計算機設計言語*

萩原 宏** 黒住祥祐***

Abstract

In this paper, we deal with computer-aided design language of computers. The steps of computer design are 1) system design, 2) logical design, 3) packaging design, and 4) circuit design. This language covers system design and logical design. Because system design and logical design are very difficult steps in computer design, we divide this language into three levels as follows: 1) transfer level, 2) block level, and 3) module level.

First, the requirments for the design language of computers are discussed.

Secondly, the structures of levels of this language are described.

1. まえがき

計算機の設計は大別すると，つぎの4分野に分けられる．(1) システム設計，(2) 論理設計，(3) 実装設計，(4) 回路設計．これらの分野の設計を計算機の援助によって行なうこと (CAD) の重要性は充分に認識されている．とくに，システム設計以外の分野では，いろいろな方法で実用化されている．しかし，個個のステップでのシミュレーションやチェックを目標としているため，プログラムやデータの作成は，ほとんど人手によって行なわれている．

システム設計の分野での機械化も早くから注目されており，まず，計算機の論理動作の記述言語として，1962年に発表された Iverson 言語[1] がある．これは，最初は記述用言語として発表され，その後，設計言語[13]として手が加えられている．シミュレーション言語として最初のものは，Gorman ほかの LDT[2] である．これはレジスタ間の転送動作をクロックごとに記述する方法であり，register transfer language と呼ばれる言語のさきがけである．この思想にそって，1964年に Proctor[3], Schlaeppi[4], Schorr[5] などの発表がある．さらに，1965年に McClure[6], Zucker[7], Griffin[8], Chu[9] などの論文がある．これらはいずれも register transfer language の流れを汲むもので，設計用よりも，むしろシミュレーション用に計算機を記述するための言語であった．1966年の Brever[10] の解説には，287 編の文献が分類されている．その後のおもな設計言語として Dully[11], Friedman[12], Cook[14] などの言語がある．設計言語の変換方法について Gerace[12] の発表がある．

わが国では 1963 年から高島ほかの論理構成のシミュレーションに関する研究[15)~18)] がある．設計言語として最初のものは 1967 年の岡田，元岡の論理設計言語である．これは設計用言語として5レベルの言語体系を開発し，各レベルで人間-機械系を可能にしようとした点がすぐれている．5レベルのうち，もっとも問題向きレベルのものは register transfer language と同等であり，もっとも機械向きレベルのものは論理図に相当する．

このように計算機システムの記述言語は，すでに多数発表されている．しかし，多くはシミュレーション言語であり，設計言語として最近の大型計算機の機能を充分な効率をもって記述できるかという点については，まだ改良の余地が残されていると思われる．そこで，筆者らは3段階の言語 T, B, M-言語[21)~25)] を導入した．これらの言語で計算機を設計・記述し，言語間の変換プログラムにより変換することにより，能率のよい設計が可能となるであろう．

2. 計算機設計言語の機能

設計とは，一端では使用用途に最適なシステムの仕様（マクロな仕様）を決定し，他端ではそのマクロな仕様に適した部品の仕様（ミクロな仕様）を決定し，ミクロな仕様でマクロな仕様を構成する最適な方法を

* Design Language of Computer, by Hiroshi Hagiwara (Faculty of Engineering, Kyoto University) and Yoshisuke Kurozumi (Faculty of Science, Kyoto Sangyo University).

** 京都大学・工学部
*** 京都産業大学・理学部

考えることである．計算機の設計では方式および論理設計がマクロな設計であり，実装および回路設計がミクロな設計であろう．計算機の設計に限らず，これらマクロな設計とミクロな設計は，同時に並行して進められるべきもので，各段階における双方同時のシミュレーション（実験）により，互いに設計変更の影響を及ぼしながら，最適設計に近づけてゆくべきであろう．したがって，マクロ，ミクロいずれの設計においても，互いに他の記述も可能であることが望まれる．

このような考えから，計算機設計言語には，つぎの機能が取り入れられるべきであろう．

2.1 マクロ言語

(1) 入力言語

計算機による設計への入力言語として，設計のあらゆる可能性を含んでいなければならない．設計が進むにつれて細部の仕様が決まってくる場合が多いから，最初は大まかな指定ができ，後になって，任意に詳細を記述できることがのぞましい．

このためには，言語にレベルを用意し，人間-機械系によって，各レベルでの設計変更を可能にすればよい．

(2) システム・ブロックの記述

計算機には機器・装置・組織と各段階でまとまった構成がある．このような構成の記述および構成間の連絡方法などの記述ができること．これは，プログラミング言語で使われているブロック構造で解決できる．

(3) 並列処理・優先処理

各ステップでの並列動作はもちろんであるが，各構成単位ごとの並列処理・優先処理があらわれる．とくに，各ステップの並列処理については，従来のプログラミング言語の記述をそのまま使うと，はん雑になるから，改良する必要がある．

(4) ソフトのハード化

設計言語を必要とする理由の1つに，ソフトウェアのハードウェア化がある，

回路素子が標準化され安価になると，処理速度向上のため，従来ソフトウェアで処理していた部分をハードウェアで置き替える．たとえば，スーパバイザ，入出力処理プログラム，言語処理プログラムの一部にこの傾向がみられる．このとき，ソフトウェアで処理する方法を，そのままハードウェアで実現できるとは限らないが，その記述は充分に可能でなければならない．つまり，一般のプログラミング言語で記述できる手順は，設計言語でも記述できなければならない．このことは一般のプログラミング言語に近い表現法を要求する．

(5) シミュレーション

設計にシミュレーションは不可欠である．シミュレーションのための時間関係などの表現は完全でなければならない．これは register transfer language でほぼ完成されている．

2.2 ミクロ言語

(1) 出力言語

この言語で記述されたプログラムを実装，または回路設計の設計プログラムに入力し，結線図，部品表，チェックのためのタイムチャートなどを作成する．このため，人間-機械系の出力言語としての役目のほか，次段のプログラムへの入力言語としての役割も果たさなければならない．もっともポピュラーなミクロ表現は，ディレーを含んだブール表現である．

これで完全に論理回路を記述することができるが，上記の出力言語としては，いくつかの機能を追加する必要がある．

(2) 回路ブロックの記述

回路はエレメント，モジュール，パッケージなど大小のブロックで構成される．これらのブロックを記述し，ブロック間の結線状態を表現することのできる言語であれば，見とおしのよい設計が可能であり，LSIを使った設計にはまさにピッタリである．

(3) ハードウェアとの対応

ミクロ言語はプログラミング言語ではアッセンブラ言語に相当し，ハードウェアと1対1に対応する必要がある．すなわち言語の各表現が現実のハードウェアを直接示していなければ，設計言語としての価値はうすれる．

以上の機能を満足するために，つぎの3段階の言語体系を導入する．

3. T-言語 (Transfer level language)

T-言語はマクロな仕様を記述する言語である．人間-機械系の最初の入力言語となる．計算機の時間的・空間的記述に適する．T-言語表現のプログラムをプロセデュアと呼び，つぎの4つの部分により構成される．

(1) 宣 言 部

計算機を設計するためには，まず，レジスタ，演算器などのモジュールまたはブロックが必要である．宣言部ではこれらの名前，種類，大きさ，応答時間など

を宣言する．宣言には，①標準的なモジュールを宣言するモジュール宣言，②標準的でない任意に記述されたブロックを宣言するブロック宣言，および③T-言語で記述された大きなユニットを宣言するプロセデュア宣言がある．ブロック宣言はB-言語そのものであり，**4.** で説明する．プロセデュア宣言はT-言語で記述されたプログラムであるから，プロセデュアがプロセデュアを含むというブロック構造をとる．

(2) 関数部

宣言部により必要なモジュールなどを宣言し，つぎに関数部で，これらモジュール間の組合せ論理（空間的結合）を記述する．順序部ではモジュール間の順序論理（時間的結合）を記述する．計算機設計のあらすじは，①各種のモジュールを集め，②おのおののモジュール間をゲートで連絡する．③このゲートに適当なタイミングで制御信号を送る制御回路を設計することである．つまり，これら①，②，③がそれぞれT-言語の宣言部・関数部・順序部に相当する．しかし，関数の必要性はもっと現実的な理由にある．たとえば，3つのレジスタ A, B, C があり，ターミナルFがオフならば A を C へ，オンならば B を C へ送る回路を記述する．順序部を使うと **Fig. 1 A** のように状態分岐が生じ，状態数が増加する．ところが **Fig. 1 B** のようにゲートを含んだセレクタブロックを関数部で記述すると，状態分岐を起こさないでレジスタの選択ができる．つまり，関数部で順序論理の一部を簡単化して記述することができる．また，オーバフローの表示など，順序部で繰り返し使う論理関数は関数部で関数として定義し，順序部では関数名（ターミナル）のみを使うことがのぞましい．これは最適化を人手で行なったことになる．順序部の最適化が変換プログラムにより完全に行なわれるならば，上記の配慮は必要ない．しかし，多出力論理関数の簡単化であり，時間的に困難な点が多い．このため，設計時に人が簡単化した記述をすることができるという点においても関数部は必要である．

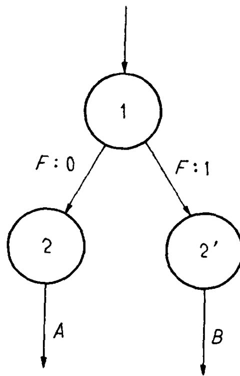


Fig. 1 A Branching of state in sequential circuit.

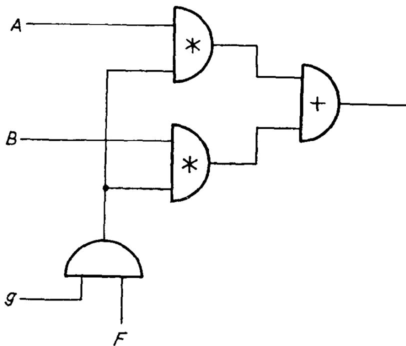


Fig. 1 B Selector block in combinational circuit.

(3) 順序部

順序部ではモジュール間のゲートによる結合およびゲートへの制御信号の発生手順を記述する．すなわち，タイムチャートをプログラム化したものとみることができる．

(4) 制御部

制御部はB-言語への変換，またはシミュレーションのためのプログラム・コントロール命令を記述する部分である．変換のためのプログラム・コントロール命令は，制御方式および制御回路の大きさの指定などに使う．シミュレーションのためのプログラム・コントロール命令は外部端子への信号入力，非同期型モジュールの時間指定，および入出力データの指定などに使う．

計算機の記述言語としては，(1)，(2)，(3) の3部分で完全に記述できる．

3.1 宣言部

モジュール宣言は標準的なモジュールを宣言するために使われる．モジュール宣言されたモジュールのインタフェース，タイミングなどは決められていて，その仕様によってM-言語に変換される．つぎに代表的なモジュールの宣言子，使用例，タイミングなどを示す．

(1) memory

memory はメモリを宣言する．メイン・メモリのみならずバッファ・メモリ，ゼネラル・レジスタ群，入出力バッファ・レジスタ群などを宣言することができ

る.

(宣言例)

memory MEMO(CORE(0: 31), AR(0: 15), 0);

は1語 32 ビット, 2^{16} 語のモジュール名 MEMO というメモリ・モジュールである. CORE は 32 ビットのデータ・レジスタ名であり, AR は 16 ビットのアドレス・レジスタ名である. 最後の0は応答時間をあらわす. メモリの呼出しはレジスタと同じく, 順序部の代入命令を使う.

(使用例)

AR=3, DATA=CORE; (3番地読出し)

AR=5, CORE=DATA; (5番地書込み)

(DATA は 32 ビットのレジスタである.)

メモリ・モジュールのインタフェースは宣言例にあらわれているレジスタ以外に, READ/WRITE 信号および終了信号が必要であるが, 変換時に必要となるため, ここでは詳述しない. タイミングはプログラム上重要である. 主記憶のサイクル・タイムは普通, ユニット・タイムの数倍または十数倍である. このため, 終了信号を検出して, つぎのステップに進む場合が多い. つまり, 非同期扱いを行なっていることになる. ところが, バッファ・メモリなどではユニット・タイムと同じか, ユニット・タイムの2倍～4倍と決まっていることがある. この場合は, 呼出すごとに終了信号との同期をとることは不要で, クロックまたは時差パルスを受けて終了とする, つまり同期扱いである. 磁気ドラム, 磁気ディスクなどを主記憶とする場合では完全に非同期である. この同期・非同期の区別のため時間指定が必要である. 応答時間の単位はユニット・タイムで, 1以上ならば同期, 0ならば非同期をあらわす.

(2) register

フリップフロップまたはそれらが集ったレジスタを宣言する. フリップフロップには RS, JK, T, D, RST などのフリップフロップがあり, その種類により, ゲートの設計方法が異なるため, register 宣言子のあとに続いて, それぞれ *rs, jk, tg, dl, st* などの性質を書く. 性質がないときは RS フリップフロップである.

(宣言例)

register DATA(0: 31), F, G, W;

register jk JKFF(0: 5);

レジスタ間の転送は代入命令を使う. この場合もタイミングが問題である. register で宣言されるレジスタの応答時間は1ユニット・タイムである (クロック周波数は制御フリップフロップの応答速度に関係し, これはデータ・レジスタの応答時間と同じである場合が多い. したがって, データ・レジスタの応答速度を1ユニット・タイムと決めてよい. なお, ユニット・タイムとクロック・タイムと同じでなくてよい. 多相パルスまたは時差パルスを使う場合は1ユニット・タイムの整数倍がクロック・タイムである). しかし, 同じ1ユニット・タイムの応答時間でも, つぎの表現ができるか否かでは, 代入命令の記述にかなりの違いが生じる. たとえば, DATA レジスタの MSB と LSB を交換する.

(表現 1)

W=DATA(0); DATA(0)=DATA(31);

DATA(31)=W;

(表現 2)

DATA(0)=DATA(31),

DATA(31)=DATA(0);

もっとも簡単なフリップフロップでは表現1の方法でなければならない. W は作業レジスタとして必要であり, 3ユニット・タイム必要とする. ところが, 高級品のフリップフロップでは表現2が可能である. この場合は作業レジスタが不要で, しかも1ユニットタイムで完了する. 表現2が可能であるか否かは, 設計時に注意すべきである.

レジスタやカウンタはプログラミング言語の変数と同じものと考えられるが, 変数よりさらにひろい性質がある. たとえば, 「F が1ならば G を1にし, F が0ならば G はそのまま」という手順は, 一般のプログラミング言語であれば, if F then G=1; などと, if 文を使って書かねばならない. ところが, G が RS, JK などのフリップフロップであれば F のセット側出力を G のセット側入力にゲートを介して結合することにより簡単に実現できる. これは set G=F; と記述する. G=F; では F が0のとき, G が0となるからダメである.

(3) counter

counter により宣言されるカウンタは, カウント数 ±1 の増加または減少カウンタである. レジスタに計数機能を付加したものとする. 状態検出は if 文で行なう.

(宣言例)

counter C(0: 3);

(使用例)

$C=C+1$;

(4) ***a*-register**

a-register は累算器を宣言する．累算器はレジスタに加算器を付加したものである（詳細は加算器を参照）．

（宣言例）

a-register ADD(ACC(0: 31), C, 3);

（使用例）

ACC＝ACC＋Y; ACC＝ACC＋Y＋C;

モジュール名 ADD という累算器は 32 ビットの長さをもち，累算レジスタ名が ACC，けた上げレジスタ名が C である．最後の数字は応答時間である．

累算器と加算器の違いは累算ができるか否かにある．累算とは，あるレジスタの演算結果を他のレジスタに移すことなく，直ちにそのレジスタに納めることである．したがって，累算器は $A=A\pm B$ という演算に適し，加算器は $C=A\pm B$ という演算に適する．もし，累算器で $C=A\pm B$，加算器で $A=A\pm B$ という演算を行なうならば1ユニット・タイムのレジスタ転送時間が増す．しかも，加算器で累算器の仕様を満足するためには，作業レジスタ1個を必要とする．これらの理由から，累算器と加算器の2つの宣言子を用意している．

(5) ***s*-register**

s-register はシスト・レジスタを宣言する．シフト・レジスタはレジフタにシフタを付加したものである（詳細はシフタを参照）．

（宣言例）

s-register SHIFT(SHIFTR(0: 31),
SHIFTC(1: 3), 4);

（使用例）

$X=C$ shr X; $X=10$ shl X;

シフト・とレジスタとシフタの違いは，累算器と加算器の違いと同じである．

(6) **adder**

adder により宣言される加算器は2つのレジスタの出力を演算数とし，それらの和または差を求め，他のレジスタに送る機能をもつ．加算器には多くの種類があるが．けた上げ保存型加算器 (carry save adder) 以外はオートマトンとしては同種のものであり，adder で宣言できる．

（宣言例）

adder ADDER(SUM(0: 31), C, 3);

モジュール名 ADDER という加算器は 32 ビットの長さをもち，和の出力端子名が SUM，けた上り出力端子名が C である．最後の数字は応答時間である．

（使用例 1）

DATA＝$X-Y$;

（使用例 2）

DATA＝$X+Y+F$, $G=C$;

加減算は算術代入命令で行なう．使用例1ではレジスタ X と Y の差をレジスタ DATA に入れる．使用例2は倍長演算などに用い，レジスタ F をけた上り入力に加え，X と Y の和を DATA に入れ，同時にけた上げ出力をレジスタ G に入れることを示す．

加算器のタイミングにおいては，入力データはすべて演算終了時まで持続させる．出力データは演算終了時に正しい値におちつくのであるが，演算開始と同時に出力ゲートも開いておく．和端子 SUM は加減算の結果のうち符号の判定，ゼロの判定，けたあふれの検出などに使うのみで，レジスタに移さない場合に使う．

（使用例 3）

SUM＝$X-Y$;

if SUM(0) then goto ABC;

除算における引きもどし法を高速化するときなどに必要である．けた上り出力端子は倍長演算に必要であるが，けたあふれを検出するときにも用いる たとえば，2の補数表示のけたあふれは $V=S(0)*rC+rS(0)*C$ である．加算器の指定に＋－などの演算子を使うため，加算器を複数使用するプログラムでは混乱が起きる．このため，加算器の宣言の順番に対応して，演算子 op⟨i⟩ を代入記号の直後に書く．i は自然数である．

（宣言例）

adder M ADDER(M SUM(0: 23), MC, 0),
E ADDER(E SUM(0: 7), EC, 0);

（使用例）

M DATA＝op 1 $X1+Y1$,
E DATA＝op 2 $X2+Y2$;

M DATA は 32 ビットのレジスタで op 1 の指定があるから M ADDER を示す．E DATA は8ビットのレジスタで，E ADDER を示す．op 1 は省略してもよい．

なお，特殊な加算器はブロック宣言で記述できるが演算子＋－は使えない．

(7) **shifter**

シフトには固定長シフトと可変長シフトがある．シフト数が整数で与えられる固定長シフトで，1ユニッ

ト・タイムに1ビットずつ行なうシフトは，単純代入命令で記述できるから問題はない．シフト数があるレジスタの内容で与えられる可変長シフトを行なうシフト，または固定長でも1ユニット・タイムに数ビットのシフトを行なうシフタは shifter で宣言する．このようなシフタは時差パルスなどの独立したクロック・パルスで制御される．

（宣言例）

shifter SHIFT(SHIFT B(0: 31),
SHIFT C(1: 4), 4);

（使用例）

DATA$=C$ shr X; X=10 shl X;

モジュール名 SHIFT というシフタは32ビットの長さであり，入出力端子名 SHIFT B，シフトカウント端子名 SHIFT C をもつ．使用例はシフト代入命令である．レジスタ DATA X, Y はすべて32ビットであり，レジスタ C は4ビットとする．shr, shl はそれぞれ右，左シフトをあらわす演算子である．タイミングおよび複数使用などは加算器とほぼ同じである．

（8） prioritor

プライオリタは優先指示をするための論理モジュールである．これは n 個の入力と出力をもち，オンである入力が多数あるとき，最左端の出力のみオンとなる回路である．一般にはレジスタともに使われ，優先制御ユニットの中核となる．これは組合せ回路であるから，応答時間は1ユニット・タイムとする．

（宣言例）

prioritor PRIO(0: 7);

（9） encoder

エンコーダはオンである入力端子の位置を値に変えて出力する論理モジュールである．一般には n 個の入力端子に対し $\log_2 n$ 個の出力端子をもつ．組合せ回路で実現でき，応答時間は1ユニット・タイムである．

（宣言例）

encoder ENCODER(INT(1: 16)/OUT(1: 4));

INT は16個の入力端子，OUT は4個の出力端子である．

（10） decoder

decoder は encoder の逆の働きをするもので説明は省略する．

（宣言例）

decoder DECODER(INT(1: 4)/OUT(1: 16));

以上10種のモジュールは論理回路と記憶回路を使って，ブロック文で記述することは可能である．しかし，標準的な計算機を設計する場合は，あらかじめ仕様の決まった既製品を使うことが多く，上記のような標準モジュールを宣言することにより，記述が簡単になる．LSI の発達により上記のモジュールを既製品化することは困難ではない．また，標準モジュールを用意することにより，シミュレーションが簡単かつ高速になる．モジュール宣言子により直ちに性質が決まるため，シミュレーション時には適当なサブルーチンを選ぶのみでよく，マクロなシミュレーションに最適である．もし，ブロック言語で表現されているならば，ビットごとのミクロなシミュレーションとなる．

（11） internal

内部端子を宣言する．モジュール間の結合，モジュールの再定義などに便利である．

（宣言例）

internal INT(0: 31);

（12） external

外部端子を宣言する．external 以外の宣言子はローカルな宣言子であり，外側のプログラムにはなんら影響は与えない．しかし，external で宣言される外部端子はグローバルな端子として扱い，外部とのデータの通信に使う．あるユニットと他のユニットとの通信は，すべて外部端子を用いて行なうものとする．レジスタやカウンタなどの能動端子をグローバルに扱うことも考えられるが，現実のハードウェアとの対応においては，端子のみを外部名とすべきであろう．

（宣言例）

external EXT(0: 31);

（13） 多次元形モジュールの宣言

レジスタや端子では，2次元のモジュールが要求される．インデックス・レジスタやチャネル端子の選択などでは，いくつかのレジスタまたは端子のうちの1群を選ぶ場合が多い．このときは，2次元の配列を考え，その断面 (cross section) を選択するように記述するのが自然である．しかし，一般のプログラミング言語のように，2次元の1要素を指定する場合はほとんどない．常に，断面を指定する．たとえば register REG(1: 10, 1: 10) と宣言しても，実際に使う場合は REG(1, ∗) とか REG(I, ∗) であり，REG(1, 2) とか，REG(I, J) などと指定する場合はほとんどない．この理由は，ハードウェアの構成上，まず，いずれかのレジスタを選択し，つぎのステップでその中の1要素を選択するからである（初めから1要素を選択するときは1次元に記述する）．しかも，断面の方向も一

定であり，REG(I, *) と REG(*, J) とが混用される場合はきわめて少ない．そこで，一般のプログラミング言語で使われている多次元配列の宣言と違った方法を採用する．

(宣言例)

register REG(1: 10)(1: 10);

(使用例)

REG(1)=REG(2)

10ビットのレジスタ群 REG が10個あることを宣言する．右のカッコがレジスタの番号，左のカッコがビット数である．使用例は2番レジスタの内容を1番に移すことをあらわす．レジスタの長さは10ビットである．なお，このように左端のカッコを最大断面として，順次，断面を定義するならば，3次元以上に拡張できるが，実用上3次元以上はあらわれない．

3.2 関数部

関数部は function 文のみである．ここで，モジュール間の空間的結合状態を記述する．モジュール間の関数関係をあらわす場合が多い．function は恒等式または論理式を含む．式中にあらわれる変数は，繰返し変数を除いて，宣言部で宣言されるものとする．

(例 1)

function ($A(0)$, $B(1)$, $C(2)$), (DATA, X);

(例 2)

function $A(0)=B(1)$, $F=A(0)\vee A(1)$;

例1は恒等式である．FORTRAN の EQUIVALENCE に相当する文であり，$A(0)$, $B(1)$, $C(2)$ は同じものとみなされる．DATA, X は数ビットのモジュールであり，それぞれ対応するビットが同一の端子とみなされる．モジュール名の再定義に有用である．例2は論理式である．$B(1)$ の出力が $A(1)$ の入力に接続されている．さらに，$A(0)$ と $A(1)$ の出力の OR が F の入力に接続されている．演算子としては，論理演算子として，∨(OR), ∧(AND), ～(NOT), ≢(EXOR), および ≡(COINCIDENCE) がある．関係演算子として >, ≧, <, ≦, ≡, ≢ がある．これらの演算子の演算順位やカッコの使い方は，一般のプログラミング言語と同じである．さらに実用上，ひんぱんにあらわれる表現を簡略にするため，OR, AND の単項演算表現と繰返し演算表現を可能とする．

(例 3)

function $F=(\vee A)$,
$G=(\wedge A(I),\ I=1, 31, 2)$,
$(A(I)=B(I)\wedge(F),\ I=1, 31)$;

第1式は単項演算表現の例である．すべての要素の OR ををあらわす．第2式は奇数番の要素の AND を求めるための単項と繰返し演算表現である．第3式は32個の同形の論理式を繰返し演算表現をした例である．単項および繰返し演算表現の式はすべてカッコでくくる．このような表現は記述上の便宜を与えるものであり，変換プログラムの最初のパス（構文分析）の段階で，一般表現に変換するため，混乱は生じない．

3.3 順序部

順序部の最初には，そのプログラムで使用するサブルーチンを記述する．サブルーチンには開サブルーチンと閉サブルーチンがあり，それぞれ open, close で始まるサブルーチン宣言文をつける．順序部の本体は時間的・空間的記述を行なう．いくつかの文で構成され，無条件文と条件文に分けられる．文と文の区切りは；(セミコロン) である．

〔A〕 無条件文

無条件文でその状態での動作を記述する．文はいくつかの命令を含み，命令と命令の区切りは，(コンマ) である．命令が集って文を構成するという表現により，動作時間の確定した並列動作は簡単に記述できる．無条件文は1ユニット・タイム以上の動作時間を必要とする．何ユニット・タイムでつぎの文へ移るかということは，文の中の命令の最大時間によって決まる．動作時間がそれぞれ異なる命令で構成された文の制御回路の構成は複雑である．これらは変換プログラムの問題であり，別稿に述べる．

(1) 代入文

代入文は，つぎの代入命令のうちのいくつかで構成される．

(例)

$A=B$, ACC=ACC+A, $F=A(0)\vee A(1)$,
$B=C$ shr B;

- 単純代入命令　モジュール間のデータ転送である．もっとも使われる回数が多い．
- 算術代入命令　加減算を記述する．演算子は＋－の2つで，演算数は2または3である．
- 論理代入命令　function の論理式と全く同じ記述ができる．function との違いはゲートが付加されることである．
- シフト代入命令　演算子は shr, shl の2つで，演算数は2である．

(2) サブルーチン・コール文

コール文はいくつかのコール命令といくつかの代入

命令で構成される．コール命令には開コール命令と閉コール命令とがあるが，みかけは同じでサブルーチン名のみを書く．

(例)

SUB; OPEN(*P*1, *P*2);

コールしたサブルーチンが開サブルーチンならば開コール命令とし，閉サブルーチンならば閉コール命令とする．開コール命令は構文分析時に開サブルーチンの本体で置き替える．しかし，閉サブルーチンは制御回路作成時までもち越され，制御方式により変換手順が変わるため，きわめてやっかいである．いずれのコール命令も多重呼出しは差し支えないが，回帰的呼出しはできない．その理由はハードウェアとの対応がつけにくいからである．

(3) goto 文

文にはラベルをつけることができる．goto 文は goto に続いてラベルを書く．

(例)

goto ABC

(4) 複 合 文

動作時間の確定しない並列動作（ループまたは待合せを含む手順）を記述するために，いくつかの文を begin と end でかこみ，複合命令とみなす．この複合命令がいくつか集って複合文となる．

(例)

begin 文，文…，文 end, begin 文，…文，end;

この場合，すべての複合命令が終了して，つぎの文へ移るとする．先まわり制御，並列処理などを行なう計算機では不可欠な文であるが，変換方法には多くの問題点がある．また，並列処理の記述には，この方法のみでは不充分である場合がある．これについては別稿で述べる．

(5) サブルーチン宣言文

サブルーチンの開始と種類を示す．サブルーチン名としてラベルを前につける．open と close があり，それぞれ開サブルーチン，閉サブルーチンをあらわす．

(例)

OPEN: open (*P*1, *P*2);

CLOSE: close;

開サブルーチンは仮パラメータを使ってもよい．閉サブルーチンは仮パラメータを使うことはできない．閉サブルーチンのパラメータの授受は，不必要な回路を生成することになるため禁止する．これらのサブルーチンはローカルなモジュールは全くもたない．メインルーチンで宣言されたモジュールを使う．

(6) end 文

プログラムの終わりに使う．

(例)

END: end;

〔B〕 条 件 文

条件文は状態分岐を示す文であり，つぎの2種がある．条件文では新しい状態は生成されない．

(1) if 文

論理式の値により2分岐する場合に使う． goto 型と wait 型がある．

(例)

if *F* then goto ABC;

if *F* then wait;

(2) switch 文

モジュールの値により多分岐する場合に使う．分岐数はスイッチ変数のビット数を n とすると，2^n である．これはデコーダと同じであるが，デコーダとの違いは，分岐先が制御回路の状態になることである．

〔C〕 ラ ベ ル

各文にはラベルをつけることができる．ラベルの区切り記号は ：（コロン）である．ラベルは状態の合流点を示すほか，状態の指示もあらわす．すなわち，ラベルの記号部のあとに*を書き，さらに自然数をかく．これによって，多相または時差クロック・パルスによる制御回路の状態指定も可能である．

(例)

ABC: ABC*1:

3.4 制 御 部

制御部では制御回路の方式を指定する．順序部で記述した制御回路は時間的表現（タイム・チャート）であり，これを制御ブロックに分割し，空間的表現の制御回路に変換する． 制御回路の変換方式には， ① sequence, ② assignment および ③ microprogram の3種がある．sequence および microprogram 方式の変換は比較的簡単である． assignment 方式は制御回路を分割するため，分割点をラベルにより，つぎのように指定する．

(例)

control assignment *jk* (*L*1, *L*2, …, *L*10);

変換を示す control に続いて変換方式の指定子を書く．つぎの *jk* は制御レジスタのフリップフロップの種類である．そのあとに，カッコでかこみ，分割点のラベルを記入する．なお，サブルーチン，および switch

文により指定されたラベルは control 文で記述しなくても分割点とみなす．なお，制御回路の設計方法については，次稿で説明する．

また，制御部でシミュレーションの制御も指定する．このための指定子として input, output および time を用意している．

4. B-言語 (Block level language)

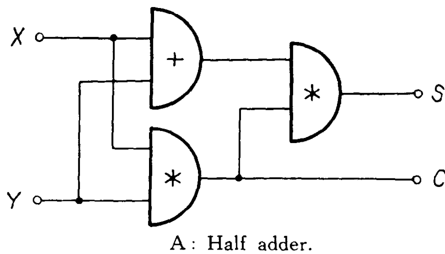
B-言語はミクロな設計言語であり，そのまま実装設計への入力言語とすることができる．計算機回路の空間的記述に適する．B-言語表現のプログラムをブロックという．ブロックは block と end でかこまれ，ブロックの本体はモジュール宣言，ブロック宣言，関数およびブロック・コールにより構成される．モジュール宣言，ブロック宣言および関数は T-言語のそれらと全く同じであるから説明を省略する．ブロック・コールは同種のブロックを開サブルーチン形式で使う方法である．ブロックがブロックを含むことから，任意のモジュール，ブロック，ユニットなどで構成されたハードウェアを1対1で記述できる．T-言語と B-言語の違いは時間的記述部分（順序部）の「ある，なし」による．

(例 1)

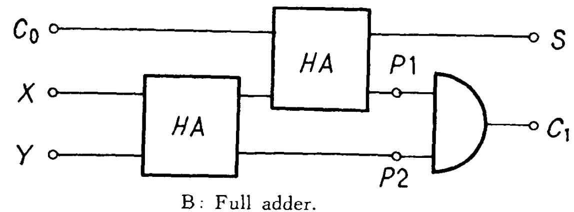
```
block HA(X, Y/S, C);
S=(X∨Y)∧~C;
C=X∧Y;
end;
```

(例 2)

```
block FA(X, Y, C0/S, C1)
internal P1, P2;
    block HA(XH, YH/SH, CH);
    SH=(XH∨YH)∧~CH;
    CH=XH∧YH;
    end;
HA(C0, P1/S, P2);
HA(X, Y/P1, P2);
C1=P1∨P2;
end;
```



A: Half adder.



B: Full adder.

Fig. 2

例1は半加算器 (**Fig. 2 A**) を記述した例である．*HA* はブロック名で，カッコに続いて入出力端子名がある．例2は半加算器のブロックを使って全加算器 (**Fig. 2 B**) のブロックを記述した例である．

HA(*C*0, *P*1/*S*, *P*2)；はブロック・コール命令である．

5. M-言語 (Module level language)

M-言語もミクロな設計言語であり，結線図の記述に適する．モジュール文のみから成り立つ．モジュール文の構成をつぎに示す．

〈モジュール文〉::=〈ラベル〉:〈モジュール名〉(〈入力端子リスト〉/〈出力端子リスト〉)

(例)

BOOL: BOOL-1*AN*(*A*(0)-*RS*1, *A*(1)-*RS*1/BOOL-3OR)

モジュール名の構成は〈名前〉-〈番号〉〈モジュール種類〉である．モジュール種類として，ME (memory), RS (RS flip-flop), AN (AND) など約20種がある．入出力端子リストには，このモジュールと接続する他のモジュールの端子名を書く．モジュール文は前後両方向へリンクするリスト構造をもつから，どの位置からでも結線図，結線表を作ることができる．

6. むすび

プログラミング言語の評価は，つぎの4点にしぼられる．①応用範囲のひろさ，②記述の難易度，③変換の難易度および④オブジェクト・プログラムの効率である．言語の設計者の目標は，上記4点の評価を最大にすることである，ところがこの4点は互いに相反する条件をもつため，評価の最大値を与える言語を決定することはむずかしい．この解決策の1つとして，人

間-機械系の利用がある．これによると，途中に人手がはいるため，多数の言語を用意しなければならない．この思想にそって，筆者らは3段階の言語を提案した．これらの言語により，各種の計算機の記述を行なっているが，かなりの能率をもって記述することができる．しかし，設計言語として充分な機能をもっているか否かは，さらに使い込むことによって判明するであろう．つぎに重要なことは，言語間の変換方法の検討，変換プログラムの作成，およびシミュレータの作成である．このうちすでに作成完了したものもあり，別の機会に述べる．

参考文献

1) Iverson, K. E.: A programming language. John Wiley and Sons, New York, 1962.
2) Gorman D. F. and J. P. Anderson: A logic design translator. Proc. FJCC, pp. 251～261, 1962.
3) Proctor, R.: A logic design translator experiment demonstrating relationship of language to system and logic design. IEEE Trans., EC-13, pp. 422-430, Aug., 1964.
4) Schlaeppi, H. P.: A formal language for describing machne logic, timing and sequencing (LOTIS). IEEE Trans., EC-13, pp. 439-448, Aug., 1964.
5) Schorr, H.: Computer aided digital system design and analysis using a register transfer language. IEEE Trans. EC-13, pp. 730-740, Dec., 1964.
6) McClure, R. M.: A programming language for simulating digital systems. J. ACM, pp. 14-22, Jan., 1965.
7) Zucker M. S.: LOCS: An EDP Machine Logic and Control Simulator. IEEE Conv. Rec., pt 3., pp. 28-50, 1965.
8) Griffin J. F. and Haims M. J.: An experiment with the simulation of machine logic and control. IEEE Conv. Rec., pt 3, pp. 51-66, 1965.
9) Chn, Y.: An ALGOL-like computer design language. C. ACM, pp. 607-615, Oct., 1965.
10) Brever M. A.: General survey of design automation of digital computers. PROC. IEEE, pp. 1708-1721, Dec., 1966.
11) Dully J. R. and Dietmeyer D. L.: A digital system design language (DDL). IEEE Trans, C-17, pp. 850-860, Sep., 1968.
12) Gerace G. B.: Digital system design automation a method for designing a digital system as a sequential network system. IEEE Trans., C-17, pp. 1044-1061, Nov., 1968.
13) Friedman T. D. and Yang S. C.: Methods used in an automatic logic design generator (ALERT). IEEE Trans., C-18, pp. 593-614, July, 1969.
14) Cook, R. W. and Flynn M. J.: System design of a dynamic microprocessor. IEEE Trans., C-19, pp. 213-222, March, 1970.
15) 高島，他：論理構成のシミュレーション・プログラム，情報処理，pp. 64-72, March, 1963.
16) 加藤，他：計算機を用いた計算機論理のデバッグについて．情報処理，pp. 73-82, March, 1963.
17) 高島，他：並列処理を用いた大容量高速論理シュミレータ．情報処理，pp. 263-272, Sep., 1966.
18) 高島：計算機の設計自動化．信学誌，pp. 290-595, April, 1967.
19) 元岡：計算機設計の自動化．情報処理，pp. 368-374, Nov., 1967.
20) 岡田，元岡：論理設計言語．信学誌，pp. 2353-2360, Dec., 1967.
21) 萩原，黒住：システム設計言語とそのコンパイラ．情報処理学会大会講演集，pp. 45-46, 1968.
22) 萩原，黒住：システム設計における B-言語．電気四学会連合大会構演集，pp. 3706, 1969.
23) 萩原，黒住：システム設計言語における M-言語からブロック図への変換．電気関係学会関西支部連合大会講演集，p. G 180.
24) 萩原，黒住：システム設計言語における T-言語から M-言語への変換．情報処理学会大会講演集，pp. 29-30, 1969.
25) 萩原，黒住：モード制御における浮動モード方式．信学会電子計算機研究会資料，1970.

(昭和45年9月21日受付)