卒業論文

OpenMP ハードウェア動作合成システムの検証(Ⅰ)

氏名	:	金森 央樹
学籍番号	:	2260050093-0
指導教員	:	山崎 勝弘 教授
提出日	:	2008 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 ハードウェア動作合成システムの生成回路と，手書きで記述した HDL による生成回路との比較と検証を行い，システムの改良点の検討を行った．

本研究で提案する OpenMP を用いた動作合成システムは, OpenMP を利用することを特徴としており，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，素数判定及びラプラシアンフィルタのプログラムを OpenMP で記述し，本システムを用いて動作合成を行い，手書きで記述し生成された回路とそれぞれシミュレーションをして比較を行い，コードジェネレータ及び本システム全体の改良点の模索と評価を行っている．素数判定に対して回路規模はシステム，手書きともに大きな差異はなく，速度はクロックサイクル数は手書きの回路が小さかったが，動作周波数はシステムのものが上回っていた．ラプラシアンフィルタでは回路規模が手書きのものに対し，システム側では 4 倍の大きさになった．ノード数を増やしたときのクロックサイクルの増加量はシステム側のほうが小さくなり理想的な速度向上を得ることができた．また素数判定，ラプラシアンフィルタともに，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．

目次

図目次

表目次

1. はじめに

近年，半導体の微細化技術の進歩によって，半導体集積回路に搭載できる回路規模が増大し，LSI に対して高度で多様な機能の実現をもたらした．その結果，LSI は大規模計算機やパーソナルコンピュータなどにとどまらず，携帯電話や家電，自動車などといった多方面の用途に用いられている．しかし，そういった回路の大規模化と用途の拡大に伴い，LSI の高機能化や求められる性能・品質は多様化し，それに比例するように LSI の開発の工程と複雑化さが増加している．その一方で，激しい市場競争により短い期間，低コストでの開発が要求されており，設計規模の増大に設計能力が追いつかないという設計生産性の危機が問題となっている．

こうした要求における開発期間短縮を実現するため，従来の HDL を用いたハードウェアの設計手法ではなく，C ベース言語を用いてシステムをより高い抽象度で記述する手法に設計手法が移行しつつある[5]．LSI の回路動作を 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することで HDL による設計に比べてより少ないコード数で機能が記述できるため大幅な設計生産性の向上が期待できる．また様々な仕様の変更も，設計段階において容易に対応できるようになる．

C 言語は空間的な概念，並列動作の概念が含まれていないため，自動合成するにあたって回路面積，性能，消費電力の面で最適なハードウェアが生成されるとは限らない[6]．さらに並列動作の概念は，ハードウェア設計において重要な要素であるが，C 言語などの逐次処理の実行モデルでは表現が難しく，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．

本研究では，以上のような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を提案し，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し，システムの改良と評価を目的とする．

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．そのため，既存の並列プログラミング言語である MPI や PVM のように通信や信号で並列動作を記述しないので，プログラミングが容易である．

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 OpenMP で書かれたプログラムから中間表現までの出力するトランスレータ，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まで実装されており，システムとしての枠組みは完成されている[2]．本研究では，素数判定プログラムと画像処理ア

ルゴリズムであるラプラシアンフィルタに対し，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，システムの改良と評価を行っている．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，実行サイクル数，そして論理合成においての動作周波数，回路規模を測定した．回路の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 環境での実行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が記述できるように 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 章では素数判定に対する動作合成の実験結果，第 4 章ではラプラシアンフィルタ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 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．

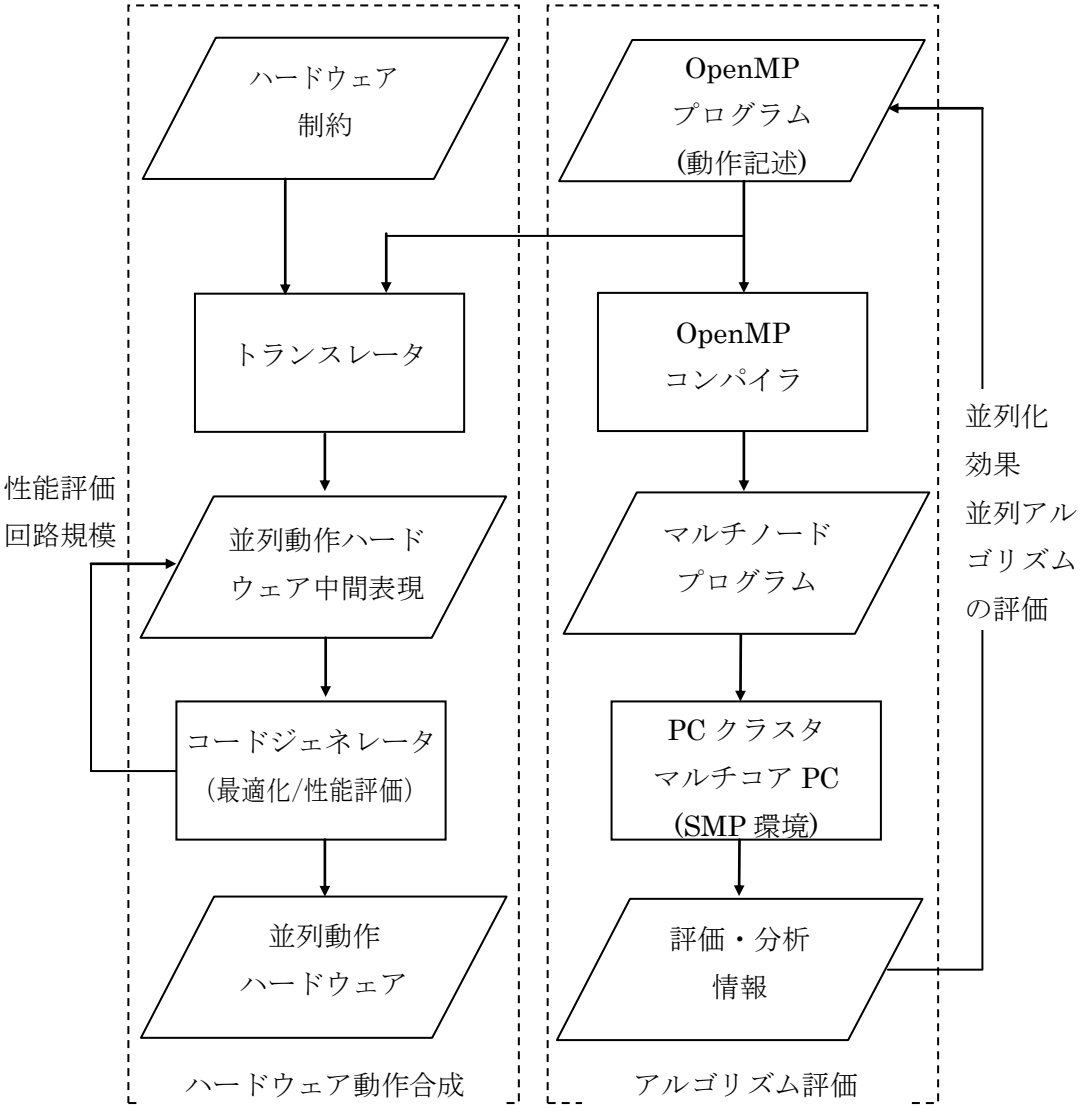

図 1：OpenMP を用いたハードウェア動作合成システム

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ノー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トランスレータによる中間表現への変換，また中間表現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，手書きで記述した HDL により生成される回路と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の説明をする．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[1]．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．

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 for 文に OpenMP におけるプラグマを挿入されており，ループ内では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，最初にシンボルテーブルのシンボル 4 で示される定数の代入を表す”: =(1 3)”の処理が行われる．” : =(1 3)”ではシンボル 1 で示される変数 i に対し，シンボル 3 で示される定数 0 の代入を示している．次に for 文の条件式である#2 へ遷移し，条件式の判定を行い分岐する．ここではシンボル 6 が真でなら#3 へ，そうでなければ#1 へと遷移する．シンボル 6 とは条件式を表す”<(1 5)”であり，これはシンボル 1 の変数 i とシンボル 5 の定数”256”との比較，”i<256”を示している．#3 ではシンボルテーブルの 9,10 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#2 へ遷移する．

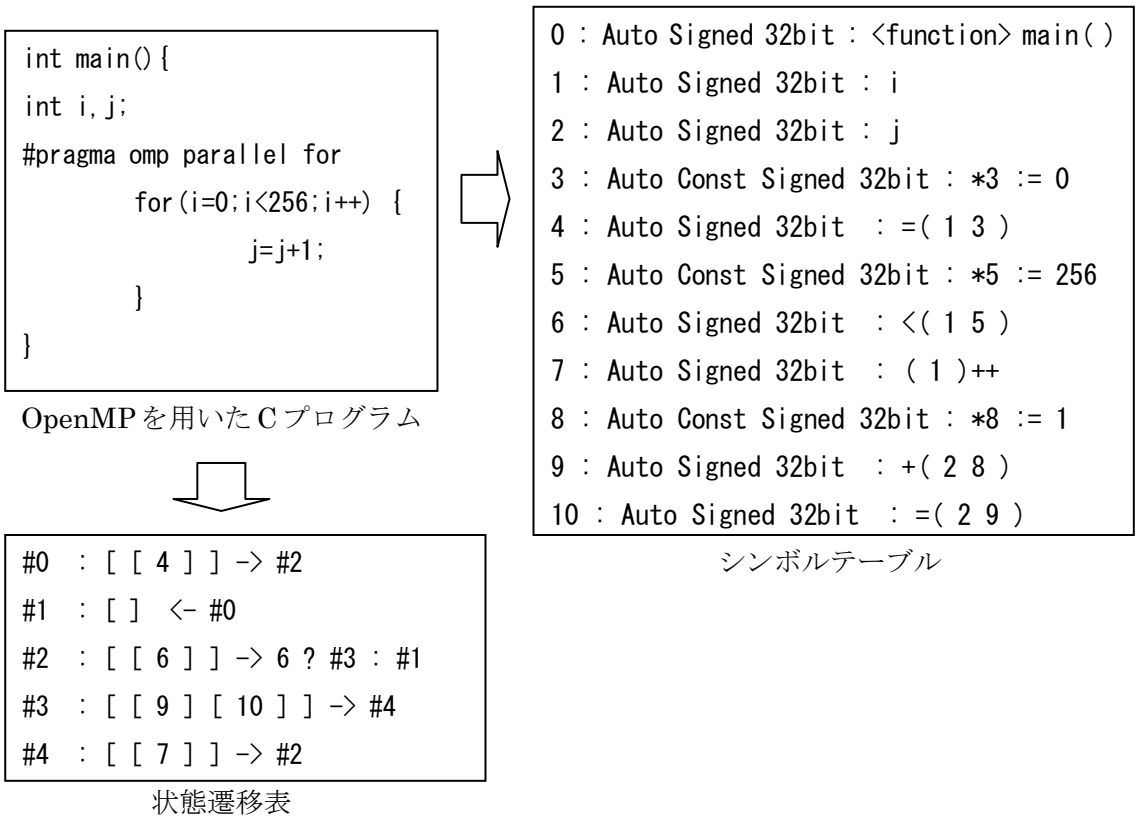

図 2：中間表現のサンプル

2.3 ハードウェアモジュールのコード生成

中間表現からコードジェネレータにより生成されるコードについて説明する．生成されるコードは，演算器部，代入部，状態遷移部に分けて生成される．演算器部と代入部はシンボルテーブル，状態遷移部は状態遷移表を元に生成されている．状態遷移部ではCurrentState を状態を表す変数とし，これに P_STATEnumber という別で用意されている複数のパラメータを代入していくことで遷移を行っている．実際はシンボル 5 を実行するときは P_STATE5 というように，number にはその状態で実行されるシンボルの番号が入る．代入部では CurrentState の値によって何を代入するのかを選択し，演算器部も同様に CurrentState の値によって何を加算するのかを選択している．その様子を図 3 に示す．

```
case(CurrentState)
P_STATE4: CurrentState <= P_STATE6;
P_INIT   : if(iSTART==1'b1) CurrentState <= P_STATE4;
        else  CurrentState<= CurrentState;
P_END    : CurrentState <=CurrentState;
P_STATE6: if(i<ConstNum5) CurrentState <= P_STATE9;
        else  CurrentState <= P_END;
P_STATE9: CurrentState <= P_STATE10;
P_STATE10: CurrentState <= P_STATE7;
P_STATE7: CurrentState <= P_STATE6;
default : CurrentState <= CurrentState;
endcase
```

状態遷移部

```
case(CurrentState)
P_END  : oEND <= 1'b1;
P_STATE4 : i <= ConstNum3;
P_STATE7 : i <= ADD_RESULT;
P_STATE9 : REG9 <= ADD_RESULT;
P_STATE10 : j <= REG9;
default : oEND <= 1'b0;
endcase
```

代入部

```
wire [31:0]ADD1_RESULT;
wire [31:0]ADD1_A,ADD1_B;
assign ADD_RESULT = ADD_A + ADD_B;
assign ADD_A = (CurrentState==P_STATE7) ? i :
(CurrentState==P_STATE9) ? j :
j;
assign ADD_B = (CurrentState==P_STATE7) ?
32'd1 :
(CurrentState==P_STATE9) ? ConstNum8 :
ConstNum8;
```

演算器部

図 3：シンボルテーブルと代入部，演算器部の対応

図 4 はハードウェア動作合成によって生成する並列ハードウェアのモデルである．データ並列では同じ処理の繰り返しになるため，各ノードの DataPath はほとんど同じとなり，FSM は繰り返し範囲に応じて生成される[4]．top_module は各ノードの入力，出力の統合を行う．

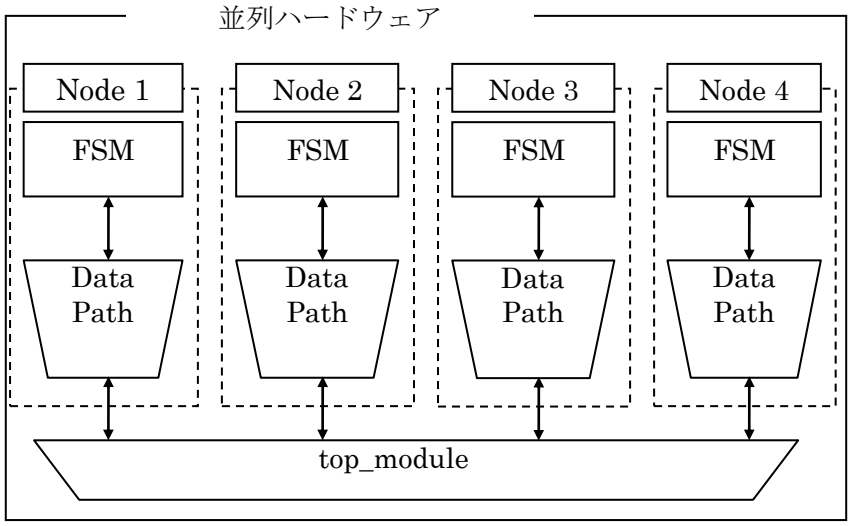

図 4：データ並列のハードウェアモデル

図 5 は並列ハードウェアの各ノードのハードウェアモジュールモデルである．FSM により使用するレジスタと演算器を選択し処理を行う．

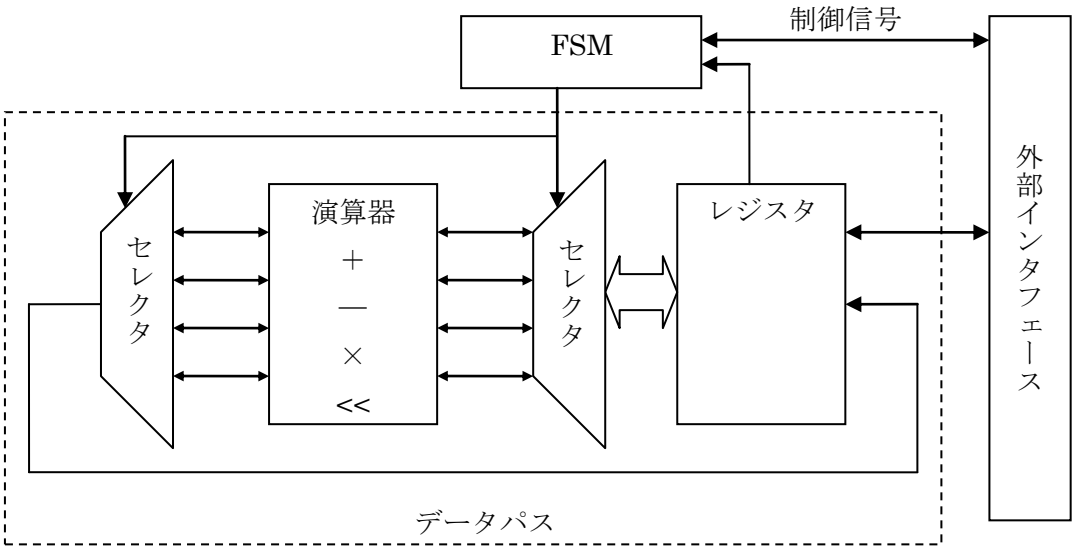

図 5：ハードウェアモジュールモデル

図 6 では実際に HDL 上においてのデータ分割を表している．C ソースコードの for 文の番兵値 256 をもとに，データを 4 つのノードに均等に 64 ずつ分割する処理を表している．この値をもとに各ノードはループ回数や，扱うデータなどを判断する．

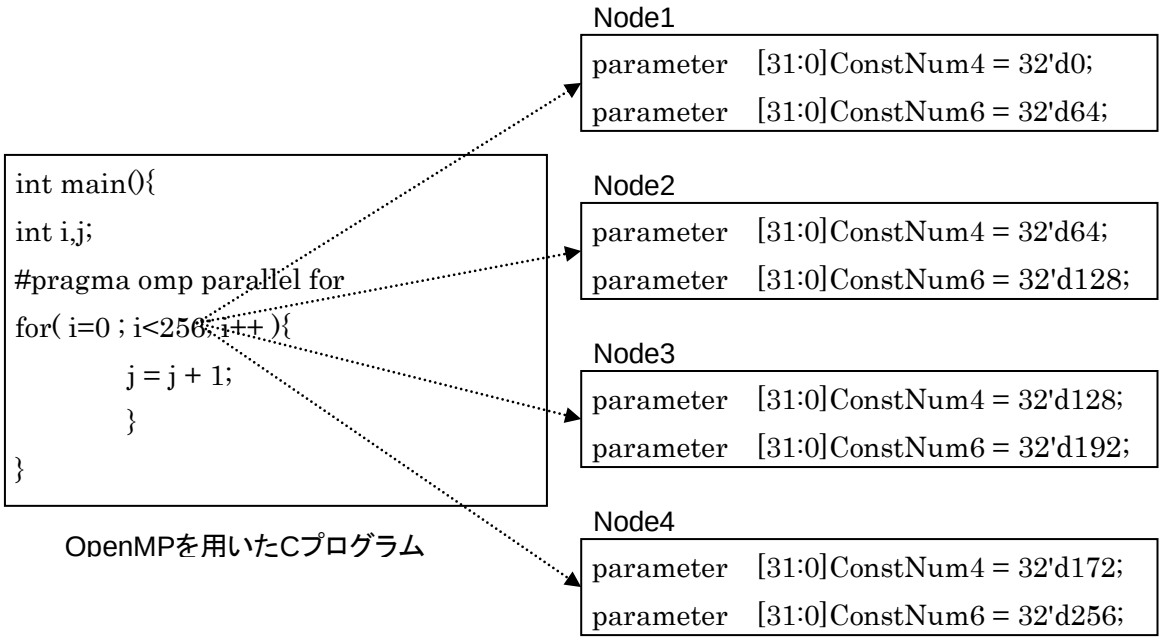

図 6：データ分割による並列化

3. 素数判定に対するハードウェア動作合成システム

3.1 素数判定のアルゴリズム

本研究で扱う素数判定は，素数であるか判定したい値に対し，2 から割っていく値を 1 ずつ増やしていき，最後まで割り切れなかった場合は素数であると判断するという，単純なアルゴリズムを用いた．しかし，コードジェネレータが剰余に対応していないため，減算を繰り返すことで除算の代用とした．そのアルゴリズムのフローチャートを図 7 に示す．並列案として i によるループ箇所を分割することにした．

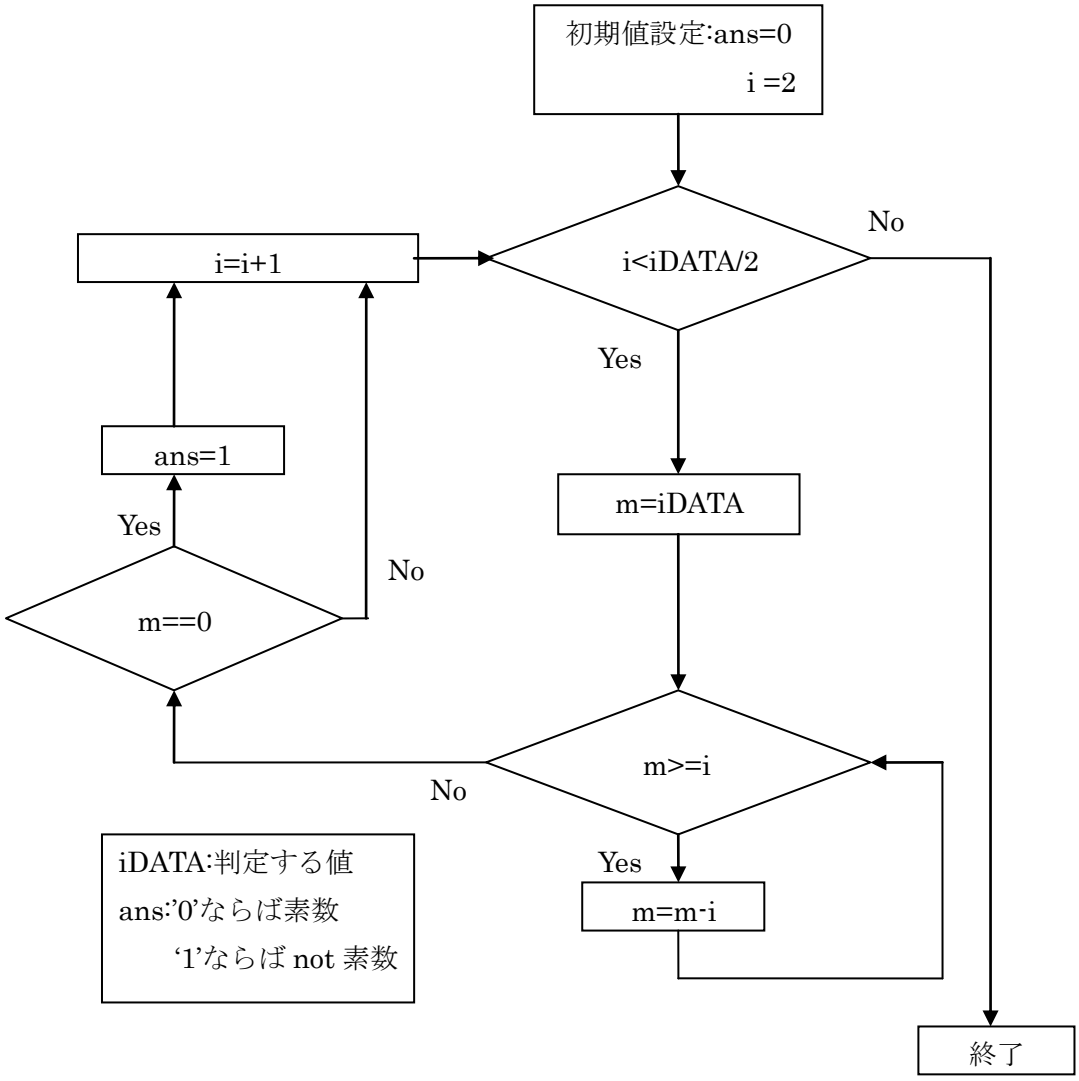

図 7：素数判定のフローチャート

図 8 のアルゴリズムをもとに作成した OpenMP によるプログラムと，そ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 8 に示す．iDATA は素数であるか判定をする入力値とする．iDATA の入力も HDL 上で制御するため省略した．

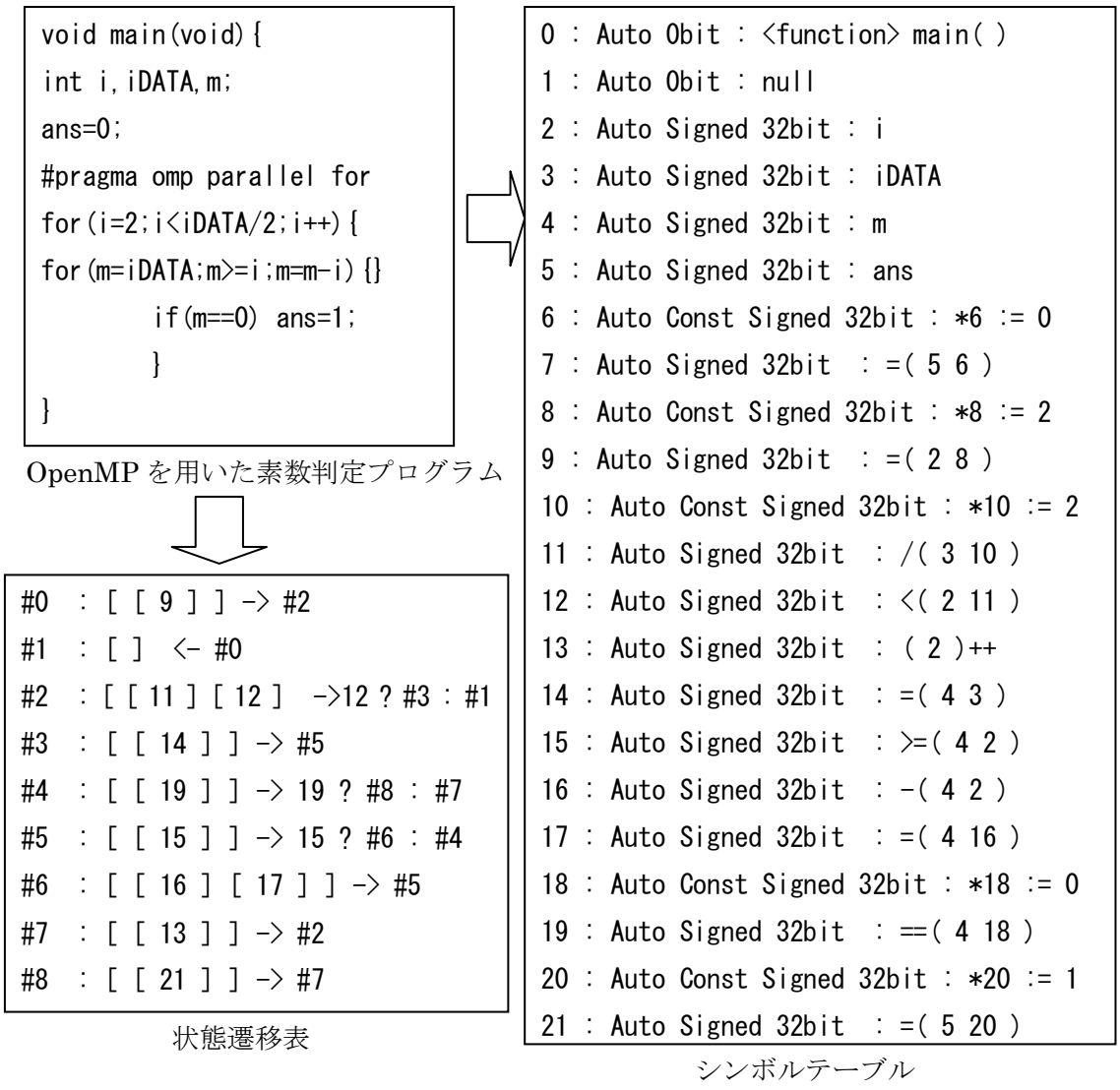

図 8：OpenMP を用いた素数判定プログラムと中間表現

3.2 手書きとシステムによる HDL 記述の生成

図 8 のプログラムをシステムにとおしたところ，並列化を行う”i<iDATA/2”の部分が中間表現では正しく生成されているが，コードジェネレータで並列化するデータが変数であるときに対応していなかったため，対応する箇所が空白で出力されており正しく生成されなかった．そのため iDATAstart と iDATAend というワイヤを用意し，それぞれのノードの扱うデータの領域を iDATAstart から iDATAend までとしてデータ分割し，手動による修正を行った．図 9 では 4 つのノードに均等にデータを分割している様子を表す．また OpenMP によるプログラムでは for 文の外で変数 ans の初期値を 0 とする演算が行われているが，HDL では初期化の演算が行われる状態に遷移がされないように生成されている．しかし，値 0 の初期化はリセットで行うことで解決した．

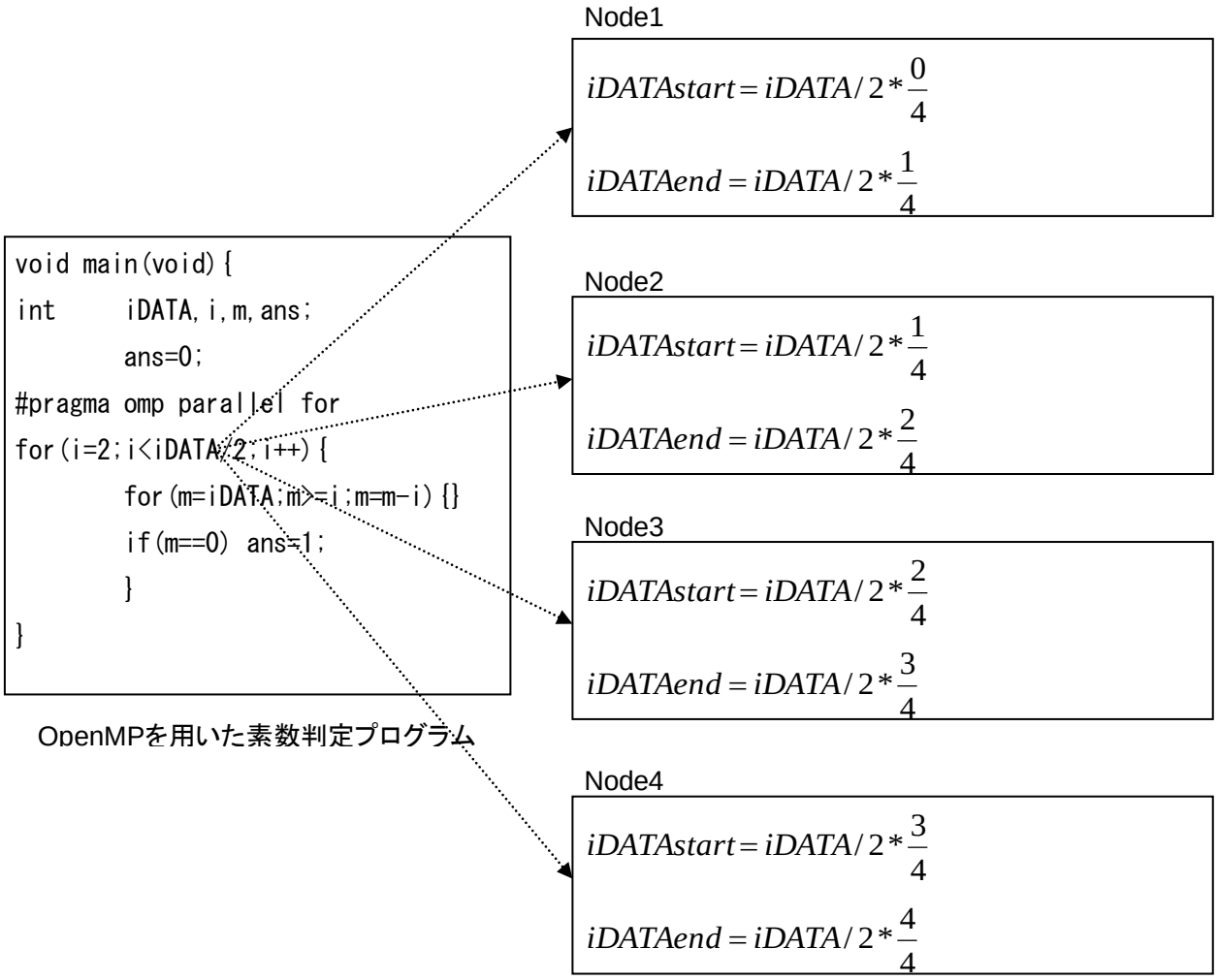

図 9：変数のデータ分割の修正

手書きによる HDL 記述は約 40 行であるのに対して，システムによる HDL 記述のほうは約 140 行と約 3 倍の量になった．それぞれの HDL 記述を付録 A，B に示す．システムのほうでは専用のパラメータの宣言がされており，演算器部，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 演算としている．手書きのほうでは演算器部を用意せず直接演算と代入の式を記述しており，1 クロックで複数の処理を行っている．1 クロック 1 演算の様子を表す代入部と，手書きの生成回路の主となる処理部を図 10 に示す．

```
assign ADD1_RESULT = ADD1_A + ADD1_B;
assign ADD1_A = (CurrentState==P_STATE13) ?
i : i;
assign ADD1_B = (CurrentState==P_STATE13) ?
32'd1 : i;
```

システムの生成回路の演算器部(加算部)

```
case(CurrentState)
  P_INIT : oEND <= 1'b0;
  P_END  : oEND <= 1'b1;
  P_STATE7 : ans <= ConstNum6;
  P_STATE9 : i <= iDATAstart; //手書き修正点
  P_STATE13 : i <= ADD1_RESULT;
  P_STATE14 : m <= iDATA;
  P_STATE16 : REG16 <= SUB1_RESULT;
  P_STATE17 : m <= REG16;
  P_STATE21 : ans <= ConstNum20;
  default : oEND <= 1'b0;
 endcase
```

システムの生成回路の代入部

```
if(i==0&&m==0)begin
    i <= iDATAfor;
    m <= iDATA;
 end else if(i==iDATAend)begin
     oEND <=1'b1;
 end else begin
     if(m>i)begin
        m <= m-i;
 end else if(m==i) begin
      ans<=1;
      oEND <=1'b1;
     end else begin
       m <= iDATA;
       i <= i+1;
     end
```

手書きの生成回路の演算部

図 10：システム生成回路の代入部と手書きの生成回路の演算部

状態の遷移に関する記述はシステム側では case 文を使っているのに対し，手書き側では if 文の入れ子などを用いており条件分岐を繰り返している. 各ハードウェアでの状態遷移のモデルを図 11 と図 12 に示す.

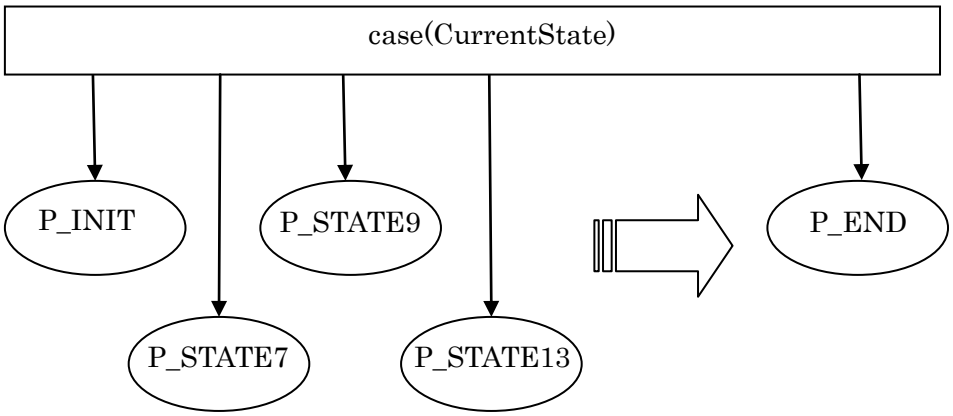

図 11：システムによる状態遷移のモデル

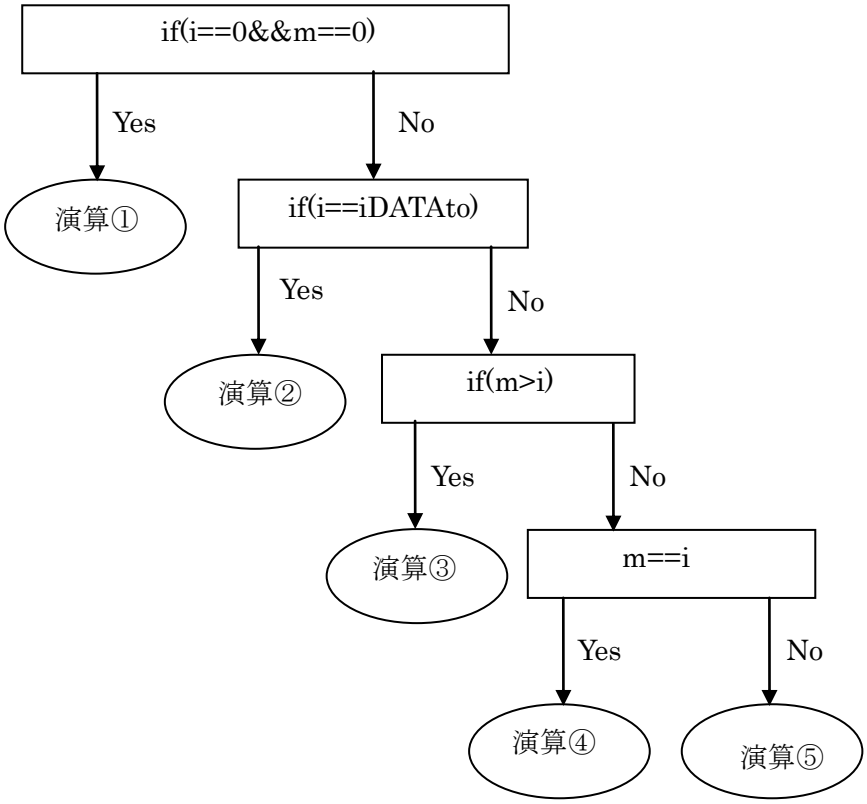

図 12：手書きによる状態遷移のモデル

3.3 手書きとシステムによる生成回路の比較

動作合成システムの実験環境として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

回路シミュレーション	PC 環境	Intel Core2 duo 2.66Ghz,Memory 4GB
	シミュレーションツール	ModelSim SE 5.8c
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 8.2i

素数の判定には 100003 という素数である値を用いた．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 2 に示す.

表 2：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	19.4	1.0
2	11.5	1.7
4	8.2	2.4

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，ノード数に対する速度向上比の伸びが小さくなっていることがわかる.

回路シミュレーションを用いて測定した，シミュレーション時間と実行にかかったクロックサイクル数を表 3 に示す. 括弧内の値はそれぞれの回路のノード 1 に対する比を表す.

表 3：シミュレーション時間と実行クロック数

ノード数	シミュレーション時間[s]		実行クロックサイクル数[Mclocks]	
	手書き	システム	手書き	システム
1	20.4(1.0)	70.5(1.0)	1.06(1.0)	3.35(1.0)
2	21.0(1.0)	78.4(1.1)	0.98(0.9)	3.02(0.9)
4	21.8(1.1)	99.2(1.4)	0.90(0.8)	2.75(0.8)

動作クロックサイクル数については，ノードの増加に対して大きな減少が見込めなかった．これは図より i の値が大きいほど減算の回数は減り，処理回数は減るため，ノードによって仕事量が不均一になったからだと考えられる．手書きとシステムの回路と比較してみるとシミュレーション時間，クロックサイクル数ともに手書きのほうが良い値が得られた．これは 3.2 でも述べたように，1 クロックにおける演算量が違うことが原因と思われる．シミュレーション時間については，ノードの増加によって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，PC にかかる負荷が増加してしまうためにあまり変わらなかったと思われる．表 2 の SMP 環境での実行時間と比べると，手書き，システムともに回路シミュレーション時間はほぼ千倍近くの時間がかかっている．すなわち，SMP 環境を用いることで高速に並列アルゴリズムやプログラムの検証を行えることが確認できる．

論理合成による最大動作周波数と回路面積を表 4 に示す．括弧内の値は表 3 と同様にノード 1 に対しての比を表す．

表 4：動作周波数と回路規模

ノード数	最大動作周波数[MHz]		スライス数[slices]	
	手書き	システム	手書き	システム
1	162.502	188.886	129(1.0)	148(1.0)
2	162.056	188.886	272(2.1)	282(1.9)
4	161.962	188.886	589(4.5)	434(2.9)

回路面積は手書き，システム間では大きな違いは見られなかった．ただノード 4 に注目してみると，システムの生成する回路面積のほうが小さくなっている．このことより，手書きよりもシステムのほうが，ノード数を増やしたときの回路面積の増加量が少ないことが考えられる．最大動作周波数においては，システム側のほうが高くなった．状態の遷移において，手書き側は if 文の入れ子による条件分岐によりは 1 クロックの周期が大きくなってしまうのに対し，システム側では case 文による一度の判定で状態遷移を行うため 1 クロックの周期が小さくなる．また 1 クロック 1 演算しか行わないことも原因に含め，システム側の動作周波数が高くなったと考えられる．

4.ラプラシアンフィルタに対するハードウェア動作合成

4.1 ラプラシアンフィルタのアルゴリズム

ラプラシアンフィルタとは画像の鮮鋭化やエッジ検出に使用されるフィルタで，二次微分を行うことで画像の明るさや色の変化を強調し検出を行う．実験では縦と横，斜めを考慮した図 13 のオペレータを用いた. OpenMP のプログラム上では 8 近傍の画素値を足し合わせた値と対象とする画素を 8 倍した値との差をとっている．

実装したラプラシアンフィルタプログラムのフローチャートを図 14 に示す．対象画像のすべての画素に対して，その近傍画素との演算を行い，演算結果を保存用のメモリに出力する．入力画像と出力画像は，ともに輝度が 0~255 の濃淡画像を想定していたため，微分演算の後に演算結果を輝度の大きさに正規化している．対象とする画像空間を分割し，各空間に対する処理を並列に実行することでデータ分割による並列化を行った. 図 15 は図 14 のフローチャートに基づき, OpenMP を用いたラプラシアンフィルタのプログラムである．図 15 のプログラムより生成した中間表現を付録 C，一部省略したものを図 16 に示す．入力画素値を in_image_data に代入し，出力画素値を out_image_data に代入する．画像の入力は素数判定プログラムのときと同様に，HDL 上で制御する．

1	1	1
1	−8	1
1	1	1

図 13：ラプラシアンフィルタのオペレータ

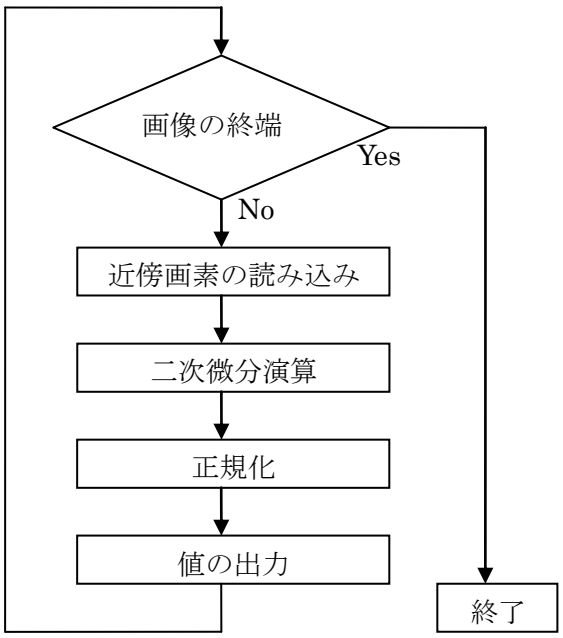

図 14：ラプラシアンフィルタのフローチャート

```
int main(void ){
int i,j,l,m, buf_num;
int out_image_data[1024][1024];
int in_image_data[1024][1024];
#pragma omp parallel for private(i,j,l,buf_num)
for(i=0;i<1024;i++){
for(j=0;j<1024;j++){
      if((i>0 && i<1023) && (j>0 && j<1023)){
        buf_num   = 0;
         l = i-1; m = j-1;
        buf_num += in_image_data[l][m];
         l = i-1;  m = j;
         buf_num += in_image_data[l][m];
         l = i-1; m = j+1;
        buf_num += in_image_data[l][m];
         l = i; m = j-1;
        buf_num += in_image_data[l][m];
         l = i; m = j+1;
        buf_num += in_image_data[l][m];
         l = i+1; m = j-1;
        buf_num += in_image_data[l][m];
         l = i+1; m = j;
         buf_num += in_image_data[l][m];
         l = i+1; m = j+1;
        buf_num += in_image_data[l][m];
         l = i; m = j;
         in_image_data[l][m]=in_image_data[l][m]*8;
         if(buf_num <= in_image_data[l][m])        buf_num = 0;
         else     buf_num=buf_num-in_image_data[l][m];
         if(buf_num >= 255)        out_image_data[i][j] = 255;
         else     out_image_data[i][j] = buf_num;
       }else     out_image_data[i][j] = in_image_data[i][j];
    }
  }
}
```

図 15：OpenMP を用いたラプラシアンフィルタ

```
0 : Auto Signed 32bit : <function> main( )
1 : Auto 0bit : null
2 : Auto Signed 32bit : i
3 : Auto Signed 32bit : j
4 : Auto Signed 32bit : l
5 : Auto Signed 32bit : m
6 : Auto Signed 32bit : buf_num
7 : Auto Signed 32bit : out_image_data[1048576][1024]
8 : Auto Signed 32bit : in_image_data[1048576][1024]
9 : Auto Const Signed 32bit : *9 := 768
～～～(省略)～～～
45 : Auto Signed 32bit  : ARRAY( 8 4 5 )
46 : Auto Signed 32bit  : +( 6 45 )
47 : Auto Signed 32bit  : =( 6 46 )
～～～ (省略) ～～～
```

シンボルテーブル

```
#0  : [ [ 10 ] ] -> #2
#1  : [ ]  <- #0
#2  : [ [ 12 ] ] -> 12 ? #3 : #1
#3  : [ [ 15 ] ] -> #5
#4  : [ [ 13 ] ] -> #2
#5  : [ [ 17 ] ] -> 17 ? #6 : #4
#6  : [ [ 20 ] [ 22 ] [ 23 ] [ 25 ] [ 27 ] [ 28 ] [ 29 ] ]->29 ? #8 : #14
#7  : [ [ 18 ] ] -> #5
#8  : [ [ 31 ] [ 33 ] [ 34 ] [ 36 ]～～(一部省略)～～[ 102 ] [ 103 ] ] -> #9
#9  : [ [ 105 ] ] -> 105 ? #10 : #11
#10  : [ [ 106 ] [ 108 ] ] -> #7
#11  : [ [ 110 ] ] -> 110 ? #12 : #13
#12  : [ [ 111 ] [ 113 ] ] -> #7
#13  : [ [ 114 ] [ 115 ] ] -> #7
```

状態遷移表

図 16：ラプラシアンフィルタの中間表現(一部省略)

4.2 手書きとシステムによる HDL 記述の生成

この実験では，入力と出力をメモリに格納しているため，メモリのアクセスを管理するアービタが必要となる．アービタとは，複数のノードが一つのメモリに対し同時にアクセスすることでデータ競合が起こってしまわないように，各ノードにストールを発生させることで防ぐ役割をするものである．アービタのモジュールは前の研究で用いられていたものをそのまま使っている[4].

システムを利用するにあたって中間表現，HDL 記述は問題なく生成された．並列化も今回は扱う画像サイズを 1024×1024 と決めていたため，1024 という定数を分割するので，素数判定のときのようなエラーはなかった.HDL 記述の行数は,手書きでは 264 行に対し，システム側では 1052 行と約 4 倍になった．今回手書き側は状態遷移に case 文を用いているため，システム側と酷似したものとなった．しかし，状態の数はシステム側では 190 個生成されたのに対し，手書きは 18 個と約 10 分の 1 の量になった．図 17 ではアクセスするメモリアドレスを計算し，メモリからのデータを読み込む部分を抜粋している．oADDR はアクセスするメモリのアドレス，oRD は読み込み応答信号，iSTALL はストール信号である．図 17 の P_STATE__ARRAY_R0，R1,，R2 ではアクセスするメモリのアドレスを計算している．OpenMP プログラムでは 2 次元配列を扱っていたが，回路上では一次元配列のメモリを扱っているため，それに対応したアドレスを算出するため加算と乗算を行う.. 図 17 を見比べると，システム側ではメモリからデータを取り出し，加算を行うのに大体 5 クロックかかるのに対し，手書き側ではメモリのアクセスから加算までに大体 2 クロックで実現している.

```
P_STATE45_ARRAY_R0 : REG45_R0 <= ADD1_RESULT;
P_STATE45_ARRAY_R1 : REG45_R1 <= MUL1_RESULT;
P_STATE45_ARRAY_R2 : REG45_R2 <= ADD1_RESULT;
P_STATE45_ARRAY_R3 : begin
                     oADDR <= REG45_R2;
                     oRD <= 1'b1;
                     end
P_STATE45_ARRAY_STALL :
if(iSTALL!=1'b1) begin
                     oRD <= 1'b0;
                     REG45 <= iDATA;
                     end else begin
                     oRD <= 1'b1;
                     end
```

システムの生成回路の代入部

```
32'd3: begin
oADDR <=address +i+(j-1)*width;
         oRD <= 1'b1;
         state <= state + 32'd1;
end
32'd4: begin
if(iSTALL!=1'b1) begin
         oRD <= 1'b0;
buf_num <= buf_num + iDATA;
         state <= state + 32'd1;
         end else begin
         oRD <= 1'b1;
         end
end
```

手書きの生成回路の演算部

図 17：システム生成回路の代入部，手書き生成回路の演算部(一部抜粋)

4.3 手書きとシステムによる生成回路の比較

実験は素数判定のときと同様に表 1 の環境で行った.

ラプラシアンフィルタに用いた画像は，解像度 1024×1024，輝度 0~255 の pgm 画像である．ラプラシアンフィルタの 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 5 に示す.

表 5：ラプラシアンフィルタの 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	40.4	1.0
2	22.5	1.8
4	14.6	2.8

ノード数が 1 の場合は逐次実行を示している．素数判定のときと同様に，ノード数が増えるに従い，ノード数に対する速度向上比の伸びが小さくなっていることがわかる.

回路シミュレーションを用いて測定した，シミュレーション時間と実行にかかったクロックサイクル数を表 6 に，手書きとシステムとの各ノードでのクロック比を表 7 にそれぞれ示す．表 6 の括弧内の値はそれぞれの回路のノード 1 に対する比を表す.

表 6：シミュレーション時間と実行クロック数

ノード数	シミュレーション時間[s]		実行クロックサイクル数[Mclocks]	
	手書き	システム	手書き	システム
1	1828(1.0)	13018(1.0)	21.95(1.0)	110.76(1.0)
2	1251(0.7)	11449(0.9)	10.97(0.5)	55.38(0.5)
4	1809(1.0)	10741(0.8)	10.98(0.5)	27.74(0.3)

手書きとシステムの回路とで比較すると，素数判定のときと同様にシミュレーション時間，クロックサイクル数ともに手書きのほうが良い値が得られた．しかし，動作クロックサイクル数の比に着目すると，ノードの増加により手書き側ではノード 2 と 4 との間に大きな減少がなかったが，システム側では理想的な比率を得ることができた．このことよりシステムで生成される回路のほうが並列性に優れていることが考えられる.

シミュレーション時間については，素数判定のときと同様の原因によりあまり変わらなかったものと考えられる．表 5 の SMP 環境での実行時間と比べると，システムの回路シミュレーション時間は 30 万~70 万倍近くの時間がかかっている．これにより，回路上でシミュレーションを行うより，SMP 環境上でシミュレーションを行うほうが莫大な時間短縮を行えることが期待できる.

論理合成による最大動作周波数と回路面積を表 7 に示す．括弧内の値は表 6 と同様にノード 1 に対しての比を表す.

表 7：動作周波数と回路規模

ノード数	最大動作周波数[MHz]		スライス数[slices]	
	手書き	システム	手書き	システム
1	104.770	70.494	642(1.0)	2445(1.0)
2	106.558	70.494	1227(2.0)	4832(2.0)
4	97.930	70.494	2329(3.6)	9484(3.9)

回路面積において，システムはノードの増加に対し同等の増加量を示しているが，手書きではノード 4 においてはスライス数の増加量は小さかった．また手書きに対してシステム側では約 4 倍の回路が生成されている．これは演算を行うときにデータの中継ぎをするレジスタやワイヤを生成していることや，状態の遷移数が多いため配線の数が多くなってしまうことが原因と思われる．動作周波数については，素数判定のときと違い手書きのほうが高い値となった.

5. OpenMP 動作合成システムの評価

素数判定，ラプラシアンフィルタの二つのアプリケーションに本システムを適用するにあたって，トランスレータによる中間表現の生成は状態遷移表，シンボルテーブルともに正確に行われており，特に問題点は無かった．しかし，手書きの生成回路と比べて速度，回路規模ともに大きく差があった．この問題点から改良点を挙げ，システム全体の考察を行い評価した．

5.1 コードジェネレータの改良点

3 章では素数判定プログラムをコードジェネレータで HDL を生成したところ，自動的にデータ分割ができておらず手動で修正をした．原因としては，データ分割する対象の値が変数の場合はコードジェネレータが対応していなかったためであった．そこで図 9 のようなデータ分割を自動で行えるようにコードジェネレータの修正を行った．iDATAstart,iDATAend というデータの範囲を自動生成することはできたが，それらのデータを扱う際に正しく動作していない点があるためまだデバグの必要がある．

コードジェネレータの生成する回路性能の向上方法として，1 状態の演算量の増加が挙げられる．例えば，現在コードジェネレータが生成する回路では”i=i+a”は図 18 のような回路が生成される．

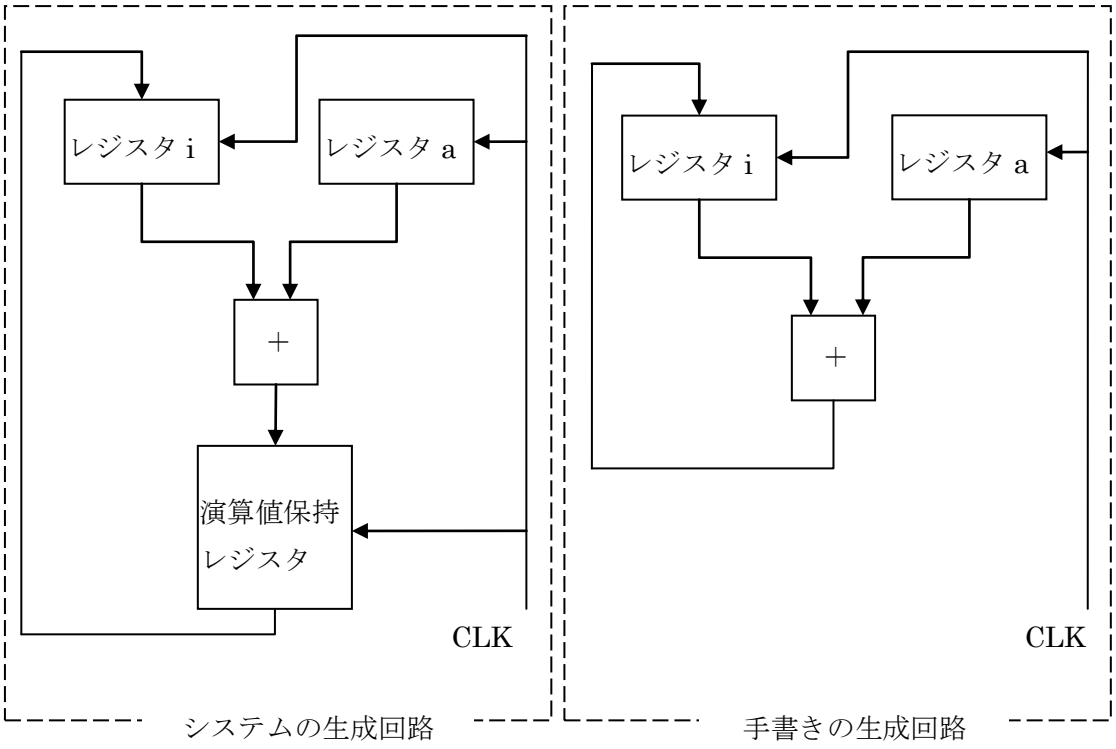

図 18：システム，手書きでの i=i+a の演算回路

この図では，システムの生成回路では加算してから代入するまでに 2 クロック必要だが，手書きでは加算と代入を 1 クロックで実現できている．このように，システムが図 18 の手

書きの回路のように生成することができれば，実行クロック数は減少し速度の向上が期待できる．また，状態遷移数が減ることで配線数が減少，さらに演算値を一時的に保持するためのレジスタも不要になるため，回路規模の縮小も実現できる．．これらの改良を実現するには，中間表現の解析時に遷移先の演算が同時に実行できるかを，コードジェネレータが判断できるように修正すればよい

5.2 考察

素数判定，ラプラシアンフィルタの二つのアプリケーションに本システムを適応するにあたって，トランスレータによる中間表現の生成は状態遷移表，シンボルテーブルともに正確に行われており，特に問題点は無かった．コードジェネレータにおいて，ラプラシアンフィルタは手書きのものに対し，システムの生成回路は状態遷移数が 5 倍ほどあるが，状態の遷移方法としてはほぼ同じである．そのため図 18 を用いて述べた改良が実現すればシステムの生成回路はより優れたものになることが期待できる．だが，1 状態での演算の量が増えるということは遅延時間の増加の可能性があり，動作周波数が下がることで結果的に速度の向上が見込めないケースも考えられる．今回実験した，ラプラシアンフィルタでは手書きの生成回路のほうが動作周波数は高い値を示しているため，そのようなことにはならなかったが，他のアプリケーションについても検証し、実行クロック数と動作周波数とのトレードオフを考察する必要がある．

6. おわりに

本研究では，素数判定プログラムとラプラシアンフィルタの OpenMP プログラムから本システムを用いて動作合成を行い，システムと手書きそれぞれで生成した回路の比較し，システム全体の評価を行った．回路規模において，素数判定では大きな差異がみられなかったが，ラプラシアンフィルタでは手書きとシステムの回路ともにノード増加に比例する回路規模となり，システムの回路は手書きのものに対し 4 倍の大きさになった．速度向上においては，素数判定ではアルゴリズムの性質上システムと手書きのものは同等の比率になったが，ラプラシアンフィルタにおいてはシステムの生成する回路のほうが並列度に対し理想的な速度向上を得られた．

1 状態 1 演算ではなく，1 状態で適度な演算を行えるようにコードジェネレータを改良し，生成回路の速度向上と回路規模の縮小化を計ることが今後の課題である．

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，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中谷嵩之氏、松崎裕樹氏に深く感謝いたします．

最後に，共同研究者である苅屋徹氏をはじめ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] 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.

[2] 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.55-60 , 2005.

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

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

[5] 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.

[6] 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.

[7] 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.

付録 A システムによる素数判定プログラムの HDL 記述

```
module sosu_omp( iSTART, oEND, ans, iDATA, CLK, XRST);
 input iSTART;
 output oEND;
 reg oEND;
output[31:0] ans;
 input [31:0]iDATA;
 input CLK,XRST;


 reg [7:0]CurrentState;

 wire [31:0] iDATAstart,iDATAend; //手書き修正


assign iDATAstart = iDATA/2/4*0;        //データ分割(手動)
assign iDATAend = iDATA/2/4*1;   //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]ans;

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;

//演算器部
```

```
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;
   ans <= 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 <= iDATAstart;      //手書き修正
    P_STATE9 : i <= ADD1_RESULT;
    P_STATE13 : i <= ConstNum12;
    P_STATE14 : m <= iDATA;
    P_STATE16 : REG16 <= SUB1_RESULT;
    P_STATE17 : m <= REG16;
    P_STATE21 : ans <= ConstNum20;
    default : oEND <= 1'b0;
  endcase
 end
end

//状態遷移部
always @(posedge CLK or negedge XRST) begin
  if(!XRST)
    CurrentState <= P_INIT;
  else
     case(CurrentState)
        P_STATE7: CurrentState <= P_STATE8;
        P_INIT   : if(iSTART==1'b1) CurrentState <= P_STATE7;
                   else CurrentState <= CurrentState;
        P_END    : CurrentState <= CurrentState;
        P_STATE8: if(i<iDATAend) CurrentState <= P_STATE11;    //手書き修正
                   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
```

付録 B 手書きによる素数判定プログラムの HDL 記述

```
module sosu_tegaki( iSTART, oEND, ans, iDATA, CLK, XRST);
 input iSTART;
 output oEND;
 reg oEND;
 output[31:0] ans;
 reg [31:0] ans;
 input [31:0]iDATA;
 input CLK,XRST;
reg[31:0] i,m;

 wire [31:0] iDATAstart,iDATAend;

assign iDATAstart = iDATA/2/4*0;        //データ分割(手動)
assign iDATAend = iDATA/2/4*1;   //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;
    ans <=32'b0;
end else begin

    if(i==0&&m==0)begin
        i <= iDATAstart;
        m <= iDATA;
    end else if(i==iDATAend)begin
        oEND <=1'b1;
    end else begin
        if(m>i)begin
            m <= m-i;
        end else if(m==i) begin
            ans<=1;
            oEND <=1'b1;
        end else begin
            m <= iDATA;
            i <= i+1;
        end
    end
end

endmodule
```

付録 C ラプラシアンフィルタの中間表現

```
----SemanticsAnalyze----
function 0 : main
0 : Auto Signed 32bit : <function> main( )
1 : Auto 0bit : null
2 : Auto Signed 32bit : i
3 : Auto Signed 32bit : j
4 : Auto Signed 32bit : l
5 : Auto Signed 32bit : m
6 : Auto Signed 32bit : buf_num
7 : Auto Signed 32bit : out_image_data[1048576][1024]
8 : Auto Signed 32bit : in_image_data[1048576][1024]
9 : Auto Const Signed 32bit : *9 := 768
10 : Auto Signed 32bit  : =( 2 9 )
11 : Auto Const Signed 32bit : *11 := 1024
```

```
12 : Auto Signed 32bit  : <( 2 11 )
13 : Auto Signed 32bit  : ( 2 )++
14 : Auto Const Signed 32bit : *14 := 0
15 : Auto Signed 32bit  : =( 3 14 )
16 : Auto Const Signed 32bit : *16 := 1024
17 : Auto Signed 32bit  : <( 3 16 )
18 : Auto Signed 32bit  : ( 3 )++
19 : Auto Const Signed 32bit : *19 := 0
20 : Auto Signed 32bit  : >( 2 19 )
21 : Auto Const Signed 32bit : *21 := 1023
22 : Auto Signed 32bit  : <( 2 21 )
23 : Auto Signed 32bit  : &&( 20 22 )
24 : Auto Const Signed 32bit : *24 := 0
25 : Auto Signed 32bit  : >( 3 24 )
26 : Auto Const Signed 32bit : *26 := 1023
27 : Auto Signed 32bit  : <( 3 26 )
28 : Auto Signed 32bit  : &&( 25 27 )
29 : Auto Signed 32bit  : &&( 23 28 )
30 : Auto Const Signed 32bit : *30 := 0
31 : Auto Signed 32bit  : =( 6 30 )
32 : Auto Const Signed 32bit : *32 := 1
33 : Auto Signed 32bit  : -( 2 32 )
34 : Auto Signed 32bit  : =( 4 33 )
35 : Auto Const Signed 32bit : *35 := 1
36 : Auto Signed 32bit  : -( 3 35 )
37 : Auto Signed 32bit  : =( 5 36 )
38 : Auto Signed 32bit  : ARRAY( 8 4 5 )
39 : Auto Signed 32bit  : +( 6 38 )
40 : Auto Signed 32bit  : =( 6 39 )
41 : Auto Const Signed 32bit : *41 := 1
42 : Auto Signed 32bit  : -( 2 41 )
43 : Auto Signed 32bit  : =( 4 42 )
44 : Auto Signed 32bit  : =( 5 3 )
45 : Auto Signed 32bit  : ARRAY( 8 4 5 )
46 : Auto Signed 32bit  : +( 6 45 )
47 : Auto Signed 32bit  : =( 6 46 )
48 : Auto Const Signed 32bit : *48 := 1
49 : Auto Signed 32bit  : -( 2 48 )
50 : Auto Signed 32bit  : =( 4 49 )
51 : Auto Const Signed 32bit : *51 := 1
52 : Auto Signed 32bit  : +( 3 51 )
53 : Auto Signed 32bit  : =( 5 52 )
54 : Auto Signed 32bit  : ARRAY( 8 4 5 )
55 : Auto Signed 32bit  : +( 6 54 )
56 : Auto Signed 32bit  : =( 6 55 )
57 : Auto Signed 32bit  : =( 4 2 )
58 : Auto Const Signed 32bit : *58 := 1
59 : Auto Signed 32bit  : -( 3 58 )
60 : Auto Signed 32bit  : =( 5 59 )
61 : Auto Signed 32bit  : ARRAY( 8 4 5 )
62 : Auto Signed 32bit  : +( 6 61 )
63 : Auto Signed 32bit  : =( 6 62 )
64 : Auto Signed 32bit  : =( 4 2 )
65 : Auto Signed 32bit  : =( 5 3 )
66 : Auto Signed 32bit  : ARRAY( 8 4 5 )
67 : Auto Const Signed 32bit : *67 := 8
68 : Auto Signed 32bit  : -( 67 )
69 : Auto Signed 32bit  : *( 66 68 )
70 : Auto Signed 32bit  : +( 6 69 )
71 : Auto Signed 32bit  : =( 6 70 )
72 : Auto Signed 32bit  : =( 4 2 )
73 : Auto Const Signed 32bit : *73 := 1
74 : Auto Signed 32bit  : +( 3 73 )
75 : Auto Signed 32bit  : =( 5 74 )
76 : Auto Signed 32bit  : ARRAY( 8 4 5 )
77 : Auto Signed 32bit  : +( 6 76 )
78 : Auto Signed 32bit  : =( 6 77 )
79 : Auto Const Signed 32bit : *79 := 1
80 : Auto Signed 32bit  : +( 2 79 )
81 : Auto Signed 32bit  : =( 4 80 )
82 : Auto Const Signed 32bit : *82 := 1
83 : Auto Signed 32bit  : -( 3 82 )
```

```
84 : Auto Signed 32bit  : =( 5 83 )
85 : Auto Signed 32bit  : ARRAY( 8 4 5 )
86 : Auto Signed 32bit  : +( 6 85 )
87 : Auto Signed 32bit  : =( 6 86 )
88 : Auto Const Signed 32bit : *88 := 1
89 : Auto Signed 32bit  : +( 2 88 )
90 : Auto Signed 32bit  : =( 4 89 )
91 : Auto Signed 32bit  : =( 5 3 )
92 : Auto Signed 32bit  : ARRAY( 8 4 5 )
93 : Auto Signed 32bit  : +( 6 92 )
94 : Auto Signed 32bit  : =( 6 93 )
95 : Auto Const Signed 32bit : *95 := 1
96 : Auto Signed 32bit  : +( 2 95 )
97 : Auto Signed 32bit  : =( 4 96 )
98 : Auto Const Signed 32bit : *98 := 1
99 : Auto Signed 32bit  : +( 3 98 )
100 : Auto Signed 32bit  : =( 5 99 )
101 : Auto Signed 32bit  : ARRAY( 8 4 5 )
102 : Auto Signed 32bit  : +( 6 101 )
103 : Auto Signed 32bit  : =( 6 102 )
104 : Auto Const Signed 32bit : *104 := 255
105 : Auto Signed 32bit  : >=( 6 104 )
106 : Auto Signed 32bit  : ARRAY( 7 2 3 )
107 : Auto Const Signed 32bit : *107 := 255
108 : Auto Signed 32bit  : =( 106 107 )
109 : Auto Const Signed 32bit : *109 := 0
110 : Auto Signed 32bit  : <=( 6 109 )
111 : Auto Signed 32bit  : ARRAY( 7 2 3 )
112 : Auto Const Signed 32bit : *112 := 0
113 : Auto Signed 32bit  : =( 111 112 )
114 : Auto Signed 32bit  : ARRAY( 7 2 3 )
115 : Auto Signed 32bit  : =( 114 6 )
116 : Auto Signed 32bit  : ARRAY( 7 2 3 )
117 : Auto Signed 32bit  : ARRAY( 8 2 3 )
118 : Auto Signed 32bit  : =( 116 117 )
Argument  ( 1 )
{
--#0  : { /0 } -> #1
--#1  : [ ]  <- #0
}
/0 Parallel FOR (2) ( P[ 2 3 4 6 ]  )
-0:#0  : [ [ 10 ] ] -> #2
-0:#1  : [ ]  <- #0
-0:#2  : [ [ 12 ] ] -> 12 ? #3 : #1
-0:#3  : [ [ 15 ] ] -> #5
-0:#4  : [ [ 13 ] ] -> #2
-0:#5  : [ [ 17 ] ] -> 17 ? #6 : #4
-0:#6  : [ [ 20 ] [ 22 ] [ 23 ] [ 25 ] [ 27 ] [ 28 ] [ 29 ] ] -> 29 ? #8 : #14
-0:#7  : [ [ 18 ] ] -> #5
-0:#8  : [ [ 31 ] [ 33 ] [ 34 ] [ 36 ] [ 37 ] [ 38 ] [ 39 ] [ 40 ] [ 42 ] [ 43 ] [ 44 ] [ 45 ] [ 46 ]
[ 47 ] [ 49 ] [ 50 ] [ 52 ] [ 53 ] [ 54 ] [ 55 ] [ 56 ] [ 57 ] [ 59 ] [ 60 ] [ 61 ] [ 62 ] [ 63 ]
[ 64 ] [ 65 ] [ 66 ] [ 68 ] [ 69 ] [ 70 ] [ 71 ] [ 72 ] [ 74 ] [ 75 ] [ 76 ] [ 77 ] [ 78 ] [ 80 ]
[ 81 ] [ 83 ] [ 84 ] [ 85 ] [ 86 ] [ 87 ] [ 89 ] [ 90 ] [ 91 ] [ 92 ] [ 93 ] [ 94 ] [ 96 ] [ 97 ]
[ 99 ] [ 100 ] [ 101 ] [ 102 ] [ 103 ] ] -> #9
-0:#9  : [ [ 105 ] ] -> 105 ? #10 : #11
-0:#10  : [ [ 106 ] [ 108 ] ] -> #7
-0:#11  : [ [ 110 ] ] -> 110 ? #12 : #13
-0:#12  : [ [ 111 ] [ 113 ] ] -> #7
-0:#13  : [ [ 114 ] [ 115 ] ] -> #7
-0:#14  : [ [ 116 ] [ 117 ] [ 118 ] ] -> #7
```