修士論文

OpenMP 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

氏名	:	松崎 裕樹
学籍番号	:	6162060116-0
指導教員	:	山崎 勝弘 教授
提出日	:	2008 年 2 月 14 日

立命館大学大学院　理工学研究科　創造理工学専攻

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に対して，コードジェネレータの設計・実装と画像処理による評価を行っている．

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，OpenMP プログラムから生成された中間表現に対し，合成するハードウェアのモデルを設定し，中間表現の解釈を定め，それに従い verilog のソースコードの生成を行うコードジェネレータを実装した．また，エッジ検出及びハフ変換の画像処理プログラムを OpenMP で記述し, 本システムを用いて動作合成を行った. それらのプログラムの PC クラスタ上での実行結果と，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．エッジ検出に対しては，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た．エッジ検出のようなデータに依存しないアルゴリズムでは，動作合成により理想的な速度向上が得られることがわかった．また，ハフ変換に対してはデータ並列を用いて４ノードでおよそ２倍の速度向上が得られ，アルゴリズム評価系におけるシミュレーションと相関のある結果となった．

目次

図目次

表目次

1. はじめに

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．現在では，大規模計算機やパーソナルコンピュータなどの計算機だけではなく，携帯電話，家電，車など様々な機器に LSI は用いられ，その用途は多岐に渡っている．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，開発工数の増大，開発工程の複雑化を招いている．

その一方で，激しい市場競争による開発サイクルの短期化，コスト削減による人的投資の困難と人材の不足が進み，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，HDL を用いてハードウェアの設計記述を行う従来の設計手法から，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し，LSI の回路構造を動作の記述から自動合成する動作(高位)合成では，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，設計段階で様々な仕様の変更も容易に対応できるようになる．

C 言語からの高位合成では，状態遷移やデータパスの合成，リソース割り当てなどが自動的に行われ，設計者はクロックタイミングや回路の詳細構成を意識することなく，処理の流れのみを記述すればよい．一方で，従来の C 言語は空間的な概念，並列動作の概念が含まれておらず，面積，性能，消費電力の面で，自動合成で最適なハードウェアが合成されるとは限らない[6]．特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，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．実際の動作合成技術においても，SpecC 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による並列化の最適化では，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．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や，シミュレーションの負荷が大きく，テスト検証にかかる時間が大きくなるといった問題がある[6]．

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を提案し，中間表現からのコードジェネレータの設計と画像処理のプログラムに対する評価を行うことを目的とする．

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り，段階的にかつシームレスに並列化を行うことができる．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.

OpenMP は 1997 年に Fortran 向けに最初の規格が発表され，その後継続的にヴァージョンアップがされている業界標準規格である．現在では，Microsoft Visual Studio や Intel C/C++ Compiler など様々な言語でサポートされている[16].

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，本システムにおけるコードジェネレータの設計・実装と画像処理による評価を行う．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が実装されており，FIR フィルタやウェーブレット変換などの信号処理に対し，データ分割において理想的な速度向上が得られている[1]．本研究では，トランスレータによって生成された中間表現から，対応するハードウェア(HDL)を動作合成し，システム全体を統合させることを目的としている．動作合成においては，生成されるハードウェアを想定し，それに合わせた中間コードの解釈を定め，verilog のコードを生成する．また，画像処理アルゴリズムであるエッジ検出，及びハフ変換に対し，本システムを用いて動作合成を行い，コードジェネレータと本システム全体の評価を行う．SMP クラスタによる OpenMP で書かれた画像処理プログラムの実行結果やシミュレーション時間と動作合成されたハードウェアの回路規模やシミュレーション時間について比較し，本システムの有効性を考察している.

本論文では，第 2 章において OpenMP の概要，及び OpenMP を用いたハードウェア動作合成システムの構成とデータ並列やタスク並列の動作モデルを示す．第 3 章ではコードジェネレータの概要と生成モジュールについて説明を行う．第 4 章ではエッジ検出に対して，第 5 章ではハフ変換に対する動作合成の実験結果を示し，コードジェネレータ及び本システム全体の考察を行う.

2. OpenMP を用いたハードウェア動作合成システム

2.1 OpenMP の概要と実行モデル

OpenMP は，SMP 環境における並列プログラミングの標準 API である．C,C++,Fortran をベースとし，プログラムに並列化指示文であるプラグマを挿入することで並列プログラミングを実現する．OpenMP は自動並列化ではなく，データの依存性やアルゴリズムの並列性はプログラマが解析し，並列化指示文であるプラグマを用いて指定する．

OpenMP の並列化指示文は, 仕様により規定された並列実行モデルへの API を提供する．そのため，仕様に基づいた記述を行うことにより，異なる環境下でも同一の実行結果が得られ，移植性が高いプログラムが作成できる．

OpenMP の特徴を以下に示す．

- 既存の逐次プログラムに並列化指示文を挿入することで，逐次プログラムから並列プログラムに段階的に，かつシームレスに並列化が可能である．
- 自動で並列化が実行されないため，設計者が並列化部の抽出や分析を行う．
- 共有変数への同時アクセスや同期・排他制御は設計者が意図しなくてよい．
- 並列性の自動抽出が難しい大規模な処理に対する並列化に適している．
- 共有メモリモデルであるため，データのノードへのマッピングが不要である．

MPI や PVM などの DMP 環境の API と違い，メッセージ・パッシングや共有変数への同期・排他制御を意図しなくてよい点が他の並列プログラミングの API と大きく異なる点である．

マスタースレッド

OpenMP の実行モデルである fork-join モデルを図 1 に示す．

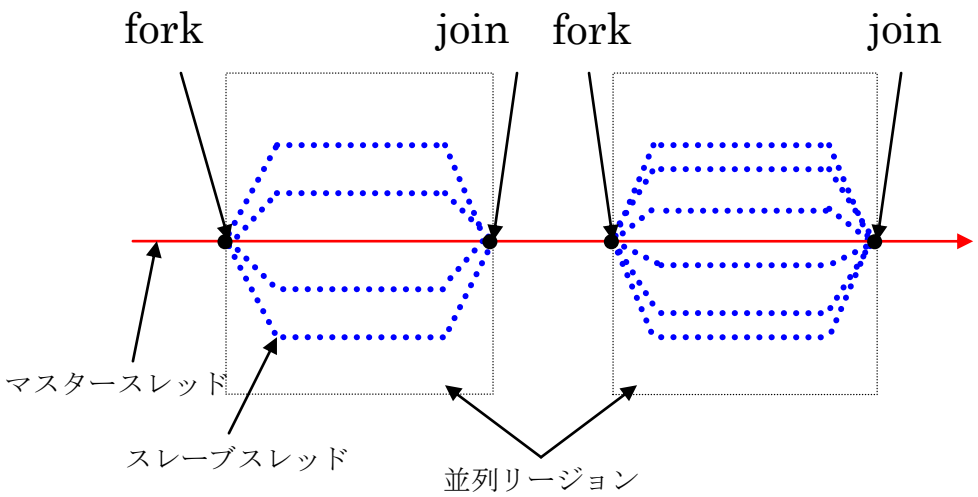


図 1：OpenMP の fork-join モデル

プログラム中における逐次処理の部分に対して，マスタースレッドが実行される．プログラムの実行が並列化指示文によって指定された並列実行部に達すると，マスタースレッ

ドがスレーブスレッドを生成し，各スレッドがプロセッサ上で並列に実行される．このスレーブスレッドの生成を fork と呼ぶ．各スレーブスレッドの処理が完了した後，マスタースレッドが各スレッドの実行結果を回収し，マスタースレッドのみが逐次処理へ復帰しプログラムの実行を継続する．このマスタースレッドとスレーブスレッドの同期制御を join と呼ぶ.

以下に代表的な並列化手法である，データ並列とタスク並列の実行モデルを示す.

(1) データ並列の実行モデル

データ並列は，大量のデータに対して処理を行う大規模なデータの分割を対象として行われ，主にループの並列化に用いられる．処理において，各データに対してアルゴズム上の依存関係がない場合に用いられる．データ並列によるループの並列化では，OpenMP の並列化指示文である “#pragma omp for” を並列化したい for ループの直前に挿入する．100 個の配列要素の値を 2 倍し格納する処理をデータ並列により並列化した記述例を　図 2 に，4 ノードで実行した場合の実行モデルを図 3 に示す.

```
#pragma omp parallel for //並列化指示文
for(i=0;i<=99;i++){ //並列化されるループ
   Array[i] = Array[i] * 2; //2 倍の計算
}
```

図 2：データ並列の記述例

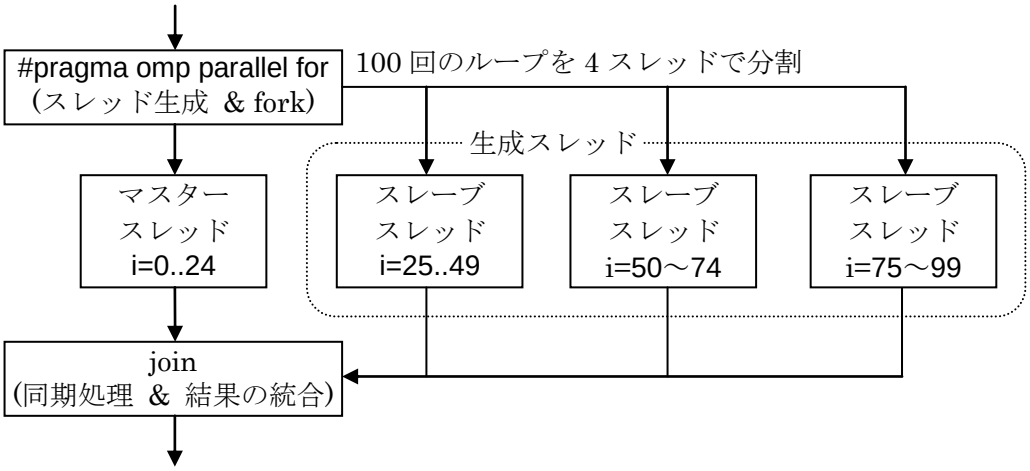


図 3：データ並列の実行モデル

ループ内の演算は依存関係がなくデータ並列性があるため，ループの変数である i の値の変化を 4 分割し，スレッドを生成して各ノードで実行している．OpenMP では同期や排他制御を記述する必要はないが，ループ内の変数や関数に対するアルゴリズム上のデータ依存性の正当性については，ユーザが保証しなければならない．また，join ではすべてのスレッドが終了しなければならないため，各スレッドが均等に負荷分散されることが理想である.

(2) タスク並列の実行モデル

タスク並列では，並列に実行可能な複数の処理を，各スレッドに割り当てて実行する．タスク並列による並列化では，並列リージョンの直前に“#pragma omp parallel sections”を挿入し，並列実行する処理を“#pragma omp parallel section”を用いて指定する．関数 funcA，funcB，funcC をタスク並列で記述した例を図 4 に，3 ノードで実行した場合の実行モデルを図 5 に示す．タスク並列で並列化効果を出すためには，各 func の負荷をできるだけ均衡させることが必要である．

```
#pragma omp parallel sections //並列化指示文
{//並列リージョン
#pragma ompsection
{funcA}
#pragma ompsection
{funcB}
#pragma ompsection
{funcC}
}
```

図 4：タスク並列の記述例

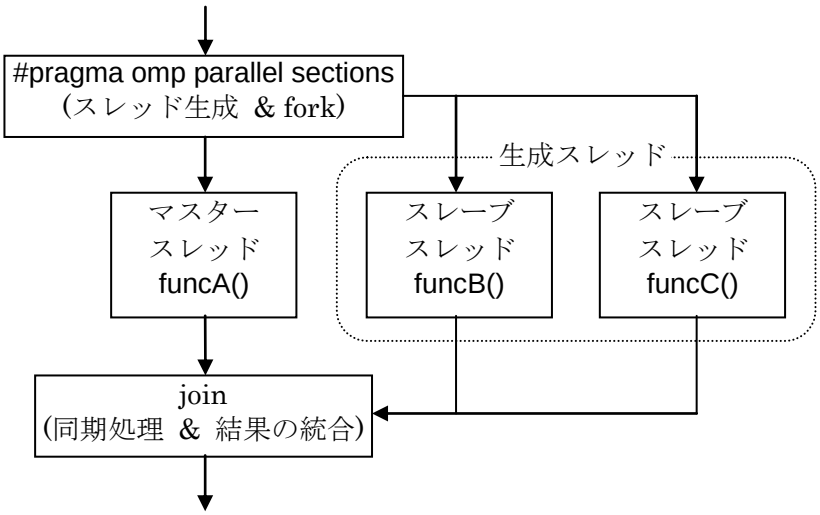


図 5：タスク並列の実行モデル

並列リージョンにおいて，各関数にスレッドが生成され各プロセッサで実行される．データ分割と同様に，各関数や処理の負荷が均等に分散されることが理想である．

2.2 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 6 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．

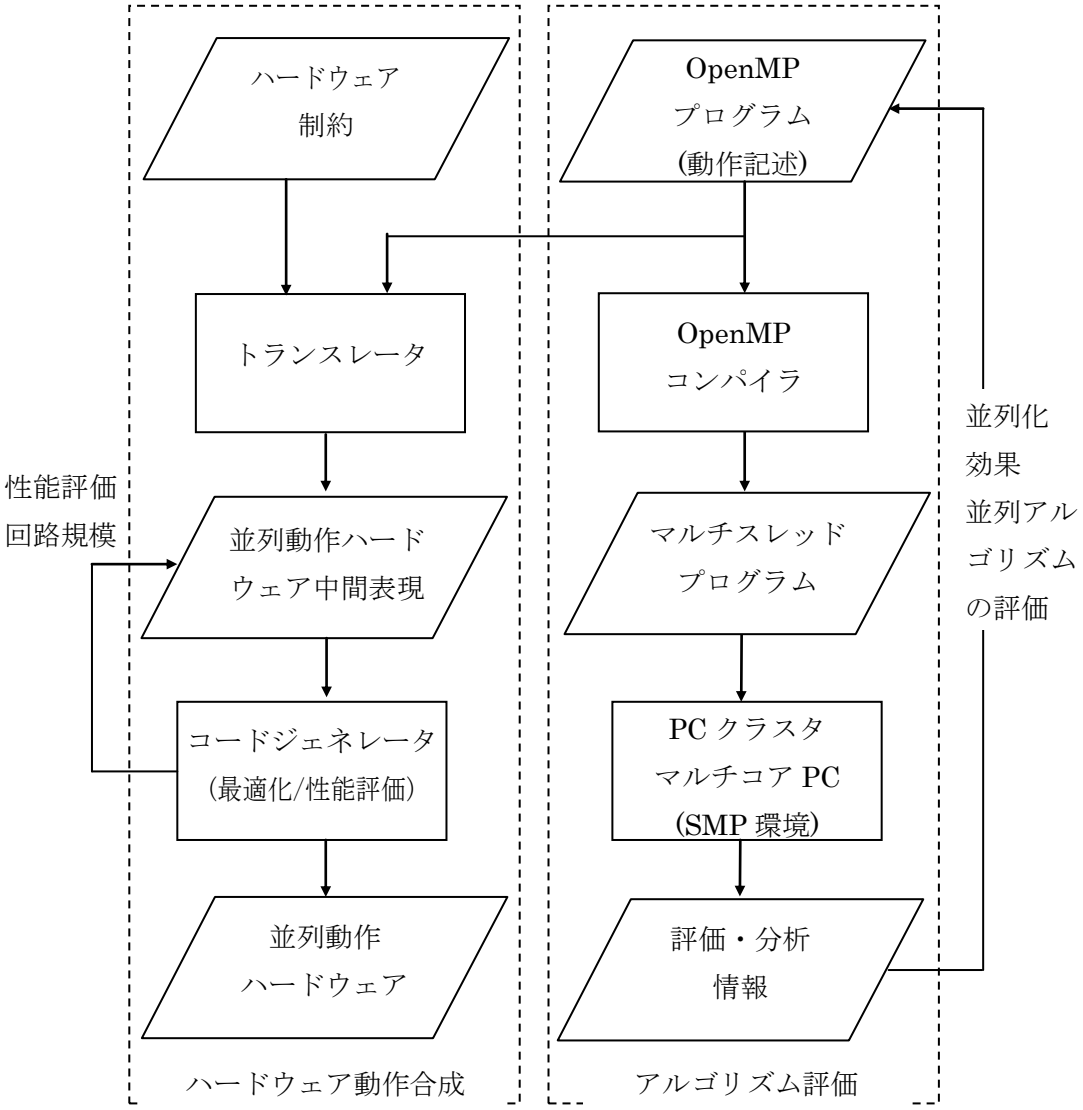


図 6：OpenMP を用いたハードウェア動作合成システム

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．本研究では，生成されるハードウェアのモデルを設定し，それに合わせた中間コードの解釈を行って最適化を行う．ハードウェアのモデルでは，演算器の共有化，内部レジスタ，及びメモリアクセスの方法を示している．生成されたハードウェアの性能評価や回路規模の推定を行い，それらの情報を中間表現にフィードバックを行って改善を行う．

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については，すでに実装されている[1]．本研究では，トランスレータから出力される中間表現に対し，コード生成を行うコードジェネレータを実装し，評価を行っている．

2.3 データ並列のハードウェア化

図 2 に示したデータ並列のプログラムを対象として，ハードウェアを動作合成によって生成するハードウェアモデルを図 7 示す．図 1，図 3 で示した fork-join モデルにおいて，逐次処理を行うマスタースレッドを逐次処理ハードウェアに，並列実行されるスレッドを並列ハードウェアとして合成する．合成されるハードウェアは各々FSM と DataPath を持っている．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．逐次処理の実行中は逐次処理ハードウェアが処理を行い，処理が OpenMP で指示された並列リージョンに到達すると，逐次処理ハードウェアが並列ハードウェアに対し処理の実行を命令する．これは OpenMP のモデルでは fork の動作に対応する．逐次処理ハードウェアは並列ハードウェアからの処理の実行を監視し，すべての処理の実行が終了した後，逐次処理の実行に復帰する．これは OpenMP のモデルでは join の動作に対応する．各共有のレジスタやメモリに関しては Arbiter を介してアクセスを行う．並列ハードウェアの各ノードがメモリに同時にアクセスした場合は，Aribter はノードに対してストール信号を発行し，ストール信号を受けたノードは処理の実行を一時停止する．

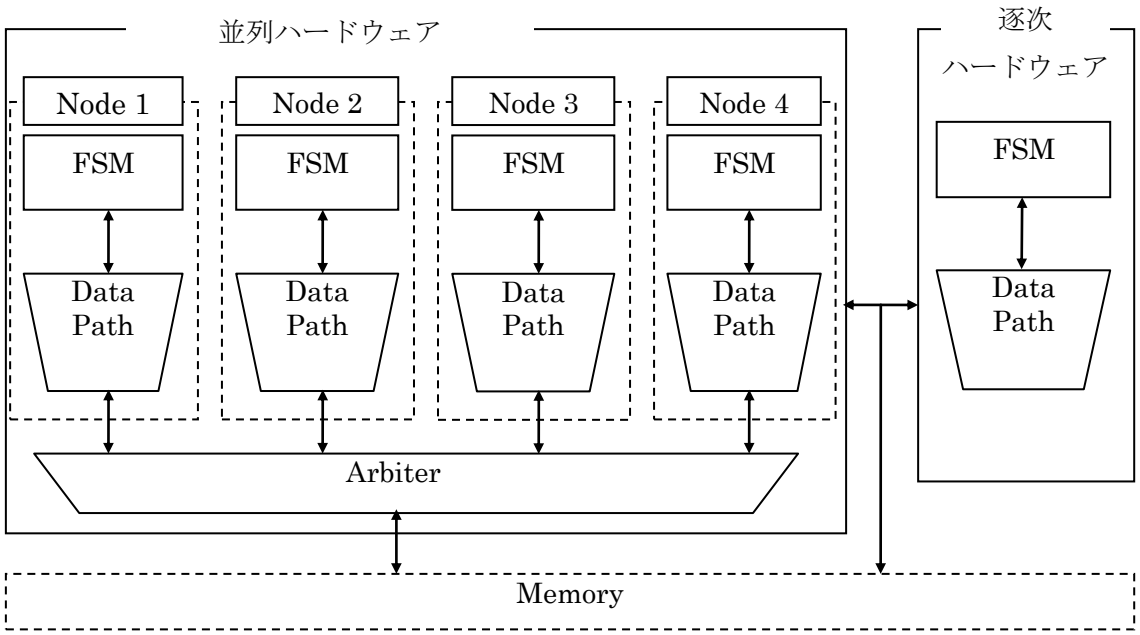


図 7：データ並列のハードウェアモデル

2.4 タスク並列のハードウェア化

図 4 に示したタスク並列のプログラムを対象として，ハードウェアを動作合成によって生成するハードウェアモデルを図 8 に示す．2.3 で示したデータ並列と同様に，逐次処理を行うマスタースレッドを逐次処理ハードウェアが実行し，並列リージョンにおいてタスク並列で実行される各処理(スレッド)を並列ハードウェアとして合成する．データ並列では，DataPath や FSM が共通する部分が多いが，タスク並列では，各々のタスクに合わせたものが生成される．共有するレジスタやメモリに対しては Arbiter を通してアクセスを行う．並列ハードウェアが処理を実行する場合は，内部の各ハードウェアは独立に並列実行を行う．そのため，同時に発行されるメモリアクセスが頻出すると，ストールの発生が増加し，ハードウェアの実行効率が下がってしまうため，この Arbiter の部分がボトルネックとなる可能性が高い．データ並列の場合も同様であり，アルゴリズムの並列化の検討を行う際には，各並列ノードの負荷均衡及び同時に発行されるメモリアクセスを減少させることが性能向上として重要である．

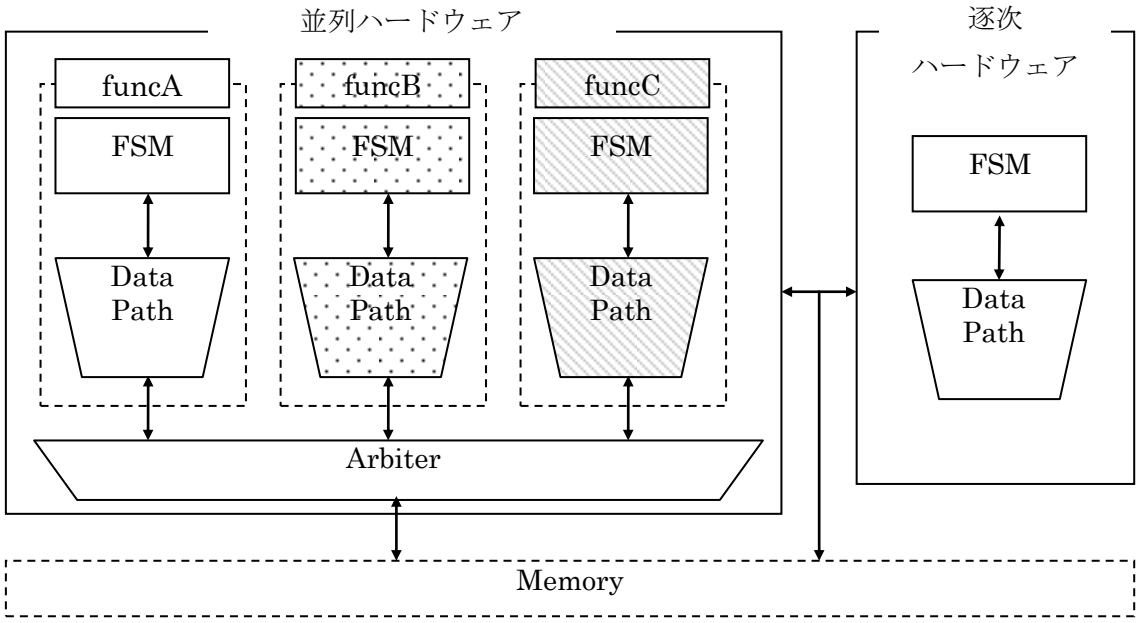


図 8：タスク並列のハードウェアモデル

3. コードジェネレータの実装

3.1 コードジェネレータの構成

ハードウェア動作合成系におけるコードジェネレータは，トランスレータによって生成された中間表現を用いてハードウェアの生成を行う．本研究で作成したコードジェネレータの構成を図 9 に示す．コードジェネレータは以下の５つのモジュールで構成されている．コードジェネレータは，はじめに前処理として並列情報解析を行う．並列情報解析では，トランスレータによって生成された中間コードと OpenMP によって指示された並列化情報に用いて，並列処理部の中間コードの複製や内部定数の変換を行う．図 9 では中間コードと並列化情報が別になっているが，実際には中間コードの CFG に含まれている．前処理によって得られた逐次処理及び変換・複製された並列処理部の中間コードを用いて，パラメータ生成，演算器生成，代入部生成，状態遷移生成の各モジュールが中間コードの解釈を行い，ハードウェアを生成する．パラメータ生成はレジスタや定数を生成する．演算器生成は用いられる各種演算器の生成とレジスタとの結線を行うコードを生成する．代入部生成では，演算された結果や，メモリなどの外部からの入出力をレジスタに代入するコードを生成する．状態遷移生成では，ハードウェアの FSM に当たる内部処理のコントロールを行うコードを出力する．HDL を直接生成する後者４つのモジュールについては次節以降で詳細を説明する．

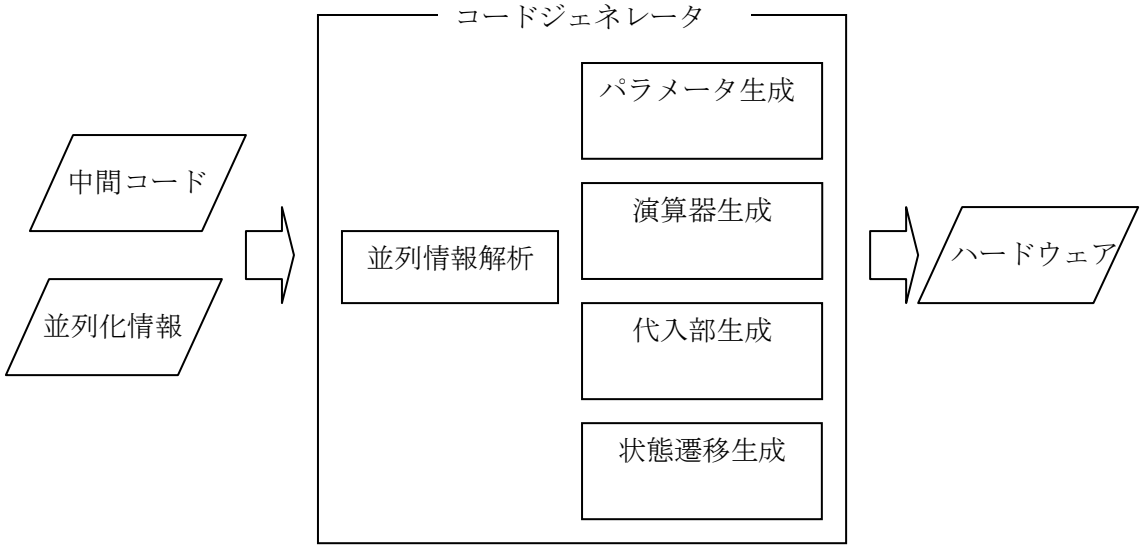


図 9：コードジェネレータの構成

3.2 中間表現からのハードウェア生成方法

3.2.1 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 10 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．

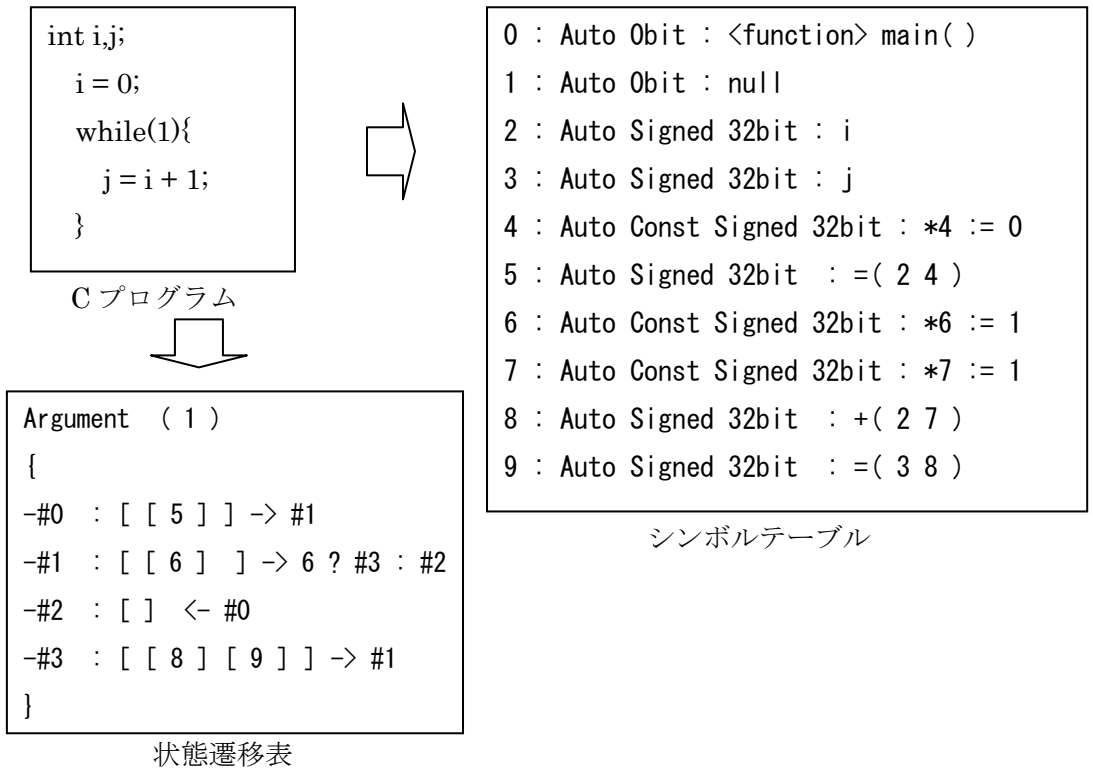


図 10：中間表現のサンプル

OpenMP の並列化指示文が挿入されている場合は状態遷移表において指示情報が明記される．コードジェネレータの前処理に当たる並列化情報解析によって，CFG の複製とパラメータの変更が行われる．

OpenMP の動作記述の変換に対して，システムの実装における制限を以下に示す．

(1) 並列化指示文は結合されたワークシェアリング(for, sections)分のみサポートする．

(2) マスター構文と同期構文をサポートしない．

(3) データ構文ではリダクション演算をサポートしない．

(4) ランタイムライブラリはサポートしない．

これらはハードウェアでの制御が困難であることから制限される． ベースとなる言語はC言語であり，以下のような制限を持つ．

(1) 配列はサポートするが，ポインタはサポートしない．

(2) ラベル文及び goto 文はサポートしない．

(3) 浮動小数点を用いることはできない．

(4) 構造体・共用対をサポートしない．

3.2.2 ハードウェアモジュールの生成方法

動作合成を行う際に想定するハードウェアモジュールのモデルを図 11 に示す．動作合成される回路全体の概要ではなく，図 7，図 8 で示した逐次ハードウェアと並列ハードウェアの各モジュールのモデルに当たる．

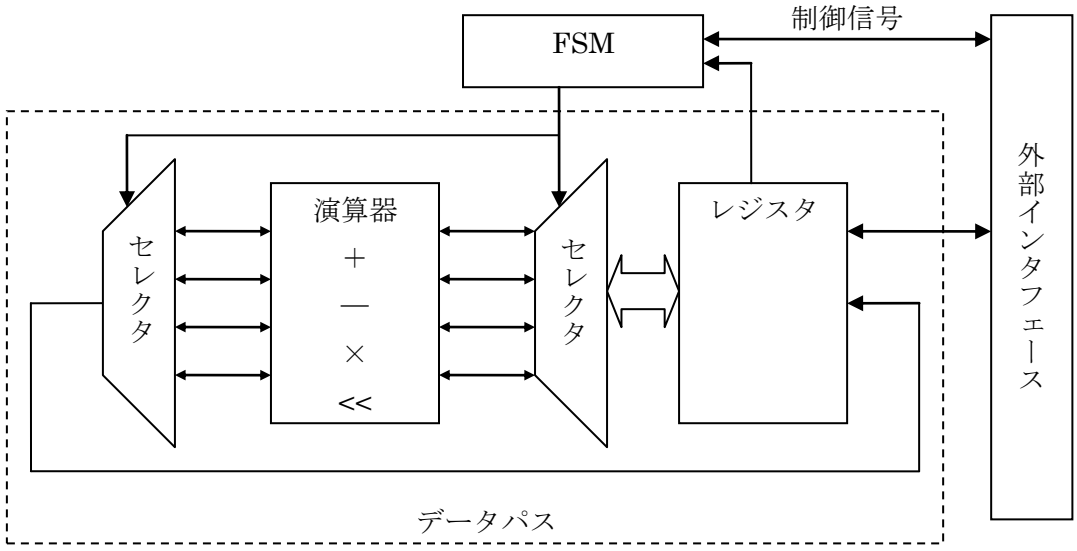


図 11：ハードウェアモジュールのモデル

ハードウェアや並列ハードウェアの各ノードは FSM とデータパスを持つ．動作合成では，プロセッサでの処理実行と違い，状態遷移表で表現される演算に対し，演算の数だけ演算器を生成することも可能である．しかし，多くの演算器を生成することは，実行中に動作しない演算器を増やすことになり冗長な回路となる．また，同時実行できる演算器の性能と回路の冗長性のバランスを取ることは一般的に難しく，これはデータと処理のバインデ

ィングの問題になる．そこで，本研究では一サイクル一演算と設定し，各モジュール単位で最低限の演算器を持つ非常にシンプルなモデルとする．想定するデータパスでは＋，－，×などの演算器に対して，変数を格納するレジスタを大きなセレクタを介して接続する．演算器からの出力は，セレクタを介して必要なレジスタへ書き戻す．FSM は状態遷移表及びシンボルテーブルから生成し，内部状態のコントロールを行う．状態遷移に必要な変数についてはレジスタの値を参照する．また，セレクタへの制御信号，及び Arbiter や外部モジュールへの制御信号をコントロールする．外部からの処理のスタートや処理の終了，Arbiter からのストールなどは外部インタフェースから入出力する．

動作合成において，コードジェネレータが行う CFG の解釈を以下に示す．

- 1 クロックにつき 1 演算で処理を行う．
- +,-,*,<<,等の演算子は，モジュール内で必要な一つの演算器を共有する．
- 配列で宣言された変数はメモリ上に確保し，アービタを通してアクセスする．
- 配列アクセスでは C 言語の仕様に従いアドレス演算を行う．
- 配列の初期化宣言は解釈しない(必要な場合は初期化ではなく処理の最初に記述する)．
- レジスタは変数，及び各演算について一つ宣言する

配列アクセスでは Arbiter を通して外部のメモリにアクセスするため，C 言語の実行と同様にアドレスのオフセット計算を行っている．C 言語では多次元配列を 1 次元配列に対する添え字とのオフセットで実現している．当然ながら物理的にもメモリは 1 次元である．メモリのオフセット計算の例を表 1 に示す．

表 1：アドレスのオフセット計算

宣言	アクセス	メモリアドレス
int a[5]	a[3]	3+a
int b[5][10]	b[3][7]	(3×10)+7+b
int c[5][10][20]	c[3][7][11]	(3×10×20)+(7×20)+11+c

表 1 の波線で示した部分については，アドレス計算で頻出し，かつ先に計算できる固定値なので，コード生成では先に計算しパラメータで宣言する．以上を演算だけまとめると

演算：1 次配列 +

2 次配列 x + +

3 時配列 x + x + +

となる．配列の次元が一つ増えるたびに「x，+」が一つ増える．この演算も演算器を共有し，演算器生成部でコード生成を行っている．

コードジェネレータが生成するハードウェアは verilog のソースコードとする．実装は ruby 言語により実装している．実装・実行環境は windows XP，i386-cygwin，ruby のバージョンは 1.8.6 である．

3.3 パラメータ生成

コードジェネレータにおけるパラメータ生成では，シンボルテーブルを用いて演算に必要な定数，レジスタ，メモリアドレスの先頭アドレスを設定する．パラメータ生成の例を図 12 を示す. パラメータ生成で宣言されるレジスタやパラメータはその他モジュールのすべてで参照もしくは代入が行われる．①では内部レジスタとして内部演算で用いる変数，演算結果を一時格納するレジスタ，状態遷移を格納するレジスタを生成する．②ではパラメータとして内部演算で用いる定数を宣言する．シンボルテーブルの解釈においてレジスタを用意して初期化で解決することも出来る冗長であるため，verilog の文法に従い prameter で宣言する．また，③の内部状態を表す状態遷移番号も parameter で宣言する．parameter で設定される値は必ずしもシンボルテーブルの番号と一致する必要はなく，タグ番号である．配列アクセスの状態遷移については配列アドレスのオフセット計算に必要な状態とメモリからのストール用の状態を宣言する．例で示している”ARRAY(18 7)”は 1 次元配列であるため, 3.2.2 で示したオフセット計算のための状態遷移を 3 つ生成している．P_STATE46_AR_STALL はアービタからのストールを受けつけるための退避用の状態遷移先である．一時格納用のレジスタについては，モジュールごとに汎用のレジスタとして必要な数だけ宣言することも可能であるが，状態遷移において，参照される場合に解析が難解であるため一演算に付き一つを宣言している．

シンボルテーブル		レジスタ，パラメータ
①レジスタ（内部変数と一次格納用レジスタ）		
2 : Auto Signed 32bit : i	→	reg [31:0] i;
3 : Auto Signed 32bit : j	→	reg [31:0] j;
8 : Auto Signed 32bit : +(2 7)	→	[31:0]REG27;
②パラメータ（定数）		
4 : Auto Const Signed 32bit : *4 := 0	→	parameter [31:0]ConstNum4 = 32’d0;
6 : Auto Const Signed 32bit : *6 := 1	→	parameter [31:0]ConstNum6 = 32’d1;
③パラメータ（状態遷移）		
モジュール共通	→	parameter P_INIT = 8’d0;
モジュール共通	→	parameter P_END = 8’d1;
8 : Auto Signed 32bit : +(2 7)	→	parameter P_STATE8 = 8’d2;
46 : Auto Signed 32bit : ARRAY(18 7)	→	parameter P_STATE46_AR_R0 = 8’d10; parameter P_STATE46_AR_R1 = 8d’11; parameter P_STATE46_AR_STALL = 8d’12;

図 12：パラメータ生成の例

3.4 演算器生成

演算器生成では，各モジュール内で必要な演算器の生成と図 11 に示したレジスタの出力から演算器へのセレクタの生成を行う．演算器生成の例を図 13 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．例で用いた シンボルテーブルでは，理解しやすいようにシンボルの後ろに＊で対応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．この部分が図 11 の右側のセレクタに対応する．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．ただし，除算については回路シミュレーションでは可能であるが，論理合成では合成を行うことはできない．そのため，OpenMP の記述の段階で，除算を bit シフトと加算と減算で表現する，もしくは固定小数点の掛け算に変換するなどを行う必要がある．現在の状態を表すレジスタは FSM にあたる状態遷移生成によって制御されている．配列アドレスのオフセット計算も加算と乗算の演算器を共有する．

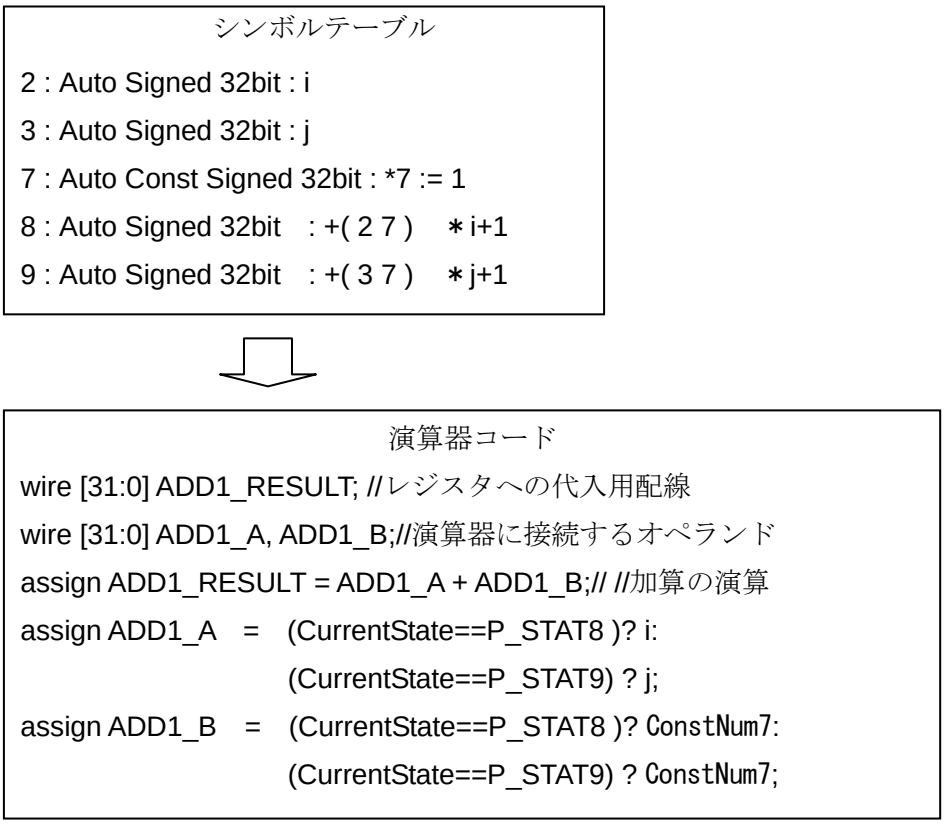


図 13：演算器生成の例

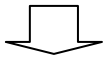
3.5 代入部生成

代入部では，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．図 14 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．3.4 で示した例と同様に＊で対応する演算を示している．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．この動作が，図 11 の左側のセレクタに当たる．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．

例では P_STATE8 において，REG8 <= ADD1_RESULT が行われている．ADD1_RESULT は演算器コードにおける足し算の演算器の出力にあたり，これを一時格納用の変数に代入する．その後，次のクロックで j のレジスタに代入している．j のレジスタへ直接代入することもできるが，構文解析による先読みが必要なのと，途中の計算結果を他の演算で再利用することも考えられるため，このようにしている．

シンボルテーブル

```
2 : Auto Signed 32bit : i
3 : Auto Signed 32bit : j
7 : Auto Const Signed 32bit : *7 := 1
8 : Auto Signed 32bit   : +( 2 7 )    ＊i + 1
9 : Auto Signed 32bit   : =( 3 8 )    ＊j = i + 1
  …
```



代入部コード

```
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;    ＊i + 1
       P_STATE9   :   j       <= REG8;                ＊j = i + 1
       …
    endcase
 end
end
```

図 14：代入部生成の例

3.6 状態遷移生成

状態遷移生成は，図 11 における FSM のコード出力に相当する．状態遷移コードの例を図 15 に示す. 例は 3.2.1 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．

状態遷移表

```
Argument  ( 1 )
{
-#0  : [ [ 5 ] ] -> #1
-#1  : [ [ 6 ]  ] -> 6 ? #3 : #2
-#2  : [ ]  <- #0
-#3  : [ [ 8 ] [ 9 ] ] -> #1
}
```

シンボルテーブル

```
0 : Auto Obit : <function> main( )
1 : Auto Obit : null
2 : Auto Signed 32bit : i
3 : Auto Signed 32bit : j
4 : Auto Const Signed 32bit : *4 := 0
5 : Auto Signed 32bit  : =( 2 4 )
6 : Auto Const Signed 32bit : *6 := 1
7 : Auto Const Signed 32bit : *7 := 1
8 : Auto Signed 32bit  : +( 2 7 )
9 : Auto Signed 32bit  : =( 3 8 )
```

状態遷移コード

```
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 15：状態遷移生成の例

4. エッジ検出に対するハードウェア動作合成

本章では，OpenMP で書かれたエッジ検出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．

4.1 エッジ検出のアルゴリズム

エッジ検出とは画像から対象物の外形を表す輪郭を抽出する処理である．人間の色覚では，画像の明るさや色が急激に変化する部分を感知し，対象物を知覚すると考えられている．画像処理では，こういった明るさや色が急激に変化する部分を微分によって強調することにより検出を行う[17]．微分には 1 次微分(グラディエント)と 2 次微分(ラプラシアン)があり，本章では，1 次微分において最もよく用いられる sobel フィルタで実験を行っている．x 方向の一次の偏微分と y 方向の一次の偏微分の定義を式(1)，(2)に，変化の強さを式(3)に示す．偏微分の重み付けに用いる sobel のオペレータを図 16 に示す．

x 方向の偏微分：

$$\Delta_x f(i,j) = \frac{f(i+1,j) - f(i-1,j)}{2} \qquad \cdots\cdots\cdots\cdots\cdots (1)$$

y 方向の偏微分

$$\Delta_y f(i,j) = \frac{f(i,j+1) - f(i,j-1)}{2} \qquad \cdots\cdots\cdots\cdots\cdots (2)$$

強さ：

$$\sqrt{\Delta x^2 + \Delta y^2} \qquad \cdots\cdots\cdots\cdots\cdots (3)$$

−1	0	1
−2	0	2
−1	0	1

$\Delta_x f$

−1	−2	−1
0	0	0
1	2	1

$\Delta_y f$

図 16：sobel オペレータ

対象の画素に対し，近傍画素の x 方向，y 方向に sobel オペレータに従って演算を行う．sobel フィルタでは x 方向，y 方向について，それぞれ 4 回の加減算と 2 回の乗算，強さの計算に 1 回の加算と 2 回の乗算が行われる．ルートの演算についてはハードウェアでは実装が難しいため，プログラムではビットシフトによる近似演算を用いて実装している．

4.2 並列ハードウェア構成

実装したエッジ検出プログラムのフローチャートを図 17 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，共に輝度が 0~255 の濃淡画像を想定したため，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. 並列化は図 17 のラスタスキャンに対してデータ並列を用いて行った. 対象とする画像空間を 4 分割し，各空間に対する処理を並列に実行する. エッジ検出の 4 並列のハードウェア構成を図 18 に示す.

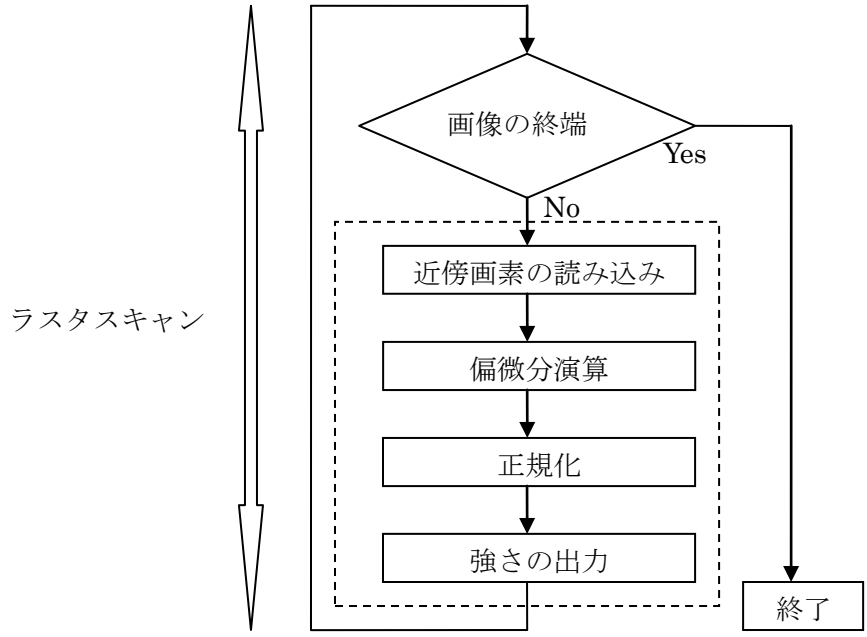


図 17：エッジ検出のフローチャート

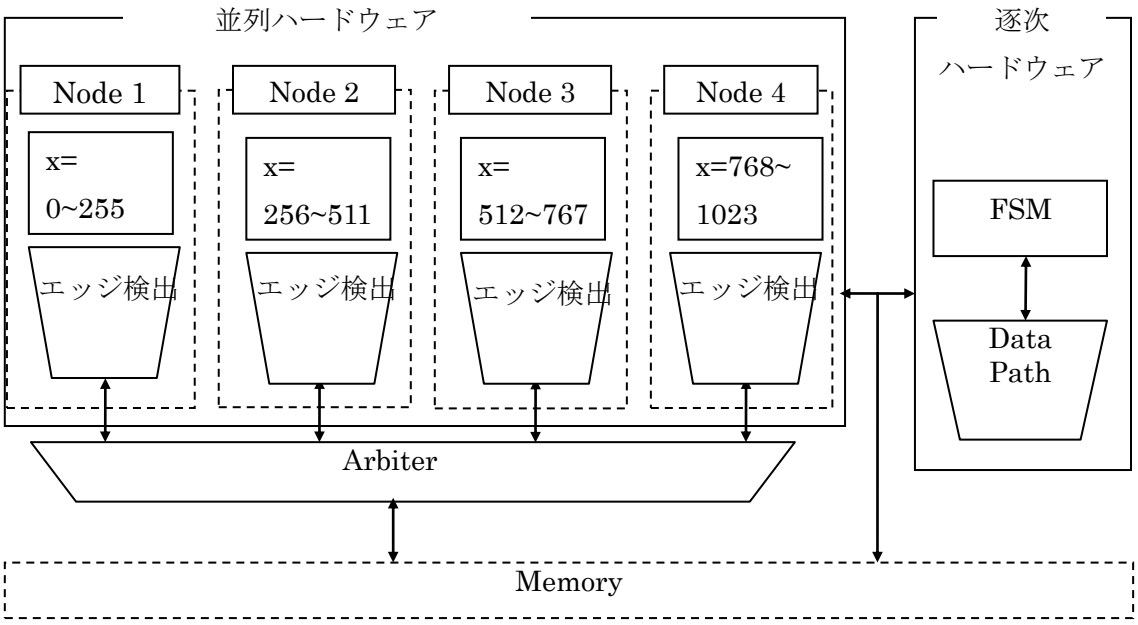


図 18：エッジ検出の並列ハードウェア構成

図 18 は画像サイズが 1024×1024 の画像に対し，x 方向に対してデータ並列を行った場合を示している．各並列ノードは逐次ハードウェアからのスタート信号によって動作を開始する. 各ノードは並列に動作し, 各々が FSM を持って自身の制御を単独で行う. ただし, 入力画像や出力画像が置かれた共有のメモリアクセスについては，各ノードが同時にアクセスした場合は，Arbiter のストール信号を受け付けて自身の動作を一時的にストップさせる．逐次ハードウェアは並列ハードウェアの実行を監視し，すべてのノードの実行が終了しだい，自身の動作に復帰する.

4.3 実験と評価

(1)実験環境

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 2 に示す.

表 2：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 8.2i
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

(2)実験結果と評価

エッジ検出に用いた画像は，解像度 1024×1024，輝度 0~255 の pgm 画像である.
エッジ検出の OpenMP プログラムを SMP クラスタで実行した場合の時間と速度向上比を表 3 に示す.

表 3：エッジ検出の SMP クラスタでの実行速度

スレッド数	1	2	4
実行時間(ms)	39.39	21.406	13.974
速度向上比	1.00	1.84	2.82

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 の場合にではほぼ理想的な速度向上であるが，4 スレッドでは 3 倍程度になっている．スレッド数が増えるに従い，スレッ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必要な動作クロック数と論理合成によって得られた動作周波数を表 4 に示す．クロック減少比は，スレッドが一つの逐次処理の場合に対する動作クロックの比率を示している.

表 4：エッジ検出の生成ハードウェアの実行速度

	スレッド数	1	2	4
回路シミュレータ	動作クロック数(Kcycle)	159,744	79,411	39,776
	クロック減少比	1.00	2.00	3.99
並列ハードウェア	動作周波数(MHz)	87.588	87.588	87.588
	速度向上比	1.00	2.00	3.99

動作クロック数については，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．動作周波数については，想定するハードウェアのデータパスがほぼ同じであるため，同一であった．結果として，スレッド数に対しほぼ理想的な速度向上を得ることができた．動作クロックの減少が並列数に対して理想的な理由として，各ノードのメモリアクセス頻度

に対し，演算に必要なクロック数の方が多かったため，ストールの発生頻度が少なかったことが上げられる．また，エッジ検出はほとんど分岐のない規則的な処理であるため，各並列ノードにおいて，メモリアクセスが生じるタイミングがストールによって一度ずれれば，次のストールが発生しにくかったことが考えられる．

論理合成による回路面積と回路シミュレーションでの実行時間を表 5 に示す．回路面積比がスレッド数の比率以上に増加するのは，並列化のオーバーヘッドにあたる arbiter やレジスタ，配線などの回路の増加によるものと思われる．比較のため，Arbiter 回路を外したエッジ検出の論理合成結果を表 6 に，Arbiter 回路の論理合成結果を表 7 に示す．

表 6 で回路面積比がスレッド数の増加より下回っているのは論理合成ツールの最適化によるものと思われる．回路として単体では動作しないため厳密には比較できないが，表 5 で回路の増加が大きいのは，Arbiter 回路や制御回路を結線することで，論理合成ツールの最適化が困難になっているためと思われる．

表 5：エッジ検出の回路規模と回路シミュレーション時間

スレッド数	1	2	4
回路規模(Slices)	2295	6484	12745
回路面積比	1.00	2.83	5.55
シミュレーション時間(s)	3916	3912	3879

表 6：Arbiter 回路を外したエッジ検出の回路規模

スレッド数	1	2	4
回路規模(Slices)	2295	4495	9084
回路面積比	1.00	1.96	3.96
動作周波数(MHz)	87.588	87.588	87.588

表 7：Arbiter 回路

回路規模(slices)	495
動作クロック(Mhz)	91.600

表 3 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，10 万~30 万倍程度であった．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．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動作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．

5. ハフ変換に対するハードウェア動作合成

本章では，４章で示したエッジ検出と同様に，ハフ変換に対する本システムの評価を示す.

5.1 ハフ変換のアルゴリズム

ハフ変換とは，画像の中から，直線，円，任意図形などの抽出を行う際に用いる手法の一つである．実装を行ったのは直線を検出するプログラムである．直線を表す代数方程式を(4)に示す．原画像を走査し対象画素 p(x,y)を検出したときに，その座標(x,y)を式(4)の x,y に代入する.

$$\rho = x\cos\theta + y\sin\theta \qquad \cdots\cdots\cdots\cdots\cdots (4)$$

ここで，ρは座標原点から直線へ下ろした垂線の長さ，θは垂線と x 軸との間の角度を示すパラメータである．点Ｐ(x,y)とθ，ρの幾何学的関係を図 19 に示す．図 19 からわかるように，式(4)のθとρは xy 座標系の原点と点 P を結ぶ直線が点 P を通る直線の垂線になるときの垂線の角度と，原点から点 P までの距離をそれぞれ表している．すなわち，点Ｐを通る直線(の法線)を原点からの角度(θ)と距離(ρ)によって表現していることになる.

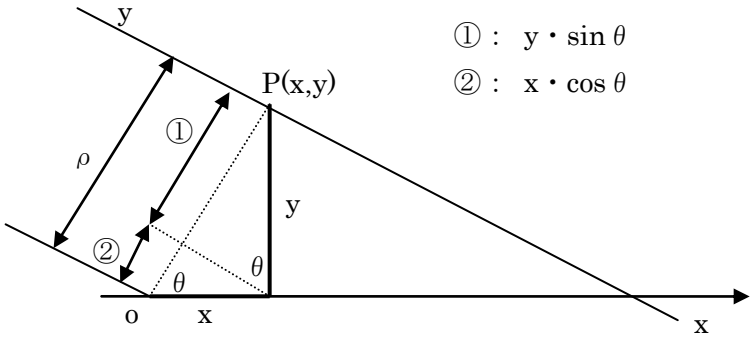


図 19：点(x,y)とθ ρの幾何学的関係

θ ρ平面への写像の例を図 20 に示す．原画像の特徴点に対し，点線で示されている直線Ｌが推測される直線である．θ ρ平面へ写像を行うことで，各々の点に対して一つの曲線を得ることができる．そうして得られた曲線の交点が，原画像上での直線Ｌを表現することになる．コンピュータ上での計算の手順を以下に示す.

1．θ-ρパラメータ空間を表す２次元配列を用意し，その値をすべて 0 に初期化する.
2．画像中の各特徴点の座標値を式(4)の x,y に代入した式を，θとρに冠する方程式とみなし，θ ρ平面で方程式の表す軌跡を描く．軌跡の描画は，θをΔθずつ増加させながら方程式を満たすρの値を計算することによって，軌跡の通過する配列要素を求め，その値を１増やす(投票する)ことによって軌跡を重ね書きする.
3．すべての特徴点に対応する軌跡を描いた後，多数の軌跡が集中している位置，すなわちθ ρ空間上で最も投票された要素を求める．[17]

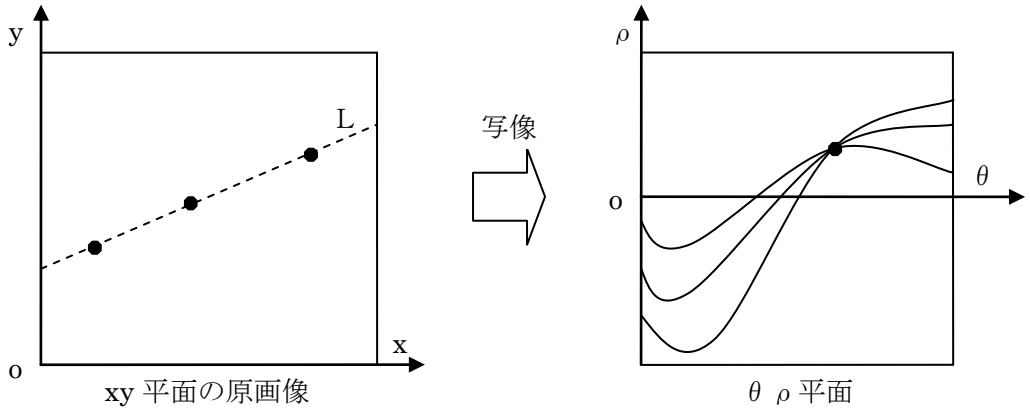


図 20：θ ρ 平面への写像の例

コンピュータ上では連続量である θ の変化をすべて計算することはできない．そのため，Δ θ を設定し，θ ρ を有理数上の離散値として表現する．Δ θ を小さくし，ρ 軸の刻みを小さくすることで計算精度は向上するが，計算負荷が増加する．また，計算されるべき特徴点が多ければ多いほど，写像の計算が増える．

ソフトウェアでは，マクローリン展開を用いて cos 関数と sin 関数の計算を行うが，ハードウェアではマクローリン展開の実装が難しいため，プログラムではテーブル演算によって実装している．

5.2 並列ハードウェア構成

実装したハフ変換プログラムのフローチャートを図 21 に示す．対象の画像に対し，ラスタスキャンを行い，特徴点を探索する．特徴点の座標に対し，θ の値を変化させながら出力用のθ ρの画像空間に対し投票を行う．並列化については，図 21 の①ラスタスキャンの部分と，②ハフ変換内部のθに対しての 2 通りが考えられるが，①の方がループの処理単位が大きいことから，①に対してデータ並列を行った．①のラスタスキャンに対するデータ並列化は，エッジ検出と同様に画像空間を 4 分割し，各空間に対して並列に実行する．ハフ変換の 4 並列のハードウェア構成を図 22 に示す．図 22 は 256×256 の画像対し，x 方向に対してデータ並列を行った場合を示している．図 18 と同様に各ノードで，対象の画像空間の探索を均等に分担する．対象の画像と出力するθ ρ空間の画像は，共有メモリ上に置かれる．各ノードは画像情報を読み取り，特徴点の探索を行う．特徴点が見つかった場合，θ ρ空間の画像に対して投票を行う．

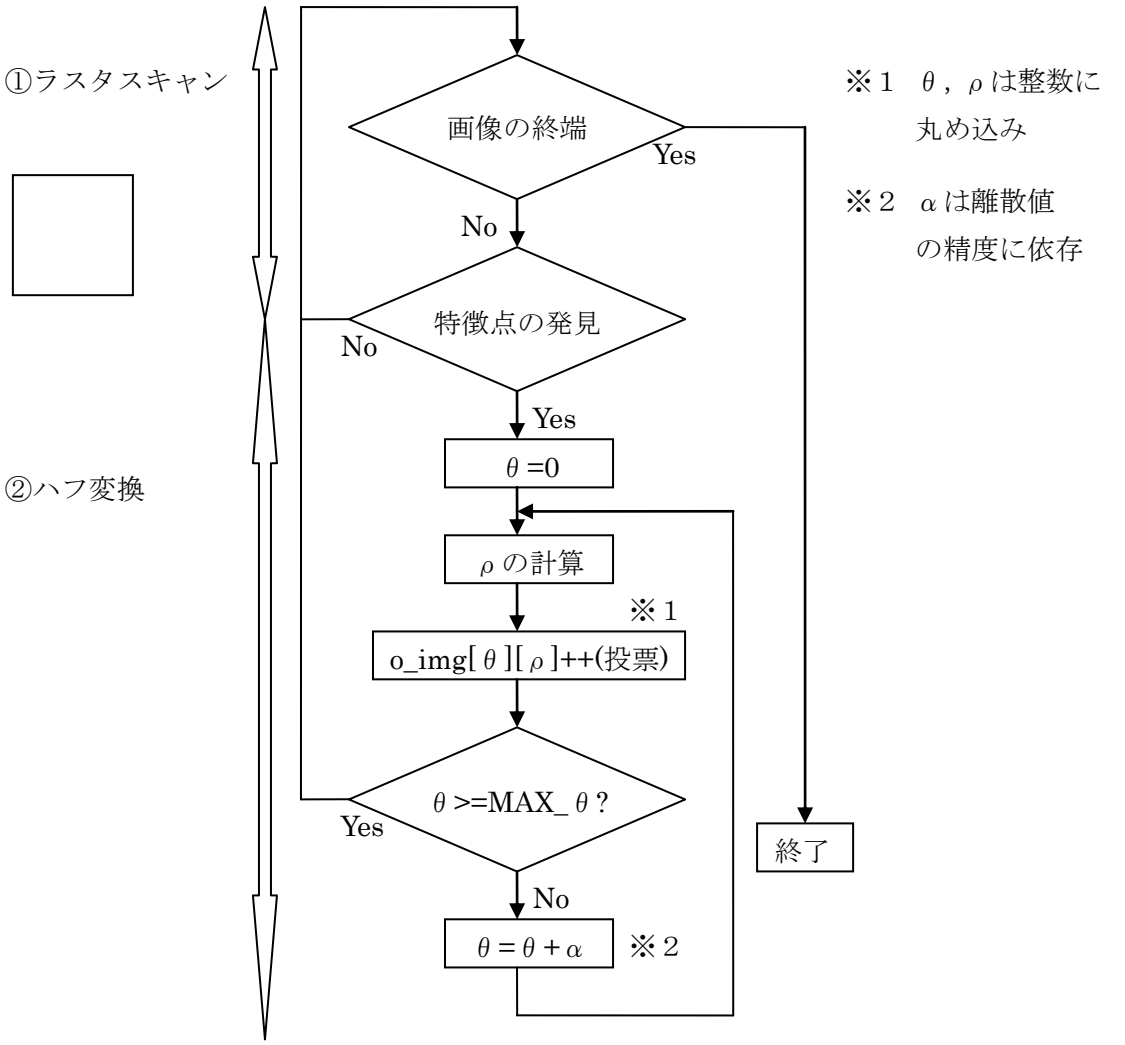


図 21：ハフ変換のフローチャート

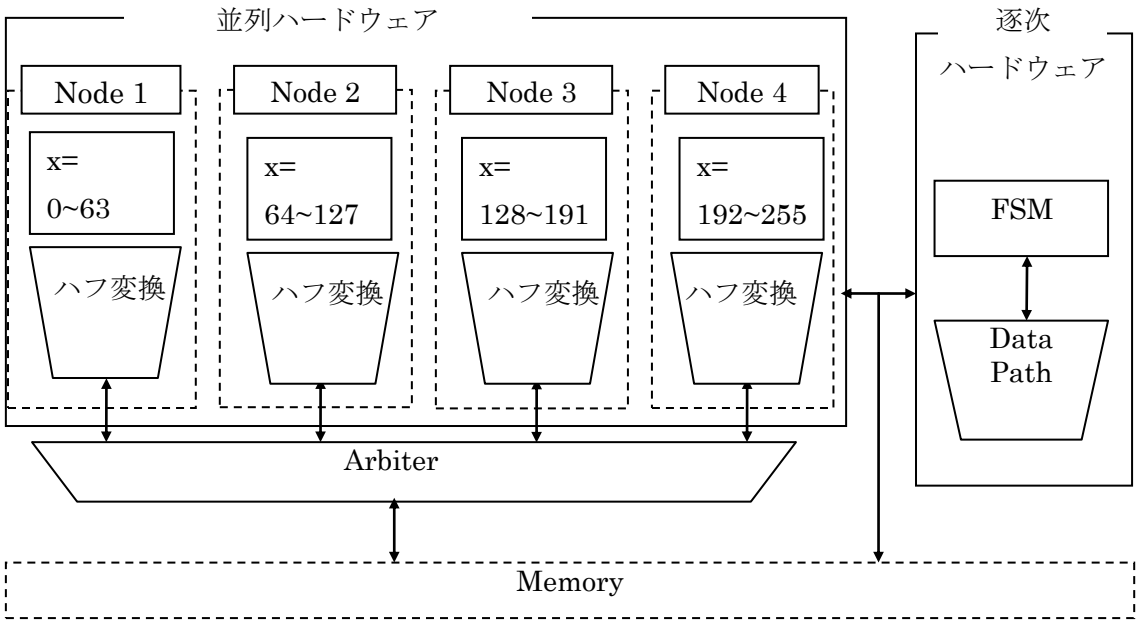


図 22：ハフ変換の並列ハードウェア構成

5.3 実験と評価

実験環境は 4.3 節で示したものと同じである. ハフ変換に用いた画像は, 解像度 256×256, 輝度が 0,255 の 2 値画像, θ ρ 空間は,720, 1440, 輝度が 0~255 である. 輝度については, 投票によって増えた数値の値をそのまま利用している. 数値が 255 を超える場合は 255 にしている. ハフ変換の OpenMP プログラムを SMP クラスタで行った実験結果を表 8 に示す.

表 8 : ハフ変換の SMP クラスタでの実行速度

スレッド数	1	2	4
実行時間(s)	38.05	28.264	25.774
速度向上比	1.00	1.35	1.48

表 3 で示したエッジ検出での実行結果と比べて, スレッド数に対する速度向上比の伸びが小さい. これは, エッジ検出のようにすべての画素に対して同じ処理を行うのではなく, 特徴点が検出された画素に対してのみ処理を行うため, 対象画像の特徴点のばらつきによって, 処理が均一にならないためと思われる.

回路シミュレーションを用いた測定した実行に必要な動作クロック数と論理合成によって得られた動作周波数を表 9 に示す.

表 9 : ハフ変換の生成ハードウェアの実行速度

	スレッド数	1	2	4
回路シミュレータ	動作クロック数(Kcycle)	51,273	29,466	26,210
	クロック減少比	1.00	1.74	1.96
並列ハードウェア	動作周波数(MHz)	88.623	88.623	88.623
	速度向上比	1.00	1.74	1.96

動作周波数についてはハフ変換とほとんど同じであり, またスレッド数の増加に依存しない. これは想定しているハードウェアモデルが同じためである. クロック減少比は 4 スレッドで 1.96 倍である. エッジ検出に比べ並列化効果が低くなっているのは, SMP クラスタでの実行と同様に, ハフ変換がデータ依存のアルゴリズムであるからだと考えられる. 4 並列で実行した場合の各ノードのストール回数を 500 クロックごとにまとめたものを図 23 に, 各ノードの動作クロック数とストール回数を表 10 に示す. メモリアクセスにおけるアービタのアクセス優先順位はノード 1 から順にノード 4 である. それにより, ノード 1 はストールが発生しないため, 図 23 からは除外している. 最も優先順位の低いノード 4 が最初にストールを多く発生させているが最初に処理が終了しているため, ノード 4 の負荷が非常に少なかったことがわかる. 最もストールが発生しているのはノード 3 であり, これは優先度の高いノード 2 に最も処理負荷がかかり, メモリアクセスを頻出しているためである. 並列ハードウェアの動作クロック数はノード 2 とほとんど同じであるため, ノード 2 がボトルネックとなっている. ストール回数は全体のクロックの 0.5 パーセント程度

であり，ストールの回数以上に負荷分散の偏りが速度向上に大きく影響を及ぼしていることがわかる．

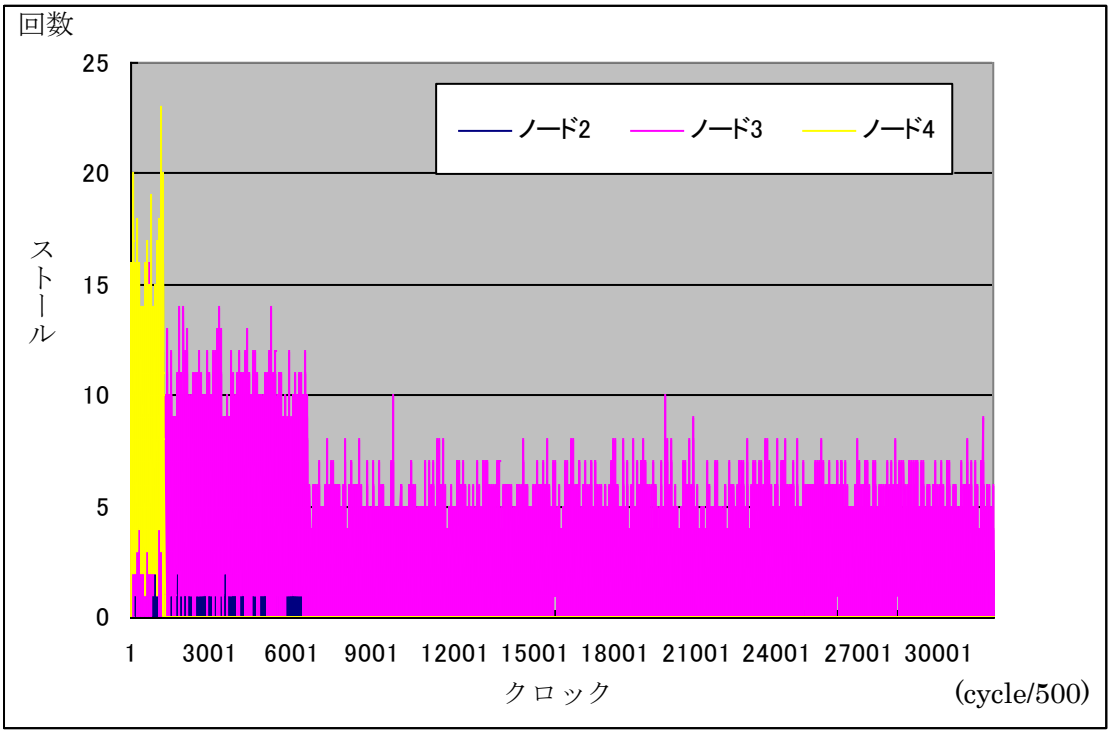


図 23：4 並列におけるストール回数

表 10：各ノードの動作クロックとストール回数

	ノード 1	ノード2	ノード3	ノード4
動作クロック数(Kcycle)	32,68	26,210	21,315	612
ストール回数	0	12,710	109,349	11,182

論理合成による回路面積と回路シミュレーションでの実行時間を表 11 に示す．回路面積についてはエッジ検出と同様にスレッド数以上の増加になっている．逐次処理での面積が比較的小さいため，増加についてもエッジ検出よりは低くなっている．比較のため，Arbiter 回路を外した回路面積を表 12 に示す．表 12 に示すように，スレッド数の増加に対して当然ながら回路面積が同倍に増加している．動作周波数が変わらないのは各スレッドに対して想定するハードウェアモデルが同じであるためと思われる．

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 2~7 万倍程度である．エッジ検出と同様に，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．スレッド数の増加に対して，少しずつシミュレーション時間が増えている．並列化効果による動作クロック数の減少が少なかったため，クロック数の減少に対して，スレッド数の増加による回路シミュレーションの PC への負担が大きくなったためと思われる．

表 11：ハフ変換の回路面積と回路シミュレーション時間

スレッド数	1	2	4
回路規模(Slices)	1627	4207	8232
回路面積比	1.00	2.59	5.06
シミュレーション時間(s)	977	1015	1021

表 12：Arbiter 回路を外したハフ変換の回路面積

スレッド数	1	2	4
回路規模(Slices)	1627	3253	6505
回路面積比	1.00	2.00	4.00
動作周波数(MHz)	88.623	88.623	88.623

動作合成システムのアルゴリズム評価系において，エッジ検出とハフ変換の実験結果から，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思われる．エッジ検出とハフ変換の両方において，速度向上比は SMP 環境よりも動作合成によって生成されたハードウェアの方が高い．これは，本研究で想定したハードウェアモデルでは，動作合成されるハードウェアの動作クロック数全体に占めるメモリアクセスが少ないため，アービタを通したメモリアクセスを効率的に処理できたからである．しかし，一般的な手作業によるハードウェア設計を行った場合の動作クロック数に比べて，動作合成で生成されたハードウェアの動作クロック数は大きく増加していると思われる．本システムのコードジェネレータの最適化において，必要な動作クロック数とメモリアクセスのバランスを計ることが重要である．

6. おわりに

本研究では，OpenMP を用いたハードウェア動作合成に対して，コードジェネレータの実装と画像処理による評価を行った．トランスレータによって生成された中間表現に対し，ハードウェアモデルを想定し，中間表現の解釈と生成される verilog コードを示した．また，エッジ検出とハフ変換の 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．回路規模において，エッジ検出，ハフ変換共に並列度に比例する回路規模となった．速度向上においては，エッジ検出では並列度に対し理想的な速度向上が，ハフ変換においてはソフトウェアシミュレーションとほぼ同等な速度向上が得られた．

今後の課題として，タスク並列によるハードウェアのパイプライン処理の実装が考えられる．パイプラインはハードウェアにおいて非常に重要な概念であり，OpenMP の構文を利用してパイプラインのシミュレーションや動作合成により，評価・分析を行うことができればシステムの機能性や汎用性が拡張される．また，コードジェネレータにおいて，本研究ではシンプルなハードウェアモデルを想定しているが，演算器やレジスタのバインディング，必要な動作クロック数とメモリアクセスのバランス，及び動作合成後の回路規模や実行速度に合わせた最適化が必要である．

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，小柳滋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に深く感謝いたします．

最後に，共同研究者である安部厚志氏，大岩広司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．

(clock/500)

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] 上平祥嗣, “並列アルゴリズムクラスに基づくハードウェア自動生成”, 立命館大学院理工学研究科修士論文, 2000

[3] 松井誠二,“並列プログラムからのハードウェア自動生成システムの検討”, 立命館大学理工学研究科修士論文, 2002.

[4] 安部厚志, 大岩広司,“画像処理プログラムの HDL 記述とハードウェア動作合成システムの検証”, 立命館大学理工学部卒業論文, 2007.

[5] 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.

[6] 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.

[7] 尾形秀範, 北川章夫, “アセンブリレベル合成法”, 電子情報通信学会研究報告, Vol.104, No.557 pp.35-40, 2004.

[8] 荒本雅夫,湯山洋一,小林和淑,小野寺秀俊,“動作合成における制約条件”, 電子情報通信学会総合大会講演論文集, p74, 2003.

[9] 荒本雅夫,冨山宏之,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.

[10] 神原弘之,梅原直人,中谷嵩之,石浦菜岐佐,冨山宏之,“高位合成処理システム CCAP を用いた AES 暗号処理の高速化”, IPSJ, 2007.

[11] Y.Y. Leow, C.Y.Ng, W.F. Wong. “Generating hardware from OpenMP programs”, IEEE International Conference on Field Programmable Technology, 2006.

[12] OpenMP：http://www.openmp.org/blog/

[13] 新實治男,“OpenMP による並列プログラミング入門”,ハイテクリサーチプロジェクト スーパクラスタの効率化とその応用,2003.

[14] 佐藤三久,“OpenMP”,JSPP’99 チュートリアル資料,1999.

[15] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, Ramesh Menon, “Parallel Programming in OpenMP”, MORGAN KAUFMANN, 2001.

[16] Intel C/C++ Compiler：http://www.intel.com

[17] 田村秀行：コンピュータ画像処理, オーム社, 2002.

[18] 高橋政義, 後藤裕蔵：たのしい Ruby, ソフトバンク　パブリッシング, 2002

[19] 青木峰朗,後藤裕蔵,高橋政義：Ruby レシピブック 268 の技, ソフトバンク パブリッシング,2004.

[20] 中田育男：コンパイラの構成と最適化, 朝倉書店, 1999.