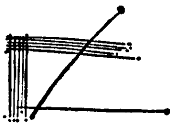
**展 望**



# ファームウェア工学[†]

馬場 敬信[††]

## 1. はじめに

マイクロプログラミング方式は，当初計算機の制御部を系統立てて組織的に設計する方法として提案された[1]. しかしながら，1970年代に入って高速で書換え可能な制御記憶が実用化されるとともに，この方式は単にハードウェアの設計手法にとどまらず，エミュレーションや故障診断，さらには高水準言語の処理など多方面に応用されている[3]. 特に最近ではマイクロプログラムの柔軟性と効率の良さを生かして，オペレーティング・システムやデータベース処理，数値処理など，従来ソフトウェアによって処理されていた種々の問題領域においてもファームウェア化が試みられており，ソフトウェアとハードウェアの両面に大きな影響を与えつつある[4~6),9~11].

このようなマイクロプログラミングの発展は，ファームウェア・ユーザの層をハードウェア技術者からソフトウェア技術者やさらに一般のユーザへと広げるとともに，マイクロプログラムの作成量の増加をもたらしている[7]. このため，能率よくマイクロプログラムを作成するための手段が求められる一方，マイクロプログラムの作成量の増加が将来「ファームウェアの危機」を招くことのないよう，長期的な展望のもとにファームウェアの将来について検討することの必要性が指摘されている[8].

ソフトウェア工学が提案されて10年を経た今日，ファームウェア工学(firmware engineering)という新語が現れた背景は，まさにこのような所にある[12~14]. その内容はまだ十分に固まっていないが，ここでは広く「マイクロプログラムを効率良く作成し，活用するための手法」と定義して，次のような項目について検討を行う.

(1) マイクロプログラムの記述

レジスタ間の転送や算術論理演算装置(ALU)による演算などの基本的な操作を判りやすい形で記述する.

(2) マイクロプログラムの翻訳

記述されたマイクロプログラムを効率の良いオブジェクト・マイクロコードに変換する.

(3) マイクロプログラムの論理チェック

論理シミュレーションや検証によって，マイクロプログラムの論理的な誤りをチェックする.

(4) ファームウェア化の評価

機能をファームウェアで実現した場合の効果を，実行時間や所要記憶量などについて評価する. これと関連して，計算機を使って評価を行い，自動的にファームウェア化を行おうとする試みもある.

このように，現在のファームウェア工学の内容は，要求仕様技法，プログラミング方法論，プログラムの検証などを主な内容とするソフトウェア工学とは格段の隔りがある. これは，マイクロプログラムの作成量がまだソフトウェア程多くないという量的な相違とともに，ソフトウェアの基礎となる機械語プログラムとファームウェアの基礎となるマイクロプログラムとの質的な相違によるものと考えられる. 特に，機械語と比較した場合のマイクロプログラムの特徴を明確にしておくことは重要と思われるので次に列挙する.

(1) 基本的な操作に対する指令

機械語の命令はレジスタ間の転送などの基本的な操作(これをマイクロ操作(micro-operation)と呼ぶ)の組合せから成るが，マイクロプログラムの命令(これをマイクロ命令(micro-instruction)と呼ぶ)はこれらの基本操作を直接制御する.

(2) ハードウェアと密着

ファームウェアは，タイミングやバス構成などのハードウェアの詳細と密接な関連をもつ.

(3) 並列動作の制御

一般に1つのマイクロ命令は並列に実行される複数のマイクロ操作を制御する.

† Firmware Engineering by Takanobu BABA (Department of Information Science, Faculty of Engineering,' Utsunomiya University).

†† 宇都宮大学工学部情報工学科

| | 仕様 | コーディング | テスト |
|---|---|---|---|
| ファームウェア平均 | 20% | 50% | 30% |

| | | | |
|---|---|---|---|
| ソフトウェア平均 | 35% | 20% | 45% |

図-1 マイクロプログラムとプログラムの開発に要した時間の割合い

また，両者の背景を考えてみると，ファームウェアでは特に効率の良いマイクロプログラムが要求される点も大きな相違である．これは，一般に制御記憶の容量が余り大きくないという物理的な制約にもよるが，マイクロプログラムが計算機のハードウェアを直接制御し，また繰り返し使用されることが多いため，その効率が計算機システム全体の効率に大きく影響することによる．この相違の影響はプログラムの作成過程にも現れており，**図-1** にその実験例を示す．この実験結果はソフトウェアと比べてファームウェアでは特にコーディングに時間を要することを示しており，これは効率の良いマイクロプログラムを得ることが1つの理由であると推測される[14),103)].

以上のような相違から，ファームウェア工学においては，単にソフトウェアを対象として開発された技術を応用するだけでなく，ソフトウェアとの相違を明確に意識した独自の手法が開発されつつある．以下，本稿ではこれらの技術の現状を紹介し，問題点を明らかにするとともに将来の展望を行う．

## 2. マイクロプログラム制御方式

### 2.1 マイクロプログラム制御方式計算機

結線論理 (wired logic) 方式の計算機では，命令はデコーダによって解読され，その命令を実行するのに必要な制御信号は制御回路によって発生される．これに対して，マイクロプログラム制御方式の計算機では**図-2** に示すように制御回路は制御記憶 (control memory) と呼ぶ記憶装置に置き換えられる．制御記憶の一語はマイクロ命令と呼ばれ，機械語命令はマイクロ命令の系列として構成される．ソフトウェアのプログラムが一連の命令系列によって構成されるのと対応して，マイクロ命令の系列はマイクロプログラム (microprogram) と呼ばれる．

### 2.2 マイクロ命令の形式

マイクロ命令に含まれる情報は次のように分けられる[3)].

(i) レジスタ間のデータ転送などのマイクロ操作の指定.

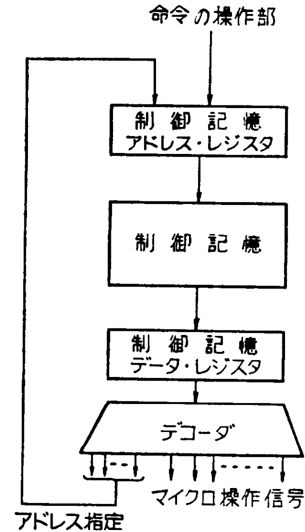


図-2 制御部の構成

(ii) 次に実行するマイクロ命令を決定するための，順序制御に関する情報.

(iii) マイクロ命令で使用する定数

これらの情報を表すためのマイクロ命令の形式は，符号化の方式や順序制御の方式などに応じて種々考えられ，それによってマイクロプログラムの作成に必要な知識も異なってくる．以下，特にマイクロプログラムを作成する立場から，これらの方式について述べる．

#### 2.2.1 符号化のレベル

マイクロ命令の形式は符号化の程度によって水平型 (horizontal type) と垂直型 (vertical type) に分けられる．

水平型マイクロ命令は，更に**図-3** に示すように直接制御方式と符号化制御方式に分けられる．直接制御

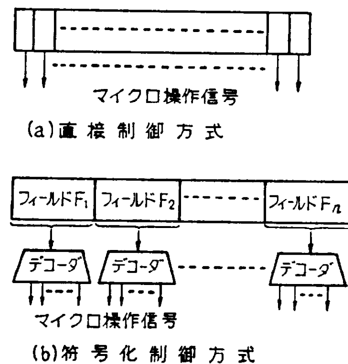


図-3 水平型マイクロ命令

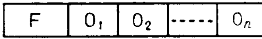



図-4 垂直型マイクロ命令

方式では，マイクロ命令の各ビットが処理装置内のゲート等の制御される点と1対1に対応する．これに対し符号化制御方式では同時に動作することのないマイクロ操作をグループにまとめて符号化してマイクロ命令のフィールドを構成する．

いずれにしても，水平型のマイクロ命令は制御ゲートと直接的な対応関係をもっており，水平型マイクロ命令を対象としてマイクロプログラムの作成を行うには，レジスタとALU間のデータ・パスやゲートの位置，それらの制御の方法やタイミングなど，ハードウェアの詳細についての知識が必要となる．

同時に動作するマイクロ操作の種類や数に制限をつけて符号化の程度を進めると，マイクロ命令のビット数をさらに減少させることができる．この場合のマイクロ命令を垂直型といい，図-4 に示すようにオペレーション部 (F) やオペランド部 ($O_1, O_2, \cdots, O_n$) を持ち，機械語に似た形式となる．

従って，垂直型マイクロ命令では，ハードウェアのゲート・レベルの細かい操作を知らなくてもその意味が判るので，ソフトウェアのプログラム作成と同様にマイクロプログラムの作成が行える．

実際には符号化の程度に応じて，水平型と垂直型の中間的な形式のものもある．また，制御記憶を2つに分け，一方を垂直型とし他方を水平型として，垂直型マイクロ命令の操作を水平型マイクロ命令（これをナノ命令と呼ぶ）の系列（これをナノプログラム(nanoprogram) と呼ぶ）によって実行する方式もあり，2レベル・マイクロプログラミング方式と呼ばれている[6]．

### 2.2.2 符号化の形式

互いに排他的なマイクロ操作信号をグループにまとめて直接符号化してフィールドを構成する方式を単一レベル符号化 (single level encoding) または直接符号化(direct encoding) と呼ぶ．これに対して図-5 に示すようにマイクロ命令のあるフィールドの意味が他のフィールドの値によって変るような方式を2レベル符号化 (two level encoding) または間接符号化 (indirect encoding) と呼ぶ*．

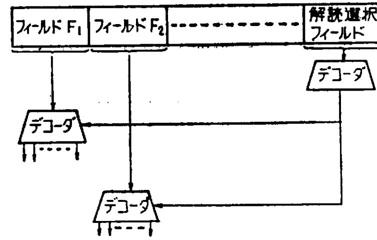



図-5 間接符号化方式（2レベル符号化方式）

2レベル符号化方式を採用するとマイクロ命令のビット長が短縮できるが，1つのマイクロ操作が複数個のマイクロ操作指令（これをマイクロオーダ (micro-order) と呼ぶ）によって制御されることになるため，マイクロプログラムの作成に当っては，フィールド間の符号化の関係に十分注意する必要がある．

### 2.2.3 間接機能制御

制御情報の一部を特別なレジスタにもたせ，同一のマイクロオーダであってもレジスタの内容によって制御されるマイクロ操作の内容が変るような方式を間接機能制御方式と呼ぶ．また，このようなレジスタを機能レジスタ (function register) と呼ぶ．機能レジスタの内容をいったんセットして変更することなく，何ステップものマイクロ命令を実行する間に何度もその内容を利用するような場合には特にレジデュアル制御 (residual control) と呼び，そのときのレジスタをセットアップ・レジスタ (setup register) と呼ぶ．

この方式の長所としては，類似の操作に対して同一のマイクロプログラムを使用することができ，制御記憶の減少に役立つことが挙げられる．一方，この方式を用いるとマイクロプログラムの作成に当っては，実行時に変更される可能性のある機能レジスタの値を考慮に入れなければならないので，それだけマイクロプログラムの作成は複雑となる．

## 2.3 順序制御

マイクロプログラムにおける順序制御の特徴は，制御記憶アドレスレジスタ (CMAR) へのアドレス生成法の種類が多いことである．具体的に列挙すれば

(i) CMARの内容をある値だけ増加する．

(ii) マイクロ命令のフィールドの値をセットする．

(iii) 結線論理によってあらかじめ決った値をセットする．

* 特に，図-5 のようにデコード選択の情報が他のフィールドにある場合にはビット・ステアリング (bit steering) と呼び，また，この情報がレジスタ等にある場合には，フォーマット・シフティング (format shifting) と呼ぶことがある．フォーマット・シフティングは，例えば入出力処理と CPU 処理の状態に応じて命令の解釈を変えるなどに使用されている[3]．

(iv) レジスタなどのハードウェア・リソースの値をセットする.

(v) テストの成立, 不成立に応じて1,0をセットする.

(vi) アドレス・スタックの先頭の値をセットする.

(vii) 現在のCMARの値をそのまま残す.

などである. 実際には, これらの生成法がいくつか組合されて実現されている. また, テストの成立, 不成立に応じて生成法を変えることにより条件分岐が行われる.

このような種々のアドレス生成法は, 制御記憶を効率良く使用し, また多方向分岐などによって処理速度を上げるために利用される. しかし, 反面マイクロプログラムを作成する際にはマイクロ命令の配置に注意する必要がある. 例えば, CMAR 全体に(iv)の方法によりレジスタの値をセットする場合には, 実行時に動的に変化するレジスタの値によって分岐する先のアドレスが決定されることになる.

### 2.4 タイミング

1つのマイクロ命令は一般にいくつかのマイクロ操作を指定するが, その指定の仕方を時間的にみると

(i) 一相のタイミング信号ですべてのマイクロ操作を制御する.

(ii) 多相のタイミング信号により, マイクロ操作信号を順次ずらして発生させる

の二通りが考えられる. (i)は単相 (monophase) 制御と呼ばれマイクロ操作の制御は簡単になるが, 同時に制御できるマイクロ操作は制限される. (ii)は多相 (polyphase) 制御と呼ばれ, 演算のオペランド・レジスタ, 結果の転送先などが自由に選択できるため, 1つのマイクロ命令で多くのリソースに関係するマイクロ操作を制御することができる. 特に多相制御方式の場合には, 1つのマイクロ命令内で更にマイクロ操作間の実行順序関係が考えられるため, この指定は図-6のように二次元的に表される[4,48].

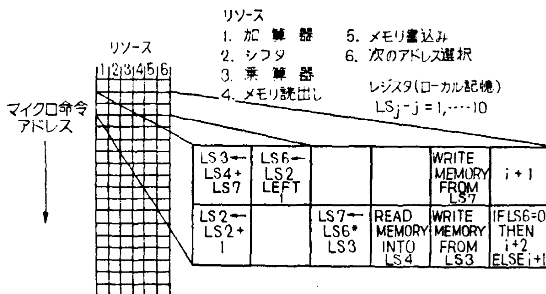



図-6 水平型マイクロプログラムの2次元表示

## 3. マイクロプログラムの記述とその処理

### 3.1 記述とその処理の方式

マイクロ命令はハードウェアを直接制御し, また機械語と比べて一般に語長が長い. このためビットパタンそのものでマイクロプログラムを作成するのは大変な作業となるため, 種々の記述言語とその処理の方式が考えられ実現されている.

処理の方式を決定する際の重要な要素は図-7に示すように, 記述のレベルと機械独立性, 及びオブジェクト・マイクロプログラムの効率に要約される. これらはそれぞれが両立させにくい要素であるため, どこにそのトレード・オフを設定するかが問題となる.

記述のレベルについては, ソフトウェアと同様, アセンブリ言語と高水準言語に分けられる. アセンブリ言語では, 記述とオブジェクト・マイクロ命令が1対1の関係をもっている. これに対して高水準言語ではこの関係がなく, 多対1あるいは1対多の対応関係となる場合が多い. 従って, 高水準言語による記述を処理するに当っては, 効率の良いオブジェクト・コードを生成するための手法が重要な役割を果す.

各レベルについて, 更に機械独立性からいくつかの記述処理システムを分類したのが表-1である. 以下, 各方式の特徴について実例を挙げながら述べる.

### 3.2 アセンブリ言語による記述とその処理[15-30]

アセンブラ・レベルでの記述は, 記述の形式によってリスト形式, 流れ図形式, レジスタ・トランスファ形式に分けられる[6].

#### 3.2.1 リスト形式

マイクロ命令を構成するフィールドごとにネモニックを対応させて羅列するとともに, 記号アドレスを用いた順序制御の記述を行うものである.

マイクロ命令の形式が機械語に近い垂直型のものでは, 一般に

ラベル　オペレーションコード　オペランド

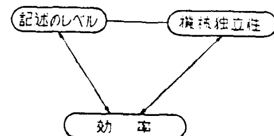



図-7 処理方式決定の要因

表-1 マイクロプログラム記述処理システム

| 分類 | | 言語または処理システムの名称 | 主な特徴 | 参考文献 |
|---|---|---|---|---|
| アセンブリ言語 | 専用 | MIRAGER | 水平型マイクロ命令をもつ MIRAGE マイクロプロセッサ専用．マイクロ操作の指定に，デフォールトの機能を使用できる． | 16 |
| | | BML | Burroughs B-1726 専用．ビット処理機能などの計算機の特徴を記述しやすいように設計されている． | 22 |
| | | MICROBE | 水平型マイクロ命令をもつ計算機専用．ネモニックによるマイクロ操作の指定から，その操作内容を判りやすく記述した注釈を作成する． | 23 |
| | | MICAS/MICSIM | VARIAN V 73 専用．アセンブラおよびリンカ (MICAS)，およびシミュレータ (MICSIM) から成る．レジスタ・トランスファ形式の記述を行う． | 25 |
| | | ANIMIL | 信号処理用演算装置 SPAU 専用．オペレータ順位文法に沿った言語仕様をもつ． | 26 |
| | 汎用 | CAS | 流れ図形式の記述を行う．IBM システム/360 のマイクロプログラムによる制御を行うすべてのモデルに使用された． | 15 |
| | | MDS | レジスタ・トランスファ形式の記述を行う．処理システムはトランスレータ記述システム (TWS) により実現されている． | 17 |
| | | MLTG | マイクロプログラム記述言語のトランスレータ生成システムである． | 28, 29 |
| 高水準言語 | 専用 | GPM | 垂直型マイクロ命令をもつ MLP-900 計算機専用．ブロック構造をもち，代入文の右辺に算術式の記述が行える． | 36 |
| | | PUMPKIN | 水平型マイクロ命令をもつ MCU 計算機専用．変数のレジスタなどへの割付けを自動化し，算術式，論理式の記述や DOWHILE 文，CASE 文などの記述が行える． | 41 |
| | | MPL 200 | 垂直型マイクロ命令をもつ FACOM M-200 L 専用．マイクロ命令の先取りなどの機械の特徴に応じた最適化を行う． | 43 |
| | | FGS | PL/1 風の高水準言語 FGL から MELCOM-COSMO 500 のマイクロ・アセンブラのソースプログラムに変換する．変数へのレジスタ割付けを自動的に行う． | 44, 45 |
| | | STRUM | 2 レベル・マイクロプログラミング方式の Burroughs D-Machine 専用．構造化プログラミングやプログラム検証などソフトウェア工学の技法を取入れている． | 103, 104 |
| | 汎用 | MFL | PL/1 風の仕様をもつ．最適化を行わないため，実用性には欠けるが，機械独立な高水準言語の可能性を初めて処理システムの作成まで行って明らかにした． | 31, 32 |
| | | MPGS | 汎用のハードウェア・シミュレータ GPMS のための記述言語である．記述は計算機の記述，機能部，マイクロプログラムの 3 つの部分から成る． | 33, 34 |
| | | SIMPL | 水平型マイクロ命令を対象とする．変数相互の依存関係に注目した最適化理論を実現している． | 38 |
| | | MPG | 計算機記述部とマイクロプログラムのアルゴリズム記述部とから成る汎用のマイクロプログラム作成システムである． | 106〜110 |

と，機械語に対するアセンブリ言語に似た記述形式となる．図-8 に，垂直型のマイクロ命令をもつ META 4 の記述例を示す．

一方符号化の程度が低い水平型の場合には，ネモニックを指定された形式にしたがって羅列する．この形式としては，カード上のコラム位置を指定する場合や羅列の順序を指定する場合などがある．

一例として，図-9 に水平型のマイクロ命令をもつ HITAC 8350 の乗算のマイクロプログラムの記述例を示す．1 行目と 2 行目で 1 つのマイクロ命令を記述する．MULT はこのマイクロ命令に付けられたラベルである．マイクロオーダ G と定数の指定 (2X), (20) によって，定数 $(20)_{16}$ を発生して，ループ・カウンタ G に転送することが記述されている．2 行目の $(0900)_{16}$ はこのマイクロ命令のアドレスを表し，＊+1 は次のマイクロ命令への無条件分岐を表す．

記述は RCM 処理システム[30]によって処理され，ビットパタン形式に変換されるとともに，指定された動作をレジスタ・トランスファ形式で表したものが出力される．

このように，リスト形式の記述ではマイクロ命令との対応が明確であり，ビット・パタン形式への変換も容易であるため，他にも多くの処理システムが実現されている[3]．

### 3.2.2 流れ図形式

リスト形式記述では，ネモニックの羅列であるため各マイクロ命令で指定される動作が判りにくいこと，およびマイクロ命令間の順序関係が判りにくいことなどの欠点がある．このため，1 つのマイクロ命令に対して 1 つのボックスを割り当てて，流れ図形式に表現する方法が採用されている．

図-10 に IBM システム/360 のマイクロプログラムの作成に使用された CAS[15,101] のボックスの記述形式を示す．

ボックスの左端には左エッジ文字があり，計算機の特定のデータ・パスを表している．たとえば，第一行

```
* META 4 MICROPROGRAM TO PERFORM A BLOCK MOVE INSTRUCTION
*  MOVE N WORDS OF MEMORY FROM ONE LOCATION TO ANOTHER.
*   OVERLAPPING FIELDS CAUSE UNDEFINED RESULT.
*   REGISTER USAGE:
*        SP REGISTER - ADDRESS OF SOURCE FIELD, UPDATED AS WE GO ALONG.
*        DP REGISTER - ADDRESS OF DESTINATION FIELD, ALSO UPDATED.
*        L REGISTER - LENGTH OF SOURCE & DESTINATION FIELDS, = 0 ON RETURN
*
MOVE BRZ  L       RET    W      IF L IS ZERO, JUST RETURN.
LOOP MOVE SP MA          MR     READ A WORD FROM SOURCE FIELD,
     ADDI SP SP   1             AND INCREMENT SP
     MOVE DP MA                 SELECT CORRESPONDING WORD IN DESTINATION
     ADDI DP DP   1              FIELD, AND INCREMENT DP.
     MOVE MD MD          PZ,MW  WRITE SOURCE WORD INTO DESTINATION
     SUBI L  L    1             DECREMENT LENGTH,
     BNZ  L       LOOP   W       AND DO ANOTHER IF NOT EXHAUSTED.
RET  RETURN
```

図-8 META 4 のマイクロプログラムの記述例

```
EOSMULT   G                          (2X)          (20)  X
  0900                               *+1
EOSL0     (SPM)  (U0)    W SRA                           X
                                     *+1
EOS*                                                     X
               T  SC=0               L1
EOS*      U      (U1)    W +         U                   X
                                     *+1
EOSL 1    U                  SRL     U      G-1          X
                                     *+1
EOS*      L                  SRC     L                   X
               T  G¬=0               L0
EOS*      (SPM)  (U0)                U                   X
                                     *+1
EOS*      (SPM)  (U1)                L                   X
                                     END
```

図-9 HITAC 8350 のマイクロプログラムの記述例

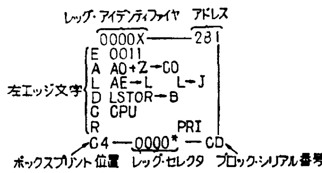



図-10 CAS におけるマイクロ命令の記述例

のEはその行が定数 (0011) の発生の記述であることを示し，第二行のAは ALU による演算であることを表す．各行の記述は単なるネモニックの羅列だけではなく，算術式的な表現もできるようになっている．また，レッグ・アイデンティファイヤとレッグ・セレクタは，アドレスや分岐の指定に使用される．

特に CAS においては SYSTEM/360 のすべてのモデルに使用できるように，左エッジ文字の選択によって種々のマイクロ命令構成を扱えるよう工夫されている．

図-11 に CAS によるマイクロプログラムの記述例を示す．図から判るように，RR 型命令 (LOAD, AND など) の実行マイクロプログラムの先頭へ機能分岐した後，それぞれの命令に対する演算を行っている．

流れ図形式では，特に水平型のように1つのマイクロ命令が種々のリソースの制御を同時に行うという場合に，これを判りやすく表現することができる．ただし，そのまま計算機の入力とはならないので，入力はリスト形式に直してカードにパンチして行われる．

### 3.2.3 レジスタ・トランスファ形式

リスト形式と流れ図形式では，いずれもソフトウェアにおける高水準言語との共通点がほとんどない．これに対して，レジスタ・トランスファ形式の言語では，レジスタ間の転送を代入文に，単項，二項の演算を算術式に，そしてテスト条件による分岐を IF 文に，それぞれ対応させて記述を行う．従って，高水準言語に近い形式ではあるが，ステートメントとマイクロ命令の対応は1対1であり，本質的にはアセンブリ言語である．

レジスタ・トランスファ形式は，ドキュメント性にすぐれ，オブジェクト・マイクロコードの効率も良いことから多くの記述処理システムに採用されているが，ここではその中から2つの例を挙げる．

BML は，垂直型のマイクロ命令をもつ B-1726 計算機専用のマイクロアセンブリ言語である[22]．B-1726 はマイクロプログラムレベルでスタックが使用でき，またビット単位の処理が容易であるなどの特徴をもっているため，BML もこれらの機能を記述できるよう設計されている．図-12 に除算の記述例を示す．

MDS は，機械独立なレジスタ・トランスファ言語を使用したマイクロプログラム設計システムである[17]．MDS のサブシステムである MDS トランスレータは TWS (Translator Writing System) により実現されており，使用者は TWS に対して計算機のファシリティ，言語の構文および意味を入力する．これら

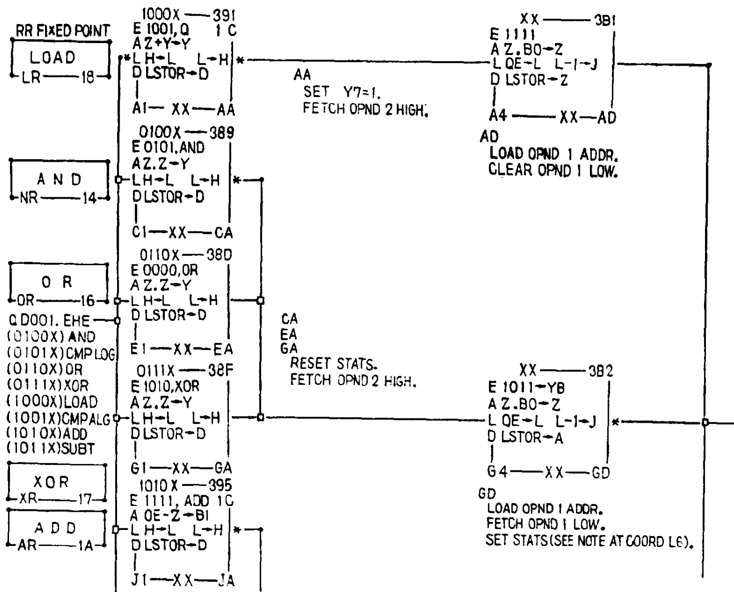



図-11 CAS におけるマイクロプログラムの記述例

```
DIVIDE  CYF←0                   *Clear carry
        X←0                     *Clear partial remainder
        FA←24                   *Set up count
LOOP    Y←T(23,1)               *Insert high bit from T into
                                 Y
        X←X+Y                   *Add into X
        T←SHL T(1)              *Shift divident quatient
        Y←L                     *Put divisor in Y
        IF X<Y GO TO NOSUB      *Do we subtract divisor from
                                 partial remainder?
        X←X−Y                   *Subtract
        TF←TF V B '0001'        *Set low order quatiant bit
NOSUB   FA←FA−1                 *Decrement count
        IF FA=0 GO TO DONE      *Are we done?
        X←SHL X(1)              *Shift partial remainder
        GO TO LOOP
DONE    RETURN
```

図-12 BML の記述例

の入力から生成されるトランスレータ (tailored translator) に対して，マイクロプログラムを記述して入力する．

富士通において開発された MLTG も MDS と同じような構想のもとに作成された汎用のマイクロプログラム・トランスレータ・ジェネレータである[28),29)]．

### 3.3 高水準言語による記述とその処理[31)~45)]

3.2 に述べたアセンブリ言語の中には高水準言語風のものもあるが，本質的にはオブジェクト・マイクロ命令と1対1の対応をするものであり，コーディングのしやすさ，ドキュメント性などについて不十分な点が多い[50)]．このため，ソフトウェアにおける高水準言語に対応して，マイクロプログラミング用高水準言語が設計され，その処理システムの作成が行われている．

マイクロプログラミング用高水準言語はその汎用性から特定の機械に依存するものと，依存しないものとに分けられる．

#### 3.3.1 機械に依存する高水準言語

機械のハードウェアの詳細が予め判っている場合には，記述言語の設計，およびその処理システムの作成は比較的容易である．このため，ソフトウェアにおける高水準言語に近い機能を取り入れた言語が多い．その特徴の主なものを次に列挙する．

(i) BEGIN, END などで示されるブロック構造をもつ．

(ii) 算術式，論理式の記述が行える．

(iii) IF THEN ELSE, DO WHILE, CASE などの条件文，繰り返し文が記述できる．

(iv) ハードウェアにない機能を記述できる．例えば，ハードウェアでは負の数に対するテスト条件しかない場合に，非負の数に対するテスト条件を記述できる．

(v) 実際のハードウェアのファシリティ*を意識せずに，変数を宣言して使用できる．

これらの特徴は，ソフトウェアにおける高水準言語では普通に実現されているが，ハードウェアと密接に関連するファームウェアでは，これらの特徴をどこまで取り入れるかは，オブジェクト・マイクロコードの効率とも関連して重要な問題となる．

例えば，(ii)については算術式の一時記憶として使用するファシリティをどうするか，(iii)については，制御文を実際のハードウェアの順序制御機能でどのように実現するか，(iv)では，どのようなマイクロ操作を組合わせて実際のハードウェアにない機能を実現す

* 本稿では，レジスタ，主記憶などの記憶要素をファシリティ (facility) と呼び，ターミナルなどの非記憶要素と合せたものをリソース (resource) と呼ぶ．

るか，(ｖ)では，バス構成などによって異なる条件をもつファシリティをどのように割付けたらよいか[42]，など解決すべき問題は多い．

機械に依存する高水準言語の一例として PUMPKIN について述べる[41]．PUMPKIN は 64 ビットの水平型マイクロ命令をもつMCU計算機に依存した高水準言語であり，上記の(ｉ)から(ｖ)の特徴のほとんどを備えている．図-13 に示す記述例は RX 型の命令を入力として，その命令コードとレジスタ・オペランドを一時記憶に書込み，実効アドレスの計算を行ってから，命令に対する実行マイクロプログラムを呼び出すものである．

### 3.3.2 機械に依存しない高水準言語

高水準言語とその処理システムが機械に独立に実現できれば，

(ｉ) 記述されたマイクロプログラムは，標準的なドキュメントとして，ソフトウェア，ファームウェア，およびハードウェア技術者間の情報の受渡しや保守に使用できる．

(ⅱ) 計算機ごとに処理システムを作成する必要がない．

(ⅲ) 計算機の設計，製作時のハードウェアの変更に対処しやすい．

などの利点が考えられる．更にマイクロプログラムの記述が完全に機械独立なら，

(ⅳ) マイクロプログラムが可搬 (portable) となるという大きな利点もある．従って，その実現が強く望まれるが，機械独立性とオブジェクト・マイクロコードの効率を両立させることは容易ではなく，広く受入れられる言語はまだ実現に至っていない．

しかし，機械独立とするための種々の試みがなされており，それらは次のように分類される．

(１) トランスレータ記述システム (TWS)

ソフトウェアと同様に，マイクロプログラム記述言語の構文と意味を入力として，目的の計算機に対するトランスレータを出力するものである．

この方式によれば汎用性は最も大きいが，機械の特長を生かした最適化を行うようなトランスレータを生成するのは困難である．このため，実現されているシステムは，先に述べた MDS や MLTG のようにアセンブラ・レベルのものが多い．

(２) 拡張可能な言語 (Extensible Language)

言語に基本的な機能の記述能力をもたせるとともに，新しいデータ構造や演算子を定義して拡張できる

```
PROC PARSERX(RX);
DCL 1 RX DWORD, "32 BIT RX INSTRUCTION"
        2 OPCODE BIT (8), "OPERATION CODE"
        2 REG 1 BIT (4), "SOURCE/TARGET REGISTER"
        2 STORAGE_ADDRESS BIT (20),
            3 INDEX BIT (4), "INDEX REG"
            3 BASE BIT (4), "BASE REG"
            3 DISPLACEMENT BIT (12);

    DCL PARSED_OPCODE WORD IN LSB (8);
                                "LOCAL STORE B, LOC 8"
    DCL REGNO WORD IN LSB (9);
    DCL EFFEC_ADDRH WORD IN LSB (10);
    DCL EFFEC_ADDRL WORD IN LSB (11);

    "SIMULATED GENERAL PURPOSE REGISTERS"
DCL 1 GPR (0 : 15) DWORD IN BSM 1 (100),
                                      "BUFFER STORE"
        2 HIGH WORD,
        2 LOW WORD;

  "THIS ROUTINE IS PASSED A/360 RX TYPE INSTRUC-
  TION AS A PARAMETER, STORES THE OPCODE AND
  REGISTER OPERAND IN LOCAL STORE, AND CALCU-
  LATES THE EFFECTIVE ADDRESS OF THE STORAGE
  OPERAND (BASE REGISTER+INDEX REGISTER+12 BIT
  DISPLACEMENT). IT THEN CALLS EXECRX TO EXECUTE
  THE INSTRUCTION."

    PARSED_OPCODE <- OPCODE; "SET OPCODE IN LSB"
    REGNO <- REG1; "SET REG # IN LSB"
    EFFEC_ADDRH <- 0; "ZERO HIGH 16 BITS OF EA"

  "CARRYOUT RETURNS THE VALUE OF THE ADDER
  CARRY WHEN EVALUATING THE EXPRESSION USED AS
  AN ARGUMENT. AS A SIDE EFFECT, THE 16 BIT RESULT
  MAY BE ASSIGNED WITHIN THE FUNCTION."

    EFFEC_ADDRH <- CARRYOUT (EFFEC ADDRL <-
        DISPLACEMENT+LOW(BASE))+HIGH(BASE);
    IF INDEX ¬=0 THEN
        IF CARRYOUT (EFFEC_ADDRL <- EFFEC_ADDRL
                                      +LOW (INDEX))
        THEN EFFEC_ADDRH <- EFFEC_ADDRH+1;

    "ZERO HIGH BYTE OF SUM (24 BIT ADDRESSING)"
    EFFEC_ADDRH <- (EFFEC_ADDRH
                  +HIGH (INDEX)) & X'00FF';

    CALL EXECRX; "EXECUTE THE RX INSTRUCTION"

END PARSERX;                  "C. BERGMAN, AUG. 73"
```

図-13 PUMPKIN の記述例

能力をもたせる方式である[46]．

この方式では，どの程度機械独立になるかは，基本的な機能の選び方と，拡張の可能性に依存することになる．また，拡張された機械依存の部分を翻訳するには，各機械に対して別々のトランスレータを必要とする．

(３) 計算機の記述とマイクロプログラムの記述の組合せ

マイクロプログラムとともに，対象とする計算機を定義して入力とするものである．つまり，入力された計算機の記述によってトランスレータを対象とする計算機用にするもので，(1)に近い方式といえる．この方式は，計算機の定義とマイクロプログラムの関係によって，更に次のように分けられる．

(i) 計算機の記述において定義された変数名やテスト条件を使って，マイクロプログラムを記述する．

(ii) マイクロプログラムの記述とともに，その中で使用したファシリティと，計算機の定義中のファシリティとの対応を記述する．

(iii) 計算機の記述と独立にマイクロプログラムの記述を行う．

このうち，(iii)の方式ではマイクロプログラムに記述された変数と定義された計算機のファシリティとの対応づけをコンパイラが行うことになるため，効率の良いマイクロプログラムの生成は困難となる．(i)と(ii)については処理システムの作成も行われており，(i)の例として MPG[106]，MPL[31),32]，(ii)の例として MPGS[33),34] などが挙げられる．

(4) 機械独立処理と機械依存処理の分離

マイクロプログラムの翻訳処理は，通常，字句解析，構文解析などの機械独立な処理と，コード生成などの機械依存の処理に分けられる．これは上記の(1)から(3)の各方式においても同様であるが，これを一歩進めて機械に独立な部分と機械に依存する部分をモジュール化して分けておき，機械に依存する部分だけを入れ替えて使用することが考えられる．

この方式は，最も直接的な実現法であり，機械ごとの特殊な機能にも対処しやすいし，また機械の特長を活かした最適化なども行いやすい．ただし，この方式では，機械に依存する部分の占める割合いが大きくなると余り意味がなくなるため，最適化処理などに関して機械独立と機械依存の処理をどのように分担するかなども問題となる[47),49),112]．

処理の方式は，一応このように分類されるが，実際にはこれらの方式を組合わせたものや，中間的なものもある．

### 3.4 課　題

以上の議論より明らかなように，ファームウェアにおいては機械のハードウェアの特徴を反映した言語が種々設計され，実現されている．これは，マイクロプログラムがハードウェアに密着しており，また効率の良さが求められるといった，ファームウェア固有の理由による．しかし，このようなファームウェアの特徴は同時に機械独立な高水準言語の開発を難しくしており，マイクロプログラム記述言語を設計するに当っては常に記述のレベルや汎用性と，オブジェクト・コードの効率をどのように両立させるかという問題に直面することになる．

しかしながら，記述のレベルを上げ，また機械独立にするための種々の試みがなされており，このような努力がやがては広く受け入れられる高水準言語の実現に結びつく可能性は十分にある．

更に，構造化プログラミングなどのソフトウェア工学における手法をファームウェアに応用することも重要な課題である．

## 4. マイクロプログラムの最適化

### 4.1 最適化の背景

マイクロプログラムは計算機の制御信号と直接的な関係があり，また一度作成されれば繰り返し使用されることが多いため，機械語によるプログラムと比較して更に効率の良いオブジェクト・コードが求められる．

アセンブリ言語では，マイクロプログラムの記述とオブジェクト・マイクロ命令とが1対1の対応をしているため，最適化*はマイクロプログラムの記述を行う際に，プログラマによって行われる．これに対して，高水準言語では，コンパイラができるだけ効率の良いオブジェクト・マイクロコードを生成する必要がある．特に，マイクロプログラムでは，並列に実行可能なマイクロ操作を1つのマイクロ命令にまとめることが，マイクロ命令数や，実行ステップ数の減少に大きく影響するという特徴がある．このためリソースの使用状況の他に，2章で述べたマイクロ命令の形式やタイミングなどを考慮したきめの細かい最適化を行える可能性がある．

### 4.2 最 適 化 法

マイクロプログラムの最適化に関しては種々の手法が提案されており，既に優れたサーベイもある[54)~56)]．ここでは視点を変えて，最適化法を実際の処理システムに実現するという工学的な立場からまとめてみる．

また，ROM を対象とした場合には，予め制御すべきマイクロ操作のリストを与えて(ビット長)×(語数)が最小となるようにマイクロ命令形式を設計する問題

* ここでいう最適化とは，効率の改善を意味する[53)]．

がある[57)~61)]. これも広くはマイクロプログラムの最適化法に入るが，ファームウェア工学の立場からはマイクロ命令形式は予め決められたものと仮定するのが適当と考え，対象から除外した.

(1) マイクロ命令の形式と最適化

垂直型マイクロ命令は機械語とほぼ同じような形式であるため，ソフトウェアにおける最適化の手法が応用できる[53),62)]. 例えば，

(i) 前に行われた操作によって既に結果の得られている操作や，結果が使用されない操作を除去する.

(ii) プログラム中で実行頻度の高い部分の操作の数を少くする.

などである.

これに対して，水平型の場合には一つのマイクロ命令が複数のマイクロ操作を制御するため，概念的には図-6 に示すような二次元での圧縮になる.

圧縮を行う際に考慮すべき事は多いが，あるプログラム・セグメント内に限れば次の通りである.

(i) マイクロ操作指令の属するフィールド.

(ii) 2レベル符号化の影響.

(iii) 多相制御の場合には，各マイクロ操作の実行タイミング.

(iv) ハードウェア・リソースの競合の問題.

(v) マイクロ操作間のデータ依存性.

(i), (ii)はマイクロ命令形式に関する条件である. (iii)は，1つのマイクロ命令の制御するマイクロ操作間に一定の時間関係があることによる条件である. (iv)については，例えば「カウンタのデクリメントを行うマイクロ操作」と「そのカウンタにデータを転送するマイクロ操作」のように互いに同じリソースを使用し，かつ同じタイミングで実行されるものは1つのマイクロ命令に圧縮できないなどである[80)]. (v)は他のマイクロ操作によって決定されたデータを使用する，あるいは他のマイクロ操作の使用するデータを破壊するなどの関係を表している[38,79)].

Ramamoorthy, Tsuchiya らは特に(iv)と(v)を考慮した最適化法を考え，SIMPL コンパイラに実現している[38)]. また，この流れをくむ理論的な考察もいくつか行われている[64~66)]. 更に，最近では実際的な面からの要求を背景に，(i), (ii), (iii)をも考慮した手法が考えられている. Dasgupta は多相制御にも適用できる方法を考えている[67,68)]. また，筆者らは，(i)から(v)の情報を計算機内に表すため，フレームと呼ぶ二次元のデータ構造を用いてマイクロプログラムの構成を行った[108)]. 所らはやはり縦軸にマシン・サイクルをとり，横軸にリソース名をとったテンプレートと呼ぶ二次元のデータ構造上にマイクロプログラムを構成している[69),135)].

このように，二次元的な処理が必要となるのが，水平型マイクロ命令に対する最適化の特徴である[70)].

(2) マイクロプログラム・セグメントと最適化

セグメントとは，実行が行われる場合には，必ずその先頭から最後までが順次実行される部分を指す. マイクロ操作指令の移動をセグメント内だけに限定するものを「ローカルな最適化」と呼び，セグメント間の移動をも行うものを「グローバルな最適化」と呼ぶ[38)].

明らかに，グローバルな最適化まで行った方が効率は良くなるが，その理由の1つとして(1)に述べたような無意味な操作の除去や圧縮の可能性が，マイクロ操作指令のセグメント間の移動によって向上することが挙げられる[71)]. 例えば，所らの方法[72)]はパイプライン制御による計算機の特徴を生かして，セグメント間のコード移動を行うことにより圧縮の可能性を上げている.

一方，グローバルな最適化を行う際の問題は，すべての移動の可能性を尽すことが実際上難しいことである. このため部分的にヒューリスティックな手法を取入れることによって，計算量を少くする工夫が必要となる.

(3) ファシリティ割付けの最適化

マイクロプログラムの記述が計算機のファシリティ名を使用して行われる場合には，コンパイラは変数へのファシリティ割付けを行う必要はない. しかし，記述のレベルをあげて，実際のファシリティとは独立な変数を使用できるようにしたり，複雑な算術式を使用できるようにしたりする場合には，コンパイラが割付けを行う必要がある.

これは，ソフトウェア・レベルでのレジスタ割付けの問題[53)]と似ているが，ソフトウェアでの汎用レジスタがほぼ一様な機能を持っているのに対して，ファームウェアでは各ファシリティが違った機能を持っていることが多い点が異なる. 例えば，あるレジスタからALUの左側端子へは1マイクロ命令で転送できるが，右側端子への入力は2マイクロ命令を要する場合などがある. また，ファシリティによっては，1マイクロ命令では読出しと書込みが同時に行えないなどの制約のあるものもある. これらは，個々の計算機のハード

ウェア構成によるものが多く，これらの制約をも考慮に入れた機械独立な手法を確立するのは容易ではない．

更に，変数名ばかりでなく，そのビット長も任意に宣言して使用できるようにしたり，実際の計算機のレジスタ数を越えて変数を宣言するなど，その自由度を高めて行くと，変換に手間を要するだけでなくオブジェクトマイクロコードの効率低下を招くことになる．

一例として，IBM の PL/MP コンパイラで開発されたレジスタ割付けの手法について述べる[113]．割付けは割振り (allocation) と割付け (assignment) の2つに分けて行う．

（i） 割振り：プログラムの任意の点で実際のレジスタより多くの変数が生きている（すなわち，後で使用される）状態にならないよう，コードの移動とロード／ストア命令の挿入を行う．

（ii） 割付け：割振られた変数を実際のレジスタに割付ける．1つの変数に2つのレジスタが割振られている場合には，レジスタ間の転送命令を挿入する．

（i）と（ii）の処理を行うために，プログラム・フロー解析を行い，変数が生きている点を明らかにする．これをもとに，最小の数となるような変数の組合せを求めるわけであるが，これは変数が共に生きている状態をコンパティブルと定義することにより，論理回路における素項のリダクション (prime implicant reduction) 手法に帰着される．

（4） アドレス空間の最適化

マイクロプログラムの順序制御には，**2.3** で述べたようにレジスタの値を利用した多方向分岐をはじめとする特徴のあるアドレス生成法がある．このことは逆に，あるマイクロ命令のアドレスの決定が，それに続くマイクロ命令のアドレスを制限することを意味する．従って，マイクロプログラマにとって，アドレス付けを行うことはかなり面倒な仕事である．更に，実際にはコンパクトな領域へ割付けを行うことが求められるが，これは

（i） 制御記憶の容量が一般に余り大きくない．

（ii） ページ単位で制御記憶の内容を主記憶の内容と入替える場合がある．

（iii） あるマイクロ命令から次に実行するマイクロ命令への分岐の範囲が限られることが多い．

などによる．また，オブジェクト・モジュールをいくつかリンクして実行する場合などに，その最大長をできるだけ短くすることが望まれる．

このようなアドレス空間の最適化は，マイクロプログラム・コンパイラを実現するための重要な問題であるが，現在のところ研究成果の報告は余りなされていない[73),74),111)]．

### 4.3 課　題

マイクロプログラムの最適化に関しては，理論的な面に偏りすぎたとの指摘もあるが[54]，実現への試みも種々行われている．しかし，マイクロプログラムが低レベルであることを考えると，ハードウェアの特徴を生かした最適化の手法について更に検討する余地がある．

また，記述処理システムに最適化法を実現する場合には，記述言語のレベルやオブジェクト・マイクロプログラムの効率をどの程度に設定するかといった，システム全体の設計方針に沿って，適当な方法を検討する必要がある．

最適化処理は，論理チェックの手法にも影響を与える可能性がある．すなわち，最適化処理を行えば行うほど，記述されたマイクロプログラムと翻訳されたマイクロプログラムの隔りが一般に大きくなるためそれを補う論理チェックの手法なども同時に開発する必要がある．筆者らも，シミュレータが完成するまでの間，ローカルな最適化を行うコンパイラの出力を書換え可能な制御記憶にロードしてステップ・ランさせデバッグを行ったが，このときソース・リストが役に立たず，オブジェクト・マイクロプログラムのリストを見ながら CPU パネルを操作したという経験がある．

## 5. マイクロプログラムの論理チェック

### 5.1 論理チェックの方式

マイクロプログラムは計算機の制御信号と直接関係しているため，論理チェックにおいても，マイクロ操作間の並列性や実行タイミングなどを考慮する必要がある．また，特に効率が問題とされるマイクロプログラムでは，あらかじめ実行所要時間などを知ることも評価の資料として重要である．

このためには，ソフトウェアによるシミュレーションを行うのが一般的であるが，最近ではそれ以外にも種々の方式が工夫されている[85]．

（1） シミュレータ[82)~84)]

ソフトウェアによりハードウェアの動作を追跡するもので，現在のところ最も一般的な手法である．ソフ

トウェアの柔軟性を生かして，論理回路レベルから，機能ユニット・レベルまで種々のレベルでのシミュレータが作成されている．

（2） デバッガ[85),86)]

マイクロプログラムを実際に制御記憶にロードして実行させながら，その途中経過を見たり，最終的な結果を得るものである．

（3） 検証[87)~97)]

ソフトウェアにおける検証と同様に，マイクロプログラムの任意の点でのレジスタや主記憶の内容についての表明 (assertion) をもとにして，エラーのないことを証明する．

以下，各方式について述べる．

### 5.2 シミュレータ

トランスレータと同じように，シミュレータも特定の機械に依存するものと，依存しないものとに分けられる．

#### 5.2.1 専用シミュレータ

特定の計算機を対象としたシミュレータであるため，対象とする計算機の特徴を反映させて，効率の良いシミュレーションを行うことができる．しかし，機種ごとに作成する必要があり，また計算機の設計段階で使用する場合には，アーキテクチャの変更があるたびに修正が必要となる．

専用方式シミュレータは，更にコンパイラ方式とインタプリタ方式に大別される[99)]．

コンパイラ方式の場合，マイクロ命令はシミュレーションに先立ってシミュレーションを行う計算機のプログラムに変換される．このプログラムの実行によってシミュレーションが行われるため，実行時間は速いが，マイクロ命令の変更などがあるとそのつどコンパイルし直す必要がある．

インタプリタ方式の場合には，マイクロ命令はそのままの形で記憶装置に入れておき，実行時にそれを解釈しながらシミュレーションを行う．従って，実行速度は遅いが，シミュレータの開発は容易であり，また融通性に富んでいる．

専用方式シミュレータは，その処理が比較的簡単なためか，処理の詳細について報告されたものは少ない[102)]．

#### 5.2.2 汎用シミュレータ

汎用シミュレータは計算機に独立なシステムとして作成されるため，対象とする計算機のハードウェアを記述してシミュレータに入力する必要がある．従って，シミュレータはこのハードウェアの記述と，マイクロプログラムおよびシミュレーションの制御情報などを入力として，シミュレーションを実行する．

処理方式は，専用方式と同様に，シミュレーションを行う計算機のプログラムへの翻訳をどの程度行うかによって，次のように分類できる[6),102)]．

（i） ハードウェアの記述もマイクロプログラムの記述も共にいったんシミュレーションを行う計算機のプログラムに変換する．

（ii） ハードウェアの記述のみを翻訳する．マイクロプログラムはインタプリティブに解釈実行される．

（iii） マイクロプログラムのみを翻訳する．シミュレーション時には表形式などのデータ構造に変換されたハードウェア記述を，翻訳されたプログラムが参照しながらシミュレーションを行う．

（iv） ハードウェア記述もマイクロプログラムも共にプログラムには翻訳しない．表形式などのデータ構造に変換されたハードウェアを参照しながら，マイクロプログラムを解釈実行する．

（i）の方式は，シミュレーションの実行は速いが，ハードウェアの記述あるいはマイクロプログラムを変更するたびに翻訳処理を行う必要がある．（iv）の方式は，ハードウェアの記述の変更あるいはマイクロプログラムの変更があっても容易に対処できるが，実行に要する時間は長くなる．（ii）と（iii）は，（i）と（iv）の中間的な段階の処理形式である．

次に，汎用シミュレータの一例として，CAS のシミュレータについて述べる[3),101)]．

CAS シミュレータは**図-14** に示すような構成で，マイクロプログラム記述，計算機記述，およびシミュレーション・データを入力とする．

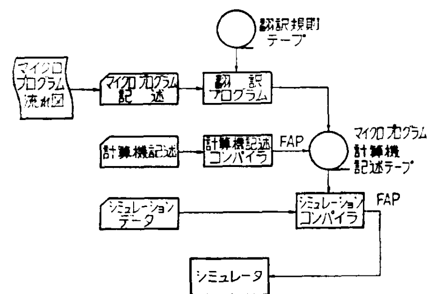



**図-14** CAS シミュレータの構成

計算機記述は，CAS 計算機記述言語で書かれ，計算機記述コンパイラによって FAP (Fortran Assembly Program) 言語に変換される．

マイクロプログラムは，3.2.2 で述べたように流れ図で記述され，翻訳プログラムによって標準の CAS マイクロプログラム言語に変換される．

シミュレーション・コンパイラは，以上の変換されたマイクロプログラムと計算機の記述，およびシミュレーションの出力を制御するためのモニタリング・コマンドやシミュレートされる計算機の初期条件などを与えるシミュレーション・データを入力として FAP 言語のプログラムを出力する．

以上からも明らかなように，CAS シミュレータは（i）の方式によるシミュレータである．

この他，（i）の方式によるシミュレータとして，LFM[98]，（iv）の方式によるシミュレータとして MPG シミュレータ[109]などが挙げられる．また，FALOS[100]は計算機記述をある程度翻訳することから，（ii）と（iv）の中間的な形式と考えられる．

### 5.3 デバッガ

マイクロプログラムを実際に制御記憶にロードして論理チェックを行う場合の利点は，シミュレータと比べて，

（i） 割込み処理，リアルタイム処理などソフトウェアのシミュレータではチェックが困難な部分もチェックできる．

（ii） シミュレータのように模擬される動作と実際の計算機の動作とが食い違う恐れがない．

（iii） ソフトウェアに関しては，シミュレータより開発が容易である．

（iv） 実行速度が速い．

などが挙げられる．

一方，欠点としては，

（i） 特別なハードウェアが必要となる．

（ii） ブレーク・ポイントの設定などが使用者の負担となる．

（iii） 汎用性がない．また，計算機の設計変更などをテストしたり評価することなどもできない．

（iv） 論理チェックを行う間，対象となる計算機が専有される．

などが考えられる．

従って，既にできあがった計算機に対して，専用の論理チェッカとして使用する場合などに適していると考えられる．

表-2 HP 21 MX デバッガのコマンド

| コマンド | 意味 |
|---|---|
| LOAD | オブジェクト・マイクロプログラムを主記憶にロードする． |
| WRITE | 主記憶にあるオブジェクト・マイクロプログラムをWCSに書込む． |
| READ | WCS にあるマイクロプログラムを主記憶に読出す． |
| DUMP | 主記憶にあるマイクロプログラムを外部の装置にダンプする． |
| PREPARE | PROM を焼くためのテープを準備する． |
| VERIFY | PROM テープの内容と主記憶にあるマイクロプログラムを比較する． |
| SHOW | 指定された WCS の内容を表示する． |
| MODIFY | 主記憶と WCS にあるマイクロプログラムの指定された部分を変更する． |
| BREAK | 指定されたマイクロプログラムのアドレスにブレークポイントをセットする． |
| CHANGE | 指定されたレジスタの内容を変更する． |
| EXECUTE | 指定されたアドレスから実行を開始する．ブレークポイントアドレスに到達するかマイクロプログラムが終了するまで実行を続ける． |
| MOVE | マイクロプログラムを他の番地に移す． |
| FINISH | デバッガの実行を終了する． |

一例として，HP 21 MX 計算機のデバッガ・コマンドを表-2 に示す[4]．

### 5.4 検　証

シミュレーションによるテストでは，エラーがあることは判るが，エラーのないことは証明できない[88]．このため，プログラムの検証理論をファームウェアに応用することが試みられている．

具体的には，表明 (assertion) によってマイクロプログラムの任意の点でのレジスタや主記憶の内容をあらかじめ記述することによって，論理チェックを行う[87]．

検証によれば，ソース文のレベルで論理チェックが行えるため，

（i） 最適化の影響を受けない

（ii） ハードウェアの詳細を知らなくても論理チェックが行える

などの特徴があり，高水準言語による記述に適した手法といえる．一方，欠点は検証のためのソフトウェア (verifier) の作成が大変なことであるが，ソフトウェア用に作成された検証システムを応用することも可能である．

マイクロプログラムの検証システムとしては，STRUM[103),104] や MIDDLE[91] などが挙げられる．また，マイクロプログラムの特徴である並列動作に対する検証なども検討されている[92]．

更に，マイクロプログラムの正当性(correctness)を証明する試みも行われている[93)~95]．

### 5.5 課　題

現在開発されているシミュレータは，ほとんどが計算機の設計時に使用することを目的に作成されている．このため，ファームウェア・ユーザからみると，得られる情報が低レベルの記述向きで，また冗長であるなどの問題がある．

従って，今後高水準言語による記述を普及させるためには，それに即した論理チェックの手法を実現する必要がある．それには，ハードウェアの細部の情報をシミュレータやデバッガの内部で処理して，ユーザの要求に応じた情報を出力すること，あるいは現在研究段階にある検証理論の実現を図ることなどが望まれる．

特に，マイクロプログラムの検証については，ソフトウェアの高水準言語とマイクロプログラムとの相違が，検証システムを実現する際にどのような影響を与えるかについて，十分検討する必要がある．

## 6. マイクロプログラム作成システム

### 6.1 システムの構成

3, 4, 5 の各章において，マイクロプログラムの記述とその翻訳および論理チェックの方式について述べたが，実際の作成支援システムはこれらの複合体となっている．従って，各システムにおいて，システム作成の目的や要求される機能に応じて適当な方式が選ばれて実現されている．

本章では，以下，実際に作成されたシステムを中心に，そのシステム概要，記述言語とその処理の方式，最適化の方式，および論理チェックの方式などについて述べる．また，実験結果などが明らかにされているものについては，これを要約して述べる．

### 6.2 STRUM システム

STRUMはマイクロプログラムの検証を行うことを主な目的として作成された構造化マイクロプログラム作成システムで，Burroghs 社の D-Machine に依存する[103),104)]．

〔言語〕 PASCAL 風の言語であるが，マイクロプログラムの証明と構造化のための文を含んでいる．その主なものは，**while, repeat, for, loop, exit, assert,** マクロ，プロセジュア，そしてブロックである．式は D-Machine の単一パスで行える演算に限定され，使用できる変数はレジスタとメモリである．また，表明を記述するための仕様言語 (specification language) として，ISP[51)] を使用している．

```
 1 |
 2 | proc MULTIPLY
 3 |      (PARTIAL_PRODUCT=B, MULTIPLICAND=A1,
                                  MULTIPLIER=A2;)
 4 |
 5 |            / WE WANT THE RESUIT TO BE LESS
               THAN 2**15 /
 6 |      assume
 7 |            0 <= MULTIPLICAND. 0 <= 181,
 8 |            0 <= MULTIPLIER. 0 <= 181;
 9 |                  /       181*181 < 2**15 /
10 |
11 |      conclude
12 |            PARTIAL_PRODUCT=MULTIPLICAND. 0
               mul MULTIPLIER. 0,
13 |            0 <= PARTIAL_PRODUCT <= 2 exp 15,
14 |            MULTIPLICAND=MULTIPLICAND. 0,
15 |            MULTIPLIER=0;
16 |
17 |      PARTIAL_PRODUCT=0;
18 |      begin  MAC 'LSB OF MULTIPLIER=1'
                          =1st (MULTIPLIER) CAM;
19 |
20 |            declare I=DUMMY;
21 |
22 |      for I=0 to 7
23 |                  assert MULTIPLICAND. 0 mul
                                       MULTIPLIER. 0
24 |                  =PARTIAL_PRODUCT
                        +MULTIPLICAND mul
                         MULTIPLIER,
25 |                  MULTIPLIER=MULTIPLIER. 0
                                                shr I,
26 |                  MULTIPLICAND
                           =MULTIPLICAND. 0 shl I,
27 |                  0 <= MULTIPLICAND. 0
                                              <= 181,
28 |                  0 <= MULTIPLIER. 0 <= 181
29 |
30 |            do
31 |                  if 'LSB OF MULTIPLIER=1'
32 |                  then PARTIAL_PRODUCT
                        =MULTIPLICAND
                         +PARTIAL_PRODUCT
33 |                  fi;
34 |
35 |                  MULTIPLICAND
                           =MULTIPLICAND shl 1;
36 |                  MULTIPLIER=MULTIPLIER shr 1
37 |
38 |      rof
39 |
40 |      end;
41 |
42 |      MULTIPLICAND=MULTIPLICAND shr 8
43 |
44 |
45 |
46 | corp;
47 |
48 |
49 |
```

図-15 STRUM による記述例

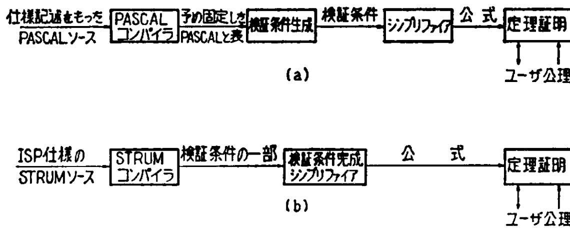



**図-16** 検証システム (a) London の検証システム
(b) STRUM 検証システム

STRUM による乗算の記述例を**図-15** に示す．検証は各ループごとに置かれた表明と入力に対する仮定 (assumption)，および出力に対する結論 (conclusion) をもとに行われる．図-18 では，文番号 23 (assert) から 28 が表明で各ループに対して成立すべき条件を記述している．また，文番号 6 (assume) から 9 が入力の値の範囲を指定しており，文番号 11 (conclude) から 15 で結果を指定している．

〔処理〕 STRUM は D-Machine のアセンブリ言語 TRANSLANG のマイクロコードを生成すると同時に，帰納的表明 (inductive assertion) 法による検証を行う．検証は London の検証システム[105]を応用して**図-16** のように行う．まず，STRUM コンパイラは ISP 仕様の STRUM ソースを入力として，検証条件 (verification condition) の一部を出力する．次に，シンプリファイアは検証条件を入力として公式 (formula) を出力し，この公式に対する定理証明が行われる．システムの実現に当っては既存のソフトウェア工学の手法と処理システムを最大限に活用している．

〔最適化〕 コンパイラは最適化を行わず，出力である TRANSLANG レベルにおいて，人手による最適化を行う．

〔論理チェック〕 検証によって行う．

〔実験〕 図-15 のマイクロプログラムの作成に関するデータを**表-3** に示す．表中，自動最適化とは，コンパイラによって自動的に行えると考えられる最適化の手法を適用した場合である．また，別の例で実験を行ったところ，TRANSLANG で直接記述したものよりも，STRUM によって作成した後で人手による最適化を行ったものの方が効率が良いという結果も得られており，その理由として高級言語で記述したものの方が制御の流れが判りやすく最適化が行いやすいなどが挙げられている．

**表-3** STRUM におけるマイクロコードの比較

| | STRUM コンパイラの出力 | STRUM の出力に自動最適化を行ったもの | STRUM の出力に人手による最適化を行ったもの | アセンブリ言語によって直接コーディングしたもの |
|---|---|---|---|---|
| ナノ命令数 (56ビット／語) | 13 | 9 | 8 | 7 |
| マイクロ命令数 (16ビット／語) | 6 | 4 | 2 | 2 |
| ビット数の合計* | 824 | 568 | 470 | 414 |
| 実行サイクル数 | 96 | 48 | 38 | 30 |

* ナノ命令数 N，マイクロ命令数Mとするとビット数の合計は N×56+M×16 で表される．

### 6.3 MPG システム

MPG は，筆者らが試作した汎用のマイクロプログラム作成システムである．その詳細は既に報告しているため[106)~110]，ここでは概要だけを述べる．

〔言語〕 機械独立な言語によって対象とする計算機とそのマイクロプログラムの記述を行う．広汎な計算機を記述できるように，計算機の記述部では間接符号化あるいは間接機能制御などが記述できるように設計されている．また，マイクロプログラムは記述された計算機のファシリティ名を使用して記述する．従って，MPG は 3.3.2(3) の方式による機械独立な高級言語である．

〔処理〕 コンパイラは，構文解析，最適化前処理，マイクロ命令の構成，および番地付けの 4 つのフェーズに分けて処理を行い，ビットパタン形式のマイクロプログラムを出力する[108]．

〔最適化〕 最適化はマイクロ命令の構成と番地付けの際に行われる．前者は，ローカルな最適化によってコンパクトなマイクロ命令の構成を行う．後者は，コンパクトな制御記憶への自動割付けを行う[111]．

〔論理チェック〕 インタプリンタ方式のシミュレータによって記述された計算機に対するシミュレーションを行う．高級言語による記述に対するシミュレータということで，セグメントごとに自動的に値を出力する変数を決定する機能を持つ．

〔実験〕 水平型マイクロ命令をもつ HITAC 8350 と垂直型マイクロ命令をもつ HP 2100 A に対して，それぞれ 4 つのマイクロプログラムを記述して処理した．この結果，コンパイラの出力と手書きのマイクロプログラムとの比が，マイクロ命令数で平均 1.12：1，実行時間で平均 1.18：1 という結果を得た．

### 6.4 CAS システム

IBM システム/360 の系列のなかで，マイクロプログラム制御を行っているすべての計算機の設計と保守に使用されたシステムである[3),15),101]．また，CAS に

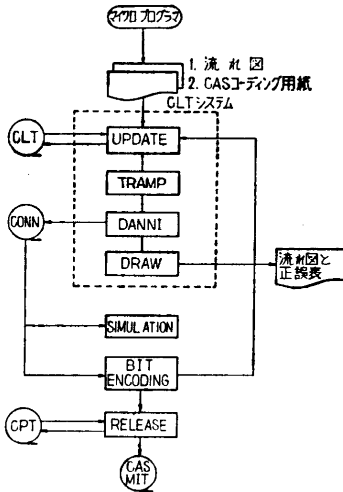



図-17 CAS システムの流れ

よる流れ図形式の記述は現在も行われており，このシステムの優れていることを実証している．

〔言語〕 マイクロプログラムの記述は **3.2.2** に述べたように流れ図形式によって行う．計算機記述言語は計算機の設計技術者を対象として設計されており，論理回路レベルからマイクロ操作のレベルまで多様なレベルでの記述が行える．

〔処理〕 処理システムの概要を図-17 に示す．

UPDATE: 特定のシステムの設計情報をCLTテープ上に記録すると共に，記号形式のマイクロプログラムを更新する．

TRAMP: マイクロプログラムをシミュレーションやアドレス割付けルーチンで使う標準形に変換する．

DANNI: マイクロルーチンの論理的な流れをチェックする．

DRAW: アドレスを割付けて流れ図形式のドキュメントを出力する．

BIT ENCODING: CLT 上のマイクロプログラムをビットパタン形式に変換する．

RELEASE: 制御記憶語のビット・パタンをCPT (CAS physical tape) に記録する．

〔最適化〕 マイクロプログラムの圧縮あるいはアドレスの割付けはプログラマに任されている．

〔論理チェック〕 **5.2.2** で述べた CAS シミュレータによって行う．

### 6.5 PL/MP システム

PL/MP は IBM のワトソン研究所で開発中のコンパイラである[112),113)]．マイクロプロセッサを主な対象としているが，開発された最適化の手法は汎用性があり，また実用的なものである．詳細は明らかにされていないが，これからのマイクロプログラミング・システムに対しても示唆するところは大きいと考える．

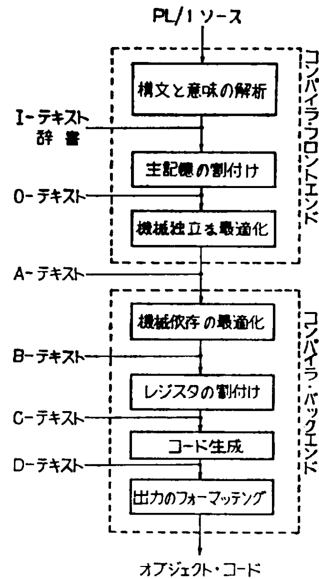



図-18 PL/MP の処理の流れ

〔言語〕 PL/I のような高級言語による記述を行う．この際に，高速レジスタは無数であるとして記述できる点に特徴がある．

〔処理〕 図-18 にシステムの流れを示す．前半は機械独立な変換と最適化を行い，後半は機械依存の最適化と変換を行う．両者のインタフェイスは，レジスタ・トランスファ言語形式の A-テキストである．A-テキストの操作の一部を表-4 に示す．

〔最適化〕 まず，プログラムのデータフロー解析を行い，変数の使用状況を調べる．これをもとに，機械独立な最適化と機械依存の最適化を行う．

特に機械依存の部分では，次のような最適化処理を行う．

(i) レジスタ割振り (register allocation): ソース・プログラムに表れる変数と一時変数を高速のレジスタに割振る．

(ii) アドレス付けのためのコードの最適化 (addressing code optimization): 番地付けのためのコードを移動したり，共通の式を除去したりする．

(iii) コード統合(code consolidation): 簡単な命令

表-4　A-テキストの操作

| オペレータ | オペランド | 操作の内容 |
|---|---|---|
| 基本的な操作 | | |
| ADD | $R_1, R_2, R_3$ | add registers 2 and 3 and store result in 1 |
| ADDC | $R_1, R_2, R_3$ | add with carry |
| BLE | $L, R_1, R_2$ | branch to L if $R_1$ less than or equal to $R_2$ |
| LOAD | $R_1, p$ | load into $R_1$ from location p |
| NOT | $R_1, R_2$ | store into $R_1$ one's complement of $R_2$ |
| CALL | L | call subroutine L |
| • | | |
| • | | |
| 複雑な操作 | | |
| LOADO | $R_1, p, n$ | load with offset n$[R_1 \leftarrow MEM(p+n)]$ |
| LOADIO | $R_1, R_2, n$ | load with offset indirect $[R_1 \leftarrow MEM((R_2)+n)]$ |
| LOADI | $R_1, R_2$ | load indirect |
| ADDI | $R_1, R_2, R_3$ | add indirect |
| ADDIM | $R_1, p, k$ | add immediate $[R_1 \leftarrow p+\text{constant } k]$ |
| ADDIO | $R_1, R_2, R_3, n$ | add indirect with offset |

をいくつか結合して，１つの命令に合成する．

（ⅰ）については，**4.2**（３）に述べた．（ⅱ）と（ⅲ）のアルゴリズムの詳細については，文献 112）を参照されたい．

〔実験〕 Motorola M 6800 マイクロプロセッサに適用した結果，上記の最適化によって PL/MP コンパイラのオブジェクト・コードが 30～40% 改善されたと報告されている．

## 7. ダイナミック・マイクロプログラミングとファームウェア工学

### 7.1 ダイナミック・マイクロプログラミング

制御記憶として高速で書換えのできる記憶装置を使った計算機で，制御記憶を書き換えることによりマイクロプログラムを変更して使用することをダイナミック・マイクロプログラミング（dynamic microprogramming）と呼んでいる[114,115]．

ダイナミック・マイクロプログラミングの発展がファームウェアの応用分野を拡大し，ファームウェア工学の発展を促したことは，始めに述べた通りである．従って，ファームウェア工学について論じるには，その土台となるダイナミック・マイクロプログラミングおよびそれを取り巻く状況について十分検討する必要がある．このような立場から，本章ではダイナミック・マイクロプログラミング方式の特徴とその活用の方法について述べる．

まず，ダイナミック・マイクロプログラミングの特徴を要約すれば，次の通りである[3]．

（ⅰ） エミュレータや故障診断のための機能をオーバレイすることにより，コストを削減できる．

（ⅱ） 計算機の処理時間がある特定の処理に費されている場合には，その処理向きの機能を容易に付け加えることができる．

（ⅲ） システムの故障の検出，診断，修復に活用することによって可用度（availability）の向上が計れる．

（ⅳ） １つの処理装置で種々のアーキテクチャ機能を実現することにより，エミュレーションを行ったり，応用に合ったアーキテクチャを実現することができる．

一方，これを汎用計算機において使用する場合には既に述べてきた，

（ⅰ） 効率の良いマイクロプログラムの作成の困難さ．

（ⅱ） マイクロプログラムの論理チェックの難しさの他にも，次のような問題が考えられる．

（ⅲ） メーカの提供するオペレーティング・システムに手を加える必要がでてくる可能性があり，この場合にはシステムの互換性を失うことになる．

（ⅳ） アーキテクチャの変更がシステム全体のスループットにどれだけ寄与するかについて十分検討する必要があり，場合によっては解析的な評価やシミュレーションを行う必要がある．

（ⅴ） 多重プログラミングにおいては，プログラムの切換えに応じたマイクロプログラムのロードのための所要時間やロードしたマイクロプログラムの管理が問題となる[116),117]．

従って，システム全体をファームウェア・レベルも含めて使いやすく構成するためには，記述とその処理以外にも解決すべき問題は多い．以下，ファームウェア化の効果について述べると共に，機能のファームウェア化を計算機を使って自動的に行う試みについて述べる．

### 7.2 ファームウェア化の効果

あるアルゴリズムを計算機に実現しようとする場合，一般にソフトウェアによれば柔軟だが，実行速度が遅くなり，ハードウェアによれば実行速度は速いが，一度作成されれば変更は容易ではない．ダイナミック・マイクロプログラミングはこの点柔軟性と高速性を兼ね備えているため，機能のファームウェア化による性能の改善が期待できる．

従って，どのような機能をファームウェア化する

かは，本来ソフトウェア，ファームウェア，ハードウェアの三者のトレード・オフに関係する問題であるが[118]，ここではハードウェアはあらかじめ与えられているものとして，ソフトウェアとファームウェアのトレードオフの問題に絞って考える．

一般に，ソフトウェアによる処理機能をファームウェア化した場合には，

（1） 処理速度が向上する．
（2） 所要主記憶量が減少する．
（3） ソフトウェア作成が容易になる．

などの効果が期待される．

（1）の処理速度の向上の原因としては，次のようなことが挙げられる．

（i） いくつかの命令をまとめて1つの命令に合成しているので，主記憶から命令を読み出し，解読する回数が減る
（ii） 命令間でのデータの受渡しのために主記憶への書込みと読出しが行われるが，これをレジスタなどを活用することにより省略できる．

また，（2）に指摘したように，マイクロプログラムによって新しい命令を合成した場合，主記憶のプログラム領域は減少する．しかし，新しい命令のため制御記憶の所要量は増大するので，所要記憶量はこれらの総和となる．

（3）のソフトウェア作成が容易になる原因としては

（i） 出現頻度の高い命令系列が一つになることによるプログラムの簡単化．
（ii） 適当な命令がないため著しく冗長なプログラムとなっている部分の改善．

など従来指摘されているものの他に，

（iii） 構造化のための命令を付け加えることによる，機械語レベルでの構造化プログラムの実現[120]．

などが考えられる．

このようにファームウェア化の効果は種々の点から検討する必要があり，その定量的な評価は容易ではないが，最近では実験による評価なども試みられている[119],[121]．

### 7.3 計算機を用いたファームウェア化

実際にファームウェア化を行って性能を向上するには，上記のようなファームウェア化による性能の向上度を評価し，適当な機能を選択する必要がある．このためには，対象とするプログラムを分析し，静的あるいは動的な構造から命令の出現頻度，実行頻度を求める必要がある．また，向上度を評価するには，処理速度はもちろんのこと所要制御記憶の容量やマイクロプログラムのロード時間なども考慮に入れる必要がある．これらをすべて人手で行うのは大変なため，計算機を使って自動的に行うことが検討されている．

その1つの方法は，命令の合成によって行うものである．これはプログラムの一部がプログラム全体の実行時間の大半を消費する[122]という事実に基づき，実行頻度の高い所から1つの命令にしてしまうものである．この方法は，合成の仕方によって

（i） プログラムの実行をトレースしてデータを収集し，これに基づいて命令の合成を行う．
（ii） コンパイル時にプログラムの動作を予測し，これに基づいて命令の合成を行う．

の2つに分けられる．

（i）はアーキテクチャ・チューニング(architecture tuning) あるいはヒューリスティック・シンセシス(heauristic synthesis)と呼ばれる．実行データに基づくアーキテクチャの再構成が行えるが，トレースのためのオーバヘッドが大きいこと，また繰り返し実行しないと合成の効果が表れないことなどが問題となる．

（ii）はアーキテクチャ再定義(architecture redefinition) と呼ばれ，コンパイラに新たなフェーズを付け加えることによって行える．ファームウェア化の効果を正確に予測するのが困難なこと，あるいはコンパイラのオーバヘッドが大きくなることなどが問題である．

（i）と（ii）は完全自動化を目指す方法であるが，これに対して階層化されたソフトウェアまで含めたチューニングの手法も考えられており，階層間で機能を移動させることから，バーティカル・マイグレーション(vertical migration) と呼んでいる．

以下，具体例について述べる．

（1） アーキテクチャ・チューニング

命令の合成をヒューリスティックに行う方法はAbd-Allaらの提案[123]に始まり，その後手法の改良や実現が行われている[124],[125]．ここでは一例として坂村らの方法[126]を示す．

合成のアルゴリズムは次の通りである．

（a） プログラムの実行をトレースして，ある中間言語に続く次の中間言語の頻度を行列の形に表す（これを遷移マトリクスと呼ぶ）．
（b） 中間言語パターン*がなくなるまで次の（c）から（e）を繰り返す．

(c) 中間言語パターンを認識する.

(d) 発見された中間言語パターンをマイクロプログラムに展開し, 最適化を行う.

(e) 遷移マトリクスを拡張してプログラムをトレースする.

この方法を YHP 2100 計算機に適用した結果, 4～6 回の最適化を繰り返すことによって, 実行速度が2～4 倍に高速化されたことが報告されている.

(2) アーキテクチャ再定義

Rauscher らは命令合成の効果を, 翻訳時に予測する方法を考え, 実験を行っている[127),128)].

この方法では, ブロックおよび各ブロック内での命令系列の実行頻度を予測し, 最もファームウェア化の効果が大きいと思われる系列から順次一つの命令に合成して行く. この方法のポイントは, ファームウェア化の効果を正確に予測する点にあり, このためファームウェア化の手順を定式化するとともに, これをもとに一定の制御記憶容量に対する効果を評価する式を導いている.

実験は PDP-11/05 に対するシミュレータを作成して行った. 対象とした高水準言語は構造化プログラミング言語 ULP である. この結果, 256 語の制御記憶を使用したとき, 25% 以上の実行速度の改善が得られたと述べている.

(3) バーティカル・マイグレーション

アーキテクチャ・チューニングを更に徹底させ, 階層化されたソフトウェアとファームウェアを合わせてアプリケーション向きに調整すれば, チューニングの効果が大きいと予想される. このような目的に沿って開発されたシステムの例として, BUGS[129)] について述べる.

BUGS では, 図-19 に示すようにアプリケーション・レベル, 2つのソフトウェア・レベル, およびファームウェア・レベルの 4 つの層に分けている. 各レベルでのプリミティブ (箱で表されている) の動作は, 順序制御やパラメータの受渡しを行う機能 (写像動作 (mapping action)) と, それ以外の各プリミティブに固有の動作 (実行動作 (execution action)) とに分けられる. 一般的にプリミティブを低レベルに移すと写像動作のオーバヘッドが軽減される. 一例として $P_{13}$ をソフトウェア・レベル 1 からソフトウェア・レベル 0 に垂直に移動すると, 写像のオーバヘッドが

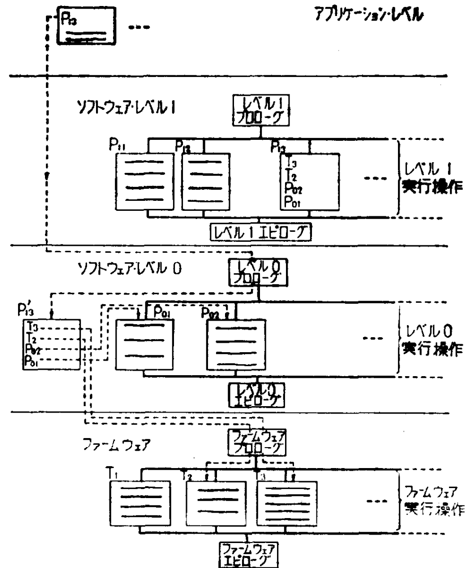



図-19 バーティカル・マイグレーションの例

1,500 μs から 100 μs になる. その要因の 1 つは $P_{13}'$ から呼ばれる $P_{01}$, $P_{02}$ プリミティブが $P_{13}'$ と同じレベルになることである.

このシステムにおいてはレベル 0 とファームウェア・レベルとの間の移動が, 前述した機能のファームウェア化に当る.

実験の結果から,

(i) 移動によって個々の機能の効率は約 10 倍に改善され, そして改善された機能を使ったアプリケーション・プログラムにおいて 2 倍に改善された.

(ii) 特定のアプリケーションには少数の高レベルでの移動が有効であり, 一般的なアプリケーションに対しては多くの低レベルでの移動が有効である.

などが得られている.

### 7.4 課　題

ダイナミック・マイクロプログラミングを活用するには, まず書き換え可能な制御記憶 (WCS) が使用できなければならないが, 現在ではまだこのような計算機は一般に普及するまでに至っていない. これには, 種々の原因が考えられる. まず, 提供する側からはファームウェア・レベルで計算機を開放した場合に起きる可能性のあるトラブルに対処するのが大変だということが挙げられる. また, 使う側からみると, 仮に WCS が使えるとしても, 記述処理システムを含むフ

* 順序づけられたマイクロルーチンの組を中間言語と定義する. 中間言語の集合は, 一定の認識基準によって, 続いて起る頻度が高いと判定されたとき, パターンとして認められる[126)].

ァームウェアの支援システムが十分でなく，使いにくいといった理由もある．

しかし，ファームウェア化による効果が種々期待され，また実証されている現在，ソフトウェア，ファームウェア，ハードウェアの三位一体となったファームウェア指向システムを推進し普及することが望まれる．このことが，種々の問題領域へのダイナミック・マイクロプログラミングの応用をもたらし，更に使いやすいファームウェア指向システム実現への契機となるであろう．

## 8. 今後の課題

最後にファームウェア工学のこれからの課題を列挙し，全体のまとめとしたい．

（1） 機械独立な高水準言語の実現

広く受け入れられる機械独立な高水準言語の実現は，ファームウェア工学の基本的な課題である．記述言語のレベルも種々のものが考えられ，ファームウェア・ユーザに対するアーキテクチャをどこに設定するかは，オブジェクト・マイクロコードの効率とも関連して重要な問題である．また，構造化プログラミング，プログラムの検証などのソフトウェア工学の手法を取り入れることも重要な課題である．

処理に関しては，構文解析や機械独立な最適化処理と，機械依存の最適化およびマイクロコード生成を明確に分けて行う方式が一般的になりつつある．

（2） 効率の良いマイクロプログラムの生成

マイクロプログラムの最適化に関しては，

（i） 機械独立な手法と機械依存の手法を明確にする．

（ii） ハードウェアの特徴を活かした，機械依存の最適化手法を個々の計算機に対して確立する．

の2点について，今後検討する必要がある．特に機械ごとに（ii）を推し進めて行くことが，汎用性と実用性の高い最適化法の実現に結びつくと考えられる．

また，レジスタの自由な使用など，実際にあるハードウェア以上のアーキテクチャをユーザに提供する場合には，これらを効率良く対応づけるための手法が必要となる．

（3） 高水準言語レベルでの論理チェック

ハードウェアに密着し，並列動作の多いマイクロプログラムの論理チェックを高水準言語レベルで行うには，何らかの工夫が必要となる．マイクロプログラムの検証理論もこのための有力な手法である．

（4） ファームウェア指向型システムの開発

現在ファームウェア・レベルでの使用はソフトウェアの命令を拡張する形で特殊モードとして行われることが多く，支援のためのソフトウェア，ハードウェアの体制も十分とは言えない．このため，今後オペレーティング・システムを含めても，ソフトウェアとファームウェアとの壁を意識せずに使用できるようなシステム作りが望まれる．

また，特にマルチプログラミングのもとでファームウェアを使用する場合には，保護あるいは共用の問題を解決しなければならない．ファームウェア・レベルでの使用はハードウェアの制御に直接かかわるだけに使い方によっては他のユーザへの影響も大きく，RAS の向上も課題の一つである．

（5） ユーザ・マイクロプログラマブル計算機の開発

多くのマイクロプログラム制御計算機では，命令セットをサポートして所期の命令ミックスを得ることに重点が置かれているため，特殊な機能をハードウェア化して高速化を図ることが少なくない．例えば，乗除算用のハードウェアなどがその例であるが，これらは通常のマイクロプログラムではほとんど使用できない．このためマイクロ操作機能を一様でかつ基本的なものとすることが望まれるが，このような要求は LSI の発展ともマッチしている[130]．ファームウェア・レベルでの定数の使用，ビット処理などの非数値処理機能，多倍長のシフト機能などの機能も強化することが望まれる．

また，構造化マイクロプログラミング（structured microprogramming)[132),133] を実現するには，そのための順序制御機能をハードウェアで備えることなども必要となろう．

（6） マイクロプロセッサの発展の影響

マイクロプログラマブル・マイクロプロセッサを使用すれば，ユーザの要求に沿った，ソフトウェア，ファームウェア，ハードウェア一体のシステムを低価格で実現できる可能性がある．ハードウェアの集積化の傾向もこのための有利な条件である．

これらのマイクロプロセッサ開発のための支援システムにも，ファームウェア工学の手法が必要となろう．

（7） マイクロプログラムの可搬性

広く受け入れられる機械独立な高級言語がなかなか実現に至らないため，マイクロプログラムを可搬にすることは難しい問題である．これは機械の特徴を生か

して効率を上げるというマイクロプログラムの本質とのジレンマとも言える．しかし，今後ファームウェアが種々の領域に応用されその作成量も増えることが予想されるため，ある機械用に作成したマイクロプログラムを他の機械に移植する作業を効率良く行うことを考えておかねばならない．特に，同じ系列の機械への移植は必要度も大きいし，またハードウェアの類似性も期待できるため，実現性が高い．

（8） アーキテクチャの自動再構成

実現されればユーザに負担をかけずに自動的に効率の良いシステムに再構成できるという利点がある．このためファームウェア化の手法として有望ではあるが，自動化するためにはまだ解決すべき問題は多い．

（9） ファームウェア向きアルゴリズムの開発

ソフトウェアで実現された機能をそのままファームウェア化するのでなく，最初からファームウェア向きのアルゴリズムを使用すれば，更に効率が向上することが期待できる[6]．いろいろな問題領域においてその可能性は残されており，新たなアルゴリズムの開発は今後の課題である．

（10） 2レベル・マイクロプログラミング方式の活用

**2.2.1** で言及したこの方式は一部の機械で実用化されているが[4),6),19)]，水平型マイクロ命令の並列動作による効率の良さと，垂直型マイクロ命令の作成のしやすさとを両立させられるため，今後の発展が期待される方式である[137]．

## 9. むすび

ファームウェア工学に関する研究やシステム開発は緒についたばかりで，まだ概念や問題点も十分に整理されてはいない．また，現状ではマイクロプログラムの作成量も大規模なソフトウェアと比べればまだそれ程多くはなく，ファームウェア工学の必要性も十分に認識されているとは言えない．

しかし，集積化技術の進歩に伴うハードウェア機能の基本化，モジュール化の傾向と，計算機の応用範囲の拡大に伴うソフトウェアの多様化の傾向という現状を観察すれば，ソフトウェアとハードウェアのギャップを埋める手段としてファームウェアの重要性は今後ますます大きくなることが予想される[130),131)]．また，現にこのような試みは，高級言語処理やオペレーティング・システムのファームウェア化といった形で実現されている[134]．

このような展望に立って考えると，ファームウェアの分野においても，大規模化に伴う「ファームウェアの危機」を招かないためにも，その問題点を整理し，ファームウェア工学を中心とした実効性のある対策を十分に講じる必要があろう．

謝辞 本稿の執筆に当り，御意見をいただいた京都大学萩原宏教授ならびに福井大学渡辺勝正教授に感謝する．

## 参 考 文 献

[マイクロプログラミング全般]

1) Wilkes, M. V.: The Best Way to Design an Automatic Calculating Machine, Report of Manchester University Computer Inaugural Conference, pp. 16-18 (1951).
2) Opler, A.: Fourth Generation Software, Datamation, Vol. 13, No. 1, pp. 22-24 (1967).
3) Husson, S. S.: Microprogramming: Principles and Practices, Englewood Cliffs, N. J.: Prentice-Hall (1970).
4) Agrawala, A. K. and Rauscher, T. G.: Foundations of Microprogramming, New York: Academic (1976).
5) Agrawala, A. K. and Rauscher, T. G.: Microprogramming: Perspective and Status, IEEE Trans. Comput., Vol. C-23, pp. 817-837 (1974).
6) 萩原：マイクロプログラミング，p. 343，産業図書．東京 (1977).
7) Ramamoorthy, C. V. and Tsuchiya, M.: A Study of User-Microprogrammable Computers, Proc. SJCC, 1970, pp. 165-181 (1970).
8) Lehman, M. M.: Microprogramming Trend Considered Dangerous, Comm. ACM, Vol. 18, No. 6, pp. 358-360 (1975).
9) Jones, L. H.: A Survey of Current Work in Microprogramming, Computer, Vol. 8, No. 8, pp. 33-38 (1975).
10) Jones, L. H. and Merwin, R. E.: Trends in Microprogramming: A Second Reading, IEEE Trans. Comput., Vol. C-23, No. 8, pp. 754-759 (1974).
11) マイクロプログラミング特集号，情報処理，Vol. 14, No. 6 (1973).

[ファームウェア工学]

12) Shriver, B. D.: Firmware: The Lessons of Software Engineering, Computer, Vol. 11, No. 5, pp. 19-20 (1978).
13) Davidson, S. and Shriver, B. D.: An Overview of Firmware Engineering, ibid, pp. 21-33 (1978).
14) Patterson, D. A.: An Approach to Firmware Engineering, Proc. 1978 Nat. Comput. Conf., pp. 643-647 (1978).

[マイクロプログラム記述用アセンブリ言語]

15) Buckingham, B. R. S., et al.: The Control Automation System, 6th Annual Symposium on Switching Circuit Theory and Logical Design (1965).

16) Clark, R. K.: Mirager, the "Best-Yet" Approach for Horizontal Microprogramming, Proc. of the ACM National Conference, pp. 554-571 (1972).

17) Dubbs, E. W. et al.: A Microprogram Design System Translator, CompCon 1972 Dig. Papers, pp. 95-98 (1972).

18) Hornbuckle, G. D. and Ancona, E. I.: The LX-1 Microprocessor and Its Application to Real-Time Signal Processing, IEEE Trans. Comput., Vol. C-19, pp. 710-720 (1970).

19) Reigel, E. W. et al.: The Interpreter-A Microprogrammable Building Block System, Proc. SJCC, pp. 705-723 (1972).

20) Evans, R. H. et al.: Design of Assembly Level Language for Horizontal Encoded Microprogrammed Control Unit, 7th Annu. Workshop on Microprogramming, Preprints, pp. 217-224 (1974).

21) Rauscher, T. G. and Agrawala, A. K.: On the Syntax and Semantics of Horizontal Microprogramming Languages, Proc. of the ACM Nat. Conf., pp. 52-56 (1973).

22) Dewitt, D. J. et al.: A Microprogramming Language for the B-1726, 6th Annu. Workshop on Microprogramming, Preprints, pp. 21-29 (1973).

23) Laws, B. A. Jr.: Microbe: A Self Commenting Microassembler, 10th Annu. Workshop on Microprogramming, Preprints, pp. 61-65 (1977).

24) Hodges, B. C. and Edwards, A. J.: Support Software for Micro Program Development, SIGMICRO Newsletter, Vol. 5, No. 4, pp. 17-24 (1975).

25) Persson, W.: A Microprogram Generator for The VARIAN V73, SIGMICRO Newsletter, Vol. 8, No. 4, pp. 14-20 (1977).

26) Rauscher, T. G.: Towards a Specification of Syntax and Semantics for Languages for Horizontally Microprogrammed Machines, Proc. of ACM SIGPLAN-SIGMICRO Interface Meeting, pp. 98-112 (1973).

27) Rauscher, T. G.: Microprogramming the AN/UYK-17 (XB-1) (V) Signal Processing Element Signal Processing Arithmetic Unit, SIGMICRO Newsletter, Vol. 5, No. 2, pp. 29-63 (1974).

28) Sawai, T. et al.: A Microprogramming Language Translator Generator, Proc. Euromicro 1977, pp. 117-122 (1977).

29) 牧之内, 他: MLTG-マイクロプログラミング・ランゲージ・トランスレータ・ジェネレーター: LALR パーサの一つの応用例, 情報処理, Vol. 20, No. 1, pp. 17-25 (1979).

30) HITAC 8350 RCM 処理システム・マニュアル, 日立製作所 (1973).

[マイクロプログラム記述用高水準言語]

31) Eckhouse, R. H.: A High-Level Microprogramming Language (MPL), State University of New York at Buffalo, Department of Comput. Science, Report 1-71-MU (1971).

32) Eckhouse, R. H.: A High Level Microprogramming Language (MPL), Proc. SJCC, pp. 169-177 (1971).

33) Hattori, M. et al.: MPGS-A High Level Language for Microprogram Generating System, Proc. ACM Annu. Conf., pp. 572-581 (1972).

34) Yamamoto, M. et al.: A Microprogrammed Computer Design and Evaluation System, First USA-Japan Comput. Conf., pp. 139-144 (1972).

35) Lawson, H. W. Jr and Blomberg, L.: The DataSaab FCPU Microprogramming Language, Proc. ACM SIGPLAN-SIGMICRO Interface Meeting, pp. 86-97 (1973).

36) Oestreicher, D. R.: A Microprogramming Language for the MLP-900, ibid, pp. 113-120 (1973).

37) Noguez, G. L. M.: Desgin of a Microprogramming Language, 6th Annu. Workshop on Microprogramming, Preprints, pp. 145-155 (1973).

38) Ramamoorthy, C. V. and Tsuchiya, M.: A High-Level Language for Horizontal Microprogramming, IEEE Trans. Comp., Vol. C-23, pp. 791-801 (1974).

39) Lloyd, G. R. and van Dam, A.: Design Considerations for Microprogramming Languages, Proc. Nat. Comput. Conf., Vol. 43, pp. 537-543 (1974).

40) Berndt, H.: Microprogramming with Statements of Higher-Level Languages, 5th Annu. Workshop on Microprogramming, pp. 76-80 (1972).

41) Lloyd, G. R.: PUMPKIN- (Another) Microprogramming Language, SIGMICRO Newsletter, Vol. 5, No. 1, pp. 45-76 (1974).

42) 山崎, 他: QA-1 用ハイレベル・マイクロプログラミング言語とその処理, 情報処理学会第 19 回全国大会予稿, pp. 175-176 (1978).

43) 重松, 他: マイクロプログラミング言語 MPL 200 とその最適化技法, 情報処理, Vol. 19, No. 8, pp. 749-757 (1978).

44) 三上, 房岡: ファームウェア・ジェネレータ・

システムの構成, 情報処理学会計算機アーキテクチャ研究会資料 32-4 (1978).
45) 三上, 房岡: ファームウェア・ジェネレータ・システムの性能評価, 情報処理学会第 19 回全国大会予稿, pp. 211-212 (1978).

[記述とその処理の方式に関する考察]

46) DeWitt, D. J.: Extensibility-A New Approach for Designing Machine Independent Microprogramming Languages, 9th Annu. Workshop on Microprogramming, pp. 33-41 (1976).
47) Tirrell, A. K.: A Study of the Application of Compiler Techniques to the Generation of Micro-Code, Proc. of ACM SIGPLAN-SIGMICRO Interface Meeting, pp. 67-85 (1973).
48) Agrawala A. K.: and Rauscher, T. G.: The Application of Programming Language Techniques to the Design and Development of Microprogramming Languages, 6th Annu. Workshop on Microprogramming, Preprints, pp. 134-138 (1973).
49) Mallet, P. W. and Lewis T. G.: Considerations for Implementing a High Level Microprogramming Language Translation System, Computer, Vol. 8, No. 8, pp. 40-52 (1975).
50) Malik, K. and Lewis, T.: Design Objectives for High Level Microprogramming Languages, 11th Annu. Workshop on Microprogramming, pp. 154-160 (1978).

[計算機記述言語]

51) Bell, C. G. and Newell, A.: Computer Structures: Readings and Examples, New York: McGraw-Hill (1971).
52) Chu, Y.: Computer Organization and Microprogramming, Englewood Cliffs, N. J.: Prentice-Hall (1972).

[コンパイラ]

53) Aho, A. V. and Ullman, J. D.: Principles of Compiler Design, Addison-Wesley (1977).

[マイクロプログラムの最適化]

54) Agerwala, T.: Microprogram Optimization: A Survey, IEEE Trans. Comput., Vol. C-25, pp. 962-973 (1976).
55) Agerwala, T.: A Survey of Techniques to Reduce/Minimize the Control Part/ROM of a Microprogrammed Digital Computer, 7th Annu. Workshop on Microprogramming, Preprints, pp. 91-97 (1974).
56) 松崎: 実用化の試み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, pp. 76-91 (1978).
57) Das, S. R. et al.; On Control Memory Minimization in Microprogrammed Digital Computers, IEEE Trans. Comput., Vol. C-22, pp. 845-848 (1973).
58) Grasselli, A. and Montanari, U.: On the Minimization of Read-Only Memories in Microprogrammed Digital Computers, ibid, Vol. C-19, pp. 1111-1114 (1970).
59) Jayasri T. and Basu, D.: An Approach to Organizing Microinstructions Which Minimizes the Width of Control Store Words, ibid, Vol. C-25, pp. 514-521 (1976).
60) Montangero, C.: An Approach to the Optimal Specification of Read-Only Memories on Microprogrammed Digital Computers, ibid, Vol. C-23, pp. 375-389 (1974).
61) Robertson, E. L.: Microcode Bit Optimization is NP-Hard, SIGMICRO Newsletter, Vol. 8, No. 2, pp, 40-43 (1977).
62) Kleir, R. L. and Ramamoorthy, C. V.: Optimization Strategies for Microprograms, IEEE Trans. Comput., Vol. C-20, pp. 783-794 (1971).
63) Astopas, F. and Plukas, K. I.: Method of Minimizing Computer Microprograms, Automat. Contr., Vol. 5, pp. 10-16 (1971).
64) Tsuchiya, M. and Gonzalez, M. J.: Toward Optimization of Horizontal Microprograms, IEEE Trans. Comput., Vol. C-25, No. 10, pp. 992-999 (1976).
65) Tsuchiya, M. and Gonzales, M. J. Jr.: An Approach to Optimization of Horizontal Microprograms, 7 th Annu. Workshop on Microprogramming, Preprints, pp. 85-90 (1974).
66) Yau, S. S. et al.: On Storage Optimization of Horizontal Microprograms, ibid, Preprints, pp. 98-106 (1974).
67) Dasgupta, S. and Tarter, J.: The Identification of Maximal Parallelism in Straight-Line Microprograms, IEEE Trans. Comput., Vol. C-25, No. 10, pp. 986-992 (1976).
68) Dasgupta, S.: Comments on "The Identification of Maximal Parallelism in Straight-Line Microprograms, ibid, Vol. C-27, No. 3, pp. 285-286 (1978).
69) Tokoro, M. et al.: An Approach to Microprogram Optimization Considering Resource Occupancy and Instruction Formats, 10th Annu. Workshop on Microprogramming, Preprints, pp. 92-108 (1977).
70) Wood, G.: On the Packing of Micro-operations into Microinstruction Words, 11th Annu. Workshop on Microprogramming, pp. 51-55 (1978).
71) Dasgupta, S.: Parallelism in Loop-Free Microprograms, Proc. IFIP Congress 77, pp. 745-750 (1977).
72) Tokoro, M. et al.: A Technique of Global

Optimization of Microprograms, 11th Annu. Workshop on Microprogramming, Preprints, pp. 41-50 (1978).

73) Tanaka, T. et al.: Proposal on Efficient Address Allocation Algorithm for Horizontal Microprograms, ibid, Preprints, page 40 (1978).

74) Wakerly, J. F. et al.: Placement of Microinstructions in a Two-Dimentional Address Space, 8th Annu. Workshop on Microprogramming, pp. 46-51 (1975).

75) Bondi, J. O. and Stigall, P. D.: Designing HMO, an Integrated Hardware Microcode Optimizer, 7th Annu. Workshop on Microprogramming, pp. 268-276 (1974).

76) DeWitt, D. J.: A Control Word Model for Detecting Conflicts Between Microprograms, 8th Annu. Workshop on Microprogramming, Preprints, pp. 6-12 (1975).

77) Tsuchiya, M. and Jacobson, T.: An Algorithm for Control Memory Minimization, ibid, pp. 18-25 (1975).

78) Blain, G. et al.: A Compiler for the Generation of Optimized Microprograms, SIGMICRO Newsletter, Vol. 5, No. 3, pp. 49-67 (1974).

79) Ramamoorthy, C. V. and Gonzalez, M. J.: A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs, Proc. 1969 FJCC, Vol. 35, pp. 1-15 (1969).

80) Tomasulo, R. M.: An Efficient Algorithm for Exploiting Multiple Arithmetic Units, IBM Journal, pp. 25-33 (1967).

81) Halatsis, C. and Gaintanis, N.: On the Minimization of the Control Store in Microprogrammed Computers, IEEE Trans. Comput., Vol. C-27, No. 12, pp. 1189-1192 (1978).

[マイクロプログラムの論理チェック]

82) Zucker, M. S.: LOCS: An EDP Machine Logic and Control Simulator, IEEE Trans. Elec. Comput., Vol. EC-14, pp. 403-416 (1965).

83) Young, S.: A Microprogram Simulator, Proc. of DA Workshop, pp. 68-81 (1971).

84) Petzold, R. et al.: A Two Level Microprogram Simulator, 7th Annu. Workshop on Microprogramming, Preprints, pp. 41-47 (1974).

85) Vickery, C.: Software Aids for Microprogram Development, ibid, Preprints, pp. 208-211 (1974).

86) Gasser, M.: An Interactive Debugger for Software and Firmware, 6th Annu. Workshop on Microprogramming, Preprints, pp. 113-119 (1973).

87) Carter, W. C. et al.: Microprogram Verification Considered Necessary, Proc. 1978 Nat. Comput. Conf., pp. 657-664 (1978).

88) Dijkstra, E. W.: NATO Conference on Software Engineering, Conference Report (1969).

89) van Mierop, D. et al.: Verification of the FTSC Microprgram, 11th Annu. Workshop on Microprogramming, Preprints, page 118 (1978).

90) Budkowski, S. and Dembinski, P.: Firmware versus Software Verification, ibid, Preprints, pp. 119-127 (1978).

91) Dembinski, P. and Budkowski, S.: AN Introduction of the Verification-Oriented Microprogramming Language 'Middle', ibid, Preprints, pp. 139-143 (1978).

92) Dasgupta, S.: Towards a Microprogramming Language Schema, ibid, Preprints, pp. 144-153 (1978).

93) Ramamoorthy, C. V. and Shankar, K. S.: Automatic Testing for the Correctness and Equivalence of Loopfree Microprograms, IEEE Trans. Comput., Vol. C-23, No. 8, pp. 768-782 (1974).

94) Maurer, W. D.: Some Correctness Principles for Machine Language Programs and Microprograms, 7th Annu. Workshop on Microprogramming, Preprints, pp. 225-234 (1974).

95) Joyner, W. H. Jr., et al.: Automated Proofs of Microprogram Correctness, 9th Annu. Workshop on Microprogramming, Preprints, pp. 51-55 (1976).

96) Bouricius, W. G.: Procedure for Testing Microprograms, ibid, pp. 235-240 (1974).

97) Leeman, G. B. Jr.: Some Problems in Certifying Microprograms, IEEE Trans. Comput., Vol. C-24, No. 5, pp. 545-553 (1975).

98) 元岡，新開：μプログラム計算機用シミュレータ，情報処理学会設計自動化研究会資料 74-9 (1973).

99) 倉地：マイクロプログラムの記述とシミュレーション，情報処理，Vol. 14, No. 6, pp. 397-403 (1973).

100) 倉地：コンピュータの設計自動化（2），情報処理，Vol. 18, No. 7, pp. 684-692 (1977).

101) 論理設計の自動化システムに関する研究——第1報—日本電子工業振興協会 (1973).

102) 論理設計の自動化システムに関する研究——第2報—日本電子工業振興協会 (1974).

[マイクロプログラム作成システム]

103) Patterson, D. A.: Strum: Structured Microprogram Development System for Correct Firmware, IEEE Trans. Comput., Vol. C-25, No. 10, pp. 974-985 (1976).

104) Patterson, D. A.: Verification of Microprograms, Comput. Science Dep., School of Engineering and Applied Science, UCLA, Report

UCLA-ENG-7707 (1977).
105) Good, D. I. et al.: An Interactive Program Verification System, IEEE Trans. Software Engineering, Vol. 1, No. 1, pp. 59-67 (1975).
106) Baba, T.: A Microprogram Generating System-MPG, Proc. IFIP Congress 77, pp. 739-744 (1977).
107) 馬場, 他: マイクロプログラム記述言語: MPGL, 情報処理, Vol. 18, No. 6, pp. 558-565 (1977).
108) 馬場, 他: MPG マイクロプログラム・コンパイラ, 情報処理, Vol. 19, No. 1, pp. 16-25 (1978).
109) 馬場, 他: MPG マイクロプログラム・シミュレータ, 情報処理, Vol. 19, No. 5, pp. 412-420 (1978).
110) 馬場, 萩原: マイクロプログラムの自動作成について, 情報処理, Vol. 19, No. 1, pp. 61-69 (1978).
111) Baba, T. and Hagiwara, H.: A Machine-Independent Microprogram Optimization Algorithm for Generating Space-Efficient Microprograms, 情報処理学会第19回全国大会予稿, pp. 205-206 (1978).
112) Tan, C. J.: Code Optimization Techniques for Micro-Code Compilers, Proc. 1978 Nat. Comput. Conf., pp. 649-664 (1978).
113) Tan, C. J.: Register Assignment Algorithm for Optimizing Compilers, IBM Research Report RC 6481 (1977).

[ダイナミック・マイクロプログラミングとその応用]

114) Tucker, A. B. and Flynn, M. J.: Dynamic Microprogramming: Processor Organization and Programming, Comm. ACM, Vol. 14, No. 4, pp. 240-250 (1971).
115) Cook, R. W. and Flynn, M. J.: System Design of a Dynamic Microprocessor, IEEE Trans. Comput., Vol. C-19, No. 3, pp. 213-222 (1970).
116) Wilkes, M. V.: The Use of a Writable Control Memory in a Multiprogramming Environment, 5th Annu. Workshop on Microprogramming, pp. 62-65 (1972).
117) Guha, R. K.: Dynamic Microprogramming in a Time Sharing Environment, 10th Annu. Workshop on Microprogramming, pp. 55-60 (1977).
118) Barsamian H. and DeCegama, A.: Evaluation of Hardware-Firmware-Software Trade-offs with Mathematical Modeling, Proc. 1971 SJCC, pp. 151-161 (1971).
119) Davidson, S.: A Case Study of the Migration of an Algorithm across Software-Firmware Boundaries, SIGMICRO Newsletter, Vol. 8, No. 4, pp. 24-33 (1977).
120) 渡辺: マイクロプログラムによる GOTO なしアセンブリ言語について, 構造的プログラミングとその経験シンポジウム報告集 (1975).
121) 坂村, 他: ファームウェア化による計算機性能改善度の予測法, 信学論, Vol. J61-D, No. 9, pp. 711-718 (1978).
122) Knuth, D. E.: An Empirical Study of FORTRAN Programs, Software-Practice and Experience, Vol. 1, pp. 105-133 (1971).
123) Abd-Alla, A. M. and Karlgaard, D. C.: Heauristic Synthesis of Microprogrammed Computer Architecture, IEEE Trans. Comput., Vol. C-23, No. 8, pp. 802-807 (1974).
124) Abd-Alla, A. M. and Moffett, L. H.: On-line Architecture Tuning Using Microcapture, Proc. 3rd Annu. IEEE Symp. Computer Architecture, pp. 165-171 (1976).
125) El-Ayat, K. A. and Howard, J. A.: Algorithms for a Self-tuning Microprogrammed Computer, 10th Annu. Workshop on Microprogramming, pp. 85-91 (1977).
126) 坂村, 相磯: 計算機アーキテクチャの自動最適化に関する考察, 信学論, Vol. J60-D, No. 11, pp. 929-936 (1977).
127) Rauscher, T. G. and Agrawala, A. K.: Developing Application Oriented Computer Architectures on General Purpose Microprogrammable Machines, Proc. Nat. Comput. Conf., pp. 715-720 (1976).
128) Rauscher, T. G. and Agrawala, A. K.: Dynamic Problem-Oriented Redifinition of Computer Architecture via Microprogramming, IEEE Trans. Comput., Vol. C-27, No. 11, pp. 1006-1-14 (1978).
129) Stockenberg, J. and van Dam, A.: Vertical Migration for Performance Enhancement in Layered Hardware/Firmware/Software Systems, Computer, Vol. 11, No. 5, pp. 35-50 (1978).

[その他]

130) 元岡: 最近の計算機アーキテクチャの動向, 情報処理, Vol. 18, No. 4, pp. 310-316 (1977).
131) Fiala, E. R.: The Maxc Systems, Computer. Vol. 11, No. 5, pp. 57-67 (1978).
132) Jones, L. H.: Microinstruction Sequencing for Structured Microprogramming, 7th Annu. Workshop on Microprogramming, pp. 277-289 (1974).
133) Hawk, D. R. and Robinson, D. M.: A Microinstruction Sequencer and Language Package for Structured Microprogramming, 8th Annu. Workshop on Microprogramming, pp. 69-75 (1975).
134) Brown, G. E. et al.: Operating System Enhancement through Firmware, 10th Annu. Workshop on Microprogramming, pp. 119-133 (1977).
135) 田村, 他: マイクロプログラムの局所最適化に関する考察, 信学論, Vol. J62-D, No. 3, pp. 185-192 (1979).
136) 滝塚, 他: マイクロプログラムの広域最適化に関する考察, 信学論, Vol. J62-D, No. 3, pp. 193-200 (1979).
137) 馬場, 小林: 2レベル・マイクロプログラミング方式による非数値処理指向型計算機のアーキテクチャ, 信学技報 EC 79-5 (1979).

(昭和54年3月30日受付)