講　座

コンピュータの設計自動化（2）*

倉　地　正**

4. ファームウェア設計

ファームウェア設計が電子計算機の開発の中に占める割合は年々増加の傾向にあるが，その原因として次の様なものがある．

（1） ファミリー化に対処するため，小型機でも複雑な命令が必要．

（2） 制御メモリの高速化，価格低減に伴う多重マイクロ制御ユニット採用の大型計算機の出現．

（3） OS のファームウェア化，高級言語用命令の採用．

（4） エミュレーション機構の採用．

（5） マイクロ診断の採用．

（6） 周辺機器制御装置のファームウェア化による機能向上．

（7） マイクロプロセッサ採用の増大．

これに伴いファームウェア設計人口が増大しその設計サポートシステムも高度なものが要求されるようになって来た．

代表的なファームウェア設計システムは**図-6** に示すような構成をしており，次の機能を有する．

- 記号言語，高級言語のビットパタンへの変換
- 番地割付
- マイクロプログラムシミュレーション
- 各種ドキュメントの作成
- ファームウェアロード媒体の作成

4.1 マイクロプログラム記述言語

マイクロプログラミングは通常のプログラミングと似ているが，高級言語の採用という点においては大分遅れている．その理由としてはマイクロプログラムの性能がそのまま計算機の性能を支配するためできるだけ効率の良いマイクロプログラムが求められること，

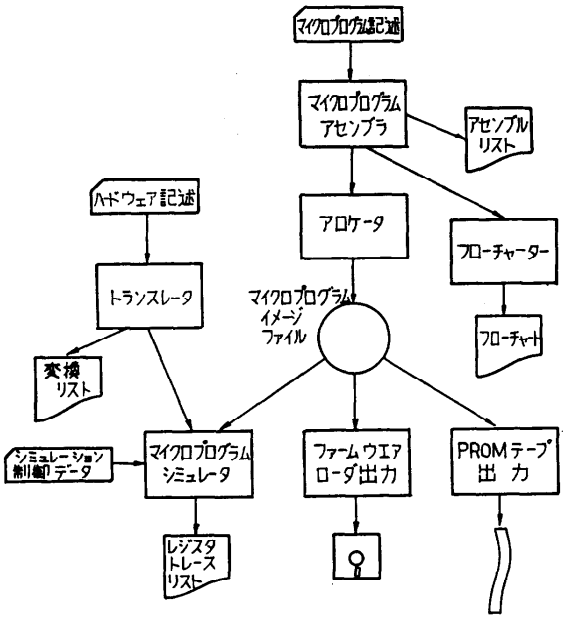


図-6 ファームウェア設計システム

一般のプログラムに比べ限定された設計者のみがマイクロプログラムを作ること，一つの計算機に必要なマイクロプログラムの量がそれ程多くないことなどがある．

従って計算機用の実用システムではプログラム語数の圧縮，ビット幅の圧縮等の最適化は人手で行い，マイクロオーダを記号言語で記述したものを単にビット変換するのみのものが多く使用されている[17),18)]．

マイクロプログラム用の記号言語は通常定義部とプログラム記述部とから成っている．定義部はメモリ容量，ビット長，マイクロ命令のフィールドの定義，パリティビットのつけ方等のパラメータ指定と各マイクロオーダごとの記号名及びそのビットパタンの対応を定義するものである．記述部は定義部で定義した記号名を用いてマイクロプログラムを記述するもので，並列型マイクロ命令の場合にはマイクロオーダを複数個

---

* Design Automation of Computers by Tadashi KURACHI (Computer Hardware Development Dept., Ome works, Tokyo Shibaura Electric Co., Ltd.)

** 東京芝浦電気(株)青梅工場ハードウェア開発部

```
1  |
2  |        This procedure will search stack from 0 to 100. If it finds a word
3  |        in the stack that matches 'key' it will set 'index' to the address
4  |        and set 'found' to true. Otherwise found will be false and 'index'
5  |        will equal 101. /
6  |proc     SEARCH
7  |        (POUND=GC1,INDEX=A1;LIST(0:100) = STACK(0:100),KEY = A2)
8  |        declare I = DUMMY;
9  |        mac 'NOT FOUND YET' =
10 |                 ((all(I) (INDEX>I>=0 , LIST(I)¬= KEY)) & 0<=INDEX)
11 |    cam;
12 |
13 |
14 |        assume TRUE;
15 |        conclude if FOUND
16 |             then  'NOT FOUND YET' & INDEX <= 100
17 |               & LIST(INDEX) = KEY
18 |             else 'NOT FOUND YET' & INDEX = 101 fi;
19 |
20 |
21 |    INDEX := 0, FOUND := FALSE;
22 |    assert FOUND = FALSE & 'NOT FOUND YET'
23 |        repeat
24 |        if KEY = LIST(INDEX)
25 |             then FOUND := TRUE, EXIT fi;
26 |        INDEX := INDEX + 1
27 |    until INDEX = 101
28 |    taeper
29 |
30 |    corp;
```

図-7 Strum による記述例[22]

並べて書くことが多い．記述を簡潔かつ見易くするためにマクロ機能を有するものも多い．またソースプログラムを管理するためシステムプログラムのエディタを利用するかまたは専用のエディタを開発した例もある．

一方マイクロプログラム設計効率の向上，ドキュメント性の向上等を目的とした高級マイクロプログラム言語 (HLMPL) の開発も大学，研究所等を中心に幅広く行われている．

HLMPL はハードウェア記述言語から派生したものと高級プログラム言語から派生したものとがある．

前者には APL[19]，CDL[20]，MPGS[21]，Strum[22]，LFM[23]，MIPS[24]等の言語があり，これらでは通常マイクロプログラムは対象となるハードウェア構成を完全に知っていてプログラムを書く必要がある．これらの言語はマイクロプログラムそのものの記述よりもむしろ後で述べるマイクロプログラムシミュレータにハードウェアの構造を入力するため使われることが多い．

一方後者はマイクロプログラムをもう少し抽象的なレベルで記述しようというもので PL/I や ALGOL 等の系統をひいている．この系統の言語はユーザマイクロプログラムを書く事を主目的としたものが多く，SIMPLE[25]，PUMPKIN[26] 等の言語がある．HLMPL の一例として Strum[22] を紹介する．

Strum は Burroughs 社のユーザマイクロプログラム可能な D-Machine のため California 大学で開発されている構造化マイクロプログラム開発システムで，ブロック構造，プロシージャ，マクロ，選択，繰り返し等の文を許す．また自動プログラム検証 (Verification) を許すため仕様を主張 (Assertion) の形で書くことが出来る．図-7 に Strum の記述例を示す．

4.2 マイクロプログラムの変換

記号言語に対する変換処理は割合簡単で定義部で宣言したマイクロオーダをテーブルに登録しておき，各マイクロステップごとに対応するマイクロオーダをテーブルから探し出し，該当するビットパタンを命令語中に埋め込んでゆけば良い[18]．この時パタンのオーバーラップ，未定義等のエラーを検出してゆく．また記号番地で書かれた分岐先等のアドレスをブロックごとに与えられる先頭番地を基に物理アドレスに変換する．変換されたオブジェクトコードはマイクロプログラムイメージファイルに記録されシミュレータ及びポストプロセス用の入力として保存される．

HLMPL に対する変換システムは一般の高級言語用コンパイラに似ているが，オブジェクトコードの性能に対する要求が厳しいため最適化処理は高度のものが要求される．図-8 (次頁参照) に文献 27) で提案されているマイクロプログラムコンパイラのフローチャートを示す．

最適化処理の一つは並列処理の認識の問題である．大型計算機用のマイクロプログラムは通常並列型であるため可能な限り多くのマイクロオーダを同一のマイクロ命令中で実行するのが性能向上に結びつく．更にマイクロ命令を格納する制御メモリは高価であるためその語数及びビット幅をできるだけ圧縮するのが望ましいこれらの問題は理論的に扱い易いため種々の研究[28,29]が行われており，最近の IEEE Trans. on Computers に Survey[30] 及び最近の状況が報告されている．

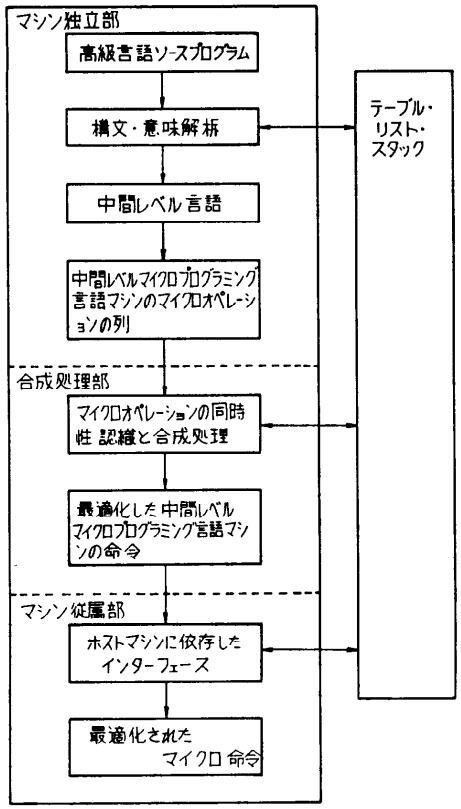


図-8 高級マイクロプログラミング言語用コンパイラ[27]

4.3 マイクロプログラムシミュレーション

一般のプログラムと同様マイクロプログラムも完全なものにするためにはデバッグが必要であるが，並列動作が多いこと，固定制御記憶に格納すること等の理由により実際のハードウェアを使ってデバッグを行うことは難しい．このためマイクロプログラムの開発にはシミュレータが重要な役割を果す．マイクロプログラム制御計算機の性能評価及び性能改善のためシミュレータが使われることもある．

マイクロプログラムシミュレータには専用方式と汎用方式がある．前者は特定のアーキテクチャをプログラムに組込んだものでありスピードは早いが，開発の途中でアーキテクチャの変更がある度にプログラマがシミュレータを修正しなければならずアーキテクチャが固定してからでないと使い易くない[18]．

汎用方式のものはハードウェア記述言語 (HDL) によりマイクロオーダに対する対象ハードウェアの動作を記述する形式のものが多い．HDL は設計者にも読み易くプログラマの世話にならずにアーキテクチャの修正が行える他，設計者間の情報交換用のドキュメントとしても有効である．

HDL はシミュレーションを実行するためにシミュレーション制御プログラムとリンクしなければならないが，その方式にコンパイル方式とインタプリティブ方式がある．

コンパイル方式の例としては古くは IBM 社の CAS[31] がある．これは IBM 360/40, 50, 65, 67 等の開発に使われたシステムで対象計算機の計述は Machine Description Compiler によりアセンブラ言語 FAP によるシミュレーションプログラムに変換される．これがシミュレーション制御プログラムとリンクされ別にアセンブルされたマイクロプログラムコードを解釈しながらシミュレーションを実行する．LFM[23], MIPS[24] 等は HDL を PL/I に変換して実行するコンパイル方式のシミュレータである．

インタプリティブ方式の例としては μ-SIM-II[18], FALOS[32] 等がある．μ-SIM-II は HDL を持たず代りにマイクロオーダ番号（ゲート番号）ごとにそれをシミュレートする機能サブルーチン番号と対象となるレジスタ，バス等のアドレスをテーブルの形で与えるシミュレータである．これは HDL コンパイラが不要なので，開発が容易であるがストラクチャの変更には熟練が必要である．

FALOS は汎用機能シミュレータでありマイクロプログラムのシミュレーションも可能である．FALOS コンパイラは CDL 言語[20]を修正した言語で記述したハードウェア構造を中間言語に変換する．中間言語は

{オペランドアドレス，ビット位置，ビット長，オペレーション}

の四つ組で表現され逆ポーリッシュ形式のテーブルになるのでシミュレータはそのままインタプリティブに実行すればよい．演算はレジスタ，メモリの状態テーブルと演算用プッシュダウンスタックの間で行われる．計算機の方式設計の一手段として性能評価に使うためマイクロプログラム制御方式のハードウェアシミュレータが開発された例[33]がある．これは対象計算機 (TM——Target Machine) のレジスタやメモリ等をハードウェアシミュレータ (HM——Host Machine) のレジスタやスクラッチパッドメモリ上に対応づけ，TM の機能は HM のマイクロプログラムによってシミュレートするもので，評価のために必要な各種の統計データもマイクロプログラムにより収集するシステムである．TM の記述は汎用計算機上の MPGS

(Micro-Program Generation System) により処理され HM 上へのマッピングと機能マイクロプログラムの作成が行われる．中型計算機の例では TM の約 40 倍の実行時間でシミュレートできたことが報告されている．

4.4 ポストプロセス

シミュレーションによりマイクロプログラムのデバッグが終了するとポストプロセスが行われる．ポストプロセスにより作られる資料には

- 制御メモリへのロードメディア
  （紙テープ，カセットテープ，カード，ディスケット等）
- PROM（プログラマブル ROM）書込テープ
- フローチャート

等がある．PROM 書込テープ作成にはマイクロプログラムの PROM チップへの割付，パリティビットの発生等の仕事も同時に行う．フローチャートは入力時にすでに配置等を決めておき，形式を整えてプリントするだけのシステムや，ページ切りや配置を自動的に行うシステム[34]が報告されている．

4.5 自動検証

マイクロプログラムが正しいことを確認する方法としてシミュレーションが使われるが，これは時間がかかる仕事であり，また虫があることは検出できても正しいことは検出できない．近年プログラムの検証の方法として主張を使うことが研究されておりある程度の成功を納めているが，同じ手法をマイクロプログラムの検証に使うことが試みられている．先に述べた Strum[22] システムがそれであるが，これはマイクロプログラムの中の各ループごとに入口，出口及びループ中にそれぞれ一つの主張を挿入しその時点で成り立つ条件を示す．これを定理証明の手法を使って等価であることがわかればプログラムの正しさが証明出来る．実際のシステムではこの証明の仕事は Theorem Prover を使って会話的に行っている．

5. 論理設計

論理装置の論理に関する自動設計は古くから研究されてはいるが必ずしも実用化が進んでいる分野ではない．論理装置の動作あるいは構造を共通の言語で正確に表現する要望から多くの研究者によって設計言語の研究が行われ，種々の言語が発表されている．これらの言語によって記述された装置の仕様を入力として論理の自動合成を行うことは産業界での強い要望であるが，人手による設計に比べ使用素子数が多くなり過ぎるためまだ研究の域を出ない．それを補うため人手により設計された論理が正しいかどうかをチェックする手段として自動論理分析や機能・論理シミュレーションが広く使われている．また設計された論理情報を設計者にフィードバックする手段として論理回路図の自動作成が実用化されている．

5.1 論理設計言語

設計者の間で装置の機能仕様を伝える手段として文章による記述やブロック図による表現が用いられてきたが，あいまいさが残るため正確な言語による表現が求められた．論理設計言語として最初のものは Iverson による APL 言語[19]である．これは計算機システム全体の機能を正式のドキュメントとして記述することを目的として作られ，方式設計者と論理設計者，オペレーティングシステムの設計者等の間の情報伝般の手段として活用された．APL は元来アルゴリズムを正確に表現することを目的として作られた言語であるため，論理機能を動作として記述するのに適している．

一方ハードウェアのレベルに近づき制御用回路の状態に応じてその動作を記述する方式の言語の一つに CDL[35] がある．CDL においてはまず Register, Bus 等データ構造を定義するハードウェアを宣言し，制御用回路の条件が成立した時それらに値をセットする表現形式をとる．この言語はマイクロプログラム制御の計算機をハードウェアと対応させて記述するのに適している．図-9 に CDL 言語の記述例を示す．

実際の設計作業においては初期の段階では余りハードウェアとは対応させず機能記述ができ，次第に設計が進むにつれてハードウェアの構造を記述できることが望ましい．このような要求を満たす言語として LDS[36),37)], TBM[38] 等の二つ以上のレベルの記述を許すものがある．例えば LDS は実装レベル，論理式レベル，状態遷移図レベル，フローチャートレベル，仕

```
REGISTER, A(0-21), B(0-21), C(0-21), F(0-5)
MEMORY,   M(C)=M(0-1077,0-21)
DECODER,  K(0-20)=F
TERMINAL, ADD=K(1), EXO=K(2)
CLOCK,    P
/ADD*P/   A=A.ADD.B, K=13
/EXO*P/   A=A*B'+A'*B
   .
   .
   .
```

図-9 CDL 言語記述例

表-2 論理設計言語の分類

状態型	動作記述型	構造記述型	デコーダ型
CDL	AHPL	ADL	ISP
DDL	APL	LDS (LEVEL 1,2)	LFM
LDS (LEVEL 3)	LALSD	MPGS	MIPS
RTL	LDS (LEVEL 4)	PMS	
TBM (T言語)		TBM (B,M 言語)	

(本文で引用したもの以外は文献48) 参照)

様レベルの5レベルの言語から構成されており，設計の進度に応じた言語を使い分けることができる．

マイクロプログラムシミュレータにおいて各マイクロ命令のフィールドの値に対応してその動作を記述するのを便利にした LFM[23], MIPS[24] 等の言語も発表されている．表-2に主要な論理設計言語の分類を示す．

5.2 論理合成と分析

ソフトウェアにおけるコンパイラの役割と類似な手法で機能レベルで記述された言語を入力として論理の発生を行い実際のハードウェアに変換するのが論理の自動合成である．IBM 社の ALERT[39] は8段階の処理を経て Iverson 言語を実装 DA の入力のレベルにまで変換するシステムである．設計作業の自動化として有力であるが人手による設計に比べて素子数が多くなり過ぎるので実用上問題があったようである．今後 LSI 化が進み素子数のコストが余り問題にならなくなると注目される技術であろう．

一方 LDS システム[36]では論理合成を言語レベル間の変換としてとらえ，高レベルの言語から低レベルの言語へ変換する複数個の自動合成プログラムを有する構成をとっている．逆に低レベルの言語から高レベルの言語への変換は論理自動分析と呼ばれるが，これはハードウェアの構造をその動作の形に変換するので設計の検証の手段として有効である．5.3 で述べる論理シミュレーションに比べ入力信号を個々に指定しなくてもよいため便利であるが，大規模な論理回路に適用するためには並列回路の検出などの抽象化の機能の開発が必要と思われる．

5.3 論理シミュレーション

大規模な論理回路の設計においてはプログラムのデバッグと同様論理のデバッグが必要である．この手段として試作機を作り実際のハードウェア上でデバッグをする手法がとられるが，不具合点を局所化するのに時間がかかること，コストが高いという欠点がある．この代りに既製の計算機上にハードウェアのモデルを作り，その論理動作をシミュレートする論理シミュレーションが古くから実用化されている．論理シミュレーションを使うと実際のハードウェアが製作されるより早い時期からデバッグが可能であること，ハードウェア上では実現が困難な任意の回路動作を実現出来ること，任意の程度まで詳細検討が可能なこことなどハードウェアデバックでは得難い種々の利点がある．高密度多層印刷基板，LSI 等修理が難しい実装方式においてはシミュレーションが不可欠である．

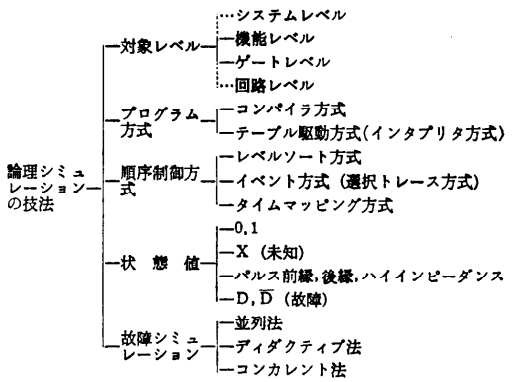


図-10 論理シミュレータ技法の分類

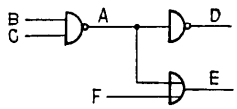
5.3.1 論理シミュレーションの目的

シミュレーションを実施する目的としては前述の論理デバッグの他に性能評価，実装面を考慮した遅延時間等のチェック，あるいは故障が存在した場合の装置の動作の予測等がある．性能評価のためには GPSS, SIMSCRIPT 等汎用シミュレーションシステムが使用されることも多い．故障シミュレーションは IC カード等の機能モジュールの故障診断データ作成システムや装置レベルでの故障自動診断データ作成システムの中で重要な役割を果している．

論理シミュレーションで用いられる技法を分類すると図-10の様になる．

5.3.2 プログラム方式

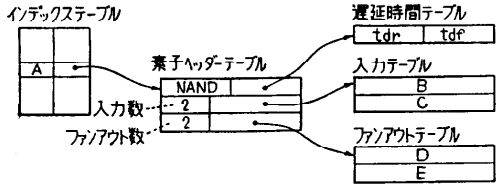
論理回路の動作をシミュレートするためには計算機中に対象のモデルを作らなければならない．その方法は大別して2種類ある．**コンパイラ方式**は回路のイメージを直接実行可能な命令の形に変換する方式で変換手順はやや複雑であるが，実行時の効率が良い．一方**テーブル駆動方式**はインタプリンタ方式とも呼ばれ，回路イメージをシミュレーション実行に都合のよいテーブル型に変換しておき，実行時にそれらを解釈しながらシミュレーションを行う方式である．実行の度に解釈が入るのでオーバヘッドが大きいがトレースの有無，遅延時間の挿入等各種の選択に対処しやすいので



(a) 回路例

命令コード	オペランド	説明
Clear Load	B	汎用レジスタへB番地の値を Loadする.
AND	C	レジスタとC番地の値のANDを行う.
Complement	—	レジスタの内容の否定をとる.
Store	A	結果をA番地へストアする.

(b) コンパイラ方式



(c) テーブル駆動方式

図-11 論理シミュレータにおける回路モデル

多くのシミュレータが採用している. **図-11** に両方式による回路モデルの例を示す.

5.3.3 順序制御方式

論理回路の中では状態の変化が並列に進行するがシミュレータの中では素子を一個ずつ順番に処理しなければならない. このためにシミュレーションの順序を決める制御が必要である. **レベルソート方式**はあらかじめ素子を信号の伝搬する順序にソートしておき入力変化のあるごとに(通常クロックごと)全体の回路のシミュレーションを行うもので同期式回路に有効である. 一方非同期回路においてはループの存在や素子ごとの遅延時間の影響のため一義的なレベルを決めるのは難しく, また回路が大きくなると実際に状態が変化する素子の割合は少なくなるのでそれらを選択的に実行する方が能率が良い. **イベント方式**ではある素子の状態が変化した時その出力先の素子をイベントキューに入れ, 時間の最も近いものから順に取り出してシミュレートする方式である. イベント処理にかなりの時間を必要とするので現時点でのイベントと時間遅れを伴うイベントとを別のキューにつなぐことが行われる[40]. **タイムマッピング方式**[41]は時間遅れを伴うイベントを能率よく処理するために考案された方式で, タイミングリングと呼ぶ時間ごとのヘッダーを持ちそれに特定時刻のインベントをリスト形式で接続する方法である.

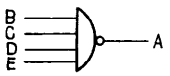
5.3.4 多値シミュレーション

最も簡単なシミュレーションにおいては論理回路の状態と同じ0と1の2値が採用されるが, 一般には未知または不確定な状態 **X** を採り入れた3値シミュレーションが広く行われている. 更にパルスの前縁 **U** や後縁 **D**[42], バス回路の高インピーダンス状態 **Z** 等を採用して効率を高めた例もある. 故障シミュレーションにおいては故障が存在した場合正常値と区別出来る状態 **D**, $\overline{\mathbf{D}}$ を採用して外部出力端子で故障の存在が検出可能かどうかを容易に判定できる[42].

5.3.5 故障シミュレーション

故障診断データを作成する時に使われる故障シミュレーションにおいては回路中の故障の数に比例したシミュレーションを行う必要があるため, 高速化の要求は非常に強い. **並列シミュレーション**[43]は計算機が語処理を行うことに着目して一回の処理で複数の入力状態または複数の故障回路のシミュレーションを行うもので多重度に逆比例した時間で処理が済む. **ディダクティグシミュレーション**[44]は複数の故障状態を計算する代りに正常値と検出可能な故障のリストを伝搬させる方法である.

図-12 に NAND ゲートにおける並列シミュレーションとディダクティブシミュレーションの様子を示した. 並列シミュレーションは語単位で論理演算を行えば出力値が計算できるので故障数が余り多くない場合には能率が良い. 一方ディダクティブシミュレーションにおいては図の場合出力の値が変化するのは入力BとCが同時に変化するような故障の時なのでBとCに含まれ, AとDに含まれない故障 f_3 のみが伝搬するという一種の集合演算を行えば良い. 故障が存在するときの回路モデルを正常な回路モデルの中に挿入して



(a) 4入力NANDゲート

	N	f_1	f_2	f_3	f_4	f_5
A	1	1	1	1	0	1
B	0	0	0	1	1	1
C	0	0	1	1	1	0
D	1	0	0	1	1	1
E	1	1	1	0	1	1

(b) 並列シミュレーション

	N	
A	1	f_4
B	0	f_3, f_4, f_5
C	0	f_2, f_3, f_4
D	1	f_1, f_2
E	1	f_3

(c) ディダクティブシミュレーション

N: 正常値
f_1-f_5: 故障 #1~#5

図-12 故障シミュレーションの例

同時にシミュレーションを進めるコンカレントシミュレーションの技法[45]も報告されている.

5.3.6 機能シミュレーション

5.1 で説明した論理設計言語の多くは機能レベル(レジスタトランスファレベルとも呼ばれる)のシミュレーションを目的としている. 論理レベルのシミュレーションと共に古くから研究, 開発の対象になっているが近年再び注目を集めている. これは MSI や LSI が多用されるようになるに伴い従来の論理レベルのシミュレーションでは時間がかかり過ぎるのでモデルのレベルを高める必要があること, マイクロプログラム制御計算機のシミュレーションに機能レベルが適していることなどの理由による.

機能レベルと論理レベルのモデルを混在させてシミュレーションする方式が報告されている[46]. また機能レベルの故障シミュレーション[47]が発表され注目を浴びたと伝えられている. Y. Chu の CDL 言語に対するシミュレータは米国で数多くインプリメントされているが, コンパイラ方式が採用されている[48]. FALOS は CDL 言語を基にした言語でテーブル駆動方式の機能シミュレータを実用にした例である[32].

5.4 論理回路図作成

設計自動化システムにおいては設計情報を設計者にフィードバックする手段として各種の形式のリストが用いられるが最も直観的で能率が良い媒体は論理回路図である. 論理回路図の自動作成は古くから実用化されており活用されている.

論理回路図の自動作成は

(1) 論理シンボルの頁への分割
(2) 論理シンボルの頁内の配置
(3) 信号線の結線

の処理によってなされるが, 美観の問題等から自動結線のみ計算機に行わせ, 頁分割及び配置は人手でやっている場合が多い.

以下前記の三機能を全て自動化したシステム[49]について処理の概要を説明する. 自動頁分割処理の場合には頁分割処理とシンボル配置とを同時に行う. すなわち設計情報ファイルから外部出力端子を1個取り出しそれを図面の右上端へ配置し, その入力素子を次々に取り出しながらレベルソートし, レベルの順に左方向へ配置して行く. 頁左端まで来たらその先のシンボルはスタックへ入れておき別の出力端子から同様の処理を繰り返す. 頁が一杯になると改頁しスタック中のシ

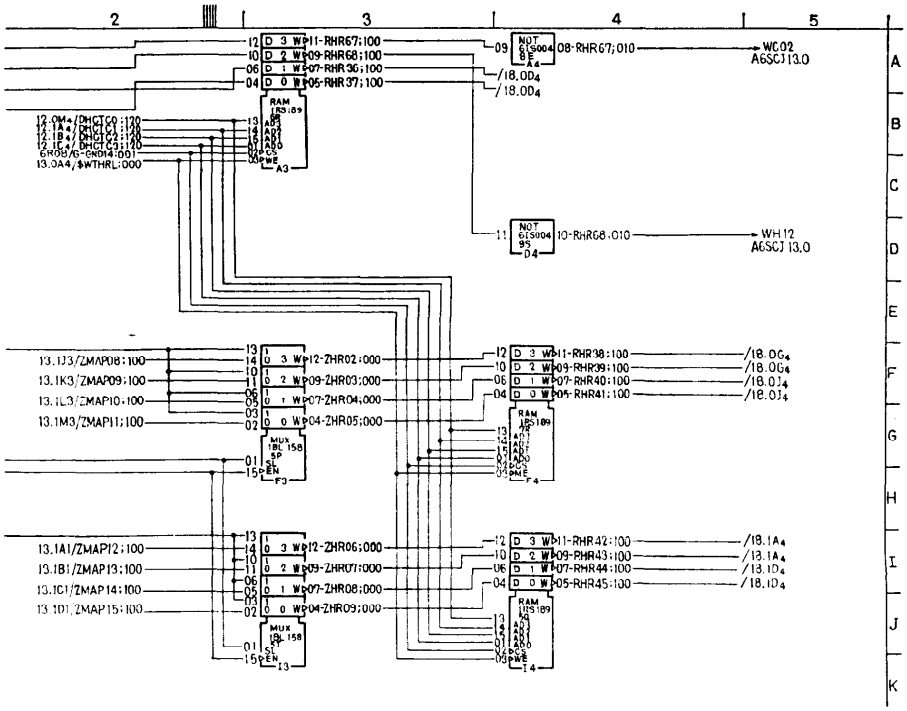


図-13 機械書論理回路図

ンボルについて配置処理を続ける．この処理では信号の流れる順に配置されるので論理が追い易い利点がある．クロスカップル NAND が上下に隣接して配置されたり孤立素子が出来にくいための工夫がしてある．結線処理はあまり複雑なアルゴリズムを用いても見易さが犠牲になるのでアンテナ法[50]に類似の発見的手法によっている．信号線間の不要な交叉が少なくなるよう結線順序や配線チャネルの選択に種々の工夫をこらしている．出力の最終図面はグラフィック COM でアパーチャーカードを作成するが，モニタ用に特殊活字の LP を用いる他，グラフィックシステムを通してディスプレーでチェック修正が可能であり，カーブプロッタで作画することも可能にしてある．出力例を**図-13**(前頁参照) に示す．

参考文献

文献 1)～16) は前号に掲載

17) 山田，川口他：マイクロプログラムの設計自動化，情報処理学会設計自動化研究会資料 74-16 (1974)
18) 倉地 正：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. 14-6, pp. 397～403 (1973)
19) K. E. Iverson: A Programming Language, John Wiley and Sons, Inc., (1962)
20) Y. Chu: Computer Organization and Microprogramming, Prentice-Hall (1972)
21) M. Hattori et al.: MPGS: A High Level Language for Microprogram Generating System, Proc. ACM National Conf., pp. 572～581 (Aug. 1972)
22) D. P. Patterson: Strum: Structured Microprogram Development System for Correct Firmwere, IEEE Trans. on Comp., Vol. C-25, pp. 974～985 (1976)
23) 元岡，新海：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究会資料 74-19 (1974)
24) 萩原，石田他：マイクロプログラムシミュレータ MIPS について，情報処理学会設計自動化研究会資料 DA-25-3 (1975)
25) C. V. Ramamoorthy et al.: A High Level Language for Microprogramming, Preprints of 6th Annual Workshop on Microprogramming (1973)
26) G. R. Lloyd: Pumpkin-(Another) Microprogramming Language, SIGMICRO Newsletter, Vol. 5, pp. 15～44 (1974)
27) P. M. Wallett et al.: Considerations for Implementing a High Level Microprogramming Language Translation System, IEEE. Computer, pp. 40～52 (Aug. 1975)
28) 馬場，藤本他：マイクログラムジェネレータ，情報処理学会設計自動化研究会資料 DA 27-1 (1975)
29) R. L. Kleir et al.: Optimization Strategies for Microprograms, IEEE Trans. on Comp., Vol. C-25, pp. 962～973 (1976)
30) T. Agerwala: Microprogram Optimization: A Survey, IEEE Trans. on Comp., Vol. C-25, pp. 962-973 (1976)
31) B. R. S. Buckingham et al.: The Controls Design Automation (CAS), 1965 Symposium on Switching Theory and Automata, pp. 279～288
32) 倉地，足立他：ファンクショナル論理シミュレータ (FALOS)，昭和 51 年度情報処理学会第 17 回全国大会，pp. 557～558 (1976)
33) 藤野，箱崎他：計算機設計評価システム——MPGS/GMPS，情報処理学会設計自動化研究会資料 73-3 (1974)
34) 川口，中沢他：汎用マイクロプログラムフローチャータ，昭和 51 年度情報処理学会第 17 回全国大会，pp. 563～564 (1976)
35) Y. Chu: An ALGOL-like computer design language, Comm. ACM, Vol. 8, pp. 607～615 (1965)
36) 元岡 達：計算機の設計自動化，情報処理，pp. 368～374 (1967)
37) T. Moto-Oka et al.: Logic Design System in Japan, 12th Design Automation Conf. Proc., pp. 241～250 (1975)
38) 萩原，黒住：計算機設計言語，情報処理，pp. 93～102 (1971)
39) T. D. Friedman et al.: Methods used in an Automatic Logic Design Generater (ALERT), IEEE Trans. on Comp., Vol. C-18, pp. 593～614 (1969)
40) 倉地 正：非同期論理シミュレータ，東芝レビュー，29 巻，pp. 636～640 (1974)
41) E. G. Ulrich: Exclusive Simulation of Activity in Digital Networks, Comm. ACM, (Feb. 1969)
42) E. W. Thompson et al.: Three Levels of Accuracy for the Simulation of Different Fault Types in Digital Systems, 12th Design Automation Conf. Proc., pp. 105～113 (1975)
43) 山田，若槻他：非同期イベント方式パラレル故障シミュレータ，情報処理，Vol. 17, pp. 569～576 (1976)
44) D. B. Armstrong: A Deductive Method for Simulating Faults in Logic Circuits, IEEE Trans. on Comp., Vol. C-21, pp. 464-471 (1972)
45) E. G. Ulrich: The Concurrent Simulation of Nearly Identical Digital Networks, 10th DA Workshop Proc., pp. 145～150 (1973)
46) 野溝，元岡：汎用論理シミュレータ，情報処理

学会設計自動化研究会資料 73-4 (1973)
47) S. G. Chappell et al.: Functional Simulation in the LAMP System, 13th DA Conf. Proc., pp. 42～47 (1976)
48) IEEE Computer Hardware Description Language 特集号, (Dec. 1974)
49) 三好, 武藤他: 論理図自動作成システム, S 50 年度情報処理学会第 16 回大会, pp. 183～184 (1975)
50) 徳永他: リスト構造を用いた回路図機械書システム, 電子通信学会交換研究会資料 (1972, 1)

(昭和 51 年 12 月 28 日受付)