HOKKAIDO UNIVERSITY

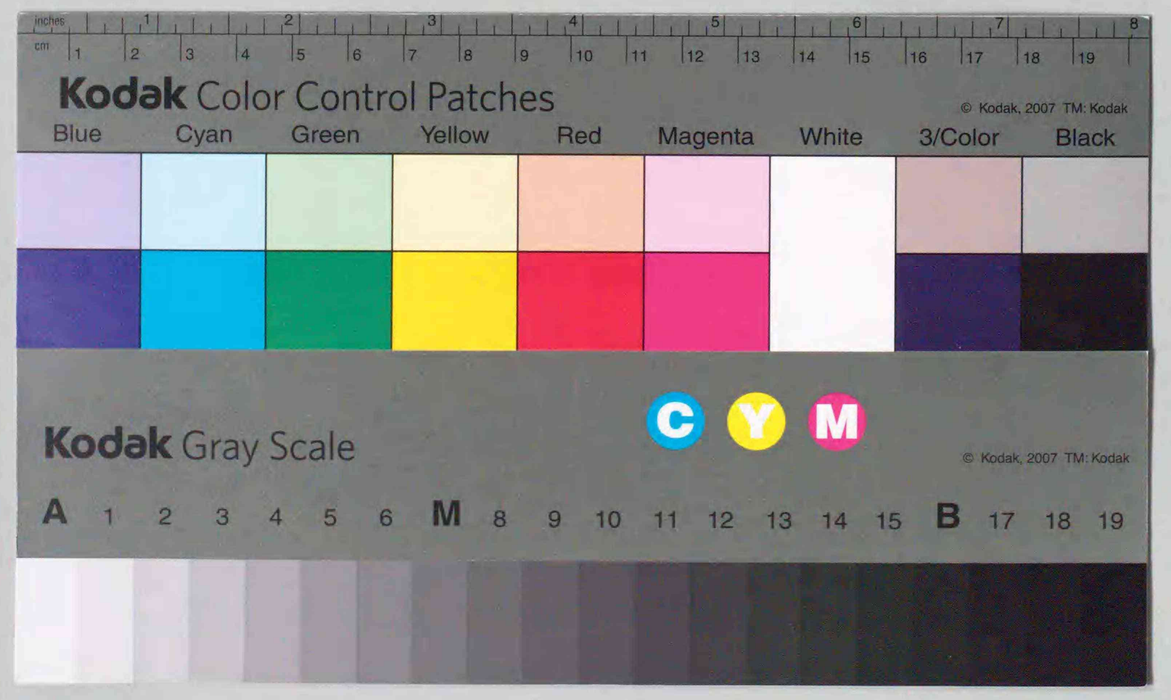
Title	VLSIシステムの構築技術に関する研究
Author(s)	平山, 正治
Degree Grantor	北海道大学
Degree Name	博士(工学)
Dissertation Number	乙第3907号
Issue Date	1991-03-25
DOI	https://doi.org/10.11501/2964276
Doc URL	https://hdl.handle.net/2115/49664
Type	doctoral thesis
File Information	000000238017.pdf

Hokkaido University Collection of Scholarly and Academic Papers : HUSCAP

VLSIシステムの構築技術
に関する研究

平山 淳

平成2年10月



(1)

ＶＬＳＩシステムの構築技術に関する研究

平成２年１０月

平　山　正　治

［目次］

第Ⅰ部　序論……(1)

1．研究の背景……(1)

1.1　ＶＬＳＩデバイス技術……(1)
1.2　ＶＬＳＩアーキテクチャ技術……(3)
1.3　ＶＬＳＩ設計支援技術……(6)

2．研究の目的……(9)

3．論文の概要……(12)

［参考文献Ⅰ］……(14)

第Ⅱ部　プログラミング言語解析チップに関する研究……(17)

1．はじめに……(17)

2．ハードウェア・モデルの設計……(20)

2.1　全体方式……(20)
2.2　字句解析アーキテクチャ……(21)
2.3　構文解析アーキテクチャ……(24)

3．アーキテクチャの評価……(30)

3.1　字句解析アーキテクチャ……(30)
3.2　構文解析アーキテクチャ……(33)
3.3　全体方式……(37)

4．ＶＬＳＩ設計環境の構築……(39)

4.1　レイアウト記述言語：ＤＰＬ……(39)
4.2　会話型レイアウト・システム：ＩＤＥＯＮ……(47)

5．言語解析チップの設計と評価……(54)

5．1　モジュール設計……(54)
5．2　チップ設計……(54)
5．3　評価……(58)

6．おわりに……(61)

［参考文献Ⅱ］……(62)

［付録Ⅱ］……(64)

第Ⅲ部　動作記述型シリコン・コンパイラに関する研究……(86)

1．はじめに……(86)

2．分散制御アーキテクチャ……(89)

2．1　動作記述言語：ＩＳＰＳ……(89)
2．2　基本モデル……(92)
2．3　構成要素の詳細と実現方式……(96)

3．動作記述型シリコン・コンパイラ……(109)

3．1　全体構成……(109)
3．2　ソースプログラム変換部……(109)
3．3　アーキテクチャ合成部……(116)
3．4　レイアウト・パターン生成部……(121)
3．5　シミュレーション環境……(125)

4．コンパイル実験……(130)

5．評価……(140)

5．1　分散制御アーキテクチャの評価……(140)
5．2　シリコン・コンパイラの評価……(143)

6．おわりに……(147)

［参考文献Ⅲ］……(149)

［付録Ⅲ］……(151)

第Ⅳ部　結論……(196)

1．研究成果のまとめ……(196)

2．今後の課題……(198)

［参考文献Ⅳ］……(203)

［謝辞］……(204)

［図・表・付録］

第Ⅰ部　序論

図Ⅰ－１　半導体デバイス技術の進歩
図Ⅰ－２　ＡＳＩＣの市場規模
図Ⅰ－３　ＶＬＳＩ設計作業の流れ

表Ⅰ－１　ＡＳＩＣの開発例
表Ⅰ－２　ソフトウェアとＶＬＳＩチップの実現形態の比較

第Ⅱ部　プログラミング言語解析チップに関する研究

図Ⅱ－１　トークンの形式
図Ⅱ－２　拡張型ＰＬＡの構造
図Ⅱ－３　字句解析チップの内部構造
図Ⅱ－４　構文解析チップの内部構成
図Ⅱ－５　Yacc のマシン・モデル
図Ⅱ－６　レイアウト・オブジェクトの参照点
図Ⅱ－７　ＤＰＬによる汎用的なモジュールの記述例
図Ⅱ－８　ＰＬＡのレイアウト方式
図Ⅱ－９　コンパイラによって生成されたＰＬＡの表示画面
図Ⅱ－１０　ＩＤＥＯＮの利用環境
図Ⅱ－１１　ＩＤＥＯＮのシステム構成
図Ⅱ－１２　exam 起動時の表示内容
図Ⅱ－１３　チャネル配線機能の使用例
図Ⅱ－１４　言語解析チップのフロアプラン

表Ⅱ－１　字句解析用ＰＬＡの諸元
表Ⅱ－２　ＰＬＡモジュールの形状
表Ⅱ－３　言語解析チップのシミュレーション結果
表Ⅱ－４　構文解析用ＰＬＡの諸元
表Ⅱ－５　モジュール・コンパイラの諸元
表Ⅱ－６　言語解析チップの諸元

付録Ⅱ－１　字句解析チップにおける状態遷移表
付録Ⅱ－２　Pascal の構文規則
付録Ⅱ－３　構文解析チップにおける状態遷移表

付録Ⅱ－４　ＰＬＡコンパイラのＤＰＬ記述

第Ⅲ部　動作記述型シリコン・コンパイラに関する研究

図Ⅲ－１　制御方式の比較
図Ⅲ－２　自己同期システムの構成要素
図Ⅲ－３　分散制御アーキテクチャの論理構造
図Ⅲ－４　チップ外部との通信方式
図Ⅲ－５　記憶モジュールの内部構成
図Ⅲ－６　演算モジュールの内部構成
図Ⅲ－７　演算モジュールの状態遷移
図Ⅲ－８　制御モジュールの内部構成
図Ⅲ－９　分散制御アーキテクチャにけるモジュール動作の例
図Ⅲ－１０　シリコン・コンパイラの全体構成
図Ⅲ－１１　ＩＳＰＳにおける制御動作の変換処理
図Ⅲ－１２　記憶モジュールのレイアウト図
図Ⅲ－１３　ＡＬＵモジュールのレイアウト図
図Ⅲ－１４　ＰＤＰ－８のフロアプラン
図Ⅲ－１５　シミュレーション環境
図Ⅲ－１６　タイミング・チャートの表示例
図Ⅲ－１７　アレイ・プロセッサの全体構成とパイプライン動作
図Ⅲ－１８　ソータチップの構成
図Ⅲ－１９　高速化した演算モジュールの状態遷移

表Ⅲ－１　ＩＳＰＳの演算子
表Ⅲ－２　演算モジュール
表Ⅲ－３　制御モジュール
表Ⅲ－４　ＩＳＰＳ構造化エディタのコマンド一覧表
表Ⅲ－５　シリコン・コンパイラで生成されたチップの仕様
表Ⅲ－６　生成されたＰＤＰ－８と HM-6100 の動作時間の比較
表Ⅲ－７　生成されたＰＤＰ－８の構成要素の面積
表Ⅲ－８　ＣＡＰｉの動作シミュレーション結果
表Ⅲ－９　ソータチップの動作シミュレーション結果
表Ⅲ－１０　シリコン・コンパイラのステップ数

付録Ⅲ－１　シリコン・コンパイラの入力記述
付録Ⅲ－２　ＧＥＮＭＡＧチップのコンパイレーション

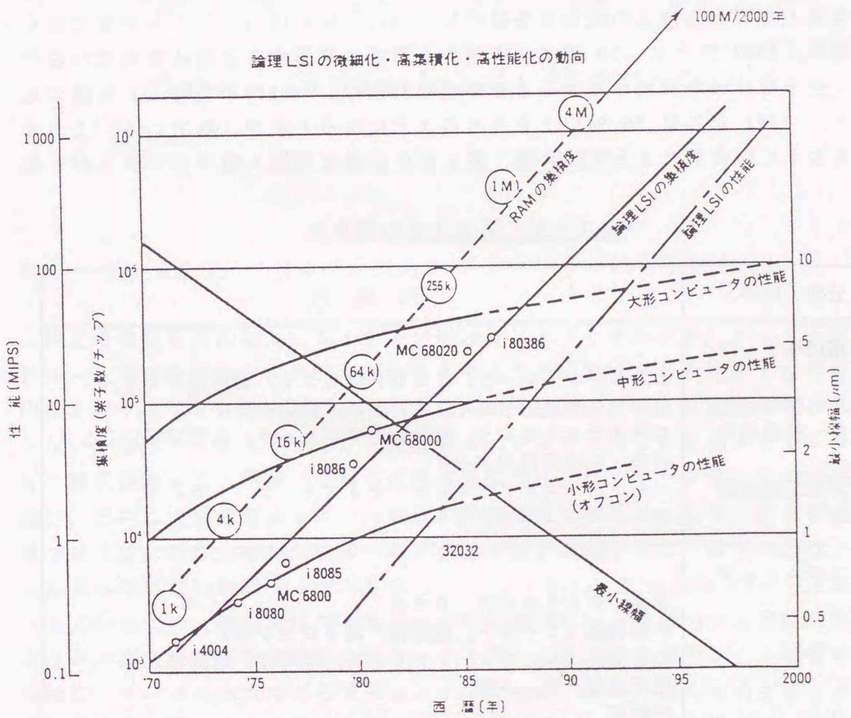
第Ⅰ部　序論

1．研究の背景

本章では，本論文で取り扱う“**ＶＬＳＩ（Very Large Scale Integration）システムの構築**”に関連する技術分野として，ＶＬＳＩデバイス技術，ＶＬＳＩアーキテクチャ技術，及び，ＶＬＳＩ設計支援技術の内容と動向について概説する．

1.1　ＶＬＳＩデバイス技術

近年のＶＬＳＩデバイス技術の進歩には目ざましいものがある．**図Ⅰ－1**に半導体デバイス技術の進歩の様子を示すように，チップ内のトランジスタの最小線幅は年々減少し，１個のチップに含まれるトランジスタ数は３年で４倍の割合で着実に増加してきている[1)]．このデバイス技術の発展は半導体メモリ（ＤＲＡＭ：Dynamic Random Access Memory）の発展に対応し，1971 年に初めて製品化さ



図Ⅰ－1　半導体デバイス技術の進歩[1)]

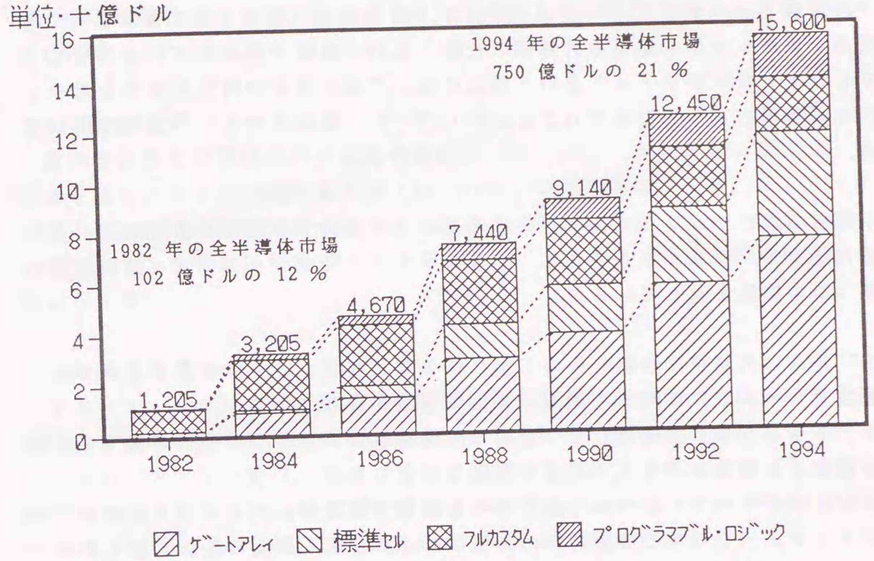
れたＤＲＡＭは１Ｋビットの容量であったが，約 20 年後の現在では，４千倍の容量をもつ４ＭビットのＶＬＳＩチップが量産体制で市販されている．また，同じ 1971 年に発表されたμプロセッサは，ＶＬＳＩ時代の幕開けとなった画期的な製品であり，多くの制御機器の電子化やパソコン等の驚異的な発展をもたらした．最初のμプロセッサは，４ビットのデータ幅で，2,300 個程のトランジスタを含む小規模なものであったが，現在では数十万トランジスタの規模で，当時の大型計算機なみの性能もつ 32 ビットμプロセッサがパソコンや制御機器等に多数利用されるようになってきている[2,3]．今後も半導体デバイス技術はこれまでとほぼ同程度の割合で継続的に発展し，21 世紀の初頭には１ＧビットのＤＲＡＭや 500 ＭＩＰＳ（Mega Instructions Per Second）の性能をもつμプロセッサが開発されると予想されている[4]．

一方，半導体メモリやμプロセッサのように多くの用途に汎用的に用いられ，大量生産されるチップに加えて，各種応用システム毎の機能に特化され，対象機器／システムを高性能化，高機能化，小型化するために専用的に用いられる**特定用途向けチップ（ＡＳＩＣ：** Application Specific Integrated Circuit）の開発も盛んになってきている[5,6]．（ＡＳＩＣの開発例を**表Ｉ－１**に示す．）**図Ｉ－２**はＡＳＩＣの世界市場規模の動向予測を示すグラフであるが，ＡＳＩＣの市場は今後とも 24 %以上の成長率を維持し，1982 年には 12 億ドル程度であった市場が，1994 年には 150 億ドルを越える市場に発展すると見込まれている[7]．また，全半導体市場に対するＡＳＩＣ市場の割合は，1982 年当時 12 %程であったが，1994 年には 20 %以上を占めるようになると予想されている．このようなＡＳＩＣの発展によって，今後，多くの大規模な情報・電子システムが１個

表Ｉ－１　ＡＳＩＣの開発例

分類・用途	開　発　例
専用標準チップ ・通信 ・音声・信号処理 ・画像・映像処理 ・大容量記憶制御	 ＬＡＮコントローラ，ＩＳＤＮ用ＬＳＩ，通信用メモリ， 音声信号処理用ＬＳＩ，音声合成／認識ＬＳＩ ＣＲＴコントローラ，映像信号プロセッサ，画像認識用ＬＳＩ 映像／画像機器用メモリ ハードディスク／フロッピーディスク・コントローラ， データベース・アクセラレータ
カスタムチップ	 電卓，デジタル時計，カメラ 家電製品（クーラー，洗濯機，電子レンジ等） ＯＡ機器（ワープロ，ファックス，複写機，電話等） 電子楽器，ゲーム機 自動車

あるいは数個のＶＬＳＩチップ上に実現されるようになり，いわゆる“**System on Silicon**”の時代が到来すると期待されている[8,9]．即ち，ＶＬＳＩ技術は，２１世紀の高度情報化社会を支える基本技術として，その重要性が益々高まっていくと考えられている．



図Ｉ－２　ＡＳＩＣの市場規模[7]

１．２　ＶＬＳＩアーキテクチャ技術

特定の機能を備えたＶＬＳＩチップを設計しようとするとき，チップ面積や設計コスト等の物理的／人為的な制約条件のもとで，要求される機能・性能を効果的に実現するＶＬＳＩチップの実現方式（**アーキテクチャ**）を決定しなければならない．即ち，ＶＬＳＩチップの設計者は，チップに含まれる演算器やレジスタ等の構成要素の数と種類，これらの構成要素間を接続するバス構造等の接続方式，及び，これらの構成要素を用いた動作の制御方式に関して，実現可能な最適解を見つけなければならない．従来の電子回路の設計においては，数個の論理ゲートしか入っていない論理チップ（ＳＳＩ：Small Scale Integration）を基本要素としていたため，設計者はできるだけ少ない論理ゲートで意図する回路を実現しようと，高度に技巧的な設計を行なってきた．しかし，数百万ゲート規模の論理回路を１チップに実現できるＶＬＳＩチップでは，ゲート数を節約することは殆ど無意味であり，それよりも，有り余るほどのゲートを有効に使って，高性能かつ高機能なチップを短期間で設計するほうが遙かに重要になってきている．従っ

て，これまでの論理回路の設計方式を根本から考え直し，ＶＬＳＩチップとして実現されることを前提として，その特質を最大限に引き出すとともに，大規模なチップを容易に実現する**ＶＬＳＩチップ向きのアーキテクチャ技術**を確立していくことが必要になっている[10.11.12.13]．

※“計算機アーキテクチャ”という用語は，命令セットやレジスタ構成等の“機械語のプログラマから見える計算機の仕様”という意味で使われているが[14]，“ＶＬＳＩアーキテクチャ”という場合には，“ＶＬＳＩの特質を活かしたチップ内部の実現方式”を意味することが多く[10.13]，本論文でもこの意味で用いている．

小規模の論理チップ（ＳＳＩ）を論理基板上に実装する従来の論理回路の実現方式と比べて，ＶＬＳＩチップは，以下に示すようなアーキテクチャ技術に関わる幾つかの特質を有している．

（１）非常に大規模な回路を極めて小さな２次元空間上に実現することができ，素子数（ゲート数）に関する制限が大幅に軽減される．
（２）トランジスタ回路としての最適な回路構成によって，従来のゲート・レベルの構成より遙かに小さく，高速な回路を実現できる．
（３）従来のゲート・レベルの設計よりも厳格で複雑なトランジスタ回路レベルやレイアウト・レベルでの設計を行わなければならず，高度の設計知識と多大の設計労力を要する．
（４）数層の限られた平面内で全ての配線を行なわなければならず，チップ上で占める配線面積や配線遅延時間等，配線に関わるコストが相対的に高くなる．また，チップの内部と外部とを接続する入出力ピン数に厳しい制約がある．
（５）チップの製造に必要な期間が長く，また，製造されたチップをあとから修正することが殆ど不可能である．

ＶＬＳＩアーキテクチャとは，このようなＶＬＳＩチップの特質を反映したものであり，以下に示すような設計方針から導かれるものと考えられる[13.15]．

（１）回路の基本単位

従来の論理回路では NAND や NOR 等の“論理ゲート”を基本単位として設計されてきたが，ＶＬＳＩチップの場合はＭＯＳ（Metal-Oxide-Silicon）トランジスタによって実現される“スイッチ”を基本単位として捉えておく必要がある．スイッチは基本的に双方向に動作するため，回路条件によっては設計者の意図した方向と逆方向に電流が流れることや，ＭＯＳトランジスタは電荷によって他の回路を駆動するということから，負荷の大きさによって動作速度が変化するなどの特性を考慮した設計を行わなければならない．

（２）構成要素

回路の基本単位としてのトランジスタ・レベルの認識は論理レベルの設計者等にとっても必要ではあるが，数百万ゲート規模のＶＬＳＩチップを設計しようとする場合には，できるだけ大きな構成要素を用いることにより，設計対象の複雑度を適切な範囲に抑える必要がある[15]．但し，この大規模な構成要素といえども，小さな回路要素を規則的に繰り返して実現された単純な構成方式のものでなければならない．

（３）接続方式

構成要素間での通信を行うための適切なバス・アーキテクチャの選択や，構成要素を張り合わせて接続する方法（abutment）の利用等によって，ＶＬＳＩチップ内の配線領域や配線遅延を減らす接続方式の設計が重要である．また，数少ない入出力ピンを有効に利用するチップ外部とのインタフェース方式を設定しなければならない．

（４）動作方式

ＶＬＳＩチップにおける高速化の要求に対して，時間的かつ空間的に最大限の並列性を追求することが必要である．このためには，データが連続的に処理されていくパイプライン動作，及び，多数の構成要素が複数のデータに対して同時に動作するＳＩＭＤ（Single Instruction Multiple Data Stream）やＭＩＭＤ（Multiple Instruction Multiple Data Stream）等の並列処理方式[16]を活用する必要がある．

（５）検証方式

論理回路を検証するために必要なテスト・ベクトルの数は，対象とする回路のゲート数の２乗に比例するといわれている．従って，１チップに含まれるゲート数の増大に伴って，従来の方法による大規模ＶＬＳＩチップの検証は殆ど不可能になりつつある．今後は，デバッグに必要な回路や，自己診断・自己修復回路を予めチップ内に組み込んでおくことなど，チップの検証方式に対する適切な設計が必要である[17]．

具体的なＶＬＳＩアーキテクチャとして，“シストリック・アレイ”[18.19]や“ＲＩＳＣ（Reduced Instruction Set Computer）方式”[20.21.22]等がこれまでに提案されている．これらのＶＬＳＩアーキテクチャでは，多数の比較的簡単な構成要素を規則正しく接続して，大規模なＶＬＳＩチップでも容易に設計できる構造をしているとともに，高度の並列性を追求するなど，ＶＬＳＩチップとしての特質が効果的に引き出されている．しかし，各ＶＬＳＩアーキテクチャのもつ特質から，適用できる分野は自ずと限定されてしまう傾向がある．例えば，シストリック・アレイを適用するためには効果的なシストリック・アルゴリズムを開発しなければならないし，ＲＩＳＣ方式は基本的にはμプロセッサの実現方式

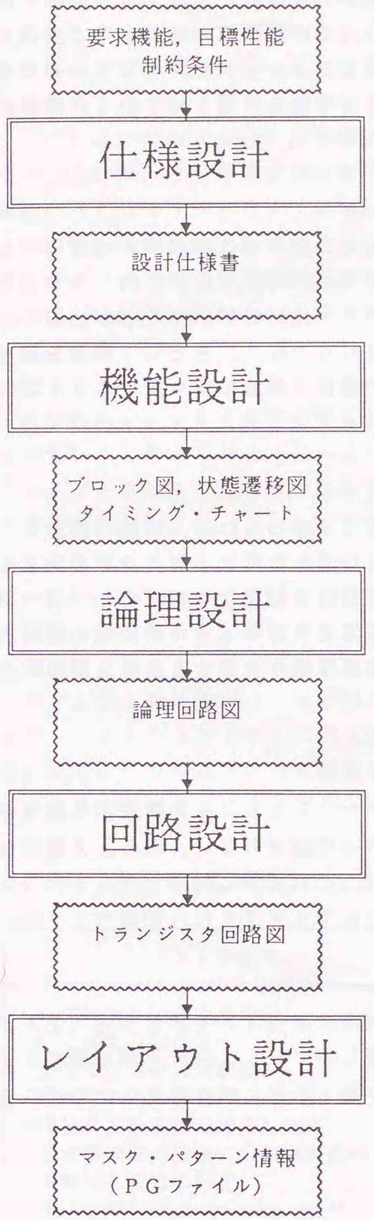
であって，様々なＡＳＩＣに適用することの効果は今後の実験・評価を待たなければならない[23]．従って，現在提案されているＶＬＳＩアーキテクチャだけで多くの応用分野に対して十分な効果が期待できるわけではなく，ある応用に対する特定のＶＬＳＩアーキテクチャの適用可能性を検討したり，あるいは，その応用に適した新しいＶＬＳＩアーキテクチャを開発するなど，専用ＶＬＳＩチップの設計実績を蓄積していくことが必要である．

1.3 ＶＬＳＩ設計支援技術

ＶＬＳＩチップは“ウェハ”と呼ぶシリコンの基板上に複数の材質（ポリシリコン層，拡散層，金属配線層等）を積み重ねて作られるため，これの製造時には各材質の幾何学的な形状を与えるマスクパターンの情報が必要である．しかし，メモリ素子のような量産品を除いて，このようなマスクパターン・レベルでチップの設計を行うことは稀であり，通常のＡＳＩＣは論理設計や回路設計等のレベルで設計されることが多い．（**図Ｉ－３**にＶＬＳＩ設計作業の流れを示す[24]．）このような各設計レベルでの設計記述を計算機に取り込み，これの正当性の検証や動作シミュレーションを行い，また，下位の設計レベルへの変換を行うために，様々なソフトウェア・ツール（**設計支援ツール**）が利用されている[25,26,27]．

半導体チップの規模がそれほど大きくなかった 1970 年代までは，論理設計レベルよりも下位の設計レベルで，論理シミュレータや回路シミュレータ等のシミュレーション・プログラムや，設計規則チェッカ等の最小限の検証プログラムが利用されていたに過ぎず，ある設計記述から下位の記述への変換作業（設計作業そのもの）は，殆ど人手作業によって行われていた．また，ＶＬＳＩチップの要求仕様や機能記述から，それを実現する論理的な構造と動作手順を決定し，さらに多数の構成要素が並列に動作する物理的な構造に展開していく高位の設計過程を支援するツールは殆ど存在しなかった．

半導体デバイスの集積度が急速に向上した 1970 年代の後半から 1980 年代にかけて，ソフトウェア開発における構造化設計の概念を取り入れた新しいＶＬＳＩ設計手法の提案[28,29,30]や，“シリコン・コンパイラ”[31,32]や“モジュール・コンパイラ”[33,34]等のＶＬＳＩ設計自動化ツールの研究開発が盛んになってきた．これらのＶＬＳＩ設計手法や設計支援ツールの発展によって，各種の高性能なＡＳＩＣの実現性が高められつつあるが，近年のＶＬＳＩチップの急速な大規模化には十分に対応できず，チップの開発期間の長期化や設計コストの高騰が余儀なくされている．また，高度の専門知識を有した少数の設計者だけにＶＬＳＩチップの設計が委ねられている状況では，本来，多品種少量生産の特質をもつＡＳＩＣの需要の急増に到底応えることができず，限られた応用分野にしかＶＬＳＩ化が進展していないのが現状である．



（１）**仕様設計**
各種の制約条件を考慮したうえで，チップ全体，あるいは，これを実現するための機能要素ごとに，入出力インタフェース，機能，性能等の外部仕様を決定する．通常，設計結果は設計仕様書として文書（ドキュメント）で表現される．

（２）**機能設計**
機能要素を実現するために，記憶要素，演算要素，バス構造等の論理的な構成と動作タイミング，制御信号等の動作方式（アルゴリズム）を決定する．設計結果はブロック図，状態遷移図，タイミング・チャート等で表現される．

（３）**論理設計**
記憶要素，演算要素，制御回路等の構成要素を具体的な論理ゲートの組合せに展開する．設計結果は論理回路図として表現される．

（４）**回路設計**
論理回路をトランジスタの回路に展開する．設計結果はトランジスタ回路図として表現される．

（５）**レイアウト設計**
トランジスタ回路を幾何学的なレイアウト・パターンに展開する．設計結果はマスクパターン情報（ＰＧファイル：Pattern Generation File）として出力され，ＶＬＳＩチップの製造工程に渡される．

図Ｉ－３　ＶＬＳＩ設計作業の流れ[24]

ＶＬＳＩチップの集積度は今後とも現在とほぼ同じ速度で増加し，西暦 2000 年頃には１千万トランジスタを越える規模の論理ＶＬＳＩが利用できるようになると予想されている[4]．しかし，一人の設計者が一定期間で設計できる対象の複雑さ，即ち構成要素の数は数千個が限度であり，この数は将来的にも殆ど変化しないと考えられる．従って，将来の大規模なＶＬＳＩチップを実現していくためには，以下に示すような方向でＶＬＳＩ設計支援技術を発展させていく必要がある．

（１）マクロ化，高位化

従来の論理ゲート・レベルでの設計を踏襲していたのでは将来の大規模なＶＬＳＩチップを開発することが現実的に不可能であり，今後は設計に用いられる個々の構成要素の規模をできるだけ大規模化（マクロ化）していくことが必要である．また，単に構成要素の規模を大きくするばかりでなく，主として構造を取り扱ってきた従来の設計から，ＶＬＳＩチップの動作や機能を取り扱うような設計レベルへと設計の高位化を図ることが必要である[15]．

（２）知能化

動作レベル，機能レベル等の高位設計を有効にするためには，下位の設計レベルへの自動的な展開が必須であり，このためには設計支援ツール自身が専門家の設計知識を保持し，最適な設計を行えるように高度に知能化されたものでなければならない．従って，人間が行っていた単純作業を代行するだけの従来の設計支援ツールから，高度に知的な設計作業自身を可能な限り支援する設計支援ツールへと発展させていく必要がある[35]．

（３）設計の蓄積

設計者が取り扱う構成要素の規模を大きくしたとしても，その構成要素自身を最初から設計するのでは設計に要する労力は何等軽減されない．即ち，大規模マクロを組み合わせてチップ設計を行うためには，これまでに開発されてきた設計遺産が多数蓄積され，誰でもが利用できるようにライブラリ化，知識ベース化されていることが必要である[36]．

これらの方向は，将来のＶＬＳＩ設計支援技術に期待される必然の方向であるが，特に，これまでの設計手法から飛躍し，新しいＶＬＳＩ設計手法を確立するという観点から，“ＶＬＳＩ設計の高位化”が最も早急に取り組まなければならない最重要課題と考えられる．

２．研究の目的

今後のＶＬＳＩデバイス技術の発展に伴って，大規模な応用システムの全体がＶＬＳＩチップとして実現される“Systems on Silicon”の時代が到来すると考えられる．この時の状況は以下のように想定される．

“ごく近い将来（when），半導体メーカではない多くの応用システムの開発現場で（where），半導体設計の専門家ではないシステム技術者等が自ら（who），現在，主としてソフトウェアによって実現されている大規模な応用システムを（what），高機能化，高性能化，小型化するために（why），高性能なＶＬＳＩアーキテクチャと高度の設計支援システムを適用して（how），ＶＬＳＩシステム： Systems on Silicon が実現されていく．”

従ってこれからは，これまで主としてソフトウェアによって実現されてきた大規模な応用システムを専用のＶＬＳＩチップによって実現しようとする傾向が強まっていくと考えられる．このような現状を踏まえ，**表Ｉ－２**に示すようなソフトウェアとＶＬＳＩチップによる実現形態の相違を考慮して，将来のＶＬＳＩシステム“Systems on Silicon”を実現していくためには，以下に述べる２つの構築方法に関する研究開発が必要である．一つは，主としてＶＬＳＩチップの設計者（**Chip Designer**）に関わる構築方法であり，チップの設計者が応用システムの処理内容を解析し，最適なＶＬＳＩアーキテクチャを適用することによって，高性能なＶＬＳＩシステムの実現を目指すアプローチである．他の一つは，主としてＶＬＳＩ設計支援ツールの開発者（**Tool Builder**）に関わる構築方法であり，応用システムのソフトウェア的な記述を受け入れてＶＬＳＩチップの製造情報を生成する設計自動化ツールを開発することによって，ＶＬＳＩシステムの短期間での容易な実現を目指すアプローチである．

表Ｉ－２　ソフトウェアとＶＬＳＩチップの実現形態の比較

ソフトウェア	ＶＬＳＩチップ
・１次元的な記述要素の並び ・記述要素の順序動作 ・記述の量が殆ど無制限 ・複雑なデータ構造 ・複雑な演算機能の簡潔な記述 ・記述要素間の関係に自由度が高い ・柔軟な入出力の記述	・２次元的な構成要素の配置 ・構成要素間の並列動作 ・実現する平面の大きさに制限 ・簡単なデータ構造のみ ・単純な基本要素を前提とした構成 ・構成要素間の接続コストが高い ・外部との通信における制約

従って，将来のＶＬＳＩシステムの構築技術を確立するためには，ＶＬＳＩアーキテクチャ技術とＶＬＳＩ設計支援技術の両方からの広範な研究開発が必要であるが，以下に示すように，我々は上記の両方のアプローチから，この分野の研究に取り組んでいる．

（１）プログラミング言語解析チップに関する研究

ＶＬＳＩチップの設計者の立場に立って，特定の応用分野の専用ＶＬＳＩチップを開発しようとする場合，以下に示すような観点から，ＶＬＳＩアーキテクチャに関する詳細な検討を行い，個々の専用ＶＬＳＩチップの設計実績を蓄積していくことが必要である．

・特定の問題に対して，どのようなＶＬＳＩアーキテクチャが適切か？
・あるＶＬＳＩアーキテクチャを実際の応用システムに適用するときの詳細な実現形態は？
・そのときの効果はどの程度か？
・実際の設計に効果的な設計支援ツールは用意されているか？

我々は，これまでソフトウェアによって実現されてきた大規模な応用システムの具体例として，プログラミング言語 Pascal のコンパイラを取り上げ，この中の言語解析処理（字句解析と構文解析）を実行する専用ＶＬＳＩチップ（言語解析チップ）に関する研究を行っている．この研究では，プログラミング言語の解析処理を実現するために，どんなＶＬＳＩアーキテクチャが適しているか，そのアーキテクチャに基づくチップの詳細な実現方式とチップの性能はどうか，また，このチップを実際に設計するために有効なＶＬＳＩ設計支援ツール等に関して詳細な検討を進め，高性能な言語解析チップを実現するための設計実績を蓄積することを目的としている．

（２）動作記述型シリコン・コンパイラに関する研究

“シリコン・コンパイラ”は，ＶＬＳＩチップの機能，動作，マクロ構造等の高位記述を入力し，これをチップの製造情報に変換するソフトウェアであり，チップの開発期間を飛躍的に短縮する設計自動化ツールとして大いに期待されている．しかし，プログラミング言語等によるチップの動作記述を受け入れる動作記述型のシリコン・コンパイラにおいては，従来からの計算機アーキテクチャを前提とすると，簡潔なソフトウェアによって効率の良いチップを自動生成することが大変困難であり，シリコン・コンパイラに適したＶＬＳＩアーキテクチャの開発が必要である．

我々は，動作記述型のシリコン・コンパイラの実現に適したＶＬＳＩアーキテクチャとしての**“分散制御型のＶＬＳＩアーキテクチャ”**と，このアーキテクチャに基づくＶＬＳＩチップを自動生成する動作記述型シリコン・コンパイラに関する研究を行っている．この研究では，チップ内の複数の構成要素が各々独自に動作する分散制御アーキテクチャが，動作記述型シリコン・コンパイラの基本アーキテクチャとして適合しているかどうか，また，このアーキテクチャが将来のＶＬＳＩシステムの実現方式として適切なものかどうかについて検討し，動作記述型シリコン・コンパイラの実現可能性を明らかにすることを目的としている．

2．論文の概要

以下に本論文の各章の概要を述べる．

第Ⅱ部では，これまでソフトウェアによって実現されてきた大規模な応用システムの具体例としてプログラミング言語 Pascal のコンパイラを取り上げ，この中の言語解析処理（字句解析と構文解析）を専用に実行するＶＬＳＩチップ（言語解析チップ）に関する研究について述べる．

第１章では，言語解析チップを開発することの意義，及び，言語解析処理のハードウェア化やＶＬＳＩ化に関する研究動向について概説し，メモリ・アーキテクチャを適用する我々のアプローチの位置付けを明らかにする．

第２章では，プログラミング言語 Pascal の字句解析と構文解析に適したハードウェア・モデル（言語解析アーキテクチャ）の設計について述べる． Pascal を対象とする字句解析は有限オートマトン，構文解析はプッシュダウン・オートマトンによって実現できるが，ハードウェアによる解析処理の効率化のためには付加機構の導入が必要であること，及び，このハードウェア・モデルが，ＰＬＡ（Programmable Logic Array）やスタック等のメモリ要素を主体とするＶＬＳＩアーキテクチャによって簡潔に実現できることを示す．

第３章では，第２章で示した言語解析アーキテクチャを評価するために行ったチップのハードウェア規模の評価，ソフトウェア的な解析手法との比較検討，及び，幾つかの例題プログラムによる動作シミュレーションについて述べる．この結果，フルセットの Pascal を対象とした言語解析チップでも簡潔なハードウェアで実現でき，実行速度の面でもソフトウェアによる解析処理に比べて２桁以上高速であることを示す．

第４章では，上記の言語解析チップを設計するために開発したＶＬＳＩ設計支援システムについて述べる．即ち，これらのチップで用いられる構成要素のモジュール・コンパイラを開発するためのレイアウト記述言語，及び，モジュール・コンパイラによって自動生成された構成要素を１個のチップとして配置・配線するための会話型レイアウト・システムについて述べる．

第５章では，第４章で示した設計支援システムを利用した言語解析チップの設計，及び，設計支援システムと設計されたチップの評価について述べる．モジュール・コンパイラによって容易に実現できるメモリ・アーキテクチャを言語解析チップに適用した結果，高性能な専用ＶＬＳＩチップが短期間で設計できたことを示す．

第６章では，プログラミング言語解析チップに関する研究のまとめとして，高性能な言語解析チップを実現するＶＬＳＩアーキテクチャとして，メモリ・アーキテクチャの有効性を明らかにできたことを示す．また，今後に残された課題について述べる．

第Ⅲ部では，動作記述型のシリコン・コンパイラの実現に適したＶＬＳＩアーキテクチャとしての分散制御型のＶＬＳＩアーキテクチャと，このアーキテクチャに基づくＶＬＳＩチップを自動生成する動作記述型シリコン・コンパイラに関する研究について述べる．

第１章では，シリコン・コンパイラについて概説するとともに，これまでに開発されている動作記述型シリコン・コンパイラの課題について述べ，分散制御アーキテクチャを適用する我々のアプローチの位置付けを明らかにする．

第２章では，動作記述型シリコン・コンパイラの基本アーキテクチャとして設定する分散制御アーキテクチャの基本モデル，即ち，構成要素，データの流れ，制御の流れ，及び，チップ外部との通信方式について述べる．また，このアーキテクチャにおける構成要素の詳細と具体的な実現方式についても述べる．特に，チップ内部での制御動作を担当する制御モジュールが簡潔な構造で実現できることを示す．

第３章では，ハードウェア仕様記述言語によるＶＬＳＩチップの動作記述から，上記の分散制御アーキテクチャに基づくＶＬＳＩチップへのコンパイル過程について述べる．この中で，シリコン・コンパイラの入力記述における条件分岐構文や並列動作の各種制御構文が，分散制御アーキテクチャにおける制御モジュールの動作に容易に変換できることを示す．また，生成されるチップの動作性能を評価するためのシミュレーション環境についても述べる．

第４章では，本シリコン・コンパイラに対して幾つか例題チップの動作記述を入力し，チップの自動合成実験を行った結果について述べる．この中で，シリコン・コンパイラで生成されたチップと手書きチップとの比較（小型計算機：ＰＤＰ８チップの場合）や，分散制御アーキテクチャにおける並列性の効果等の評価を行っている．

第５章では，第２，３，４章で述べたアーキテクチャの検討，シリコン・コンパイラの試作，及び，コンパイル実験の結果から，分散制御アーキテクチャとシリコン・コンパイラの評価について述べる．この結果，動作記述型のシリコン・コンパイラを実現するための基本アーキテクチャとして，手続き的な動作記述をＶＬＳＩチップの構造に直接反映できる分散制御アーキテクチャが有効であること，及び，分散制御アーキテクチャの各構成要素が簡潔な構造をしており，将来のＶＬＳＩシステムとしての実現性が高いことを明らかにする．

第６章では，動作記述型シリコン・コンパイラに関する研究のまとめとして，特定のＶＬＳＩアーキテクチャを適用することで，設計効率を飛躍的に改善できるシリコン・コンパイラの実現可能性を明らかにできたことを示す．また，今後に残された課題について述べる．

第Ⅳ部では，上記の研究結果をまとめるとともに，ＶＬＳＩシステムの構築に関して今後に残されている課題について考察する．即ち，ＶＬＳＩ設計における高位化，設計の蓄積，及び，統合的ＶＬＳＩ開発利用環境等の研究課題について述べる．

［参考文献Ｉ］

(1) 超ＬＳＩ Technology & Application No.5: ＡＳＩＣのすべて，エレクトロニクス別冊，オーム社，東京 (1987).
(2) 新世代マイクロプロセッサ RISC CISC TRON，日経エレクトロニクス・ブックス，日経ＢＰ社，東京 (1989).
(3) 藤井克彦編：ミニコン・マイコン応用ハンドブック，朝倉書店，東京 (1982).
(4) 通産省 電子産業中期展望懇談会 技術予測分科会編：９０年代の電子技術予測，工業調査会，東京 (1989).
(5) 富永四志夫編：ＡＳＩＣデザインハンドブック，サイエンスフォーラム，東京 (1989).
(6) 馬場敬信・小栗澄夫編：専用ＶＬＳＩプロセッサ，情報処理，Vol.31, No.4 (1990).
(7) McClean, William J., et al.: ASIC Outlook 1991, Integrated Circuit Engineering Corporation, Scottsdale (1991).
(8) Hartmann, Alfred C.: Software or Silicon? The Designer's Option, Proc. of the IEEE, Vol.74, No.6 (1986).
(9) Posa, John G.: VLSI shifts its focus from fabrication to function, Electronics, May 19 (1982).
(10) Treleaven, Philip C.: VLSI Processor Architectures, IEEE Computer, Vol.15, No.6 (1982).
(11) Ramamoorthy C.V. & Y.W. Ma: Impact of VLSI on Computer Architectures, in VLSI Electronics: Microstructure Science, Vol. 3, Edited by Norman G. Einspruch, Academic Press, Inc. New York (1982).
(12) 坂村健，石川千秋：ＶＬＳＩコンピュータ・アーキテクチャ，情報処理，Vol.24, No.2 (1983).
(13) 飯塚肇：ＶＬＳＩコンピュータアーキテクチャ，電子通信学会誌，Vol.68, No.2 (1985).
(14) Myers, Glenford J.: Advances in Computer Architecture, John Wiley & Sons, Inc., New York (1978).（渡辺勝正訳：コンピュータ・アーキテクチャの設計，共立出版，東京 (1981).）
(15) Sequin, Carlo H.: Managing VLSI Complexity: An Outlook, Proc. of the IEEE, Vol.71, No.1 (1983).
(16) Flynn, Michael J.: Some Computer Organizations and Their Effectiveness, IEEE Trans. on Computer, Vol.C-21, No.9 (1972).
(17) 樹下行三：ＶＬＳＩのテスト容易化設計技術の研究動向，情報処理，Vol.30, No.12 (1989).
(18) Kung, H.T. & Charles E. Leiserson: Algorithms for VLSI Processor Arrays, in Intorduction to VLSI Systems, Carver Mead & Lynn Conway, Addison-Wesley Publishing Co., Reading (1980).

(19) Kung, H.T.: Why Systolic Architectures?, IEEE Computer, Vol.15, No.1 (1982).
(20) Petterson, David A. & Carlo H. Sequin: A VLSI RISC, IEEE Computer, Vol.15, No.9 (1982).
(21) Katevenis, Manolis G.H.: Reduced Instruction Set Computer Architecture for VLSI, The MIT Press, Cambridge (1985).
(22) Fuber, Stephen B.: VLSI RISC Architecture and Organization, Marcel Dekker, New York (1989).
(23) パネル討論会：ＲＩＳＣはＣＩＳＣに勝るか（昭和６３年後期第３７会全国大会報告），情報処理，Vol.30, No.11 (1989).
(24) Muroga, Saburo: VLSI System Design, John Wiley & Sons, Inc., New York (1982).（渡邊 誠・菅野卓雄 日本語版監修：ＶＬＳＩシステム設計，ワイリー・ジャパン Inc.，東京 (1984).）
(25) Newton, Arthur Richard: Computer-Aided Design of VLSI Circuits, Proc. of the IEEE, Vol.69, No.10 (1981).
(26) Newton, A.R. & A.L. Sangiovanni-Vincentelli: Computer-Aided Design for VLSI Circuits, IEEE Computer, Vol.19, No.4 (1986).
(27) Rubin, Steven M.: Computer Aids for VLSI Design, Addison-Wesley Publishing Co., Reading (1987).
(28) Mead, Carver & Lynn Conway: Introduction to VLSI Systems, Addison-Wesley Publishing Co., Reading (1980).（菅野卓雄・榊 裕之 監訳：超ＬＳＩシステム入門，培風館，東京 (1981).）
(29) Smith, Robert P.: Mead-Conway: A Method for Systems in Silicon, Computer Design, May 1982 (1982).
(30) Weste, Neil H.E. & Kamran Eshraghian: Principles of CMOS VLSI Design: A Systems Perspective, Addison-Wesley Publishing Co., Readings (1985).（富沢 孝・松山泰男 監訳：ＣＭＯＳ ＶＬＳＩ設計の原理－システムの視点から，丸善，東京 (1988).）
(31) Johannsen, D.: Bristle Blocks: A Silicon Compiler, Proc. of the 16th Design Automation Conf. (1979).
(32) Southard, Jay R.: MacPitts: An Approach to Silicon Compilation, IEEE Computer, Vol.16, No.12 (1983).
(33) Stebnisky, M.W., et al.: A Fully Automatic, Technology-independent PLA Macrocell Generator, Proc. of the International Conf. on Circuits and Computers (1982).
(34) Chu K.-C. & R. Sharma: A Technology Independent MOS Multiplier Generator, Proc. of the 21st Design Automation Conf. (1984).
(35) Kowalski, Thaddeus J.: An Artificial Intelligence Approach to VLSI Design, Kluwer Academic Publishers, Boston (1985).
(36) Katz, Randy H.: Managing the Chip Design Database, IEEE Computer,

Vol.16, No.12 (1983).

第Ⅱ部　プログラミング言語解析チップに関する研究

1．はじめに

これまでソフトウェアとして実現されてきたような大規模で複雑な応用システムを高性能なＶＬＳＩシステム“Systems on Silicon”として実現しようとするとき，設計者は，その機能や性能の要求を満たすために最適なＶＬＳＩアーキテクチャを適用しなければならない．しかし，現在提案されているＶＬＳＩアーキテクチャだけで多くの応用分野に対して十分な効果が期待できるわけではなく，各応用毎に特定のＶＬＳＩアーキテクチャの適用性を検討したり，あるいは，その応用に適した新しいＶＬＳＩアーキテクチャを開発するなど，専用ＶＬＳＩチップの設計実績を蓄積していくことが必要である．

我々は，これまでソフトウェアによって実現されてきた大規模な応用システムの具体例として，プログラミング言語 Pascal のコンパイラを取り上げ，この中の言語解析処理（字句解析と構文解析）を専用に実行するＶＬＳＩチップ（言語解析チップ）に関する研究を行っている．即ち，プログラミング言語の解析処理を実現するために，どんなＶＬＳＩアーキテクチャが適しているか，そのアーキテクチャに基づくチップの詳細な実現方式とチップの性能はどうか，また，このチップを実際に設計するために有効なＶＬＳＩ設計支援ツール等に関して詳細な検討を進め，高性能な言語解析チップを実現するための設計実績を蓄積することが本研究の目的である．

プログラミング言語を対象とする言語解析処理は，コンパイラやインタプリタの中核部として，また，ＯＳ（Operating System）のコマンド・インタプリタや各種応用プログラムのユーザ・インタフェースとして広く利用され，実行される頻度の非常に高いプログラムである．従って，このような言語解析処理を効率の良いハードウェアによって実現する技術が確立されれば，これらの処理の著しい高速化が期待できるばかりでなく，プログラミング環境全体の発展にも大きく寄与すると考えられる．

プログラミング言語を対象とする言語解析法は，オートマトンや言語理論との関連で理論的に良く研究されてきた分野であり[1,2]，その成果は実際のコンパイラやインタプリタの開発に有効に活かされてきている[3,4]．一方，ＶＬＳＩ技術の急速な発展によって，プログラミング言語の解析処理等の大規模なソフトウェアを専用のＶＬＳＩシステムとして実現することの可能性が高まってきている[5]．プログラミング言語の解析処理に対するハードウェア化やＶＬＳＩ化の研究としてこれまでに，正規言語を解析するＶＬＳＩチップに関する研究[6]，高級言語の直接実行マシンにおける字句解析プロセッサの研究[7]，構文解析用ＶＬＳＩチップや専用プロセッサの研究[8,9]，複数のμプロセッサを用いたコンパイラの実現

方式に関する研究[10]等，多数の研究成果が報告されている．これらの研究では，対象とする言語の性質や適用される環境を考慮した特徴的な言語解析アーキテクチャが提案されている．

一方，ＲＡＭ（Random Access Memory），ＲＯＭ（Read Only Memory），スタック，ＦＩＦＯ（First-In First-Out）メモリ等のメモリ要素は，簡単な基本回路を規則的に繰り返して実現されるＶＬＳＩ向きの構成要素であり，このようなメモリ要素を活用して，特定の専用機能をもつＶＬＳＩシステムを実現することができる．特に，ＲＯＭと殆ど同じ回路方式で構成されるＰＬＡ（Programmable Logic Array）を利用することによって，積の和の形式で表現される任意の論理回路を実現できるため，比較的簡単な演算機能だけを含む専用プロセッサであれば，チップ全体をこのメモリ・アーキテクチャだけで実現することも可能である[11,12]．

以上のような言語解析ハードウェアやＶＬＳＩアーキテクチャに関する研究動向を踏まえ，我々は，手続き型のプログラミング言語 Pascal の言語仕様を対象として，複数のＶＬＳＩチップによってコンパイル処理全体をパイプライン的に実行するコンパイラ・マシンのアーキテクチャ，及び，コンパクトで高速な言語解析用ＶＬＳＩアーキテクチャの開発等に関する研究を行っている[13,14,15]．

言語理論やオートマトン等におけるこれまでの研究成果から，プログラミング言語を対象とする言語解析処理は有限オートマトンやプッシュダウン・オートマトンによる認識機械によって実現できることが示されており，これらの認識機械はＰＬＡやスタック等のメモリ要素によって簡単に実現できる．また，プログラミング言語の言語仕様はそれほど大きく変化しないため，言語解析処理だけを対象とするならば，μプロセッサを導入する程の柔軟性は必要なく，ＶＬＳＩチップとしてできるだけコンパクトなアーキテクチャで実現する方式が望ましいと考えられる．

従って，ＶＬＳＩシステムとしての作り易さも考慮した結果，我々は，プログラミング言語解析処理の実現方式として，“**メモリ・アーキテクチャ**”を全面的に適用した字句解析チップと構文解析チップの設計を行っている．但し，前述の言語解析処理のハードウェア化に関する研究でも指摘されているように，ソフトウェアによる実現手法をそのままＶＬＳＩチップの実現方式として適用するのではなく，ＶＬＳＩチップの特質を活かした付加機構をこの中に組み込んでいる．また，メモリ・アーキテクチャは比較的単純なＶＬＳＩシステムの実現方式ではあるが，ＶＬＳＩアーキテクチャの評価のためには実際のチップの設計が必要であり，これらの言語解析チップを設計するための設計支援ツールの開発，及び，このツールを利用した実際の言語解析チップ設計を通して，プログラミング言語の解析処理にメモリ・アーキテクチャを適用することの評価を行っている．

以下，第２章では，プログラミング言語 Pascal の字句解析と構文解析に適したハードウェア・モデル（言語解析アーキテクチャ）の設計と，このハードウェア・モデルをＶＬＳＩチップとして実現するための，ＰＬＡやスタックを主体とするＶＬＳＩアーキテクチャについて述べる．

第３章では，第２章で示した言語解析アーキテクチャを評価するために，これらのチップのハードウェア規模，ソフトウェア的な解析手法との比較検討，及び，幾つかの例題プログラムによる動作シミュレーションについて述べる．

第４章では，上記の言語解析チップを設計するためのＶＬＳＩ設計支援システムについて述べる．即ち，これらのチップで用いられる構成要素のモジュール・コンパイラを開発するためのレイアウト記述言語，及び，モジュール・コンパイラによって自動生成された構成要素を１個のチップとして配置・配線するための会話型レイアウト・システムについて述べる．

第５章では，第４章で示した設計支援システムを利用した言語解析チップの設計，及び，設計支援システムと設計されたチップの評価について述べる．

2．ハードウェア・モデルの設計

2．1　全体方式

プログラミング言語を対象とする字句解析や構文解析の解析法は，オートマトンや言語理論との関連で理論的に良く研究されてきた分野であり，その成果は実際のコンパイラやインタプリタ等の開発に有効に活用されている[3,4]．しかし，このような解析法はソフトウェアによって実現されることを前提にして提案されたものであり，専用のＶＬＳＩチップとして実現する場合には，以下に示す観点から最適なハードウェア・モデル（言語解析アーキテクチャ）を開発する必要がある．

（１）ハードウェア規模
（２）解析処理の動作速度
（３）ＶＬＳＩチップとしての実現性

ここでは，プログラミング言語 Pascal のコンパイラ[16,17]における字句解析部と構文解析部を対象として，これらの処理のＶＬＳＩ化を試みる．即ち，字句解析部は，ASCII コードの列として記述される Pascal のソース・プログラムを入力し，これを Pascal の文法で定義されている予約語（36 語），演算子，句切り記号等の特殊記号（21 種），及び，ユーザ定義の名前と定数（整数，実数，文字列の３種）からなるプログラムの言語要素を識別し，これらの言語要素を１または２バイトのトークンに変換して構文解析部に出力する．このトークンの形式を**図Ⅱ－１**に示す．

構文解析部は，字句解析部で生成されたトークンを入力し，Pascal の文法に適合しているかどうかのチェックを行い，正しいプログラムの場合にはその構文に応じた処理を促すための１から３バイトのコマンド（意味解析コマンド）を出力する．即ち，プログラムの宣言部に対しては，定数，データ・タイプ，変数，関数，手続き等のプログラム・オブジェクトを各テーブルに登録するためのコマンド（“ＴＲコマンド”と呼ぶ），また，プログラムの実行部に対しては，for，repeat 等のプログラム制御構造や数式の構造に対応したオブジェクト・コードを生成するためのコマンド（“ＣＧコマンド”と呼ぶ）を出力する．

以下では，これらの解析処理部を各々別の専用ＶＬＳＩチップ，即ち，字句解析チップと構文解析チップによって連続的に実行する方式を想定した場合の各チップのハードウェア・モデル（字句解析アーキテクチャと構文解析アーキテクチャ）について述べる．

予約語

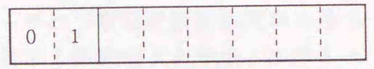
01_{16}～24_{16}（３６語）



(if, do, of, to, in, or, end, for, var, div, mod, set, and, not, nil, then, else, with, goto, case, type, file, begin, until, while, array, const, label, repeat, record, downto, packed, forward, program, function, procedure)

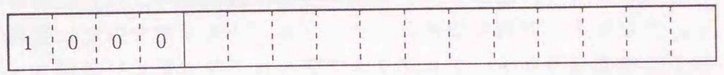
演算子，区切り記号

41_{16}～54_{16}（２１個）



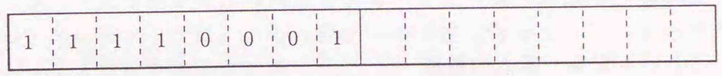
(+, -, *, /, =, <>, <, >, <=, >=, (,), [,], :=, .., ,, ;, :, ..., ^)
（※“..”と“:”は同一のコードに変換される）

名前



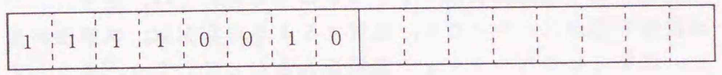
8001_{16}～$8FFF_{16}$（最大４０９５個）

整定数



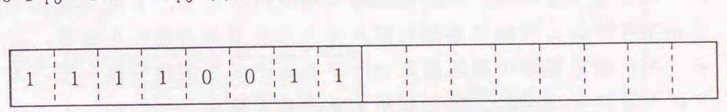
$F101_{16}$～$F1FF_{16}$（最大２５５個）

実定数



$F201_{16}$～$F2FF_{16}$（最大２５５個）

文字列



$F301_{16}$～$F3FF_{16}$（最大２５５個）

図Ⅱ－１　トークンの形式

2．2　字句解析アーキテクチャ

（１）基本方式

プログラミング言語の解析手法に関しては，従来からの研究成果として，言語の性質に応じた各種の解析モデルが作られており，以下で述べる言語解析チップもこの解析モデルをそのまま適用することから考察する．即ち，Pascal の言語要素は正規文法によって記述できるため，これを対象とする字句解析には有限オートマトンのモデルが適用できる．従って，外部から入力される Pascal ソース・プログラムの文字コードと状態レジスタの内容に応じた状態遷移を行う有限状態マシンを考えればよい．この有限状態マシンは，１個のＰＬＡ（Programmable Logic Array）と数個のレジスタから構成される簡単なハードウェアによって実

現することができる．

ユーザが定義した変数名等の名前や各種定数（整定数，実定数，文字列）もこの有限状態マシンによって識別されるが，同じ名前に対しては同一の識別コードを生成する必要がある．ここで提案する字句解析アーキテクチャでは，プログラムの言語要素に関する各種テーブル類（定数名，タイプ名，変数名，手続き名等と各々の関連情報を記憶したテーブル）を保持する記憶装置（これを“**テーブル・メモリ**”と呼ぶ）の利用を想定している．従って，字句解析チップは，名前として抽出された文字列をこのテーブル・メモリに転送し，各名前テーブルへの登録や識別コードの検索を依頼する．整定数に関しても，ユーザ定義の名前と同様の方式によって，定数テーブルへの登録や識別コードへの変換が行われる．しかし，実定数と文字列の場合は，テーブル・メモリ内で以前に登録されているかどうかの検査を行わず，プログラムで使われる毎に新しい識別コードを割り当てる．これは，可変長で比較的長い文字列からなる言語要素を検索するための処理時間を節約するためである．

（２）４文字一括入力機能

上記のような有限状態マシンによる字句解析では，通常，入力１文字毎に１回の状態遷移が行われるため，後述するように状態数，状態遷移数が非常に多くなり，結果としてハードウェア規模を大きくしてしまう．また，これに続く構文解析が言語要素１個毎に１回の状態遷移を行うため，１文字毎に１回ずつ状態遷移する字句解析との間で処理時間の格差を生じる可能性がある．このような観点から，字句解析処理の高速化とハードウェアの小型化を狙って，本字句解析チップでは１度に４文字の入力を解析する方式を採用している．

このような４文字一括入力機能を用いた字句解析処理を有効にするためには，複数の条件が同時に満たされた時に，一番きつい条件に導かれる動作だけを有効にし，緩いほうの条件による動作を抑止する機能（**抑止機能**）や，すべての条件が満たされない場合を検知する機能（**otherwise 機能**）が必要となる．例えば，以下のような４つの入力条件が定義されている場合に，文字列“ＡＢＣＤ”が入力されると，これらの条件はすべて満足するが，この場合は１番条件のきつい１番上の条件から導かれる“動作１”だけが有効であり，下の３条件からの動作を抑止する必要がある．

入力文字列＝“ＡＢＣＤ”　→　動作１　または，
“ＡＢＣ？”　→　動作２　または，
“ＡＢ？？”　→　動作３　または，
“Ａ？？？”　→　動作４

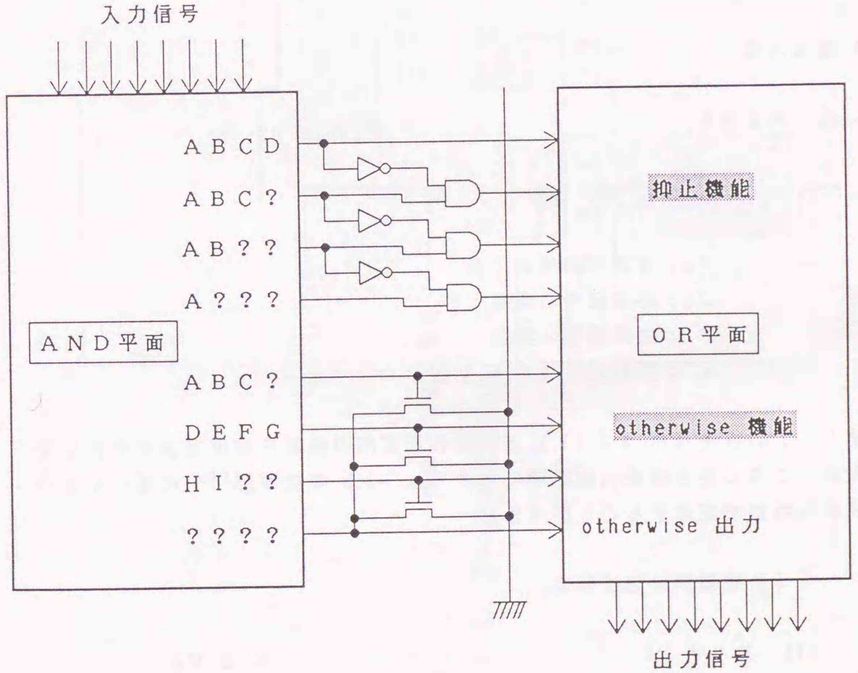
（ここで，“？”は任意の文字を許すことを意味している．）

この抑止機能が有効な例としては，連続するブランクのスキップ，“：＝”と“：”の区別，“．．”と“．”の区別等がある．

一方，otherwise 機能は，以下の例に示すように，幾つかの条件の何れも満足しなかった場合を検知するものであり，ブール式としてこの条件を定義しようとすると通常，多数の項を必要とする．

入力文字列＝“ＡＢＣ？”　→　動作１　または，
“ＤＥＦＧ”　→　動作２　または，
“ＨＩ？？”　→　動作３　または，
上記以外の場合　→　動作４

本字句解析チップでは，これらの抑止機能と otherwise 機能を実現するために，ＡＮＤ平面とＯＲ平面との接続部分に簡単なゲート回路を付加した拡張型のＰＬＡを用いている．（拡張型ＰＬＡの構造を**図Ⅱ－２**に示す．）即ち，抑止機能は，**図Ⅱ－２**の上部に示すように，上の一致出力の否定信号によって，すぐ下の一致出力をゲートする回路で実現される．



図Ⅱ－２　拡張型ＰＬＡの構造

一方，otherwise 機能は，**図Ⅱ－2**の下部に示すように，常に真となる一致出力（入力文字列＝“？？？？”）が，すべての条件が満たされないときだけ真のままＯＲ平面に出力する回路によって実現される．

（3）内部構成

字句解析チップは，**図Ⅱ－3**に示すように，字句解析のための有限状態マシンを実現する拡張型ＰＬＡ，４文字の入力文字を保持するレジスタ，状態レジスタを中核とし，これにＰＬＡから出力されるトークンを保持するトークン・レジスタ，チップ全体の制御情報を保持する制御レジスタからなる簡単な回路によって構成される．これに加えて，拡張型ＰＬＡの項数を減らすために，最初の入力文字のタイプ（英字，数字，記号の種別）を判定する小規模なＰＬＡをこれの前段に付加している．また，テーブル・メモリに転送する文字列を格納しておくためのＦＩＦＯ（First-In First-Out）メモリと，テーブル・メモリから送られる登録コードを受け取るためのレジスタ，及び，ソース・プログラムの入力とトークンの出力をスムーズに行うための２個のＦＩＦＯメモリを含んでいる．

2.3 構文解析アーキテクチャ

（1）基本方式

Pascal の文法を，

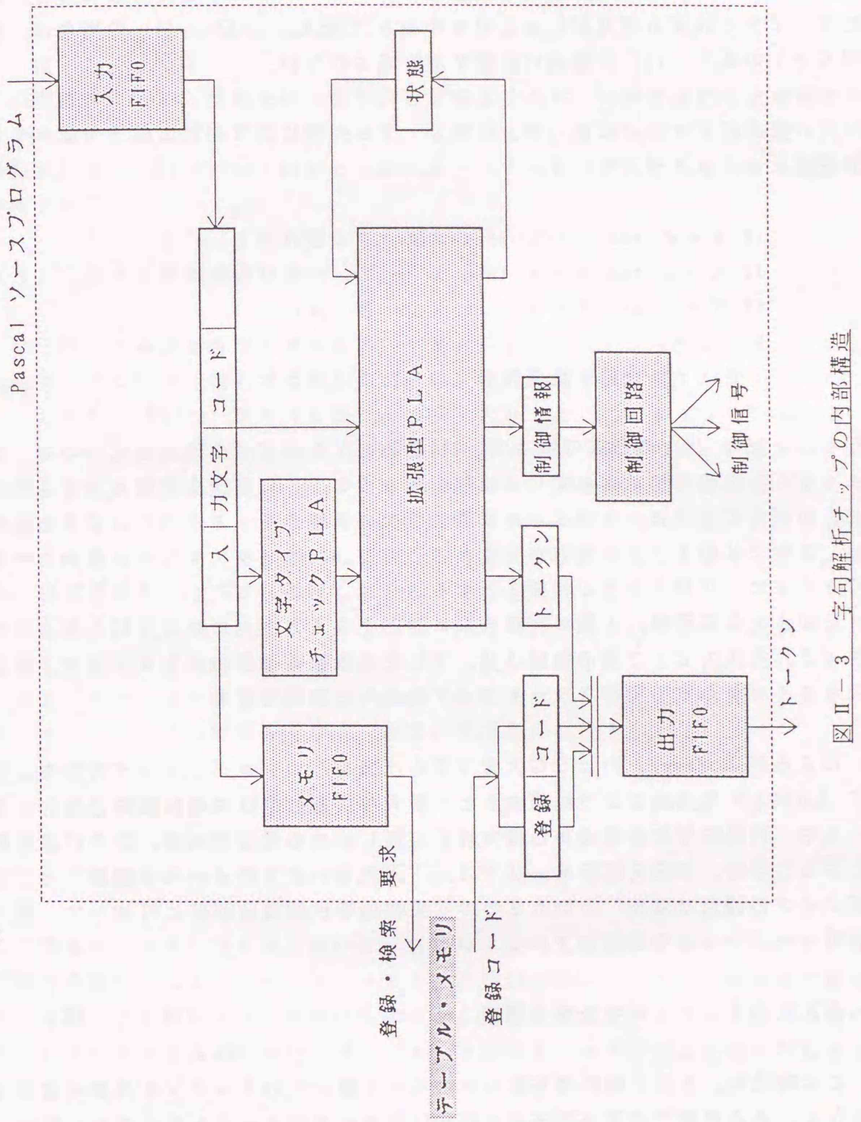
$$G = (V_N,\ V_T,\ P,\ S)$$

V_N：非終端記号の集合
V_T：終端記号の集合
P　：生成規則の集合
S　：開始記号（$S \in V_N$）

で表すと，この文法Ｇが ＬＬ(1) と呼ばれる文法の性質をほぼ完全に満たしているため，これの各生成規則は以下に示す Greibach の標準形[18] に基づく３つの形式の何れかで記述することができる．

［生成規則の記述形式］

(1)　Z → t　　$t \in V_T$
(2)　Z → t X　　$X, Y, Z \in V_N$
(3)　Z → t X Y



図Ⅱ－3　字句解析チップの内部構造

このような性質をもつ言語に対しては，プッシュダウン・オートマトンのモデルに従って，１個のトークンを先読みするだけで，バックトラックなしでトップダウンに構文解析を行うことが可能である．即ち，上記のＸ，Ｙ，Ｚを＜状態＞，ｔを＜トークン＞と考え，現在の状態Ｚにおいてトークンｔを入力したとき，状態Ｘ，Ｙを生成する構文解析マシンを考える．但し，上記の（1）の場合は，状態ＸとＹの両方，（2）の場合は状態Ｙが生成されない．

この構文解析マシンにおいて，状態Ｘ，Ｙの生成状況に応じて以下のような状態遷移とスタック操作を行う．

if Ｘ＝φ and Ｙ＝φ then スタックの読み出し → Ｚ
if Ｘ≠φ and Ｙ＝φ then Ｘ → Ｚ
if Ｘ≠φ and Ｙ≠φ then Ｘ → Ｚ，及び
Ｙ → スタックへの書き込み
（φは状態が生成されなかったことを示す．）

即ち，上記の（1）の形式の場合は，状態ＸとＹの両方が生成されないので，スタックから以前に格納されている状態を取り出し，これを次の状態Ｚとする．（2）の形式の場合は，生成された状態Ｘを次の状態Ｚとする．（3）の形式の場合は，２つの状態Ｘ，Ｙの両方が生成されるので，状態Ｙをスタックに格納しておくとともに，状態Ｘを次の状態Ｚとする．

このような処理は，１個の言語要素に対応する入力トークン（ｔ）と現在の状態（Ｚ）を入力し，２個の状態（Ｘ，Ｙ）を生成する１個のＰＬＡ，及び，１個のスタックからなる簡単なハードウェアによって実現できる．

ところが，Pascal のような大きな言語仕様をもつプログラミング言語を完全なＬＬ(1)の生成規則で記述すると，膨大な状態数や状態遷移数を必要とすることや，再帰的な文法構造から構文解析が難しくなる場合がある．この問題を解消するために，本構文解析チップでは，**“入力トークンのホールド機能”**と**“名前タイプの問合せ機能”**という２つの拡張機能を状態遷移回路に付加して，構文解析ハードウェアを実現している．

（２）入力トークンのホールド機能

この機能は，１回の状態遷移毎にかならず１個の入力トークンを消費するのではなく，ある状態での遷移が決定されないときはそのトークンをそのまま保持し，次の解析においても再度これを解析する機能である．例えば以下に示すように，状態＜block＞においてトークン“label”が入力されればラベル宣言のはじまりとして状態＜d.label1＞に遷移するが，それ以外の時はこの入力トークンをホールドして状態＜block1＞に遷移し，ここで再度，同じトークンを解析する．

＜block＞ “label” → ＜d.label1＞
＜block＞ ？ → ＜block1＞，hold トークン
＜block1＞ “const” → ＜d.const1＞
＜block1＞ ？ → ＜block2＞，hold トークン
（？は任意のトークンを意味する）

この機能によって，状態遷移の動作回数は増加するが，解析に必要な状態遷移のルール数を大幅に減らすことができる．この機能は，字句解析チップで述べた拡張型ＰＬＡの otherwise 機能と，ＰＬＡへのトークンの入力を制御する回路で実現される．

（３）名前タイプの問合せ機能

字句解析チップから出力されるユーザ定義の名前を表すトークンには，その名前のタイプに関する情報（変数名，定数名，関数名などの区別）が含まれていない．しかし，Pascal における幾つかの構文規則では，このタイプの情報によって特定の構文を決定している．（Pascal においては，名前のタイプ宣言がその利用よりも先行するため，上記のような構文チェックを行うときには既にそのタイプが決定している．）このために，本構文解析チップでは，一旦，名前を表すトークンをテーブル・メモリに転送し，そのタイプを通知してもらってから，本来の状態遷移を行う方式を採用している．

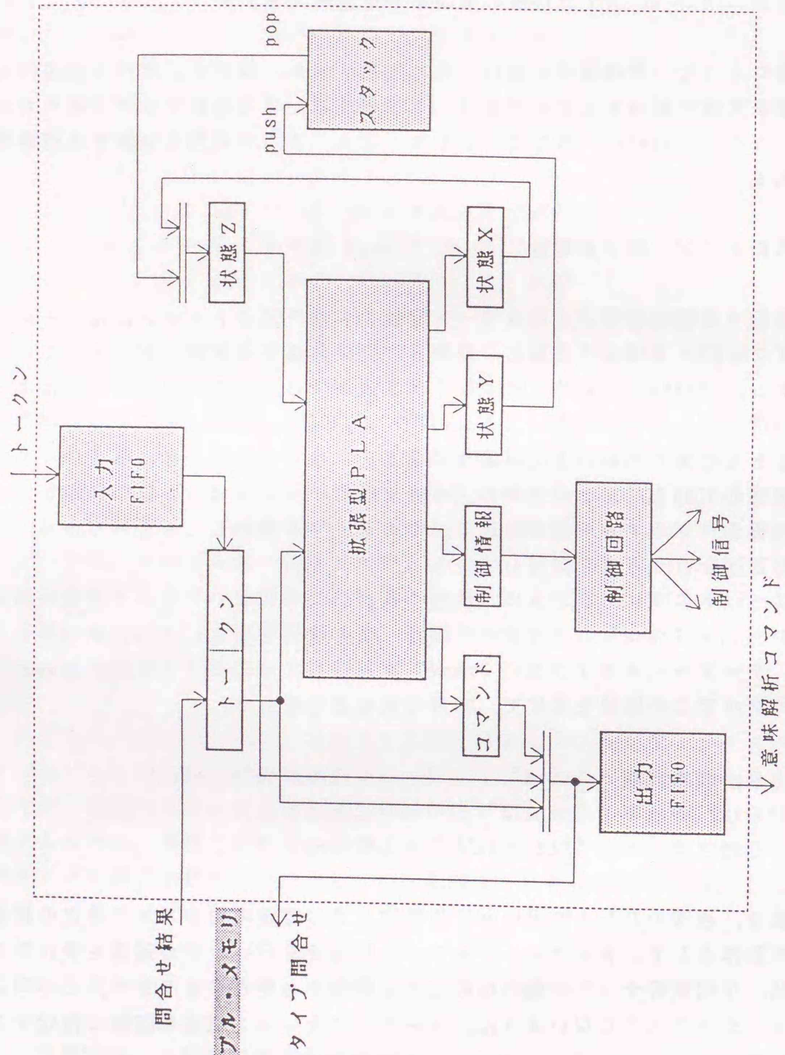
例えば，以下の例に示すように，状態＜variable＞においてユーザ定義の名前を表すトークン“name”が入力された場合，次の状態を同じく＜variable＞としたまま，テーブル・メモリにこの“name”を転送してそのタイプを問い合わせ，次のサイクルでこの結果を受けて，本来の状態遷移を行う．

＜variable＞ “name” → ＜variable＞，check トークン
＜variable＞ “var.id” → ＜var2＞
＜variable＞ “field.id” → ＜var2＞

この機能は，通常のＰＬＡを用いた状態遷移によって実現できるが，特定の状態遷移時の動作として，テーブル・メモリへの問合せを行い，その返答を受け取るまでの間，字句解析チップの動作を停止する機能が必要となる．また，この時に新しいトークンを入力しないように，トークンのホールド機能も同時に指定する必要がある．

（４）内部構成

構文解析チップは，**図Ⅱ－４**に示すように，構文解析のためのプッシュダウン・オートマトンを実現する拡張型ＰＬＡ，スタック，入力トークンを保持するト



図Ⅱ－４　構文解析チップの内部構成

ークン・レジスタ，現在の状態Ｚを保持する状態レジスタ，解析結果の状態Ｘ，Ｙを保持する２個の状態レジスタを中核として，ＰＬＡで生成された意味解析コマンドを保持するコマンド・レジスタ，チップ全体の制御情報を保持する制御レジスタからなる簡単な回路によって構成される．また，テーブル・メモリへの問い合わせを行う回路，および，トークンの入力と意味解析コマンドの出力をスムーズに行うために２個のＦＩＦＯメモリを含んでいる．

3．アーキテクチャの評価

前章で示した言語解析アーキテクチャを評価するために，以下に示す検討と実験を行った．本章では，この実験結果について述べる．

（1）フルセットの Pascal を対象として，字句解析チップと構文解析チップにおける状態遷移動作を解析し，これらのハードウェア規模を推定する．

（2）（1）と同一の文法規則を，ソフトウェアによる汎用的な字句解析ジェネレータ（Lex[19]）と構文解析ジェネレータ（Yacc[20]）に入力し，これから生成される認識マシンのハードウェア規模と（1）の結果とを比較する．

（3）ハードウェア仕様記述言語ＩＳＰＳ（Instruction Set Processor Specifications）[21]によって両ＶＬＳＩチップの構造を記述し，これに以下に示す4個の Pascal プログラム[22]を入力してチップ動作のシミュレーション評価を行う．（この時のシミュレーション結果を**表Ⅱ－3**に示す．）

ＥＸ１： 8 Queens
ＥＸ２： Topological Sort
ＥＸ３： Reference Table Generator
ＥＸ４： PL/0 Compiler

3．1　字句解析アーキテクチャ

（1）ハードウェア規模の評価

4文字一括入力機能を備えた Pascal の字句解析チップにおける状態遷移表を**付録Ⅱ－1**に示す．この状態遷移表から，本字句解析アーキテクチャにおける Pascal の字句解析は，状態数：61，状態遷移数：194 の規模の有限状態マシンによって実現できることがわかる．このとき，拡張型ＰＬＡの抑止機能は 17 カ所，otherwise 機能は 68 カ所の遷移動作で適用されている．

また，同一の Pascal 字句解析規則を字句解析ジェネレータ：Lex に入力したところ，状態数：176，状態遷移数：753 の有限状態マシンが生成された．Lex からの出力結果を以下に示す．

・状態の数：　176
・解析木のノード数：　441
・遷移の数：　753
・パックされた文字のクラスの数：　55

従って，本字句解析チップで用いられる拡張型ＰＬＡ，及び，Lex で生成された有限状態マシンを実現するＰＬＡは，**表Ⅱ－1**に示す諸元をもつと考えられる．

表Ⅱ－1　字句解析用ＰＬＡの諸元

諸　元	内　訳
字句解析チップ	
・入力信号数　40	状態：6，文字タイプ：2，文字コード：32
・積項数　194	抑止積項数：17，otherwise 積項数：68を含む
・出力信号数　21	状態：6，トークン：8，制御ビット：7
Lex の生成結果	
・入力信号数　16	状態：8，文字コード：8
・積項数　753	
・出力信号数　19	状態：8，トークン：8，制御ビット：3

これらのＰＬＡモジュールのＶＬＳＩチップ上での占有面積を推定するために，入出力信号数や積項数をパラメータとした通常のＰＬＡモジュール，及び，これに抑止機能と otherwise 機能を付加した拡張型ＰＬＡモジュールのモジュール・コンパイラをレイアウト記述言語：ＤＰＬを用いて設計した（後述）．この結果，これらのＰＬＡモジュールの占有面積は，**表Ⅱ－2**に示す計算式で算出される．（ここで，“λ”はレイアウト・パターンの形状を決める長さの基本パラメータである．）

表Ⅱ－2　ＰＬＡモジュールの形状

モジュール	パラメータ	形状（単位：λ）
ＰＬＡ	Ｉ：入力数，Ｐ：積項数 Ｏ：出力数	幅：16Ｉ＋8Ｏ＋36 長さ：8Ｐ＋123
拡張型 ＰＬＡ	（同上） Ｓ：抑止項数 Ｗ：otherwise 項数	幅：16Ｉ＋8Ｏ＋77（Ｓ≠0） 16Ｉ＋8Ｏ＋51（Ｓ＝0） 長さ：8（Ｐ－Ｓ－Ｗ）＋123 ＋15Ｓ＋9Ｗ

この結果，上記の2個のＰＬＡは，ＶＬＳＩチップ上で以下に示す大きさのモジュールとして実現される．

・字句解析チップに用いられる拡張型ＰＬＡの大きさ：　1.65 Mλ²
・Lex の生成結果を実現するＰＬＡの大きさ：　2.73 Mλ²

従って，本字句解析アーキテクチャに基づく字句解析チップは，４文字の入力データ用シフトレジスタや幾分複雑な制御回路を必要とはするものの，その本体となる拡張型ＰＬＡの面積は，通常の有限状態マシンを実現するときのＰＬＡの約 60 ％に小型化できることがわかる．

表Ⅱ－３　言語解析チップのシミュレーション結果

プログラム名		ＥＸ１	ＥＸ２	ＥＸ３	ＥＸ４
プログラム・サイズ（文字数）		９００	１,６８７	２,５８７	１４,１９３
字句解析チップ	状態遷移回数　Ｐ	８０９	１,３５５	２,０００	１１,４８１
	出力トークン数	２８０	４０９	６３５	３,４２８
	（バイト数）	３９０	５７２	８７３	４,７９８
	文字数／トークン数	３.２１	４.１２	４.０７	４.１４
	ＴＭ呼出し回数　Ｔ	１１０	１６３	２３８	１,３７０
	データ転送量（バイト）Ｄ	４５３	７８６	１,１２１	７,４５０
	実行時間（μｓ）T_1　※（２Ｐ＋１０Ｔ＋Ｄ）	３１７	５１３	７５０	４,４１１
	性能（文字数／μｓ）	２.８４	３.２９	３.４５	３.２２
構文解析チップ	状態遷移回数　Ｐ	８５９	１,１５０	１,８１２	１１,１３０
	入力ホールド回数	５７９	７４１	１,１７７	７,７０２
	スタック Ｒ／Ｗ 回数	５８４	７１６	１,１６８	７,６９６
	スタックの最大深さ	１５	１２	２１	２３
	意味解析コマンド数	４９７	６４１	１,０４５	６,５０５
	（バイト数）	６９５	９５６	１,４９０	８,９６９
	ＴＭ呼出し回数　Ｔ	７６	１２６	１６９	９５２
	データ転送量（バイト）Ｄ	３０４	５０４	６７６	３,８０８
	実行時間（μｓ）T_2　※（２Ｐ＋１０Ｔ＋Ｄ）	２７８	４０６	５９９	３,５５９
	性能（トークン数／μｓ）	１.０１	１.０１	１.０６	０.９６
全実行時間（μｓ）T_1+T_2		５９５	９１９	１,３４９	７,９７０
実行時間の比　T_1/T_2		１.１４	１.２６	１.２５	１.２４
ＶＡＸ１１／７８０での解析時間（ｓ）T_3		０.４	０.５	０.６	２.７
性能比　$T_3/(T_1+T_2)$		６７０	５４０	４４０	３４０

※ １クロック＝１００ｎｓとして計算した．　　ＴＭ：テーブル・メモリ

（２）実行速度の評価

実際の Pascal プログラムを解析したときの状態遷移回数は，**表Ⅱ－３**に示すように，プログラムの文字数の 80 ％程度である．Lex で生成される有限状態マシンは文字数と同数の状態遷移をすると考えられるため，本字句解析チップは Lex で生成されたものより約 80 ％程度高速化されていると考えられる．１個のトークンに対応するソースプログラムの文字数を計算してみると，**表Ⅱ－３**に示すように約４文字であり，４文字を一度に解析する手法は適切な設定であったと考えられる．従って，解析対象の文字数をこれ以上に増やしても，実行速度の面ではそれほど大きな性能改善を望めないことがわかる．

一方，本字句解析チップのアーキテクチャでは，名前や定数の登録と検索のために，テーブル・メモリの呼び出しが頻繁に行われ，約 2.5 個のトークンに対して１回の割合で発生している．また，この呼び出しに伴うテーブル・メモリとのデータ転送量も多く，テーブル・メモリの内部処理やデータ転送等の名前と定数の処理がチップ全体の性能に大きく影響すると考えられる．

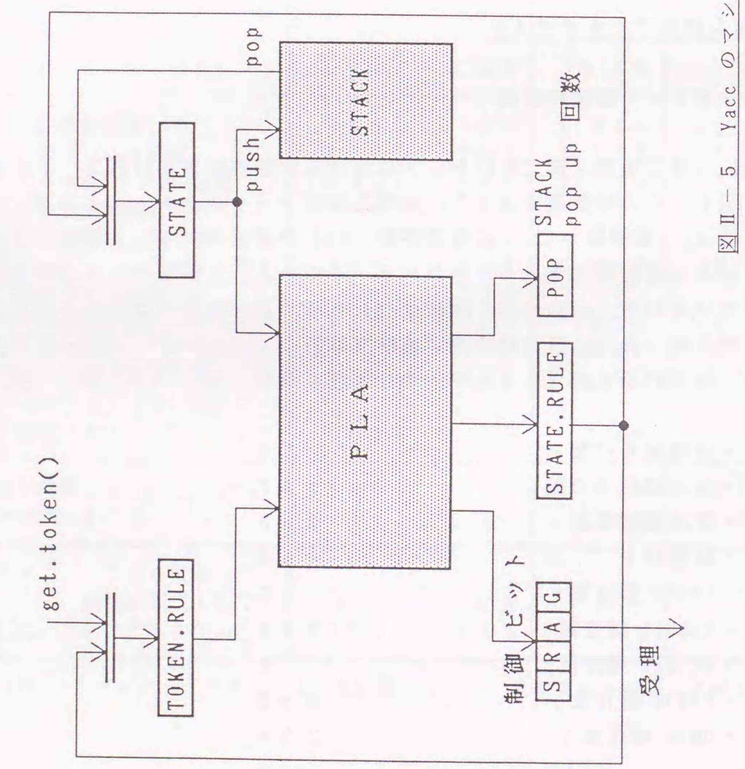
３．２　構文解析アーキテクチャ

（１）ハードウェア規模の評価

Pascal の構文規則と構文解析チップにおける状態遷移表を**付録Ⅱ－２**と**付録Ⅱ－３**に示す．この状態遷移表から，本構文解析アーキテクチャによる Pascal の構文解析は，状態数：124，状態遷移数：249 の規模のプッシュダウン・オートマトンによって実現できることがわかる．このとき，拡張型ＰＬＡの抑止機能は使われていないが，otherwise 機能は 66 カ所の遷移動作で適用されている．

また，同一の Pascal 構文解析規則を構文解析ジェネレータ：Yacc に入力したところ，以下に示す諸元をもつパーサが生成された．

- 終端記号の数：　６９
- 非終端記号の数：　１２３
- 文法規則の数：　２３３
- 状態数：　４５３
- SHIFT 動作数：　１０８５
- REDUCE 動作数：　２７３
- ACCEPT 動作数：　１
- ERROR 動作数：　１８０
- GOTO 動作数：　２６４



図Ⅱ－5 Yacc のマシン・モデル

Yacc で生成されるパーサは，**図Ⅱ－5**に示すようなハードウェアによって実現できる．即ち，このパーサは，入力トークンと状態番号，あるいは，ルール番号と状態番号の何れかを入力し，状態番号かルール番号，スタックのポップアップ回数，及び，制御ビットを出力するＰＬＡと１個のスタックによって構成される．

Yacc で生成されたパーサを実現するＰＬＡの積項数は，shift，reduce，accept，goto の各動作数を合計した数であり，これらへの入力信号数と出力信号数は各レジスタのビット幅を合計したものである．（厳密には，error 動作数も積項数に加算する必要があるが，本構文解析チップでもエラー処理を考慮していないため，これを除いて計算する．）従って，本構文解析チップで用いられる拡張型ＰＬＡ，及び，Yacc で生成されたパーサを実現するＰＬＡは，**表Ⅱ－4**に示す諸元をもつと考えられる．

表Ⅱ－4　構文解析用ＰＬＡの諸元

諸　元		内　訳
構文解析チップ		
・入力信号数	１５	入力トークン：８，状態コード：７
・積項数	２４９	otherwise 機能：６６を含む
・出力信号数	２９	コマンド：８，状態コード：１４，制御コード：７
Yacc の生成結果		
・入力信号数	１７	状態：９，トークン／規則：８
・積項数	１６２３	shift：1085，reduce：273，accept：1，goto：264
・出力信号数	１５	状態／規則：９，pop up 数：２，制御ビット：４

これらのＰＬＡのＶＬＳＩチップ上での占有面積を前述の計算式で算出してみると，以下のようになる．

・構文解析チップに用いられる拡張型ＰＬＡの大きさ：　１．１４　Ｍλ^2
・Yacc の生成結果を実現するＰＬＡの大きさ：　５．６１　Ｍλ^2

従って，Yacc によって生成される ＬＲ(１) 構文解析アルゴリズムを実行するパーサは，本構文解析チップにおけるＰＬＡより５倍近くも大きな面積を必要としている．この差は主として構文解析マシンのモデルに依存するものと考えられ，Pascal のようにトップダウンに解析できる言語の場合には，一般のＬＲパーサよりも遙かに簡単なハードウェアによって構文解析ができることを示している．

（２）入力トークンのホールド機能

本構文解析チップにおける状態遷移では，入力トークンのホールド機能を 43 カ所の遷移で利用している．この機能の使われ方としては，以下に示す３形態がある．

① 詳細な構文解析を保留し，別の状態に移って，再度同じトークンを解析する場合．（**付録Ⅱ－３**のホールド機能のうち，注釈がつけられていない 19 カ所がある．）
② 現在解析中の構文が解析終了し，１段上位の構文に戻る場合．（**付録Ⅱ－３**のホールド機能のうち，注釈１で示した 11 カ所がある．）
③ 現在解析中の構文が解析終了し，このときの終了処理用の意味解析コマンドを生成する必要のある場合．（**付録Ⅱ－３**のホールド機能のうち，注釈２で示した 13 カ所がある．）

①の場合は解析に必要なＰＬＡの項数を減らす効果があるが，状態遷移の動作回数は多くなるため，実行時間は増大する．②の場合は現在解析中の構文がどのような上位の構文において参照されているかが特定できないため，多くの構文から参照される statement や variable 等 のような構文をこの機能を使わないで実現することは大変困難と思われる．③の場合は，構文解析として本来不要な状態遷移ではあるが，意味解析コマンドを発生するために必須の状態遷移である．

表Ⅱ－３に示すシミュレーション結果では，入力トークンの数の約３倍の状態遷移を行っており，１個のトークンに対して，平均２回ずつホールド機能が働いていることが分かる．上記の②，③の利用形態でのホールド機能は必須なものであるが，①の利用形態に関しては実行速度とハードウェア量のトレードオフについて，詳細な検討を行う必要がある．

（３）名前タイプの問合せ機能

名前のタイプが指定されている構文を解析するために，本機能は 15 カ所の状態遷移に適用されている．この中で，statement の <st2>，factor の <factor>，variable の <variable>，field list の <f.list4> の４カ所に関しては，名前のタイプによって異なった状態遷移をするため，この機能が有効に利用されている．これ以外の 11 カ所（**付録Ⅱ－３**のチェック機能のうち，注釈３で示している）に関しては，各構文とも１種類の名前タイプしか許さないため，この機能を適用しなくても構文解析を進めることが可能である．但し，許されないタイプが宣言された名前を，この時点で構文エラーとして検出できる効果がある．

シミュレーションにおけるこの機能の発生状況は，入力トークンの数に対して 28 %程度であり，字句解析チップと同様にテーブル類の参照，検索処理の高速化が要求される．但し，上で述べたように，構文解析チップとしてはこの問合せ

を省力できる場合もあり，ある程度の高速化は可能である．

３．３　全体方式

現在の技術水準から基本周波数 10 ＭＨｚ（クロック幅：100 ｎ秒）の速度でこれらの言語解析チップが動作できると仮定すれば，字句解析チップは１秒間に約 328 万文字の Pascal ソース・プログラムを解析でき，また，構文解析チップは１秒間に約 98 万トークンを処理する能力を持っていると考えられる．この値は，**表Ⅱ－３**に示すように，ＶＡＸ 11/780 の Pascal コンパイラにおける言語解析能力に比べて 300 倍以上高速であると推定される．従って，言語解析専用のＶＬＳＩアーキテクチャを開発することにより，ソフトウェアに比べて２桁以上も高速化できることが示された．

また，字句解析チップにおいて，平均 4.1 文字のソース・プログラムから１個のトークンが生成されていることから，字句解析チップは１秒間に平均 80 万個のトークンが出力される．一方，構文解析チップは１秒間に約 98 万トークンを処理する能力を備えていることから，字句解析チップの出力を約 23 %程度の能力の余裕をもって処理することができる．従って，これらチップ間のデータの流れは非常にスムーズであり，４文字一括入力を行うことの効果を実証できたと考えられる．一方，Lex で生成される通常の有限状態マシンは本字句解析チップより 20 %程度性能が低下するため，構文解析チップの処理能力との格差が広がり，構文解析チップが遊んでしまう可能性が高い．

字句解析チップと構文解析チップの両チップともテーブル・メモリの呼出しが頻繁に行われ，字句解析チップでは約 10.3 文字に１回，構文解析チップでは約 3.6 トークンに１回の割合で発生している．また，この時のデータ転送量も多く，字句解析チップは約 5.2 バイト／回，構文解析チップは約 4.0 バイト／回のデータ転送が行われている．従って，テーブル・メモリとのデータ転送速度とメモリ内部の登録動作や検索動作を高速化することが，これら言語解析チップの性能向上に大きく貢献すると考えられる．

一方，今回設計した言語解析チップにおいては，解析エラーが発生したときの動作については特に考慮されていない．（字句解析エラーや構文解析エラーが発生したときは，各ＰＬＡから正常な状態情報が出力されないため，これをチェックすることによって，エラーの発生を検出することは可能である．）実際のコンパイラではエラーの発生を利用者に通知するとともに，この後の解析処理を継続したり，中止する等の操作の切り替えが必要となる．言語解析チップのような専用アーキテクチャを用いると，ハードウェア化による大幅に高速なコンパイル処理が期待できるが，エラー発生時のような例外処理に対する柔軟性が低下するという問題もある．従って，エラー発生時には必要最小限の情報だけをソフトウェアに渡して，繁雑な処理を依頼するようなソフトウェアとハードウェアの協調動

作や，エラー処理のような例外処理への対応，及び，マンマシン・インタフェース機能等に関しても，今後，検討していく必要がある．

また，言語解析用ＶＬＳＩチップの他のプログラミング言語に対する適応性を評価するとともに，言語処理システム全体のＶＬＳＩ化のために，コンパイラにおけるコード生成部やテーブル類の処理部，あるいは，インタプリタにおける実行処理部のＶＬＳＩ化等も今後の課題として残されている．

4．ＶＬＳＩ設計環境の構築

第２章で示したように，プログラミング言語解析チップ（字句解析チップと構文解析チップ）は，ＰＬＡ，スタック，ＦＩＦＯ，レジスタ等の要素を主体として構成されている．ＰＬＡは元来，ＲＯＭ（Read Only Memory）の回路技術を適用して作られており，これら全ての構成要素はメモリ構造のモジュールである．このようなモジュールは単純な基本回路を繰り返し用いて設計できるという観点から，メモリ要素を利用したＶＬＳＩチップの構成方式も典型的なＶＬＳＩアーキテクチャと考えられる．これらの繰返し構造のモジュールは，いくつかのパラメータを含んで汎用的に定義された“**モジュール・コンパイラ**”を利用することにより，任意の特性の構成要素を自動生成できる．従って，これらの構成要素を適切に配置・配線するだけでチップ全体を容易に設計することが可能である．

我々は，ＶＬＳＩチップのマスクパターン・レベルの設計を“プログラム”として記述できるレイアウト記述言語：**ＤＰＬ**（Design Procedure Language）[23]によって，プログラミング言語解析チップで用いられる構成要素のモジュール・コンパイラを試作した．また，モジュール・コンパイラによって生成された構成要素の配置作業と，これらの間の配線作業を支援する会話型レイアウト・システム：**ＩＤＥＯＮ**（Interactive DEsign envirONment）を構築することによって，言語解析チップのチップ設計を試みた．本章では，レイアウト記述言語：ＤＰＬと会話型レイアウト・システム：ＩＤＥＯＮの概要について述べる．

4．1　レイアウト記述言語：ＤＰＬ

ＶＬＳＩチップレイアウト設計は，チップの製造に使われる複数のマスク毎の幾何学的な形状を決定することであるが，何万個，何十万個という多角形のレイアウト・パターンを紙の上に描き，これの端点の座標を読みとって，検査する作業は非常に手間と時間のかかる煩雑な作業であった．1980 年代になって，このような作業を回避するために，個々のレイアウト・パターンの形状とパターン間の相対的な位置関係を簡潔に記述するレイアウト設計用のプログラミング言語が提案されるようになってきた[24]．レイアウト記述言語を用いることにより，ＶＬＳＩチップのレイアウト設計を通常のソフトウェア開発過程と同様な方式で，記述，修正，デバッグできるようになる．特に，ソフトウェア開発における“階層化設計”の概念をレイアウト設計に適用することにより，これまで物理的な１階層の平面上に描かれてきたレイアウト・パターンを，論理的に階層化された設計として取り扱えるという非常に大きな効果がある．

さらに，各構成要素（モジュール）の設計において，いくつかのパラメータを含んだ標準的なレイアウト・パターンをレイアウト記述言語で記述しておき，あとからパラメータに適当な値を設定して，任意の特性のモジュールを自動生成す

る“モジュール・コンパイラ”[25,26]を実現することが可能である．このように，レイアウト記述言語はレイアウト設計の設計効率を大幅に改善し，将来の大規模チップの設計を可能にする新しいＶＬＳＩ設計手法として，現在広く利用されるようになってきている．

我々はマサチューセッツ工科大学で提案されたレイアウト記述言語：ＤＰＬを我々の計算機環境：DEC System 2060 にインプリメントし，前章で示したプログラミング言語解析チップ等の設計に適用した．ＤＰＬはオブジェクト指向の考え方を取り入れ，数々の斬新な機能を備えたレイアウト記述言語であり，特に，モジュール・コンパイラを容易に実現できる言語として，プログラミング言語解析チップのようなメモリ・アーキテクチャのチップの実現に非常に有効である．以下ではＤＰＬの機能の概要とモジュール・コンパイラの実現について述べる．

（１）基本機能

ＤＰＬは Lisp の環境下でオブジェクト指向の考え方を導入したレイアウト・パターン記述用のプログラミング言語であり，あるレイアウト・オブジェクトの class（型）を，他の class の instance（実体）を用いて定義していくという階層的な設計を行う．この目的のために，“defcell”，“partq”と呼ぶ２個の基本関数が用意されている．defcell はレイアウト・オブジェクトの class を定義する関数であり，partq は class に属する１個の instance を生成する関数である．この class の定義においてはいくつかのパラメータを導入することができ，instance を生成する時点で設定されるパラメータの値に応じて，同じ class に属するが，その大きさ，形状等が異なる様々な instance を生成することができる．

ＤＰＬでは，最下層の class として“rectangle（矩形）”が予め組み込まれており，ＤＰＬで記述されるすべてのレイアウト・パターンは最終的に，この rectangle の組合せに展開される．この rectangle に対して，その矩形が属するマスクの層を指定する“layer”，及び，矩形の長さと幅を指定する“length”と“width”がパラメータとして設定されている．従って，rectangle に属する instance の生成は以下のように記述される．

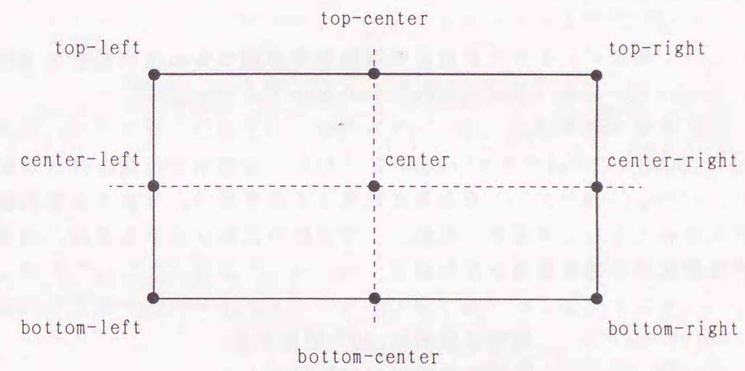
```
(partq "name" rectangle (layer <layer name>) (length m) (width n))
```

ここで，"name" はこの instance につける名前であり，ｍ と ｎ はそれぞれＹ軸方向，Ｘ軸方向の長さを表す数値である．この値は距離の基本単位“λ”によって記述される．<layer name> はマスクパターンの層名であり，適用するプロセスに応じて定義される．例えば，代表的なｎＭＯＳプロセスであれば，poly（ポリシリコン層），diff（拡散層），metal（金属配線層），ion（イオン注入層）

等の７個の層が定義されている．ＤＰＬの記述例に関しては，**付録Ⅱ－４**に示したＰＬＡの記述を参照されたい．

（２）配置機能

ＤＰＬでは，上記の partq によって生成される instance の適切な位置への配置を指定できる．この指定方法には２通りあり，１つは partq のパラメータによって instance の生成時に指定する方法であり，もうひとつは，ＤＰＬの配置関数“align”によって，既に生成されている instance を特定の位置に配置する方法である．各レイアウト・オブジェクトには，**図Ⅱ－６**のように，９個の参照点が定義されており，これらの点の何れかを，他の instance の参照点との相対的な位置関係，あるいは，座標軸上の絶対的な位置で指定することによって，適切な場所に配置することができる．



図Ⅱ－６　レイアウト・オブジェクトの参照点

例えば，instance P1 の下辺の中央を instance P2 の上辺の中央に合わせて配置するときは，P1 を生成する partq のパラメータとして，以下のように記述すればよい．

```
(partq P1 rectangle (layer 'poly) (length 4) (width 10)
                    (bottom-center (>> top-center P2)))
```

ここで，これと同じ配置を配置関数 align を用いて，以下のように記述することもできる．

```
(partq P1 rectangle (layer 'poly) (length 4) (width 10))
(align (>> P1) (>> bottom-center P1) (>> top-center P2))
```

参照点の位置を指定するために，点の座標に対する演算が必要であり，ＤＰＬでは下記のような６個の関数が用意されている．

(pt-sum <pt1> <pt2>)	点 pt1 と点 pt2 を加算した点
(pt-offset <pt> <dx> <dy>)	点 pt から（dx, dy）だけ移動した点
(pt-to-left <pt> <dx>)	点 pt から dx だけ左に移動した点
(pt-to-right <pt> <dx>)	点 pt から dx だけ右に移動した点
(pt-above <pt> <dy>)	点 pt から dy だけ上に移動した点
(pt-below <pt> <dy>)	点 pt から dy だけ下に移動した点

ここで，<pt>，<pt1>，<pt2> は点の座標であり，<dx>，<dy> は変位を表す数値である．

（３）その他の基本機能

ＤＰＬには，簡潔なレイアウト記述を可能にする幾つかの便利な機能（関数）が用意されている．このうちの代表的なものを以下に示す．

defcell によって定義された class は，特定の座標系での点の位置が規定されている．しかし，instance を実際に利用するときには，それを回転したり，向きを反転することなどが必要になる．この目的のためにＤＰＬでは，以下に示す７個の座標変換関数が用意されている．

rot90	反時計方向に 90° 回転する
rot180	反時計方向に 180° 回転する
rot270	反時計方向に 270° 回転する
negx	ｘ座標の符号を反転する
negy	ｙ座標の符号を反転する
int-pos	rot90 を実行後，negx を実行する
int-neg	rot90 を実行後，negy を実行する

多くのレイアウト設計においては，１個のレイアウト・オブジェクトを一様な方法で繰り返し用いることがある．ＤＰＬではこの目的のために，繰返し関数“replicateq”が用意されている．繰返しの方法としては，１個のレイアウト・オブジェクトを横方向に並べる“row”，縦方向に並べる“column”，及び，２次元格子状に並べる“array”の３種類がある．例えば，５個の simple-transistor を４λずつ離して横に並べる場合は，下記に示すように，最初に繰返しの要素となる simple-transistor の１個の instance を partq によって生成し，次に replicateq の引数で並べる個数とともに，この instance を指定する．（ここで ST, ST-ROW は各 instance につけた名前である．）


```
(partq ST simple-transistor (channel-width 4))
(replicateq ST-ROW row (list 5) (list (>> ST)) (h-spacing 4))
```

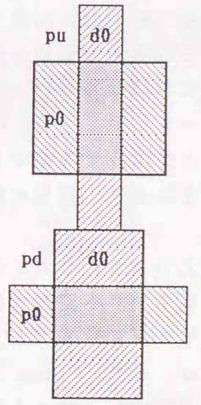
（４）モジュール・コンパイラの実現

“モジュール・コンパイラ”あるいは“モジュール・ジェネレータ”とは，ＶＬＳＩチップの構成要素として頻繁に利用される算術演算回路（ＡＬＵ：Arithmetic Logic Unit），レジスタ・ファイル，ＲＯＭ，ＲＡＭ，ＰＬＡ等のある程度大きな構成要素（機能モジュール）を，構造，回路，電気，形状等に関する幾つかのパラメータを含んだ一般的な形式で定義したものである．従って，ユーザ（設計者）が適当なパラメータの値を設定するだけで，特定の仕様を満足する機能モジュールが自動生成され，これらを用いることによりＶＬＳＩチップの設計を簡単に行うことができる．このようなモジュール・コンパイラの実現のためには，パラメータの導入，階層化，一般化等のソフトウェア的な開発手法が有効であり，ＶＬＳＩチップのレイアウト・パターンをテキスト形式で記述する“**レイアウト記述言語**”が，この目的のために利用されている．

例えば，レイアウト記述言語：ＤＰＬでは，基本的なトランジスタのレイアウト・パターン（ここでは“fet”と名付けている）の定義において，トランジスタを構成するチャネルの幅と長さをあらわすパラメータ“wid”と“len”を含んで一般的に定義しておくことができる．（**図Ⅱ－７参照**）こうすれば，形状の異なる２個の“fet”を含む“inverter”を設計するときも，パラメータの値を変更するだけで各々のレイアウト・パターンを生成することができる．

```
(defcell fet ((wid 2) (len 2))
   (partq p0 rectangle (layer `poly)
      (width (+ (>> wid) 4)) (length (>> len)))
   (partq d0 rectangle (layer `diff)
      (width (>> wid)) (length (+ (>> len) 4))
      (center (>> center p0))))

(defcell inverter ()
   (partq pd fet (wid 4))
   (partq pu fet (len 4)
      (bottom-center (>> top-center pd)))
      . . .          )
```



図Ⅱ－７　ＤＰＬによる汎用的なモジュールの記述例

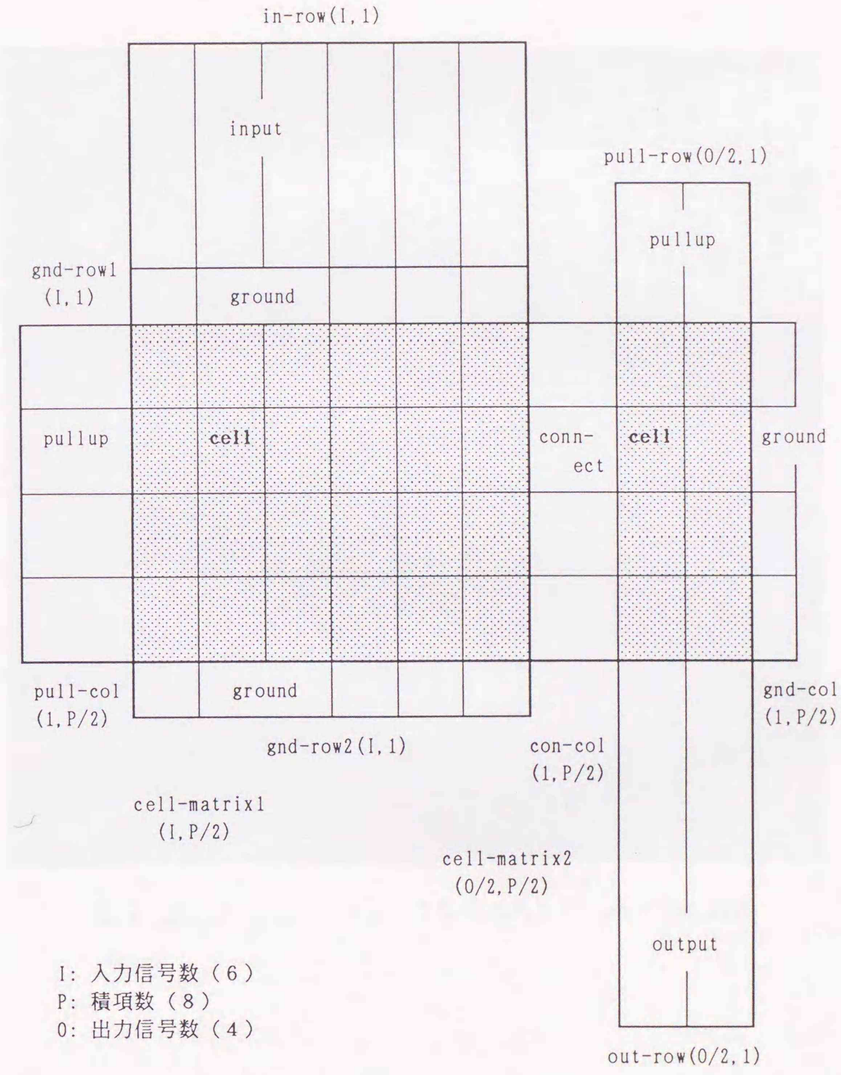
この例は非常に単純なものではあるが，このような一般的な定義を階層的に積み上げることによって，大規模な機能モジュールを自動生成するモジュール・コンパイラも容易に定義することができる．例えば，ＰＬＡは，**図Ⅱ－８**に示すように，６個の小部分（cell，input，output，pullup，ground，connect）を横方向，縦方向，行列状に並べたものを張り合わせて作られている．従って，各小部分をレイアウト・パターンと，これらを並べる個数をモジュールのパラメータとした一般的なレイアウト形式を定義しておけば，任意の大きさのモジュールをこの一般形から容易に導出することができる．ＤＰＬによるＰＬＡモジュール・コンパイラの全記述と，特定のパラメータを与えたときに生成されるレイアウト・パターンの表示画面を**付録Ⅱ－４**，**図Ⅱ－９**に示す．

ＤＰＬ自身の言語構造に加えて，ＤＰＬシステムが Lisp 処理系上に実現されていることは，モジュール・コンパイラの実現をさらに容易にしている．即ち，ＤＰＬの各機能は Lisp の関数として定義されているため，ＤＰＬによるレイアウト記述と，通常の Lisp 関数とを自由に混在して記述することができる．例えば，特定の形状のレイアウト・オブジェクトをパラメータの値（0，1，2，3）に応じて 90° 単位で回転させるには，下記のように，Lisp の条件分岐関数：cond とＤＰＬの座標変換関数：rot90，rot180 等を組み合わせて簡単に記述できる．

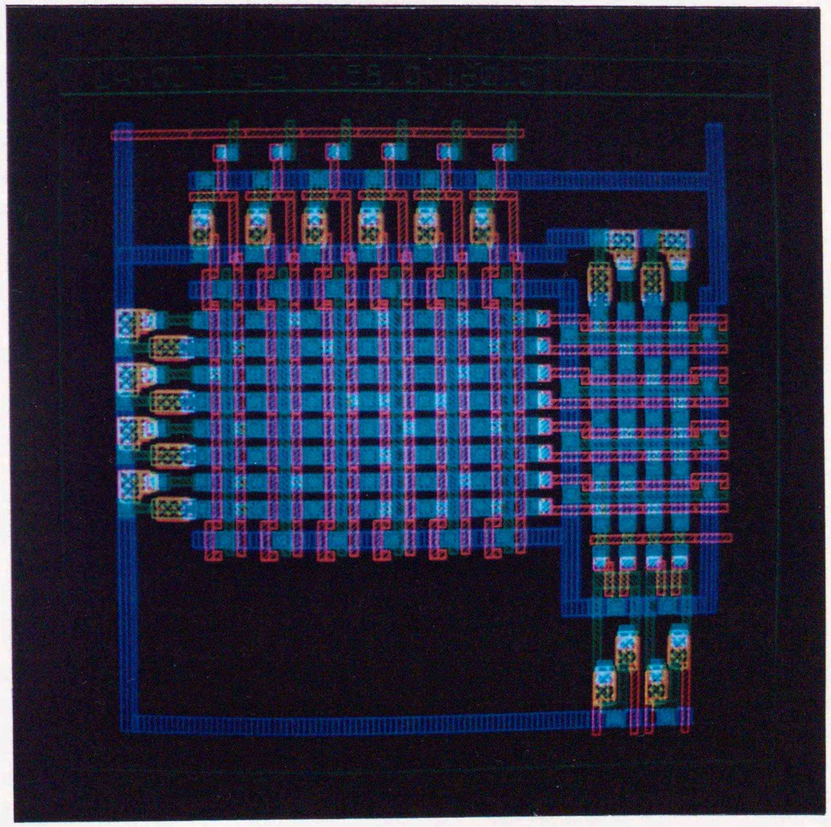
```
(defcell SAMPLE ((ROTATION 0) ... )
   . . .
   (partq FOO rectangle (layer 'poly) ... )
   (cond ((eq ROTATION 1) (rot90  (>> FOO)))
         ((eq ROTATION 2) (rot180 (>> FOO)))
         ((eq ROTATION 3) (rot270 (>> FOO))) )   . . .  )
```

また，ＰＬＡにおいては，パラーメータで指定される書込みパターン（ＡＮＤ平面とＯＲ平面に対応した０と１を要素とする２個の行列）に応じて，トランジスタを形成するための拡散層のレイアウト・パターンを適切な場所に挿入していかなければならない．このように個数が予め特定できず，不規則なレイアウト・パターンに対しても，**付録Ⅱ－４**に示すように，名前のないレイアウト・オブジェクトを生成するＤＰＬ関数：part と Lisp の繰り返し関数：do を組み合わせることにより，容易に記述することができる．

以上のようにＤＰＬは，Lisp というプログラミング言語の高度な機能を活かしたレイアウト記述言語であり，単なる繰返し構造だけのモジュールよりも数段複雑なモジュールでも，その一般形を容易に記述できるという優れた特徴をもっている．また，Lisp のインタプリタによる実行環境は，不完全な記述の段階でも支障なくレイアウト設計を進めていけるため，設計効率を大幅に向上できるレイアウト設計環境を提供している．



図Ⅱ－８　ＰＬＡのレイアウト方式



図Ⅱ－９　コンパイラによって生成されたＰＬＡの表示画面

4.2 会話型レイアウト・システム：ＩＤＥＯＮ

多くのＶＬＳＩチップで頻繁に使われる構成要素（モジュール）の一般形をレイアウト記述言語によって記述し，モジュール・コンパイラとして準備しておくことにより，任意形状のモジュールを自動生成することができる．しかし，ＶＬＳＩチップを構成するために各モジュールをチップ上に配置したり，モジュール間を配線するためには，レイアウト記述言語で記述するより，実際のレイアウト図形を見ながら会話的に行うほうが便利なことが多い．この目的のために，我々は会話型ＶＬＳＩレイアウト・システム：ＩＤＥＯＮの開発を行った．ＩＤＥＯＮは，レイアウト記述言語：ＤＰＬの処理系に，グラフィック・ディスプレイ装置と，ディスプレイ上の座標を指示するタブレット装置による会話型のレイアウト機能を付加した統合的なＶＬＳＩ設計システムである．即ち，ＤＰＬによって記述されたレイアウト・パターンをグラフィック・ディスプレイ装置に表示して検査できるとともに，この画面を見ながらモジュールの配置やモジュール間の配線を会話的に入力していくことができるシステムである．

本システムは我々の計算機環境 DEC System 2060 の MacLisp 処理系の上にインプリメントされている．（**図Ⅱ－１０**にＩＤＥＯＮの利用環境の写真を示す．）以下では，ＩＤＥＯＮの処理系とこれの会話型レイアウト機能について概説する．

（１）ＩＤＥＯＮの処理系

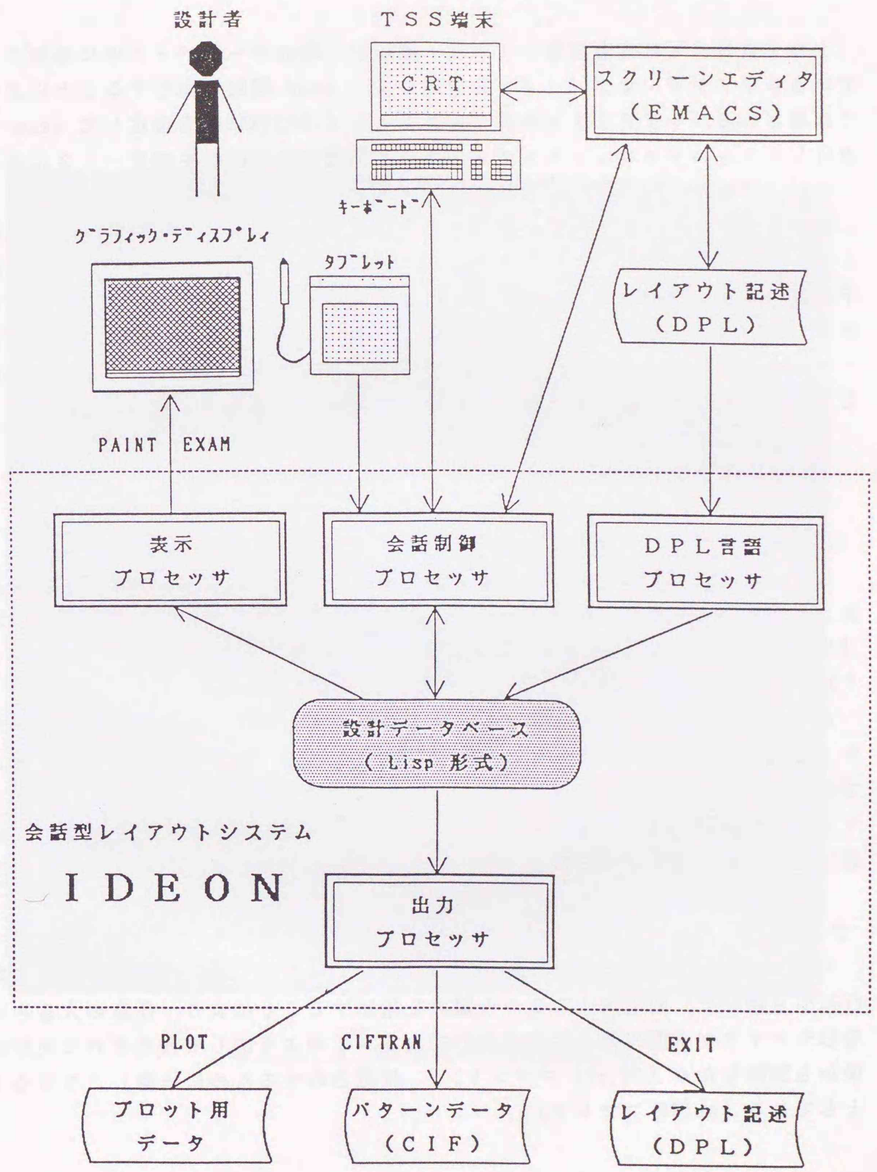
ＩＤＥＯＮの処理系は，**図Ⅱ－１１**に示すように，４個の処理プログラムと１個の設計データベースから構成されている．

ＤＰＬによるレイアウト記述は通常のスクリーン・エディタ（EMACS）等によって入力される．（EMACS とＩＤＥＯＮは各々の実行環境を保持したまま，相互に切り換えて実行することができる．）ＤＰＬ言語プロセッサは，キーボードから直接入力されるＤＰＬ記述，あるいは，ファイルとして保存されているＤＰＬ記述を入力し，内部形式に変換して設計データベースに格納する．会話制御プロセッサは，キーボードから入力されるコマンドやタブレットから指示される図形の座標情報を受け取って，設計データベースの更新や表示プロセッサの起動等の制御動作を行う．表示プロセッサは設計データベースの内容をグラフィック・ディスプレイに表示する．出力プロセッサは，設計データベースの内容を特定のデータ形式に変換し，データ・ファイルとして出力する．

ＩＤＥＯＮシステムにおける設計データベースは，ＤＰＬ記述の階層構造をそのまま表現したリスト構造の内部形式を採用しており，この中に格納されているレイアウト・オブジェクト（instance）を対象として，グラフィック・ディスプレイ装置へ表示したり（paint 関数），プロッタ装置用のデータ構造に変換して



図Ⅱ－10　IDEONの利用環境



図Ⅱ－１１　ＩＤＥＯＮのシステム構成

ファイルに出力したり（plot 関数），また，製造プロセスに渡されるマスクパターンの標準形式であるＣＩＦ（Caltech Intermediate Form）形式[27]に変換してファイルに出力する（ciftran 関数）ことができる．

（２）会話型レイアウト機能

ＩＤＥＯＮにおける会話型レイアウト機能は，設計データベース中に格納されているレイアウト・オブジェクトを指定して，exam 関数を実行することによって起動される．レイアウト・オブジェクトとして INVERTER を指定して exam を実行したときのグラフィック・ディスプレイ装置の表示内容を**図Ⅱ－１２**に示す．

このとき，ディスプレイ装置の右側に表示されているのが，会話型レイアウト用のコマンド群であり，これらをタブレット装置によって選択することにより，以下に説明する各種の会話型レイアウト機能を実行することができる．この機能を用いて入力，修正されたレイアウト情報は，ＩＤＥＯＮの動作中は設計データベースに格納されているが，ＩＤＥＯＮを終了（EXIT コマンドの実行）する時点で，文字形式のＤＰＬファイルに逆変換されて出力される．

ＩＤＥＯＮで実装されている会話型レイアウト機能の概要を以下に示す．

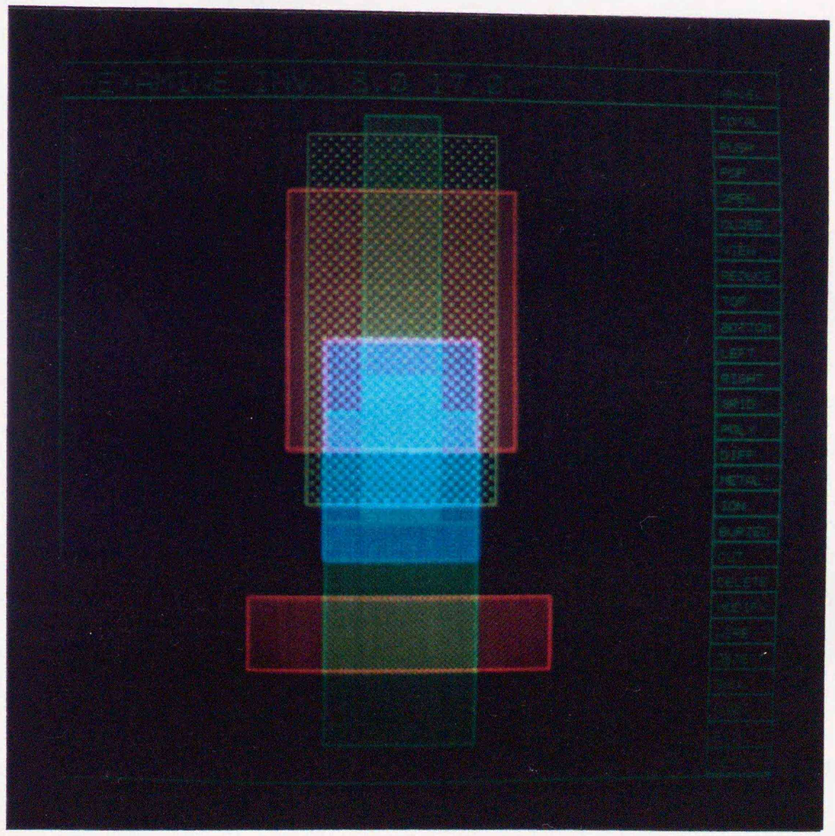
① ウィンドウと表示制御機能

ある時点で注目しているレイアウト部分だけをグラフィック・ディスプレイ装置上に表示するために，表示部分を上下左右に移動したり（TOP, BOTTOM, LEFT, RIGHT コマンド），表示されている図形の一部の領域を拡大表示（VIEW コマンド），縮小表示（REDUCE コマンド）することができる．

また，グラフィック・ディスプレイ上に複数のウィンドウを表示することができ（OPEN, CLOSE コマンド），これらに特定の部分を取り出して表示することができる．また，特定の表示状態をスタックに退避しておくことができ（PUSH コマンド），別の部分を表示したり矩形の挿入等の操作を行った後でも，以前に退避した表示状態に復帰する機能（POP コマンド）が用意されている．

② 図形の挿入と削除機能

マスクパターンの層の種類を選択するコマンド（POLY, DIFF, METAL 等）の実行に引き続いて，対角線上の２つの端点を指定することにより，任意の大きさの矩形をレイアウト図に挿入することができる．このようにして入力された矩形は，後から削除したり（DELETE コマンド），位置合わせのために移動したりすることもできる（ALIGN コマンド）．



図Ⅱ－１２　exam 起動時の表示内容

③ 配線機能

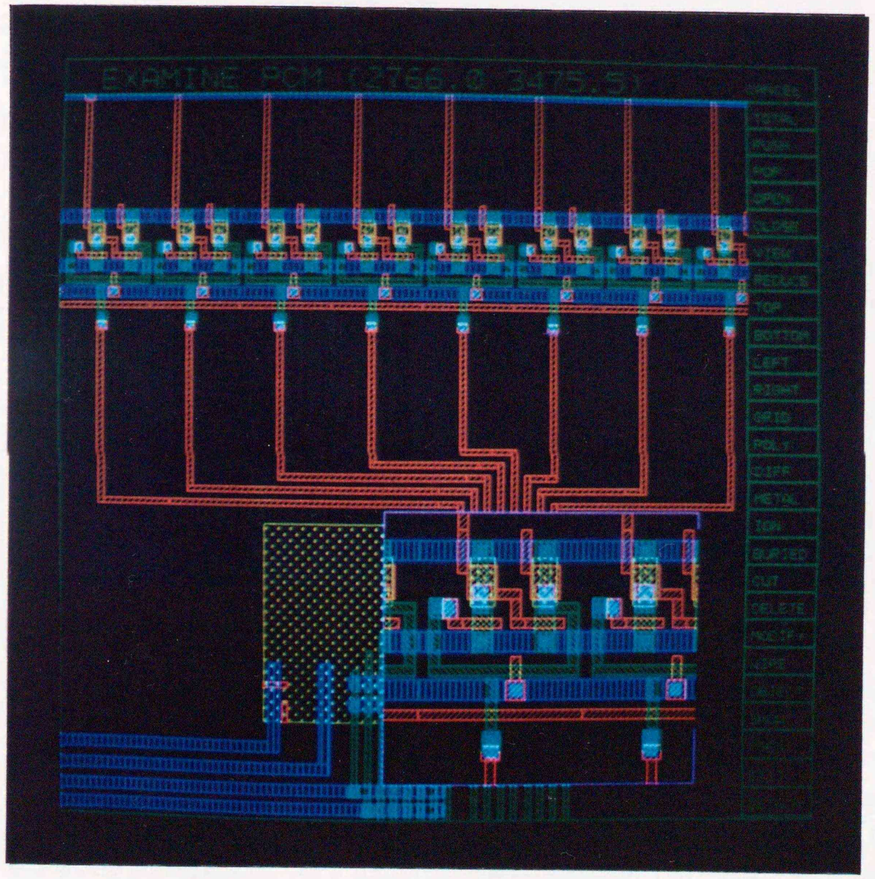
WIRE コマンドを選択するとＩＤＥＯＮは配線モードに入り，モジュール間の２点を結ぶ配線を会話的に入力することができる．このモードに入って最初に指定される矩形が配線の始点となり，その矩形の層と形状によって，配線の層と方向が決定される．これ以降に入力される点は配線の中継点となり，最後に WIRE コマンドを再度指示することによって，この直前に入力された点を終点とする１本の配線が決定される．（配線の太さは，層の種類によって決まっている最小線幅が使われる．）配線の途中で別の層を選択することにより，異なった層間を結合するための“コンタクト・パターン”が自動的に挿入される．また，配線の終点にある矩形の層が配線層と異なっている場合も，コンタクト・パターンが自動的に挿入される．

このような１本ごとの配線機能に加えて，一直線上に等間隔で並んだ端点群と，これと相対して等間隔で並んだ端点群（両者の間隔値は異なっていてもよい）とを簡単に配線する“チャネル配線機能”（CHANNEL コマンド）が用意されている．最初に一方の端点の１番目，２番目，最後の点を指定し，次にもう一方の端点の１番目，２番目，最後の点を指定することにより，これらの点間を結ぶ複数の配線を自動的に生成することができる．この機能はデータバスのようなまとまった配線を行うのに非常に有効な機能である．（チャネル配線機能の使用例を**図Ⅱ－１３**に示す．）

④ その他の機能

以上の機能の他にＩＤＥＯＮでは，特定のレイアウト・オブジェクトを“シンボル化”するためのＤＰＬ関数：vgenq が用意されている．この機能が指定されたオブジェクトは，オブジェクト全体の境界を表す矩形と，外部から接続される可能性のある矩形（境界の矩形に接したもの）だけが表示される．これは表示の見やすさと内部処理の高速化を考慮して組み込まれた機能であり，大規模なレイアウト設計を行うときには非常に有効な機能である．このシンボル化の機能はＩＤＥＯＮの表示中の任意の時点で会話的にも指定することができる．即ち，VGEN コマンドによって特定のオブジェクトを指定することにより，このオブジェクトはシンボル化され，また，すでにシンボル化されているオブジェクトは SHOW コマンドによってシンボル化を解除することができる．（**図Ⅱ－１３**において，下のモジュールはシンボル化された状態であり，上のモジュールはこれが解除され，モジュール内部の全レイアウト・パターンが表示されている．）

以上のようにＩＤＥＯＮは，比較的限定されたレイアウト機能だけしか提供されておらず，ＤＰＬによって記述されたレイアウト・パターンの配置と配線にかかわる機能だけに特化したレイアウト設計支援システムになっている．しかし，レイアウト記述言語：ＤＰＬによる高度なレイアウト・パターンの記述能力との組合せによって，少量の記述と簡単な操作で大規模なＶＬＳＩチップのレイアウト設計にも適用できる効果的なＶＬＳＩ設計支援システムと考えられる．



図Ⅱ－１３　チャネル配線機能の使用例

5．言語解析チップの設計と評価

5．1　モジュール設計

言語語解析チップのレイアウト設計には，ＰＬＡ，拡張型ＰＬＡ，ＦＩＦＯメモリ，スタック等の構成要素が必要であり，これらの要素を自動生成するモジュール・コンパイラをレイアウト記述言語：ＤＰＬによって試作した．**表Ⅱ－5**に，これらのモジュール・コンパイラの諸元，即ち，モジュールの機能，設定されたパラメータ，生成されるモジュールの大きさ，これに含まれるトランジスタ数等を示す．これらのモジュール・コンパイラのレイアウト記述には，約 3,000 行のＤＰＬソース・ステートメントを要している．

5．2　チップ設計

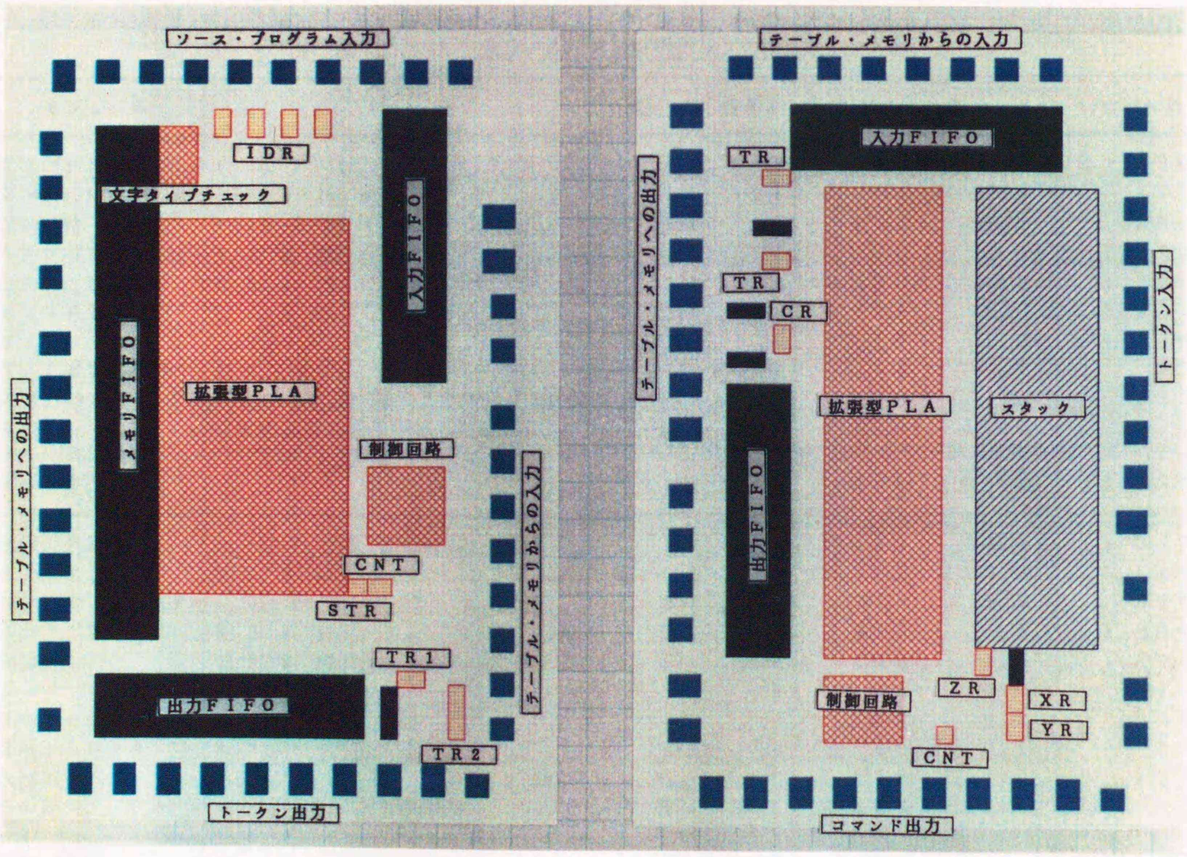
言語解析チップの最終的なチップ設計を行うために，モジュール・コンパイラに適当なパラメータ値をセットし，特定の仕様の構成要素を実体化する．この後，これらの構成要素の形状と入出力端子の位置を考慮して，チップ全体の面積ができるだけ小さくなり，かつ，配線が短くなるように各構成要素の配置（フロアプラン）を決定しなければならない．前章で示した会話型レイアウト・システム：ＩＤＥＯＮは，構成要素の最適配置を自動的に行う機能を有していないため，設計者がグラフィック・ディスプレイの画面を見ながら，ＤＰＬ記述のパラメータを調整し，試行錯誤的に配置を決定する必要がある．但し，ＩＤＥＯＮのシンボル化機能（vgenq 関数）を利用することにより，各構成要素の境界の形状とこれに接して配置されているレイアウト・パターンだけを取り扱うため，設計データベースを検索する時間やディスプレイ装置への表示時間を大幅に短縮でき，この配置作業を容易に行うことができる．

チップの構成要素の最終的な配置が決定した後，各構成要素の端子間の配線や電源，グラウンド等の配線をＩＤＥＯＮの配線機能（WIRE コマンド）を利用して入力する．この操作においても，ＩＤＥＯＮのシンボル化機能や，前章で示したウィンドウ機能，表示制御機能等を有効に利用することにより，比較的容易に配線を行うことができる．このようにして生成された言語解析チップの最終的なマスクパターン情報は，ＩＤＥＯＮの CIFTRAN 関数 によって，製造工程への標準的なデータ表現形式であるＣＩＦ形式に変換され，磁気テープに出力される．

ここで設計された字句解析チップと構文解析チップのフロアプランを**図Ⅱ－14**に示す．また，これらのチップにおける各構成要素の形状，面積，トランジスタ等の諸元，及び，チップ全体の諸元を**表Ⅱ－6**に示す．

表Ⅱ－5　モジュール・コンパイラの諸元

モジュール名	機　能	パラメータ	幅（λ）	長さ（λ）	トランジスタ数
ＰＬＡ	積和標準形で表された論理式を実現する論理要素	I: 入力数，P: 積項数 O: 出力数 T: 書込みテーブル	16I + 80 + 36	8P + 123	5I + P + 60 + I*P + O*P
拡張型ＰＬＡ	“抑止機能”と “otherwise”機能を付加したＰＬＡ	（同上） S: 抑止項数 W: otherwise 項数	16I + 80 + 77 但し，S=0 ならば 16I + 80 + 51	8(P-S-W) + 123 +15S + 9W	5I + P + 60 + I*P + O*P + 3S + W
ＦＩＦＯメモリ	先入れ先出し方式の記憶回路	B: ビット長 W: 語数	28B + 70	35W + 138	6B*W + 10W + 38
スタック （折り返し型）	先入れ後出し方式の記憶回路	B: ビット長 W: 語数	42B + 236	32W + 7B + 15	8B*W + 32W + 2
ラッチ	データを一時的に記憶する回路	B: ビット長	16B	76	6B
2→1 セレクタ	2個の入力の何れかを選択して回路	B: ビット長	12B + 77	7B + 11	2B + 8
3→1 セレクタ	3個の入力の何れかを選択する回路	B: ビット長	18B + 99	7B + 18	3B + 14
入力パド	チップの外部から信号を取り込む回路	－	106	106	1
出力パド	チップの外部に信号を出力する回路	－	106	145	10



図Ⅱ－14　言語解析チップのフロアプラン

表Ⅱ－6　言語解析チップの諸元

	モジュール	特　性	幅　λ	長さλ	面積 Kλ²	割合　%	ﾄﾗﾝｼﾞｽﾀ数	割合　%
字句解析チップ	拡張型ＰＬＡ	I: 40, P: 194, O: 21, S: 7, W: 68	885	1,862	1,648	21.7	12,460	56.5
	文字ﾀｲﾌﾟﾁｪｯｸＰＬＡ	I: 8, P: 29, O: 2	180	355	64	0.8	371	1.7
	制御回路（ＰＬＡ）	I: 13, P: 30, O: 14	356	363	129	1.7	989	4.5
	入力ＦＩＦＯ	B: 8, W: 32	294	1,258	370	4.9	1,894	8.6
	出力ＦＩＦＯ	B: 8, W: 32	294	1,258	370	4.9	1,894	8.6
	メモリＦＩＦＯ	B: 8, W: 64	294	2,378	699	9.2	3,750	17.0
	ラッチ	合計 69 ビット	1,104	76	84	1.1	414	1.9
	3→1 セレクタ	B: 8	243	74	18	0.2	38	0.2
	入出力パド	入力パド： 25 個，出力パド： 20 個	—	—	588	7.8	225	1.0
	その他	（配線領域，未使用領域）	—	—	3,614	47.7	0	0.0
	チップ全体		2,219	3,403	7,584	100.0	22,035	100.0
構文解析チップ	拡張型ＰＬＡ	I: 15, P: 249, O: 29, S: 0, W: 66	523	2,181	1,141	14.9	11,520	50.3
	スタック	B: 8, W: 64	572	2,119	1,212	15.8	6,146	26.8
	制御回路（ＰＬＡ）	I: 11, P: 30, O: 12	308	363	112	1.5	847	3.7
	入力ＦＩＦＯ	B: 8, W: 32	294	1,258	370	4.8	1,894	8.3
	出力ＦＩＦＯ	B: 8, W: 32	294	1,258	370	4.8	1,894	8.3
	ラッチ	合計 52 ビット	832	76	63	0.8	312	1.4
	3→1 セレクタ	B: 8	243	74	18	0.2	38	0.2
	2→1 セレクタ	B: 7	161	60	10	0.1	22	0.1
	入出力パド	入力パド： 25 個，出力パド： 20 個	—	—	588	7.7	225	1.0
	その他	（配線領域，未使用領域）	—	—	3,789	49.4	0	0.0
	チップ全体		2,212	3.469	7,673	100.0	22,898	100.0

5.3 評価

（1）レイアウト記述言語とモジュール・コンパイラの評価

レイアウト記述言語：ＤＰＬは高級言語のプログラマ，特に Lisp の記述に慣れたプログラマとっては，ソフトウェア開発と殆ど違和感なくレイアウト設計を行え，非常に効果的である．ＤＰＬでは１個の矩形の記述に１～２行の Lisp コードを必要とするが，オブジェクト指向のプログラミング・パラダイムにより，比較的少ない記述量で大規模なモジュールのレイアウト設計が可能である．今回，試作した９個のモジュール・コンパイラは，全体でも約 3,000 行の Lisp コードで記述されている．１個の特定の形状のモジュールを記述するためには，もっと少ない記述量で設計することは可能であるが，パラメータの指定による任意形状のモジュールを自動生成できることを考慮すると，非常に少ない記述量と考えられる．例えば，今回設計した両チップとも，約 22,000 個のトランジスタを含んでいるが，実際に記述したトランジスタの数は 200 個以下であり，メモリ・アーキテクチャであることも影響して，チップ上に存在するトランジスタ数と実際に設計したトランジスタ数の比である“規則性係数”（チップ上に存在するトランジスタ数／実際に設計したトランジスタ数）は 100 以上の非常に高い値を示している．

モジュール・コンパイラの開発において，ラッチ等のような単純な繰り返し構造のモジュール・コンパイラは比較的容易に設計できるが，繰り返し構造の不規則なモジュール・コンパイラのプログラミングは非常に繁雑である．ＤＰＬの場合は Lisp コードと共存できるという，非常に強力な特徴があるものの，種々の場合分けをした記述は繁雑で，記述量も多くなりがちである．

また，特定のモジュール・コンパイラでは，繰り返し数が多くなると１個のトランジスタで駆動する負荷が極端に大きくなってしまう可能性がある．ＭＯＳ回路は電荷を供給することによって動作しているため，負荷の大きさと充電時間は比例する．従って，このような構成要素では，スイッチング時間が長くなり，想定していたクロック時間で動作が完了できない場合が発生する．このため，繰り返し数が負荷の大きさに影響するような場合は，駆動回路を大きくしたり，途中に波形整形用の増幅回路を挿入する等の考慮が必要になる．但し，このようなモジュール・コンパイラはパラメータの値に応じて複雑な処理を挿入する必要があり，レイアウト設計は繁雑なものになるという問題がある．

（2）会話型レイアウト・システムの評価

会話形レイアウト・システム：ＩＤＥＯＮの図形表示機能は，モジュール・コンパイラとして実現するような規模のモジュールを設計するためには十分な機能を備えており，非常に効果的である．しかし，複数のモジュールからなるチップ

全体を表示するような場合には，表示に数分から 10 分以上の時間がかかってしまうことがあり，表示速度の点で問題がある．これは，通常のレイアウト設計用ＣＡＤ装置のように計算機本体とグラフィック・ディスプレイ装置をバス配線で直結せず，通信回線（RS-232C）によって低速の接続（9600 BPS）をしたことに起因するものと考えられる．このような図形表示の高速化には，ＩＤＥＯＮのシンボル化関数：VGENQ が非常に効果的であった．即ち，特定のモジュール・コンパイラが一旦できてしまえば，内部を参照する必要は殆ど無く，モジュールの配置に関係する外枠の矩形と配線に関係する周辺部のレイアウト・パターンだけを表示することによりチップ設計を迅速に行える．

チップ全体のレイアウト設計（フロアプラン）は，モジュールの入出力端子の位置によって，なるべく配線が長くならないようなモジュール間の相対的な位置関係を決め，モジュール間を通る配線数からモジュール間の距離を概算で求め，この値をＤＰＬ記述のパラメータに設定することによって行われる．従って，この操作はＤＰＬによる記述とグラフィック・ディスプレイへの表示による確認を繰り返す試行錯誤的な操作であり，比較的時間のかかる作業である．また，配線の“抜け”が見つかったとき，新しい配線を通すためにモジュール間の距離を変更する操作や，既設の配線をやり直す操作が必要であり，厳密なコンパクト化が難しい等の問題もある．

配線作業そのものはＩＤＥＯＮのマルチ・ウィンドウ機能を使って比較的容易に行える．しかし，１個のチップに含まれる配線数はかなり多く（今回設計した言語解析チップでは各チップとも約 300 本の配線が必要であった），神経を集中した作業を長時間継続する必要があるため，コンパクト化の要求からも，計算機による自動配線機能が強く望まれる．

（3）設計したチップの評価

本言語解析チップは，トランジスタの最小線幅が３μｍのｎＭＯＳ技術を想定して設計された．この時，設計上の長さの基本パラメータである“λ”は，トランジスタの最小線幅を２λとしていることから，λ＝ 1.5 μｍとなる．従って，今回設計した字句解析チップの形状は 3.3 ｍｍ× 5.1 ｍｍ（17.0 ｍｍ^2），構文解析チップの形状は 3.3 ｍｍ× 5.2 ｍｍ（17.3 ｍｍ^2）と見積られる．しかし，現在ではトランジスタの最小線幅が１μｍ程度に縮小し，デバイス技術もＣＭＯＳ技術に主流が変わってきている．ＣＭＯＳ技術の場合には，回路の実現方式が異なるためｎＭＯＳの場合よりもチップの形状が大きくなる傾向があるが，それを考慮しても，長さで 1/2，面積で 1/4 程度には 十分に小型化できると考えられる．

一方，現在のパッケージング技術では，１ｃｍ角程度のチップはごく普通に利用されていることから，これらの言語解析チップの両方を１チップ化できるばかりでなく，テーブル・メモリや解析処理に引き続く“コード生成処理”等をも含

めて，Pascal コンパイラの全体を１個のＶＬＳＩチップとして実現することも十分に可能であると考えられる．

今回の実験では言語解析チップの設計までを行い，試作には至っていないため，適用されるプロセス技術と２章で示したシミュレーション結果を前提として，これらのチップの動作速度について考察する．ハードウェア記述言語によるシミュレーション実験では，チップが 10 ＭＨｚの基本クロックで動作できるとすると，言語解析チップの性能はソフトウェアのコンパイラに比べて 300 倍以上も 高速であると推定される．今回のチップ設計で想定した３μｍのｎＭＯＳプロセスでは，インバータの遅延が３ｎ秒程度，最新の１μｍのＣＭＯＳプロセスでは１ｎ秒程度で動作することが回路シミュレーションによって保証されている．従って，30 段，あるいは 100 段の論理段数を組み入れることのできる 10 ＭＨｚの基本クロック（１クロックの幅は 100 ｎ秒） は通常の論理回路を構成する上で十分可能な動作周波数であると考えられる．但し，今回の言語解析チップでは，ＰＬＡは積項数が 200 項程度とかなり多いため，ＰＬＡの各項配線を起動する駆動トランジスタの能力が不足し，スイッチング時間を大きくしてしまう可能性がある．接続される負荷の大きさによる駆動能力と動作時間の関係を回路シミュレーションによって詳細に評価する必要があるが，10 ＭＨｚの基本クロックで動作させるためには，ＰＬＡ等の駆動回路を強化するなどの対策が必要になる可能性もある．当然ながら，駆動回路の強化はチップ・サイズの増大をも意味する．従って，実際のチップを試作するときには，適用するプロセス技術と目標とする動作速度やチップ面積の観点から，最適な方式について再度詳細な検討を行う必要がある．

これらの言語解析チップの設計に要した設計労力は，９個のモジュール・コンパイラの設計に約４人月程度，両チップの配置・配線に各々約１人月程度であり，合計６人月程度の設計労力で２個の言語解析チップの設計を完了できた．これは，通常，多くのフルカスタムＶＬＳＩの設計に数人年の設計労力がかかること，また，今回のチップ設計が，これまで半導体の設計に携わったことのなかった計算機アーキテクトによって行われたことなどから判断して，非常に少ない設計労力で設計できたと考えられる．

モジュール・コンパイラの開発に比べて，チップ設計における配置・配線処理は試行錯誤的な要素が強く，また，単純な繰り返し作業に多くの時間が費やされているため，これらの処理の自動化が強く望まれる．実際，通常の信号配線の自動化は幾つかの配線アルゴリズムが提案されているので，近い将来，これをＩＤＥＯＮに実装することも可能になると思われる．しかし，任意形状の複数のモジュールをできるだけコンパクトに配置するアルゴリズムは人手によるのに比べて十分な最適化が期待できないのが現状である．また，電源やグラウンドの配線は大電流が流れるために配線の幅を慎重に決定する必要があり，これも自動化の難しい処理である．

６．おわりに

本第Ⅱ部では，大規模な応用システムの具体例として，プログラミング言語 Pascal のコンパイラを取り上げ，この中の言語解析処理を専用に実行するＶＬＳＩチップに関する研究について述べた．

これらの言語解析用ＶＬＳＩチップは，プログラミング言語の解析処理用に拡張したＰＬＡと，スタックやＦＩＦＯメモリ等のような記憶要素を主体とする比較的簡潔なメモリ・アーキテクチャによってＶＬＳＩ化が可能であり，実行速度の面でも，汎用計算機のソフトウェアによる言語解析に比べて 300 倍 以上も高速化できることがシミュレーション結果から示された．また，レイアウト記述用高級言語：ＤＰＬを用いたモジュール・コンパイラと，会話型のレイアウトシステム：ＩＤＥＯＮの利用によって，これらのチップのレイアウト設計を試みた結果，フルカスタムＶＬＳＩチップとしては非常に少ない設計労力でチップ設計を完了できた．また，これらのチップのハードウェア規模は，両チップ合わせても十分に１個のパッケージに格納できる程度の大きさであることが示された．

即ち，プログラミング言語を対象とする言語解析チップに対して全面的にメモリ・アーキテクチャを適用し，その詳細な実現方式，予測される性能，及び，このチップの設計に適したＶＬＳＩ設計支援ツールの試作と実際のチップの設計を行った結果，言語解析チップに対するメモリ・アーキテクチャの有効性を明らかにでき，また，高性能な言語解析チップを実現するための多くの設計実績を蓄積することができた．

アーキテクチャ設計に関する今後の課題としては，様々な専用アーキテクチャでの性能やハードウェア規模を詳細に評価できるアーキテクチャ評価ツールの開発や，アーキテクチャ・レベルでの設計結果を下位のチップ設計ツールに接続する標準インタフェースの整備等が必要である．また，様々な応用システムに適した専用ＶＬＳＩアーキテクチャの具体例を数多く開発し，アーキテクチャ設計知識を蓄積していくことが重要である．一方，設計支援システム側の課題としては，できるだけ多くの構成要素をモジュール・コンパイラとして利用できる環境を整えること，また，任意形状の複数のモジュールを自動配置・配線するチップ設計の自動化ツールを早急に開発していくことが必要である．

［参考文献Ⅱ］

(1) Hopcroft, John E. & Jeffrey D. Ullman: Formal Languages and their Relation to Automata, Addison-Wesley Publishing Co., Reading (1969).（野崎昭弘他訳：言語理論とオートマトン，サイエンス社，東京 (1971).）
(2) Aho, Alfred V. & Jeffery D. Ullman: The Theory of Parsing, Translation, and Compiling, Vol.1: Parsing, Prentice-Hall, Englewood Cliffs (1972).
(3) 井上謙蔵：コンパイラ・コンパイラ，産業図書，東京 (1970).
(4) Pyster, Arthur B.: Compiler Design and Construction, Van Nostrand Reinhold (1980).（松尾正信訳：コンパイラの設計と構築，近代科学社，東京 (1983).）
(5) Hartmann, Alfred C.: Software or Silicon? The Designer's Option, Proc. of the IEEE, Vol.74, No.6 (1986).
(6) Floyd, Robert W. & Jeffrey D. Ullman: The Compilation of Regular Expressions into Integrated Circuits, Journal of the ACM, Vol.29, No.3 (1982).
(7) 板野肯三他：パイプライン型字句解析プロセッサの設計と実現，情報処理学会論文誌，Vol.28, No.1 (1987).
(8) Chiang, Y.T. & K.S. Fu: Parallel Parsing Algorithms and VLSI Implementations for Syntactic Pattern Recongnition, IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol.PAMI-6, No.3 (1984).
(9) Sanamard, Mohammad A., et al.: A Hardware Syntactic Analysis Processor, IEEE Micro, August 1987 (1987).
(10) Andre, F., et al.: KENSUR: An Architecture Oriented Towards Programming Language Translation, Proc. of the 7th Annual Symposium on Computer Architecture (1980).
(11) Muroga, Saburo: ROMs in Logic Design and Task-Realization Algorithms, in VLSI System Design, John Wiley & Sons, Inc., New York (1982).（渡邊誠・菅野卓雄監修：ＶＬＳＩシステム設計，啓学出版，東京 (1984).）
(12) Shaw, David E. & Theodore M. Sabety: Active Memory Chips: A Brief Survey and Case Study, Circuits Systems Signal Process, Vol.6, No.2 (1987).
(13) Fusaoka, Akira & Masaharu Hirayama: Compiler Chip: A Hardware Implementation of Compiler, Proc. of the Symposium on Architectural Support for Programming Language and Operating Systems (1982).
(14) Seo, Kazuo, Masaharu Hirayama, & Akira Fusaoka: Design and Evaluation of Parsing Chip, Proc. of the IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration (1983).
(15) 平山正治：プログラミング言語解析用ＶＬＳＩチップの設計と評価，情報処理学会論文誌，Vol.29, No.4 (1988).
(16) Nori, K.V., et al.: The PASCAL <P> Compiler: Implementation Notes, in Pascal - The Language and Its Implementation, Barron, D.W., (Ed), Wiley, Chichester (1981).
(17) Pemberton, Steven & Martin C. Daniels: Pascal Implementation: The P4 Compiler, Ellis Horwood, Chichester (1982).
(18) Greibach, S.A.: Formal Parsing Systems, Comm. of the ACM, Vol.7, No.8 (1964).
(19) Lesk, M.E. & E. Schmidt: Lex - A Lexical Analyzer Generator, Bell Lab. Computing Science Tech. Report No.39 (1975).
(20) Johnson, Stephen C.: Yacc: Yet Another Compiler Compiler, Bell Lab. Computing Science Tech. Report No.32 (1975).
(21) Barbacci, Mario R., et al.: The ISPS Computer Description Language, Carnegii-Mellon Univ. Tech. Report No.CMU-CS-79-137 (1977).
(22) Wirth, Nicklaus: Algorithms + Data Structures = Programs, Prentice-Hall, Englewood Cliffs (1976).
(23) Batali, John & Anne Hartheimer: The Design Procedure Language Manual, MIT A.I. Memo. No.598 (1980).
(24) van Berkel, Kees: VLSI Layout Design Requires a New Language, Proc. of the IEEE International Copnference on Computer Design: VLSI in Computers (1983).
(25) Burich, Misha R.: Design of Module Generators and Silicon Compilers, in Silicon Compilation, Edited by Daniel D. Gajski, Addison-Wesley Publishing Co., Reading (1988).
(26) Law, Hung-fai S., et al.: An Intelligent Composition Tool for Regular and Semiregular VLSI Structures, Edited by Daniel D. Gajski, Addison-Wesley Publishing Co., Reading (1988).
(27) Hon, Robert H. & Carlo H. Sequin: A Guide to LSI Implementation, 2nd Edition, XEROX Palo Alto Research Center, No.SSL-79-7 (1980).

付録Ⅱ－1　字句解析チップにおける状態遷移表

ＳＨ：シフト数,　　ＣＬ：clear buffer,
ＰＵ：push down buffer,　　ＴＭ：terminate
ＯＷ：otherwise 機能,　　⊥　：抑止，otherwise の対象項

	PLA入力		PLA出力					拡張機能	
	状態	タイプ　文字列	状態	コード	SH	CL	PU　TM	抑止	OW
基本状態	1	□□□□	1		4			○	
	1	□□□？	1		3			○	
	1	□□？？	1		2			○	
	1	□？？？	1		1			⊥	
	1	：＝？？	1	4F	2		1	○	
	1	：？？？	1	53	1		1	⊥	
	1	．．？？	1	53	2		1	○	
	1	．？？？	1	50	1		1	⊥	
	1	＜＝？？	1	49	2		1	○	
	1	＜＞？？	1	46	2		1	○	
	1	＜？？？	1	47	1		1	⊥	
	1	＞＝？？	1	4A	2		1	○	
	1	＞？？？	1	48	1		1	⊥	
	1	（＊？？	2		2			○	
	1	（？？？	1	4B	1		1	⊥	
	1	’？？？	3		1	1			
	1	＋？？？	1	41	1		1		
	1	－？？？	1	42	1		1		
	1	＊？？？	1	43	1		1		
	1	／？？？	1	44	1		1		
	1	＝？？？	1	45	1		1		
	1	）？？？	1	4C	1		1		
	1	［？？？	1	4D	1		1		
	1	］？？？	1	4E	1		1		
	1	，？？？	1	51	1		1		
	1	；？？？	1	52	1		1		
	1	＾？？？	1	54	1		1		
	1	数字	10		1	1	1		
	1	ＩＦ？？	21		2	1	1		○
	1	ＤＯ？？	22		2	1	1		○
	1	ＯＦ？？	23		2	1	1		○
	1	ＴＯ？？	24		2	1	1		○
	1	ＩＮ？？	25		2	1	1		○
	1	ＯＲ？？	26		2	1	1		○
	1	ＥＮＤ？	27		3	1	1		○
	1	ＦＯＲ？	28		3	1	1		○
	1	ＶＡＲ？	29		3	1	1		○

基本状態	1	ＤＩＶ？	30		3	1	1		○
	1	ＭＯＤ？	31		3	1	1		○
	1	ＳＥＴ？	32		3	1	1		○
	1	ＡＮＤ？	33		3	1	1		○
	1	ＮＯＴ？	34		3	1	1		○
	1	ＮＩＬ？	35		3	1	1		○
	1	ＴＨＥＮ	36		4	1	1		○
	1	ＥＬＳＥ	37		4	1	1		○
	1	ＷＩＴＨ	38		4	1	1		○
	1	ＧＯＴＯ	39		4	1	1		○
	1	ＣＡＳＥ	40		4	1	1		○
	1	ＴＹＰＥ	41		4	1	1		○
	1	ＦＩＬＥ	42		4	1	1		○
	1	ＢＥＧＩ	50		4	1	1		○
	1	ＵＮＴＩ	52		4	1	1		○
	1	ＷＨＩＬ	54		4	1	1		○
	1	ＡＲＲＡ	56		4	1	1		○
	1	ＣＯＮＳ	58		4	1	1		○
	1	ＬＡＢＥ	60		4	1	1		○
	1	ＲＥＰＥ	62		4	1	1		○
	1	ＲＥＣＯ	64		4	1	1		○
	1	ＤＯＷＮ	66		4	1	1		○
	1	ＰＡＣＫ	68		4	1	1		○
	1	ＦＯＲＷ	70		4	1	1		○
	1	ＰＲＯＧ	72		4	1	1		○
	1	ＦＵＮＣ	74		4	1	1		○
	1	ＰＲＯＣ	76		4	1	1		○
	1	英字	20		1	1	1		⊥
コメント	2	＊）？？	1		2			○	○
	2	？？＊）	1		4			⊥	○
	2	？＊）？	1		3			○	○
	2	？？？＊	2		3			⊥	○
	2	？？？？	2		4				⊥
文字列	3	’’？？	4		1			○	○
	3	’？？？	1	F3	1		1	⊥	○
	3	？？？？	3		1		1		⊥
	4	？？？？	3		1		1		
数字	10	数字	10		1		1		○
	10	Ｅ＋？？	13		2		1	○	○
	10	Ｅ－？？	13		2		1	○	○
	10	Ｅ？？？	13		1		1	⊥	○
	10	．．？？	1	F1	0		1	○	○
	10	．？？？	11		1		1	⊥	○
	10	？？？？	1	F1	0		1		⊥

	11	数字		12		1		1										
	12		E+??	13		2		1		○	○							
小	12		E−??	13		2		1		○	○							
数	12		E????	13		1		1		⊥	○							
	12	数字		12		1		1			○							
	12		????	1	F2	0			1		⊥							
指	13	数字		14		1		1										
数	14	数字		14		1		1			○							
	14	?		1	F2	0			1		⊥							
名	20	記号		1	80	0			1									
前	20	英数字		20		1		1										
	21	記号		1	01	0			1									
	21	英数字		20		1		1										
予	22	記号		1	02	0			1									
	22	英数字		20		1		1										
約	23	記号		1	03	0			1									
	23	英数字		20		1		1										
語	24	記号		1	04	0			1									
	24	英数字		20		1		1										
	25	記号		1	05	0			1									
	25	英数字		20		1		1										
	26	記号		1	06	0			1									
	26	英数字		20		1		1										
	27	記号		1	07	0			1									
	27	英数字		20		1		1										
	28	記号		1	08	0			1									
	28	英数字		20		1		1										
	29	記号		1	09	0			1									
	29	英数字		20		1		1										
	30	記号		1	0A	0			1									
	30	英数字		20		1		1										
	31	記号		1	0B	0			1									
	31	英数字		20		1		1										
	32	記号		1	0C	0			1									
	32	英数字		20		1		1										
	33	記号		1	0D	0			1									
	33	英数字		20		1		1										
	34	記号		1	0E	0			1									
	34	英数字		20		1		1										
	35	記号		1	24	0			1									
	35	英数字		20		1		1										
	36	記号		1	0F	0			1									
	36	英数字		20		1		1										
	37	記号		1	10	0			1									
	37	英数字		20		1		1										

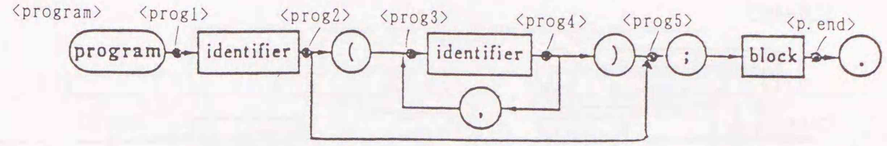
	38	記号		1	11	0			1									
	38	英数字		20		1		1										
	39	記号		1	12	0			1									
	39	英数字		20		1		1										
	40	記号		1	13	0			1									
	40	英数字		20		1		1										
	41	記号		1	14	0			1									
	41	英数字		20		1		1										
	42	記号		1	15	0			1									
	42	英数字		20		1		1										
	50		N???	51		1		1			○							
	50		????	20		0					⊥							
	51	記号		1	16	0			1									
	51	英数字		20		1		1										
	52		L???	53		1		1			○							
	52		????	20		0					⊥							
予	53	記号		1	17	0			1									
	53	英数字		20		1		1										
約	54		E???	55		1		1			○							
	54		????	20		0					⊥							
語	55	記号		1	18	0			1									
	55	英数字		20		1		1										
	56		Y???	57		1		1			○							
	56		????	20		0					⊥							
	57	記号		1	19	0			1									
	57	英数字		20		1		1										
	58		T???	59		1		1			○							
	58		????	20		0					⊥							
	59	記号		1	1A	0			1									
	59	英数字		20		1		1										
	60		L???	61		1		1			○							
	60		????	20		0					⊥							
	61	記号		1	1B	0			1									
	61	英数字		20		1		1										
	62		AT??	63		2		1			○							
	62		????	20		0					⊥							
	63	記号		1	1C	0			1									
	63	英数字		20		1		1										
	64		RD??	65		2		1			○							
	64		????	20		0					⊥							
	65	記号		1	1D	0			1									
	65	英数字		20		1		1										
	66		TO??	67		2		1			○							
	66		????	20		0					⊥							
	67	記号		1	1E	0			1									
	67	英数字		20		1		1										
	68		ED??	69		2		1			○							
	68		????	20		0					⊥							

6 9	記号		1	1 F	0		1				
6 9	英数字		2 0		1	1					
7 0		A R D ?	7 1		3	1				○	
7 0		? ? ? ?	2 0		0					⊥	
7 1	記号		1	2 0	0		1				
7 1	英数字		2 0		1	1					
7 2		R A M ?	7 3		3	1				○	
7 2		? ? ? ?	2 0		0					⊥	
7 3	記号		1	2 1	0		1				
7 3	英数字		2 0		1	1					
7 4		T I O N	7 5		4	1				○	
7 4		? ? ? ?	2 0		0					⊥	
7 5	記号		1	2 2	0		1				
7 5	英数字		2 0		1	1					
7 6		E D U R	7 7		4	1				○	
7 6		? ? ? ?	2 0		0					⊥	
7 7		E ? ? ?	7 8		1	1				○	
7 7		? ? ? ?	2 0		0					⊥	
7 8	記号		1	2 3	0		1				
7 8	英数字		2 0		1	1					

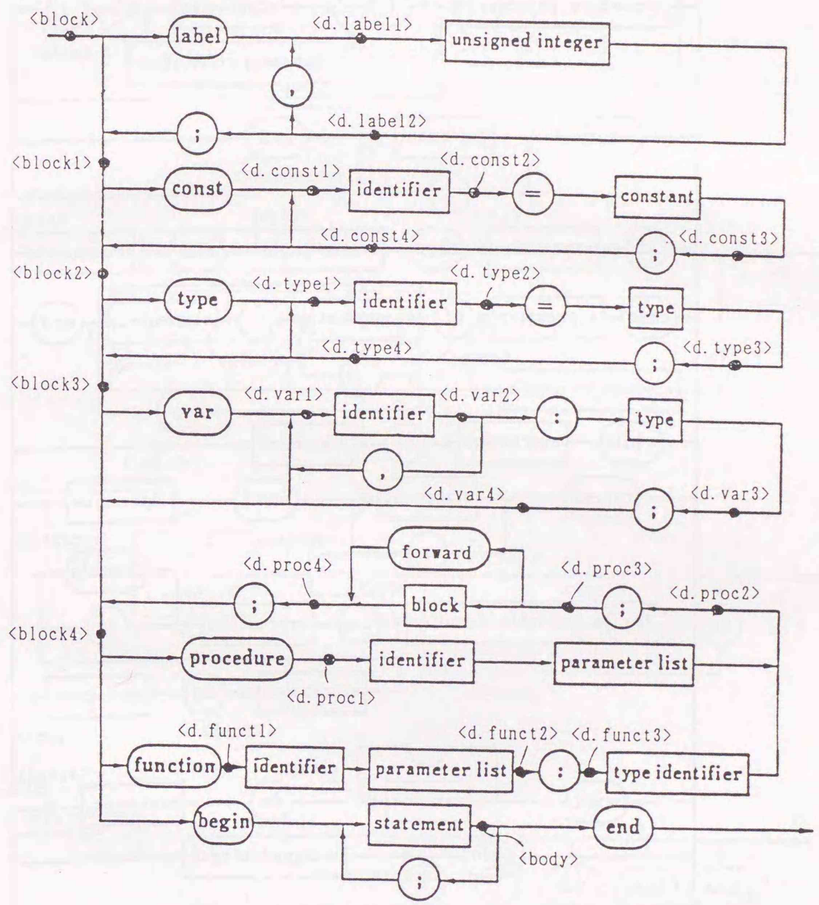
状態数：６１　　遷移数：１９４　　抑止項数：１７　　otherwise 項数：６８

付録Ⅱ－２　Pascal の構文規則

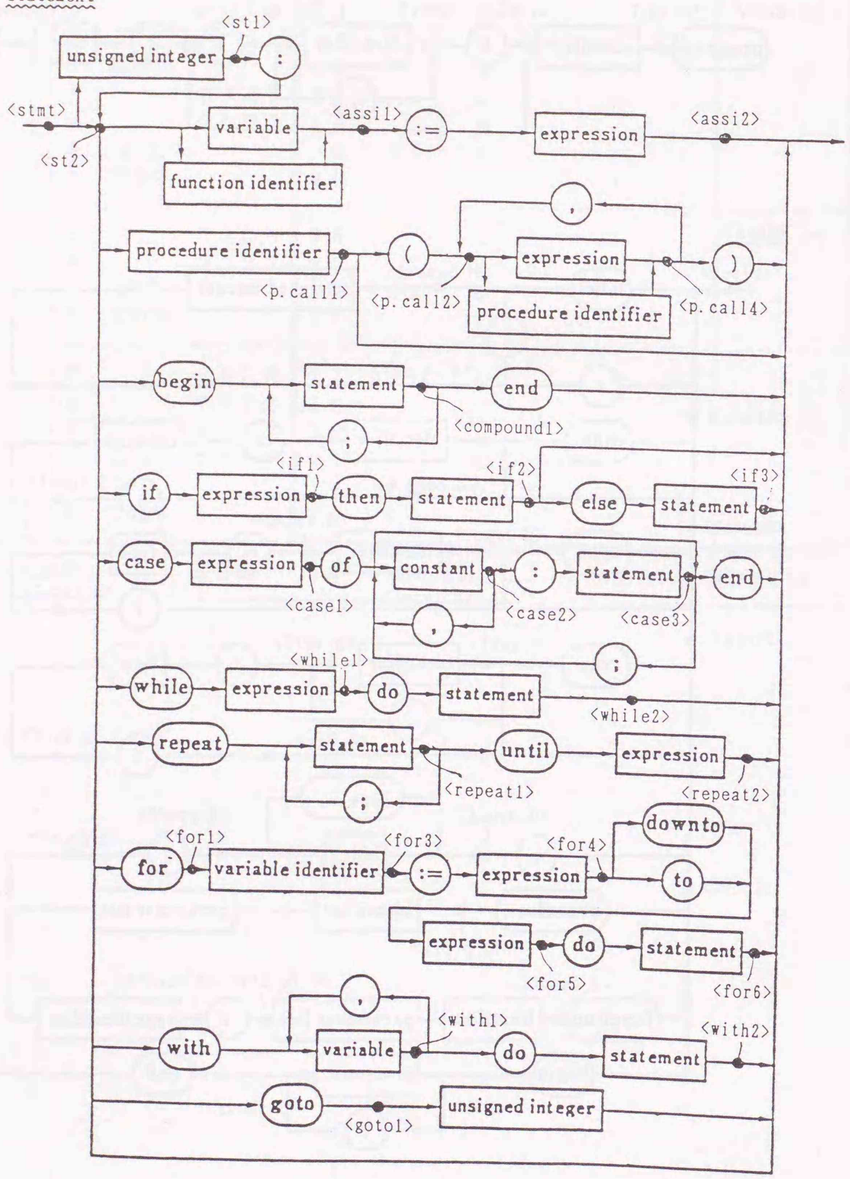
program



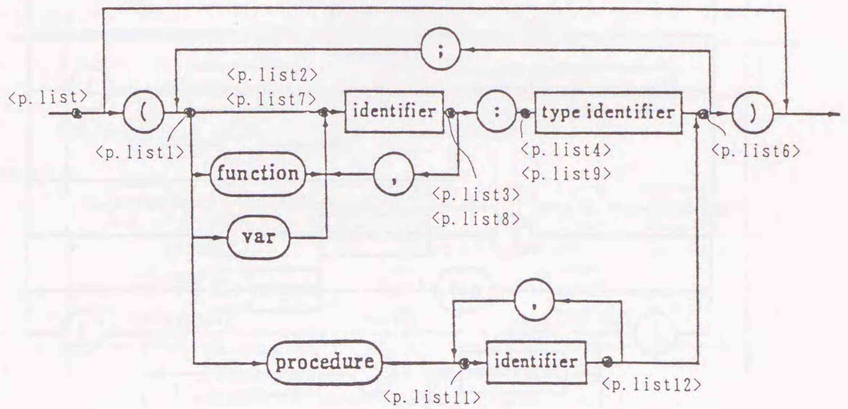
block



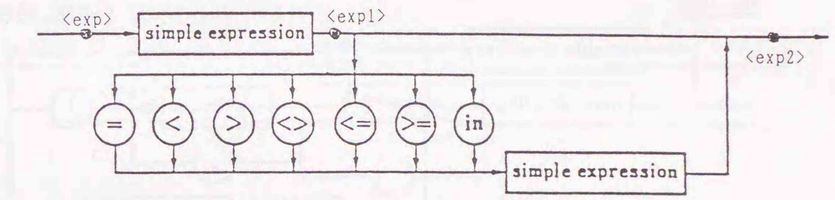
statement



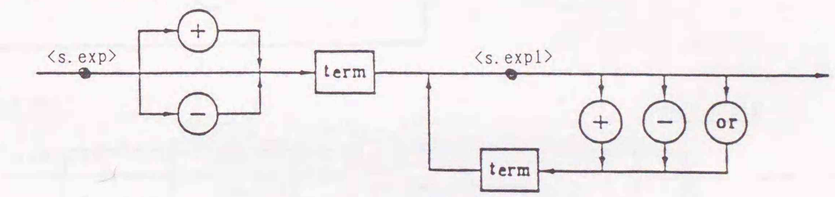
parameter list



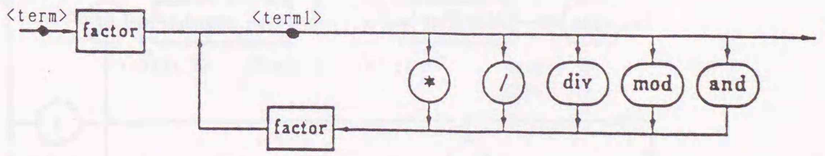
expression



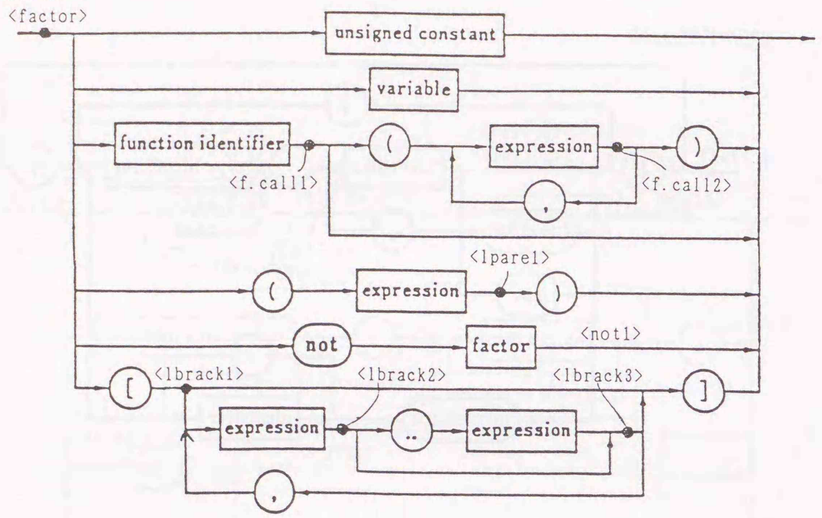
simple expression



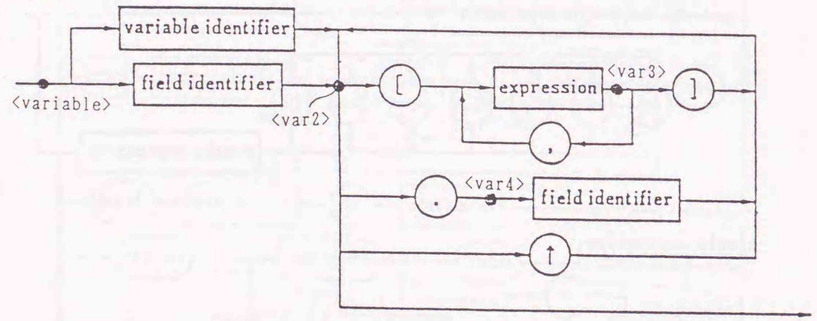
term



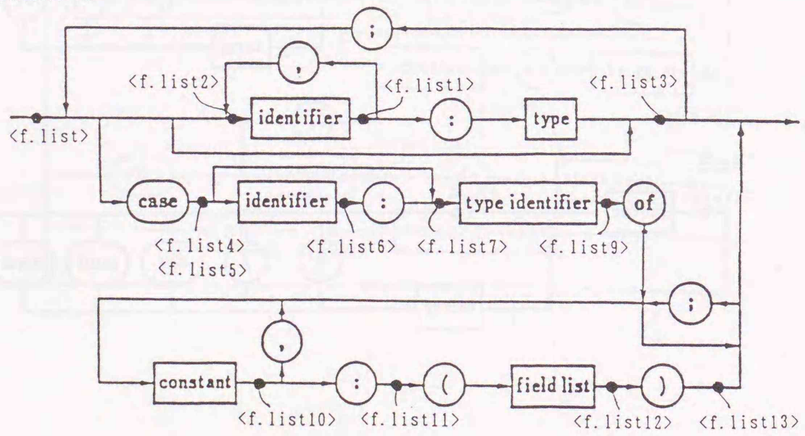
factor



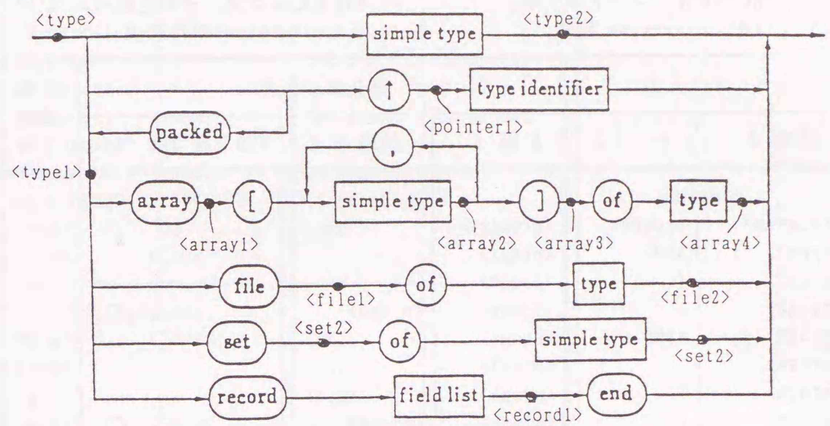
variable



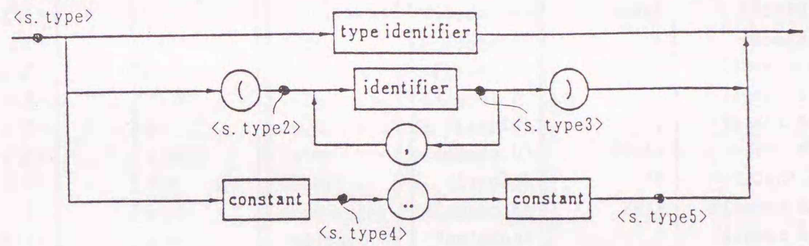
field list



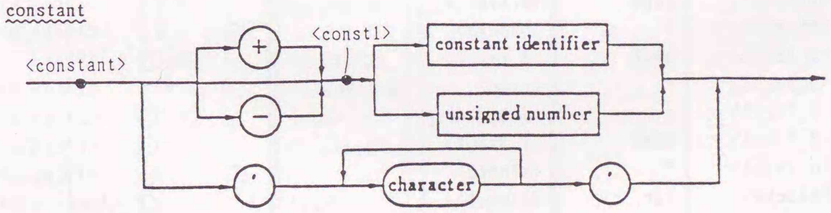
type



simple type



constant



付録Ⅱ－３　構文解析チップにおける状態遷移表

HLD：入力トークンのホールド機能，　　CHK：名前タイプの問合せ機能
TRC：ＴＲコマンドの生成，　　CGC：ＣＧコマンドの生成
ＯＷ：otherwise 機能，　　⊥：otherwise の対象項

ＰＬＡ　入力		ＰＬＡ　出力						拡張機能
状態 Z	トークン	状態 X	状態 Y	HLD	CHK	TRC	CGC	ＯＷ
program								
<program>	program	<prog1>				○	○	
<prog1>	NAME	<prog2>				○		
<prog2>	(	<prog3>						
<prog2>	;	<block>	<p.end>			○	○	
<prog3>	NAME	<prog4>				○		
<prog4>	,	<prog3>						
<prog4>	)	<prog5>						
<prog5>	;	<block>	<p.end>			○	○	
<p.end>	.	<program>				○	○	
block								
<block>	label	<d.label1>						○
<block>	?	<block1>		○				⊥
<d.label1>	INTG.C	<d.label2>				○		
<d.label2>	,	<d.label1>						
<d.label2>	;	<block1>						
<block1>	const	<d.const1>						○
<block1>	?	<block2>		○				⊥
<d.const1>	NAME	<d.const2>				○		
<d.const2>	=	<constant>	<d.const3>					
<d.const3>	;	<d.const4>				○		
<d.const4>	NAME	<d.const2>				○		○
<d.const4>	?	<block2>		○				⊥
<block2>	type	<d.type1>						○
<block2>	?	<block3>		○				⊥
<d.type1>	NAME	<d.type2>				○		
<d.type2>	=	<type>	<d.type3>					
<d.type3>	;	<d.type4>				○		
<d.type4>	NAME	<d.type2>				○		○
<d.type4>	?	<block3>		○		○		⊥
<block3>	var	<d.var1>				○		○
<block3>	?	<block4>		○				⊥
<d.var1>	NAME	<d.var2>				○		
<d.var2>	,	<d.var1>						
<d.var2>	:	<type>	<d.var3>					
<d.var3>	;	<d.var4>				○		
<d.var4>	NAME	<d.var2>				○		○
<d.var4>	?	<block4>		○		○		⊥
<block4>	procedure	<d.proc1>				○		
<block4>	function	<d.funct1>				○		
<block4>	begin	<stmt>	<body>			○	○	
<d.proc1>	NAME	<p.list>	<d.proc2>			○		
<d.proc2>	;	<d.proc3>				○		
<d.proc3>	forward	<d.proc4>				○		○
<d.proc3>	?	<block>	<d.proc4>	○		○	○	⊥
<d.proc4>	;	<block4>				○		
<d.funct1>	NAME	<p.list>	<d.funct2>			○		
<d.funct2>	:	<d.funct3>				○		
<d.funct3>	NAME	<d.funct3>		○	○[3]			
<d.funct3>	TYPE.ID	<d.proc2>				○		
<body>	;	<stmt>	<body>					
<body>	end					○	○	
statement								
<stmt>	INTG.C	<st1>					○	○
<stmt>	?	<st2>		○				⊥
<st1>	:	<st2>						
<st2>	NAME	<st2>		○	○			○
<st2>	VAR.ID	<var2>	<assi1>				○	○
<st2>	FIELD.ID	<var2>	<assi1>				○	○
<st2>	FUNCT.ID	<assi1>					○	○
<st2>	PROC.ID	<p.call1>					○	○
<st2>	begin	<stmt>	<compound1>					○
<st2>	if	<exp>	<if1>					○
<st2>	case	<exp>	<case1>					○
<st2>	while	<exp>	<while1>				○	○
<st2>	repeat	<stmt>	<repeat1>				○	○
<st2>	for	<for1>					○	○
<st2>	with	<variable>	<with1>				○	○
<st2>	goto	<goto1>						○
<st2>	?			○[1]				⊥
<assi1>	:=	<exp>	<assi2>				○	
<assi2>	?			○[2]			○	
<p.call1>	(	<p.call2>					○	○
<p.call1>	?			○[1]			○	⊥
<p.call2>	NAME	<p.call2>		○	○[3]			○
<p.call2>	PROC.ID	<p.call4>					○	○
<p.call2>	?	<exp>	<p.call4>	○			○	⊥
<p.call4>	,	<p.call2>					○	
<p.call4>	)						○	
<compound1>	;	<stmt>	<compound1>					
<compound1>	end							
<if1>	then	<stmt>	<if2>				○	
<if2>	else	<stmt>	<if3>				○	○
<if2>	?			○[1]			○	⊥
<if3>	?			○[2]			○	
<case1>	of	<constant>	<case2>				○	
<case2>	,	<constant>	<case2>				○	
<case2>	:	<stmt>	<case3>				○	

<case3>	;	<constant>	<case2>			○		
<case3>	end					○		
<while1>	do	<stmt>	<while2>			○		
<while2>	?			○2)		○		
<repeat1>	;	<stmt>	<repeat1>					
<repeat1>	until	<exp>	<repeat2>					
<repeat2>	?			○2)		○		
<for1>	NAME	<for1>		○	○3)			
<for1>	VAR.ID	<for3>				○		
<for3>	:=	<exp>	<for4>					
<for4>	to	<exp>	<for5>			○		
<for4>	downto	<exp>	<for5>			○		
<for5>	do	<stmt>	<for6>			○		
<for6>	?			○2)		○		
<with1>	,	<variable>	<with1>			○		
<with1>	do	<stmt>	<with2>					
<with2>	?			○2)		○		
<goto1>	INTG.C					○		
parameter list								
<p.list>	(	<p.list1>			○			○
<p.list>	?			○1)	○			⊥
<p.list1>	function	<p.list7>						○
<p.list1>	var	<p.list2>			○			○
<p.list1>	procedure	<p.list11>						○
<p.list1>	?	<p.list2>		○	○			⊥
<p.list2>	NAME	<p.list3>			○			
<p.list3>	,	<p.list2>						
<p.list3>	:	<p.list4>						
<p.list4>	NAME	<p.list4>		○	○3)			
<p.list4>	TYPE.ID	<p.list6>			○			
<p.list6>	;	<p.list1>						
<p.list6>	)				○			
<p.list7>	NAME	<p.list8>			○			
<p.list8>	,	<p.list7>						
<p.list8>	:	<p.list9>						
<p.list9>	NAME	<p.list9>		○	○3)			
<p.list9>	TYPE.ID	<p.list6>			○			
<p.list11>	NAME	<p.list12>			○			
<p.list12>	,	<p.list11>						
<p.list12>	;	<p.list1>						
<p.list12>	)				○			
expression								
<exp>	?	<s.exp>	<exp1>	○				
<exp1>	=	<s.exp>	<exp2>			○		○
<exp1>	<	<s.exp>	<exp2>			○		○
<exp1>	>	<s.exp>	<exp2>			○		○
<exp1>	<>	<s.exp>	<exp2>			○		○
<exp1>	<=	<s.exp>	<exp2>			○		○

<exp1>	>=	<s.exp>	<exp2>			○		○
<exp1>	in	<s.exp>	<exp2>			○		○
<exp1>	?	<exp2>		○				⊥
<exp2>	?			○2)		○		
simple expression								
<s.exp>	+	<term>	<s.exp1>			○		○
<s.exp>	-	<term>	<s.exp1>			○		○
<s.exp>	?	<term>	<s.exp1>	○		○		⊥
<s.exp1>	+	<term>	<s.exp1>			○		○
<s.exp1>	-	<term>	<s.exp1>			○		○
<s.exp1>	or	<term>	<s.exp1>			○		○
<s.exp1>	?			○1)		○		⊥
term								
<term>	?	<facter>	<term1>	○				
<term1>	*	<factor>	<term1>			○		○
<term1>	/	<factor>	<term1>			○		○
<term1>	div	<factor>	<term1>			○		○
<term1>	mod	<factor>	<term1>			○		○
<term1>	and	<factor>	<term1>			○		○
<term1>	?			○1)		○		⊥
factor								
<factor>	NAME	<factor>		○	○			
<factor>	CONST.ID					○		
<factor>	VAR.ID	<var2>				○		
<factor>	FIELD.ID	<var2>				○		
<factor>	FUNCT.ID	<f.call1>				○		
<factor>	INTG.C					○		
<factor>	REAL.C					○		
<factor>	nil					○		
<factor>	STRG.C					○		
<factor>	(	<exp>	<lpare1>					
<factor>	not	<factor>	<not1>					
<factor>	[	<lbrack1>				○		
<f.call1>	(	<exp>	<f.call2>			○		○
<f.call1>	?			○1)		○		⊥
<f.call2>	,	<exp>	<f.call2>			○		
<f.call2>	)					○		
<lpare1>	)							
<not1>	?			○2)		○		
<lbrack1>	]					○		○
<lbrack1>	?	<exp>	<lbrack2>	○		○		⊥
<lbrack2>	..	<exp>	<lbrack3>			○		○
<lbrack2>	?	<lbrack3>		○		○		⊥
<lbrack3>	,	<exp>	<lbrack2>			○		
<lbrack3>	]					○		
variable								

<variable>	NAME	<variable>		○ ○		
<variable>	VAR.ID	<var2>			○	
<variable>	FIELD.ID	<var2>			○	
<var2>	[	<exp>	<var3>		○	○
<var2>	.	<var4>			○	○
<var2>	^	<var2>			○	○
<var2>	?			○[1]		⊥
<var3>	,	<exp>	<var3>		○	
<var3>	]	<var2>			○	
<var4>	NAME	<var4>		○ ○[3]		
<var4>	FIELD.ID	<var2>			○	
field list						
<f.list>	NAME	<f.list1>			○	○
<f.list>	case	<f.list4>			○	○
<f.list>	?			○[1]		⊥
<f.list1>	,	<f.list2>				
<f.list1>	:	<type>	<f.list3>			
<f.list2>	NAME	<f.list1>			○	
<f.list3>	;	<f.list>			○	○
<f.list3>	?			○[1]		⊥
<f.list4>	NAME	<f.list5>		○ ○		
<f.list5>	NAME	<f.list6>			○	
<f.list5>	TYPE.ID	<f.list9>			○	
<f.list6>	:	<f.list7>				
<f.list7>	NAME	<f.list7>		○ ○[3]		
<f.list7>	TYPE.ID	<f.list9>			○	
<f.list9>	of	<constant>	<f.list10>		○	
<f.list10>	,	<constant>	<f.list10>		○	
<f.list10>	:	<f.list11>			○	
<f.list11>	(	<f.list>	<f.list12>			
<f.list12>	)	<f.list13>			○	
<f.list13>	;	<constant>	<f.list10>		○	○
<f.list13>	?			○[1]	○	⊥
type						
<type>	^	<pointer1>			○	○
<type>	packed	<type1>				○
<type>	array	<array1>				○
<type>	file	<file1>				○
<type>	set	<set1>				○
<type>	record	<f.list>	<record1>		○	○
<type>	?	<s.type>	<type2>	○		⊥
<type1>	array	<array1>				
<type1>	file	<file1>				
<type1>	set	<set1>				
<type1>	record	<f.list>	<record1>		○	
<type2>	?			○[2]	○	
<pointer1>	NAME	<pointer1>		○ ○[3]		
<pointer1>	TYPE.ID				○	

<array1>	[	<s.type>	<array2>		○	
<array2>	,	<s.type>	<array2>		○	
<array2>	]	<array3>			○	
<array3>	of	<type>	<array4>			
<array4>	?			○[2]	○	
<file1>	of	<type>	<file2>			
<file2>	?			○[2]		
<set1>	of	<s.type>	<set2>			
<set2>	?			○[2]	○	
<record1>	end				○	
simple type						
<s.type>	NAME	<s.type>		○ ○[3]		○
<s.type>	TYPE.ID				○	○
<s.type>	(	<s.type2>			○	○
<s.type>	?	<constant>	<s.type4>	○	○	⊥
<s.type2>	NAME	<s.type3>			○	
<s.type3>	,	<s.type2>				
<s.type3>	)				○	
<s.type4>	..	<constant>	<s.type5>		○	
<s.type5>	?			○[2]	○	
constant						
<constant>	+	<const1>			○	
<constant>	-	<const1>			○	
<constant>	NAME	<constant>		○ ○[3]		
<constant>	CONST.ID				○	
<constant>	INTG.C				○	
<constant>	REAL.C				○	
<constant>	STRG.C				○	
<const1>	NAME	<const1>		○ ○[3]		
<const1>	CONST.ID				○	
<const1>	INTG.C				○	
<const1>	REAL.C				○	

状態数：１２４　　遷移数：２４９　　抑止項数：０　　otherwise 項数：６６

1) 上位の構文に戻るための状態遷移
2) 終了処理を挿入するための状態遷移
3) 厳密な解析のための処理

付録Ⅱ－4　ＰＬＡコンパイラのＤＰＬ記述

```
(defcell cell ()
    (partq m1 m-bus (width 14))
    (partq m2 m-bus (width 14) (center (pt-above (>> center m1) 7)))
    (partq c1 contact (center-left (>> center-left m1)))
    (partq c2 contact (center-left (>> center-left m2)))
    (partq p1 p-bus (length 14)
                    (bottom-center (pt (+ 4 (>> x bottom-center c1))
                                       (+ -1 (>> y bottom-center c1)))))
    (partq p2 p-bus (length 14) (center (pt-to-right (>> center p1) 6)))
    (partq d1 d-bus (length 14) (center (pt-to-right (>> center p1) 3))))

(defcell connect ()
    (partq b1 b-contact)
    (rot270 (>> b1))
    (partq b2 b-contact)
    (rot270 (>> b2))
    (align (>> b2) (>> center b2) (pt-above (>> center b1) 7))
    (partq p1 p-bus (width 10) (bottom-left (>> apparent-bottom-right b1)))
    (partq p2 p-bus (width 10) (center-left (>> apparent-center-right b2)))
    (partq p3 p-bus (top-right (>> bottom-right p2)))
    (partq m1 m-bus (length 14) (bottom-center (pt-below (>> center p1) 2)))
    (partq c1 contact (bottom-center (pt-above (>> bottom-center m1) 4)))
    (partq d1 d-bus (width 3) (bottom-left (>> bottom-right c1))))

(defcell pullup ()
    (partq m1 m-bus (length 14))
    (partq c1 contact (bottom-center (pt-above (>> bottom-center m1) 1)))
    (partq d1 d-bus (width 4) (center-left (>> center-right c1)))
    (partq tr1 fet (length 8))
    (rot90 (>> tr1))
    (align (>> tr1) (>> apparent-center-left tr1) (>> center-right d1))
    (partq i1 rectangle (layer 'ion) (width 11) (length 5)
                        (center (>> center tr1)))
    (partq p1 p-bus (length 4) (center-right (>> apparent-center-right tr1)))
    (partq c2 rectangle (layer 'cut) (width 2) (length 2)
                        (center-right (>> apparent-center-right tr1)))
    (partq m2 m-bus (width 3) (center-right (>> apparent-center-right tr1)))
    (partq d2 d-bus (length 6) (bottom-right (>> top-right c1)))
    (partq d3 d-bus (width 5) (top-left (>> top-right d2)))
    (partq b1 b-contact)
    (rot90 (>> b1))
    (align (>> b1) (>> apparent-center-left b1)
                   (pt-to-left (>> center-right d3) 2))
    (partq d4 d-bus (width 7) (center-left (>> apparent-center-right b1)))
    (partq p2 p-bus (width 6) (length 7)
                    (top-left (pt-below (>> top-left m1) 1)))
    (partq p3 p-bus (length 6) (top-left (>> top-right p2)))
    (partq i2 rectangle (layer 'ion) (width 5) (length 8)
                        (bottom-left (pt (+ 0.5 (>> x top-left c1))
                                         (+ -0.5 (>> y top-left c1)))))
    (partq i3 rectangle (layer 'ion) (width 4) (length 5)
                        (top-left (>> top-right i2))))

(defcell ground ()
    (partq m1 m-bus (width 14))
    (partq c1 contact (center (pt-to-right (>> center-left m1) 8)))
    (partq d1 d-bus (length 3) (bottom-right (>> top-right c1)))
    (partq d2 d-bus (length 2) (top-right (>> bottom-right c1)))
    (partq p1 p-bus (length 10) (center (pt-to-right (>> center-left m1) 4)))
    (partq p2 p-bus (top-left (>> top-right p1)))
    (partq p3 p-bus (bottom-left (>> bottom-right p1)))
    (partq p4 p-bus (length 9)
                    (top-center (pt-to-right (>> top-center d1) 3))))

(defcell input ()
    (partq pb1 p-bus (width 14))
    (partq mb1 m-bus (width 14) (center (pt-below (>> center pb1) 12)))
    (partq mb2 m-bus (width 14) (center (pt-below (>> center mb1) 19)))
    (partq d1 d-bus (length 6)
                    (top-center (pt (+ -3 (>> x center-right pb1))
                                    (+  3 (>> y center-right pb1)))))
    (partq bc1 b-contact)
    (rot90 (>> bc1))
    (align (>> bc1) (>> apparent-top-right bc1) (>> bottom-right d1))
    (partq p1 p-bus (length 8) (top-right (>> apparent-bottom-center bc1)))
    (partq c1 contact (center (pt-to-right (>> center-left mb1) 4)))
    (partq tr1 fet (width 4) (center (pt-below (>> center c1) 4)))
    (partq tr2 pull-up)
    (rot180 (>> tr2))
    (align (>> tr2) (>> apparent-top-center tr2)
                    (pt (+ -1 (>> x bottom-center tr1))
                        (+  1 (>> y bottom-center tr1))))
    (partq c2 contact (center-left (>> center-left mb2)))
    (partq p2 p-bus (length 9)
                    (top-center (pt (+ 3 (>> x apparent-bottom-center tr2))
                                    (+ 4 (>> y apparent-bottom-center tr2)))))
    (partq p3 p-bus (width 4) (center-left (>> center-right tr1)))
    (partq p4 p-bus (length 16) (top-right (>> bottom-right p3)))
    (partq p5 p-bus (length 4) (center-left (>> bottom-center p4))))

(defcell output ()
    (partq pb1 p-bus (width 14))
    (partq mb1 m-bus (width 14) (center (pt-below (>> center pb1) 18)))
    (partq mb2 m-bus (width 14) (center (pt-below (>> center mb1) 29)))
    (partq c1 contact (bottom-center (pt (+ -4 (>> x center-right pb1))
                                         (+  2 (>> y center-right pb1)))))
    (partq d1 d-bus (length 4) (top-center (>> bottom-center c1)))
    (partq bc1 b-contact)
```

```
(rot180 (>> bc1))
(align (>> bc1) (>> apparent-top-center bc1) (>> bottom-center d1))
(partq c2 contact (bottom-center (pt (+ 3 (>> x center-left pb1))
                                     (+ 2 (>> y center-left pb1)))))
(partq d2 d-bus (length 4) (top-center (>> bottom-center c2)))
(partq bc2 b-contact)
(rot180 (>> bc2))
(align (>> bc2) (>> apparent-top-center bc2) (>> bottom-center d2))
(partq c3 contact (center (pt-to-right (>> center-left mb1) 7)))
(partq d3 d-bus (length 3) (bottom-center (>> top-center c3)))
(partq d4 d-bus (width 12) (length 4) (bottom-center (>> top-center d3)))
(partq p1 p-bus (length 7) (top-left (>> apparent-bottom-left bc1)))
(partq p2 p-bus (length 9)
                (top-center (pt (+ 2 (>> x apparent-bottom-center b
                                   (+ 2 (>> y apparent-bottom-center bc2)))))
(partq d5 d-bus (length 12) (top-right (>> bottom-right d4)))
(partq tr1 pull-up)
(rot180 (>> tr1))
(align (>> tr1) (>> apparent-top-center tr1)
                (pt (+ -2 (>> x bottom-center d5))
                    (+  2 (>> y bottom-center d5))))
(partq d6 d-bus (length 21) (top-left (>> bottom-left d4)))
(partq tr2 pull-up)
(rot180 (>> tr2))
(align (>> tr2) (>> apparent-top-center tr2)
                (pt (+ 2 (>> x bottom-center d6))
                    (+ 2 (>> y bottom-center d6))))
(partq d7 d-bus (length 10)
                (top-center (pt (+ -1 (>> x apparent-bottom-center tr1))
                                (+ 2 (>> y apparent-bottom-center tr1)))))
(partq d8 d-bus (width 7) (top-right (>> bottom-right d7)))
(partq c4 contact (center (pt-to-right (>> center-left mb2) 7)))
(partq p3 p-bus (length 18)
                (top-center (pt (+ 2 (>> x apparent-bottom-center tr1))
                                (+ 2 (>> y apparent-bottom-center tr1)))))
(partq p4 p-bus (length 9)
                (top-center (pt (+ -2 (>> x apparent-bottom-center tr2))
                                (+ 2 (>> y apparent-bottom-center tr2))))))

(defcell pla ((in 6) (pterm 8) (out 4) (table *table*))
    (partq in-1 input)
    (replicateq in-row array (list (>> in) 1) (list (>> in-1)))

    (partq gnd-1 ground)
    (replicateq gnd-row1 array (list (>> in) 1) (list (>> gnd-1)))
    (align (>> gnd-row1) (>> top-center gnd-row1) (>> bottom-center in-row))

    (partq cell-1 cell)
    (replicateq cell-matrix1 array (list (>> in) (quotient (>> pterm) 2))
                                   (list (>> cell-1)))
```



```
(align (>> cell-matrix1) (>> top-center cell-matrix1)
                         (pt-above (>> bottom-center gnd-row1) 1))

(partq gnd-2 ground)
(replicateq gnd-row2 array (list (>> in) 1) (list (>> gnd-2)))
(align (>> gnd-row2) (>> top-center gnd-row2)
                     (>> bottom-center cell-matrix1))

(partq pull-1 pullup)
(replicateq pull-col array (list 1 (quotient (>> pterm) 2))
                           (list (>> pull-1)))
(align (>> pull-col) (>> center-right pull-col)
                     (pt-to-right (>> center-left cell-matrix1) 1))

(partq con-1 connect)
(replicateq con-col array (list 1 (quotient (>> pterm) 2))
                          (list (>> con-1)))
(align (>> con-col) (>> center-left con-col)
                    (>> center-right cell-matrix1))

(partq cell-2 cell)
(rot270 (>> cell-2))
(replicateq cell-matrix2 array (list (quotient (>> out) 2)
                                     (quotient (>> pterm) 2))
                               (list (>> cell-2)))
(align (>> cell-matrix2) (>> center-left cell-matrix2)
                         (>> center-right con-col))

(do ((tab (>> table) (cdr tab))
     (count 0 (add1 count))
     (p1 (pt-sum (>> top-left cell-matrix1) (pt 4 -4)) (pt-below p1 7))
     (p2 (pt-sum (>> top-left cell-matrix2) (pt 3 -2)) (pt-below p2 6)))
    ((null tab))
    (cond ((eq (remainder count 2) 0) (setq p2 (pt-below p2 2))))
    (do ((taba (caar tab) (cdr taba))
         (pa1 p1 (pt-to-right pa1 14)))
        ((null taba))
        (cond ((eq (car taba) 0)
               (part nil d-bus (width 4) (center-left pa1)))
              ((eq (car taba) 1)
               (part nil d-bus (width 4)
                               (center-left (pt-to-right pa1 6))))))
    (do ((tabb (cadar tab) (cdr tabb))
         (pa2 p2 (pt-to-right pa2 7)))
        ((null tabb))
        (cond ((eq (car tabb) 1)
               (part nil d-bus (length 4) (top-center pa2))))))

(partq gnd-3 ground)
(rot270 (>> gnd-3))
```

```
(replicateq gnd-col array (list 1 (quotient (>> pterm) 2))
                          (list (>> gnd-3)))
(align (>> gnd-col) (>> center-left gnd-col)
                    (pt-to-left (>> center-right cell-matrix2) 1))

(partq pull-2 pullup)
(rot270 (>> pull-2))
(replicateq pull-row array (list (quotient (>> out) 2) 1)
                           (list (>> pull-2)))
(align (>> pull-row) (>> bottom-center pull-row)
                     (pt-below (>> top-center cell-matrix2) 1))

(partq out-1 output)
(replicateq out-row array (list (quotient (>> out) 2) 1) (list (>> out-1)))
(align (>> out-row) (>> top-center out-row)
                    (>> bottom-center cell-matrix2))

(partq v1 m-bus (length 47)
                (bottom-center (pt (+ 2 (>> x top-left pull-col))
                                   (>> y top-left pull-col))))
(partq v2 m-bus (width 18)
                (center-right (pt (>> x bottom-left in-row)
                                  (+ 4 (>> y bottom-left in-row)))))
(partq v3 m-bus (width 10)
                (center-left (pt (>> x bottom-right in-row)
                                 (+ 4 (>> y bottom-right in-row)))))
(partq v4 m-bus (width 10)
                (center-right (pt (>> x top-left pull-row)
                                  (+ -2 (>> y top-left pull-row)))))
(partq v5 m-bus (length 55)
                (top-center (pt (+ 2 (>> x bottom-left pull-col))
                                (>> y bottom-left pull-col))))
(partq v6 m-bus (width (+ 32 (times 14 (>> in))))
                (center-right (pt (>> x bottom-left out-row)
                                  (+ 4 (>> y bottom-left out-row)))))

(partq g0 m-bus (length 47)
                (bottom-right (pt (>> x top-right gnd-col)
                                  (>> y top-right gnd-col))))
(partq g1 m-bus (width 3) (bottom-right (>> bottom-left g0)))
(partq g2 m-bus (width (+ 21 (times 7 (>> out))))
                (center-left (pt (>> x top-right in-row)
                                 (+ -15 (>> y top-right in-row)))))
(partq g3 m-bus (length 26)
                (top-center (pt (+ -5 (>> x bottom-right gnd-col))
                                (>> y bottom-right gnd-col))))
(partq g4 m-bus (width 3)
                (center-left (pt (>> x bottom-right out-row)
                                 (+ 33 (>> y bottom-right out-row)))))
(partq g5 m-bus (width 7)
```



```
                (center-right (pt (>> x bottom-left out-row)
                                  (+ 33 (>> y bottom-left out-row)))))
(partq g6 m-bus (length 22)
                (top-center (pt (+ 11 (>> x bottom-left con-col))
                                (>> y bottom-left con-col))))
(partq g7 m-bus (width 9)
                (center-left (pt (>> x bottom-right gnd-row2)
                                 (+ 5 (>> y bottom-right gnd-row2)))))
(partq g8 m-bus (length 6)
                (bottom-center (pt (+ 11 (>> x top-left con-col))
                                   (>> y top-left con-col))))
(partq g9 m-bus (width 9)
                (center-left (pt (>> x bottom-right gnd-row1)
                                 (+ 5 (>> y bottom-right gnd-row1)))))
(partq p1 p-bus (width 20)
                (center-right (pt (>> x top-left in-row)
                                  (+ -3 (>> y top-left in-row)))))
(partq p2 p-bus (width 10)
                (center-left (pt (>> x top-right out-row)
                                 (+ -6 (>> y top-right out-row))))))

; Example of PLA table

(setq *table* '(((x 1 x 0 1 1) (1 x x 1))
                ((1 x 0 x x 0) (x 1 x x))
                ((0 x x x 1 x) (x x 1 x))
                ((x x 1 0 0 x) (x x x 1))
                ((1 x x 1 x 0) (x 1 x 1))
                ((x x x 0 x x) (x x 1 1))
                ((1 1 x x 1 x) (1 1 x x))
                ((1 x x 0 0 0) (1 x 1 x))))
```

第Ⅲ部　動作記述型シリコン・コンパイラに関する研究

1．はじめに

ＶＬＳＩチップの設計は，チップの仕様レベルの設計から，機能レベル，論理回路レベル，トランジスタ回路レベル，及び，レイアウト・レベルの設計へと具体化していかなければならない．通常，あるレベルから下位の設計レベルに展開する方法は多数あり，最終的なチップとしての性能を意識しながら各設計レベルでの最適化を行っていくことは，高度の専門知識や設計のノウハウを必要とし，多大の設計労力を要する困難な作業である．加えて，近年のＶＬＳＩチップの大規模化や適用分野の拡大にともなって，ＶＬＳＩチップの設計効率を飛躍的に改善する設計自動化ツールへの要求が急速に高まっている[1,2]．このような要求に応える設計自動化手法として，ＶＬＳＩチップの機能，動作，マクロ構造等の高位の設計記述を受け入れ，これからチップとして製造可能なマスクパターンの情報を直接生成する“**シリコン・コンパイラ**”が提案されている[3,4,5,6,7]．

現状の技術レベルでは，高度の専門知識を備えた設計者によって最適化されたチップと同程度に効率的なチップを，ソフトウェア的に自動生成することは大変困難であり，以下に示す２つのアプローチからの研究開発が進められている[7]．

① 構造記述型シリコン・コンパイラ

レジスタ転送レベルでの論理的な構成要素と接続関係の記述を受け入れ，これからチップの物理的な構成要素への展開と配置・配線を自動化したシリコン・コンパイラであり，生成されるＶＬＳＩチップの性能面（チップ面積や動作速度）の最適化を狙っている．このようなシリコン・コンパイラは，従来のＶＬＳＩ設計手法の延長線上に位置しているため，現在，幾つかの実用的なツールが市販され，実用化の段階に入りつつある[9,10,11]．例えば，Silicon Compiler Systems Inc. のシリコン・コンパイラ Genesil では，モジュール・コンパイラを用いてチップに用いられる構成要素を設計し，これらの接続関係をブロック図として図的に入力することにより，各応用向きに専用化されたデータパスを自動設計することができる[6]．

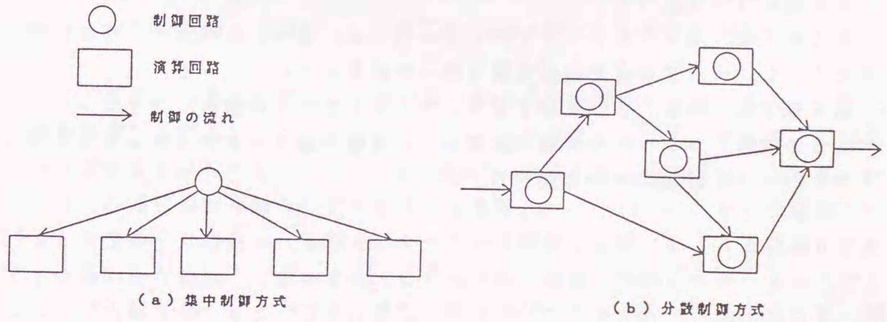
② 動作記述型シリコン・コンパイラ

プログラミング言語等による機能，動作レベルの記述から，その記述された機能を実現するＶＬＳＩチップを自動生成するシリコン・コンパイラであり，生成されるチップの性能面よりも，多くの応用技術者が自らチップ設計することを可

能にし，チップの設計期間の大幅な短縮を狙っている．このような動作記述型シリコン・コンパイラにおいて汎用的なチップの生成を目的とすると，生成されるチップの性能に限界があるため，チップの構造や適用分野を特定なものに限定することによって，シリコン・コンパイラの可能性を試作・評価する研究が盛んである．例えば，信号処理用チップ専用のシリコン・コンパイラ[12,13]，特定用途向きのμプロセッサを自動生成するシリコン・コンパイラ[14]，シストリック・アレイを対象としたシリコン・コンパイラ[15]，パイプライン・プロセッサを生成するシリコン・コンパイラ[16]等 の研究開発が行われている．

一方，特定の応用分野を限定しない汎用的な動作記述型シリコン・コンパイラでは，プログラミング言語等による動作記述からデータの依存関係や演算の並列性を解析し，ＶＬＳＩチップとしてのチップ面積や動作速度の観点から最適なデータパスを抽出することに研究の主眼が置かれている[17,18,19,20]．このような汎用的なシリコン・コンパイラでは，複数の演算・記憶要素からなるデータパスを１個の制御回路で管理する集中制御方式（**図Ⅲ－１（ａ）**参照）のチップを前提としているものが多い．即ち，生成されるチップは，通常のμプロセッサと同様に，レジスタや演算回路等からなる“データパス部”と，データパス部の動作を制御する“制御部”とで構成され，これらが共通のタイミング・クロックに同期して動作する実現方式を採用しているものが多い．

しかし，多数の構成要素からなる大規模なデータパス部を最適化することや，高い並列性を維持して動作するための大規模な制御部を設計することは，高度な設計知識を必要とする知的作業であり，手続き的に記述された大規模なプログラムからこのような設計作業をソフトウェア的に自動化することは，多大の計算時間を必要とし，大変困難である[21]．さらに，１個の制御部からチップのあらゆる場所に制御信号を供給する集中制御方式では，この配線のために多くのチップ面積を費やすばかりでなく，制御信号線の長さの違いによってタイミング・クロックの“ずれ”が発生して，チップが正常動作できなくなる可能性もある．



図Ⅲ－１　制御方式の比較

一般に，プログラミング言語による動作記述と従来型の計算機アーキテクチャとでは，設計レベルとして大きな隔たりがあり，これをソフトウェア的に自動変換することには，本質的に大変困難な操作を含んでいる．従って，従来の計算機アーキテクチャを前提とするのではなく，ＶＬＳＩチップの実現方式自体をシリコン・コンパイラの動作記述や変換方法に適したように歩み寄ること，即ち，シリコン・コンパイラ向きのＶＬＳＩアーキテクチャを開発していくことが重要である．例えば，従来の計算機における制御構造とは異なるデータフロー方式や非同期方式等の新しいアーキテクチャを基本としたシリコン・コンパイラの研究が行われている[22,23,24]．

我々は，シリコン・コンパイラの実現に適したＶＬＳＩアーキテクチャとして，ＶＬＳＩシステムの制御部を一箇所に集中するのではなく，**図Ⅲ－１（ｂ）**に示すように，適当な回路規模の構成要素毎に制御回路を分散化し，構成要素間が各々独立して自己同期的に動作する分散制御アーキテクチャに関する研究を行っている．また，ハードウェア仕様記述言語によるＶＬＳＩチップの動作記述から，この方式に基づくチップを自動生成する動作記述型シリコン・コンパイラに関する研究を行っている[25,26,27]．即ち，ここで提案する分散制御アーキテクチャが，動作記述型シリコン・コンパイラの基本アーキテクチャとして適合しているかどうか，また，このアーキテクチャが将来のＶＬＳＩチップの実現方式として適切なものかどうかについて検討し，動作記述型シリコン・コンパイラの実現可能性を明らかにすることが本研究の目的である．

以下，第２章では，動作記述型シリコン・コンパイラの基本アーキテクチャとして設定する分散制御アーキテクチャの基本モデル，即ち，構成要素，データの流れ，制御の流れ，及び，チップ外部との通信方式について述べる．

第３章では，ハードウェア仕様記述言語によるＶＬＳＩチップの動作記述から，上記の分散制御アーキテクチャに基づくＶＬＳＩチップへのコンパイル過程について述べる．特に，シリコン・コンパイラの入力記述における条件分岐構文や並列動作の制御構文から，分散制御アーキテクチャにおける制御モジュールの動作への変換について述べる．

第４章では，本シリコン・コンパイラに対して，幾つかの例題チップの動作記述を入力し，チップの自動合成実験を行った結果について述べる．

第５章では，第２，３，４章で述べたアーキテクチャの検討，シリコン・コンパイラの試作，コンパイル実験の結果から，分散制御アーキテクチャとシリコン・コンパイラの評価について述べる．

２．分散制御アーキテクチャ

２．１　動作記述言語：ＩＳＰＳ

シリコン・コンパイラの基本アーキテクチャとしての分散制御アーキテクチャについて述べる前に，本シリコン・コンパイラの入力形式であるハードウェア仕様記述言語：ＩＳＰＳ（Instruction Set Processor Specification）[28] について概説する．

カーネギーメロン大学で開発されたＩＳＰＳは，命令セット・レベルでの計算機の動作を記述したり，ＩＳＰＳシミュレータによって計算機アーキテクチャを評価するのに適したハードウェア仕様記述言語である．ＩＳＰＳでは，ハードウェアの記述に適した簡単なデータ構造と演算子しか用意されていないが，並列処理に関してはいくつかの高度な機能を備えている．即ち，複数の動作を同時に実行する記述，並列に動作するプロセスや排他的な制御を受けるプロセスの記述等が行えるため，並列計算アルゴリズムを容易に記述できるという優れた特徴を備えた高級プログラミング言語である．

以下にＩＳＰＳの基本的な言語構造について述べる．

（１）基本構造

ＩＳＰＳでは，複数の構成要素のネットワークとしてデジタル・システムを定義する．この構成要素のうち，内部の動作が定義されず，外部から見える構造だけが定義される構成要素は，データを保持するための“**記憶要素**”を表している．このような記憶要素には，レジスタやフリップ・フロップ等のような“ビット構造”のものと，ＲＡＭやＲＯＭ等のような“語構造”の２種類がある．以下に，記憶要素の定義の例を示す．

```
OPC.FF <>,                ! flip-flop (1 bit)
DATA.REG <0:15>,          ! 16 bit register
RAM [0:255] <0:15>,       ! 256 word memory (16 bits/word)
```

また，外部から見える構造と内部の動作の両方が定義される構成要素（これを“**動作要素**”と呼ぶ）は，通常のプログラミング言語における“手続き”に対応するものである．しかし，これに“process” という修飾子を付与して定義した場合は，並列動作が可能な“プロセス”を意味し，“critical”という修飾子を付与して定義した場合は，複数のプロセスから同時に起動されることが禁止される“排他的動作要素”を意味する．これらの動作要素は外部とのデータを送受するための引数を持つことができ，また，通常のプログラミング言語における“関数”のように，動作要素自身が演算結果のデータを保持する動作要素を定義する

こともできる．以下に，動作要素の定義の例を示す．

```
I.DECODE (INST<0:15>) := begin ... end,          ! 手続き
SIN (ANGLE<0:15>)<0:15> := begin ... end,        ! 関数
process ALU (F<0:3>, A<0:15>, B<0:15>)<0:15> := begin ... end,
                                                 ! プロセス
critical ARBITER := begin ... end,               ! 排他的動作要素
```

（※ begin ... end の中には動作の記述が入る）

（２）動作

ＩＳＰＳにおいては，複数の動作を“next”という予約語で結合する“順序動作の記述”と，複数の動作を“；”で結合する“並列動作の記述”の両方が可能である．ここでいう“動作”には，以下に示す５種類がある．

・ブロック動作
複数の動作を begin と end でまとめて１個の動作としたもの．

・ラベル付き動作
１個のブロック動作にラベル付けしたもの．

例）X := begin ... end,

・条件動作
“if”と“decode”という２種類の条件動作を記述する構文がある．if は１ビットの入力データが真ならば，以下に記述される動作を実行し，decode は，任意長のデータの値に応じて，実行すべき動作を変更することができる．

例）if ACC eql X => PC = PC + 2

```
decode OP<1:0> => begin
    '00 :=    ACC = 0,
    '01 :=    ACC = ACC + M[Z],
    '10 :=    M[Z] = ACC,
    '11 :=    PC = M[Z]
end
```

・制御動作
動作の繰返しを指定する“repeat”構文にくわえて，動作の終了を指定する４

個の終了構文がある．“leave” は動作要素やラベル付き動作を強制的に終了する．“terminate”は leave と同様であるが，並列に動作している外部の動作も強制終了させることができる．“restart” は動作要素やラベル付き動作を再スタートする．“resume”は自分自身を終了するとともに，他の動作の実行を再開する．

例）
```
I.CYCLE := begin
        repeat begin
            IR = MEMORY[PC] next
            PC = PC + 1;    VMA = IR<13*35> next
            EA = VMA ()<18:35> next
            I.EXECUTE ()
        end
    end
```

・演算式　（下記参照）

（３）演算式

ＩＳＰＳにおける演算式では，記憶要素等をオペランドとする各種の演算や，実パラメータを伴った動作要素の起動を記述する．演算の記述で用いられる演算子としては，データ転送，論理演算子，比較演算子，算術演算子，シフト演算子，及び，単項演算子がある．この他に，ＩＳＰＳに特有のハードウェア記述向き演算子として，２個のデータを連接して１個のデータとする連接演算子“@”がある．これらの演算子の一覧表を**表Ⅲ－１**に示す．

表Ⅲ－１　ＩＳＰＳの演算子

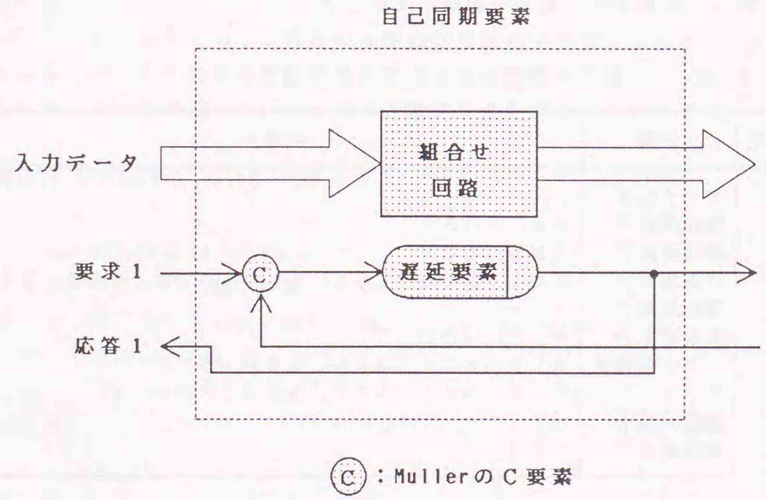
優先度	分類	演算子
高	データ転送	_，=，<=
↑	論理演算子	ＯＲ，ＸＯＲ
	論理演算子	ＡＮＤ，ＥＱＶ
	比較演算子	ＥＱＬ，ＮＥＱ，ＬＳＳ，ＬＥＱ，ＧＴＲ，ＧＥＱ，ＴＳＴ
	算術演算子	＋，－
	算術演算子	＊，／，ＭＯＤ
	シフト演算子	ＳＬ０，ＳＬ１，ＳＬＲ，ＳＬＤ，ＳＬＩ
		ＳＲ０，ＳＲ１，ＳＲＲ，ＳＲＤ，ＳＲＩ
↓	連結演算子	@
低	単項演算子	ＮＯＴ，＋，－

一方，動作要素の起動の記述において，動作要素が手続きと関数の場合は，呼ばれた動作要素の動作が終了するまで呼出し側の動作が停止する通常の“呼出し動作”であるが，動作要素がプロセスの場合は，呼出し側と呼ばれた側が並行して動作する“並列プロセスの起動”を意味する．関数の起動は，この関数自身が計算結果を保持しているため，演算式の中に記述されることも許される．

```
例) X<0:15> = Y<0:15> and (not Z<0:15>) ! 演算の記述
    ALU (F<0:3>, A<0:15>, B<0:15>)      ! 手続きの起動
    M = (I + J) * SIN (ANG<0:15>)       ! 演算式中での関数の起動
```

2．2　基本モデル

集中制御型のＶＬＳＩチップにおける内部動作の時間依存性を回避し，将来の大規模チップを実現するための新しいＶＬＳＩアーキテクチャとして，**図Ⅲ－2**に示すような要素で構成される“**自己同期システム**”が提案されている[29,30]．この構成要素は，他の構成要素からの要求によって動作を開始し，その動作が終わった後で起動要求を出した要素に応答を返すとともに，他の構成要素を起動するという動作を繰り返す．従って，１個の制御回路の管理下でシステム全体が同期して動作する従来の集中制御方式とは異なり，各構成要素が内部に制御回路を含んで自律的に動作する分散制御型のＶＬＳＩアーキテクチャと考えられる．このような自己同期システムは以下に示すような特徴がある．



図Ⅲ－２　自己同期システムの構成要素

① 各構成要素に含まれる制御回路は，チップ全体の複雑さに拘らず，構成要素内部の動作と他の要素との通信だけを制御する簡潔で小規模なものである．
② 多数の構成要素が同時に動作可能であり，高度の並列処理による処理性能の向上が期待できる．
③ 各構成要素毎の実現方式，特性等の独立性が高い．
④ 構成要素間の局所的な通信しか行われないため，制御信号のスキュー等の問題が発生しにくい．

上記のような特徴が挙げられる反面，一般的な自己同期システムを前提とするＶＬＳＩアーキテクチャでは，以下に示すような幾つかの実現上の問題がある．

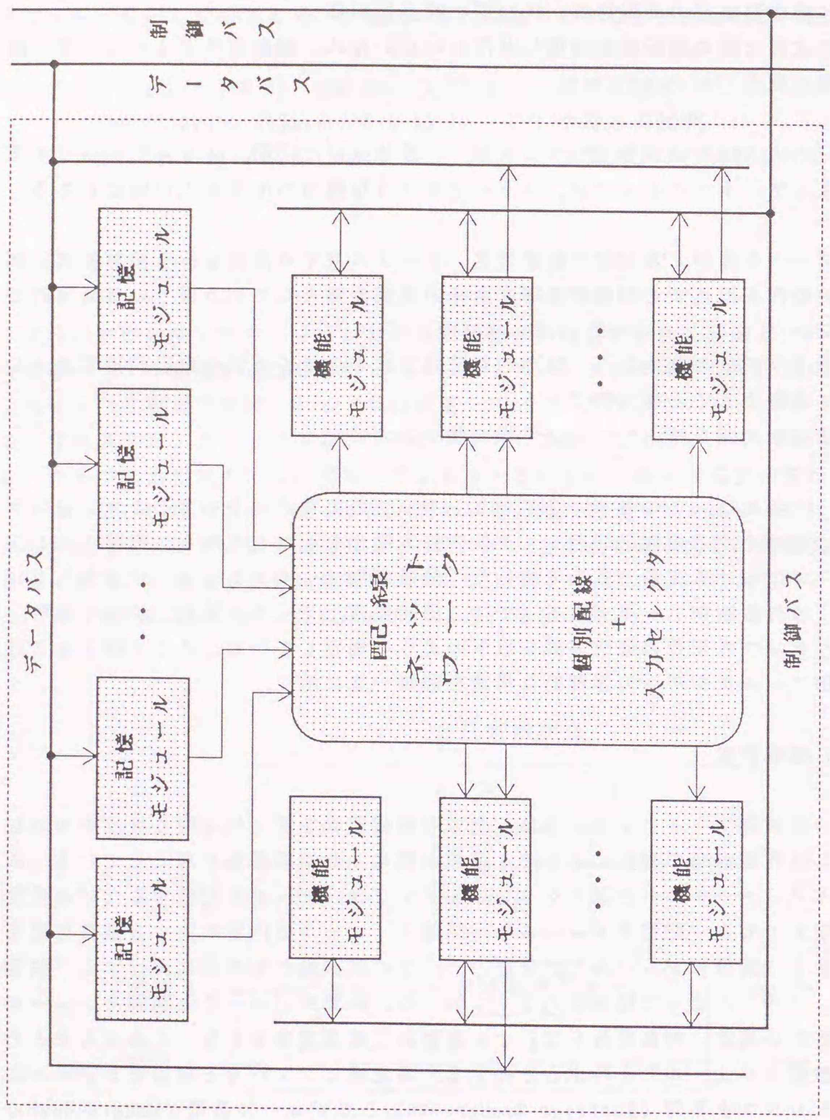
① データを保持するだけの記憶要素，データに対する操作を行う演算要素，制御動作を担当する制御要素等を画一の構成要素として取り扱うのは効率的でない．
② チップ面積の制約から，論理的な構成要素の全てを個別の自己同期要素として実現することは困難である．
③ 構成要素間を結ぶデータ信号線の数が膨大になる．

従って我々は，このモデルをＶＬＳＩチップの実現方式としてそのまま適用するのではなく，この自己同期システムの基本概念を活かしながら，実際のＶＬＳＩチップにおける実現方式を考慮した“**分散制御アーキテクチャ**”を提案している．この分散制御アーキテクチャでは，構成要素，データの流れ，制御の流れ，及び，チップ外部との通信方式を以下のように規定している．ここで提案する分散制御アーキテクチャの論理的な構造を**図Ⅲ－3**に示す．

（１）構成要素

本分散制御アーキテクチャでは，全ての構成要素を画一的に取り扱うのではなく，各構成要素毎の機能に応じて３種類の異なった構成要素を導入している．即ち，本アーキテクチャに基づくＶＬＳＩチップは，データを記憶するだけの受動的な要素である“**記憶モジュール**”，記憶モジュールの内容に対する演算処理を実行する“**演算モジュール**”，及び，チップの制御動作を専用に実行する“**制御モジュール**”によって構成される．このうち，演算モジュールと制御モジュールは何等かの演算・制御動作を実行する能動的な構成要素であり，これらをまとめて“**機能モジュール**”と呼ぶことにする．各記憶モジュールと機能モジュールには各々**レジスタ番号**（Register Number: RN）と**モジュール番号**（Module Number: MN）が割り当てられ，この値によってチップ内部での構成要素が識別される．

また，各機能モジュールはチップ全体の動作の中で様々な動作を担当することができ，起動時に指示される**起動条件**（Firing Condition: FC）によって，そのときの詳細な動作内容（対象データの選択，演算の種類等）が規定される．



図Ⅲ－３　分散制御アーキテクチャの論理構造

（２）データの流れ

データを記憶するための専用の構成要素として記憶モジュールを導入したため，本分散制御アーキテクチャにおける通常の演算モジュールは内部状態をもたないことを原則としている．（後述するように，ＲＯＭ，ＲＡＭ等を演算モジュールとして実現する場合は，演算モジュールが内部状態をもつことがある．）従って，記憶モジュールに格納されているデータは，演算モジュールに送られて何等かの演算処理を受け，演算結果のデータは記憶モジュールに返送されて保持されるものとして取り扱われる．

各演算モジュールにおける高速動作と並列動作への考慮から，演算モジュールの入力信号線，即ち，記憶モジュールから演算モジュールへの全ての配線は個別の専用配線で結ばれ，演算モジュールの前段に置かれた入力セレクタによって必要なデータを選択する方式を採用している．一方，演算モジュールから記憶モジュールへの書込み動作の発生頻度とチップ面積への影響を考慮して，演算モジュールで生成される演算結果は“**データバス**”と呼ぶ１本の共通バスを経由して記憶モジュールに返送する方式を採用している．従って，データバスは，転送されるデータ自身のためのデータ信号線と，これが格納される記憶モジュールのレジスタ番号を伝える信号線とを含んでいる．

（３）制御の流れ

本分散制御アーキテクチャにおける構成要素は前述の自己同期システムと同様の方式で動作する．即ち，各機能モジュール（演算モジュールと制御モジュール）は他の機能モジュールから特定の起動条件を伴って起動され，この起動条件に応じた何等かの演算・制御動作を実行し，この動作の完了後，特定の起動条件を指示して次の機能モジュールを起動する．このように各機能モジュールが他の機能モジュールを次々に起動していくことにより，チップ全体としての一連の動作が実現される．機能モジュールを起動するときに伝えられるモジュール番号と起動条件の組を“**起動情報**”（Module Number + Firing Condition: MNFC）と呼ぶことにする．この起動情報も，チップ面積への影響を考慮して，“**制御バス**”と呼ぶ１本の共通バスを経由して転送する方式を採用している．従って，制御バスは，起動するモジュールのモジュール番号を伝える信号線と，起動条件を伝える信号線を含んでいる．

本節の最初で示した一般的な自己同期要素では，ある機能モジュールを起動後，そのモジュールの動作の完了を待ってから，次の起動を受け入れるように動作するが，本アーキテクチャでは，起動が受け付けられた時点で起動したほうの構成要素はフリーになる方式を採用している．この結果，起動された側の機能モジュールには，動作に必要な全てのデータを自モジュール内部に保持する機構が必要となるものの，一般的な自己同期要素における動作よりも大幅な高速動作が可能となる．

（４）チップ外部との通信方式

ＶＬＳＩチップにおける外部との通信方式に関しては，通常，対象の応用システム毎に異なった構成（入出力ポート数，ポートのビット幅，入出力の区別等）を必要とするが，本アーキテクチャでは，個別の応用システムによらない一般的な通信方式として，２本の共通バス（データバスと制御バス）を外部に直接引き出す方式を採用している．このデータバスを利用して，チップの実行に必要なデータを外部から設定したり，実行結果をチップの外部に取り出したりすることが可能になる．また，制御バスも外部に引き出されていることから，本アーキテクチャに基づく複数のチップ間で，データの送受や機能モジュールの起動をチップ内部と同様の方法で行える．

さらに，チップの外部から最初の機能モジュールを起動することによって，このチップの動作が開始され，動作の終了を意味する“STOP モジュール”の起動を監視することによって，チップの動作終了を検知することができる．チップ外部に引き出された２本の共通バスによるデータ転送，及び，動作の開始と終了検知の様子を**図Ⅲ－４**に示す．

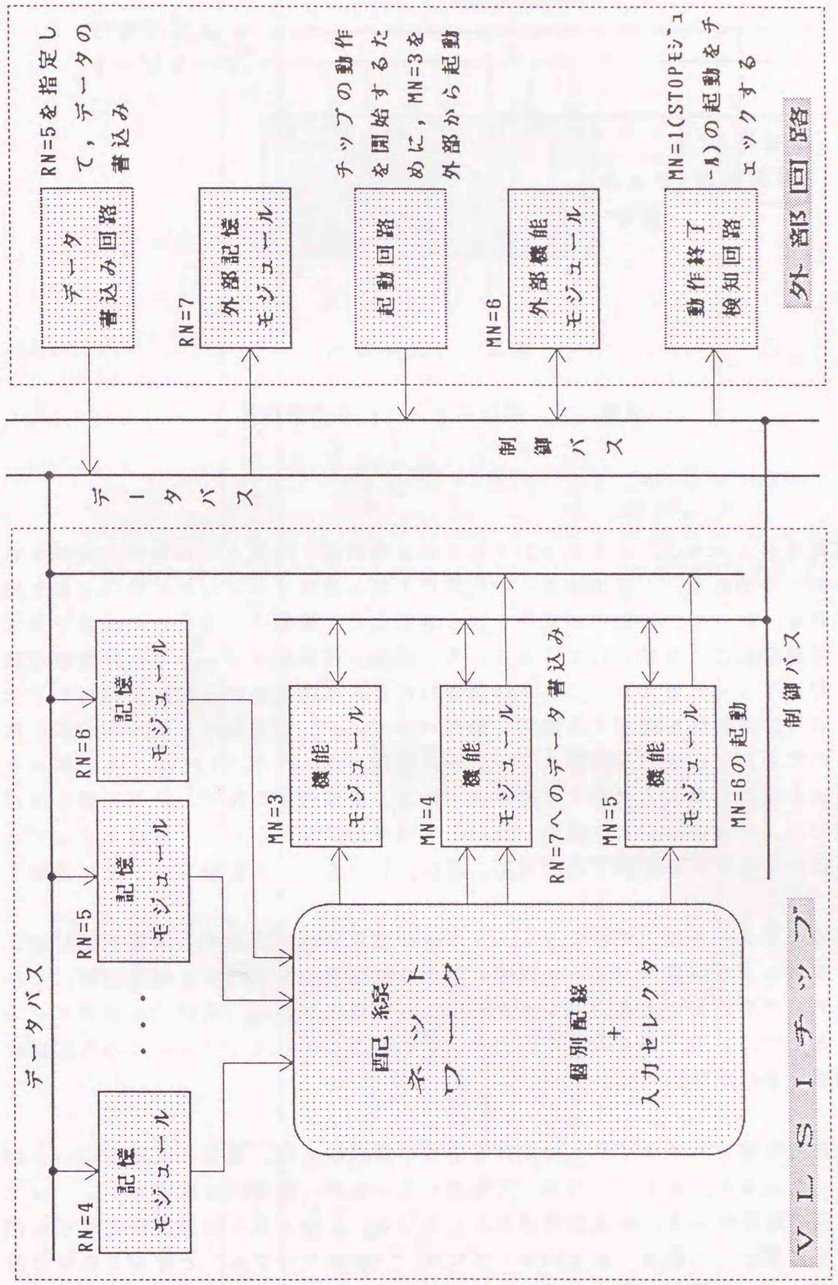
２．３　構成要素の詳細と実現方式

本節では，ハードウェア仕様記述言語：ＩＳＰＳによるＶＬＳＩチップの動作記述と，分散制御アーキテクチャに基づくＶＬＳＩチップの構成要素との対応関係を明らかにするとともに，本アーキテクチャにおける構成要素の詳細とその実現方式について述べる．

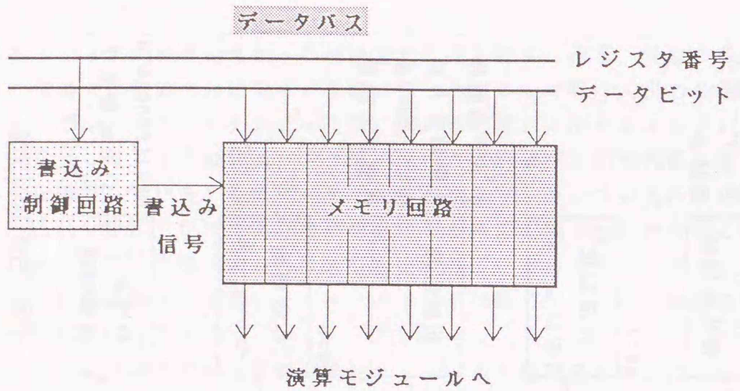
（１）記憶モジュール

本分散制御アーキテクチャにおける記憶モジュールは，ＩＳＰＳ記述おける記憶要素に対応するものであり，ユーザが記述した１個の記憶要素は，本アーキテクチャに基づくＶＬＳＩチップの中で１個の記憶モジュールとして実現される．本アーキテクチャでは実現上の容易さから，記憶モジュールとして実現される記憶要素の構造をレジスタやフリップフロップ等のようなビット構造に限定している．但し，ＲＯＭやＲＡＭ等の語構造の記憶要素は内部に記憶回路を含む演算モジュールとして実現することができる．

記憶モジュールは，１ビットのＤフリップ・フロップをビット数だけ並べたメモリ回路と，データバス中のレジスタ番号をデコードし，自レジスタ番号と一致したときにデータバス中のデータ信号の取り込みを制御する書込み制御回路から構成される．記憶モジュールの内部構成を**図Ⅲ－５**に示す．



図Ⅲ－４　チップ外部との通信方式



図Ⅲ－5　記憶モジュールの内部構成

（2）演算モジュール

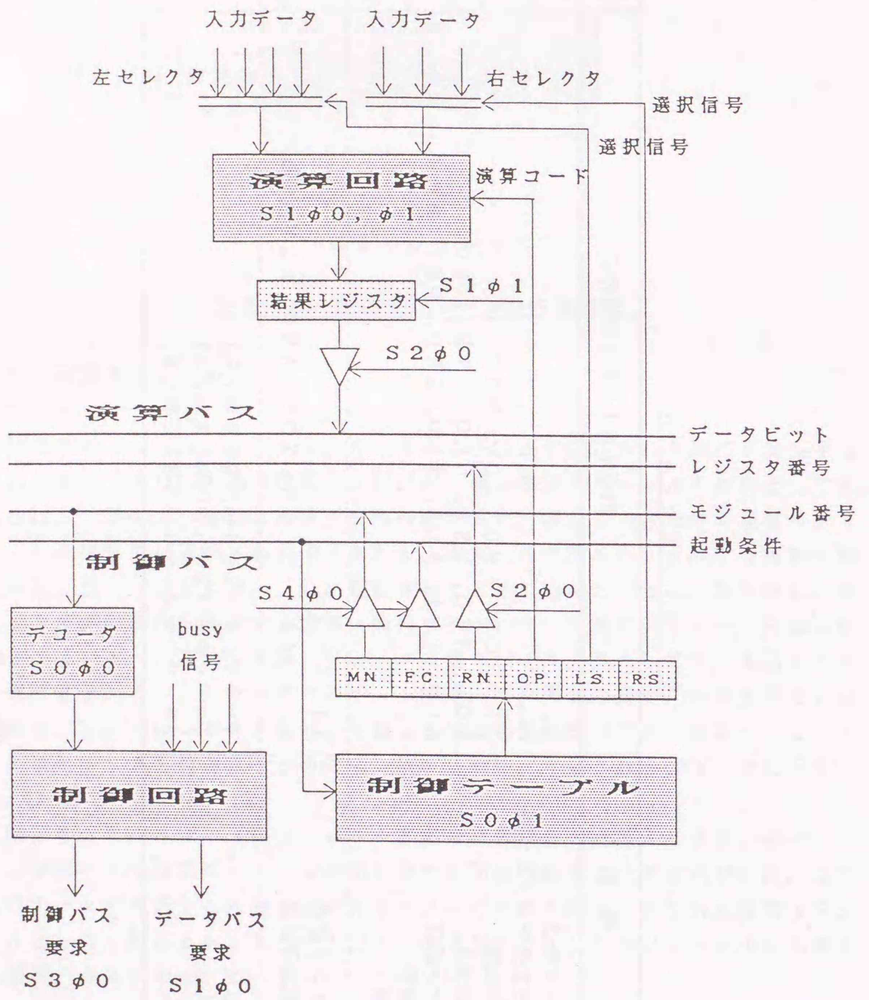
演算モジュールは，ＩＳＰＳ記述における演算式で記述される動作を実現するためのものであり，１個の演算子の記述が１個の演算モジュールの動作として実現される．従って，複数の演算子を含む演算式は，演算子の数だけの演算モジュールの連続動作に展開されて実現される．通常，各演算モジュールには複数の動作が割り当てられるため，これが起動されたときに指示される起動条件毎に，そのときの詳細動作を規定する情報（**動作パラメータ**）が演算モジュール内部に格納されている．代表的な演算モジュールの動作パラメータとしては，入力データを選択するための入力セレクタのポート指定，演算種類の指定，結果を書き込む記憶モジュールのレジスタ番号，及び，１組の起動情報がある．演算モジュールの種類とＩＳＰＳの演算子との対応，及び，動作クロック数を**表Ⅲ－２**に示す．

演算モジュールは，そのモジュール本来の演算機能を実現する演算回路に加えて，制御バスや演算モジュール内部に対する制御機能を実現する制御回路，及び，入力データを選択する入力セレクタ等によって構成される．典型的な演算モジュールとして，左右とも４入力の入力セレクタを含むＡＬＵモジュールの内部構成を**図Ⅲ－６**に示す．

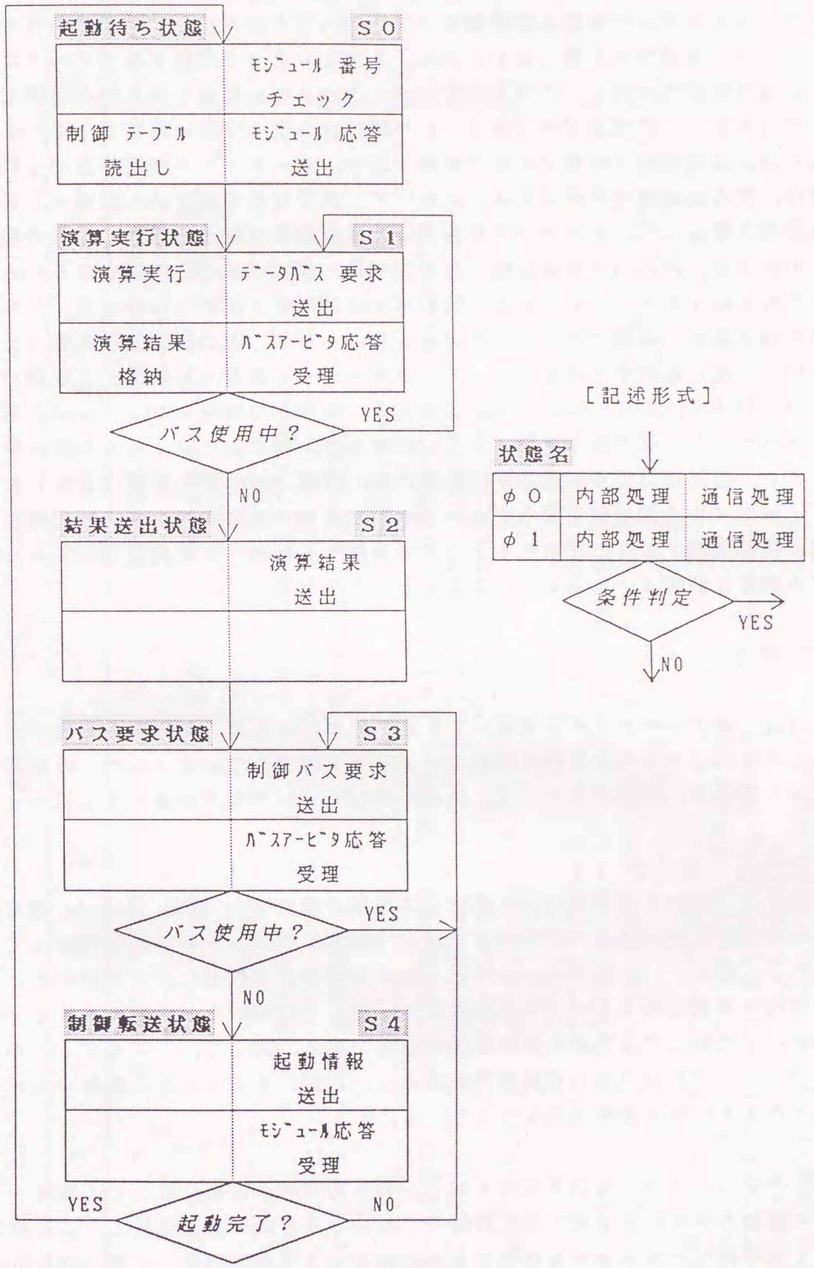
本分散制御アーキテクチャにおける全ての構成要素は，重なり区間のない２相クロック（φ０，φ１）に同期して動作しているが，典型的な演算モジュールでは１回の起動時に５回の状態遷移を行っている．このときの状態遷移の様子を**図Ⅲ－７**に示す．（**図Ⅲ－６**と**図Ⅲ－７**では，“制御テーブル”と呼ぶＲＯＭに演算モジュールの動作パラメータを格納した方式を想定している．）

表Ⅲ－２　演算モジュール

モジュール名	機　能	対応するＩＳＰＳの演算子	動作クロック数
ＬＴＲＡＮＳ	データ転送	＝	5
ＡＴＲＡＮＳ	符号拡張データの転送	＜＝	5
ＬＯＧＩＣ	論理演算	ＯＲ，ＸＯＲ，ＡＮＤ，ＥＱＶ，ＮＯＴ	5
ＣＯＭＰ	比較演算	ＥＱＬ，ＮＥＱ，ＬＳＳ，ＬＥＱ，ＧＴＲ，ＧＥＱ	5
ＳＨＩＦＴ	シフト演算	ＳＬ０，ＳＬ１，ＳＬＲ，ＳＬＤ，ＳＲ０，ＳＲ１，ＳＲＲ，ＳＲＤ	5
ＡＤＤＥＲ	加減算	＋，－	5
ＭＵＬＴＩ	乗算	＊	5
ＣＮＴ	カウンタ動作	＋１，－１	5
ＡＬＵ	算術論理演算	＋，－，＋１，－１，ＯＲ，ＸＯＲ，ＡＮＤ，ＥＱＶ，ＮＯＴ	5
ＲＯＭ	ＲＯＭ	（ユーザがモジュールの内容を定義する）	5
ＲＡＭ	ＲＡＭ	（ユーザがモジュールの内容を定義する）	5
ＰＬＡ	ＰＬＡ	（ユーザがモジュールの内容を定義する）	5



図Ⅲ－６　演算モジュールの内部構成



［記述形式］

図Ⅲ－７　演算モジュールの状態遷移

演算モジュールが何も動作していない起動待ち状態（ＳＯ）では，制御バス中のモジュール番号を検査し，これが自モジュールを指定している時は，同時に送られている起動条件に対応した制御テーブルの内容を読み出して，演算動作を開始する．次の演算実行状態（Ｓ１）では，入力セレクタで選択されたデータに対しての演算処理を実行し，これと同時にデータバスの使用権を得るための要求信号を送出する．この要求が受け付けられた時には，次の状態に遷移するが，そうでない場合は再度同じ状態での動作を繰り返す．データバスの使用が認められたときは，次の結果送出状態（Ｓ２）において，演算結果を書き込む記憶モジュールの番号を指定して，データバスを利用した演算結果の転送を行う．以上の演算動作の終了後，次のバス要求状態（Ｓ３）では，制御バスの使用権を得るための要求信号を送出する．この場合も，制御バスの使用権が認められるまで，Ｓ３の動作を繰り返す．制御バスの使用が認められたときは，次の制御転送状態（Ｓ４）において，次に起動する機能モジュールのモジュール番号と起動条件を制御バスを用いて転送し，このモジュールを起動する．相手先の機能モジュールが正常に起動された場合は起動待ち状態（Ｓ０）に移るが，相手先モジュールが既に動作中であり，起動拒否信号が返送された場合は，再度，バス要求状態（Ｓ３）に戻って，制御バスの使用権を要求する．（起動する側の最終状態：Ｓ４と起動される側の初期状態：Ｓ０は同じタイミングで動作するため，この演算モジュールの１回の動作は最短４クロックで完了すると考えられる．）

（３）制御モジュール

本分散制御アーキテクチャでは，ＩＳＰＳにおける各種の動作制御構文を実現するために，以下に示す５個の制御モジュールを用意している．これらの制御モジュールの機能，動作パラメータ，及び，動作クロック数を表Ⅲ－３に示す．

・条件動作

ＩＳＰＳにおける条件動作を記述する２種類の構文（if 構文，decode 構文）による動作記述は，各々 **IF** モジュールと **DECODE** モジュールの動作によって実現される．即ち，ＩＦモジュールは１ビットのデータを入力し，この値に応じて次に起動する機能モジュールを選択する．一方，DECODE モジュールはｎビット長のデータに対して2^n通りの分岐先が指定できる．従って，ＩＦモジュールの動作パラメータには２組の起動情報を含み，DECODE モジュールの動作パラメータには最大2^n組の起動情報を含んでいる．

ＩＦモジュールは，選択された１ビットの入力データの値に応じて，制御テーブルに格納されている２組の起動情報のうちのどちらか一方を選択し，この値によって次の機能モジュールを起動する回路によって構成される．一方，DECODE モジュールは，選択された入力データの値と起動情報が格納されている起動情報テーブルのベース番地を加算して，多数の起動情報の中から１個の起動情報を選

表Ⅲ－３　制御モジュール

モジュール名	機　　能	動作パラメータ	動作クロック数
ＩＦ	条件分岐（２方向）	入力セレクタのポート番号*1，２組の起動情報	3
ＤＥＣＯＤＥ	デコード（2^n方向）	入力セレクタのポート番号*1，最大2^n組の起動情報 （ｎ：データのビット幅）	3
ＦＯＲＫ	並列動作の起動	ｎ組の起動情報（ｎ：並列動作数）	２ｎ＋１
ＳＹＮＣ	並列動作の同期	並列動作数ｎ，１組の起動情報	２：同期待ち ４：同期完了
ＰＲＯ－ＣＯＮＴ	以下に示す動作要素の制御動作	全てに共通：動作要素番号*2，動作要素の型*3	
① ACTIVATE	手続きの呼出し	２組の起動情報（飛び先，戻り先）	3
	プロセスの起動	２組の起動情報（飛び先，次）	5
② LEAVE	手続きからの復帰	（なし）	3
	プロセスの終了	（なし）	2
	ラベル付き動作の終了	１組の起動情報	3
③ RESTART	動作要素の再スタート	１組の起動情報	3
④ TESTSET	動作要素の動作状況チェック	記憶モジュール番号*4，１組の起動情報	5

起動情報：モジュール番号と起動条件の組（MNFC）
＊１：入力データを選択するためのセレクタが組み込まれている
＊２：すべての動作要素は一意な番号が付与され，動作状況が管理される
＊３：手続き，プロセス，ラベル付き動作の区別
＊４：動作状況（０／１の値）が記憶モジュールに格納される

択し，この値によって次の機能モジュールを起動する回路によって構成される．これらの制御モジュールの内部構成を**図Ⅲ－８（ａ），（ｂ）**に示す．

・並列動作

“；”で結合される並列動作の記述は **FORK モジュール**と **SYNC モジュール**の動作によって実現される．即ち，FORK モジュールは１回の動作で複数の機能モジュールを連続的に起動し，SYNC モジュールは，指定した全てのモジュールから起動されるまで，次の機能モジュールの起動を待ち合わせる．従って，FORK モジュールによって起動された複数の動作シーケンスは，本分散制御アーキテクチャの制御構造によって並行に実行される．FORK モジュールの動作パラメータにはｎ組の起動情報を含み，SYNC モジュールの動作パラメータには同期する並列動作の個数ｎと１組の起動情報を含んでいる．

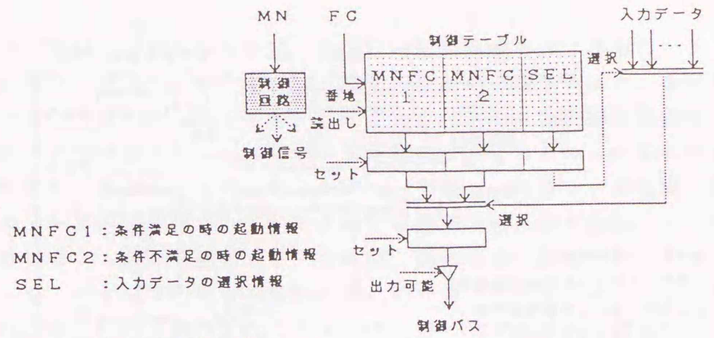
FORK モジュールは，制御テーブルに格納されている起動情報テーブルのベース番地によって，これに格納されている起動情報を次々に読み出し，起動情報の最後を示す終了コードが検知されるまで，複数の機能モジュールを連続的に起動する回路によって構成される．一方，SYNC モジュールは，これが起動される毎に完了した並列動作数を保持するレジスタの内容（初期値：ゼロ）をインクリメントし，この値が待ち合わせる並列動作数と一致したときに次の機能モジュールを起動する同期メカニズムによって構成される．これらの制御モジュールの内部構成を**図Ⅲ－８（ｃ），（ｄ）**に示す．

・動作要素の制御

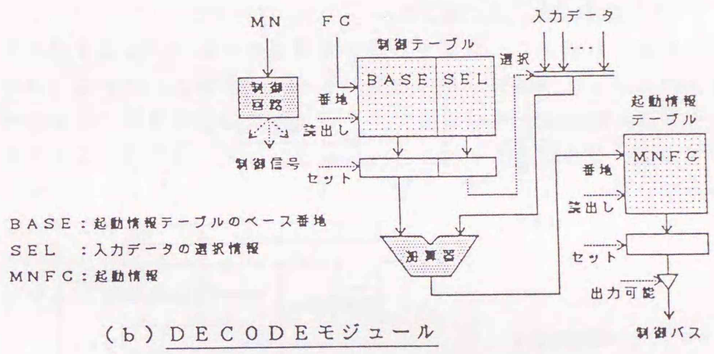
ＩＳＰＳにおける手続きやプロセス等の動作要素，及び，名前付き動作に対する各種の制御動作を実現するために，本分散制御アーキテクチャでは **PRO-CONT モジュール**が用意されている．このモジュールは，物理的には１個のモジュールとして実現されるが，論理的には以下に示す４個の制御モジュールの集合体と考えられる．

① **ACTIVATE モジュール**：動作要素の呼出しや起動の記述を実現する．
② **LEAVE モジュール**：leave 構文に対応して，動作要素や名前付き動作を終了する．
③ **RESTART モジュール**：restart 構文に対応して，動作要素や名前付き動作を再スタートする．
④ **TESTSET モジュール**：排他的動作要素の起動のために，動作要素の動作状況をチェックする．

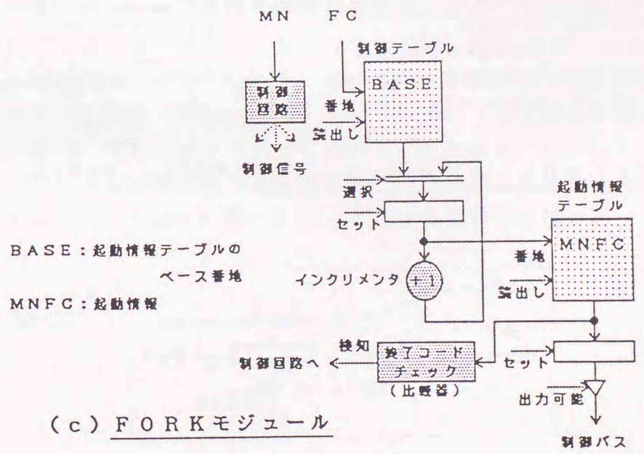
PRO-CONT モジュールは，起動条件毎の動作パラメータを保持した制御テーブルに加えて，動作要素が動作中かどうかを示す“動作中フラグ”と復帰時の起動情報を格納するための“復帰レジスタ”（手続きの場合に使われる）とからなる



（ａ）ＩＦモジュール

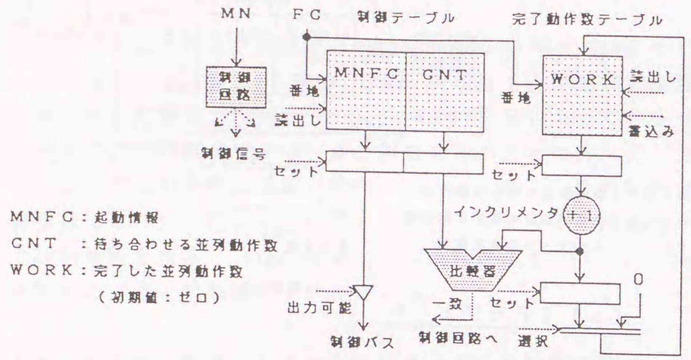


（ｂ）ＤＥＣＯＤＥモジュール

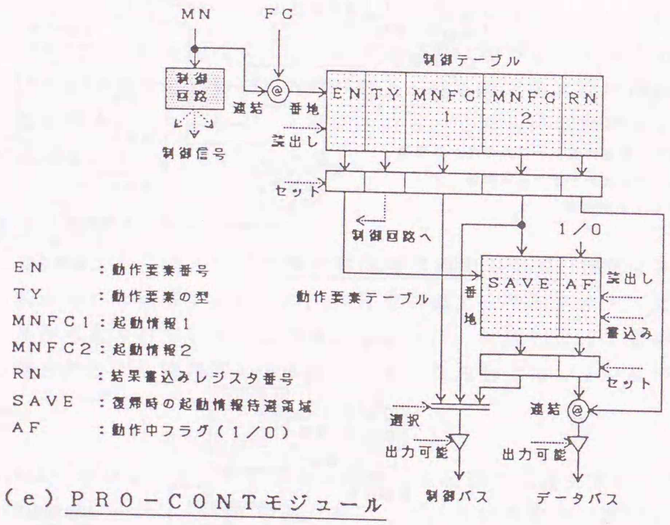


（ｃ）ＦＯＲＫモジュール

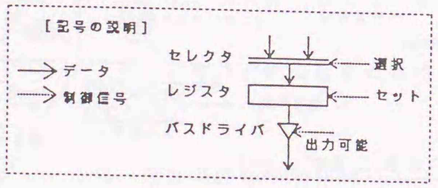
図Ⅲ－８　制御モジュールの内部構成（つづく）



（d）SYNCモジュール



（e）PRO－CONTモジュール



図Ⅲ－8　制御モジュールの内部構成

“動作要素テーブル”と，このテーブルを操作する制御回路によって実現できる．このモジュールの内部構成を図Ⅲ－8（e）に示す．

手続き／関数の呼出しの場合は，復帰時の起動情報を動作要素テーブルに格納するとともに，呼び出された手続き／関数の最初の機能モジュールを起動する．一方，プロセスの起動に関しては，起動されたプロセスの最初の機能モジュールと起動側の次の機能モジュールの両方を起動する．これらのいずれの場合も対応する動作要素の動作中フラグをセットする．手続き／関数からの復帰の場合は，動作要素テーブルに格納されている復帰時の起動情報を取り出し，この値によって次の機能モジュールを起動する．この時，動作要素の動作中フラグをリセットする．一方，プロセスの終了の場合は単に動作中フラグをリセットするだけである．また，ラベル付き動作の終了の場合は，この動作の直後の機能モジュールを起動する．

RESTART モジュールの動作は，動作要素あるいは名前付き動作の最初の機能モジュールを再起動する．TESTSET モジュールの動作は，動作要素テーブルの動作中フラグの値を記憶モジュールに転送するとともに，この時点で動作中フラグをセットする．従って，このモジュールの動作の後で記憶モジュールの内容を検査することにより，対象の動作要素が動作中であるかどうかを調べることができ，かつ，動作要素が動作中でない（動作中フラグがセットされていない）状態で，最初にこの問合せを行った1個の動作シーケンスのみが，この動作要素の使用権を得ることができる．

（4）モジュール動作の例

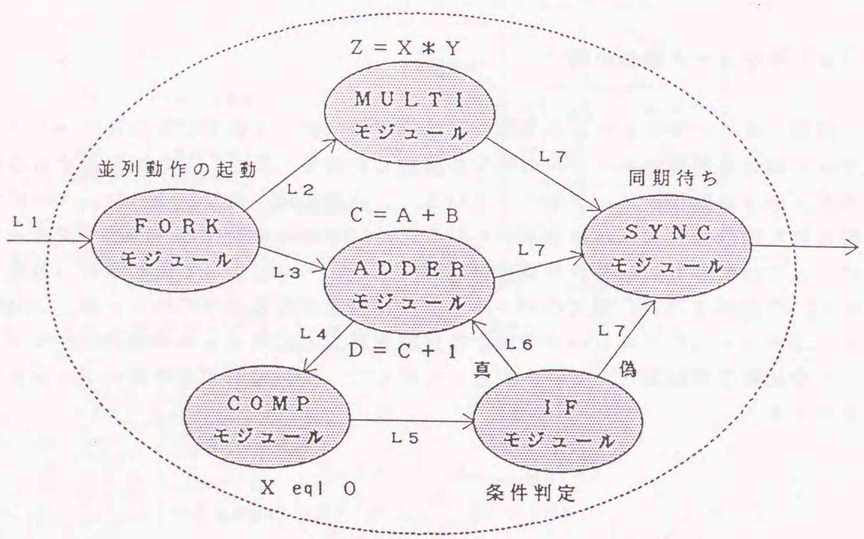
図Ⅲ－9は，ＩＳＰＳによる動作記述の簡単な例，これを分散制御アーキテクチャにおける機能モジュールの動作に展開した結果，及び，これを実現する機能モジュールの動作イメージを示している．この例では，乗算と begin － end で挟まれた節の並列動作が指定されているため，これらの動作は FORK モジュールによって起動され，各動作の最後で SYNC モジュールによって同期が取られる．また，この例では if 構文のオペランドに比較式が記述されているため，COMP モジュールと IF モジュールの動作に展開され，IF モジュールは COMP モジュールの比較演算結果（1ビットの値）に応じて，次に起動する機能モジュールを変更する．

```
Z = X * Y;          ! ";" parallel action
begin
    C = A + B next
    if X eql 0 => D = C + 1
end next
```

（ａ）ＩＳＰＳによる動作記述

```
L1  FORK                          -> L2 and L3
L2  Z = MULTI (X, Y)              -> L7
L3  C = ADDER(0¥add, A, B)        -> L4
L4  W = COMP(0¥eql, X, 0)         -> L5
L5  IF (W)                        -> L6 or L7
L6  D = ADDER(0¥add, C, 1)        -> L7
L7  SYNC                          ->
```

（ｂ）モジュール動作



（ｃ）機能モジュールの動作イメージ

図Ⅲ－９　分散制御アーキテクチャにけるモジュール動作の例

３．動作記述型シリコン・コンパイラ

３．１　全体構成

本章では，ＶＬＳＩチップの仕様記述形式であるＩＳＰＳ記述から，前章で示した分散制御アーキテクチャに基づくＶＬＳＩチップへの変換を行うシリコン・コンパイラの詳細について述べる．

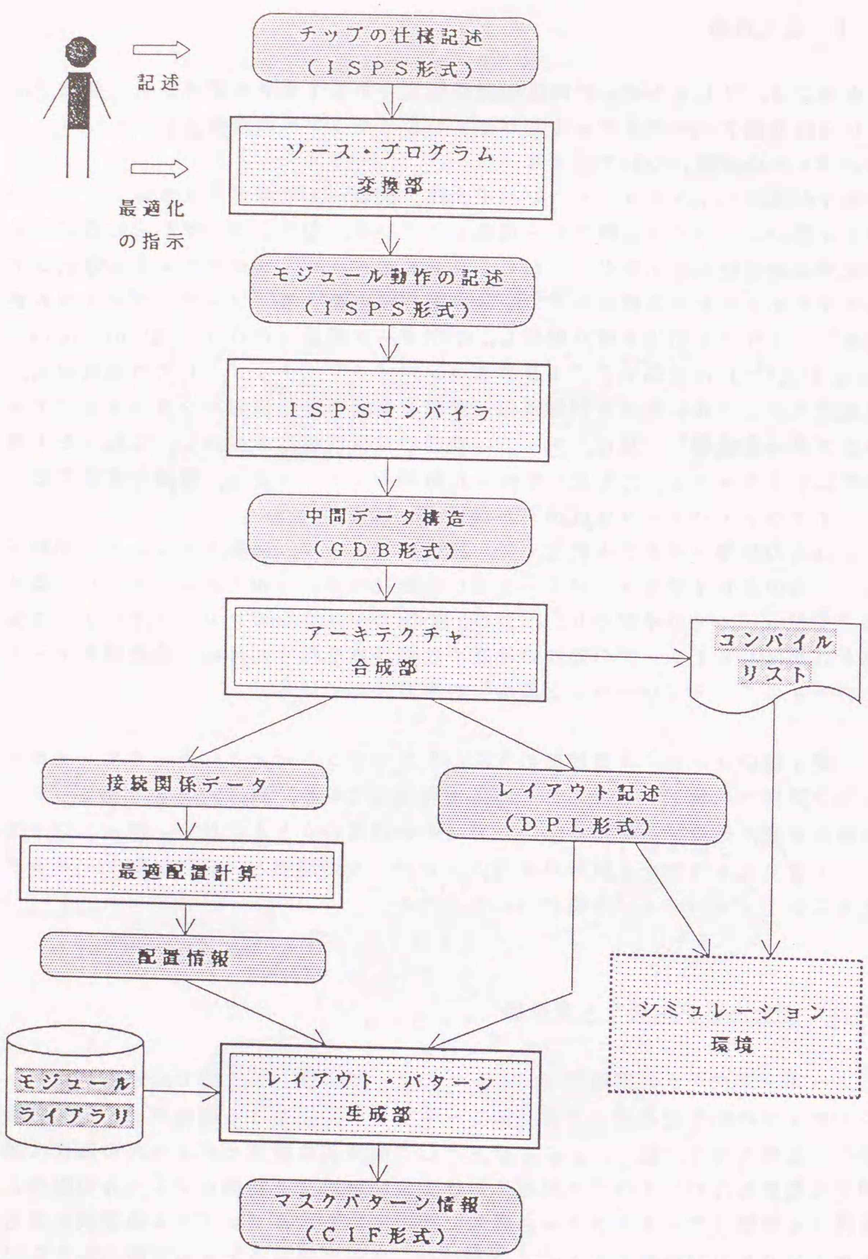
我々が試作したシリコン・コンパイラは，**図Ⅲ－１０**に示すように，大きく分けて４個のコンパイル処理部から構成されている．即ち，ユーザからの構成要素に関する最適化の指示を受けながらＩＳＰＳのソース・プログラムを分散制御アーキテクチャにおける機能モジュールの動作に変換する“**ソース・プログラム変換部**”，ＩＳＰＳ記述を構文解析し，中間データ構造（ＧＤＢ：Global Data Base 形式[28]）に展開する“**ＩＳＰＳコンパイラ**”，チップとしての構成要素，各機能モジュールにおける制御情報，構成要素間の接続情報等を抽出する“**アーキテクチャ合成部**”，及び，チップの物理的な構成要素を生成し，これらを１個のＶＬＳＩチップとして配置・配線した結果のマスクパターン情報を生成する“**レイアウト・パターン生成部**”から構成される．

これらの処理プログラムに加えて，記憶モジュール，演算モジュール，制御モジュールの各レイアウト・パターンのひな型（モジュール・コンパイラ）を集めた“**モジュール・ライブラリ**”，及び，上記のシリコン・コンパイラによって生成されたＶＬＳＩチップの動作の検証や性能評価を行うために，分散制御アーキテクチャの“**シミュレーション環境**”が用意されている．

上記４個のコンパイル処理部のうち，ＩＳＰＳコンパイラはカーネギーメロン大学で開発されたプログラムをそのまま利用しており，処理内容も中間コードへの単なる変換処理であるため，ここではその内容に立ち入らない．従って以下では，ＩＳＰＳコンパイラ以外の３個のコンパイル処理部と，分散制御アーキテクチャのシミュレーション環境について述べる．

３．２　ソース・プログラム変換部

ソース・プログラム変換部は，ユーザによってＩＳＰＳ言語で記述されたＶＬＳＩチップの動作記述から分散制御アーキテクチャにおける機能モジュールの動作への変換を行う．但し，ここでは主として演算式を演算モジュールの動作に展開する処理を行い，ＩＳＰＳ記述における各制御構文を制御モジュールの動作に展開する処理はアーキテクチャ合成部で行っている．本プログラムの変換結果もまたＩＳＰＳの記述形式をとっているため，ユーザはこのときの変換内容を容易に理解できるとともに，ＩＳＰＳシミュレータによって，この変換結果に対する動作の検証も行える．



図Ⅲ－１０　シリコン・コンパイラの全体構成

従って本プログラムは，あるＩＳＰＳ記述から別のＩＳＰＳ記述への等価変換プログラムと考えられるが，このような言語記述間の機械的な変換では，ユーザの意図しない記述が生成されたり，ＶＬＳＩチップの動作としての最適化の可能性が残っていることが多い．このため本プログラムは，ＩＳＰＳのソース・プログラムを編集できるスクリーン・エディタに，モジュール動作への変換アルゴリズムと記述の最適化を支援する機能を組み込んだ“**ＩＳＰＳ構造化エディタ**”の形式で実現されている．

表Ⅲ－４　ＩＳＰＳ構造化エディタのコマンド一覧表

コマンド	機　能	応答
［制御コマンド］		
ＩＮ　ｘｘｘ	ファイルｘｘｘの読込み	－
ＯＵＴ	ファイルへの書込み	－
ＥＸＩＴ	プログラムの終了	－
［編集コマンド］		
Ｔ	ファイルの最初を表示	－
Ｂ	ファイルの最後を表示	－
ＮＰ　（ｎ）	次ページ，あるいは，ｎページ先を表示	－
ＰＰ　（ｎ）	前ページ，あるいは，ｎページ前を表示	－
ＮＬ　（ｎ）	カーソルを次行，あるいは，ｎ行先に移動	－
ＰＬ　（ｎ）	カーソルを前行，あるいは，ｎ行前に移動	－
Ｓ　（ｘｘｘ）	文字列ｘｘｘを前方向に検索	－
Ｒ　（ｘｘｘ）	文字列ｘｘｘを後方向に検索	－
［変換コマンド］		
ＡＵＴＯ	自動変換（一括モード）	－
ＡＵＴＯＱ	自動変換（問合せモード）	Ｙ／Ｎ
ＲＥ　ｘｘｘ　ｙｙｙ	演算モジュール名／一時レジスタ名の変更	Ｙ／Ｎ
ＳＰ　ｘｘｘ	演算モジュールの分離	Ａ
ＳＰ－Ｉ	IF モジュールの分離	Ａ
ＳＰ－Ｄ	DECODE モジュールの分離	Ａ
ＳＴ－Ａ	ＡＬＵモジュールへの変換	Ｙ／Ｎ
ＳＴ－Ｃ	カウンタ・モジュールへの変換	Ｙ／Ｎ
［表示コマンド］		
ＤＲ	ユーザが記述した記憶要素の表示	－
ＤＭ	ユーザが記述した演算モジュールの表示	－
ＧＲ	システムが生成した一時レジスタの表示	－
ＧＭ	システムが生成した演算モジュールの表示	－

※ Ｙ／Ｎ：　yes/no で応答する
　Ａ：　　アルファベット１文字で応答する

（1）ＩＳＰＳ構造化エディタのコマンド

ＩＳＰＳ構造化エディタには，通常のスクリーン・エディタと同様に，ＩＳＰＳ記述に対する表示，記述，修正，検索等の基本的な編集機能に加えて，シリコン・コンパイラの前段階処理としてのモジュール動作への変換機能が組み込まれている．即ち，このエディタには，入力されたＩＳＰＳ記述全体に対して一律の変換規則を適用するコマンドや，この変換結果をユーザが判断して，機能モジュール／記憶モジュールの統合，並置等を支援する幾つかの最適化に関するコマンドが用意されている．本エディタのコマンド一覧表を**表Ⅲ－４**に示す．

ＩＳＰＳ構造化エディタで利用できるコマンドは，ファイルの入出力とプログラムの終了を指示する制御コマンド，スクリーン上に表示する内容を指示したり，特定の文字列を検索する編集コマンド，機能モジュールの動作への変換や最適化を指示する変換コマンド，及び，記憶モジュールや演算モジュールの記述／変換状況を表示する表示コマンドの４種類に分けられる．以下では，このうちの変換コマンドの動作について説明する．

（2）変換アルゴリズム

以下の例に示すように，自動変換コマンド（AUTO, AUTOQ コマンド）に組み込まれている変換規則（**表Ⅲ－２**に示したＩＳＰＳ演算子と演算モジュールとの対応関係）によって，ＩＳＰＳ記述の演算式は特定の演算モジュールの動作に変換される．各演算モジュールのビット幅は，オペランドとなるデータのビット幅から自動的に計算される．これと同時に，変換結果に現れる演算モジュール自身の記述（下記参照）が，ユーザが記述したソース・プログラムの最後に付加される．

［自動変換の例］

```
A = B<0:7>              →   A = LTRANS.8 (B<0:7>)
A<0:3> and B<0:3>       →   LOGIC.4 (2¥and, A<0:3>, B<0:37>)
A<0:15> eql 0           →   COMP.16 (0¥eql, A<0:15>, 0)
A<0:7> - B<0:7>         →   ADDER.8 (1¥sub, A<0:7>, B<0:7>)
A<0:15> sr0 3           →   SHIFT.16 (5¥sr0, A<0:15>, 3)
```

上の例において，“LOGIC.4”等 の記述は演算モジュールの名前であり，演算モジュールのタイプとそのビット幅を意味している．また，“2¥and”等 の記述は，演算モジュールに対して演算の種類を指定するパラメータであり，この場合は，論理積を意味する数値２を与えている．（“¥”は“別名”を意味する区切り記号である．）

［演算モジュールの記述例］

```
module ADDER.8 (OP<1:4>, D1<1:8>, D2<1:8>) <1:9> := begin
    decode OP => begin
        0¥add   := ADDER.8 = D1 + D2,
        1¥sub   := ADDER.8 = D1 - D2
    end
end,
```

複数の演算子を含む式は，一時レジスタ（WRG.k）を使用して，演算モジュールの一連の動作に展開される．例えば，

```
A = B<0:15> - C<0:7> * (D<0:7> sr0 4)
```

の記述は，一時レジスタ WRG.1<1:8> と WRG.2<1:16> を用いて，以下のように展開される．これらの一時レジスタのビット長も自動的に計算され，その宣言の記述がシステムによって自動生成される．

```
begin   WRG.1 = SHIFT.8 (5¥sr0, D<0:7>, 4) next
        WRG.2 = MULTI.8 (C<0:7>, WRG.1) next
        A = ADDER.16 (1¥sub, B<0:15>, WRG.2)   end
```

また，ＩＳＰＳの if, decode 構文ではオペランドに任意の演算式を許すが，分散制御アーキテクチャにおける IF モジュールと DECODE モジュールは１個のデータ値のみを入力する構造になっている．（さらに，IF モジュールの場合は１ビットのデータに限定される．）従って，これらの構文で演算式が使われた時には，この演算式を，演算結果を一時レジスタ（WRG.k）に格納する記述と，このレジスタの内容を参照する記述とに展開する．例えば，

```
if A<0:7> eql B<0:7> => begin ... end
```

の記述は，１ビットの一時レジスタ WRG.1 <> を利用して，以下のように展開される．

```
begin   WRG.1 = COMP.8 (0¥eql, A<0:7>, B<0:7>) next
        if WRG.1 => begin ... end   end
```

（3）最適化処理

上記の自動変換を行った結果では，ユーザの意図しない冗長な記述が残ってい

たり，ＶＬＳＩチップの動作として最適化できる可能性もあるため，ユーザはエディタの最適化に関するコマンドを用いて記述の部分的な変更を指示することができる．以下に，本構造化エディタに組み込まれている最適化処理の内容を示す．

・演算モジュール／記憶モジュールの統合

上記の自動変換では，演算の種類が同一であってもオペランドのビット幅が異なれば，新しい演算モジュールが生成されてしまう．しかし，論理演算や，上位の桁を無視できる算術演算等では，通常，ビット幅の大きな演算モジュールによって，ビット幅の小さな演算も正しく実行することができる．また，複数の演算子を含む演算式の展開や if 構文と decode 構文の展開において，次々に新しい一時レジスタが生成されてしまう．この場合も，ビット幅の小さな多くの記憶モジュールをビット幅の大きな少数の記憶モジュールに統合できる可能性が高い．但し，並列動作の可能性から機械的な変換は行えず，統合処理を適用するモジュールの選択はユーザの判断に委ねる必要がある．

このような演算モジュールと記憶モジュールの統合を支援する機能として，本エディタでは，演算モジュールや一時レジスタの名前を指定して，これを別の名前に変換するかどうかをユーザに問い合わせる RE （rename）コマンドが用意されている．例えば，４ビット幅の論理演算モジュールの動作を既に定義されている８ビット幅のモジュールで代用するときには，“RE LOGIC.4 LOGIC.8”のように入力すればよい．この時，全ソース・プログラムが検索され，LOGIC.4 が見つかるたびに，変換の要否がユーザに問い合わされる．

・演算モジュールの並置

演算モジュールを統合するのとは逆に，記述されたとおりの並列動作を実現するためには，同じ機能，同じビット幅の演算モジュールを複数個並置することが必要となる．そうでなければ，本来，並列に動作する記述であっても，１個の演算モジュールをシーケンシャルに利用しなければならず，性能の低下を引き起こしてしまう可能性がある．但し，演算モジュールの並置はそれだけチップ面積を多く費やすため，必ずしも望ましい方向でない可能性もあり，この処理の要否も最終的にはユーザの判断に委ねる必要がある．

本エディタではこのために，１個の演算モジュールを複数個のモジュールに分離する SP （split） コマンドが用意されている．分離の対象となる演算モジュールを指定してこのコマンドを入力すると，ソース・プログラムの先頭からこのモジュールが検索され，これが見つかるたびに，どのように分離するかがユーザに問い合わされる．ユーザは１文字のアルファベットを応答することにより，この演算モジュールは，元のモジュール名に応答した１文字の拡張子がついた名前の新しい演算モジュールとして分離され，区別される．例えば，８ビットの論理演算モジュール（LOGIC.8）を３個に分離し，これを A，B，C で識別するときは，システム側からの問合せに対して，A，B，C の何れかの１文字を応答する．この結果，LOGIC.8 は LOGIC.8.A，LOGIC.8.B，LOGIC.8.C の３個に分離される．

・制御モジュールの並置

IF モジュールと DECODE モジュールに関しても，並列動作による高速化のためにモジュールを並置する効果が期待できる．但し，この時点でのＩＳＰＳ記述においては分散制御アーキテクチャにおける制御モジュールが陽に表現されていないため，if 構文と decode 構文の修飾子（qualifier，{ } の間に記述する）として，モジュールの並置を指示しておき，後述のアーキテクチャ合成部での展開時に，実際の並置の処理を行う方式を採用している．

本エディタでは，この並置機能のために SP-I （split if）コマンドと SP-D （split decode）コマンドが用意されており，上記の演算モジュールの並置と同様の操作によって，制御モジュールの並置を指定できる．例えば，並列性を考慮した結果，２個の IF モジュールを並置する効果が大きいならば， SP-I コマンドによって，対象となる if 構文を if {A} と if {B} の記述に変換しておく．この結果，後続のアーキテクチャ合成部の変換処理によって，これらの記述は IF.A モジュールと IF.B モジュールの動作に展開される．

・ＡＬＵモジュールとカウンタ・モジュールへの変換

今回試作したシリコン・コンパイラでは，ＡＬＵモジュールとカウンタ・モジュールを用意してあり，**表Ⅲ－２**に示したように，ＡＬＵモジュールは各種の算術論理演算機能を，カウンタ・モジュールは引数に対して±１する機能を有している．従って，下記に示す変換ルールを適用して，データ転送モジュール，論理演算モジュール，算術演算モジュール等の記述を，これらのモジュールの記述に変換することができる．

本エディタでは，この変換を行うために，ST-A （special translation into ALU）コマンドと，ST-C （special translation into CNT）コマンドが用意されている．これらのコマンドの実行時にも，変換の要否がユーザに問い合わされる．

［ＡＬＵモジュール］

```
LTRANS.n (D1)          →  ALU.n (0¥or, D1, 0)
LOGIC.n (0¥or, D1, D2) →  ALU.n (0¥or, D1, D2)
LOGIC.n (1¥xor, D1, D2)→  ALU.n (1¥xor, D1, D2)
LOGIC.n (2¥and, D1, D2)→  ALU.n (2¥and, D1, D2)
LOGIC.n (3¥eqv, D1, D2)→  ALU.n (3¥eqv, D1, D2)
LOGIC.n (4¥not, D1, 0) →  ALU.n (4¥not, D1, 0)
ADDER.n (0¥add, D1, D2)→  ALU.n (5¥add, D1, D2)
ADDER.n (1¥sub, D1, D2)→  ALU.n (6¥sub, D1, D2)
CNT.n (0¥inc, D1)      →  ALU.n (7¥inc, D1, 0)
CNT.n (1¥dec, D1)      →  ALU.n (8¥dec, D1, 0)
```

［カウンタ・モジュール］

```
ADDER.n (0¥add, D1,  1)   →   CNT.n (0¥inc, D1)
ADDER.n (0¥add, D1, -1)   →   CNT.n (1¥dec, D1)
ADDER.n (0¥add,  1, D2)   →   CNT.n (0¥inc, D2)
ADDER.n (0¥add, -1, D2)   →   CNT.n (1¥dec, D2)
ADDER.n (1¥sub, D1, -1)   →   CNT.n (0¥inc, D1)
ADDER.n (1¥sub, D1,  1)   →   CNT.n (1¥dec, D1)
```

3.3 アーキテクチャ合成部

（1）構成要素の抽出

特定のアーキテクチャを前提としないシリコン・コンパイラでは，ＶＬＳＩチップの動作記述から実際のチップを形成する構成要素（モジュール群）を抽出し，これらの最適な構成を決定するのは非常に時間のかかる困難な処理である．しかし，分散制御アーキテクチャを前提とする我々のシリコン・コンパイラでは，記憶モジュールと演算モジュールの構成は前述のソース・プログラム変換部の処理によって既に決定されている．即ち，記憶モジュールに関しては，ユーザによって記述された記憶要素と，ソース・プログラム変換部によって生成された一時レジスタの各々が１個の記憶モジュールに対応する．（後述するように，動作要素の引数に対応する記憶モジュール，及び，排他的動作要素の動作状況を格納するための記憶モジュールはアーキテクチャ合成部の処理で生成される．）．また，ＩＳＰＳ記述における一般的な演算式の記述は，一連の演算モジュールの動作に変換されており，チップに必要な演算モジュールの構成も既に明確になっている．

ＩＳＰＳ記述における if 構文と decode 構文に関しても，これらの構文で記述される＜評価式＞は，ソース・プログラム変換部の処理によって，評価式の値を計算する演算モジュールの一連の動作と，この結果を格納した１個の記憶要素を参照する if 構文や decode 構文の記述に展開されている．従って，ソース・プログラム変換部の変換結果としての if 構文や decode 構文の記述から，IF モジュールや DECODE モジュールを抽出することも容易である．但し，前節で示したように，並列動作のために複数の制御モジュールを利用する場合は，これらの記述にアルファベット１文字の修飾子（{A} 等）が付与されているため，この時には名前の異なる複数の IF モジュールや DECODE モジュールを生成する必要がある．

このように，ソース・プログラム変換部の出力結果から，記憶モジュール，演算モジュール，及び，IF モジュールと DECODE モジュールを抽出することは非常に容易であるが，これ以外の制御モジュールの抽出と分散制御アーキテクチャ

における動作への変換は幾分複雑である．以下，これらの制御モジュールの抽出に伴う変換処理について述べる．

・並列動作の変換

“；”で結合される並列動作の記述は，この動作の最初に FORK モジュールの動作が挿入され，このモジュールが並列に動作する各＜動作＞の最初の機能モジュールを連続的に起動するように変換される．また，各＜動作＞の最後では，並列動作の同期をとるために SYNC モジュールの動作が挿入される．この SYNC モジュールの動作パラメータとして，並列に実行している＜動作＞の数を与えるため，すべての＜動作＞から SYNC モジュールが起動されるまで，これに引き続く動作の実行が待ち合わされる．この変換処理のようすを**図Ⅲ－１１（ａ）**に示す．

・動作要素の定義と起動の変換

ＩＳＰＳ記述における動作要素の定義に対して，本シリコン・コンパイラはその動作要素の名前や属性等を認識するとともに，動作の記述の最後に，復帰動作（手続きの場合）や終了動作（プロセスの場合）を実現する LEAVE モジュールの動作を挿入する．これに加えて，動作要素の定義において仮引数が宣言されている場合は，これに対応する記憶モジュールが生成される．また，修飾子として critical が付与された排他的動作要素の定義の場合は，この要素が動作中かどうかを示すフラグを格納するための１ビットの記憶モジュールが生成される．

動作要素の起動の記述は，実引数の記憶モジュールの内容を仮引数の記憶モジュールへデータ転送する演算モジュールの動作と，対象とする動作要素に制御を移すための ACTIVATE モジュールの動作に変換される．但し，起動される動作要素が排他的動作要素である場合には，引数の転送の前に，対象の動作要素が動作中かどうかを問い合わせる待ち合わせ処理が付加される．動作要素の定義と起動における変換処理のようすを**図Ⅲ－１１（ｂ）**に示す．

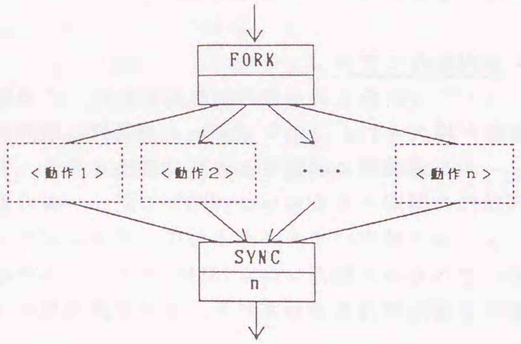
※ＩＳＰＳにおいては動作要素自身が実行結果の値を保持できる“関数”を定義できるが，演算式の中で関数の起動と実行結果の参照とが混在してコンパイル処理が複雑となるため，本シリコン・コンパイラでは関数の利用を禁止し，引数として演算結果を引き渡す方式だけに制限している．また，ＩＳＰＳにおいては leave，terminate，restart，resume という４種類の終了構文が用意されているが，本シリコン・コンパイラでは，動作要素の通常の終了を意味する leave と再スタートを指示する restart だけに限定し，他のプロセスを強制終了するための terminate やコルーチンのための resume の使用を禁止している．

・排他的動作要素の起動の変換

排他的動作要素を起動する場合は，全ての動作要素を管理する PRO-CONT モジュールに対象となる動作要素の動作状況を問い合わせ，これが動作中でない場合

（ａ）並列動作の記述

<動作１>；<動作２>；
．．．；<動作ｎ>



（ｂ）動作要素の定義と起動

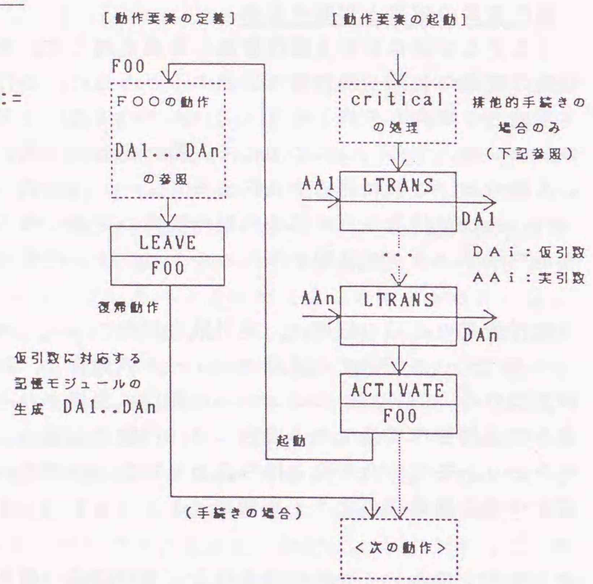
［動作要素の定義］

```
FOO (DA1<bs>...DAn<bs>):=
   begin
      <FOOの動作>
   end
```

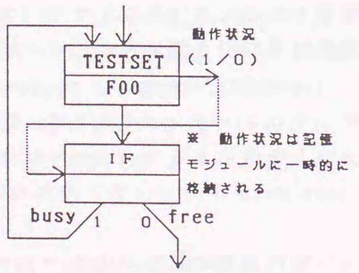
（ｂｓ：ビット構造）

［動作要素の起動］

FOO (AA1...AAn)



（ｃ）ｃｒｉｔｉｃａｌの処理



図Ⅲ－１１　ＩＳＰＳにおける制御動作の変換処理

に限って実行を開始できるが，動作中の場合には終了を待ち合わせる必要がある．従って，排他的動作要素の起動に先立って，この要素の動作状況を問い合わせるTESTSET モジュールの動作と，この動作要素の動作状況（１ビットのデータ）を参照する IF モジュールの動作が挿入される．TESTSET モジュールの動作は，動作要素の動作状況を 0／1 で応答すると同時に，この動作状況を示すフラグを動作中にセットする Test & Set の機能をもっているため，動作要素の排他的な制御が可能となる．この処理のようすを**図Ⅲ－１１（ｃ）**示す．

（２）制御情報の抽出

上で述べた構成要素の抽出処理によって，ＩＳＰＳによって記述されたＶＬＳＩチップの動作記述は，分散制御アーキテクチャにおける一連の機能モジュールの動作に展開される．従ってこの時点で，これら機能モジュールの動作順序，動作内容は厳密に規定されているため，各機能モジュールごとにそのモジュールの動作内容に関するデータを収集することにより，演算モジュールや制御モジュールの動作パラメータ，即ち，制御情報が容易に決定される．次に，各機能モジュールの１回の動作毎に起動条件を表す数値が割り当てられるとともに，このときに用いられる制御情報も数値にデコードされる．これに加えて，演算モジュール，IF モジュール，DECODE モジュールの構造パラメータ，即ち，データ・ビット幅や入力セレクタのポート数等の情報も同様に決定される．

（３）配線情報の抽出

構成要素間の配線情報に関して，演算モジュールからの演算結果の転送と，機能モジュール間での制御情報の転送は各々１本の共通バスで行う方式を採用しているため，記憶モジュールから機能モジュールへのデータ配線だけを考慮すればよい．この配線情報も，各機能モジュール毎に入力している記憶モジュールは明らかになっているため，入力セレクタのポートの割当て結果に従って，このポートと記憶モジュールの接続関係として，全ての配線情報を容易に抽出することができる．

（４）出力形式

本シリコン・コンパイラでは，上記のアーキテクチャ合成部による変換結果をレイアウト記述言語：ＤＰＬ（Design Procedure Language）[31] における１個のモジュール・クラス（class）の記述として出力している．即ち，チップ全体を１個のモジュール・クラスとして，ＤＰＬの“defcell”関数 で宣言し，このチップの構成要素である記憶モジュールと機能モジュールを，各々のモジュール・クラスに属するインスタンス（instance）として，ＤＰＬの“partq”関数 によって記述している．

例えば，ＶＬＳＩチップの構成要素となる記憶モジュール，制御（DECODE）モジュール，演算（ADDER）モジュールは以下のように記述される．

```
(DEFCELL XXX ()

  (PARTQ A_AAB REGISTER (RN 2) (BIT 8) (CONNECT (LIST 7 0)))

  {;; IN WRG_1 }
  (DV CT_8 ((1 0 ((0 (11 1)) (1 (11 3))))
           (2 0 ((0 (11 5)) (1 (11 6))))))
  (PARTQ DECODE DECODE (MN 8) (BIT 1) (IN 1) (C_TABLE CT_8))

  {;; IN_L 0, G, WRG_11 }
  {;; IN_R A_AAB, WRG_11, WRG_10 }
  (DV CT_13 ((1 1 0 0 2 (7 1))
            (2 1 1 1 9 (12 2))
            (3 0 2 2 5 (10 3))))
  (PARTQ ADDER_8_A ADDER (MN 13)(BIT 8)(IN_L 3)(IN_R 3)(C_TABLE CT_13))

  . . .  )
```

この例において，記憶モジュール A_AAB は，レジスタ番号（RN），ビット幅（BIT），及び，データバスとの接続関係（CONNECT）をパラメータとするモジュール・クラス REGISTER のインスタンスとして記述されている．制御モジュールや演算モジュールの記述も同様であり，例えば，８ビットの ADDER モジュール ADDER_8_A は，モジュール番号（MN），ビット幅（BIT），左入力セレクタのポート数（IN_L），右入力セレクタのポート数（IN_R），及び，動作パラメータを集めた制御テーブル（C_TABLE）をパラメータとするモジュール・クラス ADDER のインスタンスとして記述されている．制御テーブルは，起動条件毎の動作パラメータを集めたリストの形式（“(DV CT_8 ...)”等の記述）で与えられており，このリストの各エントリの数値は，起動条件，演算コード，左入力セレクタのポート指定，右入力セレクタのポート指定，結果を格納する記憶モジュールの番号，及び，次に起動すべき機能モジュールのモジュール番号と起動条件を表している．また，各入力セレクタの各ポートに接続される記憶モジュールの名前がコメント（“{;; ... }”の記述）として，“partq”の記述の上に示されている．

（※ Lisp システムの環境に実現されているＤＰＬシステムでは，変数名の中に“.”を含むことが許されないため，ＩＳＰＳの変数名に用いられる“.”は“_”に変換されている．）

一方，全ての接続情報は W-LIST と呼ぶリストとして記述される．このリストの要素は各々１組の配線を表し，通常，記憶モジュールの特定のビットと，演算モジュールや制御モジュールの入力セレクタの特定のポートの特定のビットが接続されることを表している．但し，定数のように常に１／０の値が決まっている場合には，１の場合は電源（VDD），０の場合は接地電位（GND）との接続が指示される．以下に接続情報の記述例を示す．

```
(DV W-LIST (((WRG_1 (0 0)) (IF_A S_0 (0 0)))
            ((B_BAB (7 0)) (COMP_8_B L_0 (7 0)))
            (GND (COMP_8_B R_0 (7 0)))
            (VDD (SHIFT_8 R_0 (0 0)))  . . .  ))
```

この例の最初の行は記憶モジュール WRG_1 の第０ビットと，IF_A モジュールの入力セレクタのポート０の第０ビットが接続されることを，２番目の行は，記憶モジュール B_BAB の第７ビットから０ビットまでと，COMP_8_B モジュールの左入力セレクタのポート０の第７ビットから第０ビットまでが接続されることを示している．また，３番目の行は，COMP_8_B モジュールの右入力セレクタのポート０の第７ビットから第０ビットまでは，すべて接地電位（論理値：０）に接続されることを表し，４番目の行は，SHIFT_8 モジュールの右入力セレクタのポート０の第０ビットは電源（論理値：１）と接続されることを表している．

３．４　レイアウト・パターン生成部

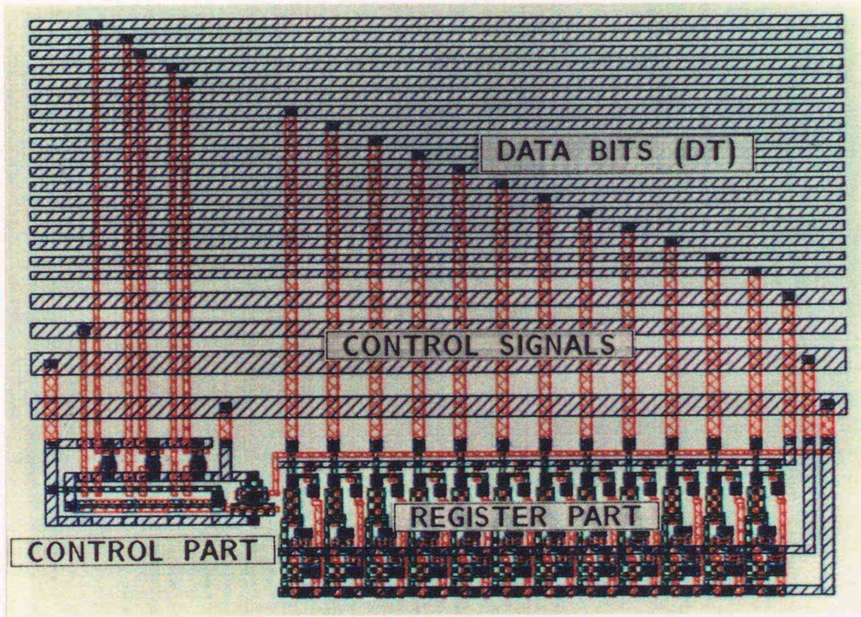
（１）モジュールの生成

本シリコン・コンパイラでは，幾つかのパラメータを含んで汎用的に定義された機能モジュールと記憶モジュールのモジュール・コンパイラが“モジュール・ライブラリ”として提供されている．従って，レイアウト記述言語：ＤＰＬの partq 関数による各モジュールの記述は，これらのモジュール・コンパイラによって，特定の構造をもつ構成要素として実体化される．

図Ⅲ－１２は，12 ビット幅の記憶モジュールのレイアウト・パターンであり，データを記憶するメモリ回路に加えて，データバスからのデータの取り込みを制御する書込み制御回路が付加されている．

また，**図Ⅲ－１３**は典型的な演算モジュールとしてのＡＬＵモジュールのレイアウト・パターンである．中央を横断するデータバスと制御バスを挟んで，この上側には入力データを選択するための２個の４対１の入力セレクタとＡＬＵモジュール本来の演算回路が配置され，共通バスの下側にはこの演算モジュール固有の制御部が配置されている．ここでは１個のＰＬＡによって全ての制御回路を実現する方式のＡＬＵモジュールが示されている．従って，各モジュールの動作パ

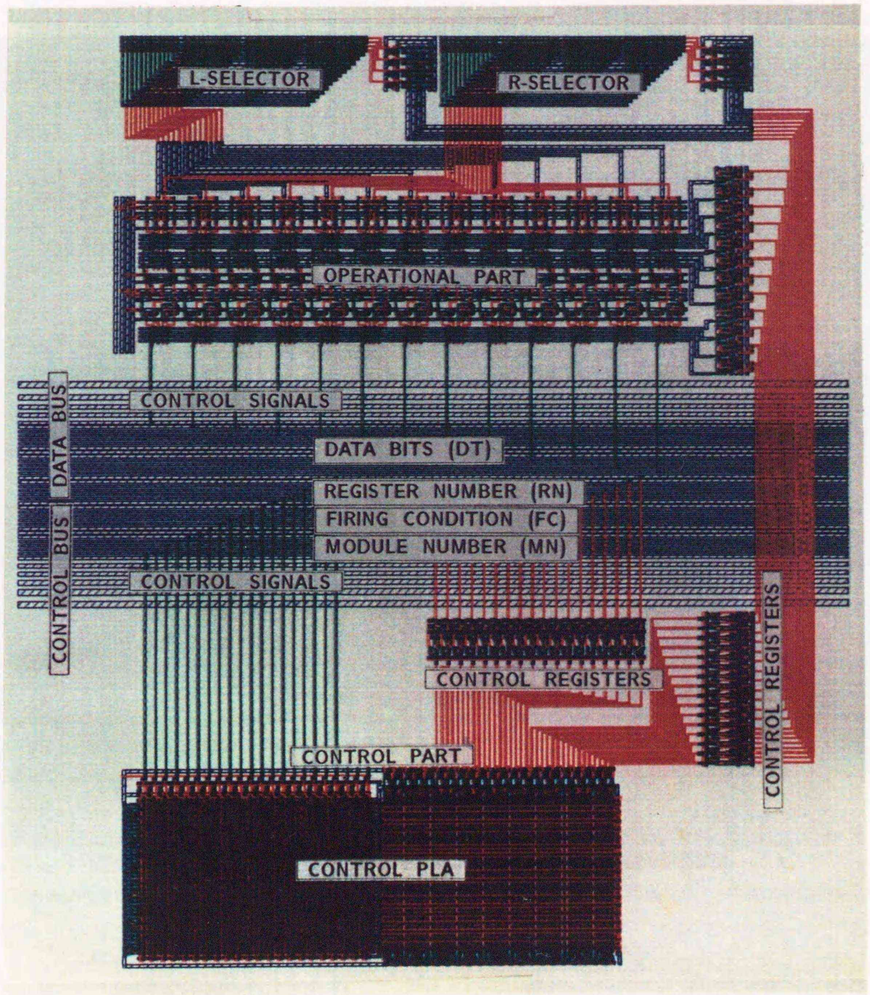
ラメータはこのＰＬＡの書込みパターンとして保持されている．ＰＬＡによって制御回路を実現する方式の他に，前章で示したような動作パラメータをＲＯＭによる制御テーブルに格納し，簡単なランダム回路によって実現する方式も考えられる．



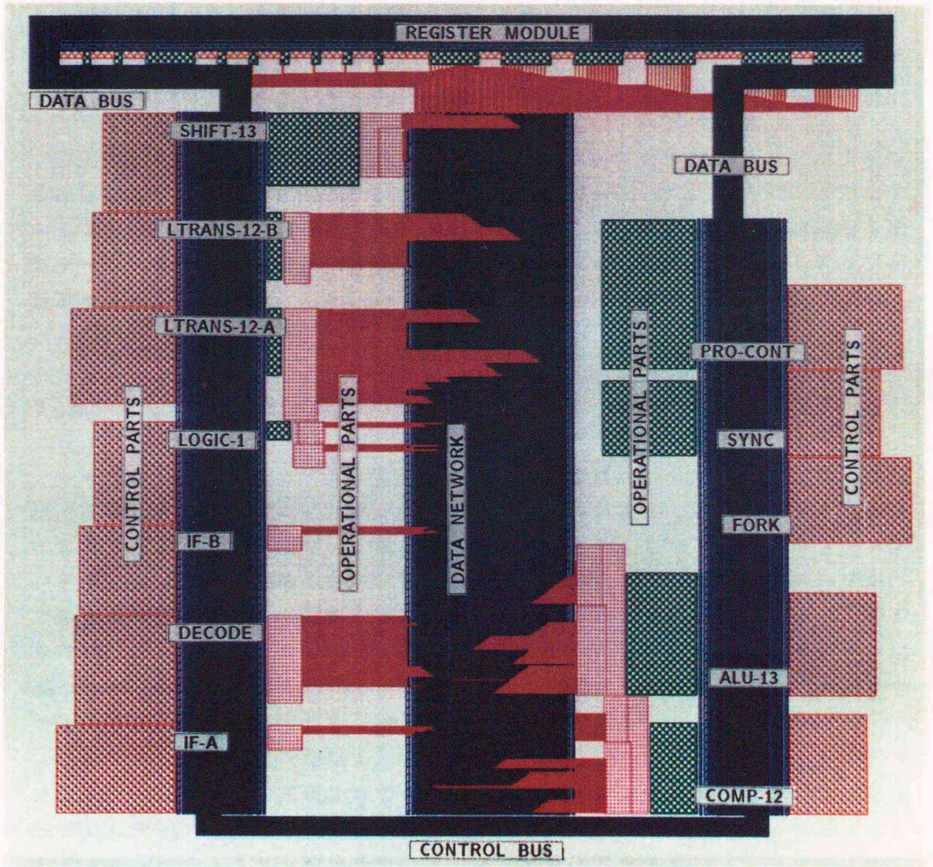
図Ⅲ－１２　記憶モジュールのレイアウト図

（２）配置・配線

各モジュールの実際の形状が決まった後，最終的なＶＬＳＩチップを生成するためにモジュールの配置とモジュール間の配線処理が行われる．今回試作したシリコン・コンパイラでは，**図Ⅲ－１４**にシリコン・コンパイラによって生成された小型計算機：ＰＤＰ－８のフロアプランを示すように，固定的なフロアプランを用いている．即ち，チップの上部には記憶モジュールを横一列に並べ，これらの記憶モジュールに記憶されているデータを取り出すための配線をチップの中央部にデータ・ネットワークとして集中化している．全ての機能モジュール（演算モジュールと制御モジュール）を２組に分け，これらを各機能モジュールの中央部を貫いている共通バス部で接合しながら，データ・ネットワークの両側に向かい合って配置している．従って，各機能モジュールの入力セレクタはデータ・ネットワークの近くに置かれているため，これから記憶モジュールのデータを水平方向の配線で簡単に取り込むことができる．また，左右に分かれた制御バスはチップの下部で接続され，機能モジュールを貫いているデータバスもチップの最上部で記憶モジュールを通るデータバスの両端と接続されている．



図Ⅲ－１３　ＡＬＵモジュールのレイアウト図



図Ⅲ－１４　ＰＤＰ－８のフロアプラン

記憶モジュールの並べ方，及び，機能モジュールの分割とその並べ方は，レイアウト・パターン生成処理に先立って実行される“最適配置計算プログラム”によって，これらの間の結合度を計算して求められる．即ち，このプログラムでは，記憶モジュールと機能モジュールは上下に各々一列に並ぶものとして，これらを結ぶ配線が交差する数をできるだけ少なくするような最適配置（並び方）を決定する．この後，機能モジュールの並びの長さがほぼ等しくなるように２つに分割し，これらをデータ・チャネルの左右に配置することにより，ＶＬＳＩチップ上での機能モジュールの最終的な配置が決定される．

（３）マスク・パターン情報の生成

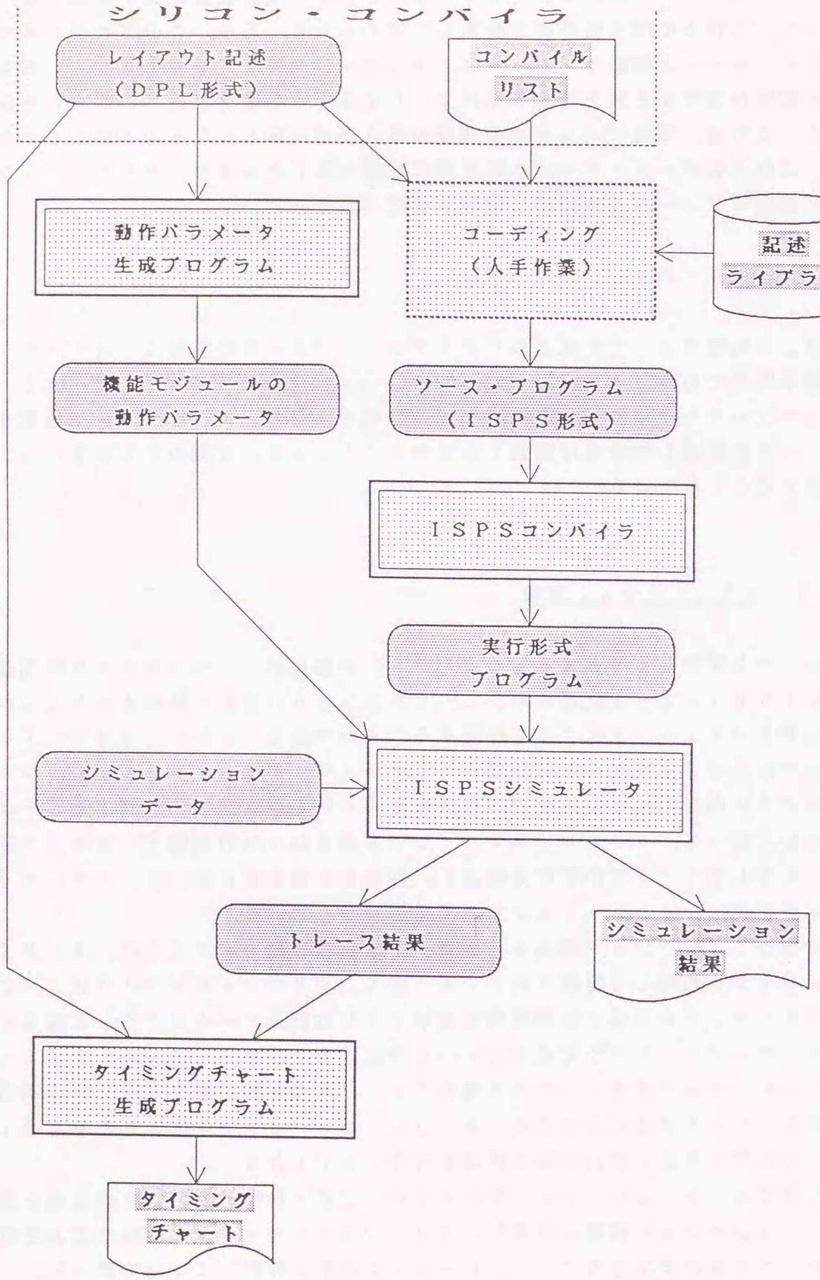
以上の処理によって生成されたレイアウト・パターンの情報はマスクパターンの標準形式であるＣＩＦ（Caltech Intermediate Form）形式[32]のデータとして，シリコン・コンパイラから最終的に出力される．このマスク・パターン情報を磁気テープに格納して半導体製造工場に渡すことにより，実際のＶＬＳＩチップを製造することができる．

３．５　シミュレーション環境

ユーザが開発しようとするＶＬＳＩチップの動作は，ハードウェア仕様記述言語：ＩＳＰＳによって記述され，このアルゴリズムが正しく動作するかどうかはＩＳＰＳシミュレータによって検証することができる．しかし，シリコン・コンパイラによって生成される分散制御アーキテクチャに基づくＶＬＳＩチップが，どのように動作し，どの程度の実行性能が得られるかは簡単に予測することができない．従って，シリコン・コンパイラで変換された分散制御アーキテクチャに基づくＶＬＳＩチップの動作を検証し，性能を評価するために，このアーキテクチャを忠実にシミュレーションする環境を構築している．

このシミュレーション環境も，**図Ⅲ－１５**に構成図を示すように，ＩＳＰＳシミュレータを利用して実現されている．即ち，シリコン・コンパイラによって生成されたチップの構成や制御情報を反映する分散制御アーキテクチャに基づくＶＬＳＩチップのＩＳＰＳ記述を行い，この記述をＩＳＰＳシミュレータによってシミュレーションする．この時の機能モジュールの起動状況や共通バスの利用状況等をトレースすることにより，シリコン・コンパイラで生成されたＶＬＳＩチップの動作の検証と実行性能の評価を行うことができる．

以下では，シミュレーション用のＩＳＰＳ記述と動作パラメータの生成を含む“シミュレーション環境の生成”，及び，シミュレーション実行時に出力されるトレース結果の表示等の“シミュレーション結果の解析”について述べる．



図Ⅲ－１５　シミュレーション環境

（１）シミュレーション環境の生成

レイアウト記述言語：ＤＰＬによるシリコン・コンパイラの出力には，構成要素，制御情報，接続情報等のＶＬＳＩチップを実現するのに必要な全ての情報が含まれている．従って，これを忠実に解釈・実行することにより分散制御アーキテクチャに基づくＶＬＳＩチップの完全な動作シミュレーションが原理的に可能である．しかし，この記述を入力してシミュレーションを行うためには，複雑なインタプリタの開発が必要であり，ここでは，ユーザがＤＰＬによるレイアウト記述や，コンパイル時に出力されるコンパイル・リストを参照して，実際のチップと同様の構成要素からなるＩＳＰＳ記述をコーディングし，この記述を対象として動作シミュレーションを行う方式を採用している．即ち，このＩＳＰＳ記述では，各機能モジュールが内蔵する制御テーブルに格納された動作パラメータに従って，１本の制御バスを経由して制御情報を転送する分散制御アーキテクチャの制御メカニズムが忠実に記述されている．

記憶モジュールに関連するデータ転送も，前章で示したのと全く同じ通信方式を忠実に記述しており，演算モジュールへのデータ転送はこれの前段に付加された入力セレクタを経由して行われ，演算モジュールからの演算結果はデータバスを経由して記憶モジュールに格納される．制御バスとデータバスには実際のチップと同様の調停機構が実現されているため，バス要求の競合や利用待ち合わせ等も忠実にシミュレートされる．従って，これらのバスの利用状況をトレースすることにより，ＶＬＳＩチップ内部での機能モジュールの動作状況やデータ転送の様子を正確に把握することができる．

このような分散制御アーキテクチャに基づくＶＬＳＩチップのシミュレータを記述するために，バスの調停機構や性能評価のための計時機構等に加えて，各機能モジュールの基本的なＩＳＰＳ記述が“**記述ライブラリ**”として用意されている．従って，ユーザはシリコン・コンパイラの出力であるレイアウト記述やコンパイル・リストを参照しながら，記述ライブラリに格納されている機能モジュールの記述を部分的に変更することにより，動作シミュレーション用のＩＳＰＳ記述を簡単にコーディングすることができる．

また，各機能モジュールの動作パラメータの設定に関しては，シリコン・コンパイラの出力であるレイアウト記述を入力し，これから制御テーブルの内容を抽出するとともに，この内容を制御テーブルに設定するためのシミュレーション・コマンド群を自動生成する“**動作パラメータ生成プログラム**”が用意されている．このコマンド群がシミュレーションの実行に先立って読み込まれることにより，実際のチップと同一の動作環境が設定される．

（２）シミュレーション結果の解析

分散制御アーキテクチャのシミュレータでは，制御バスとデータバスの利用状

況を逐一出力するようになっている．即ち，ある時刻に制御バスを用いて転送されたモジュール番号と起動条件，及び，ある時刻にデータバスを用いて転送されたレジスタ番号とデータビットが，これらの時刻とともに外部に出力される．従って，このトレース結果を解析することにより，機能モジュールの動作状況を正確に把握できるとともに，一連の動作を行うのに要した時間（クロック数）を正確に算出することができる．また，ＶＬＳＩチップ内部での機能モジュールの動作の並列度や，各共通バスの利用頻度等もこのトレース結果から簡単に算出することができる．特に，機能モジュールの動作状況に関しては，制御バスのトレース結果をタイミング・チャートに変換して表示する**“タイミング・チャート生成プログラム”**が用意されているため，機能モジュールの動作状況を視覚的に理解することができる．

図Ⅲ－１６はシリコン・コンパイラによって生成された小型計算機：ＰＤＰ－８の論理積命令を実行したときのタイミング・チャートである．図において，左端の数字がクロックの進行を表し，その右の“Ｃ”と“Ｄ”の文字の下にあるアスタリスクは，各々制御バスとデータバスの利用状況を示している．これらの右にある“Ｂ－－－Ｅ”で示される部分が機能モジュールの動作を表している．従って，これらの記号の重なり状況によって，複数モジュールによる並行動作の状況が容易に判断できる．


```
TIME C D 0-------------------1-------------------2-------------------3
0    *   B  ACTIVATE(INTERPRET)
1        |
2    *   E B  FORK
3          |                        [ＰＤＰ８の論理積命令]
4    *     | B  LTRANS
5          | |
6    * *   E | B  LTRANS
7            | |
8    * *     E | B  SYNC
9              | E
     C D 0-------------------1-------------------2-------------------3
10             |
11   *         E   B  SYNC
12                 |
13                 |
14   *             E B  FORK
15                   |
16   *               | B  ACTIVATE(M.READ)
17                   | |
18   *               E | B  ALU
19   *                 E | B  M.READ
     C D 0-------------------1-------------------2-------------------3
20     *                 | |
21                       | |
22   *                   E | B  SYNC
23                         | |
24     *                   | E
25                         |
26   *                     E   B  LEAVE(M.READ)
27                             |
28   *                         E B  SYNC
29                               |
     C D 0-------------------1-------------------2-------------------3
30                             |
31   *                         E B  LTRANS
32                               |
33     *                         |
34                               |
35   *                           E B  ACTIVATE(EXECUTE)
36                                 |
37   *                             E B  DECODE
38                                   |
39   *                               E B  ACTIVATE(EADD.CAL)
     C D 0-------------------1-------------------2-------------------3
```

図Ⅲ－１６　タイミング・チャートの表示例

4．コンパイル実験

本章では，第３章で述べたシリコン・コンパイラに対して，以下に示す例題チップの動作記述を入力し，チップの自動合成実験を行った結果について述べる．

・マンチェスタ大学で開発された初期の電子計算機：ＭＡＲＫ１（MARK1）
・$\sqrt{a^2+b^2}$ の近似値を計算するチップ（GENMAG）
・ＤＥＣ社のミニ・コンピュータ：ＰＤＰ－８（PDP8）
・アレイ・プロセッサの制御回路（CAP1，CAP2，CAP3）
・クイック・ソートを実行するチップ（QSORT1，QSORT2，QSORT3，QSORT4）

これらの例題チップのソース・プログラムの行数，記憶モジュールと機能モジュールの構成，生成されたチップの形状等を**表Ⅲ－５**に示す．また，これらのチップ（MARK1，PDP8，CAP3，QSORT4）のソース・プログラムを**付録Ⅲ－１**に示す．（GENMAG のソース・プログラムは**付録Ⅲ－２**を参照されたい．）

今回のコンパイル実験では，現在，利用できるＶＬＳＩデバイス技術として，２μｍＣＭＯＳ技術（レイアウトの基本単位：λ＝１μｍ）を前提とし，生成されたチップは 20 ＭＨｚの２相クロック（クロック幅： 50 ｎ秒）に同期して動作できることを仮定している．

（１）近似計算チップ：ＧＥＮＭＡＧ

マサチューセッツ工科大学で開発されたシリコン・コンパイラ：MacPitts[17] の評価に用いられた $\sqrt{a^2+b^2}$ の近似値計算チップ（GENMAG）を我々のシリコン・コンパイラに適用してみた．ここで用いられる近似計算アルゴリズムを以下に示す．今回のコンパイル実験では，データのビット幅が８ビットの GENMAG チップを生成している．

$$\sqrt{a^2+b^2} \approx max\left(g, \frac{7}{8}g+\frac{1}{2}l\right)$$

$$g = max\,(|a|,|b|),\ l = min\,(|a|,|b|)$$

シリコン・コンパイラに入力される最初のＩＳＰＳ記述を**付録Ⅲ－２（ａ）**，ソース・プログラム変換部によって変換されたモジュール動作の記述を**付録Ⅲ－２（ｂ）**，アーキテクチャ合成部によって出力されるコンパイル・リストを**付録Ⅲ－２（ｃ）**，及び，この時に出力されるＤＰＬによるレイアウト記述を**付録Ⅲ－２（ｄ）**に示す．また，上記のモジュール動作の様子を**付録Ⅲ－２（ｅ）**，実際のチップのフロアプランを**付録Ⅲ－２（ｆ）**に示す．さらに，シリコン・コンパイラによって生成されたチップの動作シミュレーションを行うためのＩＳＰＳ記述を**付録Ⅲ－２（ｇ）**，このシミュレーションを実行したときの機能モジュールの動作状況と各共通バスの利用状況を**付録Ⅲ－２（ｈ）**に示す．

表Ⅲ－５　シリコン・コンパイラで生成されたチップの仕様

チップ名	ISPS 行数	記憶 要素数	動作 要素数	ﾓｼﾞｭｰﾙ 動作数	記憶ﾓｼﾞｭｰﾙ 個数	記憶ﾓｼﾞｭｰﾙ ﾋﾞｯﾄ数	制御モジュール 個数	制御モジュール 内訳	演算モジュール 個数	演算モジュール 内訳	チップの形状 mm × mm
MARK1	38	4	4	30	5	94	3	IF, DECODE, PRO-CONT	3	LTRANS, COMP, ADDER	3.9 × 4.7
GENMAG	26	4	2	25	8	50	6	IF*2, DECODE, FORK SYNC, PRO-CONT	6	LTRANS, COMP*2 ADDER*2, SHIFT	2.5 × 3.2
PDP8	214	11	9	162	15	92	6	IF*2, DECODE, FORK SYNC, PRO-CONT	6	LTRANS*2, LOGIC COMP SHIFT, ALU	5.1 × 5.1
CAP1	151	13	8	87	13	221	4	IF, FORK, SYNC PRO-CONT	5	LTRANS*2, COMP ADDER*2	8.0 × 5.4
CAP2	168	13	10	88	13	221	4	（同上）	7	＋ LTRANS, ADDER	8.0 × 6.1
CAP3	183	14	12	90	14	253	4	（同上）	9	＋ LTRANS, ADDER	8.9 × 7.2
QSORT1	134	17	13	131	27	144	6	IF*3, DECODE*2 PRO-CONT	8	LTRANS*3, LOGIC COMP*3, ADDER	9.1 × 6.8
QSORT2	181	24	17	234	35	233	7	＋ IF	11	＋ LTRANS, COMP ADDER	13.3 × 8.2
QSORT3	228	31	21	337	43	322	8	＋ IF	14	＋ LTRANS, COMP ADDER	16.8 × 10.2
QSORT4	275	38	25	440	51	411	9	＋ IF	17	＋ LTRANS, COMP ADDER	20.3 × 13.4

シリコン・コンパイラによる合成実験の結果，GENMAG チップは，総計 50 ビットの記憶モジュールと，6個の制御モジュール，6個の演算モジュールで構成され，2.5 mm × 3.2 mm（8.0 mm²）の大きさである．また，シミュレーション実験の結果，このチップでは1回の近似計算に 60 クロック，即ち，約3μ秒を要することがわかる．

（2）小型計算機：ＰＤＰ８

μプロセッサを自動合成する例として，ＤＥＣ（Digital Equipments Corp.）社の小型計算機：ＰＤＰ－８を取り上げる[33]．ＰＤＰ－８は1語 12 ビットの構成で，35 個の機械語命令をもつミニ・コンピュータの先駆けとなった計算機である．このミニ・コンピュータを1チップ化したものが Intersil Corp. と Harris Corp. から市販されているため，シリコン・コンパイラによって生成されたチップと，これらの市販品をチップ面積や動作速度の観点から比較検討することができる．以下では，Harris 社から 1977 年に発表され，ＤＥＣ社のＰＤＰ－８と同じ機械語命令をもつμプロセッサ：HM-6100[34] との比較を試みる．

チップの形状を比較してみると，本シリコン・コンパイラで生成されたチップは 5.1 mm × 5.1 mm（面積：26.0 mm²），HM-6100 は 5.1 mm × 5.7 mm（面積：29.1 mm²）の大きさであり，両チップともほぼ同じ大きさをしていることがわかる．但し，HM-6100 は6μmのｐＭＯＳ技術で実現されているため，長さで3倍，面積では9倍の比率で単純に換算してみると，我々のシリコン・コンパイラによって生成されたチップは，人手によって設計されたチップに比べて，約8倍程大きな面積を必要としていることになる．

しかし，我々のシリコン・コンパイラで生成されたチップが，トランジスタ構造の複雑なＣＭＯＳ技術を用いていることや，初めから豊富なトランジスタが利用できることを前提として分散制御アーキテクチャを適用していることなどから，チップ面積の増加は，数字に現れている程，深刻な問題ではないと考えられる．むしろ，これぐらいラフな設計をしたとしても，現在の技術レベルでは十分に1チップ化できるという意味の方が重要であると考えられる．もちろん，本分散制御アーキテクチャに基づくチップの方が一般の集中制御型のチップに比べて大きくなる傾向は認められる．この原因のとしては，分散制御アーキテクチャにおいて各構成要素に分散された制御回路と多数のデータ・セレクタを導入していること，及び，簡単な配置・配線アルゴリズムによって，チップ面積の最適化が十分に行われていないこと等が考えられる．

チップの動作速度に関しては，各命令の実行時間やアドレス・モードの処理時間等を**表Ⅲ－６**に示すように，HM-6100 に比べて，シリコン・コンパイラで生成されたチップの方が約 1.6 倍程 高速であることがわかる．HM-6100 の場合は，3.33 ＭＨｚの基本クロックで動作しているが，我々のシリコン・コンパイラで

表Ⅲ－６　生成されたＰＤＰ－８と HM-6100 の動作時間の比較

	コンパイラで生成されたＰＤＰ－８				HＭ－６１００	
	動作モジュール数	データバス利用数	動作クロック数	実行時間（μ秒）	実行状態数	実行時間（μ秒）
基本命令						
論理積	２８	１０	７６	３.８０	１０	６.０
２進加算	２８	１０	７６	３.８０	１０	６.０
条件分岐（分岐）	３８	１３	９７	４.８５	１６	９.６
（分岐なし）	３７	１２	９３	４.８５		
格納とクリア	３５	１０	８３	４.６５	１１	６.６
サブルーチン呼出	３５	１０	８３	４.１５	１１	６.６
分岐	２４	８	６１	３.０５	１０	６.０
割込み許可	１９	６	４７	２.３５	１７	１０.２
アドレスモード						
直接	２８	１０	７６	３.８０	１０	６.０
間接	３５	１４	１０１	５.０５	１５	９.０
自動インデックス	４３	１６	１１９	５.９５	１６	９.６

※ アドレスモードの比較は論理積命令の動作時間で示している．

表Ⅲ－７　生成されたＰＤＰ－８の構成要素の面積

	面　積　（Kλ²）	比　率　（%）
記憶モジュール	９２６	５.７
・レジスタ部分	２９０	１.８
・データバス部分	６３６	３.９
機能モジュール	９,００４	５５.７
・演算部分（含：セレクタ）	２,１０５	１３.０
・制御部分	２,９２３	１８.１
・共通バス部分	２,６８０	１６.６
・配線部分	１,２９６	８.０
配線領域	３,５９８	２２.３
未使用領域	２,６４０	１６.３
合計	１６,１６８	１００.０

生成されるチップはこれよりも６倍高速の基本クロックを仮定している．従って，分散制御アーキテクチャにおける非同期的な制御情報の転送のような動作速度に関して効率の良くない制御方式でも，現在のＶＬＳＩデバイス技術をもちいれば，この不利を解消して余りある性能が得られることが示された．

一方，シリコン・コンパイラで生成されたチップにおいて，内部の構成要素が占める面積を計算してみたところ，**表Ⅲ－７**に示す結果が得られた．（このチップのフロアプランは**図Ⅲ－１４**を参照されたい．）この表から，機能モジュールの制御部分（チップ全体の 18 %）は演算部分（同 13 %）よりも大きな面積を占めていることが分かる．また，記憶モジュールと機能モジュールの共通バス部分とその他の配線部分を加えた全配線領域はチップ全体の 50 %以上を占めていることが分かる．今回のシリコン・コンパイラでは非常に単純な配置方式（フロアプラン）を採用したことにより，**図Ⅲ－１４**からもわかるように，**表Ⅲ－７**に示した未使用領域以外に，多くの未使用領域（空白部分）がチップ内に顕在している．

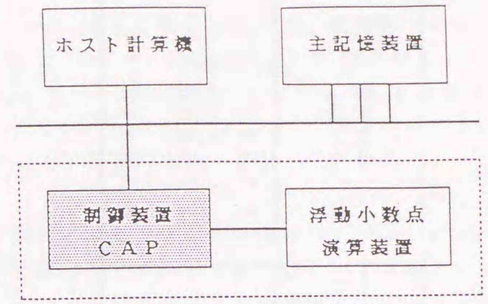
（３）アレイ・プロセッサの制御回路：ＣＡＰ

ＣＡＰ（Controller for Array Processor）は，下記に示すようなベクトル・データに対する要素毎の四則演算を計算するアレイ・プロセッサの制御回路を実現するチップである．

$$Z_i = X_i \odot Y_i \qquad i = 1 \ldots n$$

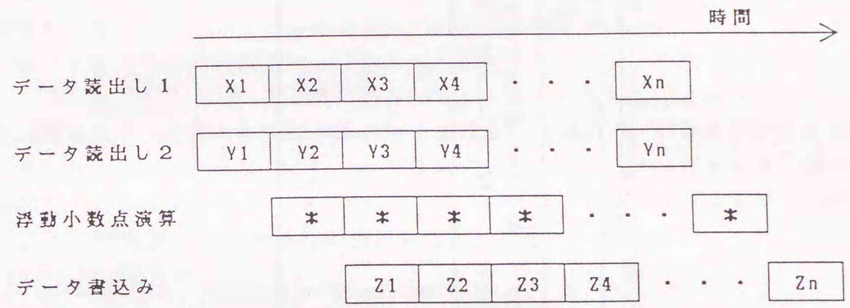
$$\odot : +, -, *, /$$

このアレイ・プロセッサは，**図Ⅲ－１７（ａ）**に示すように，ここで設計しようとしている制御回路（ＣＡＰ）と浮動小数点演算を実行する演算装置とで構成され，ホスト計算機，及び，複数の入出力ポートをもつ主記憶装置とバス結合されている．従って，アレイ・プロセッサは，ホスト計算機からの指示を受けて，主記憶に記憶されている対象データを取り込み，浮動小数点演算装置を用いて演算を行い，結果を主記憶に格納するという動作をベクトル・データの長さだけ繰り返す．この時，主記憶のポート数に応じて，ベクトル演算の並列動作が可能であり，例えば，主記憶が３個の入出力ポートをもつ場合，**図Ⅲ－１７（ｂ）**に示すように，浮動小数点演算装置による１個の演算と３個の主記憶参照（２個の読出しと１個の書込み）によるパイプライン動作によって，高速なアレイ・プロセッサが実現できる．ここでは，主記憶の入出力ポートの数に応じて，シリコン・コンパイラによってどのような制御回路（ＣＡＰ）が生成され，そのときの性能向上がどの程度のものかを評価する実験を行った．即ち，主記憶の入出力ポート数が１個の CAP1，２個の CAP2，３個の CAP3 の合成実験と動作シミュレーションを行った．ここでは浮動小数点データのビット幅を 32 ビットと仮定している．



アレィ・プロセッサ

（ａ）全体構成



（ｂ）パイプライン動作

図Ⅲ－１７　アレイ・プロセッサの全体構成とパイプライン動作

CAP1，CAP2，CAP3 は構成要素として，総計 200 ビット以上の記憶モジュールを含み，制御モジュールの数はいずれも４個であるが，主記憶装置の入出力ポートの数に応じて，32 ビット幅のデータ転送モジュール（LTRANS.32）と加算モジュール（ADDER.32）が１個ずつ増加している．CAP3 の場合，14 個（合計 253 ビット）の記憶モジュールと，32 ビットの加算器を４個，比較器を１個，データ転送器を４個含んでいるが，このチップでも１ｃｍ^2 以下であり，十分に１チップ化できる大きさである．

一方，動作速度に関しては，**表Ⅲ－８**に示すように，入出力ポート数の増加にともなう並列動作の効果から，顕著な高速化が示されている．（ここでは浮動小数点演算装置における演算，主記憶からデータ読出し，データ書込みの動作速度

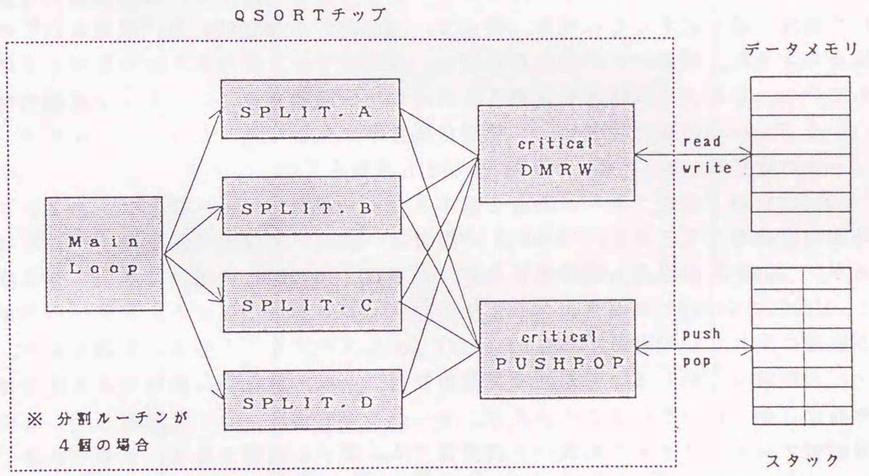
表Ⅲ－８　ＣＡＰｉの動作シミュレーション結果

チップ名	開始処理			メインループ			終了処理			実行時間（μ秒）
	動作モジュール数	データパス利用数	動作クロック数	動作モジュール数	データパス利用数	動作クロック数	動作モジュール数	データパス利用数	動作クロック数	
ＣＡＰ１	７４	４３	２４０	２９	１５	７４	２３	８	５３	３．７０ｎ＋７．２５
ＣＡＰ２	７７	４１	１４７	２９	１４	５５	２３	８	５３	２．７５ｎ＋４．６０
ＣＡＰ３	７８	４１	１３３	３０	１４	４４	２３	８	５３	２．２０ｎ＋４．９０

を各々 600 ｎ秒，400 ｎ秒，200 ｎ秒と仮定している．） ３ポートの主記憶を用いると，１ポートの主記憶の場合に比べて，並列動作のために４個の演算モジュールが余分に必要となり，チップ面積も 1.4 倍になるが， この時，動作速度は約 1.7 倍の高速化を図れることがシミュレーション実験から示された．

（４）ソータ・チップ：ＱＳＯＲＴ

並列プロセスで動作するもうひとつの例として，Quick Sort[35] を実行するソータチップの設計を試みた．この例では，**図Ⅲ－１８**に示すように，このチップの外部に，ソーティングの対象データを格納する“データメモリ”と，ソーティングの対象となるデータ領域の指標対を格納する“スタック”があることを想定している．このソータチップの動作記述では，ソーティングの対象領域を２個の独立な領域に分割する処理を行う複数の“分割ルーチン（SPLIT）” を導入している．従って，SPLIT の数に応じて，並列度の向上によるソーティング動作の高速化が期待できる．今回の実験では上記の分割ルーチンの数を１，２，３，４の４個の場合における合成実験と動作のシミュレーション実験を行った．（各分割ルーチンがプロセスとして並列に動作するが，これに加えて，これらのプロセスを起動したり，全体の動作終了を検知する“Main Loop”も プロセスとして並列に動作している．）また，このチップの動作記述の中に，データメモリに対する読出し／書込み動作を行う“DMRW”と，スタックに対するプッシュ／ポップ動作を行う“PUSHPOP”が，排他的動作要素として記述されている．



図Ⅲ－１８　ソータチップの構成

表Ⅲ－５に示すように，１個の分割ルーチンをもつ QSORT1 の場合， 27 個の記憶モジュール（合計：144 ビット），６個の制御モジュール，及び，８個の演

算モジュールから構成され，１ｃｍ^2 以下の形状をしている．しかし，分割ルーチンの数が１個増えると，８個の記憶モジュール（合計： 89 ビット），１個の制御モジュール（IF），及び，３個の演算モジュール（LTRANS, COMP, ADDER）が 追加され，４個の分割ルーチンをもつ QSORT4 の場合では， チップの形状が 20 ｍｍ × 13 ｍｍと非常に大きなチップが生成されている．ＩＳＰＳのようなハードウェア仕様記述言語では複数のプロセスからなる動作でも簡単に記述できるが，シリコン・コンパイラはチップ製造上の物理的な制約などを考慮せずにコンパイルしてしまうため，ユーザは生成結果を見ながら，適切なハードウェア規模になるような記述の変更，分割等の最適化をおこなう必要がある．

これらのチップの動作シミュレーション結果を**表Ⅲ－９**に示す．今回のシミュレーション実験では，100 個のデータに対するソーティングを行っているが，このとき，最低でも３万個以上の機能モジュールが動作し，シミュレーション時間も DEC System 2060 のＣＰＵ時間で２時間以上と非常に大規模なシミュレーションになっている．データメモリの読出し／書込み動作とスタックのプッシュ動作の回数は分割ルーチンの数によらず一定であり，各々 1,406 回，363 回，86 回である．**表Ⅲ－９**において，“遅延なし／遅延あり”という区別は，ソートチップの Main Loop において，全分割ルーチンが動作を終了したかどうかを判定する IF モジュールの動作と，各分割ルーチンにおいて，スタックからソート領域の指標を取り出したときにスタックが空かどうかを判定する IF モジュールの動作とにおける“時間遅延機能”の有無を示している．時間遅延機能の無い IF モジュールの場合は，稀にしか発生しない条件の発生を待って，無意味な判定動作を頻繁に繰り返すことになる．従って， QSORT1 と QSORT2 の“遅延なし”の場合のように，制御バスの競合が頻発し，本来のソート動作に大きな影響を及ぼしている．このような状況を回避するために，今回のシミュレーション実験では，IF モジュールの動作において，特定の条件が発生しなかったときは，次のモジュールの起動を 100 クロックだけ遅らせる機能を付加している．

QSORT1 の“遅延つき”の場合を基本として，並列プロセス数とソーティング時間の変化をみてみると， QSORT2 の場合は，約 17 %程度の性能向上がみられるが， QSORT3 の場合も同程度であり，さらに， QSORT4 の場合は約 15 %程度と QSORT2, QSORT3 の場合よりも性能が悪化している．このように並列化の効果が顕著に現れない原因の一つは， Quick Sort アルゴリズム自身の問題として，データ領域を分割していく過程で実際に並列プロセスが有効に動作できる機会はそれほど多くないということがある．もう一つの原因は，ここで想定している分散制御アーキテクチャの制御バスの実現方式に関する問題であり，１本の共通バスによる制御情報の転送方式では，各機能モジュールの所用時間との関係で４個以上の機能モジュールが並列動作しても （QSORT3 の場合に４個のプロセスが存在する），制御バスの要求競合が頻発し，結果として性能向上に反映できないということである．従って，高度の並列処理による性能向上を実現するためには，現在よりもさらに制御転送能力の高い方式を利用しなければならない．

表Ⅲ－９ ソータチップの動作シミュレーション結果

	QSORT1		QSORT2		QSORT3	QSORT4
	遅延なし	遅延つき	遅延なし	遅延つき	遅延つき	遅延つき
動作モジュール数	75,534	37,899	88,505	46,407	56,575	67,345
制御バス競合回数	2,246	247	43,223	4,913	15,324	28,210
データ書込み回数	30,440	17,770	38,249	22,096	27,182	32,563
データバス競合回数	258	24	1,005	159	401	652
動作クロック数	109,690	108,158	101,157	92,551	92,514	94,248
性能比	0.99	1.00	1.07	1.17	1.17	1.15
シミュレーション CPU時間	2h35m30s	1h58m23s	3h24m26s	2h23m18s	3h10m12s	4h8m50s

5．評価

5．1　分散制御アーキテクチャの評価

第２章で示した分散制御アーキテクチャを基本アーキテクチャとする動作記述型のシリコン・コンパイラを試作し，いくつかのＶＬＳＩチップの自動合成実験を行うとともに，これらのチップの動作性能をシミュレーション実験によって評価した．この結果，分散制御アーキテクチャに基づくＶＬＳＩチップは，従来の集中制御型のチップに比べて，以下に示すような特徴をもっていることが示された．

① 分散制御アーキテクチャは，まとまった機能を実現する構成要素が順序だって動作するため，手続き的に記述されたＶＬＳＩチップの動作記述を忠実に反映でき，設計者が直感的に理解できる自然な動作モデルである．特に，並列プロセスの制御等を含む高級プログラミング言語における制御構文も，対応する制御要素の動作として１対１に対応づけられる．

② 各構成要素の制御回路は，要素内部の制御と自分自身に関わる起動制御だけを行えばよく，簡潔なハードウェア構造で実現できる．また，要素間の通信プロトコルを満たせば，内部の実現方式や動作速度は他の構成要素に全く影響しないため，各構成要素を独立に実現できる．

③ 複数の構成要素の同時動作が可能であり，並列処理による処理速度の向上が期待できる．但し，共通バスによって起動情報の転送を行う方式では，バスの転送能力による並列動作の限界が生じる．

一方，以下に示すようなアーキテクチャ上の課題や問題点もある．

① 一般に，分散制御アーキテクチャに基づくＶＬＳＩチップは集中制御型のチップよりもチップ面積が大きくなる傾向がある．この原因として，以下に示すアーキテクチャ上の特質が考えられる．

・制御回路を各構成要素に分散したため，各々の構成要素における制御回路は簡単になるが，チップ全体としては制御回路に関わるハードウェア量を大きくしてしまう．
・ここで設定した分散制御アーキテクチャでは，機能モジュールへの入力配線を専用の個別配線としたため，このための配線領域と多数の入力セレクタを必要としている．
・この他にもモジュール・コンパイラの利用と単純なフロアプラニングよって，チップ面積を大きくしていることが考えられるが，これはコンパイル方式に関

わることであり，次の節で考察する．

② また，ここで設定した分散制御アーキテクチャでは，チップの動作速度に関しても集中制御型に比べて遅くなる傾向がある．この原因としては，以下に示す要因が考えられる．

・通常の集中制御方式では，演算に要する動作時間（クロック数）が明確に決まっているため，演算回路と制御回路が並列動作して，演算回路の動作終了と同時に次の演算回路を起動している．しかし，ここで設定した分散制御アーキテクチャの各機能モジュールでは，演算動作が完了した後で制御動作を開始しているため，動作時間が余分にかかる．
・制御情報の転送を１本の共通バスで行っているため，バスの使用権を要求したり，調停するためのステップを必要とする．
・演算結果の転送も１本の共通バスを用いているため，今回の実験では顕著に現れてはいないが，データバスの競合による動作の遅れが発生する可能性がある．

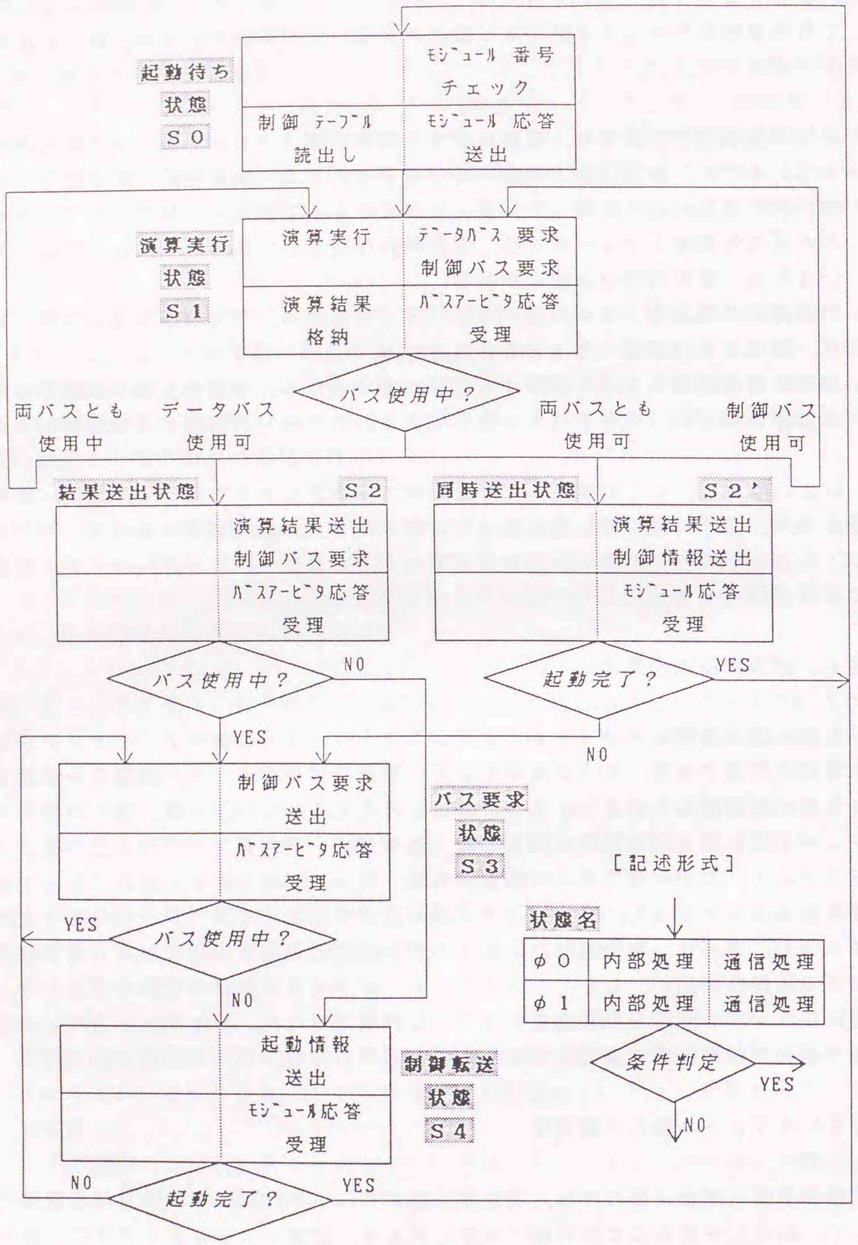
以上のように，ここで設定した分散制御アーキテクチャではチップ面積，動作速度とも，従来の集中制御型に比べて性能が低下する傾向が認められる．しかし，以下のようなアーキテクチャ上やレイアウト上の改良を行うことによって，性能の改善が期待できる．

（１）制御回路の小型化

制御回路を各機能モジュールに分散したことは，本分散制御アーキテクチャの本質的な特徴であり，チップ全体としての制御回路の大きさを，論理を共通化できる集中制御型のものより小さくできるとは考えられない．一方，個々の機能モジュールにおける制御回路に関しては，各モジュール内部での実現方式を改良することによって小型化できる可能性がある．即ち，制御回路を１個のＰＬＡで実現することもできるが，ＲＯＭとランダム回路で実現するほうが小型化できるであろうし，さらに，制御回路全体をランダム回路で実現するほうがさらに小型化できる可能性が高い．しかし，シリコン・コンパイラにおける実現方式として，複雑なランダム回路を自動合成することは容易ではなく，レイアウト方式との関連で適切な制御回路の実現方式を設定する必要がある．

（２）モジュール動作の高速化

動作速度の向上に関しては，現在の機能モジュール内部での状態遷移を最適化して，高速化を図ることが可能である．例えば，**図Ⅲ－１９**に示すように，制御バスの使用要求を前もって発行することによって，状態遷移は幾分複雑にはなるが，現在５クロックを要している演算モジュールの動作を最短３クロックに短縮



図Ⅲ－１９　高速化した演算モジュールの状態遷移

することができる．制御の起動状態（Ｓ０とＳ４）は２つのモジュールで重複しているから，上記の場合の実質的な動作クロック数は２クロックと考えられ，現在の４クロックに比べて２倍に高速化できる可能性がある．制御モジュール等ではこれほどの高速化が図れない場合もあるが，このようなモジュール内部処理の改良を施すことにより，チップ動作全体としても２倍近い高速化が期待できる．

本アーキテクチャの優れた点は，このような改良をモジュール単位で独立に行えることであり，他のモジュールの実現方式によらず，最も使用頻度の高いモジュールから順次高速化を図っていくことが可能である．但し，このような高速化は制御バスの使用頻度を高めることから，以下に述べるようなバス競合の問題が顕著になっていくと考えられ，このために機能モジュールの起動メカニズムの再検討等が必要になる．

（３）バス競合の回避

１本の共通バスを用いることによって発生する競合遅延に関しては，使用頻度の高い通信用に専用配線を設けたり，複数の共通バスを導入することなどによって，これを回避することが可能である．しかし，各チップ毎に，高速化の要求程度やバス複数化の効果は異なり，また，配線に必要なチップ面積の増加とのトレードオフの問題であるため，適切なバス構成を決定するアルゴリズムの開発は大変難しいと思われる．

５．２　シリコン・コンパイラの評価

分散制御アーキテクチャに基づく動作記述型シリコン・コンパイラの評価として，今回試作したプログラムのステップ数（コメント等を含む）を**表Ⅲ－１０**に示す．（この表におけるＬＩＳＰのステップ数は，システムに組込みの Pretty Print 関数で出力された形式を数えたものである．）これらのプログラムは何れもＤＥＣ社の中型計算機：DEC System 2060 上にインプリメントされている．

ソースプログラム変換部は 1800 行以上のステップ数を要しているが，このうちの約 500 行はＩＳＰＳ構造化エディタに関わるものであり，残りの約 1300 行にＩＳＰＳ記述に対する変換処理が記述されている．この変換処理は，単一の構文だけを対象とする局所的な変換操作であり，各変換規則毎の比較的簡単なプログラムの集まりになっている．換言すれば，本変換部には高度な最適化の処理が組み込まれておらず，記憶モジュールや演算モジュールの統合，ＡＬＵやカウンタの利用，並列動作のための演算モジュールの並置等のアーキテクチャの最適化の判断をユーザに委ねている．しかし，シリコン・コンパイラで生成されるチップの性能はこの変換部の出力によって決定してしまうため，入力されたＩＳＰＳ記述の全体的な視点に立って，最適な記憶要素や演算モジュールの構成を生成

表Ⅲ－１０　シリコン・コンパイラのステップ数

プログラム	使用言語	ステップ数（行）
シリコン・コンパイラ本体		
・ソース・プログラム変換部	ＬＩＳＰ	１,８６６
・アーキテクチャ合成部	ＬＩＳＰ	２,４０２
・最適配置計算	ＦＯＲＴＲＡＮ	７８
・レイアウト・パターン生成部	ＬＩＳＰ	４８３
※ ＤＰＬシステム	ＬＩＳＰ	２,５４５
シミュレーション環境		
・動作パラメータ生成	ＬＩＳＰ	２００
・タイミング・チャート表示	ＬＩＳＰ	２７８

できる最適化アルゴリズムをこの変換部に組み込んでいく必要がある．

アーキテクチャ合成部は約 2400 行の大きなプログラムになっているが，その処理内容は以下に示すような比較的単純な処理の連続である．

・構文解析結果（ＧＤＢ形式）の読込み
・機能モジュールによる動作への展開
・機能モジュールによる動作の順序づけ
・動作要素に関する情報の収集
・記憶モジュールの構成の決定
・機能モジュールの構成の決定
・各機能モジュールの制御情報の収集
・配線情報の収集
・コンパイル結果（ＤＰＬ記述）の生成
・コンパイル・リストの出力

レイアウト・パターン生成部における配置・配線処理も，チップのフロアプランを単純な構造に固定しているため，処理自身は非常に簡単である．しかし，各構成要素を実体化するためのモジュール・コンパイラの開発には，非常に大きな労力を必要とし，特に，パラメータの値によらずに生成されるチップの形状を整えようとすると複雑なプログラミングが要求される．現在のシリコン・コンパイラで用意されている機能モジュールでは，入力セレクタ，演算回路，制御回路，共通バス等を個々のモジュール・コンパイラで生成し，これらを特定の位置関係で一律に配置・配線する方式を採用している．従って，機能モジュールに与えられた特性値によっては，形状がいびつになったり，未使用領域が多くなってしまう可能性がある．例えば，データ転送モジュール（LTRANS）の演算部分は入力セレクタだけから成るが，データの入力数やビット幅の増加によって，非常に横幅の長い回路が生成される．この時，制御回路はそれほど変化しないため，制御回路側には大きな未使用領域が発生してしまう．

また，チップ全体のフロアプランに関しても，簡単なレイアウト方式を採用しているため，生成されるチップの面積が大きくなってしまう可能性がある．即ち，機能モジュールと記憶モジュールは，共通バスを“接合（abutment）”することによってブロック化する方式を採用しているが，接合されるモジュールの形状（特に，共通バスの上下にある部分の大きさ）が各々異なるため，接合結果のブロックには大きな未使用領域が発生してしまう．さらに，これらのブロックからチップ全体の構成を決めるときも，固定的な位置関係のフロアプランで配置・配線しているため，ここでも大きな未使用領域が発生してしまうことがある．

以上のように，今回試作した動作記述型シリコン・コンパイラは，分散制御アーキテクチャのチップ構成を容易に変換・抽出できるという特徴があるが，人手によって設計されたチップよりも大きな形状のチップが生成されてしまう傾向が認められる．しかし，以下のようなコンパイル方式の改良を行うことによって，生成されるチップの性能を改善できると考えられる．

（１）アーキテクチャの最適化

現在のシリコン・コンパイラでは，生成されるチップの小型化や高速化できる記述を提示して設計者の判断を仰いでいるが，明らかに冗長な記述などもあり，詳細なデータフロー解析を行うことにより，このような記述に対する最適化処理を自動的に行うように改良することができる．また，分散制御アーキテクチャに適した動作記述に関しての知識，ノウハウ，ヒューリスティクス等が蓄積してきた時点で，これらをシステムに組み込み，設計者が最初にＩＳＰＳ記述を行うときのコンサルティングを行うことも有効である．このようなヒューリスティクスの例として，以下に示すものがある．

“１，２個の演算しか含まない動作を並列化しても，FORK モジュールや SYNC モジュールの動作のオーバーヘッドのために高速化されないことがある”

“引数をもった動作要素は，リンケージのためだけに使われる記憶モジュールを必要とし，起動時にこれへのデータ転送を伴うことを考慮する必要がある”

（２）レイアウトの最適化

本シリコン・コンパイラでは，モジュール・コンパイラによって全ての構成要素を実体化し，これらの要素を相対的な位置関係によって配置・配線する方式を

採用している．この方式では接続するブロックの形状によって，未使用領域が発生することがある程度避けられない．しかし，アーキテクチャとしての自由度を保証しようとすると，特定のレイアウト方式だけに限定することは好ましくなく，構成回路，モジュールの枠を越えて更に大きな構成要素として，最適化を図る必要がある．例えば，モジュール・コンパイラで生成されるのは単なる論理回路として，ゲートアレイやセルベースの設計手法によって，このモジュールを高密度に実現したり，あるいは，チップ全体の論理回路を最適化して配置・配線する方法などが考えられる．

6．おわりに

本第Ⅲ部では，動作記述型のシリコン・コンパイラの実現に適したＶＬＳＩアーキテクチャとしての分散制御アーキテクチャと，このアーキテクチャに基づくＶＬＳＩチップを自動生成する動作記述型シリコン・コンパイラに関する研究について述べた．

ここで提案した分散制御アーキテクチャは，内部に制御回路を含んで独立に動作する要素で構成され，これらが局所的な通信を行いながら所定の動作を実現する．一般に，並列プロセスの実行環境下で，手続きのリンケージや排他制御等の高級言語の制御機能をハードウェア的に効率よく実現するのは容易ではないが，ここで提案した分散制御アーキテクチャを適用することにより，並列動作記述言語における各種の制御構文を制御モジュールの動作に容易に変換できることを示した．また，これらのチップを構成する演算モジュールや制御モジュールが，モジュール・コンパイラによって自動生成でき，これらの構成要素を配置・配線することによって分散制御アーキテクチャに基づくＶＬＳＩチップ全体のレイアウト情報までをソフトウェア的に変換できることを示した．

この結果，分散制御アーキテクチャが，動作記述型のシリコン・コンパイラを実現するのに適したＶＬＳＩアーキテクチャであること，及び，このアーキテクチャが将来のＶＬＳＩシステムの実現方式として適切なものであることが明らかとなった．また，プログラミング言語等による手続き的な記述を受け付ける動作記述型シリコン・コンパイラの実現可能性を明らかにすることができた．

ここで述べた分散制御アーキテクチャに基づくシリコン・コンパイラは，ＶＬＳＩチップの動作記述から物理的な構造への変換の容易性や，ＶＬＳＩチップとしての構成要素の実現性等が優れている．この結果，各応用分野のシステム設計者やプログラマ等が対象システムの動作を簡潔に記述するだけで，その機能を実現するＶＬＳＩチップを容易に実現できる．しかし，分散制御アーキテクチャに基づくチップでは，制御機能を各機能モジュールに分散したことや，比較的多数のデータ配線を必要とすることなどから，手書きで設計されたものよりチップ面積が大きくなる傾向がある．また，分散制御アーキテクチャでは，相手の機能モジュールが動作可能であることを確認する操作等の通信のオーバヘッドがあるため，集中型の制御方式に比べて動作速度が低下するという問題もある．従って，様々な記述から導かれるチップの構成要素を常にコンパクトにする物理的な実現方式や配置・配線アルゴリズム，及び，機能モジュール間の通信オーバヘッドを低減し，モジュール動作の並列性を高める等の高速化手法について，今後の改良の余地が残っている．

今回，試作したシリコン・コンパイラで自動生成されたチップは，人手で設計されたものよりも性能が低下するという問題があるものの，対象システムがこれまでソフトウェア的に実現されていたことに比べれば，動作速度や形状等の様々

な面で著しい効果が期待できる．また，ＡＳＩＣの開発者が設計支援ツールに対して要求することは，正しく動作するチップが設計労力を殆どかけずに，容易に実現するということであり，シリコン・コンパイラによるＶＬＳＩ設計手法はこの目的に最も合致したものである．

一方，シリコン・コンパイラに関わる変換過程が多岐にわたり，複雑な分だけ，今後，コンパイラの技術が大きく発展する可能性は高いと考えられる．現在，ソフトウェア技術者の殆どがアセンブラではなく，高級言語を用いているということから考えても，将来，ここで試作したような動作記述型のシリコン・コンパイラがＡＳＩＣ開発の主力ツールとして多くの設計者に利用される状況になるであろうと考えられる．

［参考文献Ⅲ］

(1) Raymond, T.C.: LSI/VLSI Design Automation, IEEE Computer, Vol.14, No.7 (1981).
(2) Shiva, Sajjan G.: Automatic Hardware Synthesis, Proc. of the IEEE, Vol.71, No.1 (1983).
(3) Johannsen, D.: Bristle Blocks: A Silicon Compiler, Proc. of the 16th Design Automation Conf. (1979).
(4) 平山正治：シリコン・コンパイラ，情報処理，Vol.25, No.10 (1984).
(5) Mostow, Jack: A Decision-Based Framework for Comparing Hardware Compilers, The Journal of Systems and Software, Vol.4 (1984).
(6) Gajski, Daniel D.: Silicon Compilation, Addison-Wesley Publishing Co., Reading (1988).
(7) 杉山尚志・今井正治：超ＬＳＩ設計シリコンコンパイレーション，サイエンスフォーラム，東京 (1988).
(8) Goldberg, Andrew V., et al.: Approaches Toward Silicon Compilation, IEEE Circuits and Devices Magazine, Vol.1, No.3 (1985).
(9) Johnson, Stephen C.: Silicon compiler lets system makers design their own VLSI chips, Electronic Design, October 4 (1984).
(10) Lee, Bebjamin, et al.: Silicon compiler teams with VLSI workstation to customize CMOS ICs, Electronic Design, November 15 (1984).
(11) Southard, Jay R.: Silicon compiler demands no hardware expertise to fashion custom chips, Electronic Design, November 15 (1984).
(12) Bergmann, Neil: A Case Study of the F.I.R.S.T. Silicon Compiler, Proc. of the 3rd Caltech Conference on VLSI (1983).
(13) De Man, H., et al.: Cathedral-II: A Silicon Compiler for Digital Signal Processing, IEEE Design & Test, Vol.3, No.6 (1986).
(14) Marwedel, Peter: The MIMOLA Design System: Tools for the Design of Digital Processors, Proc. of the 21st Design Automation Conf. (1984).
(15) Lam, Monica & Jack Mostow: A Transformational Model of VLSI Systolic Design, Proc. of the 6th International Symposium on Computer Hardware Description Languages and their Applications (1983).
(16) Park, N. & A.C. Parker: Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications, IEEE Trans. on CAD, Vol.7, No.3 (1988).
(17) Southard, Jay R.: MacPitts: An Approach to Silicon Compilation, IEEE Computer, Vol.16, No.12 (1983).
(18) Hafer, Louis J. & Alice C. Parker: Automated Synthesis of Digital Hardware, IEEE Trans. on Computers, Vol.C-31, No.2 (1982).
(19) Trickey, Howard: Flamel: A High-Level Hardware Compiler, IEEE Trans.

on CAD, Vol.CAD-6, No.2 (1987).
(20) Berstis, Viktors: The V Compiler: Automating Hardware Design, IEEE Design & Test, Vol.6, No.2 (1989).
(21) Ullman, Jeffrey D.: Computational Aspects of VLSI, Computer Science Press, Inc., Rockville (1984).（都倉信樹他訳：ＶＬＳＩ計算の諸側面，近代科学社，東京 (1990).）
(22) Kelem, Steven H.: A Method for the Automatic Translation of Algorithms from High-Level Language into Self-Timed Integrated Circuits, IEEE Circuits and Devices Magazine, Vol.1, No.2 (1985).
(23) Jhon, Chu S., et al.: Silicon Compilation Based on a Data-Flow Paradigm, IEEE Circuits and Devices Magazine, Vol.1, No.3 (1985).
(24) Meng, Teresa H.-Y., et al.: Automatic Synthesis of Asynchronous Circuits from High-Level Specifications, IEEE Trans. on CAD, Vol.8, No.11 (1989).
(25) Hirayama, Masaharu: A Silicon Compiler System Based on Asynchronous Architecture, IEEE Trans. on CAD, Vol.CAD-6, No.3 (1987).
(26) Hirayama, Masaharu: VLSI Oriented Asynchronous Architecture, Proc. of the 13th Annual International Symposium on Computer Architecture (1986).
(27) 平山正治：動作記述型シリコン・コンパイラにおける制御構造の実現方式，情報処理学会論文誌，Vol.31, No.2 (1990).
(28) Barbacci, Mario R., et al.: The ISPS Computer Description Language, Carnegii-Mellon Univ. Tech. Report No.CMU-CS-79-137 (1977).
(29) Seitz, Charles L.: System Timing, in Introduction to VLSI Systems, Carver Mead & Lynn Conway, Addison-Wesley, Reading (1980).
(30) Yakovlev, Alexandre: Designing Self-Timed Systems, VLSI Systems Design, Vol.6, No.9 (1985).
(31) Batali, John & Anne Hartheimar: The Design Procedure Language Manual, MIT A.I. Memo. No.598 (1980).
(32) Hon, Robert H. & Carlo H. Sequin: A Guide to LSI Implementation, 2nd Edition, XEROX Palo Alto Research Center, No.SSL-79-7 (1980).
(33) Bell, C. Gordon, et al.: Computer Engineering, Digital Equipment Corporation, Bedford (1979).
(34) CMOS Digital Data Book, Harris Corp. (1984).
(35) Knuth, Donald E.: The Art of Computer Programming, Vol.3, Sorting and Searching, Addison-Wesley Publishing Co., Reading (1973).

付録Ⅲ－１　シリコン・コンパイラの入力記述

（ａ）ＭＡＲＫ１

```
MARK1 := begin
    ** memory.state **
    ext.local M [0:8191] <31:0>,
    ext.register EMAR <12:0> {RN:1},
    ext.register EMDR <31:0> {RN:2},
    IMDR <31:0>,

    ext.module M.READ {MN:4; PN:1} := begin
        IMDR = M[EMAR]
    end,

    ext.module M.WRITE {MN:5; PN:2} := begin
        M[EMAR] = EMDR
    end,

    ** processor.state **
    PI¥Present.Instruction <15:0>,
        F¥Function <0:2> := PI <15:13>,
        S¥Operand <0:12> := PI <12:0>,
    CR¥Control.Register <12:0>,
    AC¥Accumulator <31:0>,

    ** Instruction.Execution ** {TC}
    main I.CYCLE := begin
        EMAR = CR next   M.READ () next   PI = IMDR next
        Decode F => begin
            0¥JMP := (EMAR = S next   M.READ () next   CR = IMDR),
            1¥JRP := (EMAR = S next   M.READ () next   CR = CR + IMDR),
            2¥LDN := (EMAR = S next   M.READ () next   AC = - IMDR),
            3¥STO := (EMAR = S next   EMDR = AC next   M.WRITE ()),
            4¥SUB := (EMAR = S next   M.READ () next   AC = AC - IMDR),
            6¥CMP := if AC lss 0 => CR = CR + 1,
            7¥STP := stop (),
        end next
        CR = CR + 1 next
        restart I.CYCLE
    end
end
```

(b) PDP8

```
PDP8 {WDT:13} := begin

** mp.state **
    ext.local M¥memory [0:4095] <0:11>,    ! memory (external)
    ext.register EMAR <0:11> {RN:1},       ! memory address (external)
    ext.register EMDR <0:11> {RN:2},       ! memory data for write (external)
    IMDR <0:11>,                           ! memory data for read (internal)

    ext.module M.READ {MN:4; PN:1} := begin
        IMDR = M[EMAR]
    end,

    ext.module M.WRITE {MN:5; PN:2} := begin
        M[EMAR] = EMDR
    end,

** pc.state **
    PC¥program.counter <0:11>,
    LAST.PC <0:11>,
    LAC <0:12>,
        field L¥link <>                 := LAC <0>,
        field AC¥accumulator <0:11>     := LAC <1:12>,
    I.STATE <>,
    I.REQUEST <>,
    SWITCHES <0:11>,

** instruction.format **
    I¥instruction <0:11>,
        OP¥operation.code <0:2>  := I <0:2>,
        IB¥indirect.bit <>       := I <3>,
        PB¥page.0.bit <>         := I <4>,
        PA¥page.address <0:6>    := I <5:11>,

        IO.SELECT <0:5>          := I <3:8>,      ! device select
        IO.CONTROL <0:2>         := I <9:11>,     ! device operation
        IO.PULSE.P1 <>           := IO.CONTROL <0>,
        IO.PULSE.P2 <>           := IO.CONTROL <1>,
        IO.PULSE.P4 <>           := IO.CONTROL <2>,

        SMA <>   := I <5>,               ! skip on minus AC
        SPA <>   := I <5>,               ! skip on positive AC
        SZA <>   := I <6>,               ! skip on zero AC
        SNA <>   := I <6>,               ! skip on AC not zero
        SNL <>   := I <7>,               ! skip on L not zero
        SZL <>   := I <7>,               ! skip on L zero
        IS <>    := I <8>,               ! invert skip sense
        GROUP <> := I <3>,               ! microinstruction group
        CLA <>   := I <4>,               ! clear AC
```



```
        CLL <>   := I <5>,               ! clear L
        CMA <>   := I <6>,               ! complement AC
        CML <>   := I <7>,               ! complement L
        RAR <>   := I <8>,               ! rotate right
        RAL <>   := I <9>,               ! rotate left
        RT <>    := I <10>,              ! rotate twice
        IAC <>   := I <11>,              ! increment AC
        OSR <>   := I <9>,               ! logical or AC with SWITCHES
        HLT <>   := I <10>,              ! halt the processor

** operand.classes **
    EADD¥effective.address <0:11>,

    EADD.CAL := begin
        decode PB => begin
            '0 := EADD = '00000@PA,
            '1 := EADD = LAST.PC<0:4>@PA
        end next
        if IB => begin
            EMAR = EADD next
            M.READ () next
            if EADD<0:8> eql #001 => begin
                EMDR = IMDR + 1 next
                M.WRITE ();
                IMDR = IMDR + 1
            end next
            EADD = IMDR
        end
    end,

** execution.processes **
    INPUT.OUTPUT := begin
        decode I<3:11> => begin
            #001¥ion := begin            ! turn Interrupt ON
                I.STATE = '1 next
                restart INTERPRET
            end,
            #002¥iof := begin            ! turn Interrupt OFF
                I.STATE = '0
            end,
            otherwise := no.op()         ! not implemented
        end
    end,

    SKIP <>,

    SKIP.GROUP := begin
        SKIP = '0 next
        decode IS => begin
            '0 := begin
```

```
            if SNL and L => SKIP = '1;
            if SZA and (AC eql 0) => SKIP = '1;
            if SMA and (AC lss 0) => SKIP = '1
        end,
        '1 := begin
            if SZL@SNA@SPA eql 0 => SKIP = '1;
            if SZL and (not L) => SKIP = '1;
            if SNA and (AC neq 0) => SKIP = '1;
            if SPA and (AC geq 0) => SKIP = '1
        end
    end next
    if SKIP => PC = PC + 1              ! SKIP
end,

OPERATE := begin
    decode GROUP => begin
        '0 := begin                     ! GROUP 1
            if CLA => AC = 0;
            if CLL => L = '0 next
            if CMA => AC = not AC;
            if CML => L = not L next
            if IAC => LAC = LAC + 1 next
            decode RT => begin
                '0 := begin
                    if RAL => LAC = LAC slr 1;
                    if RAR => LAC = LAC srr 1
                end,
                '1 := begin
                    if RAL => LAC = LAC slr 2;
                    if RAR => LAC = LAC srr 2
                end
            end
        end,
        '1 := begin                     ! GROUP 2 and 3
            decode I<11> => begin
                '0 := begin             ! GROUP 2
                    SKIP.GROUP() next
                    if CLA => AC = 0 next
                    if OSR => AC = AC or SWITCHES next
                    if HLT => stop ()
                end,
                '1 := begin             ! GROUP 3
                    if CLA => AC = 0 next
                    no.op ( )           ! eae GROUP
                end
            end
        end
    end
end,
```



```
EXECUTE := begin
    decode OP => begin
        #0¥and := begin
            EADD.CAL () next
            EMAR = EADD next
            M.READ () next
            AC = AC and IMDR
        end,
        #1¥tad := begin
            EADD.CAL () next
            EMAR = EADD next
            M.READ () next
            LAC = LAC + '0@IMDR
        end,
        #2¥isz := begin
            EADD.CAL () next
            EMAR = EADD next
            M.READ () next
            EMDR = IMDR + 1 next
            M.WRITE ();
            begin
                IMDR = IMDR + 1 next
                if IMDR eql 0 => PC = PC + 1
            end
        end,
        #3¥dca := begin
            EADD.CAL () next
            EMAR = EADD;
            EMDR = AC next
            M.WRITE ();
            AC = 0
        end,
        #4¥jms := begin
            EADD.CAL () next
            EMAR = EADD;
            EMDR = PC next
            M.WRITE ();
            PC = EADD + 1
        end,
        #5¥jmp := begin
            EADD.CAL () next
            PC = EADD
        end,
        #6¥iot := INPUT.OUTPUT (),
        #7¥opr := OPERATE ()
    end
end,

** interpretation.process **
    main INTERPRET := begin
```

```
        repeat  begin
            EMAR = PC;
            LAST.PC = PC next
            M.READ ();
            PC = PC + 1 next
            I = IMDR next
            EXECUTE () next
            if I.STATE and I.REQUEST => begin
                EMAR = 0;
                EMDR = PC next
                M.WRITE ();
                PC = 1
            end
        end
    end
end
```


(c) CAP3

```
CAP3 := begin                          ! Controller for Array Processor
** interface **
    ext.register FPP.OP <0:3> {RN:1},          ! FPP operation code
    ext.register FPP.XR <0:31> {RN:2},         ! FPP X register
    ext.register FPP.YR <0:31> {RN:3},         ! FPP Y register

    ext.module FPP.EXE {MN:3; PN:1} := begin
        decode FPP.OP => begin
            1 := IAP.ZR = FPP.XR + FPP.YR,
            2 := IAP.ZR = FPP.XR - FPP.YR,
            3 := IAP.ZR = FPP.XR * FPP.YR,
            4 := IAP.ZR = FPP.XR / FPP.YR
        end
    end,

    ext.local MM [0:4095] <0:31>,              ! MMU main memory
    ext.register MMU.A0 <0:19> {RN:4},         ! MMU address register
    ext.register MMU.A1 <0:19> {RN:5},         ! MMU address register
    ext.register MMU.A2 <0:19> {RN:6},         ! MMU address register
    ext.register MMU.D0 <0:31> {RN:7},         ! MMU data register
    ext.register MMU.D1 <0:31> {RN:8},         ! MMU data register
    ext.register MMU.D2 <0:31> {RN:9},         ! MMU data register

ext.module MMU.R0 {MN:4; PN:1} := begin
    IAP.D0 = MM [MMU.A0]
end,

ext.module MMU.W0 {MN:5; PN:1} := begin
    MM [MMU.A0] = MMU.D0
end,

ext.module MMU.R1 {MN:6; PN:1} := begin
    IAP.D1 = MM [MMU.A1]
end,

ext.module MMU.W1 {MN:7; PN:1} := begin
    MM [MMU.A1] = MMU.D1
end,

ext.module MMU.R2 {MN:8; PN:1} := begin
    IAP.D2 = MM [MMU.A2]
end,

ext.module MMU.W2 {MN:9; PN:1} := begin
    MM [MMU.A2] = MMU.D2
end,

IAP.OPC <0:3>,             ! IAP operation code
```

```
    IAP.CNT <0:9>,          ! IAP element count
    IAP.CBA <0:19>,         ! IAP control block address
    IAP.ZR <0:31>,          ! IAP Z register
    IAP.D0 <0:31>,          ! IAP data register
    IAP.D1 <0:31>,          ! IAP data register
    IAP.D2 <0:31>,          ! IAP data register

** working.resource **
    ENDF <>,                ! end flag
    XADD <0:19>,            ! X address
    XOFF <0:9>,             ! X offset value
    YADD <0:19>,            ! Y address
    YOFF <0:9>,             ! Y offset value
    ZADD <0:19>,            ! Z address
    ZOFF <0:9>,             ! Z offset value

** execution **
MAIN.LOOP := begin
    repeat begin
        begin
            FPP.XR = IAP.D0 next
            FPP.YR = IAP.D1 next
            FPP.EXE ()
        end;
        begin
            MMU.D2 = IAP.ZR next
            MMU.A2 = ZADD next
            MMU.W2 () next
            ZADD = ZADD + ZOFF
        end;
        begin
            MMU.A0 = XADD next
            MMU.R0 () next
            XADD = XADD + XOFF
        end;
        begin
            MMU.A1 = YADD next
            MMU.R1 () next
            YADD = YADD + YOFF
        end;
        begin
            IAP.CNT = IAP.CNT - 1 next
            ENDF = IAP.CNT eql 0
        end next

        if ENDF => leave MAIN.LOOP
    end
end,

main IAPC.EXE := begin
```



```
INIT := begin
    YADD = IAP.CBA + 2 next
    ZADD = IAP.CBA + 4 next
    begin
        MMU.A0 = IAP.CBA next
        MMU.R0 () next
        IAP.CBA = IAP.CBA + 1 next
        XADD = IAP.D0 next
        MMU.A0 = IAP.CBA next
        MMU.R0 () next
        XOFF = IAP.D0
    end;
    begin
        MMU.A1 = YADD next
        MMU.R1 () next
        YADD = YADD + 1 next
        MMU.A1 = YADD next
        YADD = IAP.D1 next
        MMU.R1 () next
        YOFF = IAP.D1
    end;
    begin
        MMU.A2 = ZADD next
        MMU.R2 () next
        ZADD = ZADD + 1 next
        MMU.A2 = ZADD next
        ZADD = IAP.D2 next
        MMU.R2 () next
        ZOFF = IAP.D2
    end next

    begin
        MMU.A0 = XADD next
        MMU.R0 () next
        XADD = XADD + XOFF
    end;
    begin
        MMU.A1 = YADD next
        MMU.R1 () next
        YADD = YADD + YOFF
    end;
    begin
        IAP.CNT = IAP.CNT - 1 next
        FPP.OP = IAP.OPC
    end next

    begin
        FPP.XR = IAP.D0 next
        FPP.YR = IAP.D1 next
        FPP.EXE ()
```

```
    end;
    begin
        MMU.A0 = XADD next
        MMU.R0 () next
        XADD = XADD + XOFF
    end;
    begin
        MMU.A1 = YADD next
        MMU.R1 () next
        YADD = YADD + YOFF
    end;
    IAP.CNT = IAP.CNT - 1
end next

MAIN.LOOP () next

TERM := begin
    begin
        FPP.XR = IAP.D0 next
        FPP.YR = IAP.D1 next
        FPP.EXE ()
    end;
    begin
        MMU.D2 = IAP.ZR next
        MMU.A2 = ZADD next
        MMU.W2 ()next
        ZADD = ZADD + ZOFF
    end next

    MMU.D2 = IAP.ZR next
    MMU.A2 = ZADD next
    MMU.W2 ()
  end
end
end
```


(d) QSORT4

```
QSORT4 := begin

** external.environment **

    ext.local       DM [0:255]<0:15>,          ! Data Memory
    ext.register    DMA <0:7> {RN:1},          ! DM address register
    ext.register    DMDO <0:15> {RN:2},        ! DM output register
    ext.local       STACK [0:255]<0:15>,       ! stack
    ext.local       SP <0:7>,                  ! stack pointer
    ext.register    STDO <0:15> {RN:3},        ! stack output register

    ext.module DMR {MN:4; PN:1} := begin
        DMDI = DM [DMA]
    end,

    ext.module DMW {MN:5; PN:1} := begin
        DM [DMA] = DMDO
    end,

    ext.module PUSH {MN:6; PN:1} := begin
        STACK [SP] = STDO next
        SP = SP + 1
    end,

    ext.module POP {MN:7; PN:1} := begin
        decode SP eql 0 => begin
            '0 :=  (SP = SP - 1 next STDI = STACK [SP]),
            '1 :=  STDI = "FFFF
        end
    end,

** internal.registers **

    DMDI <0:15>,                           ! DM input register
    STDI <0:15>,                           ! stack input register
    BF <0:7>,                              ! busy flag
        field BF.A <> := BF <0>,
        field BF.B <> := BF <1>,
        field BF.C <> := BF <2>,
        field BF.D <> := BF <3>,
        field BF.E <> := BF <4>,
        field BF.F <> := BF <5>,
        field BF.G <> := BF <6>,
        field BF.H <> := BF <7>,

    AD.A <0:7>,      DT.A <0:15>,          ! for data memory
    AREA.A <0:15>,                         ! for stack
    L.A <0:7>,       R.A <0:7>,            ! pointer
```

```
    MD.A <0:15>,    WD.A <0:15>,        ! middle data, working data

    AD.B <0:7>,     DT.B <0:15>,        ! for data memory
    AREA.B <0:15>,                      ! for stack
    L.B <0:7>,      R.B <0:7>,          ! pointer
    MD.B <0:15>,    WD.B <0:15>,        ! middle data, working data

    AD.C <0:7>,     DT.C <0:15>,        ! for data memory
    AREA.C <0:15>,                      ! for stack
    L.C <0:7>,      R.C <0:7>,          ! pointer
    MD.C <0:15>,    WD.C <0:15>,        ! middle data, working data

    AD.D <0:7>,     DT.D <0:15>,        ! for data memory
    AREA.D <0:15>,                      ! for stack
    L.D <0:7>,      R.D <0:7>,          ! pointer
    MD.D <0:15>,    WD.D <0:15>,        ! middle data, working data

** execution.process **

    critical DMRW (OPM<0>, PNM<0:3>) := begin
        decode OPM => begin
            '0¥read := begin
                decode PNM => begin
                    "0 := DMA = AD.A,
                    "1 := DMA = AD.B,
                    "2 := DMA = AD.C,
                    "3 := DMA = AD.D
                end next
                DMR () next             ! read data from DM
                decode PNM => begin
                    "0 := DT.A = DMDI,
                    "1 := DT.B = DMDI,
                    "2 := DT.C = DMDI,
                    "3 := DT.D = DMDI
                end
            end,
            '1¥write := begin
                decode PNM => begin
                    "0 := (DMA = AD.A next DMDO = DT.A),
                    "1 := (DMA = AD.B next DMDO = DT.B),
                    "2 := (DMA = AD.C next DMDO = DT.C),
                    "3 := (DMA = AD.D next DMDO = DT.D)
                end next
                DMW ()                  ! write data to DM
            end
        end
    end,

    critical PUSHPOP (OPS<>, PNS<0:3>) := begin
        decode OPS => begin
```



```
            '0¥push := begin
                decode PNS => begin
                    "0 := STDO = AREA.A,
                    "1 := STDO = AREA.B,
                    "2 := STDO = AREA.C,
                    "3 := STDO = AREA.D
                end next
                PUSH ()                 ! push down area data
            end,
            '1¥pop := begin
                POP () next
                if (STDI eql "FFFF) and (BF neq "00) => STDI = "0000 next
                decode PNS => begin
                    "0 := AREA.A = STDI,
                    "1 := AREA.B = STDI,
                    "2 := AREA.C = STDI,
                    "3 := AREA.D = STDI
                end
            end
        end
    end,

    process SPLIT.A := begin
        PUSHPOP ('1, "0) next           ! pop up area data
        if AREA.A eql "0000 => restart SPLIT.A next
        if AREA.A eql "FFFF => leave SPLIT.A next

        BF.A = '1 next
        L.A = AREA.A<0:7> next  R.A = AREA.A<8:15> next
        AD.A = (L.A+R.A) sr0 1 next DMRW ('0, "0) next MD.A = DT.A next
        LOOP1 := begin
            LOOP2 := begin
                AD.A = L.A next  DMRW ('0, "0) next
                if DT.A lss MD.A => (L.A = L.A + 1 next  restart LOOP2)
            end next
            LOOP3 := begin
                AD.A = R.A next  DMRW ('0, "0) next
                if MD.A lss DT.A => (R.A = R.A - 1 next  restart LOOP3)
            end next
            if L.A leq R.A => begin     ! change items
                AD.A = L.A next  DMRW ('0, "0) next WD.A = DT.A next
                AD.A = R.A next  DMRW ('0, "0) next
                AD.A = L.A next  DMRW ('1, "0) next
                AD.A = R.A next  DT.A = WD.A next DMRW ('1, "0) next
                L.A = L.A + 1 next R.A = R.A - 1
            end next
            if L.A gtr R.A => leave LOOP1 next
            restart LOOP1
        end next
        WD.A = AREA.A next
```

```
    if L.A lss WD.A<8:15> =>
        (AREA.A = L.A @ WD.A<8:15> next PUSHPOP ('0, "0)) next
    if WD.A<0:7> lss R.A =>
        (AREA.A = WD.A<0:7> @ R.A next PUSHPOP ('0, "0)) next
    BF.A = '0 next
    restart SPLIT.A
end,

process SPLIT.B := begin
    PUSHPOP ('1, "1) next          ! pop up area data
    if AREA.B eql "0000 => restart SPLIT.B next
    if AREA.B eql "FFFF => leave SPLIT.B next

    BF.B = '1 next
    L.B = AREA.B<0:7> next  R.B = AREA.B<8:15> next
    AD.B = (L.B+R.B) sr0 1 next DMRW ('0, "1) next MD.B = DT.B next
    LOOP1 := begin
        LOOP2 := begin
            AD.B = L.B next  DMRW ('0, "1) next
            if DT.B lss MD.B => (L.B = L.B + 1 next  restart LOOP2)
        end next
        LOOP3 := begin
            AD.B = R.B next  DMRW ('0, "1) next
            if MD.B lss DT.B => (R.B = R.B - 1 next  restart LOOP3)
        end next
        if L.B leq R.B => begin    ! change items
            AD.B = L.B next  DMRW ('0, "1) next WD.B = DT.B next
            AD.B = R.B next  DMRW ('0, "1) next
            AD.B = L.B next  DMRW ('1, "1) next
            AD.B = R.B next  DT.B = WD.B next DMRW ('1, "1) next
            L.B = L.B + 1 next R.B = R.B - 1
        end next
        if L.B gtr R.B => leave LOOP1 next
        restart LOOP1
    end next
    WD.B = AREA.B next
    if L.B lss WD.B<8:15> =>
        (AREA.B = L.B @ WD.B<8:15> next PUSHPOP ('0, "1)) next
    if WD.B<0:7> lss R.B =>
        (AREA.B = WD.B<0:7> @ R.B next PUSHPOP ('0, "1)) next
    BF.B = '0 next
    restart SPLIT.B
end,

process SPLIT.C := begin
    PUSHPOP ('1, "2) next          ! pop up area data
    if AREA.C eql "0000 => restart SPLIT.C next
    if AREA.C eql "FFFF => leave SPLIT.C next

    BF.C = '1 next
```



```
    L.C = AREA.C<0:7> next  R.C = AREA.C<8:15> next
    AD.C = (L.C+R.C) sr0 1 next DMRW ('0, "2) next MD.C = DT.C next
    LOOP1 := begin
        LOOP2 := begin
            AD.C = L.C next  DMRW ('0, "2) next
            if DT.C lss MD.C => (L.C = L.C + 1 next  restart LOOP2)
        end next
        LOOP3 := begin
            AD.C = R.C next  DMRW ('0, "2) next
            if MD.C lss DT.C => (R.C = R.C - 1 next  restart LOOP3)
        end next
        if L.C leq R.C => begin    ! change items
            AD.C = L.C next  DMRW ('0, "2) next WD.C = DT.C next
            AD.C = R.C next  DMRW ('0, "2) next
            AD.C = L.C next  DMRW ('1, "2) next
            AD.C = R.C next  DT.C = WD.C next DMRW ('1, "2) next
            L.C = L.C + 1 next R.C = R.C - 1
        end next
        if L.C gtr R.C => leave LOOP1 next
        restart LOOP1
    end next
    WD.C = AREA.C next
    if L.C lss WD.C<8:15> =>
        (AREA.C = L.C @ WD.C<8:15> next PUSHPOP ('0, "2)) next
    if WD.C<0:7> lss R.C =>
        (AREA.C = WD.C<0:7> @ R.C next PUSHPOP ('0, "2)) next
    BF.C = '0 next
    restart SPLIT.C
end,

process SPLIT.D := begin
    PUSHPOP ('1, "3) next          ! pop up area data
    if AREA.D eql "0000 => restart SPLIT.D next
    if AREA.D eql "FFFF => leave SPLIT.D next

    BF.D = '1 next
    L.D = AREA.D<0:7> next  R.D = AREA.D<8:15> next
    AD.D = (L.D+R.D) sr0 1 next DMRW ('0, "3) next MD.D = DT.D next
    LOOP1 := begin
        LOOP2 := begin
            AD.D = L.D next  DMRW ('0, "3) next
            if DT.D lss MD.D => (L.D = L.D + 1 next  restart LOOP2)
        end next
        LOOP3 := begin
            AD.D = R.D next  DMRW ('0, "3) next
            if MD.D lss DT.D => (R.D = R.D - 1 next  restart LOOP3)
        end next
        if L.D leq R.D => begin    ! change items
            AD.D = L.D next  DMRW ('0, "3) next WD.D = DT.D next
            AD.D = R.D next  DMRW ('0, "3) next
```

```
            AD.D = L.D next  DMRW ('1, "3) next
            AD.D = R.D next  DT.D = WD.D next DMRW ('1, "3) next
            L.D = L.D + 1 next R.D = R.D - 1
        end next
        if L.D gtr R.D => leave LOOP1 next
        restart LOOP1
    end next
    WD.D = AREA.D next
    if L.D lss WD.D<8:15> =>
        (AREA.D = L.D @ WD.D<8:15> next PUSHPOP ('0, "3)) next
    if WD.D<0:7> lss R.D =>
        (AREA.D = WD.D<0:7> @ R.D next PUSHPOP ('0, "3)) next
    BF.D = '0 next
    restart SPLIT.D
end,

main QSORT4.EXE := begin
    SPLIT.A () next
    SPLIT.B () next
    SPLIT.C () next
    SPLIT.D () next
    LOOP := begin
        if BF eql "00 => restart LOOP
    end
end
        end
    end
end
```


付録Ⅲ－２　ＧＥＮＭＡＧチップのコンパイレーション

（ａ）ＩＳＰＳによる動作記述

```
GENMAG {WDT:8} := begin
** register **

    ext.register RES <0:7> {RN:1},      ! result
    A.AAB <0:7>,                        ! input data A, |A|
    B.BAB <0:7>,                        ! input data B, |B|
    G <0:7>,                            ! G = max(|A|,|B|)
    L.SQS <0:7>,                        ! L = min(|A|,|B|), 7/8G + 1/2L

** execution.process **
    main GENMAG.EXE := begin
        if A.AAB lss 0 => A.AAB = - A.AAB;
        if B.BAB lss 0 => B.BAB = - B.BAB next
        decode A.AAB geq B.BAB => begin
            '0 := (G = B.BAB next
                   L.SQS = A.AAB),
            '1 := (G = A.AAB next
                   L.SQS = B.BAB)
        end next
        L.SQS = G - (G sr0 3) + (L.SQS sr0 1) next
        decode G geq L.SQS => begin
            '0 := RES = L.SQS,
            '1 := RES = G
        end
    end
end
```

（b）モジュール動作の記述

```
GENMAG {WDT:8} := begin

** register **
    ext.register RES <0:7> {RN:1},          ! result
    A.AAB <0:7>,                            ! input data A, |A|
    B.BAB <0:7>,                            ! input data B, |B|
    G <0:7>,                                ! G = max(|A|,|B|)
    L.SQS <0:7>,                            ! L = min(|A|,|B|), 7/8G + 1/2L

** execution.process **
    main GENMAG.EXE := begin
        (WRG.1 = COMP.8.A (2¥lss, A.AAB, 0) next
         if{A} WRG.1 => A.AAB = ADDER.8.A (1¥sub, 0, A.AAB) );
        (WRG.3 = COMP.8.B (2¥lss, B.BAB, 0) next
         if{B} WRG.3 => B.BAB = ADDER.8.B (1¥sub, 0, B.BAB) ) next
        (WRG.1 = COMP.8.A (5¥geq, A.AAB, B.BAB) next
         decode WRG.1 => begin
            '0 := (G = LTRANS.8 (B.BAB) next
                   L.SQS = LTRANS.8 (A.AAB) ),
            '1 := (G = LTRANS.8 (A.AAB) next
                   L.SQS = LTRANS.8 (B.BAB) )
         end) next
        (WRG.11 = SHIFT.8 (5¥sr0, G, 3) next
         WRG.11 = ADDER.8.A (1¥sub, G, WRG.11) next
         WRG.10 = SHIFT.8 (5¥sr0, L.SQS, 1) next
         L.SQS = ADDER.8.A (0¥add, WRG.11, WRG.10)) next
        (WRG.1 = COMP.8.A (5¥geq, G, L.SQS) next
         decode WRG.1 => begin
            '0 := RES = LTRANS.8 (L.SQS) ,
            '1 := RES = LTRANS.8 (G)
         end)
    end,

** imacs.generated **

WRG.1 <1:1>,
WRG.3 <1:1>,
WRG.10 <1:8>,
WRG.11 <1:8>,

module COMP.8.B (OP<0:3>, D1<1:8>, D2<1:8>) <> := begin
    decode OP => begin
        0¥eql := COMP.8.B = D1 eql D2,
        1¥neq := COMP.8.B = D1 neq D2,
        2¥lss := COMP.8.B = D1 lss D2,
        3¥leq := COMP.8.B = D1 leq D2,
        4¥gtr := COMP.8.B = D1 gtr D2,
        5¥geq := COMP.8.B = D1 geq D2
    end
end,

module COMP.8.A (OP<0:3>, D1<1:8>, D2<1:8>) <> := begin
    decode OP => begin
        0¥eql := COMP.8.A = D1 eql D2,
        1¥neq := COMP.8.A = D1 neq D2,
        2¥lss := COMP.8.A = D1 lss D2,
        3¥leq := COMP.8.A = D1 leq D2,
        4¥gtr := COMP.8.A = D1 gtr D2,
        5¥geq := COMP.8.A = D1 geq D2
    end
end,

module LTRANS.8 (D1<1:8>) <1:8> := begin
    LTRANS.8 = D1
end,

module SHIFT.8 (OP<0:7>, D1<1:8>, D2<1:8>) <1:8> := begin
    decode OP => begin
        0¥sl0 := SHIFT.8 = D1 sl0 D2,
        1¥sl1 := SHIFT.8 = D1 sl1 D2,
        2¥slr := SHIFT.8 = D1 slr D2,
        3¥sld := SHIFT.8 = D1 sld D2,
        4¥sli := SHIFT.8 = D1 sli D2,
        5¥sr0 := SHIFT.8 = D1 sr0 D2,
        6¥sr1 := SHIFT.8 = D1 sr1 D2,
        7¥srr := SHIFT.8 = D1 srr D2,
        8¥srd := SHIFT.8 = D1 srd D2,
        9¥sri := SHIFT.8 = D1 sri D2
    end
end,

module ADDER.8.A (OP<0:3>, D1<1:8>, D2<1:8>) <1:9> := begin
    decode OP => begin
        0¥add := ADDER.8.A = D1 + D2,
        1¥sub := ADDER.8.A = D1 - D2
    end
end,

module ADDER.8.B (OP<0:3>, D1<1:8>, D2<1:8>) <1:9> := begin
    decode OP => begin
        0¥add := ADDER.8.B = D1 + D2,
        1¥sub := ADDER.8.B = D1 - D2
    end
end
end
```

（c）コンパイル・リスト

```
      *****   ACTION-LIST   ****

*** GENMAG_EXE ***

1   (FORK (4 1))
   ===> ((2 (10 1)) (5 (9 1)))
2   (COMP_8_A (10 1))          (ACSET 2 ((A_AAB 2)) 0)              ((WRG_1 6))
   ===> (3 (5 1))
3   (IF_A (5 1))               ((WRG_1 6))
   ===> ((4 (13 1)) (8 (7 1)))
4   (ADDER_8_A (13 1))         (ACSET 1 0 ((A_AAB 2)))              ((A_AAB 2))
   ===> (8 (7 1))
5   (COMP_8_B (9 1))           (ACSET 2 ((B_BAB 3)) 0)              ((WRG_3 7))
   ===> (6 (6 1))
6   (IF_B (6 1))               ((WRG_3 7))
   ===> ((7 (14 1)) (8 (7 1)))
7   (ADDER_8_B (14 1))         (ACSET 1 0 ((B_BAB 3)))              ((B_BAB 3))
   ===> (8 (7 1))
8   (SYNC (7 1))               2
   ===> (9 (10 2))
9   (COMP_8_A (10 2))          (ACSET 5 ((A_AAB 2)) ((B_BAB 3)))    ((WRG_1 6))
   ===> (10 (8 1))
10  (DECODE (8 1))             ((WRG_1 6))
   ===> ((~0 (11 (11 1))) (~1 (13 (11 3))))
11  (LTRANS_8 (11 1))          (ACSET ((B_BAB 3)))                  ((G 4))
   ===> (12 (11 2))
12  (LTRANS_8 (11 2))          (ACSET ((A_AAB 2)))                  ((L_SQS 5))
   ===> (15 (12 1))
13  (LTRANS_8 (11 3))          (ACSET ((A_AAB 2)))                  ((G 4))
   ===> (14 (11 4))
14  (LTRANS_8 (11 4))          (ACSET ((B_BAB 3)))                  ((L_SQS 5))
   ===> (15 (12 1))
15  (SHIFT_8 (12 1))           (ACSET 5 ((G 4)) 3)                  ((WRG_11 9))
   ===> (16 (13 2))
16  (ADDER_8_A (13 2))         (ACSET 1 ((G 4)) ((WRG_11 9)))       ((WRG_11 9))
   ===> (17 (12 2))
17  (SHIFT_8 (12 2))           (ACSET 5 ((L_SQS 5)) 1)              ((WRG_10 8))
   ===> (18 (13 3))
18  (ADDER_8_A (13 3))         (ACSET 0 ((WRG_11 9)) ((WRG_10 8)))  ((L_SQS 5))
   ===> (19 (10 3))
19  (COMP_8_A (10 3))          (ACSET 5 ((G 4)) ((L_SQS 5)))        ((WRG_1 6))
   ===> (20 (8 2))
20  (DECODE (8 2))             ((WRG_1 6))
   ===> ((~0 (21 (11 5))) (~1 (22 (11 6))))
21  (LTRANS_8 (11 5))          (ACSET ((L_SQS 5)))                  ((RES 1))
   ===> (23 (3 1))
22  (LTRANS_8 (11 6))          (ACSET ((G 4)))                      ((RES 1))
   ===> (23 (3 1))
```



```
23  (LEAVE (3 1))              (GENMAG_EXE (1 (4 1)))
   ===> NIL

*** START ***

24  (ACTIVATE (2 1))           (GENMAG_EXE (1 (4 1)))
   ===> (25 (1 1))
25  (STOP (1 1))
   ===> NIL


      *****   PROCEDURE-TABLE   *****

PN = 1        GENMAG_EXE      (FIRST LAST): (1 23)      TYPE: (MAIN)

PN = 2        START           (FIRST LAST): (24 25)     TYPE: NIL


      *****   REGISTER-TABLE   *****

RN = 2        A_AAB           # OF BITS: 8              CONNECTION: (7 0)

RN = 3        B_BAB           # OF BITS: 8              CONNECTION: (7 0)

RN = 4        G               # OF BITS: 8              CONNECTION: (7 0)

RN = 5        L_SQS           # OF BITS: 8              CONNECTION: (7 0)

RN = 6        WRG_1           # OF BITS: 1              CONNECTION: 0

RN = 7        WRG_3           # OF BITS: 1              CONNECTION: 0

RN = 8        WRG_10          # OF BITS: 8              CONNECTION: (7 0)

RN = 9        WRG_11          # OF BITS: 8              CONNECTION: (7 0)


      *****   MODULE-TABLE   *****

MN = 2        ACTIVATE        TYPE: ACTIVATE

24  (ACTIVATE (2 1))    (GENMAG_EXE (1 (4 1)))        ===> (25 (1 1))


MN = 3        LEAVE           TYPE: LEAVE

23  (LEAVE (3 1))       (GENMAG_EXE (1 (4 1)))


MN = 4        FORK            TYPE: FORK
```

```
1   (FORK (4 1))          ===> ((2 (10 1)) (5 (9 1)))


MN = 5        IF_A            TYPE: IF

3   (IF_A (5 1))          ((WRG_1 6))     ===> ((4 (13 1)) (8 (7 1)))


MN = 6        IF_B            TYPE: IF

6   (IF_B (6 1))          ((WRG_3 7))     ===> ((7 (14 1)) (8 (7 1)))


MN = 7        SYNC            TYPE: SYNC

8   (SYNC (7 1))          2               ===> (9 (10 2))


MN = 8        DECODE          TYPE: DECODE

10  (DECODE (8 1))        ((WRG_1 6))     ===> ((~0 (11 (11 1)))(~1 (13 (11 3))))
20  (DECODE (8 2))        ((WRG_1 6))     ===> ((~0 (21 (11 5)))(~1 (22 (11 6))))


MN = 9        COMP_8_B        TYPE: COMP          RESULT_BIT: (<F>)

(FCSET (EHEAD OP NIL NIL (: 0 3)) (EHEAD D1 NIL NIL (: 1 8))
                                  (EHEAD D2 NIL NIL (: 1 8)))

5   (COMP_8_B (9 1))      2         ((B_BAB 3))     0               ((WRG_3 7))
    ===> (6 (6 1))


MN = 10       COMP_8_A        TYPE: COMP          RESULT_BIT: (<F>)

(FCSET (EHEAD OP NIL NIL (: 0 3)) (EHEAD D1 NIL NIL (: 1 8))
                                  (EHEAD D2 NIL NIL (: 1 8)))

2   (COMP_8_A (10 1))     2         ((A_AAB 2))     0               ((WRG_1 6))
    ===> (3 (5 1))
9   (COMP_8_A (10 2))     5         ((A_AAB 2))     ((B_BAB 3))     ((WRG_1 6))
    ===> (10 (8 1))
19  (COMP_8_A (10 3))     5         ((G 4))         ((L_SQS 5))     ((WRG_1 6))
    ===> (20 (8 2))


MN = 11       LTRANS_8        TYPE: LTRANS        RESULT_BIT: (: 1 8)

(FCSET (EHEAD D1 NIL NIL (: 1 8)))
```



```
11  (LTRANS_8 (11 1))     ((B_BAB 3))     ((G 4))         ===> (12 (11 2))
12  (LTRANS_8 (11 2))     ((A_AAB 2))     ((L_SQS 5))     ===> (15 (12 1))
13  (LTRANS_8 (11 3))     ((A_AAB 2))     ((G 4))         ===> (14 (11 4))
14  (LTRANS_8 (11 4))     ((B_BAB 3))     ((L_SQS 5))     ===> (15 (12 1))
21  (LTRANS_8 (11 5))     ((L_SQS 5))     ((RES 1))       ===> (23 (3 1))
22  (LTRANS_8 (11 6))     ((G 4))         ((RES 1))       ===> (23 (3 1))


MN = 12       SHIFT_8         TYPE: SHIFT         RESULT_BIT: (: 1 8)

(FCSET (EHEAD OP NIL NIL (: 0 7)) (EHEAD D1 NIL NIL (: 1 8))
                                  (EHEAD D2 NIL NIL (: 1 8)))

15  (SHIFT_8 (12 1))      5         ((G 4))         3               ((WRG_11 9))
    ===> (16 (13 2))
17  (SHIFT_8 (12 2))      5         ((L_SQS 5))     1               ((WRG_10 8))
    ===> (18 (13 3))


MN = 13       ADDER_8_A       TYPE: ADDER         RESULT_BIT: (: 1 9)

(FCSET (EHEAD OP NIL NIL (: 0 3)) (EHEAD D1 NIL NIL (: 1 8))
                                  (EHEAD D2 NIL NIL (: 1 8)))

4   (ADDER_8_A (13 1))    1         0               ((A_AAB 2))     ((A_AAB 2))
    ===> (8 (7 1))
16  (ADDER_8_A (13 2)     1         ((G 4))         ((WRG_11 9))    ((WRG_11 9))
    ===> (17 (12 2))
18  (ADDER_8_A (13 3))    0         ((WRG_11 9))    ((WRG_10 8))    ((L_SQS 5))
    ===> (19 (10 3))


MN = 14       ADDER_8_B       TYPE: ADDER         RESULT_BIT: (: 1 9)

(FCSET (EHEAD OP NIL NIL (: 0 3)) (EHEAD D1 NIL NIL (: 1 8))
                                  (EHEAD D2 NIL NIL (: 1 8)))

7   (ADDER_8_B (14 1))    1         0               ((B_BAB 3))     ((B_BAB 3))
    ===> (8 (7 1))
```

（d）ＤＰＬによるレイアウト記述

```
(LOADDPL SC-LIB)

(DEFCELL GENMAG ()
   (PROG (WDT WRN WMN WPI W-LIST CT_2 CT_3 CT_4 CT_5 CT_6 CT_7 CT_8
                                 CT_9 CT_10 CT_11 CT_12 CT_13 CT_14)

      (DV WDT        8)
      (DV WRN        4)
      (DV WMN        4)
      (DV WPI        3)

      (PARTQ A_AAB REGISTER (RN 2) (BIT 8) (CONNECT (LIST 7 0)))
      (VGENQ A_AAB)

      (PARTQ B_BAB REGISTER (RN 3) (BIT 8) (CONNECT (LIST 7 0)))
      (VGENQ B_BAB)

      (PARTQ G REGISTER (RN 4) (BIT 8) (CONNECT (LIST 7 0)))
      (VGENQ G)

      (PARTQ L_SQS REGISTER (RN 5) (BIT 8) (CONNECT (LIST 7 0)))
      (VGENQ L_SQS)

      (PARTQ WRG_1 REGISTER (RN 6) (BIT 1) (CONNECT 0))
      (VGENQ WRG_1)

      (PARTQ WRG_3 REGISTER (RN 7) (BIT 1) (CONNECT 0))
      (VGENQ WRG_3)

      (PARTQ WRG_10 REGISTER (RN 8) (BIT 8) (CONNECT (LIST 7 0)))
      (VGENQ WRG_10)

      (PARTQ WRG_11 REGISTER (RN 9) (BIT 8) (CONNECT (LIST 7 0)))
      (VGENQ WRG_11)

      (DV CT_2       ((1 1 1 (4 1) (1 1) 0)))
      (DV CT_3       ((1 1 1 (0 0) (0 0) 0)))
      (PARTQ PRO-CONT PRO-CONT (MN_ACTIVATE 2) (CT_ACTIVATE CT_2)
                               (MN_LEAVE 3) (CT_LEAVE CT_3))
      (VGENQ PRO-CONT)

      (DV CT_4       ((1 (10 1) (9 1))))
      (PARTQ FORK FORK (MN 4) (C_TABLE CT_4))
      (VGENQ FORK)

{;; ((0) ((WRG_1 6))) }
      (DV CT_5       ((1 0 (13 1) (7 1))))
      (PARTQ IF_A IF (MN 5) (IN 1) (C_TABLE CT_5))
```



```
      (VGENQ IF_A)

{;; ((0) ((WRG_3 7))) }
      (DV CT_6       ((1 0 (14 1) (7 1))))
      (PARTQ IF_B IF (MN 6) (IN 1) (C_TABLE CT_6))
      (VGENQ IF_B)

      (DV CT_7       ((1 2 (10 2))))
      (PARTQ SYNC SYNC (MN 7) (C_TABLE CT_7))
      (VGENQ SYNC)

{;; ((0) ((WRG_1 6))) }
      (DV CT_8       ((1 0 ((0 (11 1)) (1 (11 3))))
                      (2 0 ((0 (11 5)) (1 (11 6))))))
      (PARTQ DECODE DECODE (MN 8) (BIT 1) (IN 1) (C_TABLE CT_8))
      (VGENQ DECODE)

{;; ((0) ((B_BAB 3))) }
{;; ((0) 0) }
      (DV CT_9       ((1 2 0 0 7 (6 1))))
      (PARTQ COMP_8_B COMP (MN 9) (BIT 8) (IN_L 1) (IN_R 1) (C_TABLE CT_9))
      (VGENQ COMP_8_B)

{;; ((1) ((A_AAB 2)) ((G 4))) }
{;; ((2) 0 ((B_BAB 3)) ((L_SQS 5))) }
      (DV CT_10      ((1 2 0 0 6 (5 1)) (2 5 0 1 6 (8 1)) (3 5 1 2 6 (8 2))))
      (PARTQ COMP_8_A COMP (MN 10) (BIT 8) (IN_L 2) (IN_R 3) (C_TABLE CT_10))
      (VGENQ COMP_8_A)

{;; ((3) ((B_BAB 3)) ((A_AAB 2)) ((L_SQS 5)) ((G 4))) }
      (DV CT_11      ((1 0 4 (11 2))
                      (2 1 5 (12 1))
                      (3 1 4 (11 4))
                      (4 0 5 (12 1))
                      (5 2 1 (3 1))
                      (6 3 1 (3 1))))
      (PARTQ LTRANS_8 LTRANS (MN 11) (BIT 8) (IN 4) (C_TABLE CT_11))
      (VGENQ LTRANS_8)

{;; ((1) ((G 4)) ((L_SQS 5))) }
{;; ((1) 3 1) }
      (DV CT_12      ((1 5 0 0 9 (13 2)) (2 5 1 1 8 (13 3))))
      (PARTQ SHIFT_8 SHIFT (MN 12) (BIT 8) (IN_L 2) (IN_R 2) (C_TABLE CT_12))
      (VGENQ SHIFT_8)

{;; ((2) 0 ((G 4)) ((WRG_11 9))) }
{;; ((2) ((A_AAB 2)) ((WRG_11 9)) ((WRG_10 8))) }
      (DV CT_13      ((1 1 0 0 2 (7 1)) (2 1 1 1 9 (12 2))
                                        (3 0 2 2 5 (10 3))))
      (PARTQ ADDER_8_A ADDER (MN 13) (BIT 8) (IN_L 3) (IN_R 3)
```

```
                                                    (C_TABLE CT_13))
        (VGENQ ADDER_8_A)

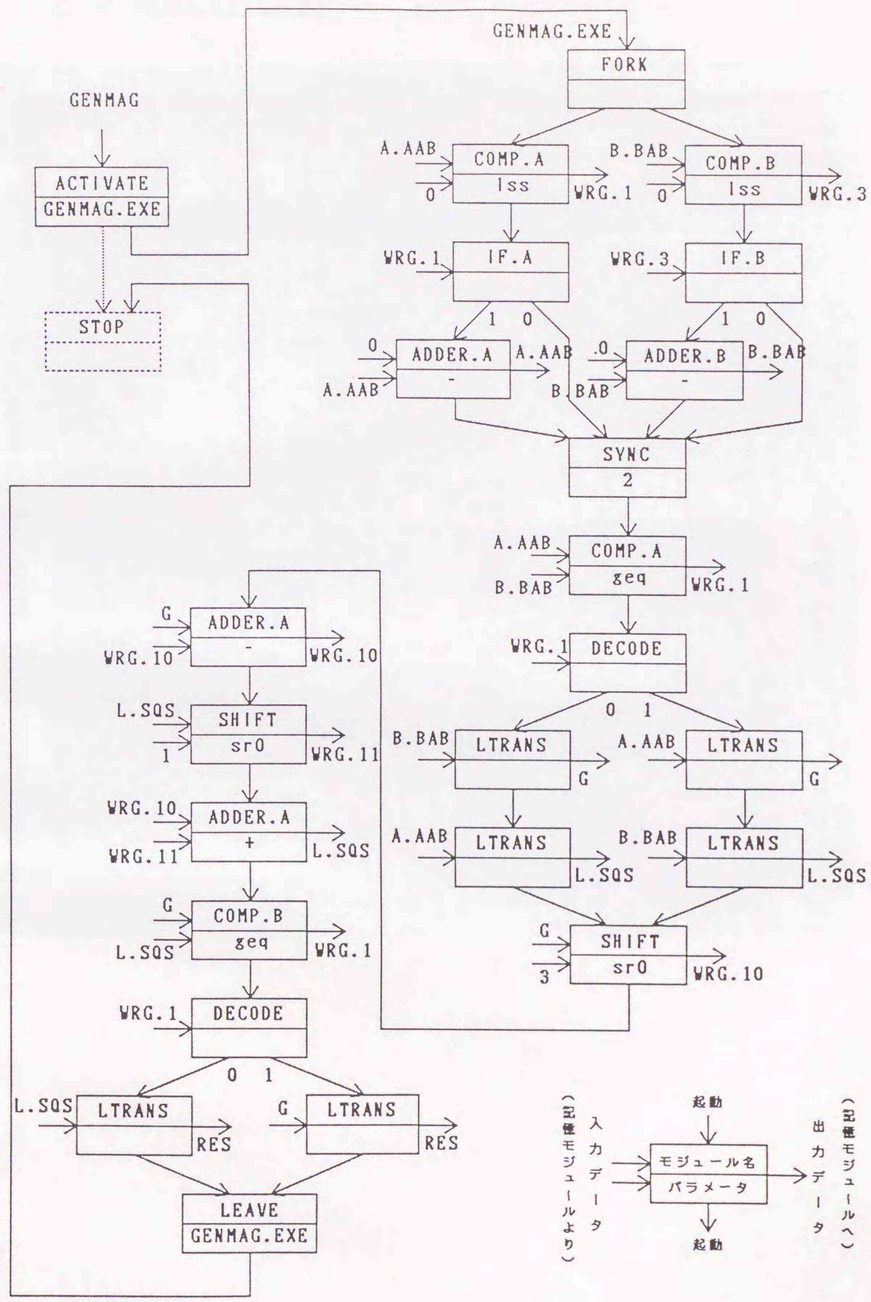
{;; ((0) 0) }
{;; ((0) ((B_BAB 3))) }
        (DV CT_14       ((1 1 0 0 3 (7 1))))
        (PARTQ ADDER_8_B ADDER (MN 14) (BIT 8) (IN_L 1) (IN_R 1)
                                                    (C_TABLE CT_14))
        (VGENQ ADDER_8_B)

        (DV W-LIST      (((WRG_1 (0 0)) (IF_A S_0 (0 0)))
                         ((WRG_3 (0 0)) (IF_B S_0 (0 0)))
                         ((WRG_1 (0 0)) (DECODE S_0 (0 0)))
                         ((B_BAB (7 0)) (COMP_8_B L_0 (7 0)))
                         (GND (COMP_8_B R_0 (7 0)))
                         ((A_AAB (7 0)) (COMP_8_A L_0 (7 0)))
                         ((G (7 0)) (COMP_8_A L_1 (7 0)))
                         (GND (COMP_8_A R_0 (7 0)))
                         ((B_BAB (7 0)) (COMP_8_A R_1 (7 0)))
                         ((L_SQS (7 0)) (COMP_8_A R_2 (7 0)))
                         ((B_BAB (7 0)) (LTRANS_8 S_0 (7 0)))
                         ((A_AAB (7 0)) (LTRANS_8 S_1 (7 0)))
                         ((L_SQS (7 0)) (LTRANS_8 S_2 (7 0)))
                         ((G (7 0)) (LTRANS_8 S_3 (7 0)))
                         ((G (7 0)) (SHIFT_8 L_0 (7 0)))
                         ((L_SQS (7 0)) (SHIFT_8 L_1 (7 0)))
                         (VDD (SHIFT_8 R_0 0))
                         (VDD (SHIFT_8 R_0 1))
                         (GND (SHIFT_8 R_0 2))
                         (GND (SHIFT_8 R_0 3))
                         (GND (SHIFT_8 R_0 4))
                         (GND (SHIFT_8 R_0 5))
                         (GND (SHIFT_8 R_0 6))
                         (GND (SHIFT_8 R_0 7))
                         (VDD (SHIFT_8 R_1 0))
                         (GND (SHIFT_8 R_1 (7 1)))
                         (GND (ADDER_8_A L_0 (7 0)))
                         ((G (7 0)) (ADDER_8_A L_1 (7 0)))
                         ((WRG_11 (7 0)) (ADDER_8_A L_2 (7 0)))
                         ((A_AAB (7 0)) (ADDER_8_A R_0 (7 0)))
                         ((WRG_11 (7 0)) (ADDER_8_A R_1 (7 0)))
                         ((WRG_10 (7 0)) (ADDER_8_A R_2 (7 0)))
                         (GND (ADDER_8_B L_0 (7 0)))
                         ((B_BAB (7 0)) (ADDER_8_B R_0 (7 0)))))

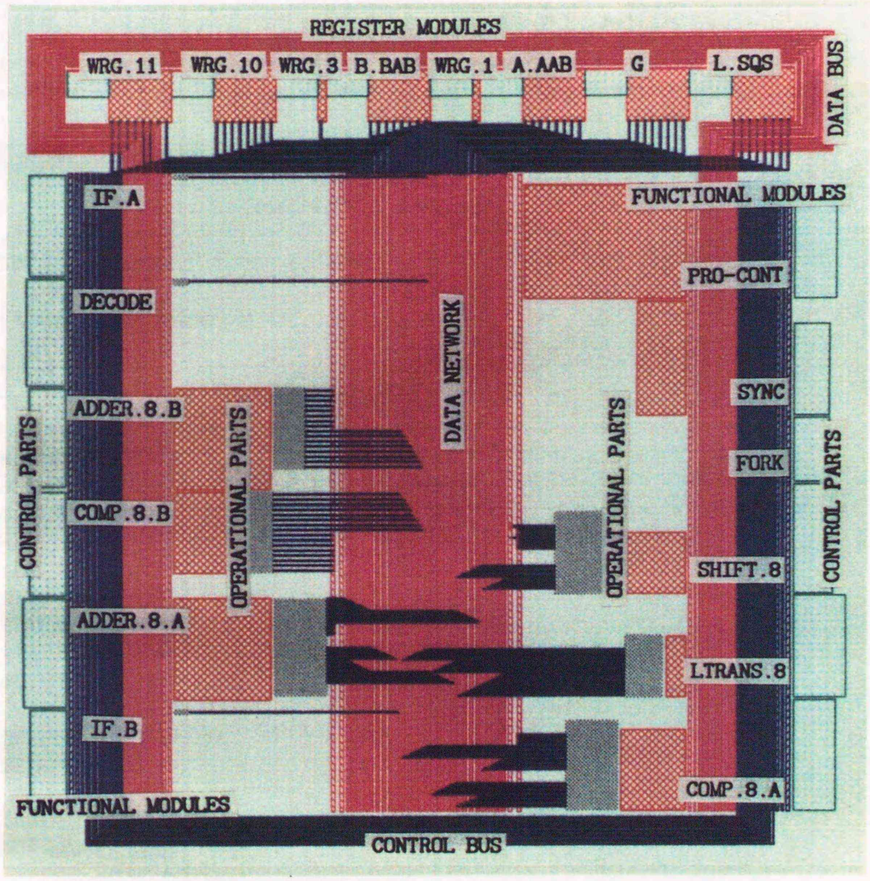
(POSITION)

(WIRING)))
```


（ｅ）モジュール動作



（ｆ）チップの構成とフロアプラン



（ｇ）動作シミュレーションのためのＩＳＰＳ記述

```
!   Distributed Control Architecture Simulator
!                  name of main module:          |NAME|  (GENMAG)
!                  width of data bus :           |WDT|   (8)
!                  # of register module:         |#RM|   (9)
!                  # of functional module:       |#FM|   (14)
!                  # of behavioral entity:       |#BE|   (2)
!
! delay time calculation
!       ex) total 8 clocks with REG write       B - - - - R - E     delay (4)
!           total 4 clocks without REG writ     B - - E             delay (2)

GENMAG := begin                 ! (**)

** simulation.resources **

TIMER <0:31>,
DWAIT <0:31>,               ! # of d-bus wait
CWAIT <0:31>,               ! # of c-bus wait
M.START <0:47>,             ! TIMER <0:31> @ MN <0:7> @ PI <0:7>
M.STOP <0:39>,              ! TIMER <0:31> @ MN <0:7>
R.WRITE <0:79>,             ! TIMER <0:31> @ MN <0:7> @ RN <0:7> @ ~000000@DT<1:8>

DBUS.BF <>,
CBUS.BF <>,
BUSY.FLAG <1:14>,           ! (**)
FIRE.FLAG <1:14>,           ! (**)
FC.TABLE [1:14] <0:7>,      ! (**)

BE.MNFC [1:2] <0:15>,       ! (**)
BE.BUSY [1:2] <>,           ! (**)

COUNT.TIME := begin
    repeat begin
        delay (1) next
        TIMER = TIMER + 1
    end
end,

** external.register **     ! (**)

external RES <0:7> {RN:1},  ! result

** register.module **       ! (**)

    A.AAB <0:7>,            ! RN: 2
    B.BAB <0:7>,            ! RN: 3
    G <0:7>,                ! RN: 4
    L.SQS <0:7>,            ! RN: 5
```

```
    WRG.1 <>,                   ! RN: 6
    WRG.3 <>,                   ! RN: 7
    WRG.10 <0:7>,               ! RN: 8
    WRG.11 <0:7>,               ! RN: 9

** data.bus.control **

critical DBBF.CHECK (ref BF<>) := begin
    decode DBUS.BF => begin
        '0¥busfree := (BF = '0 next DBUS.BF = '1),
        '1¥busbusy := (BF = '1 next DWAIT = DWAIT + 1)
    end
end,

critical REG.WRITE (MN<0:7>, RN<0:7>, DT<1:8>) := begin
    decode RN => begin          ! (**)
        1  := RES  = DT<1:8>,
        2  := A.AAB  = DT<1:8>,
        3  := B.BAB  = DT<1:8>,
        4  := G  = DT<1:8>,
        5  := L.SQS  = DT<1:8>,
        6  := WRG.1  = DT<8:8>,
        7  := WRG.3  = DT<8:8>,
        8  := WRG.10  = DT<1:8>,
        9  := WRG.11 = DT<1:8>
    end next
    R.WRITE = TIMER @ MN @ RN @ "000000 @ DT next          ! (**) caution d-bus
    delay (1) next
    DBUS.BF = '0
end,

** control.bus.control **

critical CBBF.CHECK (CMN<0:7>, NMN<0:7>, ref BF<>) := begin
    decode CBUS.BF => begin
        '0¥busfree := begin
            if CMN eql NMN => begin
                BF = '0 next
                CBUS.BF = '1 next
                leave CBBF.CHECK
            end next
            decode BUSY.FLAG <NMN> => begin
                '0¥modulefree := begin
                    BF = '0 next
                    CBUS.BF = '1 next
                    BUSY.FLAG <NMN> = '1
                end,
                '1¥modulebusy := (BF = '1 next CWAIT = CWAIT + 1)
            end
        end,
```



```
        '1¥busbusy := (BF = '1 next CWAIT = CWAIT + 1)
    end
end,

critical FM.FIRE (MN<0:7>, FC<0:7>) := begin
    FC.TABLE [MN] = FC next
    FIRE.FLAG <MN> = '1 next
    M.START = TIMER @ MN @ FC next
    delay (1) next
    CBUS.BF = '0
end,

critical FM.TERM (MN<0:7>) := begin
    BUSY.FLAG <MN> = '0 next
    M.STOP = (TIMER - 1) @ MN
end,

** external.module **          ! (**)

** control.module **

STOP.X := begin                 ! MN: 1 (always)
    wait (FIRE.FLAG<1>) next
    FIRE.FLAG<1> = '0 next
    delay (1) next
    FM.TERM (1) next
    stop ()
end,

ACTIVATE := begin               ! MN: 2 (always)
** data **
    CT.2 [1:1] <0:55>,          ! CT word size: 1
    CR <0:55>,
    BF <>,
** procedure **
    main EXEC := begin
        wait (FIRE.FLAG<2>) next
        FIRE.FLAG<2> = '0 next
        delay (1) next
        CR = CT.2[FC.TABLE[2]] next
        BE.BUSY[CR<0:7>] = '1 next
        CLOOP := begin
            CBBF.CHECK (2, CR<16:23>, BF) next
            if BF => (delay (1) next restart CLOOP) next
            FM.FIRE (CR<16:23>, CR<24:31>)
        end next
        decode CR<8:15> => begin
            "01¥procedure := BE.MNFC[CR<0:7>] = CR<32:47>,
            "02¥process := begin
                delay (1) next
```

```
                CLOOP2 := begin
                    CBBF.CHECK (2, CR<32:39>, BF) next
                    if BF => (delay (1) next restart CLOOP2) next
                    FM.FIRE (CR<32:39>, CR<40:47>)
                end
            end
        end next
        FM.TERM (2) next
        restart EXEC
    end
end,

LEAVE.X := begin                ! MN: 3 (always)
** data **
    CT.3  [1:1]  <0:55>,        ! CT word size: 1
    CR <0:55>,
    BF <>,
** procedure **
    main EXEC := begin
        wait (FIRE.FLAG<3>) next
        FIRE.FLAG<3> = '0 next
        delay (1) next
        CR = CT.3[FC.TABLE[3]] next
        BE.BUSY[CR<0:7>] = '0 next
        decode CR<8:15> => begin
            "01¥procedure := begin
                CR<16:31> = BE.MNFC[CR<0:7>] next
                CLOOP1 := begin
                    CBBF.CHECK (3, CR<16:23>, BF) next
                    if BF => (delay (1) next restart CLOOP1) next
                    FM.FIRE (CR<16:23>, CR<24:31>)
                end
            end,
            "02¥process := NO.OP (),
            "03¥labelled := begin
                CLOOP3 := begin
                    CBBF.CHECK (3, CR<16:23>, BF) next
                    if BF => (delay (1) next restart CLOOP3) next
                    FM.FIRE (CR<16:23>, CR<24:31>)
                end
            end
        end next
        FM.TERM (3) next
        restart EXEC
    end
end,

FORK := begin                   ! MN: 4
** data **
    CT.4 [1:1] <0:7>,           ! CT word size: 1
```



```
    CR <0:7>,
    FT.4 [1:3] <0:15>,          ! FT word size: 3
    FR <0:15>,
    BF <>,
** procedure **
    main EXEC := begin
        wait (FIRE.FLAG<4>) next
        FIRE.FLAG<4> = '0 next
        CR = CT.4[FC.TABLE[4]] next
        LOOP := begin
            FR = FT.4[CR] NEXT
            IF FR eql "FFFF => leave LOOP next
            delay (1) next
            CLOOP := begin
                CBBF.CHECK (4, FR<0:7>, BF) next
                if BF => (delay (1) next restart CLOOP) next
                FM.FIRE (FR<0:7>, FR<8:15>)
            end next
            CR = CR + 1 next
            restart LOOP
        end next
        FM.TERM (4) next
        restart EXEC
    end
end,

SYNC := begin                   ! MN: 7
** data **
    CT.7 [1:1] <0:23>,          ! CT word size: 1
    CR <0:23>,
    ST.7 [1:1] <0:7>,
    SR <0:7>,
    BF <>,
** procedure **
    main EXEC := begin
        wait (FIRE.FLAG<7>) next
        FIRE.FLAG<7> = '0 next
        delay (1) next
        CR = CT.7[FC.TABLE[7]] next
        SR = ST.7[FC.TABLE[7]] + 1 next
        decode SR eql CR<0:7> => begin
            '0¥uncomplete := ST.7[FC.TABLE[7]] = SR,
            '1¥complete := begin
                ST.7[FC.TABLE[7]] = 0 next
                delay (1) next
                CLOOP := begin
                    CBBF.CHECK (7, CR<8:15>, BF) next
                    if BF => (delay (1) next restart CLOOP) next
                    FM.FIRE (CR<8:15>, CR<16:23>)
                end
```

```
            end
        end next
        FM.TERM (7) next
        restart EXEC
    end
end,

IF.X.A := begin                 ! MN: 5
** data **
    CT.5 [1:1]  <0:39>,         ! CT word size: 1
    CR <0:39>,                  ! (**)
    BF <>,
** selector **
    SEL (S<0:7>) <> := begin
        decode S => begin       ! (**)
            0 := SEL = WRG.1
        end
    end,
** procedure **
    main EXEC := begin
        wait (FIRE.FLAG<5>) next
        FIRE.FLAG<5> = '0 next
        delay (1) next
        CR = CT.5[FC.TABLE[5]] next
        SEL (CR<0:7>) next
        if SEL => CR = CR sr0 16 next
        CLOOP := begin
            CBBF.CHECK (5, CR<24:31>, BF) next
            if BF => (delay (1) next restart CLOOP) next
            FM.FIRE (CR<24:31>, CR<32:39>)
        end next
        FM.TERM (5) next
        restart EXEC
    end
end,

IF.X.B := begin                 ! MN: 6
** data **
    CT.6 [1:1]  <0:39>,         ! CT word size: 1
    CR <0:39>,                  ! (**)
    BF <>,
** selector **
    SEL (S<0:7>) <> := begin
        decode S => begin       ! (**)
            0 := SEL = WRG.3
        end
    end,
** procedure **
    main EXEC := begin
        wait (FIRE.FLAG<6>) next
```



```
        FIRE.FLAG<6> = '0 next
        delay (1) next
        CR = CT.6[FC.TABLE[6]] next
        SEL (CR<0:7>) next
        if SEL => CR = CR sr0 16 next
        CLOOP := begin
            CBBF.CHECK (6, CR<24:31>, BF) next
            if BF => (delay (1) next restart CLOOP) next
            FM.FIRE (CR<24:31>, CR<32:39>)
        end next
        FM.TERM (6) next
        restart EXEC
    end
end,

DECODE.X := begin               ! MN: 8
** data **
    CT.8 [1:2] <0:15>,          ! CT word size: 2
    CR <0:15>,
    DT.8 [1:4] <0:15>,          ! DT word size: 4
    DR <0:15>,
    BF <>,
** selector **
    SEL (S<0:7>) <1:1> := begin         ! data bit size: 1
        decode S => begin               ! (**)
            0 := SEL = WRG.1
        end
    end,
** procedure **
    main EXEC := begin
        wait (FIRE.FLAG<8>) next
        FIRE.FLAG<8> = '0 next
        delay (1) next
        CR = CT.8[FC.TABLE[8]] next
        SEL (CR<0:7>) next
        DR = DT.8[CR<8:15>+'0000000@SEL] NEXT    ! (**)
        CLOOP := begin
            CBBF.CHECK (8, DR<0:7>, BF) next
            if BF => (delay (1) next restart CLOOP) next
            FM.FIRE (DR<0:7>, DR<8:15>)
        end next
        FM.TERM (8) next
        restart EXEC
    end
end,

** functional.module **

LTRANS.8 := begin               ! MN: 11
** data **                      ! data bit size: 8
```

```
    CT.11 [1:6] <0:31>,       ! CT word size:  6
    CR <0:31>,                ! (**)
    BF <>,
** selector **
    SEL (S<0:7>) <1:8> := begin        ! (**)
        decode S => begin              ! (**)
            0 := SEL = B.BAB,
            1 := SEL = A.AAB,
            2 := SEL = L.SQS,
            3 := SEL = G
        end
    end,
** procedure **
    main EXEC := begin
        wait (FIRE.FLAG<11>) next
        FIRE.FLAG<11> = '0 next
        delay (1) next
        CR = CT.11[FC.TABLE[11]] next
        SEL (CR<0:7>) next
        DLOOP := begin
            DBBF.CHECK (BF) next
            if BF => (delay (1) next restart DLOOP) next
            REG.WRITE (11, CR<8:15>, SEL)
        end next
        delay (1) next
            CLOOP := begin
            CBBF.CHECK (11, CR<16:23>, BF) next
            if BF => (delay (1) next restart CLOOP) next
            FM.FIRE (CR<16:23>, CR<24:31>)
        end next
        FM.TERM (11) next
        restart EXEC
    end
end,

COMP.8.B := begin             ! MN: 9
** data **                    ! data bit size: 8
    CT.9 [1:1]  <0:47>,       ! CT word size:  1
    CR <0:47>,                ! (**)
    RS <>,
    BF <>,
** selector **
    SEL.L (S<0:7>) <1:8> := begin      ! (**)
        decode S => begin              ! (**)
            0 := SEL.L = B.BAB
        end
    end,
    SEL.R (S<0:7>) <1:8> := begin      ! (**)
        decode S => begin              ! (**)
            0 := SEL.R = WRG.3
```



```
        end
    end,
** procedure **
    main EXEC := begin
        wait (FIRE.FLAG<9>) next
        FIRE.FLAG<9> = '0 next
        delay (1) next
        CR = CT.9[FC.TABLE[9]] next
        SEL.L (CR<8:15>) next
        SEL.R (CR<16:23>) next
        decode CR<0:7> => begin
            #0¥eql := RS = SEL.L eql SEL.R,
            #1¥neq := RS = SEL.L neq SEL.R,
            #2¥lss := RS = SEL.L lss SEL.R,
            #3¥leq := RS = SEL.L leq SEL.R,
            #4¥gtr := RS = SEL.L gtr SEL.R,
            #5¥geq := RS = SEL.L geq SEL.R
        end next
        DLOOP := begin
            DBBF.CHECK (BF) next
            if BF => (delay (1) next restart DLOOP) next
            REG.WRITE (9, CR<24:31>, RS)
        end next
        delay (1) next
        CLOOP := begin
            CBBF.CHECK (9, CR<32:39>, BF) next
            if BF => (delay (1) next restart CLOOP) next
            FM.FIRE (CR<32:39>, CR<40:47>)
        end next
        FM.TERM (9) next
        restart EXEC
    end
end,

COMP.8.A := begin             ! MN: 10
** data **                    ! data bit size: 8
    CT.10 [1:3]  <0:47>,      ! CT word size:  3
    CR <0:47>,                ! (**)
    RS <>,
    BF <>,
** selector **
    SEL.L (S<0:7>) <1:8> := begin      ! (**)
        decode S => begin              ! (**)
            0 := SEL.L = A.AAB,
            1 := SEL.L = G
        end
    end,
    SEL.R (S<0:7>) <1:8> := begin      ! (**)
        decode S => begin              ! (**)
            0 := SEL.R = 0,
```

```
            1 := SEL.R = B.BAB,
            2 := SEL.R = L.SQS,
        end
    end,
** procedure **
    main EXEC := begin
        wait (FIRE.FLAG<10>) next
        FIRE.FLAG<10> = '0 next
        delay (1) next
        CR = CT.10[FC.TABLE[10]] next
        SEL.L (CR<8:15>) next
        SEL.R (CR<16:23>) next
        decode CR<0:7> => begin
            #0¥eql := RS = SEL.L eql SEL.R,
            #1¥neq := RS = SEL.L neq SEL.R,
            #2¥lss := RS = SEL.L lss SEL.R,
            #3¥leq := RS = SEL.L leq SEL.R,
            #4¥gtr := RS = SEL.L gtr SEL.R,
            #5¥geq := RS = SEL.L geq SEL.R
        end next
        DLOOP := begin
            DBBF.CHECK (BF) next
            if BF => (delay (1) next restart DLOOP) next
            REG.WRITE (10, CR<24:31>, RS)
        end next
        delay (1) next
        CLOOP := begin
            CBBF.CHECK (10, CR<32:39>, BF) next
            if BF => (delay (1) next restart CLOOP) next
            FM.FIRE (CR<32:39>, CR<40:47>)
        end next
        FM.TERM (10) next
        restart EXEC
    end
end,

SHIFT.8  := begin              ! MN: 12
** data **                     ! data bit size: 8
    CT.12 [1:2] <0:47>,        ! CT word size:  2
    CR <0:47>,                 ! (**)
    RS <1:8>,                  ! (**)
    BF <>,
** selector **
    SEL.L (S<0:7>) <1:8> := begin      ! (**)
        decode S => begin              ! (**)
            0 := SEL.L = G,
            1 := SEL.L = L.SQS
        end
    end,
    SEL.R (S<0:7>) <1:8> := begin      ! (**)
```



```
        decode S => begin              ! (**)
            0 := SEL.R = 3,
            1 := SEL.R = 1
        end
    end,
** procedure **
    main EXEC := begin
        wait (FIRE.FLAG<12>) next
        FIRE.FLAG<12> = '0 next
        delay (1) next
        CR = CT.12[FC.TABLE[12]] next
        SEL.L (CR<8:15>) next
        SEL.R (CR<16:23>) next
        decode CR<0:7> => begin
            "0¥sl0 := RS = SEL.L sl0 SEL.R,
            "1¥sl1 := RS = SEL.L sl1 SEL.R,
            "2¥slr := RS = SEL.L slr SEL.R,
            "3¥sld := RS = SEL.L sld SEL.R,
            "4¥sli := RS = SEL.L sli SEL.R,
            "5¥sr0 := RS = SEL.L sr0 SEL.R,
            "6¥sr1 := RS = SEL.L sr1 SEL.R,
            "7¥srr := RS = SEL.L srr SEL.R,
            "8¥srd := RS = SEL.L srd SEL.R,
            "9¥sri := RS = SEL.L sri SEL.R
        end next
        DLOOP := begin
            DBBF.CHECK (BF) next
            if BF => (delay (1) next restart DLOOP) next
            REG.WRITE (12, CR<24:31>, RS)
        end next
        delay (1) next
        CLOOP := begin
            CBBF.CHECK (12, CR<32:39>, BF) next
            if BF => (delay (1) next restart CLOOP) next
            FM.FIRE (CR<32:39>, CR<40:47>)
        end next
        FM.TERM (12) next
        restart EXEC
    end
end,

ADDER.8.A := begin             ! MN: 13
** data **                     ! data bit size: 8
    CT.13 [1:3] <0:47>,        ! CT word size:  3
    CR <0:47>,                 ! (**)
    RS <1:8>,                  ! (**)
    BF <>,
** selector **
    SEL.L (S<0:7>) <1:8> := begin      ! (**)
        decode S => begin              ! (**)
```

```
            0 := SEL.L = 0,
            1 := SEL.L = G,
            2 := SEl.L = WRG.11
        end
    end,
    SEL.R (S<0:7>) <1:8> := begin       ! (**)
        decode S => begin               ! (**)
            0 := SEL.R = A.AAB,
            1 := SEL.R = WRG.11,
            2 := SEL.R = WRG.10
        end
    end,
** procedure **
    main EXEC := begin
        wait (FIRE.FLAG<13>) next
        FIRE.FLAG<13> = '0 next
        delay (1) next
        CR = CT.13[FC.TABLE[13]] next
        SEL.L (CR<8:15>) next
        SEL.R (CR<16:23>) next
        decode CR<0:7> => begin
            "0¥add := RS = SEL.L + SEL.R,
            "1¥sub := RS = SEL.L - SEL.R
        end next
        DLOOP := begin
            DBBF.CHECK (BF) next
            if BF => (delay (1) next restart DLOOP) next
            REG.WRITE (13, CR<24:31>, RS)
        end next
        delay (1) next
        CLOOP := begin
            CBBF.CHECK (13, CR<32:39>, BF) next
            if BF => (delay (1) next restart CLOOP) next
            FM.FIRE (CR<32:39>, CR<40:47>)
        end next
        FM.TERM (13) next
        restart EXEC
    end
end,

ADDER.8.B := begin              ! MN: 14
** data **                      ! data bit size: 8
    CT.14 [1:1] <0:47>,         ! CT word size:  1
    CR <0:47>,                  ! (**)
    RS <1:8>,                   ! (**)
    BF <>,
** selector **
    SEL.L (S<0:7>) <1:8> := begin       ! (**)
        decode S => begin               ! (**)
            0 := SEL.L = 0
```



```
            end
        end,
        SEL.R (S<0:7>) <1:8> := begin       ! (**)
            decode S => begin               ! (**)
                0 := SEL.R = B.BAB
            end
        end,

    ** procedure **
        main EXEC := begin
            wait (FIRE.FLAG<14>) next
            FIRE.FLAG<14> = '0 next
            delay (1) next
            CR = CT.14[FC.TABLE[14]] next
            SEL.L (CR<8:15>) next
            SEL.R (CR<16:23>) next
            decode CR<0:7> => begin
                "0¥add := RS = SEL.L + SEL.R,
                "1¥sub := RS = SEL.L - SEL.R
            end next
            DLOOP := begin
                DBBF.CHECK (BF) next
                if BF => (delay (1) next restart DLOOP) next
                REG.WRITE (14, CR<24:31>, RS)
            end next
            delay (1) next
            CLOOP := begin
                CBBF.CHECK (14, CR<32:39>, BF) next
                if BF => (delay (1) next restart CLOOP) next
                FM.FIRE (CR<32:39>, CR<40:47>)
            end next
            FM.TERM (14) next
            restart EXEC
        end
    end,

    ** main.procedure **

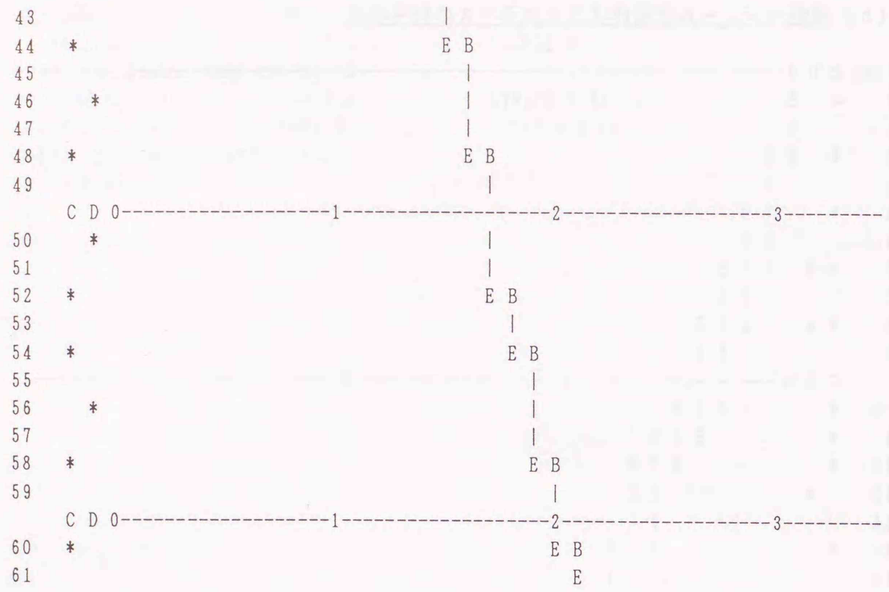
    INIT := begin
        TIMER = 0;          DWAIT = 0;          CWAIT = 0;
        DBUS.BF = '0;       CBUS.BF = '0;
        BUSY.FLAG = 0;      FIRE.FLAG = 0
    end,

    main GENMAG.EXEC := begin
    ! simulation modules
        INIT ();            COUNT.TIME ();      STOP.X ();
    ! (**) external modules
    ! (**) control modules
        ACTIVATE ();        LEAVE.X ();
```

```
    FORK ();            SYNC ();
    IF.X.A ();          IF.X.B ();          DECODE.X ();
! (**) functional modules
    COMP.8.B ();        COMP.8.A ();        LTRANS.8 ();
    SHIFT.8 ();         ADDER.8.A ();       ADDER.8.B ();
! (**) first module activation
    FM.FIRE (2,1)
end
end
```


（h）機能モジュールの動作状況と共通バスの利用状況

```
TIME C D 0-------------------1-------------------2-------------------3----------
0    *   B
1        |
2    *   E B
3          |
4    *     | B
5          | |
6    * *   E | B
7            | |
8    * *     E | B
9              | |
     C D 0-------------------1-------------------2-------------------3----------
10   *         E | B
11   *           E | B
12   *             E | B
13     *           | E
14                 |
15   *             E   B
16                     |
17                     |
18   *                 E B
19                       |
     C D 0-------------------1-------------------2-------------------3----------
20     *               |
21                     |
22   *                 E B
23                       |
24   *                   E B
25                         |
26     *                   |
27                         |
28   *                     E B
29                           |
     C D 0-------------------1-------------------2-------------------3----------
30     *                     |
31                           |
32   *                       E B
33                             |
34     *                       |
35                             |
36   *                         E B
37                               |
38     *                         |
39                               |
     C D 0-------------------1-------------------2-------------------3---------
40   *                             E B
41                                   |
42     *                             |
```



COLUMN	MN		FC	START	STOP
1	"02"	ACTIVATE	"01"	0	2
2	"04"	FORK	"01"	2	6
3	"0A"	COMP_8_A	"01"	4	8
4	"09"	COMP_8_B	"01"	6	10
5	"05"	IF_A	"01"	8	11
6	"06"	IF_B	"01"	10	12
7	"0D"	ADDER_8_A	"01"	11	15
8	"07"	SYNC	"01"	12	13
9	"07"	SYNC	"01"	15	18
10	"0A"	COMP_8_A	"02"	18	22
11	"08"	DECODE	"01"	22	24
12	"0B"	LTRANS_8	"01"	24	28
13	"0B"	LTRANS_8	"02"	28	32
14	"0C"	SHIFT_8	"01"	32	36
15	"0D"	ADDER_8_A	"02"	36	40
16	"0C"	SHIFT_8	"02"	40	44
17	"0D"	ADDER_8_A	"03"	44	48
18	"0A"	COMP_8_A	"03"	48	52
19	"08"	DECODE	"02"	52	54
20	"0B"	LTRANS_8	"05"	54	58
21	"03"	LEAVE	"01"	58	60
22	"01"	EXTERNAL	"01"	60	61

TIME	MN		RN		VALUE
6	"0A"	COMP_8_A	"06"	WRG_1	"00000001"
8	"09"	COMP_8_B	"07"	WRG_3	"00000000"
13	"0D"	ADDER_8_A	"02"	A_AAB	"00000019"
20	"0A"	COMP_8_A	"06"	WRG_1	"00000000"
26	"0B"	LTRANS_8	"04"	G	"00000022"
30	"0B"	LTRANS_8	"05"	L_SQS	"00000019"
34	"0C"	SHIFT_8	"09"	WRG_11	"00000004"
38	"0D"	ADDER_8_A	"09"	WRG_11	"0000001E"
42	"0C"	SHIFT_8	"08"	WRG_10	"0000000C"
46	"0D"	ADDER_8_A	"05"	L_SQS	"0000002A"
50	"0A"	COMP_8_A	"06"	WRG_1	"00000000"
56	"0B"	LTRANS_8	"01"	EXTERNAL	"0000002A"

第Ⅳ部　結論

1．研究成果のまとめ

本論文では，将来のＶＬＳＩシステム“Systems on Silicon”を構築する技術を確立するために実施した２つアプローチからの研究について述べた．即ち，ＶＬＳＩチップの設計者の立場で高性能なＶＬＳＩシステムの実現を目指す“プログラミング言語解析チップに関する研究”と，ＶＬＳＩ設計支援ツールの開発者の立場でＶＬＳＩシステムを短期間で容易に開発することを目指す“動作記述型シリコン・コンパイラに関する研究”について述べた．

（１）プログラミング言語解析チップに関する研究

高性能な専用ＶＬＳＩチップを設計しようとするチップの設計者の立場で，プログラミング言語 Pascal のコンパイラにおける言語解析処理（字句解析と構文解析）を専用に実行するＶＬＳＩチップの実現方式と，これを効果的に支援する設計支援システムについての研究を行った．この結果，これらの言語解析用ＶＬＳＩチップは，ＰＬＡやスタック等を主体とするメモリ・アーキテクチャによって簡潔に実現でき，実行速度の面でも汎用計算機のソフトウェアによる言語解析処理に比べて２桁以上高速であることが示された．また，レイアウト記述用高級言語：ＤＰＬを用いたモジュール・コンパイラと，会話型のレイアウトシステム：ＩＤＥＯＮを利用することにより，フルカスタムＶＬＳＩチップとしては非常に少ない設計労力でレイアウト設計できることが示された．従って，モジュール・コンパイラ等の高度な設計支援ツールが提供されているＶＬＳＩアーキテクチャが適用できれば，大規模で高性能なＶＬＳＩチップでも非常に容易に設計できることを実例を通して示すことができた．

（２）動作記述型シリコン・コンパイラに関する研究

効果的なＶＬＳＩ設計支援ツールを開発しようとするツールの開発者の立場で，シリコン・コンパイラを開発するために有効なＶＬＳＩアーキテクチャと，このアーキテクチャを基本とする動作記述型のシリコン・コンパイラについての研究を行った．この結果，内部に制御回路を含んで独立に動作する要素で構成され，これらが局所的な通信を行いながら所定の動作を実現する分散制御アーキテクチャを設定すると，並列動作記述言語における各種の演算・制御構文をこのアーキテクチャの構成要素の動作に容易に変換できることが示された．また，これらの構成要素は比較的簡潔な構造をしているため，モジュール・コンパイラの技術によって自動生成でき，これらを配置・配線することによって，分散制御アーキテクチャに基づくＶＬＳＩチップのレイアウト情報をソフトウェア的に自動生成できることが示された．従って，特定のＶＬＳＩアーキテクチャを設定することによって，手続き的な動作記述を入力とするシリコン・コンパイラを容易に実現できることを示すことができた．

以上の研究成果として，個々には解決すべき幾つかの課題が残されているが，ＶＬＳＩアーキテクチャ技術とＶＬＳＩ設計支援技術を相互に補完・協調することによって，ＶＬＳＩシステム“Systems on Silicon”を構築できることを具体的な実例を通して明らかにすることができた．同時に，将来のＶＬＳＩシステムの構築に適用できる幾つかの基本技術を確立することができた．

2．今後の課題

本論文において，ＶＬＳＩアーキテクチャ技術とＶＬＳＩ設計支援技術を相互に補完・協調することによって，将来のＶＬＳＩシステムを構築できることを明らかにできたが，今後のさらなるＶＬＳＩデバイス技術の発展や各種の高性能ＡＳＩＣへの要求に対処するために，以下に示す研究課題に対して，ＶＬＳＩアーキテクチャ技術とＶＬＳＩ設計支援技術の両面からの研究開発を意欲的かつ継続的に推進する必要性が益々高まっている．

（１）設計の高位化

第Ｉ部でも示したように，将来の大規模ＶＬＳＩチップの開発を進めるためには，これまでの論理ゲート・レベルの設計から，機能，動作，仕様レベル等への“設計の高位化”は必然の方向であり，論理ゲート・レベル等で適用されてきた従来の設計手法や実現方式（アーキテクチャ）を下記の観点において根本から見直す必要がある．

① 高位記述形式

高位設計といっても様々な設計レベルが考えられ，どのような切口で，どのような表現形式（言語，図的表現等）を設定するかは非常に重要な課題である．即ち，特定の表現形式は，様々なアーキテクチャに対する記述のしやすさ，ＶＬＳＩ設計過程や設計支援ツールとの適合性，最終的に生成されるチップの性能等に大きく依存する．例えば，従来のゲート・レベルに近い表現形式であれば，これまでの設計支援ツールとの適合性は良く，生成されるチップの性能も期待できるが，将来の大規模化には対応できなくなると考えられる．一方，プログラミング言語等の内部構造を規定しない表現形式では，そのアルゴリズム記述から物理的な構造を生成するまでの複雑な設計過程をソフトウェア的に自動化することが大変困難である．このような高位設計の切口の有望な具体例として，ハードウェア上の大まかな構成要素を意識しつつ，これらを用いた動作を手続き的に記述できる動作記述言語としてＳＦＬ（Structured Function Description Language）[1]が提案されている．

② 高位合成と最適化[2]

ＶＬＳＩチップの高位記述を受け入れ，これから低位の設計情報を自動生成する設計自動化ソフトウェアは，設計高位化に関わる多くの効果が期待できるものの，最終的には合成結果の質が問われることになる．ブール式等で与えられた論理記述から特定の論理ゲートの組み合わせを自動生成する多段論理の合成技術は，最近，人手設計によるものよりも優れた回路が生成できるようになり，実用的に利用される状況に至っている[3]．しかし，これよりも高位の設計記述を入力とする動作記述型のシリコン・コンパイラ等は，熟練した設計者に匹敵する最適化機

能を備えていないのが現状である．論理レベルよりも高位の設計レベルにおいても人手設計と同等以上の設計結果を生成するためには，特定のＶＬＳＩアーキテクチャを限定することによって最大限の最適化を図ることが，ひとつの現実的なアプローチと考えられる．

③ 検証方式

設計の高位化にともなって，設計の自動化が進展すると考えられるが，自動生成された設計結果が正しく動作することを保証する検証方式や，生成されたチップに自己テスト機能や自己修復機能を埋め込むなどのテスト容易化設計に対する十分な考慮が必要である．但し，シリコン・コンパイラによって生成されたレイアウト情報に対して設計規則検査プログラムを通したり，レイアウト情報から論理回路への逆変換を行って検証するのでは，設計記述を高位にした効果が半減してしまうため，設計の検証も記述と同レベルで行える高位検証方式を開発していかなければならない．

④ 性能予測

設計レベルが高位になるほど最終的なチップの性能や構造が把握しにくくなるが，初期の設計段階においても，実現されるＶＬＳＩチップの動作性能やハードウェア規模を的確に評価しておく必要がある．熟練した設計者はこのような性能予測をこれまでの経験から直感的に行っていると思われるが，計算機によってこのような大局的な判断を行わせるのは現時点では大変困難である．従って，このためのシミュレーション環境やアーキテクチャ評価ツール等の整備とともに，多くの設計例から類似したパターンを高速に検索して評価する推論メカニズムや設計知識ベースの研究開発が必要である．

⑤ 中間形式

高位設計を行った時には，目標性能，許される開発期間，開発労力に応じて，実際の実現形態を自由に選択できる効果がある．このためには，下位の様々なチップ設計ツールに自由に接続できる中間的な標準インタフェース形式を整備する必要がある．現在，米国を中心に設定されているＥＤＩＦ（Electronic Design Interchange Format）[4]による論理回路の接続記述は，このような目的にも適用できると思われるが，チップ設計に影響を及ぼすが中間的な設計結果には反映されない高位記述（例えば，物理的な制約条件や要求仕様等）を下位の設計ツールに伝える方式の開発が必要である．

（２）設計の蓄積

将来の大規模ＶＬＳＩチップを迅速に開発するためには，それまでに開発された設計の遺産や熟練した設計者の設計知識等を有効に活用していくことが重要である．従来のＶＬＳＩ設計支援ツールにおいては，当面のチップ開発の効率化に

主眼が置かれてきたが，今後は将来的な長期展望にたった設計の蓄積のための基本メカニズムの構築と設計様式の変革が必要である．

① ＶＬＳＩアーキテクチャ

様々な応用システムの実現に適した多数の専用ＶＬＳＩアーキテクチャを開発するとともに，これらを具体的なＶＬＳＩチップに適用するときのノウハウや効果等を，アーキテクチャ設計知識として，だれでもが参照できる形で蓄積していく必要がある．特に，ＶＬＳＩチップの構成要素の機能，種類，論理規模，及び，これらの構成要素の並列動作性等が，チップ開発の容易性や性能向上の観点から，アーキテクチャ設計上の非常に重要なポイントであり，このような設計知識の収集と蓄積が必要である．

② レイアウト・モジュール

レイアウト設計に関する過去の設計遺産を活用するために，多数のＶＬＳＩチップで共通に利用される構成要素をモジュール・コンパイラとしてライブラリ化して整備する必要がある．このモジュール・コンパイラは，特定のプロセス技術を前提とせず，かつ，利用者の様々な要求に応えられるように柔軟性と汎用性の高いものでなければならない．これまでのモジュール・コンパイラでは，モジュールの基本構造が固定されているため，他のモジュールとの組合わせや配置の自由度が制限されているものが多い．今後は，どんなパラメータの値に対しても，健全な形状のモジュールや他のモジュールとの関係で縦横比を自由に変更できるなどの柔軟な実現方式を備えたものを用意する必要がある．また，モジュール間の配線には相当のチップ面積を必要とすることから，モジュールのレイアウト・パターンの上を自由に配線できるような多層配線構造のプロセス技術を開発していくことも必要である．

③ 設計知識・ノウハウ

設計結果やライブラリ等の設計データに加えて，熟練した設計者の有している設計知識やノウハウ等を蓄積することも必要である．特に，これからＡＳＩＣの需要の急増に伴って，設計者の絶対数が大幅に不足することが予想され，熟練した設計者を早期に育成するためや，応用分野の技術者が自らチップ開発を行うためにも，多くの設計知識やノウハウ等がいつでも分かりやすい形式で参照できる知識ベースを整備していかなければならない．一般にこのような知識やノウハウは，物理の基本原理のように簡潔に記述されることは稀であり，複雑な条件下での準最適な一つの可能性を示していることが多い．従って，このような知識等を知識ベースとして蓄積するためには，自然言語を含めた各種の言語表現の他に，ブロック図・真理値表・論理図・タイミングチャート等の図的表現による知識の表現形式，及び，これら知識相互の関連付け方式等を開発していく必要がある．

（３）統合的ＶＬＳＩ開発利用環境

これまでの設計支援ツールは主として設計者が効率良くチップ設計を行うためのツールとして利用されてきたが，今後は，チップの企画から，設計，試験，製造，流通，サービスに至る多くの人々に共通に利用される統合的なＶＬＳＩ問題解決環境としての機能が要求されるようになると考えられる[5]．このようなシステムは，技術の発展，ニーズ，社会構造等の変化に応じて，これ自身が成長していくものであり，将来の様々な側面での変化にも柔軟に対応できるシステムであることが必須である．

① 設計情報ベース

ＶＬＳＩチップの設計において，各設計段階や各種ツールの利用毎に非常に多数で様々な形式の設計・検証データが作られていく．さらに，何らかの設計上の誤りが発見されると適切なレベルに戻って再設計され，設計データの“改訂”が繰り返される．これらの膨大な数と量のデータは相互に関連をもち，特定の目的毎に体系的に利用されるため，各種ツールの実行過程とともに，これらのデータを設計情報ベースとして統一的に管理する必要がある．また，上記のような設計データに加えて，その設計に適用した設計知識や，なぜ，そのような設計をしたかという理由や経緯等を示す記述も設計データとの関連をもって設計情報ベースに蓄積していく必要がある．

② システム設計

従来のＶＬＳＩ設計では単体としてのチップ開発だけに主眼が置かれてきたが，μプロセッサや各種ＡＳＩＣ等の場合には，ＶＬＳＩチップの設計に加えて，これらのチップを用いたプリント基板や装置全体のハードウェア，さらには，これに実装されるソフトウェアの開発が必須となってきている．このためには，応用システムそのものを総合的に解析し，ソフトウェア，ハードウェア，ＶＬＳＩチップの適切な分担を決め，それぞれの開発を統一的に支援する設計環境を整備していく必要がある．即ち，今後は，単にＶＬＳＩアーキテクチャというよりも，“ＶＬＳＩシステム・アーキテクチャ技術”という観点に立って，このためのシステム設計支援技術とともに発展させていく必要がある．

③ 複数デバイス技術，実装技術

大規模システム全体をＶＬＳＩチップ化するという“Systems on Silicon”の自然な要求として，複数のデバイス技術を駆使して１個のチップを設計することが必要になってくる．即ち，通常の演算・記憶機能は高集積なＣＭＯＳデバイスで実現するが，外部との入出力処理に関してアナログ演算回路やＡ／Ｄ，Ｄ／Ａ変換等のアナログ・デバイスを用いたり，非常に高速な演算機能に関してはバイポーラ素子や GaAs 素子等の高速デバイスの組込が必要になってくる．さらに今後は，ＷＳＩ（Wafer Scale Integration）や３次元ＩＣ等の新しい構造のデバ

イス／実装技術が利用されるようになっていくと考えられ，このような様々なデバイス技術や実装技術を利用したＶＬＳＩチップ／システムの開発を可能にする統合的な開発環境を整備していくことが必要である．

④ 製品の製造・販売・利用支援

従来のＣＡＤシステムはＶＬＳＩチップの“設計”という目的に沿った機能を提供してきた．しかし，ＶＬＳＩチップの設計過程の前には，様々なニーズや市場の状況を踏まえたプロジェクト企画の段階から，これまで計算機では殆どサポートされていなかった高位設計過程があり，この後には，試作品の検査，最終的な製品の製造／検査，関連するドキュメント類の整備等の出荷／販売支援，及び，利用者からの問合せ／クレームに対する応答等の製造・販売・利用過程がある．これらの各過程・業務においては，多くの情報，データが共用されることに加えて，これらに携わる人々相互のコミュニケーションが非常に重要であり，今後はこのような多くの業務で，多数の人々に利用されるような統合的な環境を提供していく必要がある．即ち，これまでの設計支援ツールは設計データをつくるためのツールであったが，これからは，様々なＶＬＳＩ関連業務を多くの人々が協力して遂行するための統合的な問題解決環境でなければならない．

⑤ プロジェクト管理

ＶＬＳＩ開発も大規模ソフトウェアの開発過程と同様に，長期にわたって，多くの設計者等が関わり，様々なツールや計算機，製造装置，検査装置等を用いる大がかりなプロジェクトの形態を採ることが多い．従って，このＶＬＳＩ開発プロジェクトを円滑に推進するためには，共通ツールや装置類の利用管理，各担当部門の進捗管理等のプロジェクト管理を行わなければならない．また，ＶＬＳＩ開発工程においては，設計部門から製造・検査部門への手配書や，検査部門から設計部門への検査報告等，部門間の様々な文書類が送受される．従って，これからの開発支援環境には，このようなＥＯＡ（Engineering Office Automation）機能や上記のプロジェクト管理機能を組み込んでいくことが必要になる．

以上，今後に残された幾つかの研究課題を示したが，新しいＶＬＳＩ設計手法を実現するＶＬＳＩ設計支援環境に関する研究開発，及び，これらを用いた具体的なＶＬＳＩシステムの研究開発を通じて，２１世紀のＵＬＳＩ（Ultra Large Scale Integration）時代を支える基本技術の確立に今後とも取り組んでいきたいと考えている．

［参考文献Ⅳ］

(1) Nakamura, Yukihiro & Kiyoshi Oguri: An RTL Logic Design Aid For Parallel Control VLSI Processor, Proc. of the VLSI '87 (1987).
(2) McFarland, Michael C., et al.: The High-Level Synthesis of Digital Systems, Proc. of the IEEE, Vol.78, No.2 (1990).
(3) Brayton, Robert K., et al.: Multi-Level Logic Synthesis, Proc. of the IEEE, Vol.78, No.2 (1990).
(4) EDIF Steering Committee: EDIF Electronic Design Interchange Format Version 2 0 0, EIA Interim Standard No.44 (1987).
(5) Harrison, David S., et al.: Electronic CAD Frameworks, Proc. of the IEEE, Vol.78, No.2 (1990).

［謝辞］

本研究は，筆者が三菱電機株式会社中央研究所において実施したものであります．本研究を進めるにあたり，また，これをまとめるにあたり，多くの方々のお力添えを得て初めて所定の成果をあげることができました．ここに，お名前を書き記して，あらためて感謝の意を表します．

本論文をまとめる機会を与えて頂くとともに，著者の不備な原稿にも辛抱強く目を通して頂き，終始心温まる御指導，御鞭撻を頂いた北海道大学工学部情報工学科　宮本衛市教授に心より御礼申し上げます．また，本論文の内容について御検討頂き，御指導を頂いた北海道大学工学部情報工学科　青木由直教授，同電子工学科　栃内香次教授，同電気工学科　田中　譲教授に深く感謝致します．

三菱電機株式会社中央研究所における直接の上司として，本論文で述べた研究を実施する機会を与えて頂くとともに，常日頃から多大な御理解と御指導を頂いた房岡　璋グループマネジャー（現在，三菱電機株式会社情報電子研究所次世代システム開発部次長）に心より感謝致します．また，本論文をまとめる機会を与えて頂くとともに，心温まる励ましを頂いた三菱電機株式会社中央研究所システム基礎研究部　武田捷一部長に深く感謝致します．

本論文の中で述べたレイアウト記述言語：ＤＰＬと会話型レイアウト・システム：ＩＤＥＯＮの開発に関して御協力頂いた三菱電機株式会社中央研究所システム基礎研究部　杉本　明主事，また，プログラミング言語解析チップに使われる各モジュール・コンパイラの開発と最終的なチップのレイアウト設計に関して御協力頂いた同部　瀬尾和男主事に感謝致します．