超大型機の演算制御*

中沢 喜三郎** 石原 孝一郎**

1. まえがき

超大型機における演算制御の発展の歴史は，あくことのない処理速度——それもプログラマに特殊なコーディングを要求しないで，汎用プログラムでの処理速度の追求にあったといえる．一方，年代が進むにつれて計算機が社会のあらゆる分野で使用され始めたこともあって，いわゆる可用性 (Availability) に対する要求がきびしくなってきた．これはある意味では処理速度の向上と矛盾する面もふくんでおり，設計者に課せられた難問がふえたことを意味する．このことを念頭において，演算速度の向上に関する種々の技術を；

（1） 演算制御系の構成方法，
（2） 演算実行の前準備，
（3） 演算そのもの．

の3種に分類して以下に述べる．

2. 演算制御系の構成方法

制御系の構成法による性能の向上は，誰しも思いつくとおり，パラレリズムを導入するのが，まずとられる手段であるが，その並列処理が時間的であるか，空間的であるかによって次のように分類できる．

（A） Single Instruction & Single Data Stream.（通常の制御方式）
（B） Single Instruction & Multi Data Stream.（空間的並列処理，Processor Array 方式）
（C） Pipe Line Control.（時間的並列処理）
（D） Multi Instruction & Multi Data Stream.（時間的な処理，空間的な処理）

（A）は通常の制御方式および多少の先行制御を行なう方式であり，その単純明快さのゆえに従来 IBM 360/75 などほとんどの大型機が多少なりとも採用してきた方式である．これを一段発展させたものが，（A）の方式のものをマルチプロセッサで用いる方式であるが，この場合でも，その構成要素である個々のプロセッサの性能を向上させることが基本となることはいうまでもない．

このマルチプロセッサ・システムを極端におし進めたときに二つの可能性が出てくる．その一つが(B)であり，もう一つは(D)である．(D)を安直に実現しているのが，従来のいわゆるマルチプロセッサ・システムであるが，これは従来高い可用性を要求される場合に試みられてきたが，性能を追求する目的で構成されたことは割合，少ないようである．これは主メモリにおけるプロセッサ間の競合や，ソフトウェアのオーバーヘッドのため，たとえば2プロセッサ・システムの場合に，システムの性能はシングル・プロセッサの2倍とはならず，1.8 倍程度[1] にしかならないことなどの理由による．しかしトータル・システムとしてみた場合，プロセッサ間のコミュニケーションの簡易さ，入出力機器（特にファイル類）の取扱いの一様さ，そして，一つの OS によって，全システムが統一的に管理できるなど，独立した二つのシステムには見られない特徴があるので，今後大いに発展することが期待される．

(B)のプロセッサ・アレイ・システムはまだ二，三の試みが行なわれているに過ぎないが，その中でも最も有名なのが ILLIAC Ⅳ[2] である．このシステムの概略構成は**図1**に示すように制御ユニット (CU) が命令を解読して64個の処理エレメント (PE) に送出し，

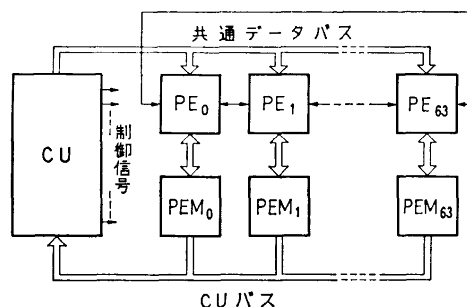

Fig. 1 ILLIAC Ⅳ Array Structure.

* Arithmetic Control in Supercomputers, by Kisaburo Nakazawa and Koichiro Ishihara (Central Research Laboratory, Hitachi Ltd.)
** 日立製作所中央研究所

これを受けた処理エレメントは，おのおのメモリ(PEM) からオペランドを取り出して実行する．PEM にある命令や共通オペランドはCUバスを通して CU に送られる．図1の構成を一つのアレイと呼び，ILLIAC Ⅳ 全体としては四つのアレイから成るという大規模なものである．以上の構成から理解されるごとく，大きな行列や偏微分方程式を扱う問題，2次元以上のパターン・データの処理などに偉力を発揮することが期待されるが，真の評価はソフトウェア[3] の開発後に受けることになろう．

(C)のパイプライン・システム[4] はプロセッサの内部を複数個のユニットに分割し，各ユニットでは一つの命令の処理を次のユニットへ渡したら直ちに後続の命令の処理を開始することにより，各ユニットを時間的にあきなく使おうという考え方である．したがって，前述の空間的な並列処理と矛盾するものではなく，むしろ併用してこそ一層の性能向上が望めるものである．この考えは非常にすっきりしており IBM 360/195, CDC 7600 など最近の超大型機はほとんど採用しているが，これを徹底して取り入れたのが CDC の STAR である．**図2** に STAR の処理装置の概略構成を示す．図で Origin Streaming Unit と Destination Streaming Unit が命令およびオペランドをパイプ・ライン方式で各ユニットに供給し，また結果をメモリにもどす制御を行なっている．

パイプライン・システムの設計上の第1の要点は，各ユニット間の処理能力（パイプの太さ）のバランスをとることである．個々のユニットの処理能力の目安としてマシンサイクルがあげられるが，これは 80 ns (360/85) から 27.5 ns (7600) ときわめて高速になっている．

第2の要点はパイプの流れをみだす要因にどう対処するかという点である．パイプの流れをみだす要因としては割込み発生など特殊な場合を除けば，分岐命令とプログラムの命令相互間の依存性（インデックス・レジスタやオペランドの書替えなど）がおもなものである．分岐命令については，分岐条件が確定するまで分岐先とそうでないほうの両方の命令を用意しておくのが超大型機では一般になっている．また命令相互間の依存性については，命令のデコード時点でこれを検出して依存性のある場合にはシーケンシャルな制御に移行するのが普通であるが，レジスタ・オペランドの共通バスによる制御[5] やストアするオペランドを続いて演算に使う場合，メモリに書き込む前にメモリの制御のところで置き替える方法[6]，さらには複機個の命令を依存性がない限り並列にデコードしていく考え方[7] など幾つかの技術が試みられている．いずれにせよ，従来はソフトウェアからの助けなしに，純ハードウェア的に解決しようとしてきたわけであるが，ソフトウェアに若干の制限を設ければ楽に解決できる点もあり，今後は両方からのコンプロマイズが必要となろう．

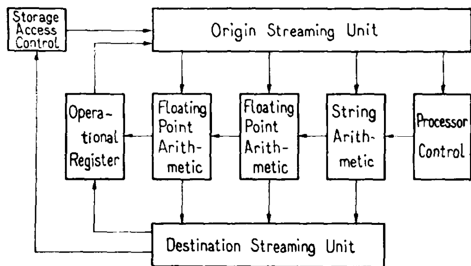

Fig. 2 STAR Processor.

3. 演算の準備

個々の演算の方式について述べる前に，演算実行の準備として命令やオペランドの読出しと実行順序の制御についてふれておこう．

3.1 命令およびオペランドの先取り

大型機の分野で古くから先行制御という名で呼ばれている制御方式の主題は，CPU の速度とメモリのそれとのギャップをいかにして埋めるか，いいかえれば命令やオペランドをいかに能率よく先取りし，これを演算器に供給するかという点にあった．**図3** に代表的

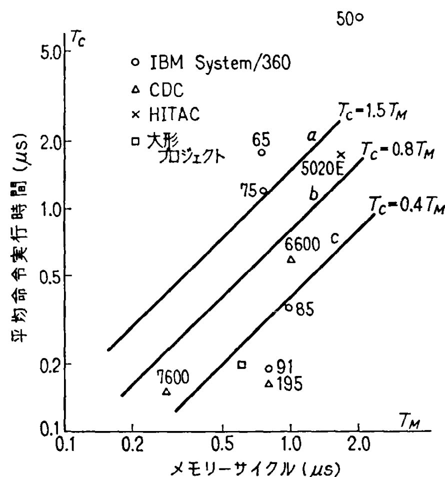

Fig. 3 Memory Speed vs CPU Speed.

な機種のメモリ・サイクルと CPU の性能の関係を示す（なお性能は筆者らの試算によるもので，公表された値ではないことをお断わりしておく.）．図で直線 a より上に属する機種ではほとんど先行制御を行なっていない．このことは1命令の実行に要するメモリ参照回数が大型機で 1.2 回程度（命令 0.5回，オペランド 0.7 回），中型機でも2回程度であることから直線 a より上の領域ではメモリ・リミットよりは，演算リミットとなることからも当然といえる．同様の観点から，直線 a と b の間では比較的素直な先行制御ですむが，直線 b より下の領域では高度の先行制御を必要としている．なお直線 c より下の機種は 360/91 以外はバッファ・メモリ方式を採用している点は注目すべきである．以下 360/91 と 360/85 の制御方式について述べよう．

（1） 360/91 の先行制御[6),9)]

360/91では図3からもわかるとおり，性能に比してメモリ・スピードがかなり遅く，これをカバーするため大がかりな先行制御が行なわれている．その中心をなしているのは命令バッファ (IB) とオペランドバッファ (OB) および主記憶制御ユニットである．命令バッファは8バイトのレジスタを8個持ち十分な先取りを行なうほか，この中におさまるプログラム・ループ (16 ステップ程度) なら2回目以後はメモリを参照することなく実行することができる．主記憶制御ユニットは最大 16 way にインターリーブされたメモリと CPU の間に介在し，おもに次の部分から構成されている．

a）受入れスタック　　現在メモリで読出し中の要求に関する情報を記憶する．5組のアドレス・レジスタその他から成る．

b）要求スタック　　メモリのビジーが解けるのを待っている要求を記憶する4組のアドレス・レジスタ．

c）ストア命令の処理をするための3組のアドレスおよびデータ・レジスタ．

また，これらの間で同一のアドレスを参照していないかなどのチェックも行なっており，きわめて複雑な制御となっている．

（2） 360/85 のバッファ・メモリ方式[10)]

主メモリのスピードが CPU のサイクルにマッチしたものであれば，前述のような複雑な先行制御を必要としないわけであり，事実上これに近い形で実現したのがこのバッファ・メモリ方式である．すなわち CPU 内部に比較的小容量 (4～32 kB) で高速 (50～100 ns) のメモリを持ち，ここへは主メモリの写しの情報を記憶する．普段はこのバッファ・メモリから情報を読み出すが，たまたま必要とする情報がバッファ内になかったときには，主メモリからこの情報を含む一連の情報（ブロック，32～128 B 程度）を高速に転送して格納しておく．この方式の計算機の1命令当りの処理時間 T は近似的に次式で与えられる．

$$T=T_0+n\cdot p\cdot T_B$$

n　：1命令当りメモリ参照回数（1.2 程度）．

T_0：すべて情報がバッファにあるときの処理時間．

p　：情報がバッファにない確率．

T_B：ブロック転送時間．

したがって，$n\cdot p\cdot T_B \ll T_0$ であれば見かけ上バッファ・メモリのスピードを持った主メモリがあるのと同じわけであり，この方式の要点は p をいかに小さくするかという点にかかってくる．

このような考え自身は古くから存在したが，IC メモリというハードウェアの裏づけができて，初めて実用化されたといえよう．

3.2 演算実行順序の制御

現在までのところは，ほとんどすべてのプログラム言語が（したがって，それを変換した機械語もまた）シーケンシャルな実行を前提としていることはいうまでもない．しかし 2. 章で述べたごとく，演算の高速化のためにまず考えられるのは，独立なプログラム・ステップの並列実行であり，場合によっては実行順序がプログラム上のシーケンスと逆になることもありうる．この方式をとるためにはまずプログラム・ステップ間の独立性を検出しなければならない．この手法を二つの大型機について見てみよう．

（1） CDC 6600 の並列演算器[11)]

6600 の CPU は 10 個の演算ユニットを持っている．これらはアドレス加算器 2，浮動小数点加算器 1，固定小数加算器 1，シフト演算器 1，論理演算器 1，乗算器 2，除算器 1，分岐ユニット1であり，オペランドの読出し，格納はすべて 24 個のレジスタを対象としている．これら演算器とレジスタ間を結びつけているのがスコアボードと呼ばれる制御ユニットである．命令が解読されると，オペランド・レジスタおよび演算器が使用可能であれば，ただちに演算器に送られて実行される．このどちらかが使用中であれば命令はスコア・ボードの待ち行列に入れられ，使用可能となるのを待つと同時に次の命令の解読へと制御が進む．この方式で問題となるのは，演算器の数とその処

理時間が命令の使用頻度とバランスがとれているか，特定のユニットのみに負荷が集中しないかという点である．ちなみに上位機種の 7600 ではアドレス加算器．乗算器は各1となり，代わりにデータ中の1の数を数える特殊なユニットが追加されている．

（2） 360/91 の演算器[5]

システム/360の命令系は固定小数点演算と浮動小数点演算をはっきり分離し演算レジスタまで独立に持っている．このため 6600 と同様モデル 91 でも固定小数点演算器と浮動小数点演算器を独立に設け，並列処理を行なっていることはいうまでもないが，さらに特徴的なのは浮動小数点演算器内でも並列処理を行なっている点である．すなわち，加減算器，乗除算器，転送（ロード，ストア）が独立に実行可能な構成をとり，かつそれぞれの演算器に2～3組の演算レジスタを持って命令をスタックすることができる．これら演算器間のデータ転送を供通バスで結び，また演算器間のオペランドの依存性を記憶する．一例として次のプログラムがあったとしよう．

LD	F_0, A_i	Load
DD	F_0, B_i	Divide
STD	F_0, C_i	Store
LD	F_0, D_i	Load

ストア命令のオペランドは浮動小数点レジスタ F_0 から読み出すのではなく，乗除算器の出力を使うという具合に制御を変えてしまうのである．これによりレジスタ F_0 はあきになるので，二つ目のLD命令は先行するDD命令よりも先に実行してしまうことが可能になる．

以上二つの並列演算処理の方法について述べたが，このような手法の最大の問題点は演算器におけるプログラム割込みとマシン・エラーの発生である．すなわち，現在多くの計算機のプログラム割込み（むしろ割り出しといった方がよいかもしれない）の仕様では割込みを起こした命令以後の命令は実行してはならないことになっており，前述のごとく実行順序を狂わせているとこの仕様を守ることができない．またマシン・エラーが発生した場合，再実行により間欠的な誤動作の回復をしようとしても，後続の命令の実行がすんでしまっていると，一般には再実行不可能となってしまう．並列処理の効果を十分にいかすためには，今後この問題の解決が必要である．

4. 演算そのものの高速化

演算方式が数値を表現する内部コードによって大きく変わることはいうまでもない．過去の事務用大型機には 10 進コードを使用するものも若干見られた．また演算の高速化，チェックの簡易化などをねらいとした Residue コードなどの提案もあるがまだ一般化するにいたっていない．少なくとも超大型機に関する限りはほとんどが2進コードを採用しており，ここ当分この大勢は動きそうにないので以下の話も2進コードに限定し，代表的な演算の高速化の手法について述べてみよう．

4.1 加算器

2進加減算の時間を決めるのはけた上げ伝搬時間であることはいうまでもない．

けた上げ伝搬に要する論理ゲートのレベル段数を減らすため，n ビットずつのグループ単位にけた上げ発生関数とけた上げ伝搬関数を作り，これをピラミッド形に組み上げていく方法が古くから行なわれている[8]．この n を幾つに取れるかは回路の fan-in, fan-out に依存するが，高速の回路になればなるほどこれらが制限されるのが普通であり，n は普通4程度にとられる．したがって 56～64 ビットの加算を行なうには，けた上げのピラミッド部分とビットごとの最終けた上げ発生回路を合わせて論理素子のレベルで6レベルは必要となる．加算器としてはさらに和の発生ゲートなどを含むわけで，これ以上のスピード・アップは論理ゲート当りの遅延時間を減らすより方法がない．一例として大形プロジェクトの CPU の加算器のゲート段数と遅延時間の内訳を上げると**表1**のごとくなり，回路の遅延のほか，実装規模を小さくして配線を短くすることがいかに大切であることを示している．なお表1の値はレジスタの遅れは含んでいない．

表1 大型プロジェクト CPU 加算器の遅れ時間

項目	数量	単位遅れ	遅れ (ns)
ゲート遅れ	9 レベル	1.5 ns/レベル	13.5
負荷による遅れ	8	0.3 ns/ゲート	2.4
ラインの遅れ	72"	0.2 ns/inch	14.4
計			30.3

また通常の論理回路によらない特殊なけた上げ伝搬回路が幾つか提案されたことがあるが，実装上の問題や誤動作の検出ができないこともあって最近ではほとんど使われていない．

4.2 乗算

乗算は部分積をシフトしながら乗数が1のところだけ被乗数を加算していくのは周知のとおりであるが，この加算回数を減らすことおよび1回の加算時間を減らすことが乗算のスピード・アップのかぎとなる．

（1）　乗数のデコード方式

加算回数を減らす方法として，乗数の中で0または1のストリングがある場合これをスキップできることに着目する．たとえば乗数が 01110 と続く場合，2+4+8 の3回の加算の代わりに 16−2 の2回の加減算ですますことができる．この方式によるスピード・アップ率はデータに依存するが乱数を仮定しても3倍程度にしかならず，これ単独で用いられることはほとんどないが，ほかの方式と組み合わせて使用されている．

（2）　360/91 の方式[13]

1回当りの加算時間を減らすため，乗算の繰り返し加算中にはけた上げを加え込む必要はなく，いわゆる半加算器 (Carry Save Adder――CSA) を使ってスピード・アップしたものである．図4が全体のブロック図で1回のループで12ビットの乗数を処理している．すなわち前述の乗数デコード方式により，12ビットの乗数をデコードしての6個の乗数に変換しこれにより被乗数をゲートしている (M_1〜M_6)．これを4個の半加算器 (CSA 1〜CSA 4) を使って2出力 (C_1, S_1) とし，これをさらに2個の半加算器 (CSA 5 と CSA 6) を使って前回のループの出力 C_0 と S_0 とともに加算している．Model 91 の場合，乗数は 56 ビットであり，この乗算を5回繰り返した後，C_0 と S_0 を全加算器で加えて最終的な積を得ているが，1回のループを 30 ns で行ない，全体の乗算時間を 180 ns という超高速で行なっている．

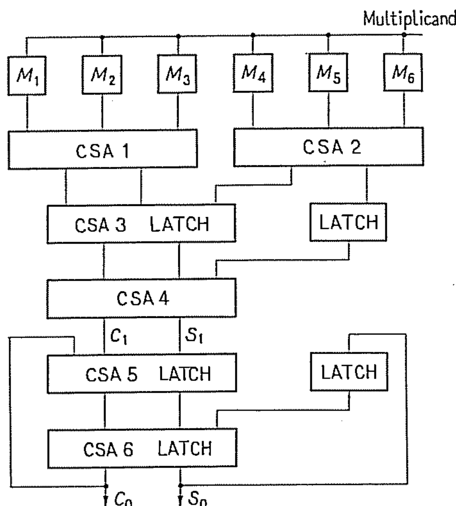

Fig. 4　Multiplier of IBM S 360/91.

4.3　除　算

除算の基本はよく知られている引き放し法であるが，これによると商のビット数と同じだけの加減算が必要となり，これをスピード・アップするための種々の方式が提案されている．そのいずれも演算に先立って除数を正規化しているが，これは一度に複数ビットの商を立てるために除数と部分剰余との大小関係を決定する必要があるからである．

（1）　正規化方式

部分剰余(R)の先頭に0または1が連続する場合，そのビット数だけRを左シフトする（つまり正規化する）とともに商として同じビット数だけの0または1を立てることができる．この手法によるスピード・アップの割合は，全くデータに依存し，正規化が平均的に何ビットずつ可能かによるわけであるが，一般的には2〜3ビット程度であり[8]，むしろ毎サイクルの商のビット数が一定しないための制御の複雑さなどのマイナス面もあって実際にはあまり使用されていない．

（2）　大形プロジェクト CPU の方式

これは Robertson の方法[12]を基本としたもので，除算の繰り返しを前述の Carry Save Adder を用いて高速化をはかったものである．CSA を用いるためけた上げ伝搬が行なわれず，部分剰余の上位のビットは常に1ビットの誤差を含んでおり，これによって立てる商も1ビットの誤差を含むが，たとえば商を大きく立て過ぎたときは次回の繰り返しのとき，負の商を立てることにより補正していくものである．したがって除算の途中では正の商と負の商の二つが立っており，これを加えて最終的な商を得ている．大形プロジェクトの CPU ではこの方式により1サイクル (50 ns) に2ビットの商を立てている．

（3）　360/91 の方式[13]

通常の方式とは異なって高速乗算器をフルにいかした特殊な方法である．

$$\frac{N}{D}=\frac{N\times R\times R_1\times R_2\times R_3\times R_4}{D\times R\times R_1\times R_2\times R_3\times R_4}$$

という関係を利用し，R_i を $DRR_1R_2R_3R_4=1$ となるように選べば $NRR_1R_2R_3R_4$ が商となる．すなわち R_{i+1} は，$DR\cdots R_i$ の逆数に近い12ビットを選び，前節で述べた高速乗算器で上記の乗算を行なっていく．360/91 の場合，56 ビットの商を立てるのに5回の乗算ですみ，演算時間として 720 ns で終了して

いる．この方式はきわめて高速であるが，最大の難点は商の最後の1ビットがあらかじめ定められたアルゴリズムに一致せず，下位機種と異なることが起こりうることである．

5. むすび

超大形機の演算制御の要点について概観してきた．計算機の演算速度が限界に近ずきつつあるといわれながらも，いままでは地道な技術改良によって着実に発展したきたが，今後もこの調子で速度が向上するためには，解決しなければならない多くの問題があるが，そのうち大きなもの2点について述べてむすびとしたい．

その一つはアーキテクチャの問題である．現在の多くのシステムは主として汎用の中・小形機向けのアーキテクチャとなっているが，たとえば，プロセッサ・アレイやパイプライン・システムを有効にいかすためには，それ向きの言語や問題解法のアルゴリズムから再検討する必要があろう．また小さくは命令語の仕様を取ってみても初期のストアド・プログラム時代の名残りがかなり残っている．一例として現在のソフトウェアでは，特にリエントラント・プログラムにでもなると，命令語を書き替えることはほとんどなくなっているにもかかわらず，ハードウェアではあらゆるストアタイプの命令について後続の命令を書き替えるかどうかのチェックを行なっており，これが物量的にもスピード的にもオーバーヘッドとなっている．このほか並列実行またはパイプ・ライン制御を困難にする割込みや条件付き分岐命令などの再検討が必要となろう．

その2はハードウェア的な問題であるが，LSI(Large Scale Integration) と論理構造の問題がある．前述 (4.1 節) のごとく，たとえば加算器をとってみても演算時間は半導体の中の遅延時間と配線上の伝送時間が同程度になっており，この両者を同時に短縮するLSI 技術が要求される．しかし集積規模を増せば増すほど，小量多品種の LSI が必要となってコスト高となるので，いかにして小品種多量の LSI で CPU を構成するかという論理構成法の研究が必要となろう．

参考文献

1) G. M. Amdahl: Validity of the single processor approach to achieving large scale computing capabilities. Proc. SJCC 1967, pp. 483〜485.
2) G. H. Barnes: The ILLIAC Ⅳ Computer. IEEE Trans. EC, Vol. C-17, No. 8 (1968) pp. 746〜757.
3) D. J. Kuck: ILLIAC Ⅳ Software and Application Programming. 同上 pp. 758〜770.
4) L. W. Cotten: Maximum pipeline systems. Proc. SJCC 1969, pp. 581〜586.
5) R. M. Tomasulo: An Efficient Algorithm for Exploiting Multiple Arithmetic units. IBMJ. Vol. 11, No. 8 (1967) pp. 25〜33.
6) L. J. Boland, et al.: The IBM System/360 Model 91 Storage System. 同上 pp. 54〜68.
7) G. S. Tjaden, et al.: Detection and Parallel Execution of Independent Instructions. TIEEE Vol. C-19, No. 10 (1970) pp. 889〜895.
8) Q. L. MacSorley: High Speed Arithmetic in Binary Computers, PIRE, Vol. 49, No. 1 (1961) pp. 67〜91.
9) D. W. Anderson, et al.: The IBM System/360 Model 91: Machine Philosophy and Instruction Handling. IBMJ, Vol. 11, No. 8 (1967) pp. 8〜24.
10) C. J. Conti: Structural aspects of the System/360 Model 85. IBM Systems J., Vol. 7, No. 1 (1968) pp. 2〜14.
11) J. E. Thornton: Parallel Operation in the Control Data 6600, Proc. FJCC (1964), pp. 33〜40.
12) J. E. Robertson: A New class of digital division methods, T. IRE, Vol. EC-7, No. 3 (1958)
13) S. F. Anderson, et al.: The IBM System/360 Model 91: Floating-Point Execution Unit. IBMJ, Vol. 11, No. 8 (1967), pp. 34〜53.

(昭和46年5月13日受付)