卒業論文

OpenMP ハードウェア動作合成システムの検証（Ⅱ）

氏名	:	苅屋 徹
学籍番号	:	2260050094-9
指導教員	:	山崎 勝弘 教授
提出日	:	2009 年 2 月 19 日

立命館大学　理工学部　電子情報デザイン学科

内容梗概

本論文では，OpenMP を用いたハードウェア動作合成システムから生成された回路と，手書きで記述した HDL から生成された回路との比較検証を行い，OpenMP ハードウェア動作合成システムの改良と検証を行っている．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，N の総和，及びマンデルブロ集合プログラムを OpenMP で記述し，本システムを用いて動作合成を行った．また手書きで HDL を記述し生成し，それらのプログラムの高位合成された回路シミュレーションとの比較を行い，コードジェネレータ，及び本システム全体の評価を行っている．N の総和，マンデルブロ集合ともに，SMP 環境上での実行時間が回路シミュレーション時間よりも大幅に小さくなった．N の総和に対しては，データ並列を用いて 4 ノードでおよそ 4 倍の速度向上が得られ，回路規模もノード数に対して同程度の増加を確認した．また，システムよりも手書きの方が実行クロック数，回路規模ともに優れていた．マンデルブロ集合に対しては，データ並列を用いて 4 ノードでおよそ 2.5 倍の速度向上が得られた．回路規模についてはノード数に対して同程度の増加を確認した．また,N の総和とは異なり,回路規模についてはシステムの方が小さくなった.これにより，回路の設計方針によっては，システムの方がコンパクトに回路を生成できることが分かった．

目次

図目次

表目次

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[3-8]．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[9,10]．

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[11]．

本研究では，これらの問題を解決するために，並列プログラミングに使用されるOpenMPを用いたハードウェア動作合成手法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回路と，手書きで記述したHDLにより生成された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]．

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]，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]，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，N の総和プログラムとマンデルブロ集合に対して，システムにより生成された回路と，手書きで生成された回路の比較を行い，システムの改良と検証を行っている．比較を行いやすくするため，手書きによる記述のアルゴリズムは，システムにより生成される回路と同じものとする．それぞれで生成された回路において，HDL 記述の行数やシミュレーションにかかった時間，実行クロック数，そして論理合成においての回路規模を測定した．回路シミュレーション時間は最初に作成した OpenMP を用いたプログラムの SMP 環境での実効時間と比較を行った．システムの改良点としては，より抽象的にプログラム記述ができるように OpenMP のプログラムの記述の制約について，そして生成回路の冗長性の削減と最適化を計るため中間表現から HDL を生成するコードジェネレータに着目し考察した．

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，および中間表現について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 N の総和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 章ではコードジェネレータの改良点を挙げ，本システム全体の評価と考察を行う．

2. OpenMP を用いた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．

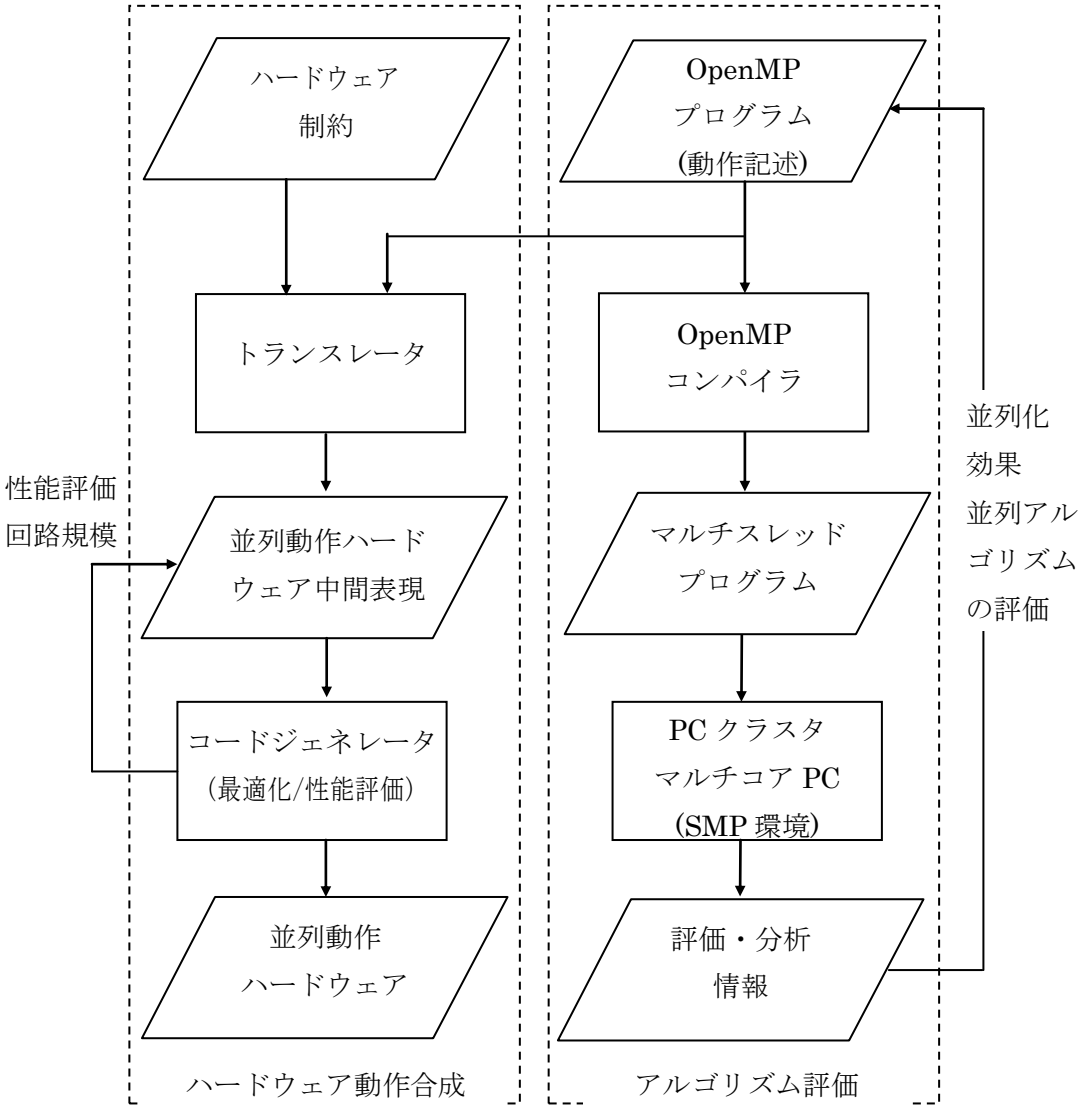

図 1：OpenMP を用いたハードウェア動作合成システム

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，手書きで記述し生成した回路と，本システムを用いて動作合成を行い生成された回路の比較を行っている．

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 ループの条件式である#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．

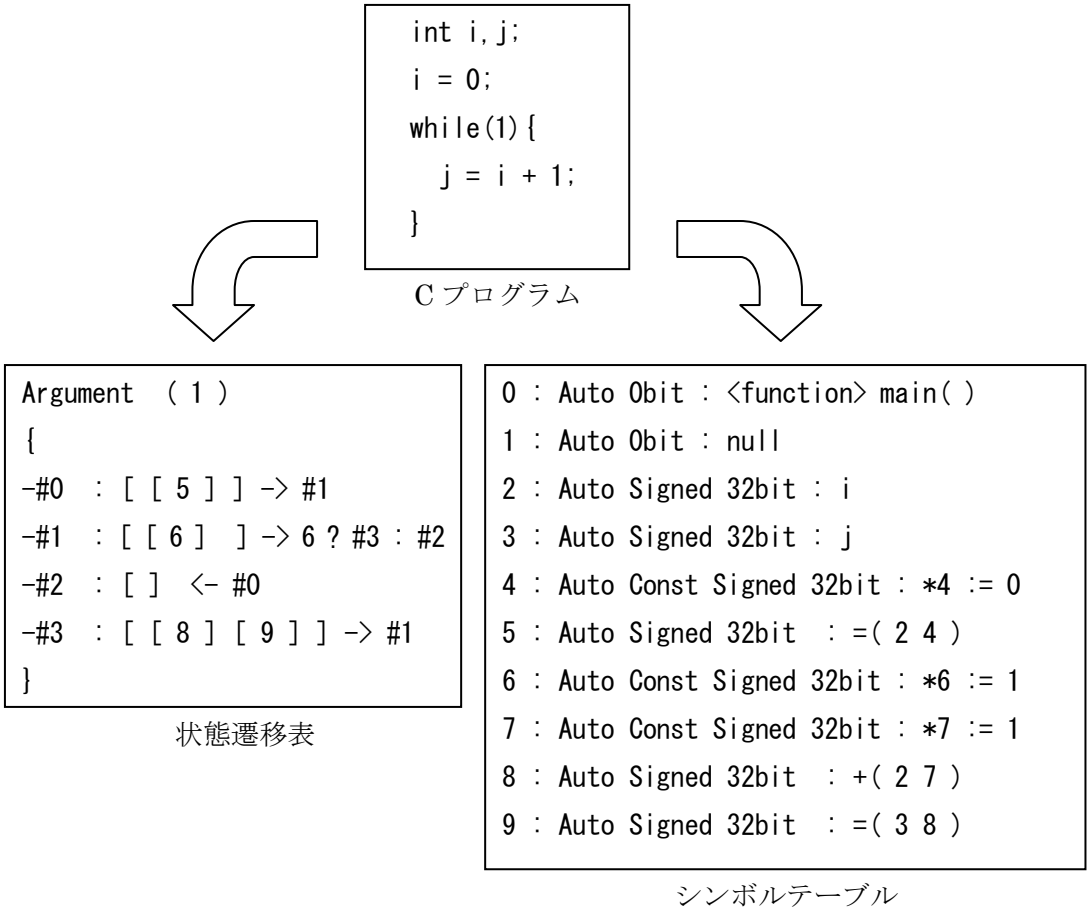

図 2：中間表現の例

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の生成方法について説明する．生成されるハードウェアは演算器部，代入部，状態遷移部からなる．

2.3.1 演算器部

演算器部の生成はシンボルテーブルから行われる．演算器部生成の例を図 3 に示す．例として，変数 i と j に定数である 1 を加算する演算を示している．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する．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(配線)で宣言する．オペランドに対し，assign 文を用いて組み合わせ回路を表現する．assign 文の条件式の部分には現在の状態を表すレジスタ(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．図 4 に演算器部のハードウェアモデルを示す．

```
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：演算器部生成の例

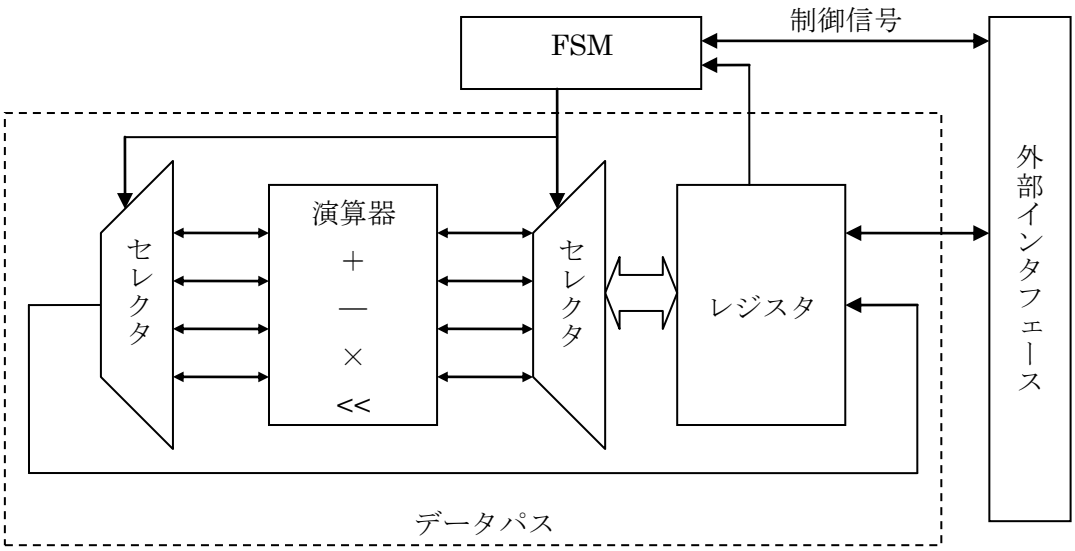

図 4：演算器部のハードウェアモデル

2.3.2 代入部

代入部の生成は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 5 の例では，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 文で記述する．always 文の最初の if(!XRST)は非同期リセットであり，基本的にこの部分でレジスタの初期化を記述しなければならない．記述がない場合は，論理合成ツールで合成エラーになる可能性がある．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 文の参照を現在の状態を表すレジスタ(CurrentState)にし，状態遷移の条件を列挙していく．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 線，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 XRST) begin
 if(!XRST) begin
   i <= 32'd0;
   ...
 end else begin
   case(CurrentState)
     P_INIT    : oEND <= 1'b0;
     P_STATE8  : REG8 <= ADD1_RESULT;
     P_STATE9  : j    <= REG8;
     ...
   endcase
 end
end
```

図 5：代入部の例

2.3.3 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から行われる．状態遷移コードの例を図 6 に示す．例は図 2 で示した無限ループの C プログラムから生成した場合を示している．例で示す状態遷移表は，#0 から始まり，”[ ]”で囲まれた状態番号の演算を順に行い，”->”で示す次の状態に遷移する．状態遷移ではシンボルテーブルと状態遷移表を利用する．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．例では，P_INIT で示される初期状態から#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，無限ループでえあるため，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 で示される演算のである[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 XRST) 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 6：状態遷移部生成の例

3. Nの総和に対するハードウェア動作合成

本章では，OpenMP で書かれた N の総和のプログラムに対して，本システムを用いてシミュレーションと動作合成を行った実験結果を示す．得られた結果に対し考察を行い，コードジェネレータ，及び本システム全体の評価を行う．

動作合成システムの実験環境として，アルゴリズム評価系，ハードウェア動作合成系，回路シミュレーション時間の比較として用いた環境を表 1 に示す．

表 1：実験環境

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 9.2i
回路シミュレーション	PC 環境	Intel Core2 duo 2.66GHz,Memory 3GB
	シミュレーションツール	ModelSim SE 6.3c

3.1 Nの総和のアルゴリズム

加法が定義された集合 M の元の列 x_1, x_2, ..., x_n に対する n 項演算（n は順序数）である．それは，帰納的に次のように定義される．

- $s_1 = x_1$.
- $s_i = s_{i-1} + x_i$.

こうして得られる s_i は各々が部分和と呼ばれる（一般には，部分列を取り出して和をとったものを総じて部分和と呼ぶ).n が有限であれば，この操作は有限回で終了し，x_1, x_2, ..., x_n の総和は s_n に等しい．これを

$$S_n = \sum_{i=1}^{n} x_i$$

と記す．記号 Σ は，ギリシャ文字のシグマである．これは，Sum（和）の頭文字 S に相当するものである．また，Σ の上下の添字は，添え字 i を 1 よりはじめて n まで動かすことを表す．総和は，線型性を持つ演算である．

3.2 手書きとシステムによる HDL 記述の生成

図 7 に OpenMP プログラム，図 8 に中間表現，図 9 に手書きの HDL，図 10 にシステムの HDL の抜粋を示す．手書きによる HDL 記述は約 26 行であるのに対し，システムによる HDL 記述は約 118 行と約 4.5 倍の量になった．システムの方では専用のパラメータが宣言されており，演算基部，代入部，状態遷移部それぞれを分けて記述される．また，1 状態につき 1 演算行っている．手書きの方では，演算と代入を同時に記述されており，1 クロックで複数の処理が可能になるように記述されている．また，手書きでは専用のパラメータを利用していないこともあり，システムで生成されるものより，少ない行数で記述される．

演算器部では，状態 8 と状態 9 で加算が行われているので，加算器が生成されている．状態 8 では”(2)++”が行われる．状態 2 は”i”であり，”++”はインクリメントを表している．ゆえに P_STATE8 では”ADD1_A=i”，”ADD1_B=1”となり”ADD1_RESULT”はそれらの計算結果を表すことになる．状態 9 では”+(3 2)”が行われる．状態 3 は”n”であり，状態 2 は”i”を示している．ゆえに P_STATE9 では”ADD1_A=n”，”ADD1_B=i”となり”ADD1_RESULT”はそれらの計算結果を表すことになる．

代入部では，状態 5 で”=(2 4)”が行われる．状態 2 は”i”であり，状態 4 は定数 0 である”ConstNum4”を示している．また”=”は代入を表しているので P_STATE5 で”i<=ConstNum4”が生成される．状態 8 では”(2)++”が行われる．状態 2 は”i”であり，”++”はインクリメントを表している．ゆえに P_STATE8 では”i<=ADD1_RESULT”が生成される．他の代入も同様に行われる．

状態遷移部では，P_INIT で示される初期状態から#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，#2 に示されるループの状態分岐判定を行う[7]に移動するため P_STATE7 に遷移する．P_STATE7 では条件分岐を行う．条件式は”i<ConstNum6”となっており，これが真の時，#3 で示される演算のである[9]と次の[10]の状態へ遷移し，#4 で示される演算である[8]を行った後に#2 のループの最初に遷移する．

```
#define N 100

void main(void)
{
  int i;
  int sum=0;

  #pragma omp parallel for
  for( i=0 ; i<N ; i++ )
  {
    sum += i;
  }
}
```

図 7：N の総和の OpenMP プログラム

```
----SemanticsAnalyze----
function 0 : main
0 : Auto 0bit : <function> main( )
1 : Auto 0bit :: null
2 : Auto Signed 32bit :: i
3 : Auto Signed 32bit :: n := 0
4 : Auto Const Signed 32bit :: *4 := 0
5 : Auto Signed 32bit  : =( 2 4 )
6 : Auto Const Signed 32bit :: *6 := 100
7 : Auto Signed 32bit  : <( 2 6 )
8 : Auto Signed 32bit  : ( 2 )++
9 : Auto Signed 32bit  : +( 3 2 )
10 : Auto Signed 32bit  : =( 3 9 )
Argument  ( 1 )
{
--#0  : { /0 } -> #1
--#1  : [ ]  <- #0
}
/0 Parallel FOR (2) ( P[ 2 ]  )
-0:#0  : [ [ 5 ] ] -> #2
-0:#1  : [ ]  <- #0
-0:#2  : [ [ 7 ] ] -> 7 ? #3 : #1
-0:#3  : [ [ 9 ] [ 10 ] ] -> #4
-0:#4  : [ [ 8 ] ] -> #2
```

図 8：N の総和の中間表現

```
module sum( n_start, n_end, n, ef, clk, rst);
        input   [31:0]  n_start, n_end;
        wire    [31:0]  n_start, n_end;
        output  [31:0]  n;
        reg             [31:0]  n;
        output  ef;
        reg             ef;
        input   clk,rst;
        wire    clk,rst;
        reg             [31:0] i;

always @ (posedge clk or negedge rst) begin
        if(!rst) begin
                n = n_start;
                ef= 1;
                i = 0;
        end else begin
                i = i + 1;
                if( i<=n_end ) begin
                        n = n + i;
                end else begin
                        ef = 0;
                end
        end
end
endmodule
```

図 9：N の総和の手書きの HDL

```
//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_STATE8) ? i :
                (CurrentState==P_STATE9) ? n : n;
assign ADD1_B = (CurrentState==P_STATE8) ? 32'd1 :
                (CurrentState==P_STATE9) ? i : i;

//代入部
always @ (posedge CLK or negedge XRST) begin
 if(!XRST) begin                                    // レジスタの初期化
   oEND <= 1'b0;
   i <= 32'd0;
   n <= 32'd0;
   REG9 <= 32'd0;
 end else begin
  case(CurrentState)
    P_INIT : oEND <= 1'b0;
    P_END  : oEND <= 1'b1;
    P_STATE5 : i <= ConstNum4;                      // i = 0
    P_STATE8 : i <= ADD1_RESULT;                    // i = i + 1
    P_STATE9 : REG9 <= ADD1_RESULT;                 // i = i + 1
    P_STATE10 : n <= REG9;                          // n = n + i
    default : oEND <= 1'b0;
  endcase
 end
end

//状態遷移部
always @(posedge CLK or negedge XRST) begin
 if(!XRST)
  CurrentState <= P_INIT;                                   // レジスタの初期化
 else
  case(CurrentState)
    P_STATE5: CurrentState <= P_STATE7;                     // i = 0
    P_INIT   : if(iSTART==1'b1) CurrentState <= P_STATE5;
               else CurrentState <= CurrentState;
    P_END    : CurrentState <= P_INIT;
    P_STATE7: if(i<ConstNum6) CurrentState <= P_STATE9;     // i<=n_end
               else CurrentState <= P_END;
    P_STATE9: CurrentState <= P_STATE10;                    // i = i + 1
    P_STATE10: CurrentState <= P_STATE8;                    // n = n + i
    P_STATE8: CurrentState <= P_STATE7;                     // i = i + 1
    default : CurrentState <= CurrentState;
  endcase
end
```

図 10：N の総和のシステムの HDL（抜粋）

手書きの FSM は図 9 の always ブロックより，図 11 のようになるのが分かる．システムの FSM は図 8 の状態遷移表より,図 12 のようになるのが分かる.番号は状態を示している．状態は 5 から始まり，状態 8 まで遷移すると状態 5 に戻り，繰り返し遷移する．

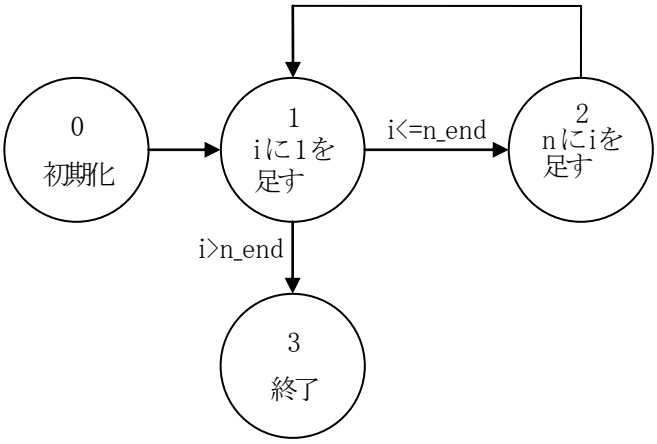

図 11：N の総和の手書きの FSM

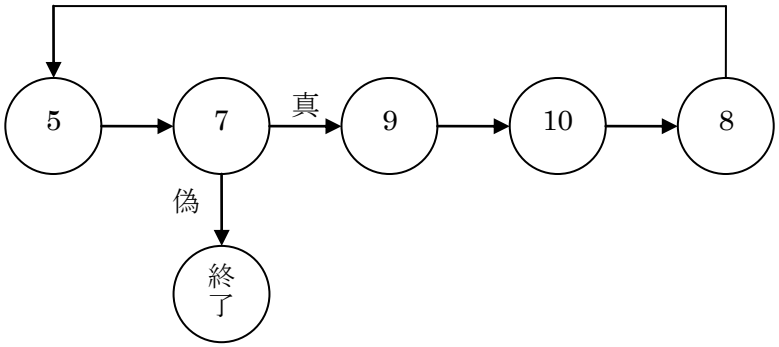

図 12：N の総和のシステムの FSM

3.3 手書きとシステムによる生成回路の比較

本研究では 1 から 100 までの総和の計算を行った．手書きとシステムで生成した回路の実行クロック数を表 2 に示す．スレッド数が 1 の場合は逐次実行を示している．スレッド数が 2 の場合と 4 の場合，ともにスレッド数の増加に対して理想的にクロック数が減少した．実行クロック数の減少が並列数に対して理想的な理由として，N の総和はほとんど分岐がない規則的な処理であり，また各ノードにが他ノードに依存しない処理であるため，ストールなどを起こすことなく理想的な並列処理が行われたと考えられる．

システムの方が手書きよりも実行クロック数が約 4 倍以上かかっている．理由として手書きは 1 クロックで複数の演算を行っているが，システムは 1 クロックに 1 演算しか行っていないことが原因だと考えられる．

表 2：N の総和の実行クロック数

スレッド数		1	2	4
実行クロック数 (cloks)	手書き	200	100	50
	システム	817	417	217
クロック減少比	手書き	1.00	2.00	4.00
	システム	1.00	1.96	3.76

手書きとシステムで生成した回路の回路規模を表 3 に示す．どちらもスレッド数の増加に対して，同程度の回路面積比の増加が確認できる．

システムの方が手書きよりも回路規模が 2 倍以上大きい．理由として手書きは必要最低限の回路しか行っていないが，システムはパラメータ生成，演算基部，代入部，状態遷移表などの回路が必要であるため，手書きでは必要のない回路を生成していることが原因だと考えられる．

表 3：N の総和の回路規模

スレッド数		1	2	4
回路規模 (Slices)	手書き	39	77	155
	システム	105	187	375
回路面積比	手書き	1.00	1.97	3.97
	システム	1.00	1.78	3.57

SMP クラスタで実行した場合の時間と速度向上比を表 4 示す．また，論理合成による回路面積と回路シミュレーションで実行した場合の時間を表 5 に示す．表 4 と表 5 から，回路シミュレーション時間については，SMP 環境での実行と比較して，約 190×10^6～430×10^6 倍程度であった．すなわち，並列アルゴリズムやプログラムの検証に SMP 環境を用いることで，高速に検証を行えることが確認できる．スレッド数によって回路シミュレーション

時間が変わらないのは，スレッド数の増加によって実行クロック数が減少するが，代わりにシミュレーションしなければならない回路が増えるため，結果としてPCにかかる負荷があまり変わらなかったのではないかと思われる．

表 4：Nの総和のSMPクラスタでの実効時間

スレッド数	1	2	4
実行時間(μs)	0.056	0.032	0.025
速度向上比	1.000	1.789	2.233

表 5：Nの総和の回路シミュレーション時間

スレッド数	1	2	4
手書き(s)	10.7	10.8	10.8
システム(s)	10.9	10.7	10.7

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，次の漸化式

$$z_{n+1} = z_n^2 + c,$$

$$z_0 = 0$$

で定義される複素数列 $\{z_n\}_{n\in N}$ が n → ∞ 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．複素数 c を複素数平面上の点として（あるいは同じことだが c = a + ib と表して c を xy-平面上の点（a，b）として）表すと，この平面上でマンデルブロ集合は自己相似的なフラクタル図形として表される．

マンデルブロ集合はヒョウタンのような図形の周囲に自己相似的な図形が無数にくっついた形状をしている．拡大図には「飛び地」のような黒い部分がいくつか見られるが，これらは全てマンデルブロ集合本体に連結していることが 証明されている．

なお，上式で z0 を 0 以外の複素数にした場合，マンデルブロ集合は上記の図形をゆがめたものになる．

マンデルブロ集合を複素数を使わずに書き直すには，zn を点（xn，yn）に，c を点（a，b）にそれぞれ置き代えて，

$$x_{n+1} = x_n^2 - y_n^2 + a,$$

$$y_{n+1} = 2x_n y_n + b$$

とすればよい．

平面幾何学上で，マンデルブロ集合の周を拡大すると，元のものとよく似た形が繰り返して現れるが，全て少しずつ違っている．つまりマンデルブロ集合の周は自己相似ではないフラクタルの一種であり，その相似次元は平面内の曲線としては最大の 2 次元である．このことはマンデルブロの予想と呼ばれ未解決問題の一つだったが，宍倉光広によって肯定的に証明された．

4.2 手書きとシステムによる HDL 記述の生成

図 13 に OpenMP プログラム，図 14 に中間表現の抜粋，図 15 に手書きの HDL，図 16 にシステムの HDL の抜粋を示す．手書きによる HDL 記述は約 66 行であるのに対し，システムによる HDL 記述は約 374 行と約 5．7 倍の量になった．システムの方では専用のパラメータが宣言されており，演算基部，代入部，状態遷移部それぞれを分けて記述される．また，1 状態につき 1 演算行っている．手書きの方では，演算と代入を同時に記述されており，1 クロックで複数の処理が可能になるように記述されている．また，手書きでは専用のパラメータを利用していないこともあり，システムで生成されるものより，少ない行数で記述される．

演算器部では，状態 21 と状態 34 で加算が行われているので加算器が生成され，状態 33 で減算が行われているので減算器が生成されている．また状態 24，26，29，31，37，39 でシフト演算が行われているのでシフト演算器が生成され，状態 27，32，40 で乗算が行われているので，乗算器が生成されている．状態 21 では” +(7 20)”が行われる．状態 7 は”count”であり，状態 20 は定数 1 を示している．ゆえに P_STATE21 では”ADD1_A=count”，”ADD1_B=ConstNum20”となり”ADD1_RESULT”はそれらの計算結果を表すことになる．状態 24 では”>>(3 23)”が行われる．状態 3 は”xn”であり，状態 23 は定数 8 を示している．ゆえに P_STATE24 では”RSFT1_A=xn”，”RSFT1_B=ConstNum23”となり”RSFT1_RESULT”はそれらの計算結果を表すことになる．他の演算も同様に行われる．

代入部では，状態 9 で”=(3 8)”が行われる．状態 3 は”xn”であり，状態 8 は定数 0 である”ConstNum8”を示している．また”=”は代入を表しているので P_STATE9 で”xn <= ConstNum8”が生成される．状態 11 では”=(4 10)”が行われる．状態 4 は”yn”であり，状態 10 は定数 0 である”ConstNum10”を示している．また”=”は代入を表しているので P_STATE11 で”yn <= ConstNum10”が生成される．他の代入も同様に行われる．

状態遷移部では，P_INIT で示される初期状態から#0 で示される最初の演算である[17]を行うため，P_STATE17 に遷移する．P_STATE17 では，#2 に示される状態分岐判定を行う[19]に移動するため P_STATE19 に遷移する．P_STATE19 では条件分岐を行う．条件式は”count<ConstNum18”となっており，これが真の時，#3 で示される演算のである[24]と次の[26]の状態へ遷移する．他の状態遷移も同様に行われる．

```
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 13：マンデルブロ集合の OpenMP プログラム

```
----SemanticsAnalyze----
function 0 : main
0 : Auto Signed 32bit : <function> main( )
1 : Auto Signed 32bit :: a
2 : Auto Signed 32bit :: b
3 : Auto Signed 32bit :: xn
4 : Auto Signed 32bit :: yn
5 : Auto Signed 32bit :: xn1
6 : Auto Signed 32bit :: yn1
7 : Auto Signed 32bit :: count
8 : Auto Const Signed 32bit :: _8 := 0
9 : Auto Signed 32bit  : =( 3 8 )
10 : Auto Const Signed 32bit :: _10 := 0
11 : Auto Signed 32bit  : =( 4 10 )
12 : Auto Const Signed 32bit :: _12 := 0
13 : Auto Signed 32bit  : =( 5 12 )
14 : Auto Const Signed 32bit :: _14 := 0
15 : Auto Signed 32bit  : =( 6 14 )
16 : Auto Const Signed 32bit :: _16 := 0
17 : Auto Signed 32bit  : =( 7 16 )
18 : Auto Const Signed 32bit :: _18 := 30
19 : Auto Signed 32bit  : <( 7 18 )
20 : Auto Const Signed 32bit :: _20 := 1
21 : Auto Signed 32bit  : +( 7 20 )
22 : Auto Signed 32bit  : =( 7 21 )
23 : Auto Const Signed 32bit :: _23 := 8
24 : Auto Signed 32bit  : >>( 3 23 )
25 : Auto Const Signed 32bit :: _25 := 8
26 : Auto Signed 32bit  : >>( 3 25 )
27 : Auto Signed 32bit  : *( 24 26 )
28 : Auto Const Signed 32bit :: _28 := 8
29 : Auto Signed 32bit  : >>( 4 28 )
30 : Auto Const Signed 32bit :: _30 := 8
31 : Auto Signed 32bit  : >>( 4 30 )
32 : Auto Signed 32bit  : *( 29 31 )
33 : Auto Signed 32bit  : -( 27 32 )
34 : Auto Signed 32bit  : +( 33 1 )
35 : Auto Signed 32bit  : =( 5 34 )
36 : Auto Const Signed 32bit :: _36 := 8
37 : Auto Signed 32bit  : >>( 3 36 )
38 : Auto Const Signed 32bit :: _38 := 8
39 : Auto Signed 32bit  : >>( 4 38 )
40 : Auto Signed 32bit  : *( 37 39 )
41 : Auto Signed 32bit  : =( 6 40 )
42 : Auto Const Signed 32bit :: _42 := 2
～～～省略～～～
Argument  ( )
{
--#0  : [ [ 9 ] [ 11 ] [ 13 ] [ 15 ] ] -> #1
--#1  : { /0 } -> #2
--#2  : [ ]  <- #0
}
/0 Parallel FOR (2) ( )
-0:#0  : [ [ 17 ] ] -> #2
-0:#1  : [ ]  <- #0
-0:#2  : [ [ 19 ] ] -> 19 ? #3 : #1
-0:#3  : [ [ 24 ] [ 26 ] [ 27 ] [ 29 ] [ 31 ] [ 32 ] [ 33 ] [ 34 ]
           [ 35 ] [ 37 ] [ 39 ] [ 40 ] [ 41 ] [ 44 ] [ 46 ] [ 48 ]
           [ 49 ] [ 50 ] [ 51 ] [ 52 ] [ 53 ] ] -> #4
-0:#4  : [ [ 55 ] [ 57 ] [ 58 ] [ 60 ] [ 62 ] [ 63 ]
           [ 64 ] [ 67 ] [ 68 ] ] -> 68 ? #1 : #5
-0:#5  : [ [ 21 ] [ 22 ] ] -> #2
```

図 14：マンデルブロ集合の中間表現（抜粋）

```
module mandel( a,b, color, count, ef, clk,rst );

  input signed [31:0] a,b;
  wire  signed [31:0] a,b;
  input signed [31:0] color;
  wire  signed [31:0] color;
  output        [4:0] count;
  reg           [4:0] count;
  output        ef;
  reg           ef;
  input         clk,rst;
  wire          clk,rst;

  reg           signed [31:0] xn,yn, xn1,yn1;
  reg           [2:0] state;

always@(posedge clk or negedge rst) begin
  if(!rst) begin
    count=0;
    xn=0;
    yn=0;
    xn1=0;
    yn1=0;
    state=0;
    ef=1;

  end else begin
    if( state == 0 ) begin
      count=0;
      xn=0;
      yn=0;
      xn1=0;
      yn1=0;
      state=1;
      ef=1;
    end else if( state == 1 ) begin
      if( count<color ) 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
        state = 2;
      end

    end else if( state == 2 ) begin
      state = 0;
      ef=0;
    end

  end
end
endmodule
```

図 15：マンデルブロ集合の手書きの HDL

```
//演算器部
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_STATE21) ? count :
                (CurrentState==P_STATE34) ? REG33 : REG58;
assign ADD1_B = (CurrentState==P_STATE21) ? ConstNum20 :
                (CurrentState==P_STATE34) ? a : REG63;
//演算器部
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_STATE24) ? xn :
                 (CurrentState==P_STATE26) ? xn : yn;
assign RSFT1_B = (CurrentState==P_STATE24) ? ConstNum23 :
                 (CurrentState==P_STATE26) ? ConstNum25 : ConstNum61;
//代入部
always @ (posedge CLK or negedge XRST) begin
  case(CurrentState)
    P_INIT : oEND <= 1'b0;
    P_END  : oEND <= 1'b1;
    P_STATE9 : xn <= ConstNum8;
    P_STATE11 : yn <= ConstNum10;
    P_STATE13 : xn1 <= ConstNum12;
    P_STATE15 : yn1 <= ConstNum14;
    P_STATE17 : count <= ConstNum16;
    P_STATE21 : REG21 <= ADD1_RESULT;
    P_STATE22 : count <= REG21;
    P_STATE24 : REG24 <= RSFT1_RESULT;
    ～～～省略～～～
  endcase
end

//状態遷移部
always @(posedge CLK or negedge XRST) begin
  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(count<ConstNum18) CurrentState <= P_STATE24;
               else                 CurrentState <= P_END;
    P_STATE24: CurrentState <= P_STATE26;
    ～～～省略～～～
  endcase
end
```

図 16：マンデルブロ集合のシステムの HDL（抜粋）

手書きの FSM は図 15 の always ブロックより，図 17 のようになるのが分かる．システムの FSM は図 14 の状態遷移表より，図 18 のようになるのが分かる．番号は状態を示している．状態は 17 から始まり，状態 22 まで遷移すると状態 19 に戻り，繰り返し遷移する．

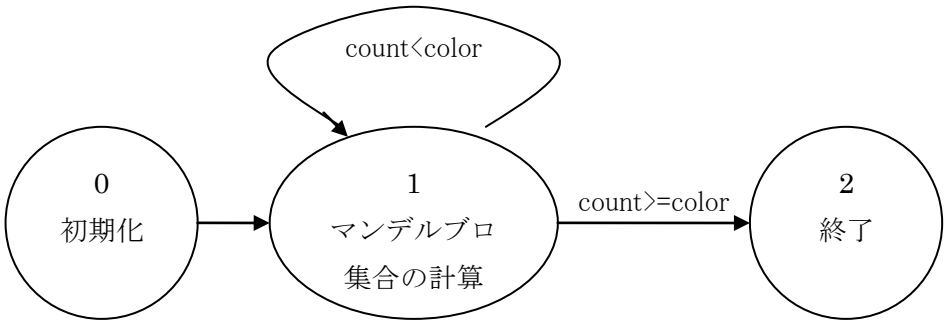

図 17：マンデルブロ集合の手書きの FSM

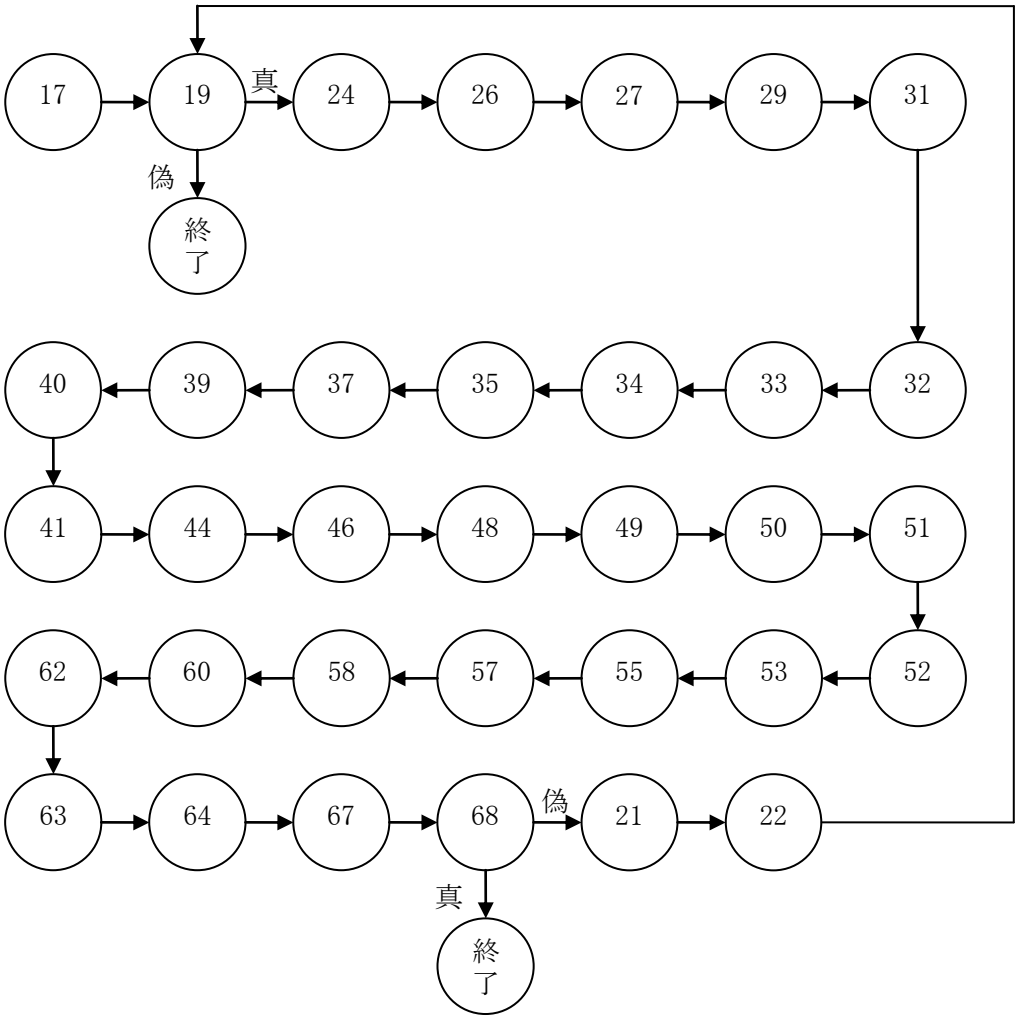

図 18：マンデルブロ集合のシステムの FSM

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 章で記したものと同じである．マンデルブロ集合に用いた画像は 100×100 である．

手書きとシステムで生成した回路の実行クロック数を表 6 に示す．表 2 で示した N の総和での実行結果に比べて，手書きとシステムともに，スレッド数に対するクロック減少比の伸びが小さい．これは，同じ処理を行う N の総和とは違って，マンデルブロ集合はデータ依存のため各ノードでの処理が均一にならないためだと思われる．

システムの方が手書きよりも実行クロック数が約 24 倍かかっている．理由として手書きは速度重視のアルゴリズムで回路を設計したのとは違い，システムでは一定の法則に従って回路を生成しているためだと思われる．また手書きは 1 クロックで複数の演算を行っているが，システムは 1 クロックに 1 演算しか行っていないことも原因だと考えられる．

表 6：マンデルブロ集合の実行クロック数

スレッド数		1	2	4
実行クロック数 (cloks)	手書き	76015	43130	31247
	システム	1838095	922860	773220
クロック減少比	手書き	1.00	1.76	2.43
	システム	1.00	1.99	2.38

手書きとシステムで生成した回路の回路規模を表 7 に示す．どちらもスレッド数の増加に対して，ほぼ同程度の回路面積比の増加が確認できる．

システムの方が手書きよりも回路規模が約 0.75 倍程度になっている．理由として手書きは速度重視の回路設計を行ったため，多くのレジスタが必要になり，レジスタを多く生成したことが挙げられる．システムは 1 クロック 1 演算しか行わないため，手書き以上のレジスタで必要なかったため，それらにレジスタを生成していないからであると考えられる．

表 7：マンデルブロ集合の回路規模

スレッド数		1	2	4
回路規模 (Slices)	手書き	1264	2558	5092
	システム	965	1931	3862
回路面積比	手書き	1.00	2.02	4.03
	システム	1.00	2.00	4.00

SMP クラスタで実行した場合の時間と速度向上比を表 8 示す．また，論理合成による回路面積と回路シミュレーションで実行した場合の時間を表 9 に示す．表 8 と表 9 から，回路シミュレーション時間については，SMP 環境での実行と比較して，約 7×10^3～40×10^3 倍程

度であった．動作合成システムのアルゴリズム評価系において，N の総和とマンデルブロ集合の実験結果から，共に SMP 環境での速度向上比と動作合成時のスレッド数による速度向上比に同様な相関が見られたことは，並列アルゴリズムを検証する際に有効であると考えられる．

表 8：マンデルブロ集合の SMP クラスタでの実行時間

スレッド数	1	2	4
実行時間(ms)	3.528	3.254	3.027
速度向上比	1	1.084405	1.165684

表 9：マンデルブロ集合の回路シミュレーション時間

スレッド数	1	2	4
手書き(s)	25.2	12.0	12.1
システム(s)	81.3	80.3	119.3

5. OpenMP 動作合成システムの評価

5.1 コードジェネレータの改良点

シフト演算を使用する OpenMP プログラムをトランスレータに通すと，中間表現ではシフト演算が表現できていたが，その中間コードをコードジェネレータに通すと，シフト演算は実現できなかった．本研究を行う課程で，コードジェネレータにシフト演算を生成ができるように改良した．図 19 にシフト演算を使用しているマンデルブロ集合を，中間コードから HDL を生成した抜粋の例を示す．文字に網がかかっている行が，今回の改良で新たに生成されるようになった行である．

HDL では浮動小数点を利用できないので，小数を表現するには固定小数点を用いるしかない．そして固定小数点の四則演算では，シフト演算が必須である．今回の改良でシフト演算が利用可能になったことにより，HDL でも少数の表現ができるようになった．これにより，本システムの利便性が向上したと思われる．

```
//演算器部
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_STATE24) ? xn :
                 (CurrentState==P_STATE26) ? xn : yn;
assign RSFT1_B = (CurrentState==P_STATE24) ? ConstNum23 :
                 (CurrentState==P_STATE26) ? ConstNum25 : ConstNum61;
//代入部
always @ (posedge CLK or negedge XRST) begin
  case(CurrentState)
    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;
    …
  endcase
end

//状態遷移部
always @(posedge CLK or negedge XRST) begin
     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;
        …
     endcase
end
```

図 19：シフト演算の生成

5.2 考察

N の総和とマンデルブロ集合について，手書きから生成された回路と，システムから生成された回路について比較を行った．

速度については，手書きでは 1 クロック複数演算を行っており，システムでは 1 クロック 1 演算を行っているので，N の総和とマンデルブロ集合ともに，手書きによって生成した回路がシステムから生成した回路よりも，およそ 4〜24 倍優れた結果になった．これより，システムの速度向上を図るには，1 クロック 1 演算を 1 クロック複数演算が行えるようにすることで実現できると思われる．例えば”i=a+b”の演算を 1 クロック 1 演算で行うと，加算と代入で 1 クロックずつ必要になり，合計 2 クロックかかる．これを 1 クロック複数演算で行うと，加算と代入が同時に行えるので，合計 1 クロックで処理可能になる．

回路規模については，N の総和は手書きの方が回路規模が小さい回路を生成できたが，マンデルブロ集合についてはシステムの方が回路規模が小さい回路を生成できた．手書きのマンデルブロ集合では，速度を重視した設計を行ったため，このような結果になったと思われる．更に回路規模を削減するには，速度向上を図る方法と同じく，1 クロック複数演算にすることが挙げられる．例えば”i=a+b”の演算を 1 クロック 1 演算で行うと，加算の結果を保存するレジスタと，代入先のレジスタが必要になり，合計で 2 つのレジスタを生成しなければならない．これを 1 クロック複数演算で行うと，加算の結果を保存するレジスタが不要になるので，合計で 1 つのレジスタで実現できる．またこれにより，状態遷移数が減少するので，配線数も削減可能になる．よって，回路規模の削減が可能となる．

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対して， N の総和とマンデルブロ集合の OpenMP プログラムから本システムを用いて動作合成を行った．速度向上においては，N の総和では並列数に対して理想的な速度向上が，マンデルブロ集合においては並列数よりも低い速度向上が得られた．また，システムから生成された回路は，手書きによって生成した回路よりも実行クロック数がかかる結果が得られた．回路規模において，N の総和，マンデルブロ集合共に並列度に比例する回路規模となった．また，システムから生成された回路は，手書きによって生成された回路より，回路規模が小さな回路が生成された．

今後の課題として，1 クロック複数演算の実装が考えられる．これによって本システムから生成される回路の，更なる速度向上と回路規模の縮小を実現できると思われる．

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，本動作合成システムを立ち上げ，事あるごとに相談に乗って頂き，貴重な助言を頂いた松崎裕樹氏に深く感謝いたします．

最後に，共同研究者である金森央樹氏，本研究に関して貴重な意見をいただきました高性能計算研究室の皆様に心より感謝致します．

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”，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”，FIT2008，C-008，2008.

[3] 森江善之，冨山宏行，村上和彰:“動作合成の効率化を指向した動作レベル記述・トランスフォーメーション”，情報処理学会研究報告，Vol.2003，No.120，2003.

[4] 鈴木，亀井，富田，森下，山田，久保:“C 言語によるサイクル精度でのハードウェア/ソフトウェア協調検証手法”，シャープ技法第 92 号，pp.78-83，2005.

[5] 西口健一，石浦菜岐佐，西村啓成，神原弘之，冨山宏之，高務祐哲，小谷学:〝ソフトウェア互換ハードウェアを合成する高位合成システム CCAP における変数と関数の扱い〞電子情報通信学会技術研究報告，VLD2005-79，ICD2005-174，DC2005-56，pp. 19-24.

[6] 岩田，田中，山崎:“C 言語による有限ステートマシンベースのプロセッサ生成”，信学技報,VLD2001-07,ICD2001-162,FTS2001-64,pp33-38，2001.

[7] 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 回全国大会論文集分冊 1，pp.99-100,2005.

[8] 松浦努，内田順平，宮岡祐一郎，戸川望，柳澤政生，大附辰夫，“ネットワークプロセッサ合成システム”，電子情報通信学会技術研究報告，VLD2003-145，pp.55-60,2003.

[9] Spec C：http://www.specc.gr.jp/

[10]Handel-C：http://www.celoxica.co.jp/products/system_tools.asp/

[11]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ェ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.55-60.，2005.

付録 A マンデルブロ集合の中間表現

```
----SemanticsAnalyze----
function 0 : main
0 : Auto Signed 32bit : <function> main( )
1 : Auto Signed 32bit :: a
2 : Auto Signed 32bit :: b
3 : Auto Signed 32bit :: xn
4 : Auto Signed 32bit :: yn
5 : Auto Signed 32bit :: xn1
6 : Auto Signed 32bit :: yn1
7 : Auto Signed 32bit :: counter
8 : Auto Const Signed 32bit :: _8 := 0
9 : Auto Signed 32bit  : =( 3 8 )
10 : Auto Const Signed 32bit :: _10 := 0
11 : Auto Signed 32bit  : =( 4 10 )
12 : Auto Const Signed 32bit :: _12 := 0
13 : Auto Signed 32bit  : =( 5 12 )
14 : Auto Const Signed 32bit :: _14 := 0
15 : Auto Signed 32bit  : =( 6 14 )
16 : Auto Const Signed 32bit :: _16 := 0
17 : Auto Signed 32bit  : =( 7 16 )
18 : Auto Const Signed 32bit :: _18 := 30
19 : Auto Signed 32bit  : <( 7 18 )
20 : Auto Const Signed 32bit :: _20 := 1
21 : Auto Signed 32bit  : +( 7 20 )
22 : Auto Signed 32bit  : =( 7 21 )
23 : Auto Const Signed 32bit :: _23 := 8
24 : Auto Signed 32bit  : >>( 3 23 )
25 : Auto Const Signed 32bit :: _25 := 8
26 : Auto Signed 32bit  : >>( 3 25 )
27 : Auto Signed 32bit  : *( 24 26 )
28 : Auto Const Signed 32bit :: _28 := 8
29 : Auto Signed 32bit  : >>( 4 28 )
30 : Auto Const Signed 32bit :: _30 := 8
31 : Auto Signed 32bit  : >>( 4 30 )
32 : Auto Signed 32bit  : *( 29 31 )
33 : Auto Signed 32bit  : -( 27 32 )
34 : Auto Signed 32bit  : +( 33 1 )
35 : Auto Signed 32bit  : =( 5 34 )
36 : Auto Const Signed 32bit :: _36 := 8
37 : Auto Signed 32bit  : >>( 3 36 )
38 : Auto Const Signed 32bit :: _38 := 8
39 : Auto Signed 32bit  : >>( 4 38 )
40 : Auto Signed 32bit  : *( 37 39 )
41 : Auto Signed 32bit  : =( 6 40 )
42 : Auto Const Signed 32bit :: _42 := 2
43 : Auto Const Signed 32bit :: _43 := 16
44 : Auto Signed 32bit  : <<( 42 43 )
45 : Auto Const Signed 32bit :: _45 := 8
46 : Auto Signed 32bit  : >>( 44 45 )
47 : Auto Const Signed 32bit :: _47 := 8
```

```
48 : Auto Signed 32bit  : >>( 6 47 )
49 : Auto Signed 32bit  : *( 46 48 )
50 : Auto Signed 32bit  : +( 49 2 )
51 : Auto Signed 32bit  : =( 6 50 )
52 : Auto Signed 32bit  : =( 3 5 )
53 : Auto Signed 32bit  : =( 4 6 )
54 : Auto Const Signed 32bit :: _54 := 8
55 : Auto Signed 32bit  : >>( 3 54 )
56 : Auto Const Signed 32bit :: _56 := 8
57 : Auto Signed 32bit  : >>( 3 56 )
58 : Auto Signed 32bit  : *( 55 57 )
59 : Auto Const Signed 32bit :: _59 := 8
60 : Auto Signed 32bit  : >>( 4 59 )
61 : Auto Const Signed 32bit :: _61 := 8
62 : Auto Signed 32bit  : >>( 4 61 )
63 : Auto Signed 32bit  : *( 60 62 )
64 : Auto Signed 32bit  : +( 58 63 )
65 : Auto Const Signed 32bit :: _65 := 4
66 : Auto Const Signed 32bit :: _66 := 16
67 : Auto Signed 32bit  : <<( 65 66 )
68 : Auto Signed 32bit  : >( 64 67 )
Argument  ( )
{
--#0  : [ [ 9 ] [ 11 ] [ 13 ] [ 15 ] ] -> #1
--#1  : { /0 } -> #2
--#2  : [ ]  <- #0
}
/0 Parallel FOR (2) ( )
-0:#0  : [ [ 17 ] ] -> #2
-0:#1  : [ ]  <- #0
-0:#2  : [ [ 19 ] ] -> 19 ? #3 : #1
-0:#3  : [ [ 24 ] [ 26 ] [ 27 ] [ 29 ] [ 31 ] [ 32 ] [ 33 ] [ 34 ] [ 35 ] [ 37 ] [ 39 ]
[ 40 ] [ 41 ] [ 44 ] [ 46 ] [ 48 ] [ 49 ] [ 50 ] [ 51 ] [ 52 ] [ 53 ] ] -> #4
-0:#4  : [ [ 55 ] [ 57 ] [ 58 ] [ 60 ] [ 62 ] [ 63 ] [ 64 ] [ 67 ] [ 68 ] ] -> 68 ?
#1 : #5
-0:#5  : [ [ 21 ] [ 22 ] ] -> #2
```

付録 B マンデルブロ集合のシステムの HDL

```
module mandel( iSTART, oEND, oADDR, oDATA, iDATA, oRD, oWR, iEN, iRUN, iSTALL, CLK,
XRST, a,b);
 input iSTART;
 output oEND;
 reg oEND;
 output signed [31:0]oADDR,oDATA;
 reg signed [31:0]oADDR,oDATA;
 input [31:0]iDATA;
 output oRD, oWR;
 reg oRD,oWR;
 input iEN;
 input [31:0]iRUN;
 input iSTALL;
 input CLK,XRST;

reg [7:0]CurrentState;

parameter  [31:0]ConstNum8 = 32'd0;
parameter  [31:0]ConstNum10 = 32'd0;
parameter  [31:0]ConstNum12 = 32'd0;
parameter  [31:0]ConstNum14 = 32'd0;
parameter  [31:0]ConstNum16 = 32'd0;
parameter  [31:0]ConstNum18 = 32'd30;
parameter  [31:0]ConstNum20 = 32'd1;
parameter  [31:0]ConstNum23 = 32'd8;
parameter  [31:0]ConstNum25 = 32'd8;
parameter  [31:0]ConstNum28 = 32'd8;
parameter  [31:0]ConstNum30 = 32'd8;
parameter  [31:0]ConstNum36 = 32'd8;
parameter  [31:0]ConstNum38 = 32'd8;
parameter  [31:0]ConstNum42 = 32'd2;
parameter  [31:0]ConstNum43 = 32'd16;
parameter  [31:0]ConstNum45 = 32'd8;
parameter  [31:0]ConstNum47 = 32'd8;
parameter  [31:0]ConstNum54 = 32'd8;
parameter  [31:0]ConstNum56 = 32'd8;
parameter  [31:0]ConstNum59 = 32'd8;
parameter  [31:0]ConstNum61 = 32'd8;
parameter  [31:0]ConstNum65 = 32'd4;
parameter  [31:0]ConstNum66 = 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;

input signed [31:0]a;
input signed [31:0]b;
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;

reg signed [31:0]REG0;
reg signed [31:0]REG19;
reg signed [31:0]REG21;
reg signed [31:0]REG24;
reg signed [31:0]REG26;
reg signed [31:0]REG27;
reg signed [31:0]REG29;
reg signed [31:0]REG31;
reg signed [31:0]REG32;
reg signed [31:0]REG33;
reg signed [31:0]REG34;
reg signed [31:0]REG37;
reg signed [31:0]REG39;
reg signed [31:0]REG40;
reg signed [31:0]REG44;
reg signed [31:0]REG46;
reg signed [31:0]REG48;
reg signed [31:0]REG49;
reg signed [31:0]REG50;
reg signed [31:0]REG55;
reg signed [31:0]REG57;
reg signed [31:0]REG58;
reg signed [31:0]REG60;
reg signed [31:0]REG62;
reg signed [31:0]REG63;
reg signed [31:0]REG64;
reg signed [31:0]REG67;
reg signed [31:0]REG68;

//演算器部
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_STATE21) ? counter :
(CurrentState==P_STATE34) ? REG33 :
(CurrentState==P_STATE50) ? REG49 :
(CurrentState==P_STATE64) ? REG58 :
```

```
REG58;
assign ADD1_B = (CurrentState==P_STATE21) ? ConstNum20 :
(CurrentState==P_STATE34) ? a :
(CurrentState==P_STATE50) ? b :
(CurrentState==P_STATE64) ? REG63 :
REG63;

//演算器部
wire signed [31:0]SUB1_RESULT;
wire signed [31:0]SUB1_A,SUB1_B;
assign SUB1_RESULT = SUB1_A - SUB1_B;
assign SUB1_A = (CurrentState==P_STATE33) ? REG27 :
REG27;
assign SUB1_B = (CurrentState==P_STATE33) ? REG32 :
REG32;

//演算器部
wire signed [31:0]MUL1_RESULT;
wire signed [31:0]MUL1_A,MUL1_B;
assign MUL1_RESULT = MUL1_A * MUL1_B;
assign MUL1_A = (CurrentState==P_STATE27) ? REG24 :
(CurrentState==P_STATE32) ? REG29 :
(CurrentState==P_STATE40) ? REG37 :
(CurrentState==P_STATE49) ? REG46 :
(CurrentState==P_STATE58) ? REG55 :
(CurrentState==P_STATE63) ? REG60 :
REG60;
assign MUL1_B = (CurrentState==P_STATE27) ? REG26 :
(CurrentState==P_STATE32) ? REG31 :
(CurrentState==P_STATE40) ? REG39 :
(CurrentState==P_STATE49) ? REG48 :
(CurrentState==P_STATE58) ? REG57 :
(CurrentState==P_STATE63) ? REG62 :
REG62;

//演算器部
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_STATE24) ? xn :
(CurrentState==P_STATE26) ? xn :
(CurrentState==P_STATE29) ? yn :
(CurrentState==P_STATE31) ? yn :
(CurrentState==P_STATE37) ? xn :
(CurrentState==P_STATE39) ? yn :
(CurrentState==P_STATE46) ? REG44 :
(CurrentState==P_STATE48) ? yn1 :
(CurrentState==P_STATE55) ? xn :
(CurrentState==P_STATE57) ? xn :
(CurrentState==P_STATE60) ? yn :
(CurrentState==P_STATE62) ? yn :
yn;
assign RSFT1_B = (CurrentState==P_STATE24) ? ConstNum23 :
(CurrentState==P_STATE26) ? ConstNum25 :
```

```
(CurrentState==P_STATE29) ? ConstNum28 :
(CurrentState==P_STATE31) ? ConstNum30 :
(CurrentState==P_STATE37) ? ConstNum36 :
(CurrentState==P_STATE39) ? ConstNum38 :
(CurrentState==P_STATE46) ? ConstNum45 :
(CurrentState==P_STATE48) ? ConstNum47 :
(CurrentState==P_STATE55) ? ConstNum54 :
(CurrentState==P_STATE57) ? ConstNum56 :
(CurrentState==P_STATE60) ? ConstNum59 :
(CurrentState==P_STATE62) ? ConstNum61 :
ConstNum61;

//演算器部
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_STATE44) ? ConstNum42 :
(CurrentState==P_STATE67) ? ConstNum65 :
ConstNum65;
assign LSFT1_B = (CurrentState==P_STATE44) ? ConstNum43 :
(CurrentState==P_STATE67) ? ConstNum66 :
ConstNum66;

//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  : oEND <= 1'b1;
     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

endmodule
```