# 並列処理を用いた大容量高速論理シミュレータ*

高島堅助**　加藤満左夫**　高村真司**
新井克彦**　菅野文友***　今出英雄***

## 要　約

従来の論理シミュレータ[1),2)]をさらに実用的にするために高速化，大容量化，ならびに汎用化を計った．これらを実現するために考案した方法を中心としてそのあらましを述べる．

## 序　言

筆者らは論理シミュレータを設計に利用して良い結果を納めてきたが，実時間のデータ処理システム[3)]を実用化するに当たりさらに実用的な論理シミュレータを十分に使用することが必要となったのでいくつかの方法を用いて表題のシステムを実現したものである．

論理設計の自動化を目的とする処理に対しては，論理シミュレーションの他に，論理設計トランスレータ[4),5)]などの方法が試みられているが，未だ十分実用的な段階に至っていない．また実時間動作のデータ処理システムに重要な障害検出，判断，切替，ならびに診断などの障害処理機能の設計とチェックには論理シミュレータが有用である．

従来の論理シミュレータでは，1クロック当たりのシミュレーション速度が数千個を対象にして分の単位であったのを，数万個に対して秒の単位に高速化した．さらに並列処理によって15種の入力条件に対するシミュレーションを時間の増加なしに同時に実行する方法を講じ，結果として従来のシミュレータに比して約100倍を越える速度に改善した．この速度と容量の改善は使用計算機の性能とプログラムの工夫によってなされたものである．ハードウェアによる改善部分は使用計算機の機能に負うところのものであり，内部処理の速度とメモリの容量が最も重要である．ソフトウェアによる改善は本論文の主眼である複数個(15まで)の初期条件に対するシミュレートを並列に行なわせることを採り入れた点であり，このほかシミュレーション言語とアセンブリ言語の混用を認めたことにより作業プログラム（シミュレーションを行なわせるためのオブジェクト・プログラム）の機能を拡充し，また汎用化をはかることができた．これらの改善により，Turn Around Time が著しく短縮され，論理設計のデバッグ時間はむしろ他の原因（たとえば修正すべき論理の検討など）で支配され少なくともMachine Limited ではなくなった．従来は数十ステップのテストプログラムのシミュレーションに一昼夜を要したものであった．また大容量化により2～3万個の素子数をシミュレートできるのでたいていのデータ処理装置は一度に取り扱い得る．したがってブロック分け，あるいはその相互の接続が不要であり作業を容易にする．

また，汎用性の点に関しては，論理データのファイルメンテナンスを設けたこと，AND・OR・NAND・NOR および FF（セット・リセット形フリップフロップ）を基本論理単位として取り扱えること，さらにシミュレーション言語とアセンブラ言語を混用できることが三つの大きな改善である．

なお本シミュレータの適用できる論理回路としては同期式論理回路で，規模は3万素子（1素子の入力数は9以下）までを1度に処理可能である．またシミュレートされるシステムのメモリに対して 65 K 語の凝似メモリ（一つの初期条件に対して 4 K 語）を使用することができる．

## 1. 高速化の方法

KATZ[6)]は論理シミュレーションを高速化する方法として

（イ）項の順序の再配列
（ロ）加法標準形と乗法標準形の混用
（ハ）中間結果の判定

を行なうことを提案している．

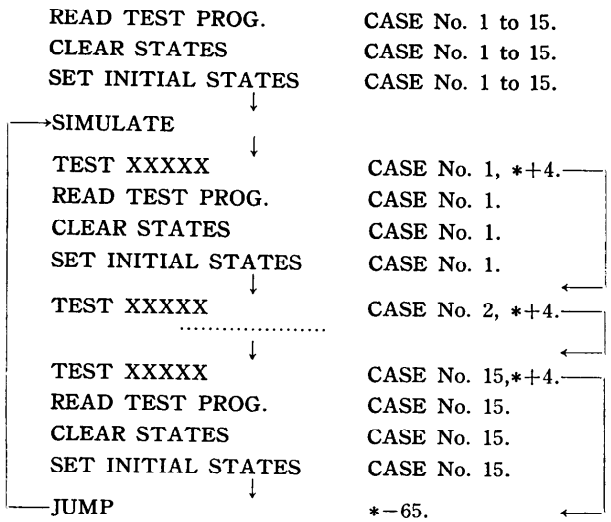
BREUER[7)]は複数個の函数を同時に処理する方法を示している．

KATZ の方法をプログラム化するとコンパイル処

---

* A Parallel Processing General Purpose Logic Simulator, by Kensuke Takashima, Masao Kato, Shinji Takamura, Katsuhiko Arai (Elecirical Communication Laboratory) and Ayatomo Kanno, Hideo Imade (Hitachi, Ltd., Yokohama)

** 電気通信研究所
*** 日立製作所




第0図 並列処理用のシミュレーション作業プログラムの一例
XXXXX はシミュレーション終了を示す素子.
*はその命令の番地を示す.

理に負担がかかり，シミュレーションの開始に至るまでの時間が増大する．彼の三つの方法の中で一番容易に考え得る（イ）にしても項による分類処理が必要で，これはコンパイル時間よりも大きい時間となるであろう．一方高速化の程度は彼の示している結果のように2倍くらいである．複数個の函数を同時に計算する方法では，変数の数が多くなると，樹枝状構造が大きくなりすぎそのままでは記憶容量の点で問題がある．

以上の方式に対して，簡単で効果のある方法にシミュレーションの並列処理がある．これは一つの函数を複数個の変数の組に対して計算するものでビット単位で論理演算のできる計算機があれば容易に実行できる．実際にシミュレーション作業においては幾種類もの入力条件に対して論理回路の動作を確める必要があるから，作業全体の高速化に役だつ．ここでの問題はその制御の方法である．

**第0図**は 15 種類のシミュレーションの並列処理を実行する作業プログラム* の一例である．作業プログラムならびにマクロ命令の細部については後述するが，まづ論理動作の検査を行なう試験プログラム**を15 種類続けて読み込む．初期状態を設定してシミュレーションを行なう．素子XXXXXによってシミュレーションの終了を監視し，特定のCASEについてシミュレーションが終了すれば，その CASE に新らたな試験プログラムを読み込んでシミュレーションを続ける．このようにすれば常にむだなく 15 種類のシミュレーションを実行することができる（素子XXXXXはたとえば計算機停止を表示する素子をとり，試験プログラムの最後に停止命令をおけばよい）．

* シミュレーションの進行を制御するプログラムで，通常マクロ命令で書かれる．シミュレーションプログラムと呼ぶことがある．

** 論理シミュレーションにより実行されるプログラム．擬似メモリにおかれ，これを変えることによって異なった命令の組み合わせによるシミュレーションが行なわれる．

## 2. シミュレーションプログラムと使用計算機

新らしい論理シミュレータは**第1表**に示す規模ならびに性能を有する HITAC 5020 型計算機を使用した．

論理シミュレーションにはシミュレーションの対象とする論理を記憶する記憶装置と，計算した各論理素子の状態を記憶する記憶装置が必要である．前者を論理表といい，後者を状態表という．論理表は逐次接近形記憶装置でよいが，状態表は，即時接近形記憶装置でないとシミュレーションを高速に行なえない．前者は一素子当たり数十～数百ビット（たとえば4入力 NAND の例で6語）を要する．後者は1素子1ビットでよいから，1語を割り当てれば，5020 の語長は 32 ビットであるから 16 状態まで記憶できる．これはあるクロック $n$ の状態 $S^{(n)}$ と次のクロックの状態 $S^{(n+1)}$ を記憶する必要からきている．

第1表 使用計算機（HITAC 5020）の規模と規格

| | | |
|---|---|---|
| 語長 | 32 ビット／語 | |
| 加算時間 | 8 μs | |
| 磁心記憶装置 | 65,536 語 | |
| 磁気ドラム装置 | 65,536語／台 | 6 台 |
| 磁気テープ装置 | 48 K 桁／秒 | 6 台 |
| 高速度印刷装置 | 1,000行／分 | 1 台 |
| カード読取装置 | 600枚／分 | 1 台 |
| カード穿孔装置 | 200枚／分 | 1 台 |

5020 の内部記憶装置は 65 K 語あるから，その半分を状態表に使用するとして約3万素子を1度に記憶できる．3万素子分の論理表は4入力 NAND で換算すると 18 万語を要するが，これは外部記憶装置である 65 K 語1台の磁気ドラムを使用すれば3台で収容できる．

シミュレーションの実行に当たっては2面のバッファエリアを内部記憶装置に用意し，ドラムから転送したコンパイルド・ロジックを一方の面で実行中に次に実行すべきロジックを他方の面に読み込む．

このときシミュレーションの実行速度は磁気ドラムのアクセスならびに転送時間 $T_1$ とコンパイルド・ロジックの実行時間 $T_2$ のどちらか大きい方の時間で決まる．$T_1$ および $T_2$ はこのシステムの場合一面分のバッファエリアを$W$語として次のようであった．

$T_1=10{,}000+20\,W$　　($\mu$s)

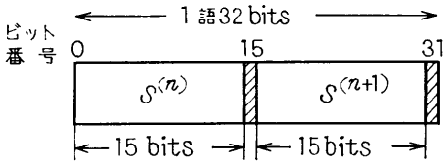
$T_2=10\,W$　　($\mu$s)

すなわち $T_1$ の方できまる．$T_1$ の中ではアクセス時間を示す最初の項が大きいから，1語当たりのシミュレーション時間を小さくするには $W$ を大きくとる必要がある．

実際には $5K$ 語のバッファを2面設けた．このとき 3 万素子のシミュレーションは最小 4 秒* を要する．

論理素子の機能をコンパイルするとき5,020の機械語には否定，ならびに論理和をとる命令がないから，否定は負数を作る命令で，論理和は論理積と否定で代行する．

このためビット 31 は常に1としておく必要がありシミュレーションには使えない．したがって可能な系列数は 15 となる．**第1図**は論理素子の状態表の形式を示す．



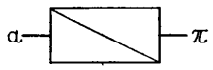

第1図　語構成と状態表の記憶形式

第2表　論理シミュレーション用基本命令（5,020 型計算機）

| 略号 | 命令名称 | REG | OPERAND | 操作 |
|---|---|---|---|---|
| CS | Complement Store | A, | $x$ | $2^{31}-(x)\rightarrow$ A　否定 |
| MB | Multiply Binary | A, | $x$ | A・$(x)\rightarrow$ A　論理積 |
| CA | Clear Add | A, | $x$ | $(x)\rightarrow$ A |
| TH | Transfer Half | A, | $x$ | A$\rightarrow x$ |

**第2表**は論理シミュレーションに使用する基本的な命令である．これらの命令を使用して AND, OR,

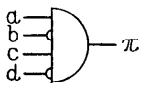
* コンパイルドプログラムを分割せずかつシミュレーション以外の時間を含まない．




$\pi=\bar{a}$

CS　A1, a*B1
TH　A1, π*B1

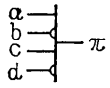
第2図　NOT




$\pi=a\cdot\bar{b}\cdot c\cdot\bar{d}$

CS　A1, b*B1
CS　A2, d*B1
MB　A1,A2
MB　A1, a*B1
MB　A1, c*B1
TH　A1, π*B1

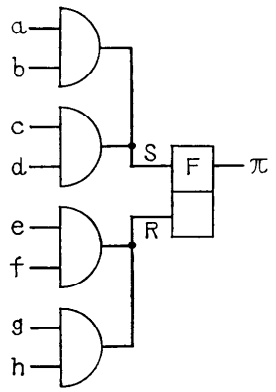
第3図　AND




$\pi=a+\bar{b}+c+\bar{d}$
$=\overline{\bar{a}\cdot b\cdot\bar{c}\cdot d}$

CS　A1, a*B1
CS　A2, c*B1
MB　A1, A2
MB　A1, b*B1
MB　A1, d*B1
CS　A1, A1
TH　A1, π*B1

第4図　OR




S=ab+cd
R=ef+gh
$\pi=\overline{R(S+F)}$
$=\overline{\bar{R}\cdot\bar{S}\cdot\bar{F}}$

SIML PROG

CA　A1, a*B1
MB　A1, b*B1
CS　A3, A1
CA　A1, c*B1
MB　A1, d*B1
CS　A5, A1
MB　A3, A5…$\bar{S}$
CA　A1, e*B1
MB　A1, f*B1
CS　A4, A1
CA　A1, g*B1
MB　A1, h*B1
CS　A5, A1
MB　A4, A5…$\bar{R}$
JS　FFSR, *B3

FFSR

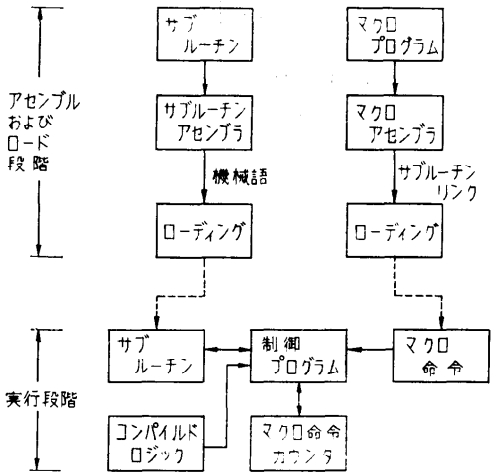
CS　A1, A3
CS　A2, A4
MB　A1, A2
JNZH　A1, *B2…ERROR
　　S=R=1
CS　A1, π*B1
MB　A1, A3
CS　A1, A1
MB　A4, A1…F
VCA　A4) 0,
　　π)16*B1, 15
JC　*B3
JS　Subroution Jump 命令
JNZH　Jump on Non Zero 命令
VCA　Variable Length Clear Add 命令
JC　Return Jump 命令

第5図　フリップフロップのシミュレーション

NAND, NOR および FF などの基本論理素子のシミュレーションは**第2図～第5図**のように行なう．図中 #*Bi* はインデックス *i* の指定で B1 の内容はどちらの半語が前の状態 $S^{(n)}$ であるかを示している．FF 以外の論理シミュレーションは $S^{(n)}$ より $S^{(n)}$ に行ない，FF のシミュレーションは $S^{(n)}$ より $S^{(n+1)}$ に行なう．FF のセット側入力を *S*，リセット側入力を *R* とすると $S=R=1$ であることは許されない．このときは誤りとして表示を行なう．FF の誤りの検出ならびに結果の記憶はサブルーチンとして共通化しコンパイルド・ロジックを節約した．

## 3. シミュレーションの実行制御

シミュレーションの実行に当たっては論理函数値の計算以外に入力条件の設定，結果のとり出しを行なうプログラムが必要である．これらの作業はシミュレーション全体の能率を左右するから前節のプログラムに劣らず重要である．一方これらの作業に使用するプログラムならびにそのパラメータは対象とする回路，試験条件によって異なってくるから，それぞれに応じてプログラムを作り変え，取り換えておいたのではプログラム手数，運転共に取り扱いが困難になる．



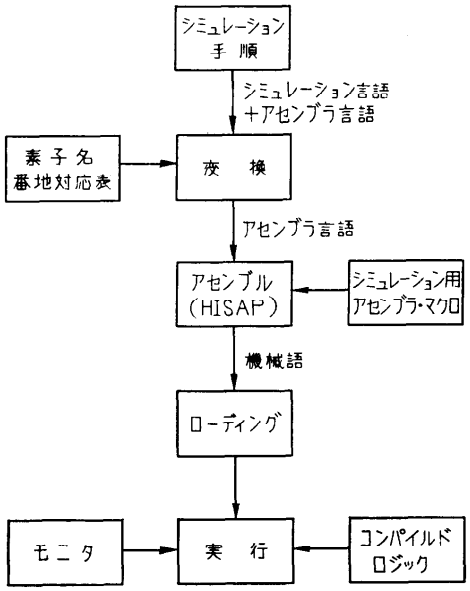

第6図　解釈マクロ命令法式によるシミュレーションの実行制御

この問題の解決にマクロオータ方式[1),2)]を使用して効果を納めた．すなわち，**第6図**に示すようにシミュレーション作業に必要なプログラムはすべてサブルーチンとして作成しておき，シミュレーションを行なう者はマクロプログラムによって使用するサブルーチンの種類，サブルーチンに与えるパラメータ，ならびにその実行順序を指定する．アセンブラによってマクロプログラムを読み込んだあと，制御プログラムを起動する．制御プログラムはマクロプログラムに従って必要なサブルーチンを呼び出し実行する．

この方式によって作業手順はかなり自動化されたが未だ若干の不便が残っていた．その一つはマクロプログラムはすべて登録したシミュレーション言語で書かなくてはならないことからきている．すなわち，システムに何かわずかの変更を加えたいときでも登録作業を介す必要があり，新らたなサブルーチンのアセンブルとマクロオーダのアセンブルを別に行なわねばならずその時間ならびに作業の誤りのためにむだな時間が使われていた．

これに対して新しいシステムではシミュレーション言語と一般のアセンブラ言語との混用を許し，この問題を解決した．さらに，これらのプログラムが素子の状態を参照するのにすべて素子名称によるシンボリックアドレスが使用できるようにしてその効果を高めた．以前のシステムでは素子の状態を参照するのに絶対番地を使用しなければならないところがあり，素子数の変更が生ずると関係するプログラムをリアセンブルする必要があった．

上記の処理は**第7図**に示す手順で実行する．まず，



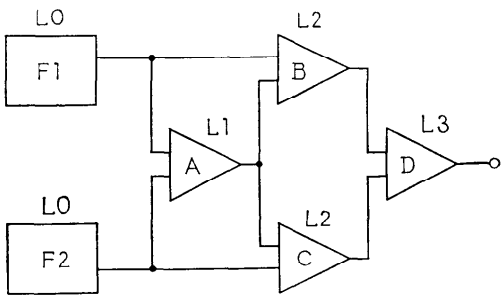

第7図　コンパイル・マクロ方式によるシミュレーションの実行制御

シミュレーション言語およびアセンブラ言語で書いたシミュレーションプログラムをアセンブラ言語に変換する．このとき素子名称と番地対応リストにより素子の状態を参照するシンボリックアドレスを絶対番地に変換する．変換されたアセンブラ言語を HISAP アセンブラでアセンブルしてロードし実行する．シミュレーションサブルーチンは HISAP のマクロとして取り扱い，その本体はアセンブルの際に追加する．この変換とアセンブルはモニタの制御によりオペレータを介さず連続して実行するようにし，シミュレーションプログラムのローディングを著しく簡略化することができた．

## 4. コンパイルとレベル・ソート

論理函数は素子ごとに1枚のカードを使用し，その素子名，論理機能，ならびに入力変数となる素子名を記入してある．論理シミュレーションを行なうためには，この論理函数を第2図～第5図のプログラムに変換しなければならない．これをコンパイルという．

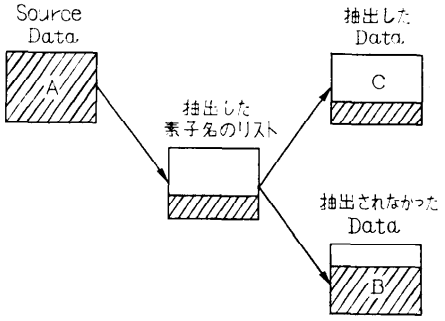
コンパイルを行なうに先だって，プログラム上いくつかの問題がある．その一つはコンパイルしたプログラムを頭からコンパイルの順に実行するために論理素子を論理レベルの順に排列することである．これをレベル・ソートという．論理レベルとは**第8図**に示すようにフリップフロップをレベル0とし，レベル $i$ 以下




$F_1$, $F_2$ は記憶作用を有する素子
A, B, C, D は記憶作用を持たない素子は $L_0$～$L_8$ 論理レベル

第8図　論理レベル

の素子を入力とする素子をレベル $i+1$ として定義する．レベル・ソートを行なうにはまずフリップフロップを論理カード中から抽出して，その素子名をレベル0のリスト L0 に登録する．次に L0 にある素子のみを入力とする素子を抽出してこれを L1 とする．L0 と L1 より L2 を作り，以下同様に繰り返えす．そして遅延時間から許される段数 $n$ に至るか，$n$ 以下のレベルで抽出できる素子がなくなったときにレベル・ソートを終了する．このとき抽出されずに残る素子があればそれは $n$ 段を越えているかあるいは論理素子だけでループを成している回路で素子のレベルが定義できないものである注)．このようにシミュレーションに至る前に論理上の誤りを指摘できる．**第9図**にレベル・ソート処理を示した．1段終了するとAとB




1段終了するとAとBを交換して次の段の抽出を行なう．

第9図　レベル・ソート

を交換して次の段の抽出を行なう．Cにソートされたデータができる．新しいシステムではプログラムに工夫を行ない，ソートに磁気ドラムを使用するので高速である．実際 20,000 素子のレベル・ソートは約7分で終了する．以前のシステムでは5,000素子のレベル・ソートに通常5時間程度を要し，これを越える素子数ではほとんど不可能に近い作業であった．レベル・ソートを終了した論理情報はレベルの順に排列され，その順序にコンパイルする．

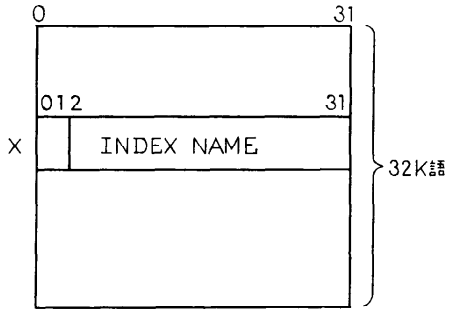
レベル・ソートならびにコンパイル処理では素子名称よりその素子の状態アドレス，あるいはレベルその他その素子に関する情報を記憶しているアドレスを高速に求めることが必要である．従来のシステムではテーブル・ルックアップ法を用いていたが，新しいシステムでは次のような方法を用いて高速化した．すなわち素子名称は5文字 30 ビットの情報を有しているが，対象とする素子数は4万を越えないからそのおのおのを区別するのに 15 ビットの情報があれば十分である．よって，30ビットの情報に適当な変換を施してできるだけランダムに分布するように 32 $K$ の領域に割りつける．すでに割りつけてある素子がある場合

(注)　間に FF の入らない論理素子のみのループは本方式のような同期式ではシミュレートできない．このほか論理ハザードなどの検出も行なっている．

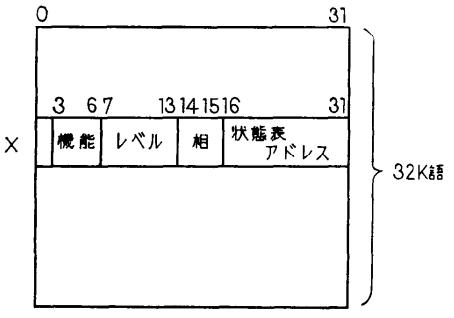
は，その場所から番地を一つづつ増して空いている番地を探す．こうしてできたリストを INDEX NAME LIST (INL) という．Peterson[8] はこのような索表方式においてメモリ利用率と平均探索回数との関係を求めているがその結果を利用すれば，3万個の素子では4回である．

上記の変換を入力変数についても施し，論理情報自体も半分に圧縮する．ソートの対象とする情報を圧縮することによってドラムの使用を可能とし，また転送回数が減る．実際に使用した例では4入力の素子の情報が4語に圧縮された．したがって同じ素子3万個を収容するためには6.5万語のドラム2本を使用すればよい．

レベル・ソートを高速化するためには，各入力変数についてそのレベルが定ったか否かを索表する操作が問題である．この操作を3万個の素子についてテーブル・ルックアップ法を用いたのでは1回の索引に平均 1.5 万回の表アクセスを要する．レベルの定ったものだけを別のリストにしてそのリストを参照するにしても後段のレベルに至ればアクセス回数は増加してくる．




第 10 図 (a) INDEX NAMELIST

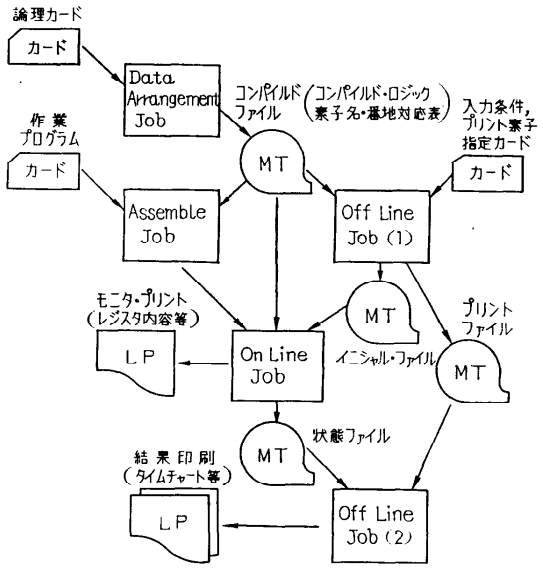



第 10 図 (b) NUMERICAL INDEX LIST

この問題を解決するために，圧縮された素子名を番地に対応させ，その番地にレベルの決定の有無および決定された場合にはそのレベルを記入したリストを使用する．これを NUMERICAL INDEX LIST(NIL) という．NIL を用いれば入力変数1個につき1回のアクセスで前段レベルの決定の有無を知ることができる．INL と NIL を**第 10 図**に示す．

## 5. 処理の流れ

システム全体の処理の流れは**第 11 図**に示すように




第 11 図 システム全体の処理の流れ

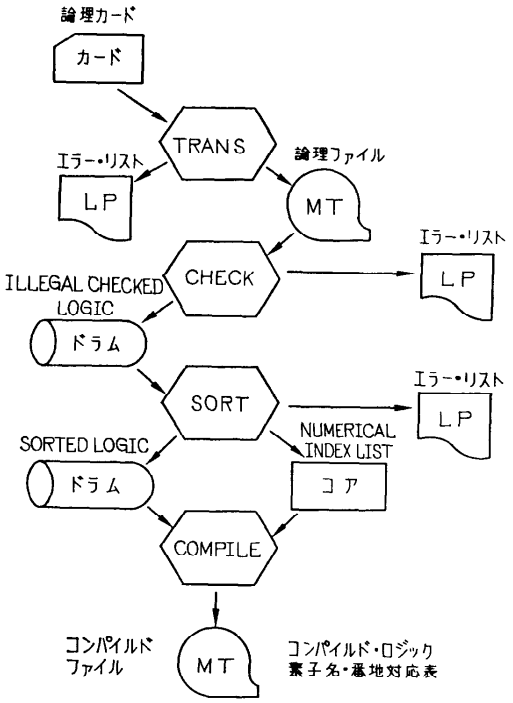
(i) Data Arrangement Job
(ii) Assemble Job
(iii) On Line Job
(iv) Off Line Job (1) および (2)

から構成されている．

### 5.1. Data Arrangement Job

これは論理カードを読み，誤りをチェックし，ソートを行なってコンパイルドファイルを作るまでの処理である．この一連の処理は

(i) TRANSFER
(ii) CORRECT
(iii) EXTRACT
(iv) CHECK
(v) SORT
(vi) COMPILE

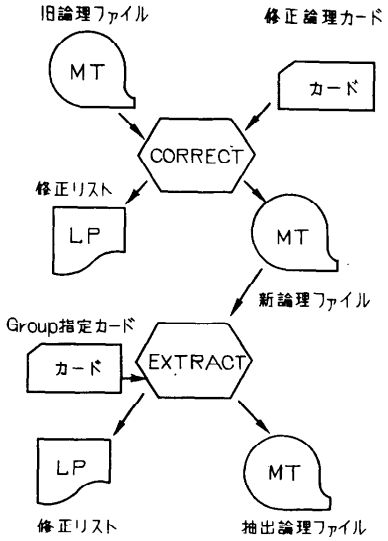



第 12 図　Data Arrangement Job

の6種のプログラムをその順に使用して実行する．このうち (ii) と (iii) は常に必要なものでないのでそれらを除いた処理の流れを**第 12 図**に示す．

まず TRANS 作業で論理カードを読み，論理情報を磁気テープに移す．このときカードに記載された情報のフォーマットチェックを行ない，INDEX NAME LIST を作る．次に CHECK 作業によって素子接続関係に関するチェックを行なう．ここで正規でない入力変数または定義されていない外部端子が検出される．チェックの完了した論理情報をドラムにおいて SORT 作業に引きつぐ．SORT を終了すると SORT した論理情報をドラムに置き，NUMERICAL INDEX LIST をコアに置いて COMPILE に渡す．SORT 作業でも CHECK 作業では検出できなかった種類の素子接続関係の誤りが検出される．COMPILE 作業では最終的に論理情報をコンパイルし，素子名・番地対応表をつけて磁気テープに書き込む．

CORRECT あるいは EXTRACT は論理情報の修正，あるいは一部分の抽出に使用する．これらは通常のデータ処理におけるトランザクションの処理と同様




第 13 図　CORRECT および EXTRACT

である．処理の流れを**第 13 図**に示す．

### 5.2. Assemble Job

Assemble Job はシミュレーション言語とアセンブラ言語で記述した作業プログラムカードを読み込み，アセンブラ言語に変換してから HISAP でアセンブルし，ロードし，シミュレーションの準備を行なう．Assemble に際しては，シンボリックで書いた素子名を番地に変換するために，番地対応表の入ったコンパイルドファイルも必要である．

### 5.3. Off Line Job

シミュレーション作業のうち，入力条件，プリント素子の指定，ならびに結果の取り出しにも 5.2 と同じ理由で番地対応表が必要である．これは素子数だけの語数，すなわち3万素子に対して3万語を要するので同じ語数を有する状態表と同時にコア上におくことができない．

一方これらの作業はシミュレーションの実行のあとまたは先に行なう性質のものであるので Off Line Job として独立させた．Off Line Job に使用する作業プログラムはだいたい決っているのでアセンブルの終了した標準的なものを準備し使用の便を計った．

入力条件を指定する Off Line Job では設定すべき素子名，状態（1または0），および状態をセットすべきクロックを記入したカードを読み，イニシャルファイルを作る．

プリントすべき素子はプリントファイルで指定される．したがって，結果の印刷に先立ってプリントファイルを作らねばならない．プリントファイルの作成に際して印刷すべき素子の順序を

（i） カードの穿孔順

（ii） アルファベット順（カード順に関係なく）

のどちらかに指定できる．

結果の印刷を行なう Off Line Job では On Line Job の結果である状態ファイルとプリントファイルを入力とする．印刷形式として

（i） タイムチャート形式

（ii） ON 素子名形式（ON である素子の名称）

（iii） 変化素子名形式（状態を変えた素子の名称）

の3者のいずれかが撰択できるが，プリントファイルはいずれにも共通に使用できる．

### 5.4. On Line Job

On Line Job には Assemble の完了したシミュレーションプログラムとコンパイルドファイルならびにイニシャルファイルが必要である．

シミュレーションの結果として各クロックごとの状態表の内容を磁気テープに書き込むことができる．これを状態ファイルという．また，モニタ・プリントとしてシミュレーションの進行の大まかな状況をラインプリンタに印刷する．たとえば主たるレジスタの内容などがモニタに役だつ．これらはいずれも作業プログラムで指定する．On Line Job に使用するマクロの種類を**第3表**に示す．

第3表 On Line Job のマクロ命令

| | 種 別 | マ ク ロ 命 令 |
|---|---|---|
| 1. | 状態のテスト，設定 | SET, RESET, CLEARST, SETIST, TESTST, COMRARE |
| 2. | 結果の取り出し（On-Line） | DUMPAS, PRINTST, PRINTRG |
| 3. | シミュレーション | SIML, MOVEST |
| 4. | 制 御 | RAISECL, SETCL, TESTCL, JUMP, HALT, SETES, TESTES, |
| 5. | そ の 他 | READPM, TRANSPM, DEF |

On Line Job のマクロに特徴的なことは擬似メモリ関係の機能であろう．擬似メモリによってシミュレーションの対象とする論理回路にその外部との情報交換を含めてシミュレートできる．疑似メモリとして 65 $K$ 語のドラム1台を割りあてたので 1 CASE 当たり 4,352 語の記憶容量を持ち，たいていの論理シミュレーションには十分な値と思われる．

この機能を実行するために READPM と TRANSPM の2種のマクロ命令がある．前者は入力条件を疑似メモリにあらかじめ読み込む処理を行ない，後者は対象とする論理回路と疑似メモリとの情報交換を行なう．情報交換を行なうには入出力端子となる素子をひとまとめとして名称を与え，これを別のマクロ DEF で定義しておき，TRANSPM ではその名称と READ, WRITE の別，タイミングを決める素子名，ならびに遅延時間（クロック数）をパラメータとして与える．

## 6. 誤りのチェック

データ処理に際して常に誤りのチェックが問題になるが，論理シミュレーションシステムにおいても同様である．ここでは特に論理情報に関する誤りのチェックについて述べる．

論理情報に関する誤りのチェックは次の4段階に分けることができる．

（i） さん孔誤りのチェック

（ii） フォーマット・チェック

（iii） 素子接続関係のチェック

（iv） シミュレーションによるチェック

### 6.1. さん孔誤りのチェック

設計者による論理設計の結果を設計カード（ホールソートカード）に記載することは前に報告したシステム[1]と同様であるが，機械処理のためにカード（IBM カード）を使用したところが異なる．さん孔したカードは検孔機にかけて検孔する．さん孔誤りはカード枚数で約1％程度であった．

### 6.2. フォーマット・チェック

プログラムにより行なう最初のチェックはフォーマット・チェックである．フォーマット・チェックの項目を**第4表**に示す．これらは1枚のカードの中で情報の記載について約束してある規定との合否を調べるものである．ただし⑨は INDEX NAME LIST への登録の際に見い出される．

### 6.3. 素子接続関係のチェック

フォーマット・チェックを完了すると素子接続関係のチェックを行なう．素子接続関係のチェックは，2個またはそれ以上の素子間の接続が正規であるか否かを調べるものである．そのチェック項目を**第5表**に示す．

この中で①と②は入力変数，すなわち前段素子を調

第4表　フォーマット・チェック項目

| No | 表　　示 | 内　　容 |
|---|---|---|
| ① | Identification Error | 論理カードである表示がない |
| ② | Group Error | 同一 Group 内に他の Group がある |
| ③ | Sequence Error | Sequence 番号が順序どおりでない |
| ④ | Fan-in Error | Fan-in が規定数を越えた |
| ⑤ | Pairless Error | FF の Set 側または Reset 側ゲートがない |
| ⑥ | FF Gate Error | ゲート数の表示とゲート数の不一致 |
| ⑦ | FF Phase Error | 機能欄と入力欄のクロックの相の不一致 |
| ⑧ | Double Index Error | 同一名称の素子が2個以上ある |

第5表　素子接続関係のチェック

| No | 表　　示 | 内　　容 |
|---|---|---|
| ① | Sourceless Error | 入力変数に対応する素子がない |
| ② | -Sign Error | FF 以外の素子が負の入力変数として使用されている |
| ③ | Fan-out Error | Fan-out が規定数を越えた |
| ④ | Common Coll. Error (1) | コモン・コレクタ以外に出力がある |
| ⑤ | Common Coll. Error (2) | FF の出力がコモンコレクタに接続されている |
| ⑥ | Level Unknown | レベル・ソートで順序づけできなかった |
| ⑦ | Abnormal FF | FF の入力の相とクロックの相が一致している |

べることによって行なう．②～⑤は後段素子を調べる．⑥および⑦はレベル・ソートを終了したときその有無がわかる．

### 6.4. シミュレーションにおけるチェック

6.2 および 6.3 のチェックは Data Arrangement Job 中で行なう．ここまでの段階で論理素子の形式上および電気的の誤りが論理動作の如何に拘わらず指摘される．これに対してシミュレーションにおけるチェックは論理動作の実行中あるいはその結果に対して行なうもので，前者を静的なチェックとすれば，これは動的なチェックである．シミュレーションにおけるチェックの項目を**第6表**に示す．

FF Error は FF のセットおよびリセットゲートが共に1である場合を示す．このシステムでチェックする Hazard とはある相の FF の変化が同相の FF の変化を招いたことを示すもので，このような場合に

第6表　シミュレーションにおけるチェック

| No | 表　　示 | 内　　容 |
|---|---|---|
| ① | FF Error | FF のセットおよびリセット・ゲートが共に1である |
| ② | Hazard | ある相の FF の変化が同相の FF の変化を招いた |
| ③ | Compare-Out | 論理動作の進行が仕様と一致しない |

は遅延時間の大小によって装置が確率的な動作をする．この誤りは同一クロックを2回シミュレートして結果を比較することでチェックできる．これらの誤りは On Line Job で検出される．

一般に論理動作が仕様どおりであるか否かは，モニタ・プリントあるいは結果印刷から得られるタイムチャートなどを視察により判定するのであるが，検査すべきデータが多量になると作業に時間を要し誤りを見逃す恐れもでてくる．われわれの場合にはプログラムシミュレータが先だって完成していたのでこれを利用した．すなわちプログラムシミュレータに試験プログラムをかけて1命令ごとの演算結果（レジスタ内容）を磁気テープに記録し，同じプログラムを論理シミュレータで実行しその結果と照合する．この方法によって検査を自動化することができた．照合は Off Line で行なう．

## 7. 結　　言

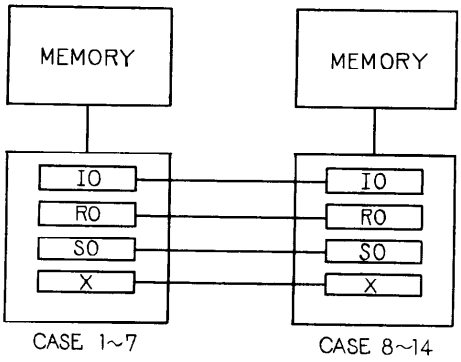
新しいシステムの完成後これを先に述べた実時間のデータ処理システムのシミュレーションを行なって，その論理デバッグその他の作業に使用している．

素子数は2万，コンパイルドプログラムの大きさは8万語であった．1素子当たり平均4語で入力数の少ないものが多いことによる．1クロック当たりの綜合シミュレーション時間（レジスタ内容の印刷，状態表の書き出しを含めて）は6秒であった．

約 20 人を越える人達がこの論理シミュレータを使用しているがほとんどの人が並列処理を使用した．したがって現在のシステムでは並列処理の使用上の難しさはないと考えられる．

約10,000ステップ近くの試験プログラムを作り，プログラムシミュレータであらかじめ結果をもとめ，論理シミュレータの結果と比較している．この検査により，2万素子に対して約5%の誤りが発見された．

現在並列処理を利用して重複系のシミュレーションを行なうことを計画中である．すなわち**第 14 図**に示




第 14 図 複数プロセッサのシミュレーション

すようにおのおのメモリを有し，レジスタ間で照合を行ないながら動作する同型の2台のデータ処理装置はただひと通りのコンパイルドプログラムを使用し，CASE 間で情報交換を行なうマクロ CONVERT を使用してシミュレーションできる．

## 謝 辞

終始御指導頂いた電気通信研究所，岸上，藤本各室長に厚く御礼申し上げます．また本システムの検討に有益な御教示を頂いた富士通信機製造株式会社山田博，中村洋四郎，黒田昭の各氏，ならびに本システムの作成の一部に御尽力頂いた日本電気株式会社西村愿郎氏に深く感謝致します．

## 参考文献

1) 高島，津田他：論理構成のシミュレーション・プログラム，情報処理 Vol. 4, No. 2, 1963 年3月
2) 高島，加藤他：論理シミュレータ LSS 4，電気通信学会交換研究会 1965 年 2 月
3) 高島，加藤他：プログラム記憶式電子交換機における長語短語の切替と多段先行制御方式，電気通信学会計算機研究会 1965 年 6 月
4) R.M. Proctor: A Logic Design Translator Experiment Demonstrating Relationships of Language to Systems and Logic Design, IEEE Trans. EC-8 Aug. 1964.
5) H.P. Schlaeppi: A Formal Language for Describing Machine Logic, Timing, and Sequencing (LOTIS) IEEE Trans. EC-8, Aug. 1964.
6) J.H. Katz: Optimizing Bit-Time Computer Simulation, Comm. ACM Vol. 6, No. 11, 1963.
7) M.A. Breuer: Techniques for the Simulation of Computer Logic, Comm. ACM Vol. 7, No. 7, 1964.
8) W.W. Peterson: Addressing for Random Access Storage, IBM J. Vol. 1, No. 2, 1957 年4月

（昭和41年2月1日受付）