論　文

構造主義にもとづく計算機システムの構成法について*

林　　達　也**

Abstract

Previously we have developed a design and implementation language, DEAPLAN, which enables us to describe any hierarchical and logical structure of operating systems. In this paper we present a new method for constructing computer systems based on the principle of structuralism.

In our method the implemented version of the system is identical with the original form described in DEAPLAN. Therefore, extremely speaking, there are no needs for compiler, preprocessor and linkage editor except the such like loader or creater. In other words, it may be said that a kind of high level language machine is considered in the more throughgoing way.

Our hardware becomes quite different from the conventional one and only consists of large number of quantum processing units. Since we apparently have neither memory nor CPU, the contemporary concepts such as virtual space, reenterability and multiplexing of CPU are also disappeared in our system.

1. まえがき

ここ10年位の間に，N. チョムスキー(言語学)，R. ヤコブソン (音韻学)，C. レヴィ=ストロース (文化人類学)，ブルバキ (現代数学)，J. ピアジェ (心理学) などの人々が構造主義的立場をとり，それぞれの分野においてかなりの成果を収めてきた．構造主義について明確な定義を下すことは難かしいが，一口にいえばその基本的立場は構造が機能を決定するということで，例えばプログラミングのような知的活動を行う場合で言えば，それを行う人間（プログラマ）の精神構造的わく組がプログラミングのやり方（方法）に一定の方向ないし制約を与えるという立場からその構造的わく組を考察することである．したがって，E. W. Dijkstra の structured programming[1] も，充分ではないかも知れないがこのような流れの一つとして捉えることができよう．我々も前にそのような立場をとって，システム設計言語 DEAPLAN を考案した[2]．この言語は OS の特定構造を前提とするものではなく，任意の OS 全体の階層的な論理構造を直接的に記述することができるものであった．今度は，我々はこの論文で一つの新しい計算機システムの構成法について考察する．これは DEAPLAN で記述されたシステムの原型に省略や変形を一切加えることなく，そっくりそのままの形で実現 (implement) しようとするものである．いい換えれば，原型と実現とが完全に一致する方法である．現在の OS (MULTICS[3], OS/VS II[4] など) は，本来何らかの階層構造を有していると考えられるが，それが実現される場合には，通常上層部のモジュール***(ジョブ，ジョブステップ，タスク，ロードモジュールなど) は制御データの形で間接的にしか存在せず，直接的には実現されない．また下層部のモジュール(手続き，ステートメント，演算子など)も，コンパイラやプリプロセッサによって変換されない限り実現されないといってよい．

我々の方法は，DEAPLAN 表現を直接実現しようとすることから，いわゆる高水準言語計算機 (high level language machine)[5)~18)] とも関連が深い．しか

* A Design Methodology of Computer Systems based on the Principle of Structuralism by Tatsuya HAYASHI (Computer Science Lab., Fujitsu Laboratories Ltd.)

** (株)富士通研究所電子研究部

*** DEAPLAN ではデータに作用を及ぼす実体を全てモジュールと呼んでいる

し，それらがいずれも，システムの下層部のみを対象にしていること，また何らかの形で原表現を中間形式に変換している点で，我々の方がより徹底しているといえよう．上述したような事情は，高水準言語計算機も含めて一般に現在のハードウェア構成が，人間にとって理解しやすい形で（実現された）システムに構造を与えるには，適していないことを意味する．

我々のシステムで用いられるハードウェア構成は，したがって，従来のものとは異なる必要があり，実際そのようになっている．我々の場合，ハードウェアは(カード読取機，ラインプリンタなどを除き)量子処理装置 (QPU) と呼ばれる構成要素だけの複合体から成り，見かけ上主記憶，2次記憶および CPU は存在しない．したがって，仮想記憶 (VS), CPU の多重化，再入可能 (reenterable) などといった，今日の OS における代表的な概念は我々のシステムで消滅する．またシステムの原型と実現とが一致することから，我々の場合極端な言い方をすれば，コンパイラやリンケージエディタは不要であり，ローダがありさえすればよい．

以下では，まず **2.** でハードウェア構成について述べる．次いで **3.** でシステムの構成法を例題にもとづいて説明し，同時に非同期処理やシステムの安全性がどのように実現されるかを述べ，**4.** で開発の見通しについて触れる．最後に **5.** で本方式の意義を考察する．

2. ハードウェア構成

我々のシステムで用いられるハードウェアは **Fig. 1** の通りである*．図から明らかなように，QPU 群は従来の CPU と主記憶，2次記憶を統合再編成したものと見なすこともできる．各 QPU はマイクロプロセッサを微小化したようなもので，**Fig. 2** のように算術論理ユニット (ALU)，ローカルメモリ (LM)，書き込み可能制御記憶 (WCS) および通信ユニット (CU) から成る．QPU の具体的な機能は WCS に格納されるプログラム（量子プログラムと呼ぶ）によって規定され，他の QPU とは CU によって通信する．

QPU をプロセッサの一種と見なせば，我々のシステムは記憶装置を持たない多重プロセッサシステムということになる．一方，QPU を記憶装置のそれぞれの小部分が独立して活性化したものと考えれば，我々のシステムはプロセッサを持たないことになる．

* チャネルや入出力制御装置などに関しても，それらが論理機能を果たす限り，QPU (群) で実現することができるが，簡単の為ここでは図の通りとしておく．

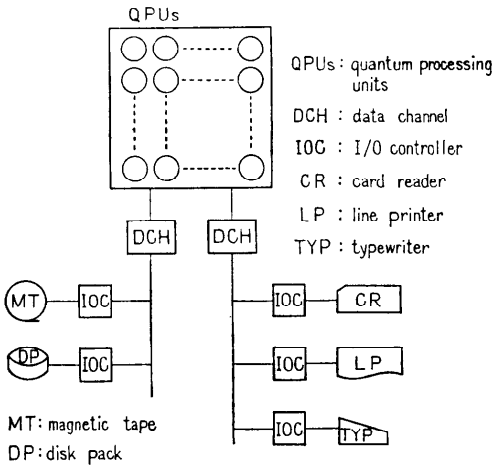

Fig. 1 hardware system

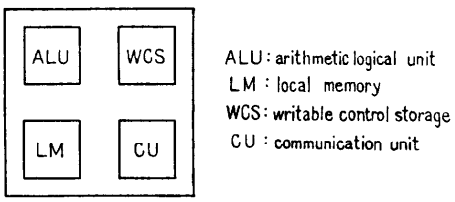

Fig. 2 QPU organization

3. システムの構成法

3.1 実 現 方 法

システムは実現される場合，その原型 (DEAPLAN 表現)が，構造を保持しながら歪みを受けることなく，前章で述べた多数の QPU の上に分散して展開される．たとえば，**Fig. 3**(次頁参照)および **Fig. 4**(次頁参照) の例を考えよう．これらはそれぞれ，プログラム PROG およびその主外部手続き MAINX の DEAPLAN 表現である．記述内容はおのずから明らかであろう．これらの原型は，我々のシステムでは，**Fig. 5**(次頁参照) に示すように実現される．図において，円は QPU を表わし右肩の数字は QPU 番号を示す．また円内の文字は QPU に割付けられたモジュールあるいはデータの名称である．

モジュールやデータには，それらが QPU に割付けられた時に，固有の ID コードが与えられる．ID コードの用途については **3.3** で述べる．

外部手続きおよびそれ以上のモジュールが割付けられた QPU の LM には，モジュールディレクトリ (MD) とデータディレクトリ (DD) が作成される．

```
MODULE PROG PROGRAM;
   TSPAN CONTROLLED (USER_JOB);
   ENTRY PROGE;
   DATA (A (100), B) BIN (31);
   DATA C ORG 1 D BIN (31),
              1 E,
                2 F BIN (31),
                2 G CHAR (8);
   MODULE MAINX PROC MAIN;
      FNTRY ME; END MAIN;
   MODULE SUBX PROC;
      ENTRY XE; END SUBX;
   MODULE SUBY PROC;
      ENTRY YE; END SUBY;
   ACCESS MAIN (READ (A, C), WRITE B),
          SUBX (WRITE A, READ C),
          SUBY UPDATE C;
   CALL FROM MAIN TO SUBX,
        FROM SUBX TO SUBY;
END PROG;
```

Fig. 3 original form of PROG in DEAPLAN

```
MODULE MAINX PROC (PROG) MAIN;
   ENTRY ME;
   DATA EXT(A (100), B) BIN (31);
   DATA EXT C ORG 1 D BIN (31),
                  1 E,
                    2 F BIN (31),
                    2 G CHAR (8);
   DATA (S, I) BIN (31); TSPAN AUTOMATIC;
   MODULE SUB PROC;
      ENTRY E;
         ⋮
      END SUB;
   LOGIC
            B=C. D;
            ⋮
            DO S=S+A(I); I=I+1 WHILE I≤100;
            ⋮
            XE;
            ⋮
   ACCESS EXT MAIN (READ (A, C), WRITE B);
   CALL EXT FROM MAIN TO SUBX ENTRY XE;
END MAINX;
```

Fig. 4 original form of MAINX described in DEAPLAN

MD には成分ないし関係する外部モジュール名，属性，所在 (QPU 番号)，ID コード，呼出し関係などが記録され，DD には内部ないし関係する外部データ名，属性，所在 (QPU 番号)，ID コード，アクセス関係などが記録される．モジュール間の呼出し関係やモジュール～データ間のアクセス関係は，HYDRA[19] や J. B. Dennis ら[20]の capability の概念に相当する．これについても **3.3** で改めて触れよう．

次に QPU 間の実線は，ローダによって実現される時に，それらの QPU の間に静的にリンケージがなされたことを示す．すなわち，矢印の先にある QPU の番号が矢印の根の方にある QPU に格納されていることを表わす．ローダが利用できる情報を増やせば増やす程，そのようなリンケージを静的に行うことができる．たとえば，Fig. 4 の記述 (MAINX) を実現する場合，QPU PROG の MD や DD をローダが参照できれば，QPUXE と QPU SUBX 間や QPU =(B, C. D) と QPU B あるいは QPU D 間のリンケージを静的に行うことができよう．

一方，配列や構造体データの場合には，A や C, E のようにサブフィールドを有するデータが割付けられた QPU の LM に，フィールドディレクトリ (FD) が作成される．FD には，サブフィールド名，属性，所在 (QPU 番号)，ID コード，アクセス関係などが記録される．FD は DD の延長と考えることもできる．

3.2 動作形式

Fig. 5 の例にもとづいて，システムがどのように動作するかを説明しよう．まず QPU PROG (以下 QPU を省略する) が他の QPU によって起動されたとする．PROG は直ちに MAINX を起動し，MAINX からの終了通知を待つ．MAINX は MD および DD を調べ，データ S, I が AUTOMATIC であることを知る．そこで，空き QPU #300, #301 を確保して，その上に S, I をそれぞれ割付け，DD にそれらの所在や ID コードを記録する．次いで，第1の

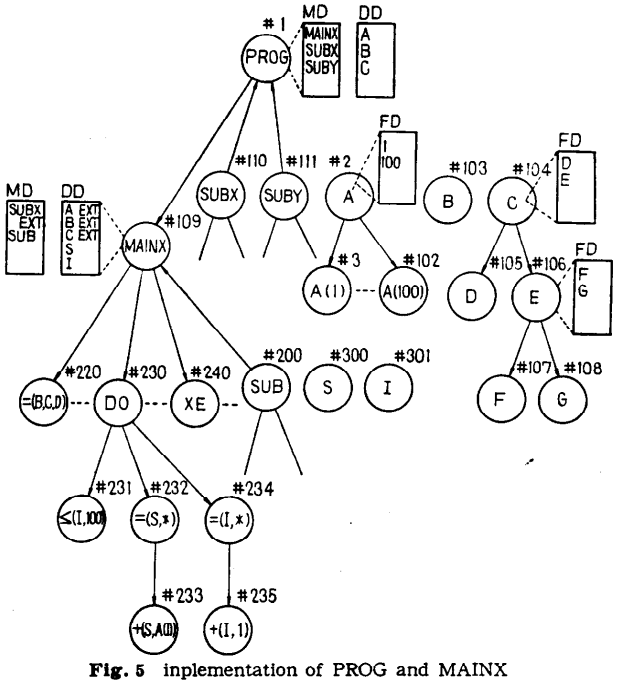

Fig. 5 inplementation of PROG and MAINX

コマンドに対応する=(B, C. D) を起動し，終了通知を待つ．終了通知があれば，MAINXは第2のコマンドに対応するQPUを起動し，それからの終了通知を待つ．以下これをくり返し，最後のコマンドに対応するQPUから終了通知を受けとると，MAINXはS, Iを解放し，DDの更新を行ってから，PROGに対して動作終了を通知する．これによって，PROGは全動作が終了したことを知り，呼出し元に対して終了通知を行う．

さて今度は，=(B, C. D) が起動された場合を考えよう．=はBやC. Dの所在が分らないので，親のMAINXにアクセス申請を行う．MAINXはDDを参照してアクセス属性に違反していないことを確めた後，(外部データなので) 親のPROGにアクセス申請を行う．PROGも同様にアクセス属性をチェックした後，BとC. Dの所在をMAINXに通知する (C. Dの場合はCと交信することによってその所在を知る)．MAINXはそれをDDに記録すると同時に=に通知する．これによって，=とB, C. Dの間に動的なリンケージがなされたわけである．=はそこで，Dに対して転送指令を出し，受けとった値を今度はBに転送する．

他のQPUの動作も同様にして理解されよう．なお，=(S, ＊) や =(I, ＊) において，＊は子の +(S, A(I)) や +(I, 1) から終了通知と共に返される値を示す．またXEはコマンドそのものであり，対応するQPUはSUBXである．したがって，本来はMAINXが直接SUBXを起動すべきであるが，ここでは分りやすくするために，図のようにXEなるQPUを設けてある．

3.3 安 全 性

我々のシステムでは，2つの手段によってその安全性の向上がはかられている．1つはIDコードであり，他の1つはディレクトリである．

モジュールやデータは，レベルの高低にかかわりなく，それらが生成されQPUに割付けられる際に，システムの全期間を通じて固有なIDコードが原則としてQPU単位に与えられる．IDコードとしては，たとえば生成 (割付) 時刻をとればよいだろう．与えられたIDコードは当該QPUに保持される(これをロックIDと呼ぶ)．一方，このQPUと交信しようとする他のQPUは，その妥当性をチェックされた後，静的あるいは動的なリンケージを通して相手QPUの番号とIDコード (キイIDと呼ぶ) を受けとる．そこで **Fig. 6** に示すように，送信QPUは交信の際に受信QPUの番号だけでなく，キイIDも呈示する．受信QPUはキイIDをロックIDと照合することによって，その交信に誤りがないかどうかを判定することができる．

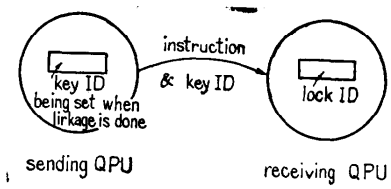

Fig. 6 inter-QPU communication

IDはMULTICSなどのセグメントに相当し，更にそれを徹底させたものと考えられる．Fig. 5から分るように，QPU単位に異なるIDコードが与えられるので，たとえばCのような構造体データは複数個のID (セグメント) に分けられるからである．こうしてシステムは，QPUに割付けられるような極めて小さな部分に分割されて保護されることになる．

一方，我々のMDやDD, FDは，前に述べたように，MULTICSのセグメントテーブルやHYDRAのLNS (local name space), Dennisらのcapability listに相当する部分を含んでいる．しかしそれらが，プロセス単位とか外部手続き単位にしか設けられないのに対して，MDやDDは階層構造のほとんどあらゆるレベルに設けられるので，呼出し属性やアクセス属性，いわゆるインタフェイス，のチェックを厳格にきめ細かく行うことができる．

3.4 モジュール QPU の役割

一般にモジュールが割付けられたQPUは，すでに述べたように，コマンドに対応する他のQPUを起動するか基本的な演算処理を行う．特に外部手続き以上のレベルのモジュールが割付けられたQPUの場合は，これも**3.3**で述べたように，MDやDDに記録されている呼出し属性やアクセス属性を利用して，動的なリンケージを管理するわけである．

さてそれ以外に，モジュールQPUは一般にAUTOMATICあるいはCONTROLLED属性を持つ成分モジュールないし内部データの生成，消去を行う．その場合，モジュールQPUが行う処理内容は，そのレベルの高低に何らかかわりなく，全く同一なのである．すなわち，手続きQPUが行ういわゆるプロローグ／エピローグ処理も，よりレベルの高いモジュールQPUが行う処理も，違いを生じない．

3.5 データ QPU の役割

単純データが割付けられた QPU は，すでに述べたように，モジュール QPU との交信によって定められた型のデータ値を格納したり送出したりする．

構造を持ったデータは，例えば Fig. 5 の C のように，複数個の QPU に樹枝上に割付けられる．その中には，QPU C や QPU E のように，FD を持つ QPU (これをインデクス QPU と呼ぶ) が存在する．インデクス QPU は，MULTICS のファイルディレクトリの概念をより徹底させて，全ての構造を持ったデータに対して適用したものとも考えられる．それ故，アクセス属性も C.D などのように容易に部分データ単位できめ細かく指定できるわけである．

一方，データが処理能力を持つ QPU に割付けられることから，我々の場合同期処理[23]は **Fig. 7** のようにデータ QPU 自身で行うことができる．

特に排他制御は **Fig. 8** に示すように，semaphore を用いずにアクセスされるデータ自身で管理することもできる．モジュール QPU は排他的にアクセスしたいデータ QPU に対して，それがファイル全体であろうと単なるサブフィールドあるいは単純データであろうと関係なく，ロック命令を出して申請し，受付けられたらそのデータを使用し，その後アンロック命令によって排他的使用を解除すればよい．

4. 開発の見通し

ここに述べた QPU の機能は，今日のマイクロコンピュータにほぼ相当する．ただし，モジュールディレクトリ，データディレクトリあるいはフィールドディレクトリを含む場合は，1～4 kB 程度のローカルメモリ (LM) が必要であろう．次に計算機システムを構成する QPU の数は，現在の OS を考えると大体 10^4～10^6 のオーダとなるので，QPU 間の通信路の設定が問題となるが，すでに確立されている交換機の原理を利用すれば技術的には解決できるのではないかと思われる．

一方，LSI 技術の進歩は目ざましく，年に 2 倍の割合で集積度が向上しつつあり，すでに電卓の 1 チップ化が実現されている．そして，マイクロコンピュータは数年後に 1 チップになり，ミニコンもそれに引き続いて間もなく 1 チップ化されるだろうという[24]．

したがって，その時の QPU 1 個当りの容積を，交換スイッチも含めて仮に数センチ立方と大ざっぱに仮定した場合には，10^6 個の QPU から成る計算機システムは，周辺装置を除いた主要部で，およそ数メートル立方の大きさに納まることになり，それ程非現実的ではないと思われる．

また，このように計算機システムを多数の単一種類の要素 (QPU) から構成することは，LSI 技術を量産効果の面でより一層刺激することが考えられる．

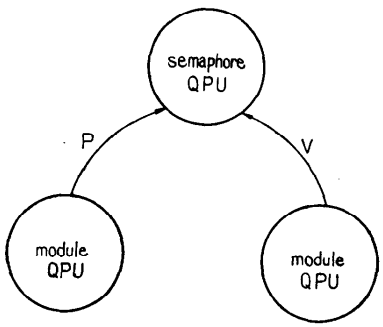

Fig. 7 synchronization

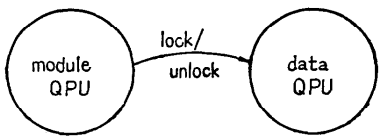

Fig. 8 exclusive control

5. 本方式の意義

この論文では，新しい計算機システムの構成法を，DEAPLAN にもとづいて説明したが，この方法は PL/I などの他の計算機言語に対しても同じ様に適用できる．また，DEAPLAN はここで述べた方法でしか実現できないわけではなく，従来の計算機上でも他の計算機言語と同様にして実現することができる．

さて，ここでは本方式の意義についていくつか考えてみよう．

(1) 並列処理の徹底

計算機言語は DEAPLAN も含めて，通常タスク (プロセス) レベルの並列処理しか明示できないが，ここで述べた構成法だと演算子レベルや文レベルの並列化が可能でより徹底した並列処理の潜在能力を有している．

(2) 開発限界量の引き上げ

今日の OS もすでに相当な大きさを持っているが，それでも大量の人手をかけてどうにか作成することができている．したがって，我々の方法で現在の OS を仮に実現したとしても，効率が悪くなるばかりで利点

がありそうにも思えない．しかしながら，計算機を人間のよきアシスタントたらしめようとする立場から見れば，現在の計算機システムの知的水準を更に引き上げることが必要であろう．

一方，システムは MMI（マンマシンインタフェイス）を高度にすればする程，量的にも極めて大規模なものとなるであろう．そこで，プログラミングの生産性や信頼性を高めるために現在提唱されている，structured programming やシステム作成言語のハイレベル化などの手法の他に，原型と実現とを一致させる我々の方法を更に加えれば，このような問題により一層有効に対処できるのではないかと思われる．我々の方法では，実現されたシステムはいわば立体的な構造を持ち，その内部や動作時の振舞が透明になるからである．

（3） 信頼度の向上

我々のシステムでは，全体が微細な部分に分けられて多数の QPU 上に分散して格納される．そして，ディレクトリや ID コードを用いて，インタフェイスが QPU 単位で管理される．また，原型と実現との間に何らの変換も施されていないので，全体の動作状況を原型の階層的な論理構造の上で直接とらえることができる．これらはシステムの信頼度を高める働きをするであろう．

（4） 知識の手続き化

最近 artificial intelligence の分野で，知識の手続き化（procedural embedding of knowledge）なる概念が注目されている[21),22)]．我々のシステムでは，これを容易に効果的に実現できるであろう．

PL/I や ALGOL 68 には手続き変数に相当する概念が存在する．しかし，いずれも現在のシステムの影響を受けており，プログラムの作成時に定められたいくつかの手続きに対して，その中のどれかを実行時に動的に固定するもので，手続きそのものを動的に作成できるわけではない．我々の場合は，コンパイルないしリンケージエディットは不要なので，手続きあるいはモジュール一般を動的に定義することは，従来のシステムよりも容易であろう．この点は，OS に高度な学習機能を持たせ OS が自からを成長変化させようという場合に有利であろう．

（5） policy と mechanism の分離

我々のシステムは，HYDRA などで提唱されている policy と mechanism の分離という方向にも沿っている．すなわち，mechanism に関しては，QPU と量子プログラムが基本的な非同期処理やシステムの安全性などを管理することでその役割を果している．また policy に関しては，そのような QPU 複合体によって，任意の論理構造を OS に与えることができるからである．

（6） 関連分野への寄与

ここに述べたシステムが，QPU という構成要素だけからなる複合体の上に形成されることで，生体のニューロンネットワークと何らかの関連を持たせられるのではないかと期待される．もしそうならば，神経生理学や生体工学，パターン認識などの関連分野に対してもいくらかの情報を提供できることになろう．またこの新しいシステムの情報処理能力の限界については，オートマトン理論の分野での研究が望まれる．

6. あとがき

この論文では，我々が前に提案したシステム設計言語 DEAPLAN で記述されたシステムを，何ら変形することなく直接的に実現（implement）する方法について考察した．その結果得られた計算機システムの新しい構成法は，DEAPLAN に限らず他の計算機言語に対しても同様に適用でき，かつ並列処理を徹底して行える潜在能力も有している．

この方式で必要とされるハードウェアは，多数の QPU（量子処理装置）からなる．この QPU 複合体によって，実現されたシステムに対しても，原型（DEAPLAN 表現）と全く同一の見やすい構造を与えることができる．これは従来の計算機では不可能なことである．原型と実現とが一致することによって，我々の方法は OS の巨大化の問題に対して根本的な解決をもたらし，OS に現在よりもなお一層高度なマンマシンインタフェイスを与える道を開くものと思われる．

また，ここで述べた構成法は，神経生理学や生体工学，パターン認識などの分野に対しても少なからぬ関連を持つであろう．この構成法がもたらす理論的，技術的諸問題を明らかにし，それらを解明することが今後の課題である．

謝辞：富士通に居られた間，色々な面で指導して戴いた井上謙蔵氏（現東工大教授）に厚くお礼を申し上げる．

参考文献

1) Dijkstra, E. W.: “Notes on Structured Programming”, Technological U. Eindohoven, The Netherlands, (Aug. 1969).

2) 林 達也: "システム設計言語 DEAPLAN について", 情報処理, Vol. 14, No. 9, pp. 652～660 (1973).
3) Organick, E. I.: "The Multics System: An Examination of Its Structure". MIT Press, (1972).
4) Scherr, A. L. et al.: "Functional Structure of IBM Virtual Storage Operating Systems", IBM Syst. J. Vol. 12, No. 4, pp. 368～411 (1973).
5) Rice, R. & Smith, W.R.: "SYMBOL-A Major Departure from Classic Software Dominated Computing System", Proc. SJCC Vol. 38, pp. 575～587 (1971).
6) Smith, W. R. et al.: "SYMBOL-A Large Experimental System Exploring Major Hardware Replacement of Software", Proc. SJCC Vol. 38, pp. 601～616 (1971).
7) Bashkow, T. R., Sasson, A. & Kronfeld, A.: "System Design of a FORTRAN Machine", IEEE Trans. on Electronic Computers, Vol. EC-16, No. 4. pp. 485～499 (1967).
8) Anderson, J. P.: "A Computer for Direct Execution of Algorithmic Languages", FJCC, pp. 184～193 (1961).
9) Mullery, A. P. et al.: "ADAM-A Problem-Oriented Symbol Processor" SJCC, pp. 367～380 (1963).
10) Melbourne, A. J., et al.: "A Small Computer for the Direct Processing of FORTRAN Statements", Computer Journal, Vol. 8, No. 1, pp. 24～27 (1965).
11) Weber, H.: "A Microprogramed Implementation of EULER on the IBM System/360 Model 30", CACM, Vol. 10, No. 9, pp. 549～558 (1967).
12) Sugimoto, M.: "PL/I Reducer and Direct Processor", 24th National Conf. of ACM, pp. 519～538 (1969).
13) Thurber, K. J., et al.: "System Design for a Cellular APL Computer", IEEE Trans. on Computers, Vol. C-19, No. 4, pp. 291～300 (1970).
14) Zaks, R., et al.: "A Firmware APL Time-Sharing System", SJCC, pp. 179～190 (1971).
15) Hassit, A., et al.: "Implementation of a High Level Language Machine", 4th Annual Workshop on Microprogramming (1971).
16) Shapilo, M. O.: "A SNOBOL Machine: A High-Level Language Processor in a Conventional Hardware Framework".
17) Wilner, W.T.: "Design of Burroughs B 1700", FJCC, pp. 489～497 (1972).
18) Wilner, W. T.: "Burroughs B 1700 Memory Utilization", FJCC, pp. 579～589 (1972).
19) Wulf, W., et al. "HYDRA: The Kernel of a Multiprocessor Operating System", CACM. Vol. 17, No. 6, pp. 337～345 (1974).
20) Dennis, J. B. & Van Horn, E. C.: "Programming Semantics for multiprogrammed Computations", CACM, Vol. 9, No. 3, pp. 143～155 (1966).
21) Winograd, T.: "Procedures as a representation for Data in a Computer Program for Understanding Natural Language", Project MAC TR-84 MIT (1971).
22) Hewitt, C., Bishop, P. & Steiger, R.: "A Universal Modular ACTOR Formalism for Artificial Intelligence", Proc. 3rd IJCAI, pp. 235～245 (1974).
23) Dijkstra, E. W.: "Co-Operating Sequential Processes", Programming Languages, pp. 43～112, Academic Press, New York (1968).
24) 須藤常太: "LSI の技術と動向", 情報処理, Vol. 15, No. 12, pp. 976～981 (1974).

(昭和49年11月29日受付)
(昭和51年7月21日再受付)