社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2003－ＳＬＤＭ－112　(15)
2003／11／28

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

森江　善之†　　冨山　宏之††　　村上　和彰†††

† 九州大学大学院システム情報科学府情報理学専攻　〒816-8580 福岡県春日市春日公園 6-1
†† 名古屋大学大学院情報科学研究科情報システム学専攻　〒464-8603　名古屋市千種区不老町
††† 九州大学大学院システム情報科学研究院情報理学部門　〒816-8580 福岡県春日市春日公園 6-1
E-mail: †y-morie@c.csce.kyushu-u.ac.jp, ††tomiyama@is.nagoya-u.ac.jp, †††murakami@i.kyushu-u.ac.jp

あらまし　半導体の微細加工技術の著しい進歩による設計生産性の危機ため，動作合成システムがツール化されるようになった．しかし，動作合成ツールを利用した設計では所望の回路性能，回路規模が得られないことが多い．本論文では動作レベル記述を変換することによって回路性能，回路規模に対してどのような変化が起こるか調べ，動作レベル設計における設計・変換手法の提案を行う．

キーワード　動作合成，動作レベル設計，設計手法，変換手法

Behavior level description and Transformation pointed to efficiency in Behavioral synthesis

Yoshiyuki MORIE†, Hiroyuki TOMIYAMA††, and Kazuaki MURAKAKI†††

† Department of Informatics Graduate School of Information Science and Electrical Engineering Kyushu University　6-1 Kasuga-koen, Kasuga, Fukuoka 816-8580 JAPAN
†† Department of Information Engineering Graduate School of Information Science Nagoya University Furo-cho, Chikusa-ku, Nagoya 464-8603, JAPAN
††† Department of Informatics Graduate School of Information Science and Electrical Engineering Kyushu University　6-1 Kasuga-koen, Kasuga, Fukuoka 816-8580 JAPAN
E-mail: †y-morie@c.csce.kyushu-u.ac.jp, ††tomiyama@is.nagoya-u.ac.jp, †††murakami@i.kyushu-u.ac.jp

Abstract　For the crisis of the design productivity by remarkable progress of the deep submicron technology of a semiconductor, so that a bebavioral synthesis tool appears. However by the design with a behavioral synthesis tool we don't acquire a desired circuit performance and a circuit scale in many cases. by transforming a behavior level design description, we investigate what change takes place to circuit performance and a circuit scale in paper. we proposes the design and transformation methodology in a behavior level design.

Key words　behavioral synthesis, behavior level design, design methodology, transformation methodology

1. はじめに

半導体の微細加工技術の著しい進歩により，Large Scale Integration(以下 LSI) チップの集積度は今まで複数のチップで構成されていた機能を 1 つのチップで集積できるほど向上している．一方で，一人のハードウェア開発者の設計生産性 (設計者一人の単位時間当たりの生産性) は LSI チップの集積度の増大を補えるほどは向上していない．

現在の LSI の設計手法は，レジスタ転送レベル設計が主流である．レジスタ転送レベルる設計は，現在の回路規模に対して設計の抽象度が低く，設計生産性が悪い．将来，さらに回路は大規模な回路が集積可能となると予測されるため，レジスタ転送レベル設計では回路の設計を行うことは難しいと考えられる．設計生産性の向上のためのより高い抽象度設計手法としては，動作レベル設計があげられる．動作レベル設計では，設計対象の処理を記述することによって設計するため，設計生産性を向上させることができる．動作レベル記述を自動的にレジスタ転送レベル記述に変換することを動作合成といい，動作合成ツールもすでにいくつか開発されている．

しかし，現在の動作合成ツールを使用した設計では，設計者の意図した仕様を満たしてない回路が生成されることが多い．原因は動作合成ツールの動作合成技術が未熟であること，動作合

成ツールを使った設計事例がまだ少なく，設計手法が確立していないことなどが挙げられるが，動作合成技術への理解を深め，ノウハウを蓄積し，動作合成ツールの設計手法を確立すること改善することできると考えられる．本研究ではどのようなアプリケーションの動作レベル記述をどのように設計もしくは変換を行うことによって生成される回路性能・回路規模・構造にどのような変化が起こるかということを調べ，動作合成を用いた動作レベル設計における設計手法の確立を目指す．本論文では，手始めとして動作レベル記述の変換手法についての評価を行う．

本論文では，まず，2 章で動作合成について説明を述べる．次に 3 章で動作レベルの設計・変換手法について述べる，次に 4 章で評価実験について述べ，5 章のおわりにで本論文のまとめを述べる．

2. 準　　備

2.1 諸 定 義

動作レベル記述とは，設計対象の処理の流れやアルゴリズムを変数，四則演算，if, for, case などの制御文を基本要素とし，C 言語や SystemC といった C ベース言語を用いて記述したものである．

レジスタ転送レベル記述とは，設計対象をレジスタと演算器，メモリ，マルチプレクサなどのレジスタ以外の部分に分け，レジスタ間の接続関係にもとづく回路構造を，Verilog-HDL や VHDL といったハードウェア記述言語で記述したもので，クロックサイクル精度のタイミング情報を持つものである．

スケジューリングとは，時間制約，資源制約，もしくは両方の制約がある中において，各演算を実行するタイミングを決定することである．時間制約とは，実行サイクル数の下限がすでに決定されていることを示し，資源制約とは，演算器の数の上限と演算器の種類がすでに決定されていることを示す．

アロケーションとは，データパス回路の構成を決定するもので，レジスタや演算器，バスやマルチプレクサなどのハードウェアリソースの種類と数を決定し，リソースを固定することである．

2.2 動作合成とは

動作合成とは，動作レベル記述をレジスタ転送レベル記述に変換する作業のことである．他に動作合成や機能合成などと呼ばれることもある．一般に動作レベル記述は C ベース言語で記述される．図 1 に一般的な動作合成の処理を示す．動作合成では，まず，動作レベル記述をコントロールデータフローグラフ (CDFG) などの中間表現に変換する．次に中間表現に対して冗長性を取り除くための最適化を行う．この中間表現に対して時間制約と資源制約を決め，スケジューリングおよびアロケーションを行う．スケジューリングとアロケーションは相互に依存関係にあるので，同時もしくは繰り返し行う．最後にスケジューリングとアロケーションを行った CDFG から有限状態機械を生成し，有限状態機械をもとにレジスタ転送レベル記述を生成する．

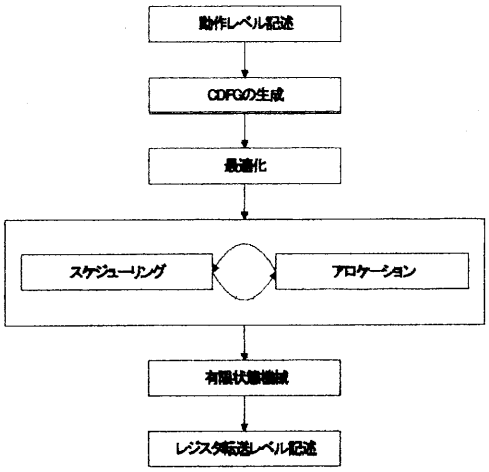

図 1　動作合成の処理

3. 設計・変換手法

3.1 設計・変換手法

動作合成では入力動作レベル記述を制御回路とデータパス回路と RAM やレジスタファイルなどの記憶素子で構成される回路へと変換し，レジスタ転送レベル記述として出力する．動作レベル記述において制御，演算，変数や配列の記述の仕方が制御回路やデータパス回路や記憶素子の構成や回路性能に影響を与える．このため，動作レベル設計で回路生成を行う際は動作合成で生成される回路構成を意識して設計する，もしくは生成したい回路の処理だけを記述し，制御回路，データパス回路や記憶素子の構成を意識した変換を行うことにより，動作合成によって生成される回路の性能や規模を改善させるべきである．

以下に回路性能や回路規模の要求によって設計を変更するための変換手法について述べる．

動作レベル記述の冗長性を取り除く変換として共通部分式削除，定数の畳み込み・定数伝播，複写の削除，手続き呼び出しの特殊化，ループ不変式の移動などがある．これらはソフトウェアのコンパイラでも最適化として行われているものである．動作レベル記述の並列性を抽出する変換として，ループ展開，ループパイプライニング，ループ融合，Tree Height Reduction などがある．動作レベル記述の演算器や記憶素子といったリソースやリソースの構成を適切なものとするための変換として演算の強度の軽減，メモリ構造の最適化，ビット幅最適化などがある．[1][2][3][4] ここでは 3 つに分けて挙げたが，それぞれに変換には色々な効果がある為，完全に分類できるものではない．変換手法は他にもまだあるが，筆者が代表的であると考えたものをあげた．

3.2 共通部分式削除

共通分式削除とは，複数回数出現する部分式について，最初に計算した値を 2 回目以降に使用することにより，演算の回数を減少させ，実行サイクル数を減少させる．以下に共通部分式削除の適応例を示す．

```
x = a * b + c;
x = a * b * c + 1;
```
というような記述があるならば,
```
tmp = a * b;
x = tmp + c;
y = tmp * c +1;
```
と変換する

3.3 定数の畳み込み・定数伝播

定数の畳み込みとは, 定数計算をあらかじめに実行することをいう. 定数の計算を先に行うことで演算を減少させることが出来る. 定数伝播とは, データフロー解析の結果, 大域的に定数の値が保たれることを利用して定数の畳み込みを行うこという. これは演算の数を減少させることため, 演算の回数が減少しているため, 実行サイクル数を減少させる. 以下に定数の畳み込みと定数伝播の適応例を示す.

```
a = 3 * 2;
...
b = a + 10;
```
というような処理がある時, 次のように変換する. 但し, b = a + 10 を実行する前に a = 3 * 2 が必ず実行されており, b = a + 10 を実行するまでの間 a = 3 * 2 の値に変化はないものとする.
```
a = 6;
...
b = 16;
```
となる.

3.4 複写の削除

複写の伝播とは,
```
a = b;
...
c = a + d;
```
というような命令文があり, c = a + d の a の値を定義するのが a = b だけで, a = b を実行してから c = a + d を実行するまでの間に b の定義が無い場合に
```
c = b + d;
```
と変換することをいう. 複写の伝播を行うことで, 意味の無い代入がなくなるため, 実行サイクル数を減少させる効果がある.

3.5 手続き呼び出しの特殊化

手続き呼び出しの展開とは, 手続き呼び出しの場所に, 呼び出される手続きの本体を展開し, 展開された部分に対して定数伝播, 定数の畳み込み, 複製の削除, 無用命令の削除などの変換を適用することによって関数の冗長な部分を削除することである. 展開されたことにより, 適応された最適化によって様々な効果がある. 以下に手続き呼び出しの特殊化の適応例を示す.
```
foo(i){
    if(i < 10)
        return i + 2;
    else
        return i * 2;
}
```
という手続き呼び出しあり,
```
x = foo(5);
```
のように呼び出されている時, 関数 foo を展開して, 定数伝播を適応する.
```
if (TRUE)
    x = 7;
else
    x = 10;
```
さらに, 条件分岐の一方しか選択されないことがわかるので, 無用命令の削除を行って,
```
x = 7
```
とすることができる. 例の場合は条件文, 演算が無くなったため, 制御回路の構造が単純になったり, 演算器を割り当てる必要がなくなったりするため, 回路性能や回路規模, 実行サイクル数において改善する.

3.6 ループ不変式の移動

ループ不変式の移動とは, ループ内で値に変化の無い計算があるならば, ループの外に出して計算を回数を減らすことをいう. これによりレジスタへの代入や演算がループの回数行われることがなくなるため, 実行サイクル数を減少させる効果がある.

3.7 ループ展開

ループ展開とは, ループの繰り返しの回数分, ループの本体を展開を行うことをいう. ループを展開することにより, ループインデックスに対する条件判定や更新のための演算が削除されたり, ループインデックスのアドレス計算が削除されたり, 演算の並列性が抽出されたりすることにより, 実行サイクル数が減少する. しかし, ループ展開の結果として状態数が増えてしまうため, 制御回路が複雑になり, 回路規模が増加する.

以下にループ展開の適応例を示す.
```
for(i = 0;i < 4;i++){
    c[i] =x + a[i] * b[i];
}
```
というループがあるならば,
```
c[0] =x + a[0] * b[0];
c[1] =x + a[1] * b[1];
c[2] =x + a[2] * b[2];
c[3] =x + a[3] * b[3];
```
と変換する. 上の例では, ループ展開を行う前は状態数は 3 でサイクル数 12 はであるが, ループ展開を行った後は状態数が 5 でサイクル数が 5 となる. 但し, 加算器, 乗算器も 1 サイクルで動作し, 配列はレジスタにマッピングしたものとする. また, 性能向上と制御回路の複雑さのトレードオフを考えてループの展開を途中まで行う部分的ループ展開もある.

3.8 ループパイプライニング

ループパイプライニングとは, 異なるイタレーションを時間的にオーバーラップをさせて実行することをいう. イタレーションとは繰り返しの基本部分のことである. イタレーション

を時間的にオーバーラップさせることは，ループパイプライニングを行っていない時より演算が並列に実行されるため，実行サイクル数を減少させる効果がある．オーバーラップされるイタレーションとイタレーションの開始間隔をスループットと呼び，イタレーションにかかるサイクル数のことをレイテンシと呼ぶ．スループットとレイテンシは短い方が良い．これはスループットが短いということは，ループパイプライニングを行わないときより実行サイクル数を減少させることを示す．レイテンシが短いということは，制御回路を簡単化させることを示すからである．ループパイプライニングは部分的ループ展開を行ない最適化なイタレーションを作ってから行うことも多い．イタレーション間にデータ依存関係がある場合のループパイプライニングをループフォールデングと呼ぶ．図2にループパイプライニングの例を示す．

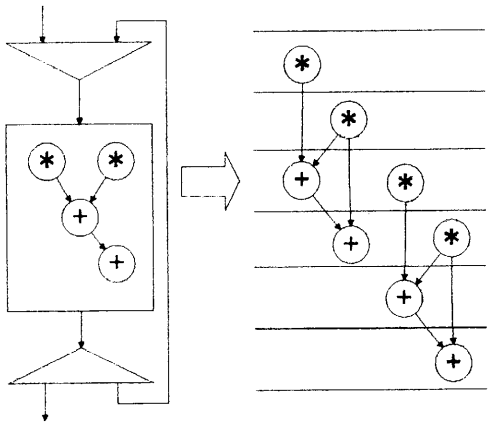

図2　ループパイプライニングの適応例

3.9　ループ融合

ループ融合とは，複数のループを1つのループに変換することをいう．制御を超えた依存関係を調べることは難しいため，複数のループがそれぞれ独立してスケジューリングされてしまう．ループ融合をすることによって，依存関係のない演算，代入を並列にスケジューリングすることが可能となるため，実行サイクル数を減少させる効果がある．また，ループインデックスの更新と条件判定式を実行する回数がループ融合する前のループの数分の1とするため，実行サイクル数を減少させる効果がある．

3.10　Tree Height Reduction

Tree Height Reduction とは，演算の結合則，分配則と交換則を利用し，演算の並列性を高めることによりDFGの最長パスを短くし，実行サイクル数を減少させる効果がある．演算が並列に実行されるため，演算器を複数必要とする．以下にTree Height Reductionの適応例を示す．

$((a * b + c * d) + b * c) + a * d;$

という命令文ならば

$(a + c) * (b + d);$

と変換する．演算器やレジスタの数を少なくしたい場合はTree Heght Reductionの逆の変換をすることもある．

3.11　演算の強度の軽減

演算の強度の軽減とは，演算器のコストや遅延などが減少するように数式を変形させることをいう．これによりコスト・遅延の小さい演算器を割り当てやすくなる．一般に加算は乗算よりコスト・遅延が小さく，シフト演算は加算よりコスト・遅延が小さいため，同じ計算結果が得られるならば，なるべく小さいシフト演算に演算へ変形を行うのような変換を行う．これは回路規模を減少させたり，動作周波数を向上させたりする効果がある．

3.12　メモリ構造の最適化

動作レベル記述における配列はメモリにマッピングされることが多い．配列がメモリにマッピングされると依存関係が無かったとしてもメモリポート数に依存して同時にアクセスすることが出来なかったり，アドレス計算ための演算が必要だったりメモリを使うことによってのネックが生まれてしまう．このようなネックが小さくなるようにメモリ構造の最適化を行う．例えば，メモリアクセスがあるたびにメモリにアクセスせずに，頻繁にアクセスのあるメモリ要素や並列性のあるメモリ要素を一時的にレジスタに保存し，アドレス計算を削減したり，メモリ要素同士の並列性を抽出したりすることでメモリネックを改善する．しかし，メモリネックを改善するためメモリアクセスをの処理をあまり複雑にすると状態数が増えるため，制御回路が複雑なり，回路規模が増加することがある．

```
for(i=0; i < 10; i++){
        k = 10 * i + 123;
}
```

というような処理があるならば，

```
k=123;
for(i=0; i < 10; i++){
        k = k+10;
}
```

と変換する．

3.13　ビット幅最適化

動作レベル記述内の各変数について，実行時において必要な最低限のビット幅を有効ビット幅といい，有効ビット幅を解析し，変数や配列に割り当てる最適化をビット幅最適化という．この最適化を行うことはCベース言語を入力とする動作合成では一般に変数や配列はレジスタやRAMとして扱われるため，ビット幅を小さくさせることになる．レジスタやRAMのビット幅が小さくなることは回路規模を減少させる効果がある．ビット幅が小さくなることにより，レジスタの遅延が小さくなるため，ビット幅最適化を施したレジスタがクリティカル・パスに関係するものであったなら動作周波数を向上させる効果がある．

4.　評価実験

動作合成ツールへの入力動作レベル記述を変換することによって回路性能や回路規模にどのような変化があるかを調べる評価実験を行う．

4.1　実験環境

実験サンプルとしてRijndaelアルゴリズムを用いた．この実

験サンプルは共通鍵暗号の一種で, 次世代の米国政府標準暗号として AES(Advanced Encryption Stadard) に選定されたアルゴリズムである. 動作合成ツールには Y Explorations Inc. の eXCite を使用した. 論理合成・配置配線は Altera の QuartusII を使用した. ターゲットデバイスとして FPGA の ApexII を指定した.

4.2 実験手順

以下の順序で初期動作レベル記述に対して変換を行う. まず, 今回ターゲットデバイスとして FPGA を選んだ. FPGA では RAM が別に用意されている. この RAM に配列がマッピングされるように配列のセル数を 176 から 256 へと初期動作レベル記述を変換した. 以上の変換を変換 1 とする.

変換 1 の動作レベル記述に対して, 演算の強度の軽減を行った. eXCite では乗算に対する演算の強度の軽減を行えるので, 除余算器に対する演算の強度の軽減を以下のように行った.

```
b[i][j]=mul(2,a[i][j])^mul(3,a[(i+1)%4][j])
  ^a[(i+2)%4][j]^a[(i+3)%4][j];
```

などを

```
b[i][j]=mul(2,a[i][j])^mul(3,a[(i+1)&3][j])
  ^a[(i+2)&3][j]^a[(i+3)&3][j];
```

と変換したり,

```
return Alogtable[(Logtable[a] + Logtable[b])%255];
```

を

```
unsigned short w;
w=Logtable[a] + Logtable[b];
if(w>255)w=w−255;
return Alogtable[w];
```

などと変換した. 以上の変換を変換 2 とする.

次に変換 2 の動作レベル記述に対して手続きの特殊化を行う. Substitution, ShiftRows, MixColumns, AddRoundKey mul を展開し, 定数伝播, 定数の折り畳み, 無用命令の削除を行った. 以上の変換を変換 3 とする.

次に変換 3 の動作レベル記述に対してループ融合を行う. この変換を変換 4 とする.

次に変換 4 の動作レベル記述に対してループ展開を行う. この変換を変換 5 とする.

次の変換 5 の動作レベル記述に対してメモリ構造の最適化を行う. eXCite では配列はメモリにマッピングされる. メモリはシングルポートであるので, 依存関係が無くても, 同時アクセスすることが出来ない. このため, メモリのレジスタへの展開を行った. 以上の変換を変換 6 とする.

変換 5,6 では状態数が増えて回路規模大きくなってしまう. このため, 以下の変換を行う. 今回の実験サンプルでは符号化を行う配列には依存関係から, 同時に 4 つの配列のへのアクセスが可能となっている. eXCite では配列はシングルポートの RAM にマッピングされてしまうので, 4 つ同時にアクセスすることは出来ない. このため, 配列を 4 つに分割して, 配列の要素に 4 つ同時にアクセスできるように変換 4 の動作レベル記述を変換した. 以上の変換を変換 7 とする.

これらの変換を行い, それぞれの変換記述を動作合成・論理合成・配置配線を行い, Logic Cell 数, 動作周波数, 実行サイクル数を調べる.

4.3 実験結果・考察

実行サイクル数に関する実験結果を図 3, Logic Cell 数に関する実験結果を図 4, 動作周波数に関する実験結果を図 5 を示す. 実験サンプルの初期動作レベル記述では実行サイクル数が 36176, Logic Cell 数が 5536, 動作周波数 76.59MHz であった. 実行サイクル数が非常に長くなっているのは実行時に多くのサイクル数が必要な除余算をループ内で繰り返し使用しているからである. 変換 1 では要素数が 176 の配列が Logic Cell を使用して合成されるという無駄であったため, 要素数を 256 の配列に変換した. この変換で Logic Cell 数が 3070 と 55.4 %に減少し, 動作周波数が 78.09MHz と向上した. 変換 2 では実行サイクル数を抑えるため, 除余算を他のサイクル数を必要としない演算器に置き換えることによって実行サイクル数が 6294 と変換 1 に比べ 17.2 %に減少した. 回路規模の大きい除余算器が無くなったため, Logic Cell 数も 1996 と変換 1 と比べ 65 %に減少し, 動作周波数も 84.86 と変換 1 と比べ 108.7 %に向上してる. 変換 3 では手続きの特殊化により定数伝播, 定数折り畳み, 無用命令の削除を行うため, 状態数が減少し, Logic Cell 数が 1792 と変換 2 と比べ 89.7 %や実行サイクル数が 4286 と変換 2 と比べ 68.1 %にそれぞれ減少した. 変換 4 ではループ融合を行うことによりループインデックスに対する加算や比較をの回数を減らした. このため, 実行サイクル数 2978 と変換 3 と比べ 69.5 %に減少した. 状態数が増加したため, Logic Cell 数が 1914 と変換 3 と比べ 106.8 %に増加した.

変換 5 ではループ展開を行うことによりループインデックスに対する加算や比較やアドレス計算が無くなったため, 実行サイクル数が減少した. しかし, ループ展開を行ったため状態数が増加し, Logic Cell 数が 5500 と変換 4 と比べ 287.3 %に増加した. 回路が複雑になり, 動作周波数も 44.85 と変換 4 と比べ 57.1 %に減少した.

変換 6 では RAM としてマッピングされた配列要素を変数へと変換し, レジスタとしてマッピングするための変換を行った. RAM がシングルポートのために並列にアクセスが出来なかったメモリ要素がレジスタとなって並列にアクセスが可能となったため, 実行サイクル数が 368 と変換 5 と比べ 25.6 %に減少した, 制御回路はループ展開を施したものとほぼ同等のため, Logic Cell 数が 5424 と変換 5 と比べ 98.6 %. 動作周波数は 50.85MHz と変換 5 と 113.4 %となった.

変換 7 では配列を 4 つが RAM にマッピングされており, 4 つに並列にアクセスできるため, 実行サイクル数が 1473 と変換 4 と比べ 49.5 %に減少した. 変換 6 同等の実行サイクル数となった. しかし, ループ展開を行っているわけでないので, 状態数が大幅に増加することがないため, Logic Cell 数が 2478 と変換 4 に比べ 129.5 %とさほど増加しなかったし, 動作周波数も 75.22MHz と変換 4 に比べ 95.8 %とそれほど減少しなった.

これらの変換から, 仕様を満たす動作レベル記述があれば, 動作合成を用いた設計で十分設計可能であると言える.

変換全体を通して配列やループに関する変換が特に効果が

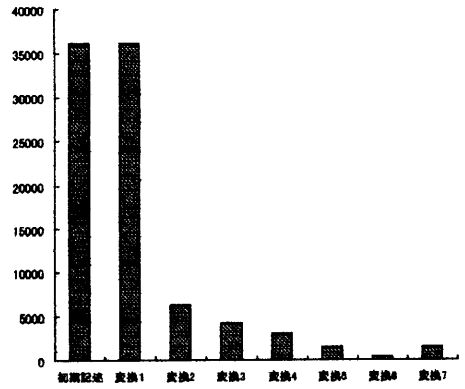

図 3 実行サイクル数

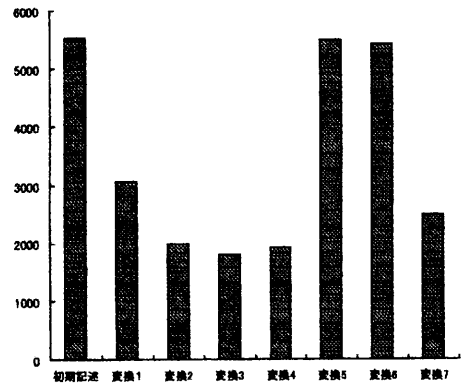

図 4 Logic Cell 数

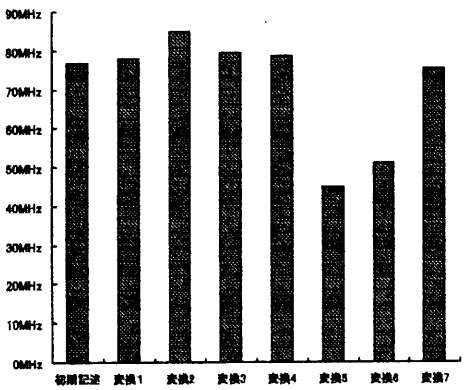

図 5 動作周波数

あった. これはメモリのアドレス計算やメモリアクセスのループの繰り返しで起こるデータの滞りなどが動作合成で生成される回路の中で性能の支配的な部分となっているからだと考えられる.

5. おわりに

本論文では, 動作合成ツールを使用した動作レベルにおける設計の設計生産性と回路性能, 回路面積を実用に耐えうるものとするために設計・変換手法を提案し, 変換手法についての評価実験を行った. 実験では動作合成ツールの出力回路構成を意識した設計・変換を行うことで回路性能や回路規模に変化が起こった. 今後の課題としては設計・変換手法, 特に並列性を抽出するための変換は動作合成ツールにおける内部表現やスケジューリング, アロケーションと密接に関係しているため, これを詳細に調査することと動作レベル設計で生成される回路のメモリ構成やループ部分の回路構成について調査することが挙げられる.

謝　辞

本研究は一部, 日本学術振興会 科学研究費補助金 基盤研究 (A)(2) 展開研究「システム LSI 向けカスタム化可能 IP コアのアーキテクチャおよび設計支援技術の開発」(課題番号 : 12358002), 日本学術振興会 科学研究費補助金 基盤研究 (A)(2) 展開研究「ハードウェア構成を動的に最適化する「ハードウェア・モーフィング」技術の開発」(課題番号 : 13308015) による. 九州大学安浦・村上・松永研究室の諸氏に感謝致します.

文　献

[1] D. Gajski, N. Dutt, A. Wu, and S. Lin, "High-Level Synthesis: Introduction to Chip and Design, " Kluwer Academic Publishers, 1992

[2] A. V. Aho, R. Sethi, and J. D. Ullman, "Compiler: Principles, Techniques, and Tools," Addison-Wesley, 1986

[3] 中田育男, "コンパイラの構成と最適化," 朝倉書店, 1999 年

[4] H. Yamashita, H. Tomiyama, A. Inoue, F. N. Eko, T. Okumura, and H. Yasuura, "Variable size analysis for datapath width optimization," In Proc. of 5th Asian Pacific Conf. on Hardware Description Languages(APCHDL), pages 69-74, Jul. 1998.