有限状態機械（FSM）とシンボリック状態探索を利用したコード生成手法

瀬戸謙修[†] 藤田昌宏[††]

高速処理が必要な組み込みシステム向けLSIとして，並列命令や特殊なデータパスユニット，レジスタ構成あるいはバス構成を持つ，アプリケーションに特化したプロセッサが数多く組み込み向け市場に出現している[5)~10)]．しかし従来のコンパイラ技術[1)]だけでは，このような特殊なプロセッサの性能を十分引き出すコードを生成できない．この問題を解決する1つの方法として論文2)，3)ではまず，コード生成対象のデータフローグラフ（DFG）と，対象とするアーキテクチャの命令セットから有限状態機械（FSM）を作り，そしてそのFSMの状態を1つずつ探索することによって最適コード生成を行う方法を提案している．本稿では論文2)，3)と同様のアプローチをとるが，それらの論文と異なりFSMからコード生成する際にFSMをシンボリックに解析することで，はるかに多くの状態を高速に探索したうえで，より最適な並列コードを生成する方法を提案する．論文21)~23)も本稿と同様にFSMのシンボリック状態探索によりマイクロコードを生成する手法を提案しているが，オペランドの寿命が判定できないため，必要以上に多くの状態を計算してしまう可能性がある．本稿では新たなFSM変数を導入してオペランドが保持する必要があるかどうかを決定するような条件を導き，不要な状態の削減を図る．さらに本稿ではRAMへのスピルおよびリロードも考慮し，コード生成のすべてのフェーズを同時に実行したうえでステップ数最小のコードを生成する．最後に提案手法の実験結果を示す．

Code Generation Using FSM and Symbolic State Traversal

Kenshu Seto[†] and Masahiro Fujita[††]

As key components in high-speed embedded systems, a number of application specific processors with parallel instructions, special datapath units, registers or bus architecture are coming into the embedded market [5)~10)]. However, it is difficult to generate code that fully utilize hardware units in these processors only by traditional compiler techniques [1)]. To resolve this problem, Roemer et al. [2),3)] built FSM from given DFG (Data Flow Graph) and instruction set. Then they traversed the states of the FSM explicitly to generate code. In this paper, we basically use the same approach with the paper [2),3)]. Differently from the approach, our approach analyzes the FSM symbolically and traverses much more states so that it can possibly generate parallel code with smaller number of steps in less time. Monahan et al. [21)~23)] also proposed the use of symbolic state traversal to generate microcode. However the approach cannot determine operand lifetime so that it may produce many unnecessary states. Our approach uses a new FSM variable to build a condition which determines if an operand must be kept or not so as to reduce the unnecessary states. In addition, our approach generate codes with the minimum number of steps in completely phase-coupled manner considering memory spills and reloads. Finally, experimental results are shown.

1. はじめに

1.1 背景

画像処理，通信処理等の組み込みシステム向けLSIでは，要求仕様を満足させるためにハイパフォーマンス動作が必要不可欠である．たとえば，基幹ルータ用の規格であるOC-192では，1パケットの処理を処理クロックが1 GHzでは50ステップ以内，200 MHzでは10ステップ以内で行う必要があると見積もられている[4)]．また，このような高速処理への対応だけでなく，設計期間を短縮し，頻繁な仕様変更に迅速に対応していく必要もある．

このような2つの要求に対して，すべてをASICとして専用ハードウェアで設計する手法では，パフォーマンスの点で満足できるが，設計期間が膨大となって

† パシフィック・デザイン株式会社
Pacific Design Inc.
†† 東京大学工学部電子工学科
Department of Electronics Engineering, University of Tokyo

しまう．一方，一般の汎用プロセッサを使用したソフトウェア設計では，設計期間は満足できるがパフォーマンスが不足する．

そこでこのようなトレードオフを満足する最良のシステム LSI として，画像処理や，通信処理等の特定のアプリケーションに特化したハードウェアを有するプロセッサが数多く組み込み向け市場に出現している[5)~10)]．これらのプロセッサでは，アプリケーションに特化した専用命令を用意することでハイパフォーマンスの実現を目指している．同時に，プロセッサベースであるため設計期間の短縮，頻繁な仕様変更への対処も可能である．

このようなプロセッサベースのシステム LSI の設計手法としては，C，Java 等の高級言語を使用した抽象度の高いレベルでの設計手法が望ましい．これらの高級言語でプログラムできれば，アセンブリ言語による設計に比べて設計期間を大幅に短縮することができ，またコードの再利用も可能となる．

このような専用プロセッサではプログラムの性能向上の一手法として，プロファイラを使ってアプリケーションプログラム中からボトルネックとなる部分を見つけ出し，その部分を重点的にスピードアップする方法が考えられる．通常そのような部分は比較的単純な処理を行っているループとなることが多く，そのループ本体の基本ブロック（DFG）に対して，時間をかけてコンパイルすることにより，できるだけステップ数の少ないコードを出力することが望まれる．

しかしながら，以上で述べたようなアプリケーションに特化したプロセッサの設計では次のような問題がある．それらのプロセッサではハイパフォーマンスを実現するために，並列命令や特殊なデータパスユニット，さらには特殊なレジスタ構成あるいはバス構成等，VLIW と DSP が融合したような特殊なアーキテクチャをとることが多いが，これらに対して従来のコンパイラ技術[1)] では十分満足できるコードが生成できないという問題である．たとえば，従来のコンパイラ技術では並列命令は基本的に扱えない．またレジスタファイルは 1 つのみで，レジスタファイル内のデータに対してすべての演算器が等しくアクセス可能であるような，限定されたアーキテクチャを扱っている．さらに，基本ブロックはデータフローグラフ（DFG）ではなく，データフローツリー（DFT）に分解して扱っている．以上のような制限があるため，従来のコンパイラ手法[1)] で生成されるコードの品質は悪くなる場合が多い．

以上の問題点から，特殊なハードウェアを持つプロセッサ用コンパイラの研究が重要な研究課題となっている．このような目的のためのコンパイラでは汎用プロセッサ用のコンパイラと異なり，コンパイル時間をある程度かけてもよいからできるだけステップ数あるいはコード量の小さなコードを生成することが望まれる．

また効率的なコードを生成するだけでなく，対象アーキテクチャの命令セットが，自動的にそのアーキテクチャ向けのコンパイラを生成するコンパイラ，すなわちリターゲッタブルコンパイラも重要である．リターゲッタブルコンパイラを使えば，与えられたアプリケーションに対してどのような専用命令を用意すればパフォーマンスが向上するか，迅速な評価が可能となる．以下にこのようなコンパイラの研究のうち，本稿に関連深いものをいくつか取り上げ説明する．

1.2 関連研究

このような特殊なアーキテクチャのコード生成を扱った仕事として，Araujo らの論文[17)] がある．アーキテクチャとして $[1, \infty]$ モデルという限定的なモデルを仮定し，さらに RTG 条件と呼ばれる独自の条件が成り立つものを仮定している．この条件のもとで，$O(n)$ という高速な時間で最適コードを生成する方法が提案されている．しかしこのアルゴリズムではアーキテクチャが限定されているうえに，並列コードの生成はできない．

この問題点を解決する 1 つの方法として，Roemer ら[2),3)] はまず，コード生成対象のデータフローグラフ（DFG）と，与えられた対象とするアーキテクチャの命令セットから有限状態機械（FSM）を作り，そしてその FSM の状態探索によって最適コード生成を行う方法を提案している．その方法では，DFG ノード N の出力オペランドがレジスタ R にある場合 1，それ以外のときに 0 となるようなブール変数の集合を FSM の状態変数として用意する．この FSM の初期状態を DFG の計算を始める前の状態（メモリあるいはレジスタに DFG の外部入力ノードの出力オペランドが格納された状態），最終状態を DFG の計算が終了した状態（メモリあるいはレジスタに DFG の外部出力ノードの出力オペランドが格納された状態）とし，初期状態から最終状態までプロセッサの命令を実行することによって状態遷移させる．このように状態遷移させたとき初期状態から最終状態までの状態遷移列が求めるコードを与える．良いコードを見つけるには，できるだけ多くの状態を短時間で探索する技術が必要である．

論文 3) では，その状態探索の方法として FSM の現状態を 1 つ 1 つ取り出し，それらに対して次状態を 1

つずつ列挙する方法をとっている．この方法ですべての状態を探索し続けるのは状態数が爆発してしまい不可能なため，各ステップで状態遷移を続ける状態の数を，適当なヒューリスティックを使用して M 個（ただし $M \leq 50$）までに絞っている．

しかしながら，以上の方法には次のような問題点がある．まず各レジスタ要素に対してブール変数を設けているため，たとえば 16 個のレジスタを持つアーキテクチャを使用して 20 個のノードからなる DFG のコードを生成しようとすると，16×20 のブール変数が必要となり FSM の状態空間の大きさは $2^{16\times20}$ と巨大となる．そのため通常のサイズのレジスタファイルを扱うことができない．また各ステップで状態探索の対象とする状態の数を M 個（ただし $M \leq 50$）に絞っているが，そのように制限しても数えあげによる状態探索では時間がかかってしまっている．またヒューリスティックを使用して探索対象の状態を大幅に限定しているため，必ずしも良いコードを生成するとは限らない．

Brewer グループは以上の Roemer らの研究[2),3)] の問題点を，いくつか解決している[20)~24)]．なかでも Monahan による論文[21)~23)] は本稿と近いため，以下にこの論文についてもう少し説明する．

その論文[21)~23)] は，すでに設計済みの DSP データパス上で，与えられた DFG を計算するためのコントロール信号列，すなわちマイクロプログラムを生成する手法を提案している．基本的な考え方は論文 2)，3) と同じであり，問題を FSM で定式化した後，FSM の状態探索により解を見つけ出す方針である．論文 2)，3) と大きく異なる点は，その FSM をシンボリック状態探索することにより，最小ステップ数のマイクロコードを比較的短時間で探索している点である．またレジスタファイルも扱っている．

問題を定式化した FSM にそのままシンボリック状態探索を適用すると時間がかかりすぎるため，論文 21)~23) は最適解に寄与しない無駄な状態および状態遷移を削減し，CPU 時間を短縮する有用な工夫をいくつか提案しており，本稿の手法に対しても有効なものが多い．その中の一例を，以下に説明する．まず与えられたステップ数制約の下で，データ転送のオーバヘッドまで考慮した DFG の ALAP（As Late As Possible）スケジューリングを行う．その結果，各オペレーションに対して，ステップ数制約を満たす範囲で一番遅いスケジューリングステップ数が求まる．オペレーションがこのステップ数より遅くスケジューリングされるような状態は，制約を絶対に満たさないので削除することができ，状態探索を高速化可能である．

この論文の問題点は以下のとおりである．まず，DFG 中のオペランドの再計算を許しているため，オペランドがいつ不要になるか判定するのが難しく，その判定方法が示されていない．すなわちどのオペランドをレジスタファイルから削除するのがよいか不明である．その結果不要なオペランドを持つ状態や，逆に必要なオペランドを削除した状態のように，最適コード生成に寄与しないような状態がかさむため，扱う状態数が増大し CPU 時間がかかるようになると思われる．確かにオペランドの再計算を自由に行うことでよりステップ数がより小さいコードが生成される可能性があるが，必ずしも実際にはそのようなケースはあまり多くないと考えられる．また他の問題として，コード生成で通常行う必要がある，レジスタファイルに格納されたオペランドの RAM への退避（スピル）およびその復帰（リロード）が扱われていない．すなわちコード生成のサブタスクであるレジスタアロケーションが完全には考慮されていない．

1.3 本稿の提案手法の新規性

本稿では，論文 2)，3) と同様のアプローチをとるが，FSM からコード生成する際に FSM をシンボリックに解析することで，論文 3) に比べてはるかに多くの候補の中から一番良い解を選ぶ方法を提案する．

またシンボリック状態探索を使用した FSM の状態探索を行う点では論文 21)~23) と同様のアプローチをとるが，それらの論文ではオペランドの再計算を自由に許しているためオペランドの寿命を判定することが難しい．本稿では新たな FSM 変数（カバー状態変数）を導入してオペランドの寿命を制限したうえで，寿命を終えた不要なオペランドだけをレジスタファイルおよびメモリから削除するような定式化を行う．この結果，論文 21)~23) に比べ不要な状態を抑え，CPU 時間の短縮を図る．なお本稿のアプローチではオペランドの再計算が行えないため，論文 21)~23) よりも解の候補が少なくなる．しかし多くの場合，その影響はほとんどないと思われる．さらに本稿の手法は，レジスタファイルに格納されたオペランドの，RAM への退避およびその復帰も考慮し，コード生成のすべてのフェーズを同時に実行したうえでステップ数最小のコードを生成する．

1.4 本稿の構成

本稿の構成を述べる．まず 2 章で本稿を読み進めるのに必要な予備知識の説明および用語の確認を行う．3 章で提案手法の概要を説明し，簡単な例題を示す．4 章で FSM およびシンボリック状態探索を利用

したコード生成の新しい定式化を提案する．5 章では，FSM を利用したコード生成で状態探索の時間を短縮する工夫を示す．6 章で本稿の方法の実装結果を示し，7 章で結論を述べる．

2. 準　　備

本章では，用語の確認および予備知識の説明をする．特にコード生成，有限状態機械（FSM）およびシンボリック状態探索について簡単に説明する．

2.1 コード生成

コンパイラのプログラム構成は，フロントエンド，最適化フェーズ，コード生成と大きく 3 つに分類される．まずフロントエンドで，入力された高級言語を読み込み CDFG（コントロールデータフローグラフ）等の中間表現に落とす．次に最適化フェーズでは，読み込まれた中間表現に対して様々な最適化を行う．最後にコード生成で，最適化処理後の中間表現を，ターゲットプロセッサのアセンブリ命令列に変換する処理を行う．

本稿ではコード生成の新しい方法を提案する．特に，CDFG 中の各基本ブロックに対するコード生成手法を対象とする．ただし，コード生成では CDFG 中で基本ブロック間を結合している各コントロールフローに対して分岐命令等を生成する必要があるが，このようなコントロールフロー部分のコード生成方法については本稿の対象外とする．

各基本ブロックはデータフローグラフ（DFG）の形で表現される．DFG は，計算すべき式を DAG（閉路なし有効グラフ）で表現したものである．DFG の部分グラフは，計算すべき式の部分式を表す．DFG において，各ノードは演算，メモリアクセス等のオペレーションを表す．ノード N のオペレーションの結果生成される値を，ノード N の出力オペランドと呼ぶ．またノード N のオペレーションに必要な入力値を，ノード N の入力オペランドと呼ぶ．なお，各ノードの出力オペランドを正しく計算するには，正しい入力オペランドを使用しなければならない．そのような入力オペランドとして，通常他のノードの出力オペランドが使用される．入力オペランド，出力オペランドを区別する必要がない場合は，単にオペランドと呼ぶ．DFG において，各エッジはノードの間のデータ依存性を表す．具体的には，ノード N のオペレーションがノード M の出力オペランドを入力オペランドとして使用するとき，ノード M からノード N にエッジが描かれる．DFG において，ノード N の出力に直接接続されたノードを N のファンアウト，入力に直接接続されたノードを N のファンインと呼ぶ．DFG 中でファインインを持たないノードを外部入力ノード，ファンアウトを持たないノードを外部出力ノードと呼ぶ．

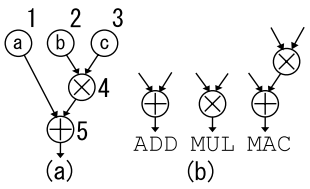


図 1 DFG と命令パターン
Fig. 1 DFG and instruction patterns.

図 1 (a) に DFG の例を示す．この DFG は，計算式 $a + b * c$ をグラフで表したものである．外部入力ノード 1，2，3 はそれぞれ DFG の入力変数 a，b，c を表し，ノード 4 は乗算，外部出力ノード 5 は加算を表す．また，ノード 1，2，3 の出力オペランドとは，（メモリに格納された）変数 a，b，c の値，ノード 4 の出力オペランドはノード 2 および 3 の出力オペランドを乗算した結果の値，ノード 5 の出力オペランドはノード 1 および 4 の出力オペランドを加算した結果の値である．

コード生成は次の 3 つのフェーズから構成される．

- コード選択
- レジスタアロケーション
- コードスケジューリング

コード選択について，ここではグラフベースのコード選択法[18),19)] を説明する．グラフベースのコード選択法では，命令セット中の各アセンブリ命令を命令パターンと呼ばれる DAG で表現する．なお，命令パターン中で使用されるノードのオペレーションの種類としては，DFG 中のノードのオペレーションの種類と同じものが使用される．命令パターンも，DFG と同様に外部出力ノード，外部入力ノードを持つ．命令が実行されると，所定のロケーション（メモリあるいはレジスタファイル）から値を読み取り，所定のロケーションに結果を格納する．なお以降では問題のない限り，命令および命令パターンを用語として区別せず使用する．図 1 (b) に例として，3 つの命令パターンを示す．図において `ADD`，`MUL`，`MAC` はそれぞれ，加算命令 $\mathtt{R1} \leftarrow \mathtt{R2} + \mathtt{R3}$，乗算命令 $\mathtt{R1} \leftarrow \mathtt{R2} \times \mathtt{R3}$，積和命令 $\mathtt{R1} \leftarrow \mathtt{R2} + \mathtt{R3} \times \mathtt{R4}$ に対応した命令パターンを表す．

命令 P と DFG 中の部分グラフとのパターンマッチングが成功したとき，命令 P は DFG にマッチするという．特に，命令 P のパターン中の外部出力ノードが DFG 中のノード N と対応するとき，ノード N において命令 P はマッチするという．このとき命令

P によって覆われた DFG の部分グラフを，命令 P のノード N におけるマッチ M と呼び，命令 P をマッチ M に対応する命令と呼び，マッチ M は命令 P によって計算される．

マッチ M によって表される DFG の部分グラフの計算を実行することを，マッチ M の実行と呼ぶ．ただし実際には，マッチ M を実行するということは，そのマッチ M に対応する命令 P を実行することを意味する．一般に命令の並列実行とは，あるクロックサイクルの間に，複数の命令を同時に実行することである．対応する命令の並列実行により，複数のマッチの同時に実行することが可能であり，このことをマッチの並列実行と呼ぶ．

マッチが正しく実行されるには，マッチの入力ノードに正しい入力オペランドが用意されている必要がある．ノード N における命令 P によるマッチ M の実行により，マッチの出力ノード N の出力オペランドが計算され，それが命令 P で定められたロケーション L に格納される．このときマッチ M はノード N の出力オペランドをロケーション L に格納するという．なお，マッチの出力ノードとは命令 P が表すパターン中の外部出力ノードに対応する DFG 中のノード N のことを表す．一方マッチの入力ノードとは，マッチの計算に必要な入力オペランドを提供するような DFG 中のノードの集合のことである．

DFG 中のノード N がマッチ M に含まれ，かつそのマッチが実行されたとき，ノード N はマッチ M にカバー（被覆）されたという．DFG の命令セットによるカバーとは，DFG の各ノードを命令セット中の命令パターンのいずれかで洩れなく覆うようなマッチの集合のことをいう．DFG のカバーのうち決められたコストを最小化するカバーを選ぶ問題を，コード選択問題と呼ぶ．

図 1 の DFG および命令パターンを使用した場合のマッチングの例を**図 2** に示す．図 2 (a) には，ノード 5 に命令 ADD がマッチし，ノード 4 に命令 MUL がマッチした様子を示してある．ノード 5 における命令 ADD によるマッチ $M1$ は，ノード 5 のみからなる DFG の部分グラフであり，ノード 4 における命令 MUL によるマッチ $M2$ は，ノード 4 のみからなる DFG の部分グラフである．マッチ $M1$ の入力ノードはノード 1 および 4 であり，出力ノードはノード 5 である．同様に，マッチ $M2$ の入力ノードはノード 2 および 3 であり，出力ノードはノード 4 である．

一方，図 2 (b) には，ノード 5 に命令 MAC がマッチした様子を示した．このときノード 4，5 は命令 MAC にカバーされている．ノード 5 における命令 MAC によるマッチ $M3$ は，ノード 4 および 5 からなる部分グラフである．マッチ $M3$ の入力ノードは，ノード 1，2，3 であり，出力ノードはノード 5 である．なお外部入力ノード 1，2，3 にはマッチを行わないものとする．図 2 (a) および (b) はそれぞれ DFG の異なるカバーを表す．たとえば命令数をコストにとった場合，コード選択問題の最適解として 1 命令のみで済む図 2 (b) のカバーが選択される．

以上のような命令パターンを持つ命令によるマッチ以外に，転送命令系のマッチがある．転送命令では計算は行わず，オペランドの移動を行うだけのため，通常命令パターンは持たない．

転送命令には，異なるレジスタファイルあるいはメモリ間の転送命令，スピル命令，リロード命令の 3 種類ある．スピルとは，レジスタファイル内に格納されているオペランドを一時的にメモリに退避することを意味する．一方リロードとはその逆で，一時的にメモリに退避されたオペランドをレジスタファイルに復帰することを意味する．スピル命令およびリロード命令は，オペランドを格納するレジスタが不足する場合に，レジスタファイル内の領域を一時的に空けるために実行される．なお命令セット中，スピルにはストア命令，リロードにはロード命令が使用される．例として図 1 の DFG で，レジスタファイルに格納されたノード 4 の出力オペランドをメモリにスピルしたり，あるいはメモリから，スピルした出力オペランドをレジスタファイルにリロードしたりするのが転送命令系のマッチである．

次にレジスタアロケーションおよびコードスケジューリングの説明に移る．レジスタアロケーションでは，各オペランドを，必要に応じて特定のレジスタに割り当てる．もし利用可能なレジスタの数よりも保持しておくべきオペランドの数が多い場合には，レジスタ数の制約を満たすために，適当なオペランドを選んでメモリに割り当てる．そのためのスピル命令，リロード命令等の転送系命令を挿入するのもレジスタアロケーションのフェーズである．

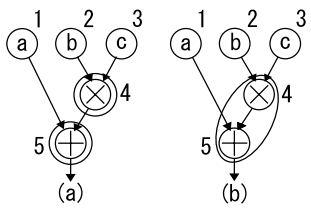


図 2 DFG および命令パターンのマッチング
Fig. 2 Matching between DFG and instruction patterns.

コードスケジューリングでは，コード選択で選択された命令や，レジスタアロケーションで導入された転送系命令の実行順序を，リソース制約やデータ依存関係を考慮したうえで決定する．各クロックサイクルごとに，ある命令が実行開始され，他の命令の実行が完了する．クロックサイクルのことをステップとも呼ぶ．並列命令は通常，スケジューリングのフェーズにおいて，複数の命令を同一ステップにパッキングすることによって生成される．アセンブリプログラムのステップ数とは，プログラムの実行を開始してから終了するまでのトータルステップ数のことをいう．プロセッサの動作周波数が同じであれば，アセンブリプログラムのステップ数は小さいほど実行時間が短い．

従来のコンパイラ技術[1]では，これら3つのフェーズを一度に行うのは計算時間がかかるため，各フェーズを1つずつ順次実行していた．しかしながら，これら3つのフェーズは本来相互密接に関連しあっているため，各フェーズを個別に実行した場合出力コードの最適性は失われてしまう．本稿ではこれら3つのフェーズを同時に行ったうえで，ステップ数最適なコードを出力する方法を示す．この方法では，有限状態機械（FSM）およびその状態探索を利用して定式化するため，次にそれらについて説明する．

2.2 有限状態機械（FSM）とその状態探索

まず有限状態機械（FSM）の定義を述べる．FSMはブール変数でエンコードされた入力，出力，状態を持つ．$x = \{x_1, \ldots, x_n\}$，$y = \{y_1, \ldots, y_n\}$，$w = \{w_1, \ldots, w_p\}$ をそれぞれ，ブール値 $B = \{0, 1\}$ のいずれかの値をとるブール変数の集合とする．このとき有限状態機械（FSM）M は次の3つのブール関数で表される．

$$M =< T(x, w, y), I(x), F(x) >$$

ここで状態遷移関係 $T : B^{2n+p} \rightarrow B$ は，w によってエンコードされた入力によって，x によってエンコードされた状態が y によってエンコードされた状態へ状態遷移するときに1となり，それ以外では0となる関数である．また $I : B^n \rightarrow B$ は，x によってエンコードされた状態が初期状態のとき1となり，それ以外では0となる関数である．$F : B^n \rightarrow B$ は，x によってエンコードされた状態が最終状態のとき1となり，それ以外では0となる関数である．集合 x，y および w をそれぞれ現状態変数，次状態変数，入力変数と呼ぶ．なお，状態遷移は各クロックサイクル（ステップ）ごとに起こるものとする．

このとき状態の集合 S はブール関数 $R(x)$ で表すことができる．ただしブール関数 $R : B^n \rightarrow B$ は，x によってエンコードされた状態が状態集合 S に属するとき1，それ以外では0となる関数である．

状態と入力のペアの列 $(x_0, w_0), (x_1, w_1), \ldots, (x_l, w_l)$ を経路と呼ぶ．ただし，$T(x_i, w_i, y_{i+1}) = 1$ である．また，次状態変数 y_{i+1} の各ブール変数に現状態変数の対応するブール変数を代入したものを，x_{i+1} とする．

初期状態 $S_i = \{x | I(x) = 1\}$ からスタートして，FSMが到達しうる状態を調べることを状態探索と呼ぶ．本稿では特に，初期状態 $S_i = \{x | I(x) = 1\}$ から最終状態 $S_f = \{x | F(x) = 1\}$ にできるだけ最短で到達するような経路の探索を行う．

FSMの状態探索の効率的な方法として，シンボリック状態探索と呼ばれる方法が提案されている[11),12)]．現状態を順番に1つずつ取り出してその次状態を1つ1つ列挙していく素朴な方法とは異なり，シンボリック状態探索では複数の状態を1つのブール関数で表現し，その次状態の集合を一度の計算で求める．多くの場合明示的な数えあげの方法に比べて，シンボリック状態探索によって指数関数的な探索速度の向上が望める．

次にシンボリック状態探索の方法についてもう少し詳しく説明する．シンボリック状態探索では，状態遷移関係 $T(x, w, y)$ および状態集合 $R(x)$ 等のブール関数をBDD（Binary Decision Diagram，二分決定グラフ）[13),14)]で表現する．このとき現状態の集合 $R_k(x)$ から到達可能な次状態の集合 $R_{k+1}(x)$ は次の像計算の式で計算される．

$$R_{k+1}(y) = \exists x, w(T(x, w, y) \cdot R_k(x)) \qquad (1)$$

像計算の式(1)に現れる各操作は，BDD上の操作として実現できる．式(1)を繰り返し適用して，初期状態 $R_0(x)$ から到達可能な状態集合を探索する方法が，シンボリック状態探索である．

このシンボリック状態探索を使用すれば多くの状態を短時間で探索できるが，問題点もある．まずFSMの規模が増大すると，状態遷移関係 $T(x, w, y)$ を表すBDDのサイズ（BDDのノード数）が非常に大きくなってしまう．また同じくBDDで表された状態集合 $R_k(x)$ についても，状態遷移につれて状態数が増大し，BDDのサイズが増大してしまう．次状態の計算のために，式(1)により存在作用子 $\exists$ を計算する必要があるが，扱うBDDのサイズが大きいために計算が終了しなくなってしまう．以上の理由から，シンボリック状態探索によるコード生成では $T(x, w, y)$ および $R_k(x)$ を表すBDDができるだけ簡単になるような工夫が必要となる．これらの工夫については，5章で議論する．

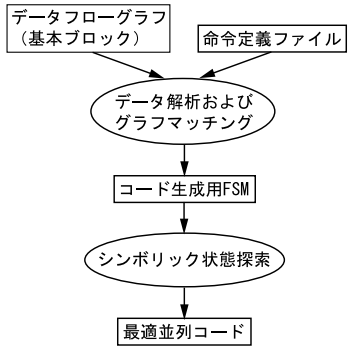


図 3 コード生成プログラム全体の流れ
Fig. 3 Overall flow of our code generation program.

3. 提案手法の概要

ここでは提案手法について，全体の流れを説明した後，例題を使用してアイデアを具体的に解説する．

3.1 コード生成全体の流れ

図 3 に全体の流れを示す．コード生成プログラムには，対象の DFG を記述したファイルおよび対象プロセッサの命令定義ファイルの 2 つを入力する．

DFG はコンパイラのフロントエンドによって生成され，最適化フェーズで最適化されたものを使用する．また DFG がツリーからなる場合，共通部分式をまとめることで DAG に構成しなおす．なお DFG のファイル内には，DFG 中の各変数がどのメモリあるいはレジスタファイルへ割り当てられるかも記してある．

一方，命令定義ファイルには対象プロセッサのメモリ構成，レジスタ構成，リソース情報およびコード生成に必要な命令情報が記述してある．命令情報としては，ニーモニック，命令のグラフパターン表現，ソースレジスタ，デスティネーションレジスタ，および使用するリソースの情報を含んでいる．

これら 2 つのファイルをもとにして，以下に示す流れで最適並列コードを生成する．その際，最適化の目的としてステップ数の最小化を考える．

以上の 2 つのファイルを読み込んだ後で，DFG と命令パターンの間のグラフマッチングを行い，すべてのマッチを生成する．DFG ファイル，命令定義ファイルおよび生成されたマッチの情報を解析することにより，シンボリック状態探索によるコード生成を効率良く行えるような FSM を作る．この方法については，4 章で詳しく説明する．

このような FSM に対してシンボリック状態探索を行う．コード生成用 FSM では，各状態遷移が各ステップで実行する並列命令に相当する．そのため，必要なオペランドがメモリ等に用意されている初期状態から，すべての計算が終了した最終状態までの最短経路を求めることでステップ数最小の並列コードが生成できる．深さ優先探索と組み合わせることも可能である．

なお簡単化のため，すべての命令が 1 ステップで実行できるものとする．もちろん本稿で提案した手法によって，複数ステップかかるようなマルチサイクル命令あるいはパイプライン命令も扱うことができる．その方法について次に簡単に概要を説明する．実行に k ステップかかるような命令によるマッチには，$\log k$ ビット分の状態変数を用意する．これらの状態変数はマッチの実行状態を表すカウンタの役目をする．マッチの実行が開始されると状態変数（カウンタ）の値が k にセットされ，各サイクルごとにその状態変数をデクリメントしていく．このような状態変数を利用して，マッチの実行状態に応じて使用するリソースを変化させたり，k ステップ経過した時点で出力オペランドを所望のロケーションに格納するような FSM を構成できる．

3.2 例　　題

前節では，本稿で提案するコード生成手法について，全体の流れを説明した．ここでは例題を使用してもう少し具体的に解説する．例題としては，図 1 に示した DFG および命令セットを使用する．また，次のような簡単なアーキテクチャを仮定する．

例題のアーキテクチャとして，メモリおよびレジスタファイルを 1 つずつ持つものとし，それぞれ M，R と表す．なお，これらのオペランドの格納場所をロケーションと呼ぶ．また命令 `ADD`，`MUL`，`MAC` はレジスタファイルからオペランドを読み，再びレジスタファイルに結果のオペランドを書き込むものとする．リソース制約については，最大 2 つまでのメモリ転送命令に加えて最大 1 つまでの演算命令が並列実行可能とする．

提案手法の流れは次のようになる．まず，DFG および命令パターンのマッチングを行う．ノード N における命令 P によるマッチを $m_{N,P}$ と表すことにすると，この例題ではマッチングの結果 $m_{4,\mathtt{MUL}}$，$m_{5,\mathtt{ADD}}$，$m_{5,\mathtt{MAC}}$ の 3 つのマッチが得られる．一方転送命令系のマッチとして，ノード N の出力オペランドをロケーション l からロケーション l' に転送する命令によるマッチを $t_{N,l,l'}$ と表すことにする．このときレジスタファイル R に格納されたノード 1，2，3，4，5 の各出力オペランドをメモリ M にスピルするマッチは $t_{1,R,M}$，$t_{2,R,M}$，$t_{3,R,M}$，$t_{4,R,M}$，$t_{5,R,M}$ と表せる．またメモリ M に格納されたノード 1，2，3，4，5 の出力オペランドをレジスタファイル R にリロードする

マッチは $t_{1,M,R}$，$t_{2,M,R}$，$t_{3,M,R}$，$t_{4,M,R}$，$t_{5,M,R}$ と表せる．

以上の合計 13 個のマッチに対応するブール変数を用意し，それらの集合をコード生成用 FSM の入力変数とする．マッチの名前をそのマッチに対応するブール変数の名前として使用すると，入力変数 w は以下のようになる．

$$w = \{m_{4,\texttt{MUL}}, m_{5,\texttt{ADD}}, m_{5,\texttt{MAC}}, t_{1,R,M}, t_{2,R,M}, t_{3,R,M}, t_{4,R,M}, t_{5,R,M}, t_{1,M,R}, t_{2,M,R}, t_{3,M,R}, t_{4,M,R}, t_{5,M,R}\}$$

このような入力変数を用意することでマッチの並列実行，すなわち並列命令を表すことができる．

次に FSM の状態変数を用意する．まず基本となる状態変数の要素に出力オペランド状態変数がある．出力を持つ各 DFG ノードに対して，ロケーションの個数分，出力オペランド利用可能状態変数を用意する．具体的には，出力オペランド状態変数 $a_{N,L}$ とは，ノード N の出力オペランドがロケーション L に格納されていて利用可能な場合に 1，そうでない場合 0 となるブール変数である．今の例の場合，ロケーションとしてメモリ M およびレジスタファイル R の 2 つがあり，図 1 の DFG では 5 つのノードすべてが出力を持っているため，それに対応して全部で以下の 10 個の出力オペランド状態変数を用意する．

$$x_1 = \{a_{1,M}, a_{1,R}, a_{2,M}, a_{2,R}, a_{3,M}, a_{3,R}, a_{4,M}, a_{4,R}, a_{5,M}, a_{5,R}\}$$

また状態変数の別の要素として，カバー状態変数を用意する．カバー状態変数 c_N は各ノード N ごとに 1 ビットずつ用意され，直前のステップまでにノード N がすでに命令パターンいずれかによってカバーされた場合に 1，それ以外のとき 0 になる変数である．今の例題に対しては次の 5 つを用意する．

$$x_2 = \{c_1, c_2, c_3, c_4, c_5\}$$

ただし，外部入力ノード 1，2，3 にマッチするのは転送命令系のマッチのみなので，それらのノードはカバーされず，c_1，c_2，c_3 はつねに 0 のままとなる．なおこの変数は，本稿で新しく導入されたものである．カバー状態変数を用いることで，使用済みの不要になった出力オペランドをレジスタファイルから解放し，他の必要な出力オペランドを格納できる．またこの変数により，一度カバーされたノードをそれ以降のステップで再びカバーするようなマッチの実行を禁止し，不要な状態遷移を省くことができる．

以上でコード生成用 FSM に必要な各変数を用意した．次に，初期状態から最終状態までの状態遷移を求める．**図 4** に状態遷移の一例を示す．図において，各

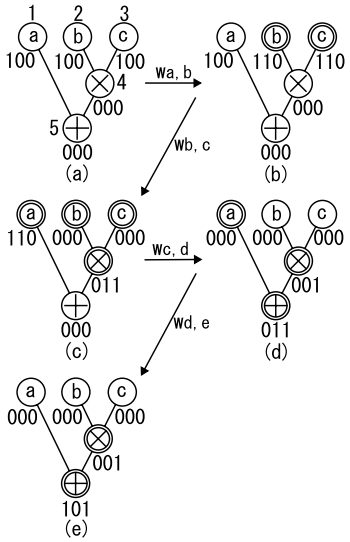


図 4 コード生成用 FSM の状態遷移
Fig. 4 State transitions of FSM for code generation.

ノード N に付けられた 3 つのビットは状態変数であり，それぞれ順に $a_{N,M}$，$a_{N,R}$，c_N を表す．また，いずれかのビットが変化したノードは二重丸にしている．

初期状態は，DFG の各外部入力ノード 1，2，3 の出力オペランドが，メモリ M に格納された状態とする．このとき図 4 (a) のように $a_{1,M}$，$a_{2,M}$，$a_{3,M}$ の値はすべて 1 となる．その他の状態変数はすべて 0 である．一方，最終状態は DFG の外部出力ノード 5 の出力オペランドがメモリ M に格納された状態とする．このとき $a_{5,M}$ の値が 1 となる．図 4 に示した状態遷移は，このように定めた初期状態から最終状態に到達する経路の 1 つである．

図の状態遷移は，FSM に次の入力

$$w = \{m_{4,\texttt{MUL}}, m_{5,\texttt{ADD}}, m_{5,\texttt{MAC}}, t_{1,R,M}, t_{2,R,M}, t_{3,R,M}, t_{4,R,M}, t_{5,R,M}, t_{1,M,R}, t_{2,M,R}, t_{3,M,R}, t_{4,M,R}, t_{5,M,R}\}$$

が加わることによって起こる．特に，状態 (a) から (b)，(b) から (c)，(c) から (d)，(d) から (e) への 4 つの状態遷移は，それぞれ次の入力によって引き起こされたものである．

$$w_{a,b} = \{0,0,0,0,0,0,0,0,0,1,1,0,0\}$$
$$w_{b,c} = \{1,0,0,0,0,0,0,0,0,1,0,0,0,0\}$$
$$w_{c,d} = \{0,1,0,0,0,0,0,0,0,0,0,0,0,0\}$$
$$w_{d,e} = \{0,0,0,0,0,0,0,0,1,0,0,0,0,0\}$$

次に状態遷移の中からいくつか選び，具体的に解説する．まず，状態 (a) から (b) への状態遷移は，入力

$$w_{a,b} = \{0,0,0,0,0,0,0,0,0,0,1,1,0,0\}$$

によって起こる．具体的には転送命令を表す 2 つの

マッチ $t_{2,R,M}$，$t_{3,R,M}$ が並列実行され，メモリ M に格納された外部入力ノード 2，3 の出力オペランドがそれぞれレジスタファイル R にリロードされる．

次に状態 (b) から (c) への状態遷移は，入力

$$w_{b,c} = \{1,0,0,0,0,0,0,0,1,0,0,0,0\}$$

によって起こる．これはノード 1 において転送命令を表すマッチ $t_{1,R,M}$ を実行し，同時にノード 4 において乗算命令を表すマッチ $m_{4,\mathtt{MUL}}$ を実行している．このマッチを実行するには，マッチの入力ノードであるノード 1，2 の出力オペランドがレジスタファイル R に格納されている必要があるが，この条件は，先に実行した命令によって満足されている．なおマッチ $m_{4,\mathtt{MUL}}$ を実行した際，ノード 4 が乗算命令のパターンにカバーされるので，カバー状態ビット c_4（3 ビット目）が 1 となる．このときノード 2，3 のファンアウトノード 4 がカバーされるため，ノード 2，3 の出力オペランドは使用済みとなり（本稿で提案する出力オペランド消滅条件が 1 となり），各ロケーションに格納されたこれらのオペランドは削除され，ノード 2，3 のメモリ M およびレジスタファイル R における出力オペランド状態変数 $a_{2,M}$，$a_{3,M}$，$a_{2,R}$，$a_{3,R}$ は 0 となる．

状態 (e) は，DFG ノードの外部出力ノード 5 の出力オペランドが，所定のロケーションであるメモリ M に格納された状態（最終状態）となっており，この時点で状態探索は終了する．なおメモリ M へ転送した結果出力オペランド消滅条件が 1 となり，レジスタファイル R における出力オペランド状態変数 $a_{5,R}$ は 0 となる．先ほど見たように，この最終状態まで到達させるような入力列が並列コードを表す．

なお，以上の入力列において入力変数は勝手な値をとってよいわけではなく，データ依存やリソース制約が守られていなければならない．今扱っている例題のアーキテクチャでは最大 2 つまでのメモリ転送命令および最大 1 つまでの演算命令が並列実行可能である．たとえば後者の条件は，演算による 3 つマッチを表す変数のうち最大 1 つまで 1 の値をとれる条件として次式で表される．

$$\begin{aligned} LTE_1(m_{4,\mathtt{MUL}}, m_{5,\mathtt{ADD}}, m_{5,\mathtt{MAC}}) = \\ m_{4,\mathtt{MUL}} \cdot \overline{m_{5,\mathtt{ADD}}} \cdot \overline{m_{5,\mathtt{MAC}}} \\ + \overline{m_{4,\mathtt{MUL}}} \cdot m_{5,\mathtt{ADD}} \cdot \overline{m_{5,\mathtt{MAC}}} \\ + \overline{m_{4,\mathtt{MUL}}} \cdot \overline{m_{5,\mathtt{ADD}}} \cdot m_{5,\mathtt{MAC}} \\ + \overline{m_{4,\mathtt{MUL}}} \cdot \overline{m_{5,\mathtt{ADD}}} \cdot \overline{m_{5,\mathtt{MAC}}} \end{aligned}$$

以上に示した入力列は次のような並列コードに相当する．並列コード中 LD はロード命令，ST はストア命令を表す．また，たとえば R(2) は，レジスタファイル R に格納されたノード 2 の出力オペランドを表す．

```
LD  R(2),M(2) || LD  R(3),M(3)
LD  R(1),M(1) || MUL R(4),R(2),R(3)
ADD R(5),R(1),R(4)
ST  M(5),M(5)
```

なおメモリに格納される変数に対しては別途メモリアドレスを割り当てる必要がある．また各ノードの出力オペランドが格納されるレジスタファイル内のレジスタの番号も具体的に割り当てる必要がある．ただし本稿の方法では，コード選択，レジスタアロケーション，スケジューリングを同時に行ったうえでコードを生成しており，生成されたコードは必ずレジスタファイルの容量制約を満たすようになっている．したがってレジスタ番号の割当ては容易に行うことができる．具体的には次のように行う．

コード生成後の各出力オペランド状態変数の変化を見れば，各ノードの出力オペランドがどのステップからどのステップまでレジスタファイルに存在するか分かる．存在期間が重なる変数どうしについては異なるレジスタ番号を適当に選んで割り当て，それ以外の変数については適当なレジスタ番号を選んで割り当てていくという貪欲戦略をとることで，レジスタ番号を割り当てることができる．

以上が本稿の提案する FSM を利用したコード生成の概要である．次章ではどのようにコード生成問題に対する FSM を組み立て，シンボリックに解析するか詳しく説明する．

4. FSM を利用したコード生成問題の定式化

本章ではコード生成用 FSM の作り方およびそのシンボリック状態探索について説明する．まず FSM の入力変数および状態変数を導入する．次に各状態変数に関して，初期状態，最終状態の与え方を示す．また，FSM の状態遷移関数を与える．その後，複数のファンアウトを持つノードを含む DFG は以上のコード生成手法によってどのように扱われるのかという少し立ち入った問題に関して，例を用いて説明する．最後に制約条件の組み立て方を説明し，どのようにシンボリック状態探索に適用するかを示す．特に論文 21)〜23) と異なり，カバー状態変数を利用した条件式 $CONS$（出力オペランド消滅条件）によって，オペランドの寿命を判定し，寿命を終えたオペランドはロケーションから削除する定式化を行う．本章の方法に従って FSM を作り，そのシンボリック状態探索を行えば，コード生成の 3 つのフェーズを同時に行ったうえで最適な並列コードを生成することが可能となる．

4.1 FSM の変数

本節ではまずコード生成用 FSM の入力変数および状態変数について説明する．

4.1.1 入力変数

FSM の入力変数を用意するために，まず DFG の各ノードと命令セットの各命令パターンのマッチングを行う．その結果得られる各マッチごとに，ブール変数を用意する．ブール変数が 1 のとき対応するマッチの実行を表し，0 のときは非実行を表す．特に k ステップ目に FSM に与える入力変数 $w_k = \{w_1, \ldots, w_p\}$ の値によって，k ステップ目で実行するマッチの集合が定まり，その結果対応する並列命令が定まる．

マッチの種類には大きく分けると，DFG ノードをカバーする通常のマッチと，DFG ノードをカバーしない転送命令系のマッチの 2 種類ある．さらに転送命令系のマッチを細かく分類すると，メモリスピル，リロード，レジスタファイル（あるいはメモリ）間転送の 3 種類ある．大きく分類した場合の 2 種類のブール変数を以下に示す．

定義 1（マッチ実行変数）
マッチ実行変数 e_m は，マッチ m を実行開始するとき 1，それ以外のとき 0 となる．

定義 2（転送命令実行変数）
転送命令実行変数 $et_{N,L1,L2}$ は，ロケーション $L1$ に格納された DFG ノード N の出力オペランドをロケーション $L2$ に転送する命令を実行するとき 1，それ以外のとき 0 となる．

なお定義 2 において $L1$ がレジスタファイルで $L2$ がメモリのときスピル，$L1$ がメモリで $L2$ がレジスタファイルのときリロード，$L1$ および $L2$ が異なるレジスタファイルのとき，レジスタファイル間転送，$L1$ および $L2$ が異なるメモリのとき，メモリ間転送となる．

4.1.2 状態変数

FSM の状態変数は，役目に応じて以下の 2 種類のブール変数からなる．

定義 3（出力オペランド状態変数）
出力オペランド状態変数 $a_{N,L}$ は，DFG 中のノード N の有効な出力オペランドがロケーション L に格納されているときに 1，それ以外のとき 0 となる．

定義 4（カバー状態変数）
カバー状態変数 c_N は，DFG 中のノード N がいずれかの命令パターンにカバーされ，そのカバーに対応するマッチが実行されたときに 1，それ以外のとき 0 となる．

4.2 初期状態と最終状態

コード生成用 FSM の状態変数中の各ブール変数について，初期状態と最終状態でどのような値をとるか説明する．

4.2.1 初期状態

初期状態は，DFG の計算開始前の状態を表し，DFG の計算を最後まで進めていくうえで必要な DFG 外部入力ノードの出力オペランドが少なくともどれか 1 つのロケーションで利用可能な状態である．具体的には次のように定める．

(1) 出力オペランド状態変数 $a_{N,L}$ について，DFG の外部入力ノード N の特定のロケーション L で 1，他は 0.
(2) カバー状態変数 c_N はすべて 0.

4.2.2 最終状態

最終状態は，DFG の計算が最後まで終了し，計算結果が所定のロケーションに格納された状態を表す．この状態では，DFG の特定の外部出力ノード N に対応するオペレーションが実行されるか，あるいは DFG の特定の外部出力ノード N の出力オペランドが所定のロケーション L に格納済みとなる．具体的には次のように定める．

(1) 出力オペランド状態変数 $a_{N,L}$ について，特定の外部出力 DFG ノード N の特定のロケーション L で 1，他はドントケア.
(2) カバー状態変数 c_N について，特定の外部出力 DFG ノード N で 1，他はドントケア.

4.3 状態遷移

前節で導入した状態変数中の 2 種類のブール変数の状態遷移関数 $\delta : B^n \times B^p \rightarrow B^n$ を次に示す．状態遷移関数 $y = \delta(x, w)$ とは，x によってエンコードされた現状態に，w によってエンコードされた入力が加わるとどのようにエンコードされた次状態となるか与える関数である．状態遷移関数 δ が求まれば，$T(x, w, y) \equiv y \bar{\oplus} \delta(x, w)$ によってシンボリック状態探索に必要な状態遷移関係 T が得られる．なお本稿では現状態変数 x 中のブール変数 v に対して，次状態変数 y 中の対応するブール変数を v' のようにダッシュをつけて記す．

4.3.1 出力オペランド状態変数の状態遷移関数

出力オペランド状態変数の状態遷移関数では，以下で説明する条件式 $PROD_{N,L}$ および $CONS_{N,L}$ を使用する．$PROD_{N,L}$ を出力オペランド生成条件，$CONS_{N,L}$ を出力オペランド消滅条件と呼ぶ．$PROD_{N,L}$ はロケーション L に DFG ノード N の正しい出力オペランドが格納される条件を表す．ま

た $CONS_{N,L}$ は，ロケーション L に格納されている DFG ノード N の出力オペランドが必要なだけ消費されて不要となるときの条件を表す．以下に 2 つの条件式を示す．ただし条件式中で使用する記号の意味は次のとおりである．ノード N の出力オペランドをロケーション L に格納するようなマッチの集合を $M_{N,L}$ と記す．またノード N をカバーするマッチの集合を M_N，および DFG 中のノード N のファンアウトノードの集合を $FO(N)$ と記す．

$$PROD_{N,L} = \sum_{m \in M_{N,L}} e_m + \sum_{L'} et_{N,L',L}$$

$$CONS_{N,L} = \prod_{K \in FO(N)} (c_K + \sum_{m \in M_K} e_m) + \sum_{L'} et_{N,L,L'}$$

出力オペランド生成条件 $PROD_{N,L}$ の右辺第 1 項は，DFG 中のノード N にマッチした命令のうち，出力オペランドをロケーション L に格納するマッチのいずれかを実行する条件を表す．右辺第 2 項は，DFG ノード N の出力オペランドをロケーション L' からロケーション L に転送する命令のいずれかを実行する条件を表す．以上の説明から分かるように，$PROD_{N,L}$ が 1 のとき，ノード N の出力オペランドがロケーション L に格納され利用可能な状態となる．

一方，出力オペランド消滅条件 $CONS_{N,L}$ の右辺第 1 項は，DFG 中のノード N のすべてのファンアウト K がすでにカバーされているか，あるいは現在のステップでカバーされようとしている条件を表す．右辺第 2 項は，スピル命令あるいはレジスタファイル間転送命令等によりロケーション L に格納されているノード N の出力オペランドをロケーション L' に転送する命令を実行する条件を表す．

ただし，L がメモリで L' がレジスタファイルの場合すなわちリロード命令のマッチに関しては，右辺第 2 項の条件は適用してはならない．適用してしまうと出力オペランド消滅条件が成立して，一度リロードしただけで，まだ保持しておく必要のあるそのメモリ中のオペランドを消滅させてしまい，FSM が最終状態に到達できなくなる可能性がある．

本稿の定式化ではこのように $CONS_{N,L}$ が 1 のとき，ロケーション L に格納されているノード N の出力オペランドはすでに不要となったものとする．このときロケーション L の出力オペランド状態変数を 0 にリセットする．その結果，ノード N の出力オペランドを格納していたロケーション L 内の領域が解放され他のノードの出力オペランドを格納するのに利用できる．

以上の 2 つの条件式を使用して，出力オペランド状態変数 $a_{N,L}$ の状態遷移関数は次式のようになる．なおスペースの都合で，$PROD_{N,L}$ を P，$CONS_{N,L}$ を C と略記してある．

$$a_{N,L}{}' = \begin{cases} 1 & (P = 1 \text{ かつ } C = 0) \\ 0 & (C = 1) \\ a_{N,L} & (\text{それ以外}) \end{cases}$$

4.3.2 カバー状態変数の状態遷移関数

カバー状態変数 c_N は DFG ノード N がカバーされたとき 1 となる．それ以外のときは前の値 c_N を保持する．カバー状態変数の状態遷移関数は次のように表せる．なお式の中で DFG ノード N をカバーするマッチの集合を M_N と記している．

$$c_N{}' = \begin{cases} 1 & (\sum_{m \in M_N} e_m = 1) \\ c_N & (\text{それ以外}) \end{cases}$$

4.4 複数のファンアウトを持つノードの処理

本稿の提案する手法ではコード生成を行う際，与えられた DFG をツリーに分解せず，そのまま DAG として扱う．そのためプログラム中の共通部分式は 1 つのノードにまとめられ，そのようなノードは複数のファンアウトを持つようになる．この節では，DFG の中に複数のファンアウトを持つノードがあるとき，本稿の提案手法がどのようにしてその DFG を処理するか，例題を通して詳しく説明する．

まず，どのように複数のファンアウトを持つノードの出力オペランド利用変数の値を 0 にリセットするか解説する．4.3.1 項でも説明したように，ノード N に関して出力オペランド消滅条件 $CONS$ が成立するとき，その出力オペランドが不要になる．このとき，ノード N の出力オペランド状態変数が 0 となり，ノード N の出力オペランドをすべてのロケーションから削除し，他の出力オペランドがその解放された領域を上書きできるようにする．具体的には，ノード N における出力オペランド状態変数が 0 にリセットされる．次に，この様子を例を使用して説明する．**図 5** に，複数ファンアウトを持つノード 1 を含む DFG を 4 つ示す．な

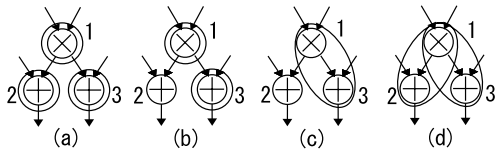


図 5 複数ファンアウトノードにおける出力オペランド消滅条件
Fig. 5 Output operand condition at multiple-fanout node.

おノードがすでにカバーされた場合，すなわちカバー状態ビットがすでに 1 であるか，あるいはノードが現ステップでカバーされようとしているとき，ノードの外側を丸あるいは楕円によって囲んで示す．図 5 (a) では，まずノード 1 がカバーされているものとする．その後のステップで，ノード 2 および 3 が両方ともカバーされたとき，ノード 1 のファンアウトはすべてカバーされた状態となる．このとき，ノード 1 の *CONS* は 1 となり，その出力オペランド状態変数はリセットされる．図 5 (b) および (c) では，ノード 1 のファンアウトのうち，まだ 1 つのファンアウトしかカバーされていないため，ノード 1 の *CONS* は 0 である．このときノード 1 の出力オペランド状態変数は 1 のままである．図 5 (d) は，同じステップ中にノード 1，2，3 すべてを 2 つの `MAC` 命令によってカバーした例である．このとき，ファンアウトであるノード 2 および 3 がカバーされているため，ノード 1 の *CONS* は 1 であり，次のステップでノード 1 の出力オペランド状態変数はリセットされる．

次に，複数のファンアウトを持つノードにおける，カバー間の重なりについて解説する．本稿の手法では，同一ステップにおいて実行されるマッチによるカバーの間の重なりは許される．しかしながら異なるステップに実行されるマッチによるカバー間の重なりは，*CONS* 条件により間接的に禁止される．そのことを，複数のファンアウトを持つノードにおけるカバーに絞って例にそって説明する．**図 6** に，複数ファンアウトを持つノード 3 を含む DFG およびそのカバー（の一部）を 3 つ示した．ただし図 6 においてノード 1 および 2 は，すでに以前のステップにおいてカバーされており，それらの出力オペランドがロケーションに格納されていて利用可能であるものとする．またノード 4 および 5 の入力オペランドのうち，ノード 3 からのもの以外はロケーションに格納されていて利用可能であるものとする．このとき，図 6 (a)，(b)，(c) で示されたマッチが実行可能である．図 6 (a) には 2 つの `MAC` 命令によって，ノード 3 および 4 をカバーするマッチ $M1$ と，ノード 3 および 5 をカバーするマッチ $M2$ が同時に実行されている様子を示す．ただし，このような重なりのある 2 つのマッチ $M1$，$M2$ の実行は，2 つの `MAC` 命令が同時に同じステップで実行されたときにのみ可能となる．仮に，同時ではなくマッチ $M1$ を実行した後のステップに，マッチ $M2$ が実行できるか考えてみる．このときマッチ $M1$ を実行した時点で，ノード 3 がカバーされるため，ノード 1 および 2 それぞれについて *CONS* 条件が成立し，ノード 1 および 2 の値は，すべてのロケーションから消されてしまう．したがって，マッチ $M2$ を実行しようとしても，ノード 1，2 の出力オペランドが利用可能ではないため実行することができない．図 6 (b) および (c) のマッチはそれぞれ実行可能であるが，図 6 (b) に示されているようなノード 5 におけるマッチを実行した後，その後のステップで図 6 (c) で示されているようなノード 3 におけるマッチを実行することは，図 6 (a) の例で説明したのと同様の理由で不可能である．

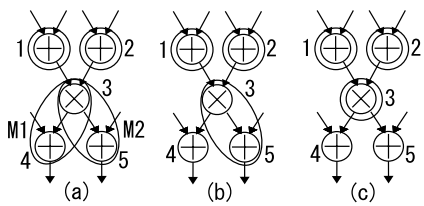


図 6 複数ファンアウトノードにおけるカバー
Fig. 6 Cover over multiple-fanout node.

4.5 各種制約条件

以上でコード生成用 FSM の基本的部分を示した．しかしながら，その FSM の状態探索を行っても正しい値を計算するコードは得られない．なぜなら制約条件が満たされていないからである．

コード生成用 FSM の状態遷移の際には，守られなければならない各種制約条件がある．それらは一般に，入力変数および状態変数の間の条件式で表される．以下にそれらを 1 つずつ説明する．

4.5.1 マッチ実行制約

マッチ m を実行する際には，データ依存が満たされていなければならない．すなわち，実行しようとしているマッチの各入力ノードの出力オペランドが所定のロケーションになければならない．この条件をマッチ実行制約と呼ぶ．マッチ m の入力ノードを $N_{m_1},\ldots,N_{m_k}$ とする．マッチ m を実行するには，これらの入力ノードの出力オペランドがロケーション $L_{m_1},\ldots,L_{m_k}$ に利用可能となっている必要があるものとする．このときマッチ m に関するマッチ実行制約 $E_m(x,w)$ は，マッチ実行変数および入力ノードの出力オペランド状態変数を使用して次のように表される．

$$E_m(x,w) = (e_m \rightarrow a_{N_{m_1},L_{m_1}} \cdot \ldots \cdot a_{N_{m_k},L_{m_k}})$$

同様に DFG ノード N における転送命令実行変数に関するマッチ実行制約は次のようになる．

$$E_{et_{N,L1,L2}}(x,w) = (et_{N,L1,L2} \rightarrow a_{N,L1})$$

以上の条件式では条件が満たされるときに 1 の値をとり，条件違反のとき 0 の値をとる．コード生成用 FSM はこれらの条件を満たすような入力変数の値のみ受け付けるようにする．各 DFG ノード N の各

マッチについて，上に示したマッチ実行制約を立て，それらの論理積をとることで，全体のマッチ実行制約 $E(x,w)$ が得られる．

4.5.2 リソース制約条件

プロセッサ内の利用可能な演算器等のリソースやレジスタの個数はあらかじめ決められている．コード生成では，このようなリソース制約を考慮してコードを生成する必要がある．この節では，演算器，バス，ポート等の通常のリソースに関する制約について説明する．なおレジスタファイルに関するリソース制約は異なる方法で扱うため次の項で説明する．

各クロックサイクルにおいて同時に最大 k 個の命令まで利用可能なリソース r があるとする．また，このリソースを使用するマッチが全部で n 個あるとする．このときリソース r に関しては，それら n 個のマッチのうち最大で k 個まで並列実行可能である．マッチの実行および非実行は FSM の入力変数（マッチ実行変数）の 0，1 の値で表されるから，これら入力変数を使用してリソース制約を表す式を立てることができる．

具体的には次のようになる．リソース r を使用するマッチを表すブール変数の集合を，$\{w_1, w_2, \ldots, w_n\}$ とする．リソース r に関する制約が満たされるのは，これら n 個のブール変数のうち最大で k 個のブール変数の値が 1 となる場合である．すなわち，リソース制約は次式のように表せる．

$$P_r(w) = LTE_k(w_1, w_2, \ldots, w_n) \qquad (2)$$

ここで $P_r(w)$ の値は，入力値 w が制約条件を満たすときに 1，条件違反のときには 0 となる．各リソース r ごとの制約式 $P_r(w)$ の論理積をとることで，全リソース制約 $P(w)$ が求まる．

なお制約式 (2) の論理式を積和形で表すと ${}_nC_k$ の数の積項が必要な巨大な式になってしまう．そこで論文 20) で提案されたリソース制約の表現方法を利用し，制約を直接 BDD で簡潔に表現する．この場合，制約式をノード数が nk 程度の BDD で表現できる．

4.5.3 レジスタファイル制約条件

次にレジスタファイル制約の取扱いについて説明する．レジスタファイル制約とは，レジスタファイル L 内の利用可能なレジスタ数（レジスタファイルの容量）が最大で k 個までという制約である．前項に示したように一般のリソース制約は入力変数 w の論理式で表したが，レジスタファイル制約については，出力オペランド状態変数 $a_{N,L}$ に関する論理式となる．

具体的には次のようにして制約式を立てる．レジスタファイル L のレジスタの総数を k，レジスタファイル L に関する出力オペランド状態変数 $a_{N,L}$ の総数を n とする．このとき n 個の $a_{N,L}$ のうち最大 k 個まで 1 の値をとることができる．すなわちレジスタファイル制約は次式で表現される．

$$Q_L(x) = LTE_k(a_{N_1,L}, \ldots, a_{N_n,L})$$

ここで $Q_L(x)$ は，状態変数 x（出力オペランド状態変数）が制約条件を満たすときに 1，条件違反のときには 0 の値をとる．レジスタファイルが複数ある場合，各レジスタファイル L ごとに条件式 $Q_L(x)$ を立てそれらの論理積をとることで，全レジスタファイル制約 $Q(x)$ が求まる．

4.6 制約条件のシンボリック状態探索への適用

この節では，以上で説明した各制約条件をどのようにシンボリック状態探索に取り込んで，制約を満足する次状態を計算するか説明する．

まずマッチ実行制約 $E(x,w)$ およびリソース制約 $P(w)$ は次式のように状態遷移関係 $T(x,w,y)$ との積をとることで，像計算の式 (1) に取り込む．

$$R_{k+1}(y) = \exists x, w((T(x,w,y) \cdot E(x,w) \cdot P(w)) \cdot R_k(x))$$

一方レジスタファイル制約の像計算への適用方法は，上で示した方法とは異なり，次の方法をとる．まず上で示したマッチ実行制約 $E(x,w)$ およびリソース制約 $P(w)$ を考慮した像計算の式によって現状態 $R_k(x)$ から次状態 $R_{k+1}(x)$ を計算する．この後，前節で導入したレジスタファイル制約式 $Q(x)$ と $R_{k+1}(x)$ の論理積をとることで，レジスタファイル制約を満たした次状態 $R_{k+1}(x)'$ を次のように計算する．

$$R_{k+1}(x)' = Q(x) \cdot R_{k+1}(x)$$

5. アルゴリズムの高速化

前章では，コード生成用 FSM の作り方を示した．その FSM をシンボリックに状態探索することで，コード生成 3 つのフェーズを同時に行ったうえでステップ数最適な並列コードを生成することができる．ただしシンボリック状態探索では，扱う BDD のサイズが大きくなりすぎると処理が終了しなくなる欠点がある．この章では，コード生成用 FSM のシンボリック状態探索を効率的に行う方法をいくつか提案する．それらを使用することでコード生成の時間を劇的に短縮することが可能となる．コード生成用 FSM のシンボリック状態探索を効率化する際，以下の 2 通りのアプローチが考えられる．

- 明らかに最適解に寄与しないような無駄な状態および状態遷移を削減する．
- 最適性を多少犠牲にしても短時間でコード生成が

終了するように状態および状態遷移を削減する．

前者ではもちろん最適解が保証される．後者はヒューリスティックであり最適性は保証されないが，できるだけ最適解に近いコードを生成する方法を考える必要がある．以上のいずれかのアプローチのもとで，状態および状態遷移をどのように削減するかについては，以下の方法が考えられる．

- 状態遷移関係 $T(x,w,y)$ の BDD を簡単化する．
- 状態集合 $R(x)$ の BDD を簡単化する．

以下に 2 つの方法をそれぞれ示す．

5.1 状態遷移関係の簡単化

状態遷移関係 $T(x,w,y)$ の BDD を簡単化するには，$T(x,w,y)$ の BDD 自身を何らかの方法で簡単化する以外に，ブール関数 $H(x,w)$ をうまく見つけて $T(x,w,y)$ と積をとり，それを像計算の式の中で使用する方法があげられる．このとき $T(x,w,y)\cdot H(x,w)$ を表す BDD がもとの状態遷移関係 $T(x,w,y)$ をそのまま用いた場合より簡単になれば，計算時間を短縮することができる．$H(x,w)$ を与える方法として，たとえば以下の項目が考えられる．

(1) 以前にすでにカバーされたノードはカバーしない．
(2) メモリにすでに値がある場合は，スピルしない．
(3) リロードの回数はファンアウトの数までとする．

ここでは例として，一番最初の項目について条件式 $H(x,w)$ を立てる．前のステップですでにカバーされた DFG ノード N は，カバー状態ビットが 1 となっている．このようなノード N をカバーするマッチを実行しなければよい．この条件は，以下のように表される．ただし，M_N は，ノード N をカバーするマッチの集合を表す．

$$H(x,w)=\prod_{N}(c_N \rightarrow \prod_{m\in M_N}\overline{e_m})$$

5.2 状態の簡単化

この節では状態集合 $R(x)$ の中から不要な状態を削除し，$R(x)$ を表す BDD を簡単化する方法を示す．k ステップ目の状態 $R_k(x)$ に対して，次のようなブール関数 $G_k(x)$ を導入して簡単化された状態 $R_k(x)'$ を計算する．

$$R_k(x)' = G_k(x)\cdot R_k(x)$$

$G_k(x)$ を与える方法としては，ステップ k とそれ以降で考える必要のない状態を見つけ，そのような状態を削除するような条件式を求める．

この方法の一例として，論文 3)，23) に提案されているように，状態集合 $R_k(x)$ の中からできるだけ多くのノードがカバーされた状態のみを残して，それ以外の状態を捨てる方法が考えられる．論文 23) で提案された方法では，まず状態集合 $R_k(x)$ の中で最大いくつのノードがカバーされているか調べ，その最大値 M を求める．そしてカバーされているノード数が最大値 M から $M-k$（ただし k は適当な整数）の間にある状態のみを残すような $G_k(x)$ を用意し，$R_k(x)'$ を計算する．このようにして状態を削減した場合，必ずしも最適解を生成するとは限らないが，コード生成の時間を大幅に短縮し，たいていの場合最適解に近い解を出力することが期待できる．

なお以上で説明した方法とは異なる高速化のアプローチも考えられる．たとえば，検証の分野で開発されているシンボリック状態探索に対する様々なヒューリスティック[15),16)] を利用する方法が考えられる．なお本稿の実験では，論文 15) のヒューリスティックを使用した．

6. 実装および実験結果

本稿で提案したコード生成手法を実装し，実験を行った．プログラムは VIS[25)] 上に実装した．VIS はカリフォルニア大学，コロラド大学を中心に開発された FSM 合成および検証用システムである．また実験には，Pentium3（850 MHz)-Linux および 256 M バイトのメモリを積んだマシンを使用した．

ベンチマークプログラムには，DSPStone[26)] を使用した．使用したプログラムの概要を**表 1** に示す．表中のノード数，エッジ数および *cse* は，それぞれ DFG 中のノード数，エッジ数および共通部分式の数を表す．

コード生成対象のアーキテクチャとしては，**図 7** に示すようなクラスタ型 VLIW アーキテクチャを対象とした．クラスタ型 VLIW アーキテクチャでは，各

表 1 使用した DSPStone ベンチマークプログラム
Table 1 DSPStone benchmark programs.

プログラム名	ノード数	エッジ数	*cse*
cm	10	18	4
iir	17	22	3
cu	14	18	4

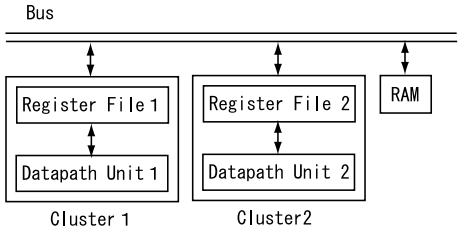


図 7 ターゲットアーキテクチャ
Fig. 7 Target architecture.

表 2 ターゲットアーキテクチャのバリエーション
Table 2 Various target architectures.

	S1	R1	DP1
A	1	4	ALU, MULT
B	4	16	ALU × 2, MULT × 2
C	4	16	ALU × 2, MULT × 2, MAC × 2
D	2	16	ALU, MULT
E	2	16	ALU, MULT

	S2	R2	DP2	BUS	PORT
A	—	—	—	2	R1W1
B	—	—	—	4	R4W2
C	—	—	—	4	R4W2
D	2	16	2ALU × 2	4	R4W2
E	2	16	MAC × 2	4	R4W2

```
w  = x - a1 * w1 ;
w -= a2 * w2 ;
y  = b0 * w ;
y += b1 * w1 ;
y += b2 * w2 ;

w2 = w1 ;
w1 = w  ;
```

(a)

```
1: LD M,R || LD M,R || LD M,R || LD M,R || LD M,R
   || LD M,R
2: LD M,R || LD M,R || MUL R,R,R || MUL R,R,R
3: MUL R,R,R || MUL R,R,R || ST R,M || SUB R,R,R
4: SUB R,R,R
5: MAC R,R,R,R || ST R,M
6: ADD R,R,R
7: ST R,M
```

(b)

図 8 生成されたコードの例
Fig. 8 Example code generated by our method.

クラスタがレジスタファイルおよび複数の演算器からなる演算ユニットを持つ．

表 2 に，扱うアーキテクチャのバリエーションを示す．なお表中で，S1，R1，DP1 はそれぞれクラスタ 1 のスロット数，レジスタファイルの容量，演算器の種類とその数を表す．同様に，S2，R2，DP2 はそれぞれクラスタ 2 のスロット数，レジスタファイルの容量，演算器の種類とその数を表す．各クラスタでは，最大でそのスロット数までの命令を並列実行できる．また表において，BUS はバスの本数，PORT はメモリの READ および WRITE のポート数を表す．また DP2 中の 2ALU は，3 入力 ADD や ADDSUB 等を実行できる演算器である．

図 8 に出力されたコード例（プログラムとして `iir` を使用）を示す．使用したベンチマークプログラム中の共通部分式は 1 つのノードにまとめ，DFG として構成しなおしたものを，コード生成プログラムに入力した．また生成されたコードとしては，レジスタ番号はまだ付けていない段階のものを示している．

表 3 アーキテクチャA を対象としたコード生成結果
Table 3 Code generation result for architecture A.

	変数数	ステップ数	探索状態数	時間
cm	62	9	7.6×10^3	1.2
iir	98	12	2.1×10^4	2.5
cu	80	10	1.6×10^4	7.0

表 4 アーキテクチャB を対象としたコード生成結果
Table 4 Code generation result for architecture B.

	変数数	ステップ数	探索状態数	時間
cm	62	5	1.1×10^4	0.7
iir	98	8	2.7×10^4	1.1
cu	80	5	1.7×10^4	1.1

表 5 アーキテクチャC を対象としたコード生成結果
Table 5 Code generation result for architecture C.

	変数数	ステップ数	探索状態数	時間
cm	64	4	6.7×10^3	0.5
iir	99	7	2.6×10^4	1.9
cu	82	5	1.8×10^4	2.5

表 6 アーキテクチャD を対象としたコード生成結果
Table 6 Code generation result for architecture D.

	変数数	ステップ数	探索状態数	時間
cm	92	7	7.7×10^5	1.6
iir	151	8	2.0×10^6	9.9
cu	124	7	1.0×10^6	6.4

表 7 アーキテクチャE を対象としたコード生成結果
Table 7 Code generation result for architecture E.

	変数数	ステップ数	探索状態数	時間
cm	94	6	6.4×10^5	1.6
iir	150	8	2.0×10^6	4.5
cu	124	7	8.8×10^5	11.1

表 3，**4**，**5**，**6**，**7** に，アーキテクチャA，B，C，D，E に対するコード生成の結果をまとめた．なお，表中の変数数はコード生成用 FSM の入力変数および状態変数の合計を表す．ステップ数は，生成されたコードのステップ数を表す．探索状態数は，本稿の手法によって探索された状態数の合計を表す．表中の右端の時間は，コード生成に要した合計 CPU 時間（単位は秒）を表す．

以上のコード生成結果を見ると，短時間に非常に多くの状態を探索していることが分かる．特に探索状態数の増加に比べて，CPU 時間の増加は低く抑えられている．また以上の実験ではヒューリスティックは使用していない．したがってオペランドの再計算を行わない条件のもとで，ステップ数最小のコードを生成し

た結果を示してある．参考までに論文 3) の方法では，各ステップ 50 状態までしか探索していないため，10 ステップの場合でも 500 状態しか探索していない．

7. 結　　論

本稿では，論文 2)，3) と同様のアプローチに基づき，FSM を利用してコード生成を行う手法を提案した．ただし論文 2)，3) とは異なり，FSM の状態探索方法としてシンボリック状態探索を適用するような定式化を行った．論文 21)〜23) も同様にシンボリックな状態探索を利用しているが，ノードの再計算を自由に行えるようにしているためにいつオペランドが不要になるか判別が難しく，その方法が示されていない．論文[21)〜23)] とは異なり，本稿では DFG 中のノードのファンアウトすべてがカバーされたとき，そのノードの出力オペランドをロケーションから消滅させるような状態遷移を考えた．その結果本稿の手法ではノードの再計算できないが，DAG が扱え，さらに並列命令のカバーするノードの重なりも許されるため，制限はそれほど大きくないと考えられる．

以上の新しい定式化の有効性を実験的に確かめるため，プロトタイププログラムを作成し，いくつかのベンチマークプログラムに対して実際にコード生成を行った．その際，論文 22)，23) では扱われていなかったレジスタに格納された値の RAM へのスピルおよびリロードも考慮した．実験の結果，シンボリック状態探索を使用することで，非常に多くの状態を短期間に探索可能であることが分かった．具体的には，現実的な命令セットを使用した場合，ノード数が 10〜20 程度の DFG に対しては，数秒から数十秒程度でコード生成のすべてのフェーズを同時に行ったうえでの，ステップ数最適なコードが生成できることが分かった．本稿で提案したコード生成手法は，ループ本体中の，サイズは小さいが実行回数が多い基本ブロックに適用すると効果的であると考えられる．

本稿の手法の問題点および今後の課題としては以下の項目があげられる．まず大きな問題点として，DFG のサイズが大きい場合や命令セットが複雑な場合に，急激に CPU 時間が増大してしまう点がある．この問題を解決するため，コード生成用 FSM に特化したより効果的なシンボリック状態探索のヒューリスティックを開発する必要がある．またその他の課題として，本稿の手法の適用範囲を広げるため，SIMD 命令へ対応したコード生成手法の考案すること，また生成されたコードのパフォーマンスをさらに改善するためソフトウェアパイプラインを適用したコードを生成手法の考案することがあげられる．またステップ数が小さいだけでなく，コード量についてもできるだけ小さな解を見つける方法を考案することがあげられる．さらに，単に与えられた命令に対するコードを生成するだけではなく，アプリケーションに特化した専用命令を自動的に生成することがあげられる．

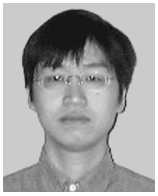
謝辞　パシフィック・デザイン株式会社の下郡慎太郎氏および査読者の方々には多くの有益なコメントをいただきました．ここに感謝いたします．

参 考 文 献

1) Aho, V.A., Sethi, R. and Ullman, J.D: *Compilers Princeples, Techniques, and Tools*, Addison-Wesley (1988).
2) Roemer, A. and Fettweis, G.: Flow Graph based parallel Code Generation, *4th International Workshop on Software and Compilers for Embedded Systems* (*SCOPES 1999*) (1999).
3) Roemer, A. and Fettweis, G.: Optimierte parallele Code-Erzeugung, *DSP Deutschland* (2000).
4) Paulin, P.: Design Automation Technology Challenges for Application-Specific Architecture Platforms, *Keynote speech, 5th International Workshop on Software and Compilers for Embedded Systems* (*SCOPES 2001*) (2001).
5) Texas Instruments: *TMS320C6000 CPU and Instruction Set Reference Guide* (Oct. 2000).
6) Phillips Semiconductors: *TriMedia TM-1300 Media Processor Databook* (Sep. 2000).
7) MOTOROLA, agere systems: *SC140 DSP Core Reference Manual* (2001).
8) http://www.bops.com/
9) http://www.improvsys.com/
10) http://www.chameleonsystems.com/
11) Coudert, O., Berthet, C. and Madre, J.C.: A unified framework for the formal verification of sequential circuits, *Proc. IEEE Int. Conf. Computer Aided Design*, pp.126–129 (Nov. 1990).
12) Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L. and Hwang, L.J.: Symbolic Model Checking: 10^{20} states and beyond, *Information and Computation*, Vol.98, No.2, pp.142–170 (June 1992).
13) Akers, S.B.: Binary decision diagrams, *IEEE Trans. Comput.*, Vol.C-27, No.6, pp.509–516 (1978).
14) Bryant, R.: Graph-based algorithms for boolean function manipulation, *IEEE Trans. Comput.*, Vol.C-35, No.8, pp.677–691 (1986).
15) Ranjan, R.K., Aziz, A., Brayton, R.K., Plessier, B. and Pixley, C.: Efficient BDD al-

gorithms for FSM synthesis and verification, *Proc. IEEE/ACM International Workshop on Logic Synthesis* (May 1995).
16) Bloem, R., Moon, I.-H., Ravi, K. and Somenzi, F.: Approximations for Fixpoint Computations in Symbolic Model Checking, *Systemics, Cybernetics and Informatics*, Orlando, FL, pp.23–26 (July 2000).
17) Araujo, G. and Malik, S.: Optimal code generation for embedded memory non-homogeneous register architectures, *Proc. 1995 Intl. Symp. on System Synthesis*, pp.36–41 (1995).
18) Liem, C., May, T. and Paulin, P.: Instruction-Set Matching and Selection for DSP and ASIP Code Generation, *EDAC-ETC-EUROASIC 1994*, pp.31–37 (1994).
19) Leupers, R. and Bashford, S.: Graph based Code Selection Techniques for Embedded Processors, *ACM TODAES*, Vol.5, No.4 (2000).
20) Radivojevic, I. and Brewer, F.: On Applicability of Symbolic Techniques to Larger Scheduling Problems, *Proc. European Design and Test Conf.*, pp.48–53 (1995).
21) Monahan, C. and Brewer, F.: Symbolic Modeling and Evaluation of Data Paths, *32nd Design Automation Conference Proceedings*, pp.389–394 (1995).
22) Monahan, C. and Brewer, F.: Scheduling and Binding Bounds for RT-Level Symbolic Execution, *Proc. IEEE Int. Conf. Computer-Aided Design*, pp.230–235 (1997).
23) Monahan, C.: Symbolic Data Path Analysis, Ph.D thesis, Department of Electrical and Computer Engineering, University Of California, Santa Barbara (1997).
24) Haynal, S. and Brewer, F.: Efficient Encoding for Exact Symbolic Automata-Based Scheduling, *IEEE Int. Conf. Computer-Aided Design*, pp.477–481 (1998).
25) Brayton, R.K., et al.: VIS: A system for verification and synthesis, *8th Conference on Computer Aided Verification* (*CAV '96*), Henzinger, T. and Alur, R. (Eds.), pp.428–432. Springer-Verlag, Rutgers University, LNCS 1102 (1996).
26) Zivojnovic, V., Velarde, J.M., Schlager, C. and Meyr, H.: DSPStone—A DSP-oriented Benchmarking Methodology, *Int. Conf. on Signal Processing Applications and Technology* (*ICSPAT*) (1994).

(平成 13 年 9 月 21 日受付)
(平成 14 年 3 月 14 日採録)



瀬戸 謙修

1997 年東京大学工学部電気工学科卒業．1997 年〜1998 年カリフォルニア大学バークレイ校 CAD グループ交換留学生．1999 年東京大学工学系研究科電子工学専攻修士課程修了．現在パシフィック・デザイン株式会社勤務．主に論理合成，アプリケーション特化型プロセッサ向けコンパイラの研究に従事．

藤田 昌宏（正会員）

1985 年東京大学大学院工学系研究科情報工学博士課程修了．工学博士．同年富士通入社．富士通研究所にて，VLSI CAD の研究に従事．1988 年〜1989 年イリノイ大学客員研究員．1993 年米国富士通研究所出向．VLSI CAD 研究グループの立ち上げ．2000 年より東京大学大学院工学系研究科電子工学専攻教授．論理合成，論理検証，システムレベル設計支援技術等の研究に従事．SpecC コンソーシアム言語ワーキンググループ主査．IEEE，ACM 会員．