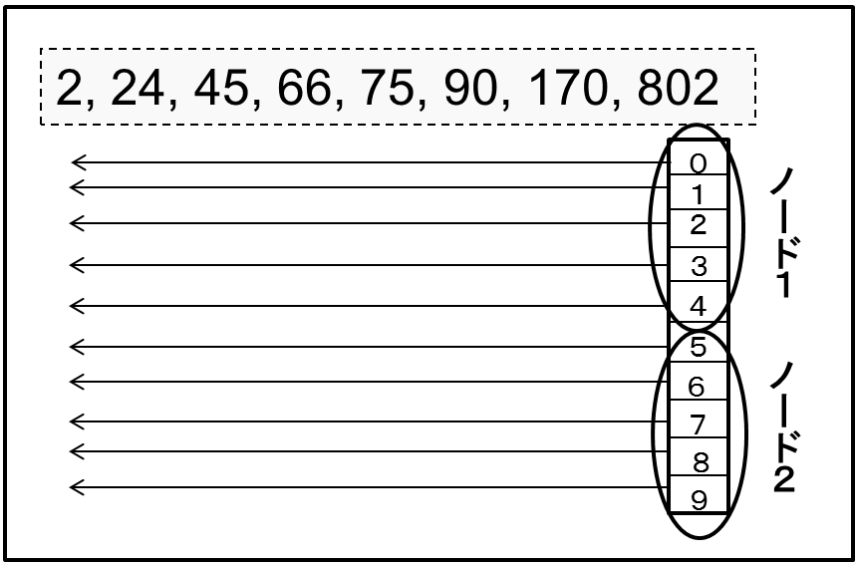


図 14　Radix sort の並列化

4.3.3　各生成回路の比較

検証方法は 4.1.2 と同様に、各コードジェネレータの生成回路の回路規模と実行クロック数、最大動作周波数、実行時間の測定を行う。表 3 に結果を示す

表 3　Radix Sort の回路規模と実行時間

コードジェネレータ	S	M	L
演算器数	6	12	12
回路規模(Slices)	24357	23001	22995
実行クロック数(clocks)	53,584	51,612	51,372
クロック減少比(%)	-4.2	-0.05	0
最大動作周波数(MHz)	121.587	121,687	121,676
実行時間(ms)	0.44	0.42	0.42
実行時間減少比(%)	-4.6	0	0

演算器数を比較すると S ジェネレータが一番少なく、M、L ジェネレータが同数となった。演算器部を実装したのにもかかわらず、演算器数が変わらなかった原因は、4.1.11 と同様に ISE シミュレータによる最適化が原因だと考えられる。

回路規模を比較するとＳジェネレータ、Ｍジェネレータ、Ｌジェネレータがほぼ同程度となった。また実行クロック数、実行時間も同様に、差が出ない結果となった。Radix Sort の OpenMP プログラムでは基数を求める際、除算、剰余を使用している。しかし、HDL では、除算、剰余は使用出来ないため、引き算を繰り返し利用するアルゴリズムを手書きで全ジェネレータに書き加えている。この演算が処理の大部分を占めているため、結果に差が出なかったと考える。表 4 に除算、剰余を使用した Radix Sort の測定結果を示す。

表 4　除算、剰余を使用した Radix Sort の実行時間

コードジェネレータ	S	M	L
実行クロック数(clocks)	2761	1775	1415
クロック減少比(%)	-49	-21	0
想定実行時間(ns)	22.7	14.6	11.6
実行時間減少比(%)	-49	-21	0

除算、剰余の代用を行わない場合では、Ｌジェネレータの実行クロック数をＳジェネレータと比較して 21%、Ｓジェネレータと比較して 49%削減できた。除算、剰余を用いた場合、論理合成が行えないため、動作周波数等を調べる事は出来ないが、表 3 と同様の周波数で動作すると想定すると、実行時間はＬジェネレータがＳジェネレータと比較して 49%、Ｓジェネレータと比較して 21%削減できると考える。

4.4　OpenMP 記述時の考慮点とメモリアクセス制御信号の設計

4.4.1　　OpenMP 記述時の考慮点

本システムは並列ハードウェア生成をＣ言語ベースの入力で実現する事で、並列ハードウェア設計をより多くの人に、より簡単に行えることを目的にしている。実際にＣ言語でかかれたプログラムの並列化を行える箇所に、#pragma omp parallel for という記述を１行足すだけで、マンデルブロ集合などのハードウェア生成を実現している。

一方で fork-join モデルのハードウェアを想定している本システムでは、ハードウェア化後を想定して設計を行うことが望ましい場合もある。例えば、図 15 に示すバイトニックソートでは、それぞれの値交換のみを並列で行っている。しかしこれでは、マスター部とスレーブ部でのデータのやりとりに時間がかかってしまい、かえって処理が遅くなる。4.2.1 のように、ソートをスレーブ部で行い、マスター部でデータの連結を行うなど、スレーブ部での処理を増やすことで、並列性を上げることができる。

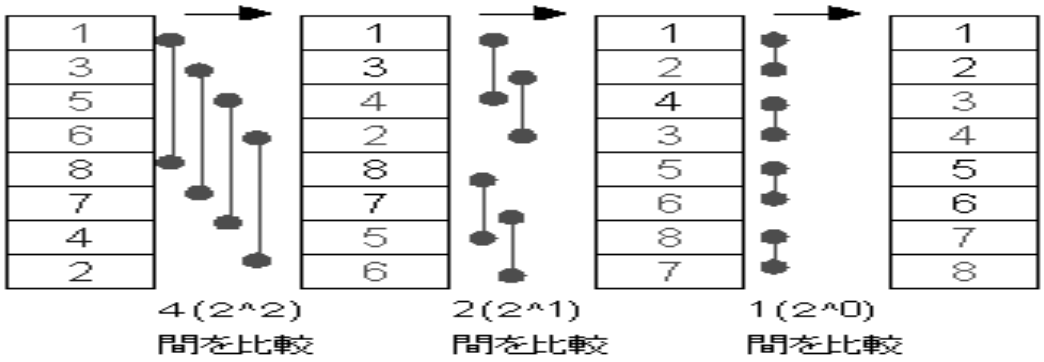


図 15　ハードウェア化を考慮しない並列化

4.4.2　メモリアクセス制御信号

バイトニックソート、基数ソートでは、各スレーブの実行後のデータをマスター部で処理し、再度スレーブ部で実行している。そのためマスター部とスレーブ部でデータのやり取りがあり、メモリ書き込みのタイミングなどを制御する必要があった。図 16 に今回手書きで書きだしたメモリアクセス制御を示す。

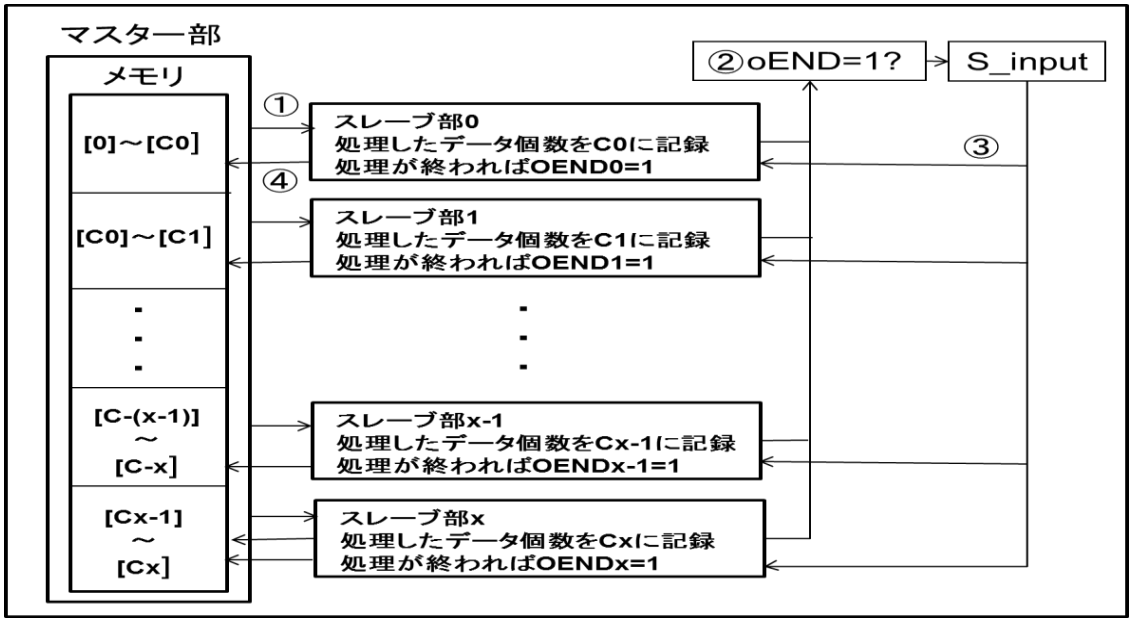


図 16　メモリアクセス制御

マスター部で処理後、①でスレーブ部に担当するメモリの内容を送る。スレーブ部では処理を行いながら、処理した個数をカウントする。処理が終了すれば oEND に１を入れ、終了フラグを立てる。②にてマスター部はすべてのスレーブ部が終了フラグを立てるまで待機し、すべてのフラグが終了すれば、③にてＳ_input に１を入れ、メモリアクセスフラグを立てる。メモリアクセスフラグがたてば、④にて処理したデータの個数だけ、スレーブ部からマスター部にデータが送られる。

4.4.3　除算、剰余演算

ＨＤＬ記述では除算、剰余が使用できない。剰余、除算を使用している場合は書き換えが必要である。図 17 に基数ソートで使用したアルゴリズムを示す。

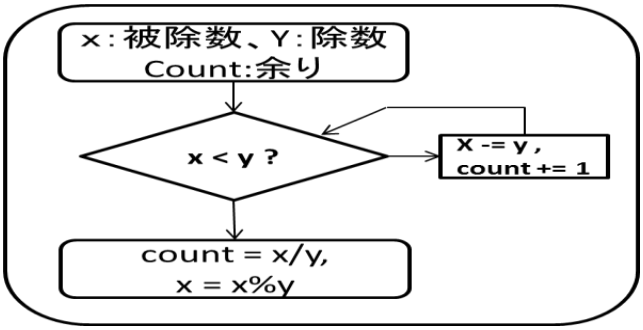


図 17　除算、剰余のアルゴリズム

被除数に対して、減算を繰り返すことで除算、剰余を実現している。しかし 4.3.3 に示したように、このアルゴリズムでは実行時間、回路規模ともに大規模になってしまう。シフトを使用する等、アルゴリズムを工夫する必要がある。

4.5　考察と今後の課題

今回コードジェネレータを状態遷移数の削減と演算器部の作成の２点において改良した。状態遷移数においては、全ての検証にて実現でき、回路の実行時間は既存の２つのコードジェネレータより優秀な結果を示した。演算器部においては作成したものの、ISE の最適化によって効果は確認できなかった。しかし今回作成した演算器部は、OpenMP プログラムに記述された演算子と同じ数だけの演算器の記述を行うものである。これでは回路規模が大きくなるため、最適化を課題と考えていた。ISE によって演算器数が最適化され、回路規模を抑えながら、一番実行時間の短い回路生成できたことはうれしい誤算であった。

次に今後の課題について述べる。課題は(1) メモリアクセス制御の自動生成化(2) 除算、剰余への対応の２点である。

(1) メモリアクセス制御の自動生成化

バイトニックソートや奇数ソートのように、マスター部とスレーブ部でデータやりとりがある場合、タイミング制御を行う必要があり、現在は手書きで書き足している。メモリアクセスの有無を調べ、必要に応じてメモリアクセス制御の記述を行えるよう改良する必要がある。

(2) 除算、剰余への対応

別モジュールで除算器、剰余器を作成しておき、必要に応じてハードウェアに組み込こめるよう改良する必要がある。また除算器、剰余器も、シフト演算などを使用し、実行時間を短くする、または負の数や浮動小数点にも対応させるなどの工夫が必要である。

5. おわりに

本研究では、既存のコードジェネレータの抱えるデータ依存計算が不十分という課題を解決すべく、セグメント単位でのデータ依存計算を行う新たなコードジェネレータを作成し、評価を行なった。実行時間ではＬジェネレータはマンデルブロ集合、バイトニックソートにおいて、既存の２つのコードジェネレータより優秀な結果を示した。奇数ソートにおいては、除算と剰余の書き換えをした場合、若干の実行時間短縮に留まったが、除算と剰余を使用する場合では、他の評価回路と同様に、実行時間の短縮を実現した。　回路規模においては、全ての評価回路において、Ｓジェネレータより小さく、Ｍジェネレータと同程度という結果になった。演算器の一時値保存レジスタを削減することにより、演算器数増加による回路規模増大を抑えられたためである。

今後の課題としては、メモリアクセス制御の自動生成化と除算、剰余への対応があげられる。メモリアクセス制御の自動生成化はメモリアクセスの有無を調べ、4.4.2 に示したメモリアクセス制御の記述を必要に応じて行う事で実現できると考える。除算、剰余への対応は、別モジュールで除算器、剰余器を作成しておき、必要に応じてハードウェアに組み込むことで実現できると考える。その際は、4.4.3に示したアルゴリズムではなく、シフト演算などを使用し、実行時間を短くする、あるいは負の数や浮動小数点にも対応させるなどの工夫が必要である。

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、孟林助手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.

[13] 松本剛史, 斉藤寛, 藤田昌宏, “線形計画法に基づく逐次化を利用したシステムレベル設計での動作並列化前後での等価性検証手法”, 情報処理学会シンポジウム論文集, pp.157-162 , 2006.

[14] Design Wave Magazine, “組み込みソフトと SoC のシステム設計最前線”,CQ 出版社 , pp.83-84 , 2001.

[15] 西川諒, “OpenMP ハードウェア動作合成におけるのコード生成手法の検討”, 立命館理工学部電子情報デザイン学科卒業論文, 2011.

[16] 西川諒, 孟林, 山崎勝弘“OpenMP ハードウェア動作合成におけるのコード生成手法の改良と検証”,情報処理学会　第７４回全国大会, 2012.