論理装置の汎用機能シミュレーションシステム—LFM†

上森 明†† 新開慶武††† 元岡 達††

近年，マイクロプログラム制御方式は，広範囲の論理装置に用いられるようになった．LFM システムは，このような論理装置の機能を正確かつ簡潔な形で記述することを可能にし，そのまま機能シミュレータの入力となる言語である．並列動作部分の記述が容易で，時間関係を必要な精度で記述可能なこと，機能を中心にした記述法，などの特徴を持っている．本論文では，LFM 言語の設計思想や記述法，および機能シミュレータとしての構成について述べた．記述能力やシミュレーションの速度を，2種類のマイクロプロセッサを例題として評価し，その結果，種々の機能レベルで記述可能なこと，論理設計援助のためのシミュレータとして，充分な速度を有することが判明した．

1. はじめに

LFM 言語 (Language For Microprogram Computer) は，マイクロプログラム化された装置を主な対象として，その機能を記述する言語である．M. V. Wilkes によって提案されたマイクロプログラム制御方式は，広範囲の論理装置に用いられるようになった．LFM は，このような論理装置の機能を正確かつ簡潔な形で記述することを可能にし，そのまま機能シミュレータの入力となる言語である．本論文では，最初に LFM 言語の設計概念とその仕様を説明する．次に LFM 言語による記述法，およびシミュレータの構成について述べる．最後に，2種類のマイクロプロセッサの記述例とそのシミュレーション結果を示し，シミュレータの性能を評価する．

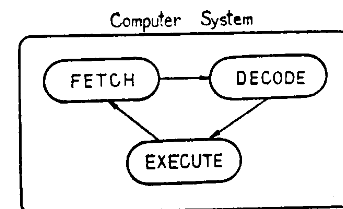
2. LFM 言語

2.1 LFM 言語の特徴

ハードウェアの論理機能を記述する言語として，LFM 言語は，以下のような特徴を持っている．

（1） 処理機能のモジュール化

LFM で定義しているモジュールとは，計算機システム内の論理機能のまとまりを意味し，互いに同期または非同期的に信号を交換して，並列に動作することができる．モジュールは，絶対時刻を示すシステム全体に共通な時計とは独立に，各自固有の時計を持つ．モジュールは，active と dead の２つの状態を持ち，active のときに，各自の時計に合わせて，論理機能が



LFMによる表現

```
LFM;
  リソース宣言
  MODULE FETCH;
        :
  MODULE DECODE;
        :
  MODULE EXECUTE;
        :
END_LFM;
```

図 1 モジュール概念を用いたマシンの記述
Fig. 1 Machine description with module concept.

動作する．したがって，ある時点でのシステム全体の動作は，その時点での active なモジュールの機能の集合に等しい．モジュールによるシステム表現の例を図１に示す．

（2） 非同期システムの記述が可能

同期システムでは，同時に並行して実行されるオペレーションの時間間隔は，各オペレーションの実行時間の最大値以上なので，固定の時間間隔を割り当てればよいが，非同期システムでは，これが一定とならないので問題となる．このため，LFM ではオペレーションの開始時刻だけでなく，終了時刻をも記述できるようになっている（図２参照）．

（3） 順序（時間）関係が必要な精度で記述可能

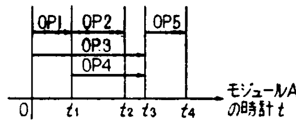
論理機能の動作の開始時刻と終了時刻は，任意の整定数で指定できる．

（4） マイクロプログラム制御以外の部分も記述が容易

マイクロプログラム化された論理装置を主な対象と

† Functional Logic Simulation System for Logic Device—LFM by AKIRA UEMORI, YOSHITAKE SHINKAI, and TOHRU MOTO-OKA (Faculty of Engineering, Tokyo University).
†† 東京大学工学部電気工学科
††† 富士通(株)ＯＳ部

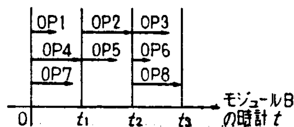
a）非同期的なモジュールA



［記述例］

MODULE A;
　ELM (0-t_1); OP 1;
　ELM (0-t_3); OP 3;
　ELM (t_1-t_2); OP 2;
　ELM (t_1-t_3); OP 4;
　ELM (t_3-t_4); OP 5; STOP;

b）同期的なモジュールB



［記述例］

MODULE B;
　ELM (0-t_1); OP 1; OP 4; OP 7;
　ELM (t_1-t_2); OP 2; OP 5;
　ELM (t_2-t_3); OP 3; OP 6; OP 8; STOP;

図 2 同期および非同期システムの記述例
Fig. 2 Examples of synchronous and asynchronous system.

```
SEL (2) MIR (1-2);
        C=A & B;
   /    C=A|B;
   /    C=ADD (A, B);
   /    C=¬A;
SEND;
```

図 3 多分岐機能の表現
Fig. 3 The expression of multi-way branch.

はしているが，マイクロ命令のフェッチ，デコード，実行という3つの段階を特に意識していないので，拡張性・融通性に富む．割込み機能など，かなり独立に動く部分を含めて，システム全体の記述ができる．

（5） 多分岐（デコード）機能の実現

SEL (select) 文をシンタックス中に含めることによって，デコード機能の記述が容易となり，かつリストも読みやすくなる（**図3**参照）．

（6） 記述レベルの階層性

LFM では，入力と出力の関係を表わすブラックボックスとして関数を宣言できる．したがって，機能をまとめて関数とすることにより，LFM のプログラマは，ゲートレベルからマイクロプログラムや機械語のレベルまで，任意のレベルで記述可能である．これは，シミュレータとして利用するときにシミュレーションの目的に応じて，記述レベルを自由に変えられることを意味する．

（7） PL/I 風の言語

言語仕様が PL/I に似ており，PL/I に慣れたユーザには記述が容易である．

（8） 並列に処理される部分を正確かつ容易に記述可能

各モジュールは自分自身の時計を持ち，互いに独立に並列に動きうる．LFM は，WAIT 文，ACT 文など，モジュール間の非同期な相互作用を記述するための文法を含む．モジュールの中のレベルでの並列動作も記述可能で，同一エレメント（記憶回路の記述部分）内の複数の ASSIGN 文（記憶素子への代入）や，同一パス（組合せ回路の記述部分）内の EQUAL 文（端子信号作成）は同時に動作する．

（9） 論理設計者にとっては，ハードウェア設計のための仕様書として利用可能

（10） マイクロプログラマにとっては，マイクロプログラム作成のためのマニュアルとして利用可能

（9）と（10）は，マイクロプログラムと論理設計者との間のインタフェースとして，あいまいさのないマイクロプログラムの仕様書を作成できることを目標としている．

（11） 機能シミュレータのシステム記述入力言語としても利用可能

LFM 言語による記述は，シミュレータプログラムに自動的に変換できるので，汎用の機能シミュレータとして利用可能である．汎用の機能シミュレータとしては，CAS シミュレータ[8]，計算機の方式設計を目的とした GPMS[9] や，最近では MPG[10] などがある．

2.2 LFM 言語の開発目的

新たに LFM 言語を提案した理由としては，以下のような従来のハードウェア記述言語の問題点の解決を必要としたことがあげられる．

（1） LSI 化，マイクロプログラム化の傾向に対応して，従来のゲートレベルで設計することに重点を置いた言語（LDS[1]）から，機能記述に重点を置いた言語が必要になったこと．

（2） システム各部の並列処理をより自由に記述するために，モジュール相互間の信号交換を自由に記述できること．

（3） 機能の意味をわかりやすく記述可能なこと．

（4） 時間関係の自由な記述ができること．LFM では，IBM の CAS システムの timing ring のように一括してタイミングを記述する方式は採用しなかった．

（5） モジュールの階層構成を除去して，言語処理の簡単化を図り，モジュール間通信の自由度を増したいこと．

以上のような点を設計目標として，LFM システムは開発された．

2.3 LFM 言語の文法仕様

この節では，LFM 言語の文法[7] について説明し，論理装置や計算機システムが LFM によってどのように記述されるかについて述べる．

LFM のプログラムの構成要素は，**図4** のようになる．一つの論理装置ないし計算機システムは，LFM 言語では，ハードウェアの名前やサイズを決めるリソース宣言部と，それらの論理機能を記述する複数のモジュールで表現される．各モジュールは，更に組合せ回路の記述をするパスと，状態を変えるような順序回路の記述をするエレメントとに分かれる．パスでは，端子信号を指定された期間だけ出力することを記述する．エレメントは，遅延を伴って信号がセットされるような，メモリ，レジスタなどへの代入を記述し，信号の観測時刻と動作（代入）時刻を指定する．さらにエレメントは，各モジュールの状態や時計を制御する文を含み，モジュール間の相互作用を記述できる．以下に，LFM プログラムの各構成要素について説明する．

2.3.1 リソース宣言

レジスタ，パスなど計算機のハードウェア（言語の変数）は個々のモジュールとは独立に，計算機全体に対して宣言される．**図7** のように，宣言部は記述の対象となるシステム全体に対して一つあり，モジュール

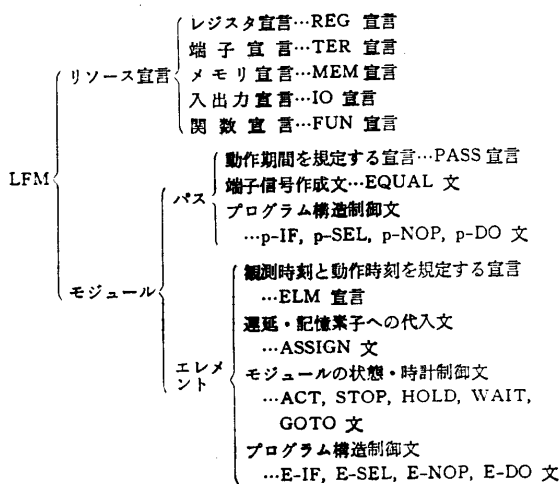


図 4 LFM プログラムの構成要素
Fig. 4 Components of LFM program.

は，この宣言部で宣言された変数について記述される．

宣言の対象となるリソースの種類としては，レジスタ，端子，メモリ，入出力，関数の5種類がある．REG，TER，MEM の3つの宣言文は，レジスタ，端子，メモリの名前とビット幅を（メモリの場合は，語数も）宣言する．レジスタ，メモリは，後述する ASSIGN 文で内容が置換されるまで変わらない．これらの出力は，いつでも式の右辺で参照できる．メモリとレジスタとの相違点は，メモリが2次元配列であること，語のアドレスを端子名やレジスタ名を使って指定できることである．端子は，バスなどの記述に使われ，それ自身記憶能力はないから，後述の PASS 文と EQUAL 文で指定されない限り，値は0である．

IO 宣言は，メモリと同様にレジスタの集合であるが，異なる点は，アドレスカウンタを内臓しており，sequential に入力あるいは出力されることである．シミュレーションコマンドによって初期設定され，プログラムから参照されるごとに，アドレスが一つずつ進む．すなわち，sequential な読み出ししかできない．

関数宣言は，最大3入力で1出力のブラックボックスであり，入出力の信号幅（ビット数と，固定長 [F] か可変長 [V] かの区別）を宣言する．関数の内容は別に PL/I のサブルーチンとして記述する．したがって，複雑な機能やマクロな機能を関数で置き換えられるので融通性がある．

2.3.2 モジュール構成要素

リソース宣言の後に，複数のモジュールが書かれる．各モジュールはそれぞれ自分の時計を持ち，通常 active になるとその時計が 0 から動き始める．各モジュール内の論理操作は，この時計のタイミングに合わせて記述され，それに合わせて動作する．一般には，指定された時間が経過すると，モジュールは自ら dead 状態に戻るように記述する．モジュールは，モジュール宣言の後に続く任意個のエレメントと任意個とパスの集合で構成される．

2.3.3 パス

パスは，パス宣言 PASS (t_1-t_2); で始まる組合せ回路の記述文（P文と呼ぶ）の集合である．パスを含むモジュールの時計が，パス宣言の2つの引数の示す期間内をさすとき，そのパス内のP文の集合は評価される．すなわち，パス宣言は組合せ回路の動作期間を指定する．P文の種類には，以下のものがある．

a．EQUAL 文

組合せ回路の入力と出力の関係を示し，この組合せ回路がパス宣言で指定された期間中 active であることを示す．したがって，その間に入力が変化すると出力も変化する．EQUAL 文の左辺は端子であり，active でないとき（モジュール自身が dead 状態か，または指定期間外のとき）は，その値は0である．組合せ回路の遅延は考えないので，

A=B; B=C; C=D;

のような場合，A, B, C はすべて同じ値となる．

b．プログラム構造制御文（p-IF, p-SEL, p-DO, p-NOP 文）

p-IF 文は，IF ⟨predicate⟩ THEN ⟨p-statement⟩ [ELSE ⟨e-statement⟩]; で一般の IF 文と同じである（⟨predicate⟩ は条件式である.）．p-IF 文が2分岐なのに対し，レジスタや端子のビットフィールドによる多分岐機能として，p-SEL 文がある．前述の図3は p-SEL 文の例であるが，これは MIR というレジスタの第1,2番目の2ビットによる分岐であることを示し，これが 00 なら，AとBの論理積を，01 であれば論理和を，10 であれば和を（ADD は関数），11 であればAの否定を，端子Cの出力信号にすることを意味する．そのほかに，動作のないことを示す p-NOP 文や，DO; と END; の対によってブロックを構成する p-DO 文がある．

2.3.4 エレメント

エレメントは，エレメント宣言 ELM (t_1-t_2); の後に順序回路の動作に関する文（E文と呼ぶ）を並べたものである．モジュールが active になってから，モジュールの時計が t_1 時刻のところで入力値を観測し，その動作は t_2 時刻に行われる．E文の種類には，以下のものがある．

a．ASSIGN 文

ASSIGN 文は，レジスタやメモリへの代入文であって，

AREG:=LBUS & RBUS;

のような形をしており，観測時刻 $(t=t_1)$ のときのバス LBUS と RBUS の値の論理積を取り（つまり右辺が評価され），その結果が動作時刻 $(t=t_2)$ に左辺のレジスタ AREG へセットされる．したがって，t_1 から t_2 の間は，AREG というレジスタ変数には，前の値が保持されていることに注意すべきである．

b．モジュールの状態・時計制御用E文

ACT 文は，ACT ⟨module name⟩ [⟨time⟩]; の形であり，dead 状態にある指定されたモジュールを起動するのに用いる．時刻を指定すると，起動されたモジュールの時計はその時刻にセットされ，省略したときは0にセットされる．

```
MODULE MAIN;
    ⋮
  ELM (t1-t2); HOLD;
    ACT TRAP 0;
  ELM (t2-t3);
    ⋮
MODULE TRAP;
    ⋮
  ELM (0-t4);
    ACT MAIN;
    STOP;
```

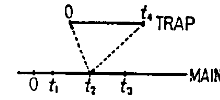


図5 HOLD 文の例
Fig. 5 The example of HOLD statement.

STOP 文は，STOP; だけであり，モジュールが自分自身を dead 状態にするために用いる．モジュールの時計は0にセットされる．

HOLD 文も，HOLD; だけであり，モジュールが自分自身を dead 状態にするために用いるが，モジュールの時計を HOLD 文の動作時刻に止めたままでリセットしない点が，STOP 文と異なる．これは割込み処理などの記述に便利である．（図5参照）

WAIT 文は，WAIT [:⟨interval⟩:]⟨predicate⟩; の形であり，⟨predicate⟩ が満たされるまで，そのモジュールの時計の進行を止める．⟨interval⟩ は ⟨predicate⟩ を調べる間隔であり，省略値は1である．非同期的な入力や，モジュール間の同期をとるのに使われる．

GOTO 文は，GOTO ⟨time⟩; の形で，この文が実行されると，モジュールの時計は ⟨time⟩ で指定された値にセットされる．ループ機能を与えるために用いられる．

これらの命令文の使用例を図6に示す．

```
(例1) サイクリックなモジュール
      MODULE A;
          ELM (0-t1); OP1;
          ELM (t1-t2); OP2;
            ⋮
          ELM (tn-1-tn); GOTO 0;
(例2) 一定期間のみ active なモジュール
      MODULE B;
          ELM (0-t1);
            ⋮
          ELM (t2-t3); ACT A;
              STOP; (または HOLD;)
(例3) 他のモジュールまたはある事象により起動されるモジュール
      MODULE C;
          ELM (0-t1);
            ⋮
          ELM (t3-t4);
              WAIT: 1: FLAG=1B;
```

図6 モジュール状態制御命令の例
Fig. 6 Some examples of module status control instruction.

c．プログラム構造制御文

これには，E-IF，E-SEL，E-DO，E-NOP 文があり，パスの場合と機能や使用法は同じである．

2.3.5 LFM による記述例

図7に示す例は，モトローラ社の8ビットマイクロプロセッサ M6800[2] を記述したものの一部である．モジュール EXEC は，メインメモリ MM から，命令レジスタ IR への命令フェッチと，IR の上位4ビットのデコードを行う．このデコードによって，M0 から MF までの 16 種のモジュールに制御が移り，命令のデコードと実行が行われる．各モジュールの処理が終わると，モジュール EXEC へ制御が戻る．この記述例では，各命令の実行に必要なサイクル数にあわせて，M0 から MF のモジュールを記述してある．マイクロプロセッサの詳細な内部仕様書さえあれば，更にオペランドバイトのフェッチが何サイクル目で行われるかなど，時間関係を必要な精度で記述できる．

```
LFM;
   REG  A(8),B(8),IX(16),PC(16),SP(16),
        IR(8),GOMI(8),MAR(16),DATA(8),STEP(16),
        H(1),I(1),N(1),Z(1),V(1),C(1),
        EMOD(1),IMOD(1);
   TER  AB(16),DB(8),RST(1),NMI(1),HALT(1),IRQ(1),
        TSC(1),DBE(1),BA(1),VMA(1),RW(1);
   FUN  INC(V16,V16),DEC(V16,V16),SBC(F8 F8 F1,F8),
        ADC(F8 F8 F1,F8),DAA(F8,F8),REL(F16 F8,F16),
        SUB(F8 F8,F8),ADD(F8 F8,F8),NEG(F8,F8),COM(F8,F8),
        ADDX(F16 F8,F16),
        AND(F8 F8,F8),OR(F8 F8,F8),CPX(F16,F16),
        STK(F16,F16),EOR(F8 F8,F8),LSR(F8,F8),
        ROR(F8,F8),ASR(F8,F8),ASL(F8,F8),ROL(F8,F8),
        DECR(F8,F8),INCR(F8,F8),TST(F8,F8),
        CLR(F8,F8),MOVE(F8,F8);
   MEM  MM(2048,8);
   MODULE  EXEC;
        ELM(0-7);     IR:=MM(PC);  PC:=INC(PC);
                      STEP:=INC(STEP);
        ELM(7-8);     IF EMOD=1B THEN STOP;
                      SEL(4) IR(1-4);
            ACT M0; / ACT M1;
          / ACT M2; / ACT M3;
          / ACT M4; / ACT M5;
          / ACT M6; / ACT M7;
          / ACT M8; / ACT M9;
          / ACT MA; / ACT MB;
          / ACT MC; / ACT MD;
          / ACT ME; / ACT MF;
                      SEND; STOP;
   MODULE  M0;
        ELM(0-12);  SEL(4) IR(5-4);
            ACT ERROR;  / ACT ERROR;
          / NOP; * NOP *
          / ACT ERROR;
          / ACT ERROR;  / ACT ERROR;
          / DO; H:=A(3); I:=A(4); N:=A(5);    * TAP *
                Z:=A(6); V:=A(7); C:=A(8); END;
          / DO; A(1-2):=11B;      * TPA *
                A(3-6):=H||I||N||Z||V||C; END;
          / DO; IX:=INC(IX); GOTO 20; END; * INX *
          / DO; IX:=DEC(IX); GOTO 20; END; * DEX *
          / V:=0B; * CLV *
          / V:=1B; * SEV *
          / C:=0B; * CLC *
          / C:=1B; * SEC *
          / I:=0B; * CLI *
          / I:=1B; * SEI *
                SEND;
            IF IR(5-3)¬=100B THEN    * EXCEPT INX,DEX *
               DO; ACT EXEC; STOP; END;
        ELM(20-40); ACT EXEC; STOP;
   MODULE M1;
        ELM(0-12);  SEL(4) IR(5-4);
            A:=SUB(A,B); * SBA *
          / GOMI:=SUB(A,B); * CBA *
          / ACT ERROR;
```

図 7　LFM プログラムの例
Fig. 7　Example of LFM program.

図7の例では，1サイクルを 10 単位にとってあり，M0 の中で INX, DEX 命令が4サイクルの他は，すべて2サイクルの命令である．

3. LFM システムの構成

第2章で述べた言語を用いて記述された論理装置をシミュレートするためのシステム構成は図8のようになる．シミュレーション方式はコンパイラ方式であり，次の3つのセクションからなる．

（1） トランスレーションセクション
（2） コンパイルセクション
（3） シミュレーションセクション

各セクションで使用するプログラム名とその大きさは**表1**の通りである．入力となるマシン記述は LFM 言語によって記述され，変換プログラム（トランスレータ）によって PL/I で記述されるシミュレータプログラムに変換される．シミュレータの大部分は，シミュレータベースと呼ばれる固有部分であり，変換プログラムが生成するプログラムは，論理装置によって変わる部分だけである．この両者と関数部分とを組合せ，PL/I コンパイラを通すことにより，実行用プログラムとなる．マイクロプログラム，入力データ，シミュ

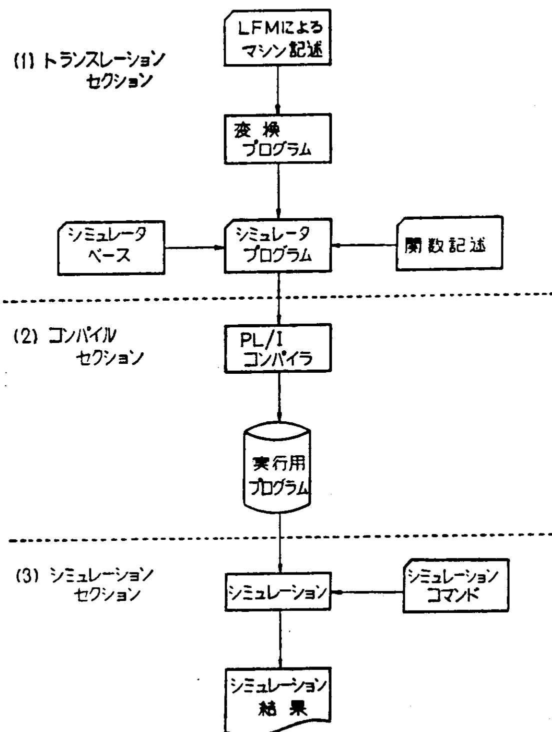


図 8　シミュレーションの流れ
Fig. 8　Simulation Diagram.

表1 シミュレーションプログラムの名前と大きさ
Table 1 The name and size of the simulation programs.

a) 各セクションで使用するプログラム

セクション名	プログラム名
トランスレーション	トランスレータ (PL/I で記述)
コンパイル	HITAC 8800/8700 OS-7 PL/I コンパイラ
シミュレーション	シミュレータベース (PL/I で記述) と変換された PL/I プログラムとの実行用オブジェクト＋関数サブルーチンの実行用オブジェクト

b) 各プログラムの大きさ

プログラム名	ソースカード枚数	外部手続きの数	PL/I ソースステートメント数
トランスレータ	3,403	41	3,399
シミュレータベース	878	1	902

レーション出力の指定，シミュレーション時間などは，シミュレーション実行時にシミュレーションコマンドの形で与えられる．

シミュレーション開始時には，すべてのモジュールは dead 状態であり，シミュレーションコマンドによって，必要なモジュールの起動と，レジスタ・メモリへの初期値設定やプログラムロードが行われる．また，シミュレーションを終わるのための条件は，#EOS という外部手続きで記述する．

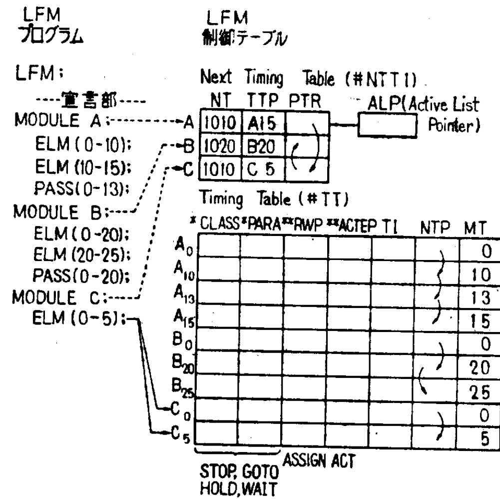
シミュレーションの出力コマンドには，値が変化したときのみ出力するモード(チェック指定の出力)と，一定の間隔ごとに値を出力するモード（インターバル指定の出力）との2つのモードがある．出力はすべて，ビットパターンで出る．出力の書式や条件を変えたいときは，#EOS ルーチンの中で変えることができる．

4. シミュレータの構造

LFM 言語で記述された計算機のシミュレーションを行う上で，非常に重要な役割を果たすデータ構造について述べ，次に，それがどのように操作されるかについて述べよう．

4.1 シミュレーションに必要な各種のテーブル

シミュレーションは，active なモジュールのみを各時点で処理の対象としていけばよい．また，各モジュールの機能は，モジュールの時刻で規定され，モジュール内で宣言された時刻に達しないと処理を要求しない．通常は，アクティブなモジュールの時計は実時間と共に進むので，現在のモジュール時刻から次に処理を要求する実時刻を決定できる．他方，WAIT 命令によって WAIT のかけられているモジュールは，計算機の状態（レジスタ，メモリ，端子の値）が変わった時点で WAIT 条件のチェックを行う必要がある．このような active なモジュールの要求する処理や WAIT 条件のチェックを調べやすくするため，次のようなデータ構造を採用する．



* このエレメント内に HOLD 文と STOP 文があるときは，1，GOTO 文，WAIT 文があるときは，それぞれ 2,3 が，その他(パスを含む)の場合は 0 が CLASS 欄に入る．PARA 欄には，(STOP 文や HOLD 文によって) 停止後モジュールが保持すべき #TT エントリ番号や WAIT 条件が入る．

** RWP 欄には，この #TT エントリの時刻に更新されるレジスタ名とその更新値を持つテーブル (#RWET) へのポインタが，ACTEP 欄には，この時刻に起動されるモジュールの情報 (#ACTET) へのポインタが入る．

図9 シミュレータの制御表
Fig. 9 Control tables of simulator.

まず，各モジュールに一つずつ，Next Tming Table (#NTT1) のエントリを割り当てる．#NTT1 エントリは，NT (Next Time)，TTP (Timing Table Pointer)，PTR (Pointer) の3つの欄からなる（図9）．次に，すべてのモジュールについて，各モジュールで宣言されている時刻に対し，一つずつ，Timing Table (#TT) のエントリを割り当てる．#TT は，各時刻で要求される処理をテーブルの形にしたもので，各エントリは，CLASS, PARA, …, MT の各欄からなる．

さて，active な状態にあって wait状態にはないモジュールに対応する #NTT1 エントリを PTR 欄を介して接続したリストを考え，これをアクティブリストと名付ける．active リストに属する #NTT1 エントリの NT 欄には，そのモジュールが次に処理を要求する実時刻が書き込まれてあり，active リストに属するエントリは，この値の順につながれている．また，active リストに属する #NTT1 エントリの TTP

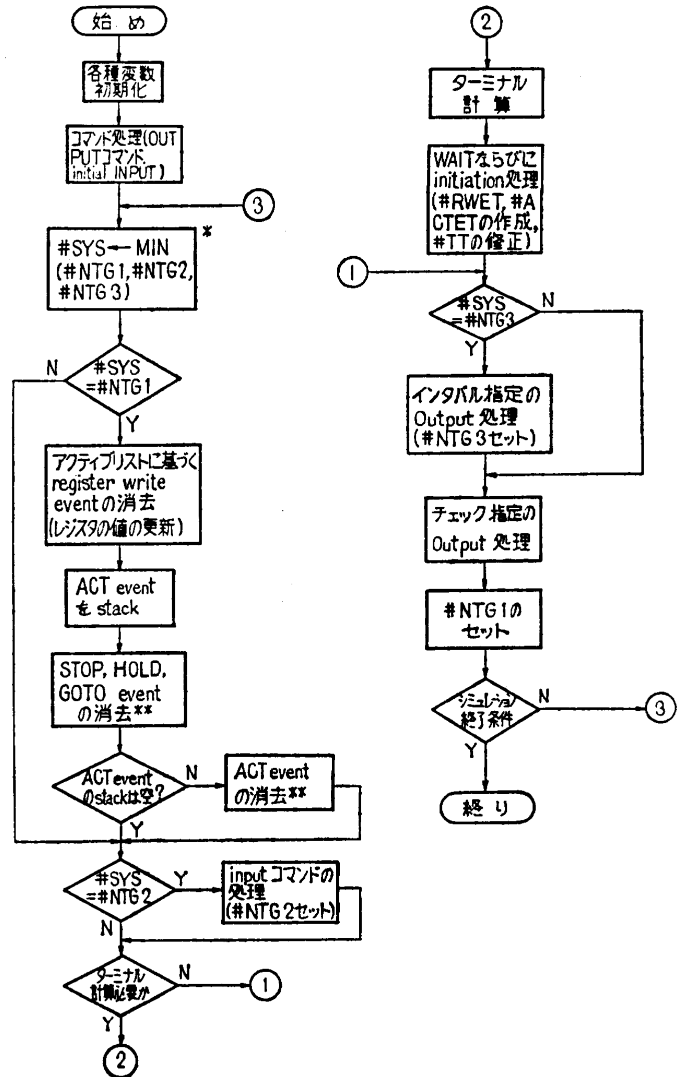
欄には，実時刻がその #NTT 1 エントリの NT 欄と一致したときに，モジュールが要求する処理を示す #TT エントリへのポインタが書き込まれている．他方，wait 状態にあるモジュールに対応する #NTT 1 エントリを PTR 欄でつないだリストを考え，wait リストと名付ける．この場合 TTP 欄には wait 状態のモジュールのモジュールタイムに対応する #TT エントリへのポインタが入っている．図 9 に簡単な例を示す．

次に，モジュールが要求する処理について考えよう．モジュールは，あるモジュールタイムに到達すると，現在の計算機の状態を判定し，種々の命令を出す．このとき，各命令による状態変化が起きるのは，モジュールの時計がエレメントの第 2 引数で示される時刻になったときである．また，エレメントの第 1 引数の時刻では，あらかじめ命令による状態変化を予測しておかねばならない．すなわち，命令を出すことは，命令の完了時刻に対応する #TT エントリに，命令の完了時刻で起こるべき状態変化を書き込むことであり，命令が終了することは，命令を出したモジュールの時計が命令の終了時刻に一致したとき，この時刻に相当する #TT エントリから既に用意されていた情報を取り出し，実際に状態変化を起こすことに対応する．命令をどのように出すかは，#TT の TI (Time Initiation) 欄からわかり，またその #TT エントリの MT (Module Time) 欄の時刻で終了する命令の情報は，以下のような #TT の各欄に貯えられているので，これからわかる．

STOP, GOTO, HOLD, WAIT 命令
　　　　　　: CLASS 欄と PARA 欄
ASSIGN 命令 : RWP 欄
ACT 命令　 : ACTEP 欄

なお，#TT エントリの MT 欄には，そのエントリに対応するモジュールの時刻が，NTP 欄には，そのモジュールで次に宣言されている時刻に対応する #TT エントリへのポインタが格納されている．

以上に述べたテーブルを用いて，シミュレーションは図 10 のフローチャートのように行われる．



* #NTG 1 はアクティブリストの最初のエントリの NT 欄の値であり，#NTG 2 (#NTG 3) は次に入力 (出力) 要求の起きる時刻である．また，#SYS はシミュレーションのタイマである．
** 「event の消去」とは，各命令による状態変化を実際に起こすことを意味する．

図 10 LFM シミュレータの流れ図
Fig. 10 The flowchart of LFM simulator.

5. LFM 言語による記述とシミュレーション結果

LFM 言語の記述能力およびシミュレーション速度についての評価を得るために，例として，電子技術総合研究所のパターン情報処理システム (PIPS) プロジェクトで開発されたマイクロプロセッサ PULCE[3] とモトローラ社の 8 ビットマイクロプロセッサ M 6800[2] を採用した．各マイクロプロセッサの仕様，LFM による記述とシミュレーションの結果を表 2 に示す．これらの処理結果を以下に要約する．

(i) PULCE や M 6800 のようなゲート数の多い LSI の論理機能を，関数も含めて約 800 行程度で記述することができた．マシンに固有の部分の記述だけでよく，マシンごとに専用のシミュレータを作成するよ

表 2 2種のマイクロプロセッサのシミュレーション結果

Table 2 The simulation result of two microprocessors.

i) LFM による記述結果

評価項目	M 6800	PULCE
トランスレーション時間	32.712 秒	30.900 秒
ベースを含むコンパイル時間	136.715 秒	100.177 秒
関数のコンパイル時間	16.676 秒	15.722 秒
LFM ソース行数	435 行	467 行
関数の行数	352 行	331 行
生成された PL/I ソース行数 (ソースステートメント数)	1,321 行 (2,747)	1,376 行 (1,745)
関数の数	26 個	27 個

時間はすべて CPU 時間

ii) シミュレーション速度

{PULCE… 1 マイクロ命令当り平均 216.5 ms
{M 6800 … 1 サイクル当り平均 26.7 ms

M 6800 の場合の実行結果

テストプログラム	実行命令数	実行サイクル数	CPU 時間 [秒]	サイクル当りの CPU 時間 [ms]
モニタ (MINIBUG)	252	1,000	21.366	21.4
8ビット2進乗算	112	247	7.156	28.6
2進-10進変換	145	646	19.325	29.9
16ビット2進乗算	136	699	18.394	26.3
16ビット2進除算	197	1,007	27.331	27.1

りは容易である.

(ii) PULCE はマイクロ命令, M 6800 は機械語命令のレベルでのシミュレーションを目標としており, 目的に応じて記述レベルを変えられる.

(iii) PULCE の記述では, 1サイクル当りのエレメントの数が多く, かつ, すべてのエレメントの動作期間中でパスの評価が行われるようにしたため, シミュレーション速度が遅い. M 6800 の記述では, このためパス文の使用をやめ, 命令のタイプ別にモジュールを割当て, 余分なパスやエレメントの評価が行われないようにしたので, 1サイクル当りの時間が約8分の1に減っている. したがって, PULCE の場合, 1マイクロサイクル内の各フェーズをモジュールとして記述すべきであり, LFM プログラムは, 効率の点で, マシン機能のモジュールへの分割の方法に注意しなければならない.

(iv) 1サイクル当りのシミュレーション時間は, HITAC 8800 を用いた場合, PULCE が約 220 ms, M 6800 が約 27 ms であった. 速度の改善法としては, 上述の点の他に, シミュレータの記述言語を PL/I から FOFTRAN やアセンブラに変えることがあげられる.

6. おわりに

論理装置用の汎用機能シミュレーションシステムとして, 新たにハードウェア記述言語の仕様を決め, この言語を入力とするシミュレータを作成した. 2種類のマイクロプロセッサについて, 記述した結果, マイクロ命令レベル, 機械語命令レベルのいずれも少ない行数で記述できた. 実際に, シミュレーションを実行させた結果, 処理速度は記述方法に依存することがわかった. また, 細かいタイミングの記述が容易であり, 種々のレベルでシミュレーションが可能なため, 本システムは汎用機能シミュレータとして有用であると考える.

参考文献

1) 岡田, 元岡: 論理設計言語, 信学誌, 50, 12, p. 2353 (1967).
2) Motorola: M 6800 Microprocessor Programming Manual.
3) 電子技術総合研究所: パターン情報処理システムプロジェクトマイクロプロセッサ PULCE 解説書.
4) 元岡, 新開: マイクロ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資, 74-19, (1974).
5) 新開, 元岡: 汎用マイクロプログラムシミュレータ, 情報処理学会全国大会 (1973).
6) 上森, 元岡: 論理装置の汎用機能シミュレータの性能評価, 通信学会情報部門全国大会(1977).
7) 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).
8) Buckingham, B. R. S., et al.: The Controll Automation System, 6th Annual Symposium on Switching Circuit Theory and Logical Design (1965).
9) Yamamoto, M., et al.: A Microprogrammed Computer Design and Evaluation System, First USA-JAPAN Computer Conference, pp. 139-144 (1972).
10) 馬場, 萩原, 藤本, 高橋, 碇谷: 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).

付 録 1 LFM の言語仕様

SYNTAX OF LFM (Language For Microprogram)

1. ⟨lfm⟩=LFM; ⟨declaration⟩ $\int_1^\infty$ [⟨module⟩]END_LFM;
2. ⟨declaration⟩= $\int_1^\infty$ [⟨register declaration⟩
   |⟨terminal declaration⟩
   |⟨memory declaration⟩
   |⟨function declaration⟩
   |⟨io declaration⟩
3. ⟨register declaration⟩=REG ⟨register name⟩ (⟨decimal constant⟩)
   $\int_0^\infty$ [, ⟨register name⟩(⟨decimal constant⟩)];
4. ⟨terminal declaration⟩=TER ⟨terminal name⟩ (⟨decimal constant⟩)
   $\int_0^\infty$ [, ⟨terminal name⟩(⟨decimal constant⟩)];
5. ⟨memory declaration⟩=MEM ⟨memory name⟩ (⟨decimal constant⟩, ⟨decimal constant⟩)
   $\int_0^\infty$ [, ⟨memory name⟩(⟨decimal constant⟩, ⟨decimal constant⟩)];
6. ⟨io declaration⟩=IO⟨io name⟩ (⟨decimal constant⟩, ⟨decimal constant⟩)
   $\int_0^\infty$ [, ⟨io name⟩(decimal constant⟩, ⟨decimal constant⟩)];
7. ⟨function declaration⟩=FUN ⟨function name⟩
   $\left(\int_0^3\right.$[⟨parameter⟩], ⟨parameter⟩$\left.\right)$;
8. ⟨module⟩=⟨module declaration⟩ $\int_1^\infty$ [⟨pass⟩
   |⟨element⟩]
9. ⟨module declaration⟩=MODULE ⟨module name⟩;
10. ⟨pass⟩=⟨pass declaration⟩ $\int_1^\infty$ [⟨p-statement⟩]
11. ⟨element⟩=⟨element declaration⟩
    $\int_1^\infty$ [⟨e-statement⟩]
12. ⟨pass declaration⟩=PASS ⟨timing part⟩;
13. ⟨element declaration⟩=ELM ⟨timing part⟩;
14. ⟨timing part⟩=(⟨decimal constant⟩-⟨decimal constant⟩)
15. ⟨p-statement⟩=⟨equal statement⟩
    |⟨p-if statement⟩|
    |⟨p-sel statement⟩|⟨p-do statement⟩
    |⟨p-nop statement⟩
16. ⟨e-statement⟩=⟨assign statement⟩
    |⟨e-if statement⟩
    |⟨act statement⟩|⟨wait statement⟩
    |⟨e-sel statement⟩
    |⟨stop statement⟩|⟨hold statement⟩
    |⟨e-do statement⟩
    |⟨e-nop statement⟩|⟨go to statement⟩
17. ⟨assign statement⟩=[⟨register⟩
    |⟨memory⟩]: ⟨expression⟩;
18. ⟨equal statement⟩=⟨terminal⟩=⟨expression⟩;
19. ⟨e-if statement⟩=IF ⟨predicate⟩ THEN ⟨e-statement⟩;
    $\int_0^1$[ELSE ⟨e-statement⟩;]
20. ⟨p-if statement⟩=IF ⟨predicate⟩ THEN ⟨p-statement⟩;
    $\int_0^1$[ELSE ⟨p-statement⟩;]
21. ⟨act statement⟩=ACT ⟨module name⟩ $\int_0^1$[⟨time⟩];
22. ⟨wait statement⟩=WAIT $\int_0^1$[:[interval⟩:]
    ⟨predicate⟩;
23. ⟨stop statement⟩=STOP;
24. ⟨go to statement⟩=GOTO ⟨time⟩;
25. ⟨e-nop statement⟩=NOP;
26. ⟨p-nop statement⟩=NOP;
27. ⟨e-do statement⟩=DO; $\int_1^\infty$ [⟨e-statement⟩]END;
28. ⟨p-do statement⟩=DO; $\int_1^\infty$ [⟨p-statement⟩]END;
29. ⟨e-sel statement⟩=SEL(⟨decimal constant⟩) ⟨expression⟩;
    $\int_1^\infty$ [/|⟨e-statement⟩]SEND;
30. ⟨p-sel statement⟩=SEL (⟨decimal constant⟩) ⟨expression⟩;
    $\int_1^\infty$ [/|⟨p-statement⟩]SEND;
31. ⟨register⟩=⟨register name⟩ $\int_0^1$[(⟨subpart⟩)];
32. ⟨terminal⟩=⟨terminal name⟩ $\int_0^1$[(⟨subpart⟩)];
33. ⟨memory⟩=⟨memory name⟩([⟨expression⟩
    |⟨decimal constant⟩]
    $\int_0^1$[, ⟨subpart⟩])
34. ⟨io⟩=⟨io name⟩ $\int_0^1$[(⟨subpart⟩)]
35. ⟨fun⟩=⟨function name⟩(⟨expression⟩)
    $\int_0^2$[, ⟨expression⟩])
36. ⟨subpart⟩=⟨decimal constant⟩
    $\int_0^1$[−⟨decimal constant⟩]
37. ⟨expression⟩=⟨operand⟩ $\int_0^\infty$ [⟨operator⟩⟨operand⟩]
38. ⟨operand⟩=¬⟨operand⟩|(⟨expression⟩)

|⟨binary constant⟩
|⟨fun⟩|⟨register⟩|⟨memory⟩
|⟨terminal⟩|⟨io⟩

39. ⟨operator⟩::=& | | | ‖
40. ⟨predicate⟩::=⟨predicate atom⟩|(⟨predicate atom⟩) $\int_0^\infty$ [⟨concatenator⟩(⟨predicate atom⟩)]
|¬(⟨predicate atom⟩)
41. ⟨concatenator⟩::=& | |
42. ⟨predicate atom⟩::=⟨expression⟩⟨comparator⟩⟨expression⟩
43. ⟨comparator⟩::= = | > | < | > = | < = | ¬ = | ¬ > | ¬ <
44. ⟨binary constant⟩::= $\int_1^\infty$ [0|1]B
45. ⟨time⟩::=⟨decimal constant⟩
46. ⟨interval⟩::=⟨decimal constant⟩
47. ⟨parameter⟩::=V ⟨decimal constant⟩
|F⟨decimal constant⟩
48. ⟨decimal constant⟩::= $\int_1^5$ [⟨integer⟩]
49. ⟨register name⟩::=⟨name⟩
50. ⟨terminal name⟩::=⟨name⟩
51. ⟨memory name⟩::=⟨name⟩
52. ⟨io name⟩::=⟨name⟩
53. ⟨function name⟩::=⟨name⟩
54. ⟨module name⟩::=⟨name⟩
55. ⟨name⟩::=⟨alphabet⟩ $\int_0^5$ [⟨alphabet⟩|⟨integer⟩]
56. ⟨integer⟩::=0|1|2|3|4|5|6|7|8|9
57. ⟨alphabet⟩::=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P
|Q|R|S|T|U|V|W|X|Y|Z
58. ⟨hold statement⟩::=HOLD;

(注) $\int_{n_1}^{n_2}$[ ] は [ ] の中が n_1 個から n_2 個の範囲でくり返されてよいことを示す.

付　録 2 シミュレーションコマンドの書き方

HOW TO WRITE INPUT COMMAND OF LFM SIMULATOR

1. ⟨simulator command⟩::=⟨output command⟩
INPUT;
⟨input command⟩ EOC;
2. ⟨output command⟩::=⟨timing part⟩⟨object⟩
3. ⟨timing part⟩::=/⟨D.C⟩-⟨D.C⟩/
4. ⟨object⟩::=⟨object atom⟩ $\int_0^\infty$ [, ⟨object atom⟩]
5. ⟨object atom⟩::=⟨module name⟩⟨register name⟩
⟨memory name⟩ (⟨D.C⟩ $\int_0^1$ [−⟨D.C⟩])
6. ⟨input command⟩::=/⟨D.C⟩/ $\int_1^\infty$ [input statement⟩]
7. ⟨input statement⟩::=ACT ⟨module name⟩(⟨D.C⟩);
⟨register name⟩ $\int_0^1$ [(⟨D.C⟩ $\int_0^1$ [−⟨D.C⟩)]
=⟨H.C⟩;
⟨memory name⟩ (⟨D.C⟩ $\int_0^1$ [−⟨D.C⟩])
=⟨H.C⟩ $\int_0^\infty$ [, ⟨H.C⟩];

COMMENT:

3. 1st ⟨D.C⟩ of ⟨timing part⟩=first time of output request
2nd ⟨D.C⟩ of ⟨timing part⟩=time interval of output request
(except for 0. If this value is zero, output is requested whenever the contents of register, memory are changed.)
5. 1st parameter of ⟨memory name⟩=first address of memory to be output
2nd parameter of ⟨memory name⟩=last address of memory to be output
6. ⟨D.C⟩ of ⟨input command⟩=input time
7. ⟨D.C⟩ of ⟨module name⟩=activation time of module
1st parameter of ⟨register name⟩=first bit # of register to be output
2nd parameter of ⟨register name⟩=bit count of register to be output

⟨D.C⟩=decimal constant
⟨H.C⟩=hexadecimal constant
(The value of ⟨H.C⟩ is assigned into register or memory after it is right-justified.)

(昭和 53 年 8 月 14 日受付)
(昭和 54 年 11 月 16 日採録)