北海道大学工学部研究報告
第133号（昭和61年）

Bulletin of the Faculty of Engineering.
Hokkaido University, No. 133 (1986)

アーキテクチャの知識を用いた マイクロプログラム開発システム

三 谷 和 史* 宮 本 衛 市*

（昭和61年6月30日受理）

A Support System for Microprogramming based on Architecturel Knowledge

Kazufumi MITANI and Eiichi MIYAMOTO

(Received June 30, 1986)

Abstract

This paper describes a support system for microprogramming which requires programmers to master the sophisticated structure of the micro-architecture and aquire skilled knowledges to use it. The system aims to give support to programmers in writing microprograms without special skills regardeing micro-architecture and microprogramming.

Under the support of the system, the programmer writes microprograms in the high-level language, and the system compiles the micro-instructions. The parsing phase of the system is machine independent and is processed based on typical compiling techniques, while the code generation phase is processed based on architectural knowladges which are represented by three kinds of predicated in PROLOG ; "type", "path" and "function". Each means the class of architecture units, interconnections between the units and functions of units respectively.

This system can also be applied to architecture design, because the system is adapted to any architecture by replacement of the knowledges.

1. はじめに

マイクロプログラミングに関する研究は1970年代に精力的に行われ，それらの成果が現在利用されている[1,8,9,23]。しかし，ハードウェアとソフトウェアの接点を担うハードウェアのゲートレベルからマイクロプログラムのレベルまでの統一的記述法が確立されておらず[3,18,19]，まだ研究すべき課題は多い。

近年，ユーザ・マイクロプログラム可能なDSP (Digital Signal Processor)が急速に発展・実用化してきており，これまで一般のプロセッサで処理がなされていたものを専用のDSPを用いて高速化を図ることが行なわれている。そこで，これからは一般のユーザのための，マイクロプログラム作成ツールの完備，普及が必要となる。本研究では，特に高速化を要求されるＦＦ

* 情報工学専攻 情報システム工学講座

Tやフィルタリング処理等のソフトウェアルーチンをマイクロプログラム化する場合を想定し，その作成を支援するシステムの開発を目的としている。

DSPのマイクロプログラム開発においては，マシンアーキテクチャを隅々まで熟知していなければ，良いマイクロプログラム，即ちマイクロ命令のダイナミックステップが短かいマイクロプログラムが書けないため，これまではマシンを設計する側の人間が行う場合が多かった。

マイクロプログラムの生産性を向上させるためには，マシンアーキテクチャに熟知していなくてもマイクロプログラムが作成できる環境を整備して，実際のアプリケーションをマシン上で実行させたい側の人間がマイクロプログラムの作成に当らなくてはならない。このようなシステムでは，以下の点が考慮されていなければならない。

1)　少なくともソース言語は高級言語レベルのものであること。開発用のホストマシン上で走る高級言語とソースを共有できるならばなおよい。

2)　マシンアーキテクチャを外部から入力できるようにし，汎用性を持たせる。

3)　熟練したマイクロプログラマと同程度のオブジェクトマイクロコードを生成する能力を持ったコンパイラを有する。

特に2)に関しては，マシンアーキテクチャを外付けにすることにより汎用性が得られるばかりでなく，出力されたマイクロプログラムを検討する事によって，アーキテクチャの仕様を変更した場合に，どれほどの性能の向上が見込めるかを決定する手掛りとする事ができる。又，アーキテクチャの記述方式が決定されると，ハードウェアの設計時にもこれらの情報が活用できる。アーキテクチャは，内部のハードウェア・ユニットと，その間の繋がりという構造的な情報と，ユニットに制御信号を与えればそのユニットの入力と出力の間にどのような関係が生ずるかという情報を持っている。又，この情報を用いてアーキテクチャの検証を行なう事もできる[7]。本研究では，アーキテクチャに関するこれらの知識を利用する事によって，マイクロコードを生成する方法の実現を目的としている。

2.　これまでのマイクロプログラム作成技法

2.1　ソース言語レベルでの分類

マイクロプログラム作成言語の設計においては，オブジェクト・マイクロプログラムの実行効率を考慮したうえで，その仕様を決定する必要がある。この場合，「記述のレベル」，「機械独立性」，「オブジェクト・マイクロプログラムの実行効率」の3点が考慮すべき重要な要素となる。「記述のレベル」と「機械独立性」の2つの組合せで，一般にマイクロプログラム作成言語は **MDLL** (Machine Dependent Low-level Language), **MILL** (Machine Independent Low-level Language), **MDHL** (Machine Dependent High-level Language)及び **MIHL** (Machine Independent High-level Language)の4つに大別される。

MDLLは特定のマシンに依存したアセンブラ言語であり，そのハードウェアを作成したところで一緒に作成され，そのハードウェアのためのマイクロプログラムの作成に一般に用いられる。MILLはハードウェアによらないアセンブラ言語の仕様を定めて，それを用いてマイクロプログラムを作成するためのものである。しかし，アーキテクチャの可変性が大きすぎるので，そのすべてを言語仕様に取り入れることは不可能であるため，比較的似たアーキテクチャやシリーズ化されたアーキクチャについて，MILLが用いられる。

MDHLは特定のマシンのための高級言語であり，PL/IやALGOL風のブロック構造をもっ

たものが多い。又，MIHL の試みとしては，トランスレータ記述システム，機能拡張可能なシステム，計算機アーキテクチャの記述とマイクロプログラムを組み合せたもの，機械依存部分と機械独立部分を分離させたもの等の方式が考えられている[28,29]。

2.2　最適化に関する方法

マイクロプログラムの最適化[30]は，大きく分けて2つの方向がある。第1の方向は，マシンの設計時に「最小の制御記憶量で最大の実行速度を得る」ために，マイクロ命令の語長を短かくする事である[27]。第2の方向は，既に設計されたマシンのマイクロプログラムを開発する場合に，マイクロプログラム中の命令語数を減らす事(即ち計算速度を上げる事)である。ここでは，第2の方向についてのみ考えることにする。

通常のコンパイラで行なっている最適化は，マイクロプログラムの場合でも当然考慮する必要がある。さらに，マイクロプログラムでは高度の並列動作が可能であるため，並列実行可能なマイクロオペレーションのデータ依存性を守り，かつマイクロアーキテクチャの資源(データ・バス，ALU，シフタ等)の使用の衝突が起こらずに，さらには与えられたマシンの性能をフルに活用する形でマイクロオペレーションをマイクロ命令中に詰め込むという最適化が必要である。

水平型のマイクロプログラムの最適化における，入口1つ出口1つの SLM (Straight Line Microprogram)と呼ばれるマイクロオペレーション列に対する最適化では，SLM 中の最小性はいえるが，全体の語数の最小性の保証はない[4,13,22,26]。又，グローバルな最適化に関しては，データフロー解析を用いた方法がある[16,20,21,24,25]。他には，機械独立なレベルでの最適化と，機械固有の部分に対する最適化を分ける方式をとるものもある。

Patterson[15]は，実際のマイクロプログラムで SLM となる部分は大変短かく，そこだけを相手にしても効果はあまりないと述べている。そして，STRUM[14]ではピープホール・オプティミィゼイションを取り入れているが，1,000個や2,000個のマイクロ命令を相手にすると，人手によるものより良いコードが得られると述べている。

2.3　知識を用いたアプローチ

TI990用のマイクロプログラム作成用エキスパートシステムである MIXER[5]では，マイクロプログラミングの知識とは，マイクロプログラム上の仮想リソース，ファンクションをマイクロアーキテクチャにマッピングするための知識であると考えた。そして TI990の制御記憶(μROM)から抽出した知識を基に知識ベースを作成し，これを用いて PASCAL の P コード・インタプリタのエミュレータを作成した。更に，この知識ベースを一般化し，知識を構文タイプと意味タイプに分類して，一般のマイクロプログラムの作成にも拡張している[6]。

これまでの一般的なアセンブラ・コンパイラによる方法以外に，エキスパートシステムの面からマイクロプログラムの作成を捉えたという点で，この MIXER のアプローチは重要である。

3.　本研究で提案する方法

3.1　基本構想

本システムは，高級言語レベルのソースプログラム(C の関数程度のプログラム)を水平型マイクロ命令列に展開する事を想定している[11]。システム全体の構成を図1に示す。

本システムは2．1で述べた分類上，計算機アーキテクチャの記述とマイクロプログラムを組み合わせたものであり，機械依存部分と機械独立部分を分離させた方式である。又，機械依存部分に対しては知識を用いたアプローチを用いている。アーキテクチャの知識は外付けであり，この部分を取換えることにより様々なアーキテクチャに対応できるように考えている。この知識は2種類あり，1つはエキスパートによるマイクロプログラムに関する知識であり，もう1つはアーキテクチャの設計段階で得られるものである。

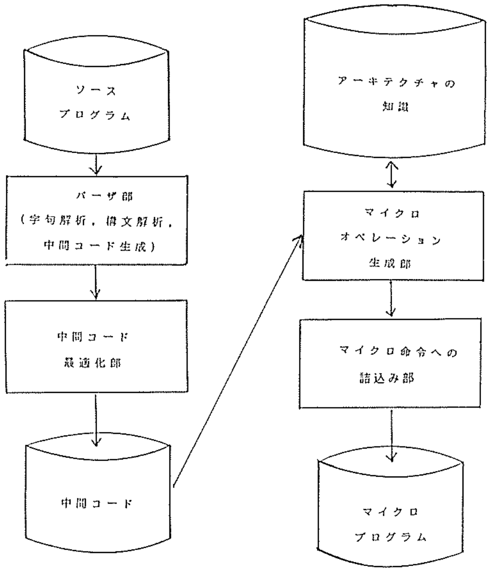

図1　システムの構成

3．2　表層知識と深層知識

ある問題に対する答えをあらかじめすべて用意しておくことは不可能であって，一般には問題をより深いレベル又は一般化したレベルで考え，そこから答えを導きだす方法がとられる。この深いレベルで考える場合の基礎となる事実の集りが深層知識である。例えば，〝2×3〃という問題に対して，“2×3＝6”という答えをあらかじめ知識として持っていれば，その知識は表層知識である。これに対し，乗算A×BはAをB回加えたものであるという知識で問題を解決しようとする場合，乗算は加算で計算できるという知識が乗算に対する深層知識である。

マイクロコードの生成では，中間コードをマイクロオペレーションの列に変換することが解決すべき問題であるから，中間コードに対応するマイクロオペレーション列のレベルが表層知識である。又，次に述べる，アーキテクチャを機能ブロックとそれらを結ぶパスにより表現したものが，マイクロプログラムの作成における深層知識である。

3．3　計算機アーキテクチャの知識表現と中間コード

知識を用いることの利点としては，

1)　人間にとっての記述のしやすさ。
2)　人間にとっての理解のしやすさ。
3)　知識自体の他の問題への転用が容易である。

等のことがあげられる。

本システムではアーキテクチャの記述表現に図2のような方法を用いた。これは，アーキテクチャを機能ブロックとそれらを結ぶパスにより表現したものであり[7]，Prologの節形式によって表現している。例えば，

　　type (register, inrx, [[in], [out]]).

は，レジスタinrxが入力ポートinと出力ポートoutをもつことを意味する。

　　path ([inrx, out], [mplx _2, 1], 127).

は，inrx の出力ポート out からマルチプレクサ mplx _2の入力ポート 1 へ127という結線があることを意味する。

function (add, [[alu, c], [[alu, a], [alu, b]], [[alu, add]], [tl]).

は，ALU に add の制御信号を送ることで ALU の入力ポート a, b の和が ALU の出力ポート c に出力され，そのためには t1の時間がかかることを意味する。ターゲットマシンとして想定した μDSP を図 2 の表現形式で表したものの一部を図 3 に示す。

中間コードは，ソースプログラムの持つ情報を，パーザの解析した式木 (Expression Tree) を表すリスト構造によって表現している。例えば，“A＝B+C”を中間コードで表すと，

[set, A, [add, B, C]]

となる。

```
Machine Architecture Knowledge ::= (type_declaration}
                                   (path_declaration}
                                   (function_declaration} ;
type_declaration ::= type(type_name,unit_name,
                          [input_port_list,output_port_list]). ;
path_declaration ::= path(source,destination,path_name). ;
function_declaration ::= function(function_name,[destination],
                                   [source,(source)],μop,timing).
                       | function("set",[unit_name],[source],
                                   μop,timing). ;
input_port_list ::= input_port{,input_port};
output_port_list ::= output_port{,output_port};
destination ::= [unit_name,output_port] ;
source ::= [unit_name,input_port] ;
type_name ::= "register" | "bus" | "multiplexer" | "alu" | "sifter"
            | "memory" | "multiplyer" | "gate"  etc. ;
function_name ::= "add" | "sub" | "mul" | "div" | "and" | "or"
                | "not" | "xor" | "lsft" | "rsft" | "neg" | "bcc"
                | "conn"| "set"   etc. ;
```

図 2　マシン・アーキテクチャ知識の記述形式

```
type(register,inrx, [[in],[out]]).
type(register,inry, [[in],[out]]).
type(register,mlr,  [[in],[out]]).
type(gate,gate_1, [[in],[out]]).
type(gate,gate_2, [[in],[out]]).
                   :
                   :
path([lit(X),out],        [gate_1,in],     l1).
path([gate_1,out],        [e_bus,in],      l2).
path([e_bus,out],         [cfmar,in],      l3).
                   :
                   :
function(set,   [inrx], [[inrx,in]], [[inrx,set]], [tx]).
function(and,   [[alu,c]],[[alu,a],[alu,b]],[[alu,and]],[t1]).
function(or,    [[alu,c]],[[alu,a],[alu,b]],[[alu,or]], [t1]).
function(add,   [[alu,c]],[[alu,a],[alu,b]],[[alu,add]],[t1]).
function(sub,   [[alu,c]],[[alu,a],[alu,b]],[[alu,sub]],[t1]).
function(conn,  [[alu,c]],[[alu,a]],[[alu,through_a]],  [t1]).
                   :
                   :
```

図 3 ターゲットマシンの記述（一部）

この中間コードによって，B と C を足して A に代入することが表現されている。又，アーキテクチャの記述中には、述語 function (add…) によって ALU は足すことができ，function (set …)によってメモリやレジスタなどは代入することができるという知識が表現されている。

本システムでは，これら **set** や **add** といったファンクションを，アーキテクチャの知識から情報を引き出すために用いている。又，アーキテクチャの知識のうちタイミングに関する部分は，マイクロオペレーションをマイクロ命令に埋めこむ場合に用いられる。

3．4　コード生成のための戦略的知識

マイクロコードの生成の場合は，中間コード上のファンクションを実行するマイクロアーキテクチャ上のユニットを決定して，アーキテクチャ上で中間コードに対応するデータが流れるパスを発見することが必要で，そのためには，どことどこを繋げばよいか，どうやって繋げばよいかを探す必要がある。本システムではこれを，中間コードとアーキテクチャを表現した深層知識を基礎として行なうことができる。これに，中間コードに対応する表層知識を加える（当然それが適応可能か否かの判断を行なう部分も含めて）事により，コード生成の高速化を図ることができる。

しかし，上記の方法だけでコードを生成しようとすると，知識の探索回数が爆発的に増加する。そこで，なんとかして探索回数を減らしてやらなければ実用上使用できない。そのためには，探索回数を制限するための知識を取り入れる必要がある。パス探索のための知識としては，

1)　有望なパスから探索を行う。

2) ループとなるパスはできるかぎり避ける。
3) 一度発見したパスを有効に使う。

等がある。これらは，知識を利用してマイクロコードを生成するプログラムの内部に埋め込まれる形で使用されている。又，3)に関しては，発見したパスを表層知識として知識ベースに蓄積してゆくことにより実現している[2)]。

この他には，各々リソースの有効な使用法や，中間リソースの割付などを効率よく行う必要がある。

4. システムの構成

4.1 本システムの開発・動作環境

本システムはNECのPC-9801を開発及び稼動環境として，PC-9801用のUNIX[17)]であるPC-UX上のLEX，YACC，C及びMS-DOS上のPrologを用いて作成されている。

4.2 ソース言語の仕様

マイクロプログラム作成用の高級言語は，本システム専用のものを作成した。これは，現在のところ既存の言語ほど大きな仕様のものを必要としないためである。この言語の特徴としては，

1) proc～corpによるモジュールの指定
2) if～elseif～else～fiによる条件判断
3) loop～poolとbreak，continueによるループ制御
4) for～rofによるループ制御
5) callとreturnによるプロシジャ呼び出し

等があげられる。次に，一般の変数宣言は

dcl 変数名 on 実リソース名

の形で行ない，メモリ上の配列については，

dcl 配列名［要素数］at メモリ名［開始アドレス］

の形によって宣言される1次元配列が使用できる。

4.3 パーザ部

ソースプログラムから中間コードへの変換を行うパーザ部は，YACCを用いてLRLA(1)方式によるボトムアップパージングによって行っている。アクション部の意味解析部では，入力ソースプログラムに対応した式木の構造を作りだす。パーザ部で式木はブロック構造を反映した形式をとっており，さらには“定数　二項演算子　定数”及び“単行演算子　定数”の形式に対して，定数の畳込みに関する最適化を行っている。

4.4 中間コード最適化部

ここでは，分岐関係の最適化を行う。そのために必要となる分岐と分岐先のラベルに関するテーブルを式木より作成し，これを用いて分岐の連鎖を見つけ，分岐先のラベルの付替えと不要となった分岐命令及びラベルの削除を行う。さらに，分岐条件を逆にすることで1命令減らせる部分に対する最適化も行っている。これらの最適化は機械に依存しないものであり，中間コードレベルで行うことが，コード生成部に余計な負担をかけないためにも望ましい。

現在，本システムでは上記の最適化を組み込んでいるが，さらにループ展開･融合，局所的な部分式の削除，式の評価順序の決定，大域的共通部分式の削除，ループに関する最適化，複写の伝搬，コード巻き上げ等の，一般のコンパイラで行われている最適化の手法を組み込む必要がある[10,12]。

最適化のすんだ式木はPrologの述語としてファイルに出力され，次のマイクロオペレーション生成部に渡される。

4.5　マイクロオペレーション生成部

マイクロオペレーションの生成部は，パーザの出力する中間コードを入力とし，それを実行するパスを求める事によりマイクロオペレーションを得るという方法を用いている。簡単に述べると，

1)　中間コードのファンクションを軸に，それを実行するアーキテクチャ上のユニットを見つける。

2)　入力と出力をそのユニットに繋ぐためのパスを見つける。

というものである。ここで得られたマイクロオペレーションをマイクロインストラクションにマッピングして，長さが最小のものを選ぶ事により，最適なコードを出力することができる。

Prologを用いる場合，ツリー状の解の探索では時間がかかるので，これをいかにして減らすかが，実行速度に大きく効いてくる。そのため，一度探索した解は表層知識化して知識ベースに取り入れている。又，最適化とも関連して，式の評価に用いる中間値のためのリソースの最適な割り当てに関する方法等，順次取り入れていく必要があるが，現在のところ，本システムではこの部分はリソースが使えるか否かという単純な方法しか行なっておらず，検討中の段階である。

生成のアルゴリズムは，表層知識で解が与えられているか否かを試したのち，解が与えられていなければファンクションを実行するユニットを発見し，そこに入出力を導くべくパスを探索する方法である。パス探索では，あるユニットへの出力から繋ぐべきユニットの入力へ直接繋がるパスが存在すればそれを返し，なければその入力ユニットの出力に繋がるユニットを発見し，そこから最終的に繋げるべきユニットのパスを再帰的に探索する。ここで繋がるということは，アーキテクチャの知識のなかに述語pathによって示される結線が存在するか，またはconnという接続を示すファンクションを実行することにより接続できるかのいずれかが成立することである。そして，探索の過程で発見したパスを成立させるために必要となる操作が，マイクロオペレーションとなって返される。これらはアーキテクチャのスタティックな構造を用いたものである。

4.6　マイクロコードへの詰め込み部

マイクロプログラムへの詰め込み部では，マイクロオペレーション生成部が生成したマイクロオペレーションを，それが実行される時間の依存関係に基づいて最適なマイクロインストラクションに詰め込むことを行う。現在，この部分は作成中であり，ピープルホール方式による最適化を取り入れる予定である。ピープホール方式で十分であるためには，機械独立レベルでの最適化が十分になされている必要がある。その他には，機械依存な最適化が幾つか存在する。これは主に多方向ブランチの用い方に関係した部分である。

マイクロコード生成部では，１つの中間コードに対して複数個のマイクロオペレーション列を生成しうるので，そのなかで最適なもの，即ちマイクロインストラクションに詰め込んでみて

命令数が最小のものを選べばよい。

本詰め込み部の出力は，MDLL であるマイクロアセンブラのソースとするのが，アーキテクチャを変更した場合との比較や人手による最適化等のためにも望ましい。

5. 実 行 結 果

図4に簡単なサンプルプログラムのソースを示す。これをパーザに通して最適化を施した後の中間コードが図5である。ここで式木は，それと同形のリスト構造として表現されており，述語名は program で，第1番目の項がプログラム名，第2番目の項が中間コードのリストである。この中間コードを用いてマイクロオペレーションを生成した例が図6である。

ここで“Inner=”は中間コード，“Path=”は発見したパスを成立させるためのマイクロオペレーションに関する情報，“Timing=”はタイミングに関する情報である。この例では，初めに見つかったパスを示している。

```
proc  example;
   dcl     a on reg(1),b on reg(2),
           c on reg(3),d on reg(4)
   dcl     data[10] at storge[0];
        for a = 0 to 9 ;
                b = data[a] ;
                c = data[a+1] ;
                d = c - b ;
                data[a] = d ;
        rof
corp.
```

図4 サンプルプログラムのソース

```
program(example,[
[set,reg(1),0],
lb1,
[sub,reg(1),9],
[blt,lb2],
[set,reg(2),[read,reg(1)]],
[set,reg(3),[read,[add,reg(1),1]]],
[set,reg(4),[sub,reg(2),reg(3)]],
[write,reg(4),reg(1)],
[set,reg(1),[add,reg(1),1]],
[bra,lb1],
lb2,
[] ]).
```

図5 サンプルプログラムの中間コード

6. お わ り に

データパスを発見してそこからマイクロオペレーションを得るという方法が可能であることは実証された。しかし，今回はアーキテクチャの構造的な面を主に用いて，コード生成を部分ごとのサブゴールに分解してパスを繋げることを主に考えているため，最適なパスの発見に至らない場合も起こりうる。又，今回ターゲットとしたアーキテクチャのように可能なパスの数が少ない場合はよいが，パスの数が増えた場合は，中間リソースの割付けを可能なものをすべて試すという方法ではパス探索の時間が爆発的に増える恐れがある。これを避けるためには，エキスパートによるパス探索の戦略的な知識が重要となる。

図6の例は，最初に見つかったパスを示しており，Prolog のバックトラックを用いて別のパスを探索して，他のパスを発見することも可能である。最適であることの評価は，マイクロ命令に詰込んだときに，発見したパスを用いればマイクロプログラムの語数が最小となることである。しかし，そのためにはバックトラックの量が膨大になるので，ある数よりマイクロオペレーションの数が少ない等の簡単な制限を設ける必要がある。現在この部分は人手によっており，より最適なパスが考えられる場合には人間がバックトラックをかける必要がある。

尚，現在の時点ではマイクロオペレーションからマイクロインストラクションへの詰込みは行っておらず，人手によるマイクロプログラムとの比較は行っていない。

今後はリソースの有効な利用法や割当て，パス探索の効率的な制限方法等についての考察を行い，より良いシステムとしていく計画である。

```
| ?-compile('ex.cod').
file read complet
Generating example
Inner    = [set,reg(1),0]
Path     = [[[[gate_1,on],[e_bus,t],[gate_5,on],[id_bus,t],
  [inrx,set],[mplx_2,sel1],[gate_9,on],[t1_bus,t],[shift_a,through],
  [gate_12,on],[pb_bus,t],[alu,through_b],[r_bus,t],reg(1)]],
  [[reg(1),set]],[]]
Timing   = [[[t0,t0,t0,t0,tx,t0,t0,t0,t0,t0,t0,t1,t0]],[tx],[]]
Label  = lb1
Inner    = [sub,reg(1),9]
Path     = [[[[gate_13,on],[pb_bus,t],[alu,through_b],[r_bus,t],
  [accx,set],[mplx_3,sel1],[pa_bus,t]],[[gate_1,on],[e_bus,t],
  [gate_5,on],[id_bus,t],[inrx,set],[mplx_2,sel1],[gate_9,on],
  [t1_bus,t],[shift_a,through],[gate_12,on],[pb_bus,t]]],[[alu,sub]]]
Timing   = [[[t0,t0,t1,t0,tx,t0,t0],[t0,t0,t0,t0,tx,t0,t0,t0,t0,t0,t0]],[t1]]
Inner    = [blt,lb2]
Path     = [blt,lb2]
Timing   = [t1]
Inner    = [set,reg(2),[read,reg(1)]]
Path     = [[[[storge,out]],[[gate_13,on],[pb_bus,t],[alu,through_b],
  [r_bus,t],[mar,set],[gate_15,on],[a_bus,t]]],[[storge,read]],
  [[d_bus,t],[gate_6,on],[id_bus,t],[extr,set]],[[[gate_4,on],
  [e_bus,t],[gate_5,on],[id_bus,t],[inrx,set],[mplx_2,sel1],
  [gate_9,on],[t1_bus,t],[shift_a,through],[gate_12,on],
  [pb_bus,t],[alu,through_b],[r_bus,t],reg(2)]],[[reg(2),set]],[]]
Timing   = [[[nil],[t0,t0,t1,t0,tx,t0,t0]],[t1],[t0,t0,t0,tx],
  [[t0,t0,t0,t0,tx,t0,t0,t0,t0,t0,t0,t1,t0]],[tx],[]]
Inner    = [set,reg(3),[read,[add,reg(1),1]]]
Path     = [[[[gate_13,on],[pb_bus,t],[alu,through_b],[r_bus,t],
  [accx,set],[mplx_3,sel1],[pa_bus,t]],[[gate_1,on],[e_bus,t],
  [gate_5,on],[id_bus,t],[inrx,set],[mplx_2,sel1],[gate_9,on],
  [t1_bus,t],[shift_a,through],[gate_12,on],[pb_bus,t]]],
  [[alu,add]],[[r_bus,t],[gate_8,on],[id_bus,t],[extr,set]],
  [[[storge,out]],[[gate_4,on],[e_bus,t],[gate_5,on],[id_bus,t],
  [inrx,set],[mplx_2,sel1],[gate_9,on],[t1_bus,t],[shift_a,through],
  [gate_12,on],[pb_bus,t],[alu,through_b],[r_bus,t],[mar,set],
  [gate_15,on],[a_bus,t]]],[[storge,read]],[[d_bus,t],[gate_6,on],
  [id_bus,t],[extr,set]],[[[gate_4,on],[e_bus,t],[gate_5,on],
  [id_bus,t],[inrx,set],[mplx_2,sel1],[gate_9,on],[t1_bus,t],
  [shift_a,through],[gate_12,on],[pb_bus,t],[alu,through_b],
  [r_bus,t],reg(3)]],[[reg(3),set]],[]]
Timing   = [[[t0,t0,t1,t0,tx,t0,t0],[t0,t0,t0,t0,tx,t0,t0,t0,t0,t0,t0]],
  [t1],[t0,t0,t0,tx],[[nil],
  [t0,t0,t0,t0,tx,t0,t0,t0,t0,t0,t0,t1,t0,tx,t0,t0]],
  [t1],[t0,t0,t0,tx],[[t0,t0,t0,t0,tx,t0,t0,t0,t0,t0,t0,t1,t0]],[tx],[]]
Inner    = [set,reg(4),[sub,reg(2),reg(3)]]
Path     = [[[[gate_13,on],[pb_bus,t],[alu,through_b],[r_bus,t],
  [accx,set],[mplx_3,sel1],[pa_bus,t]],[[gate_13,on],[pb_bus,t]]],
  [[alu,sub]],[[r_bus,t],[gate_8,on],[id_bus,t],[extr,set]],
  [[[gate_4,on],[e_bus,t],[gate_5,on],[id_bus,t],[inrx,set],
  [mplx_2,sel1],[gate_9,on],[t1_bus,t],[shift_a,through],
  [gate_12,on],[pb_bus,t],[alu,through_b],[r_bus,t],reg(4)]],
  [[reg(4),set]],[]]
Timing   = [[[t0,t0,t1,t0,tx,t0,t0],[t0,t0]],[t1],[t0,t0,t0,tx],
  [[t0,t0,t0,t0,tx,t0,t0,t0,t0,t0,t0,t1,t0]],[tx],[]]
Inner    = [write,reg(4),reg(1)]
Path     = [[[[gate_13,on],[pb_bus,t],[alu,through_b],[r_bus,t],
  [outr,set],[gate_7,on],[d_bus,t]],[[gate_13,on],[pb_bus,t],
  [alu,through_b],[r_bus,t],[mar,set],[gate_15,on],[a_bus,t]]],
  [[storge,write]],[]]
Timing   = [[[t0,t0,t1,t0,tx,t0,t0],[t0,t0,t1,t0,tx,t0,t0]],[t1],[[]]]
Inner    = [set,reg(1),[add,reg(1),1]]
Path     = [[[[gate_13,on],[pb_bus,t],[alu,through_b],[r_bus,t],
  [accx,set],[mplx_3,sel1],[pa_bus,t]],[[gate_1,on],[e_bus,t],
  [gate_5,on],[id_bus,t],[inrx,set],[mplx_2,sel1],[gate_9,on],
  [t1_bus,t],[shift_a,through],[gate_12,on],[pb_bus,t]]],
  [[alu,add]],[[r_bus,t],[gate_8,on],[id_bus,t],[extr,set]],
  [[[gate_4,on],[e_bus,t],[gate_5,on],[id_bus,t],[inrx,set],
  [mplx_2,sel1],[gate_9,on],[t1_bus,t],[shift_a,through],
  [gate_12,on],[pb_bus,t],[alu,through_b],[r_bus,t],reg(1)]],
  [[reg(1),set]],[]]
Timing   = [[[t0,t0,t1,t0,tx,t0,t0],[t0,t0,t0,t0,tx,t0,t0,t0,t0,t0,t0]],
  [t1],[t0,t0,t0,tx],[[t0,t0,t0,t0,tx,t0,t0,t0,t0,t0,t0,t1,t0]],[tx],[]]
Inner    = [bra,lb1]
Path     = [bra,lb1]
Timing   = [t1]
Label  = lb2
End Generation
no
| ?-nolog.
```

図6　サンプルプログラムより得られたマイクロオペレーション列

参考文献

1) 相磯秀夫，飯塚 肇，坂村 健編：ダイナミック・アーキテクチャ，bit 臨時増刊，共立出版（1980）.
2) 北上 始，國藤 進，宮地泰造，古川康一：論理型プログラミング言語 Prolog による知識ベース管理システム，情報処理，Vol.26, No.11, pp.1283-1295（1985）.
3) 坂村 健：ハードウェア記述言語，情報処理，Vol.22, No.6, pp.570-573（1981）.
4) 実用化が始まったマイクロプログラムの最適化，日経エレクトロニクス，1978／5／1, No.185, pp.76-91.
5) 清水 徹，坂村 健：知識工学に基づいたマイクロプログラム開発システム，MIXER，電子通信学会論文誌（D），Vol.J66-D, No.7, pp.864-871（1983）.
6) 清水 徹，坂村 健：マイクロプログラム開発用のエキスパートシステムにおける知識ベースの一般化，電子通信学会論文誌（D），Vol.J66-D, No.12, pp.1331-1338（1983）.
7) 高木 茂：データパスの検証・補正への一アプローチ，情報処理学会論文誌，Vol.26, No.1, pp.9-18（1985）.
8) 萩原 宏：マイクロプログラミング，コンピュータサイエンス・ライブラリー，産業図書（1977）.
9) 馬場敬信：マイクロプログラミング，ソフトウェア講座23，昭晃堂（1985）.
10) 中田育男：コンパイラ，コンピュータサイエンス・ライブラリー，産業図書（1981）.
11) 三谷和史，宮本衛市：アーキテクチャの知識を用いたマイクロプログラム生成系の開発，昭和61年度電子通信学会総合全国大会（1986）.
12) Alfred V. Aho, Jeffery D. Ullman: Principles of Compiler Design, Addison-Wesley (1977).
13) C. V. Ramamoorthy, Masahiro Tsuchiya: A High-Level language for Horizontal Microprogramming, IEEE Trans. Comput., Vol. C-23, No. 8, pp.791-801 (1974).
14) David A. Patterson: Strum: Structured Microprogram Development System for Correct Firmware, IEEE Trans. Comput., Vol. C-25, No. 10, pp.974-985 (1976).
15) David A. Patterson: An Experiment in High Level Language Microprogramming and Verification, Comm. ACM, Vol. 24, No. 10, pp. 699-709 (1981).
16) David Landskov, Scott Davidson, Bruce Shirver: Local Microcode Compaction Techniqes, Computing Surveys, Vol. 12, No. 3, pp. 261-294 (1980).
17) D. M. Ritchie, K. Thompson: The UNIX Time-Sharing System, Comm. ACM, Vol. 17, No. 7, pp. 365-375 (1974).
18) James R. Duley, Donald L. Dietmeyer: A Digital System Design Language (DDL), IEEE Trans. Comput., Vol. C-17, No. 9, pp. 850-861 (1968).
19) James R. Duley, Donald L. Dietmeyer: Translation of a DDL Digital System Specification to Boolean Equations, IEEE Trans. Comput., Vol. C-18, No. 4, pp. 305-313 (1969).
20) Joseph A. Fisher: Trace Scheduling: A Technique for Global Microcode Compaction, IEEE Trans. Comput., Vol. C-30, No. 7, pp. 478-490 (1981).
21) Mario Tokoro, Eiji Tamura, Takashi Takizuka: Optimization of Microprograms, IEEE Trans. Comput., Vol. C-30, No. 7, pp. 491-504 (1981).
22) Masahiro Tsuchiya, Mario J. Gonzalez: Toward Optimization of Horizontal Microprograms, IEEE Trans.. Comput., Vol. C-25, No, pp. 992-999 (1976).
23) Scott Davidson, Brunce D. Scriver: An Overview of Firmware Engineering, IEEE COMPUTER, pp. 21-33 (May, 1978).
24) Scott Davidson, David Landskov, Bruce D. Shriver, Patrick W. Mallett: Some Experiments in Local Microcode Comapction for Horizontal Machines, IEEE Trans. Comput., Vol. C-30, No. 7, pp. 460-477 (1981).
25) Sadahiro Isoda, Yoshizumi Kobayashi, Toru Ishida: Global Compaction of Horizontal Micropograms Based on the Generalized Data Dependency Graph, IEEE Trans. Comput., Vol. C-32, No. 10, pp. 922-933 (1983).
26) Subrata Dasugupta, John Tartar: The Identification of Maximal Parallelism in Straight-Line Microprograms, IEEE Trans. Comput., Vol. C-25, No. 10, pp. 986-992 (1976).
27) Subrata Dasugupta: The Organization of Microprogram Stores, Computing Surveys, Vol. 11, No. 1, pp. 39-65 (1979).
28) Subrata Dasuguputa: Some Aspects of High-Level Microprogramming, Computing Surveys, Vol. 12, No. 3, pp. 295-323 (1980).
29) Takanobu Baba, Hiroshi Hagiwara: The MPG System: A Machine-Independent Efficient Microprogram Generator, IEEE Trans. Comput., Vol. C-30, No. 6, pp. 373-395 (1981).
30) Tilak Agerwala: Microprogram Optimization: A Survey, IEEE Trans. Comput., Vol. C-25, No. 10, pp. 962-973 (1976).