

ペクトルアーキテクチャの構築と
その実現方式の研究

内田 啓一郎

①

ベクトルアーキテクチャの構築と
その実現方式の研究

内田 啓一郎

目 次

第1章	序 論	1
1. 1	本研究の背景	1
1. 2	本研究の目的	4
1. 3	本論文の構成	5
1. 4	言葉の定義	6
第2章	コンピュータの高速化アーキテクチャと本研究の位置づけ	9
2. 1	コンピュータの高速化の歴史	9
2. 2	ベクトルアーキテクチャと高速化	20
2. 3	高速化技術の中での本研究の位置づけ	27
第3章	ベクトルプロセッサの分析と問題点	32
3. 1	ベクトルプロセッサのシステム構成	32
3. 2	ベクトル化の可能性の分析と問題点	33
3. 3	命令の並列実行可能性に関する分析と問題点	37
第4章	ベクトルアーキテクチャの機能拡張	42
4. 1	条件処理のベクトル化	42
4. 2	ベクトル処理拡張機能	53
第5章	並列命令実行アーキテクチャ	59
5. 1	ベクトルレジスタの構成と機能	59
5. 2	命令の多重発信と並列実行	67
5. 3	命令書換えの禁止	76
5. 4	アドレス変換方式	78
5. 5	割り込みと命令並列実行	79

第6章	ベクトル命令の制御方式とその実装	86
6. 1	ベクトルプロセッサの構成	86
6. 2	ベクトルパイプラインの構成	87
6. 3	ベクトル命令のパイプライン制御	90
6. 4	ステー징制御とパイプライン制御	93
6. 5	ベクトル演算パイプラインの物理的多重構成	95
6. 6	ベクトルレジスタの構成	97
6. 7	ベクトル命令の発信タイミング	99
6. 8	メモリアクセスと状態制御	101
6. 9	ベクトル命令の連結実行	102
6. 10	ベクトルデータのメモリ直接アクセス	105
6. 11	スカラキャッシュコヒーレンシ	106
6. 12	チャネルバッファ	110
6. 13	メモリアクセスの同期制御	111
第7章	総合性能評価	115
7. 1	ベクトル化可能性の評価	116
7. 2	第1世代アーキテクチャとの性能評価	117
7. 3	実アプリケーションプログラムによる評価	119
7. 4	ライブラリの性能評価	123
7. 5	まとめ	124
第8章	結 言	125
8. 1	結 言	125
8. 2	今後の展望	127
	謝 辞	129
	執筆論文	131
	参考文献	133

目 次

第2章

図2. 1	逐次型加算器	1 1
図2. 2	並列型加算器	1 1
図2. 3	並列型乗算器	1 2
図2. 4	命令のパイプライン制御	1 3
図2. 5	ハードウェアイメージでのパイプライン動作	1 4
図2. 6	1 アドレス命令形式	1 5
図2. 7	2 アドレス命令形式	1 6
図2. 8	3 アドレス命令形式	1 8
図2. 9	スカラ命令の命令実行タイムチャート	2 3
図2. 10	ベクトルロードパイプラインのタイムチャート	2 5
図2. 11	ベクトルパイプラインのタイムチャート	2 6

第3章

図3. 1	想定したシステム	3 2
図3. 2	ベクトル化率とベクトルプロセッサ性能	3 4
図3. 3	プログラム分析結果	3 5

第4章

図4. 1	演算命令形式	4 3
図4. 2	マスク機能	4 4
図4. 3	ベクトル収集命令	4 6
図4. 4	ベクトル間接ロード命令形式	4 7
図4. 5	ベクトル間接ロード命令の動作説明	4 8
図4. 6	条件ベクトル処理プログラム例1	4 9
図4. 7	条件ベクトル処理プログラム例2	5 0
図4. 8	条件ベクトル処理の選択	5 2
図4. 9	連続乗算の実行	5 7

第5章

図5. 1	ベクトルレジスタの連結	6 1
図5. 2	命令のレジスタ指定部とベクトルレジスタ	6 2
図5. 3	プログラム例 (ループ1)	6 3
図5. 4	プログラム例 (ループ2)	6 3
図5. 5	ベクトルレジスタの構成と性能	6 5
図5. 6	ベクトルデータ	6 8
図5. 7	メモリデータの順序関係	7 0
図5. 8	メモリアクセスID	7 3
図5. 9	POST/WAIT命令	7 5
図5. 10	割り込み発生と実行中の命令	8 0

第6章

図6. 1	FACOM VP-200のシステム構成	8 6
図6. 2	命令制御パイプライン	9 1
図6. 3	スカラデータの送受信	9 3
図6. 4	演算パイプラインと制御段数	9 4
図6. 5	ベクトルレジスタのバンク構成	9 7
図6. 6	バンクスロット	1 0 0
図6. 7	ベクトル命令の連結実行	1 0 3
図6. 8	ベクトルアクセスとキャッシュコヒーレンシ	1 0 8
図6. 9	IDによる命令パイプラインの割当	1 1 1
図6. 10	メモリアクセス命令の逐次実行	1 1 2
図6. 11	POST/WAIT命令	1 1 4

目 次

第3章

表3. 1	メモリアクセス頻度	39
-------	-----------------	----

第5章

表5. 1	サンプルプログラムの分析	66
表5. 2	多重割り込み条件と優先順位	85
表5. 3	ベクトル命令におけるプログラム例外とその優先順位	85

第6章

表6. 1	演算の出現確率	88
-------	---------------	----

第7章

表7. 1	ループのベクトル化	116
表7. 2	F230-75APUとの比較	118
表7. 3	リバモアループの性能	120
表7. 4	実アプリケーションプログラムでの性能(1)	121
表7. 5	実アプリケーションプログラムでの性能(2)	122
表7. 6	線形1次方程式の性能	123
表7. 7	FFTの性能	123

第1章 序 論

1. 1 本研究の背景

コンピュータの性能向上は、テクノロジーの性能向上とアーキテクチャ上の高速化とによって決まる。

コンピュータの発明以来、1940年代から1970年代まではテクノロジーの高速化がもたらしたコンピュータの高速化に寄与したが、それ以降はテクノロジー高速化の向上率が鈍化したため、高速化要因はアーキテクチャによる高速化が主に貢献したと言ってもよい。

最初のコンピュータであるENIACは真空管で作られた。当時のコンピュータの素子としてはリレーも使われた。パラメトロンやトランジスタがそれに続き、最終的にSi素子トランジスタによるTTL回路、ECL回路へと発展した。更にGaAs系の素子へと発展し、今後とも高速化への可能性がある。

1970年代の後半から素子の高集積化が始まり、高速素子としてのECL回路の高集積度(LSI)化が始まった。コンピュータはこれにより、高速化が大いに進んだ。

一方、CMOSテクノロジーによって低電力消費LSI素子が出現したことは、後の高速コンピュータに大きな影響を与える。いまだにECL/GaAsに比べて高速性能は追いついていないが、プロセス技術発展にともなう微細加工技術が進歩すれば、高速性についても競合できる状況になる可能性もある。

1980年以降は、高速コンピュータの使用素子であるECLテクノロジーの高速化は着実に進歩したが、それ以前ほどの率での高速化の進歩はなかった。例えば、VP-200のシステムクロックサイクルタイムは7nsでありが、最近の最高速クロックサイクルである日立S3800シリーズでのそれは2nsであり、10年間にわずか3.5倍の高速化しか達成できなかったことになる。

従って、素子による高速化の限界を突破するために、アーキテクチャの改良が進んだ。

アーキテクチャによる高速化とは物理的空間的並列化、時間的並列化を意味しており、演算を多重、並列に実行し、単位時間あたりの演算実行数を増大させる方式である。

コンピュータの歴史の中で、高速化は空間的な並列化から始まった。

足し算では、一桁ずつに逐次実行する加算器から多数桁のデータを一度に演算する並列加算器へ発展した。掛け算では、加算と桁シフトの繰り返し演算で実行していたものを、並列演算による乗算器として高速化を達成した。

多重プロセッサを並列に動作させれば、並列同時実行による実効的な高速化が実現でき、これは多重並列プロセッサシステムとして実現された。

一方、時間的な並列化はパイプライン方式として実現された。

機能回路を小さな時間で実行可能な単位機能に分解し、小さな時間毎にその機能回路に異なるイベント（またはデータ）を次々に繰り返し投入するように制御し、数多くのイベントが処理されるため、見かけ上、前記の小さな時間で処理が実行されるのと等価に見させる方式である。

本方式は通常のスカラプロセッサにおいて、パイプライン命令制御の基本方式として定着した。

例えば、空間的な多重演算器を並列同時動作をさせる高速コンピュータではCDC 6600/7600があり、時間的な並列化パイプラインを主な構成要素としたコンピュータではIBM 360/91, 95が上げられる。これらのコンピュータはパイプライン方式、演算器の多重化により、命令の非同期並列実行を行って、スカラ計算を実行するコンピュータとして、一つの頂点に立った。

しかし、逐次的スカラ命令の実行では、投入した演算器の使用効率の面で十分とは言えず、さらにアーキテクチャとしての工夫が必要であった。

ここで登場したのが、いわゆるスーパーコンピュータであった。

広義には上記の機械もスーパーコンピュータといえることができるが、一般的にはベクトルプロセッサをスーパーコンピュータといふことが多い。

スーパーコンピュータは科学技術計算の高速化を目指し、FORTRANのDO LOOP文を高速化することを目標とした。同一演算を多数のデータに対して繰り返すSIMD (Single Instruction stream Multiple Data stream) 方式とすることができる。

SIMD方式による演算実行方法にはベクトル方式と並列方式がある。

ベクトル方式は時間的な並列処理として、同一演算を実行するパイプライン演算器に繰り返し、多数のデータを送りこむものである。この方式を本格的に実現した最初の機械としてはTI ASC, CDC STAR100がある。

一方、並列方式は空間的にプロセッサを多重に設置する並列処理方式のことを言う。本格的に実現できた最初の機械はILLIAC IVである。

スーパーコンピュータの製品化は、ベクトル方式から始まった。ベクトル方式は、スカラコンピュータと同一のプログラムの記述ができ、自動ベクトル化コンパイラによって性能を高め、スカラコンピュータの延長線上に位置づける製品として世の中に広まった。

一方、並列方式は、プログラムの並列記述が新規に必要であり、製品化には時間が必要であった。さらに、SIMD方式では、適用範囲がベクトル方式と変わらずメリットが少なかった。従って、後に応用範囲を広めるためには、各プロセッサが独立に動作するMIMD (Multiple Instruction Stream Multiple Data Stream) 方式の方が有利であり、並列方式の製品化はこの方式によって実現された。

ベクトル方式を実現したプロセッサをベクトルプロセッサと呼ぶ。

現在のベクトルプロセッサは、本研究によって実現した高速化ベクトルアーキテクチャなどにより、パイプライン演算器をさらに空間的に多重に配備して並列に同時実行させ、演算器の使用効率も高く、プロセッサ設計として最も複雑な形態に達している。

ベクトルプロセッサは単一プロセッサで実現する最高性能を達成することを目的として素子技術と並列処理アーキテクチャを駆使したプロセッサ設計を行ってきた。

本研究では、ハードウェア実装技術とコンパイラ最適化技術を駆使して、プログラム実行効率を向上できるようなベクトルアーキテクチャを構築した。

幸いにも本研究の成果は効率の高いアーキテクチャとして、VP-100シリーズ以来、富士通のスーパーコンピュータに実現され、世の中に受け入れられてきた。

ベクトルプロセッサは繰り返し演算実行には極めて効率がよく、今後も科学技術計算処理分野で広く使用されていくと思われる。

1. 2 本研究の目的

スーパーコンピュータは、主に科学技術計算の高速実行を目的としており、気象、流体力学、素粒子物理、分子化学などの分野で、際限ない高速化が期待されている。ベクトルプロセッサは1970年代から実用化（製品化）され、スーパーコンピュータの代名詞となった。しかし、実際の一般的な科学技術計算プログラムに対して、初期のベクトルプロセッサは性能向上倍率に限界があった。この限界を打破してベクトルプロセッサの性能向上率をスカラプロセッサ比で50倍以上とすることが本研究の目標である。

本研究の目的は、次のとおりである。

- 1) ベクトルプロセッサにおけるアーキテクチャを分析し、その問題点を明らかにする。
- 2) その結果、広範囲のプログラムで高速化を実現するベクトル化率を高めるベクトルアーキテクチャの拡張を提案する。
- 3) 投入したハードウェア資源を出来るだけ有効に動作させるため、命令の非同期並列実行を実現するベクトルアーキテクチャを提案する。
- 4) これらのアーキテクチャに基づいたハードウェア実現方式を検討し、提示する。
- 5) 実現したハードウェアおよびコンパイラの性能を評価し、上記アーキテクチャとその実現方式が有効であることを検証する。

その目標を達成するには、並列パイプライン演算器を最大限に稼働させるアーキテクチャによって、大量に投入されたベクトルハードウェアを使い切るために、コンパイラ実現方式とコンパイラの最適化技術を十分に反映することが必要である。本研究がベクトルコンパイラ技術の発展に寄与し、さらに、ベクトルコンパイル技術が他の並列処理技術にも応用されることが望まれる。

本研究の目的を言い換えると、使い易い製品化を考慮して、最高の性能を最少のハードウェア資源で実現するための、投資効果の最も高い製品を供給するための、ベクトルアーキテクチャの構築とベクトルプロセッサの実現である。

ベクトルプロセッサは高速スカラプロセッサの自然な発展系として位置付けることができ、既存のプログラムからの自然な形での発展として、ソースプログラム互換性を維持しな

がらリコンパイルによって高速の性能を達成できることに意味がある。これは製品としての重要な資質であり、プログラミングの容易さも合わせて必要である。

1. 3 本論文の構成

本論文では、本章の序論を始め、8章によって構成されている。

第2章ではコンピュータアーキテクチャを高速化の面からレビューし、本研究への先行技術の影響を示し、本研究の位置づけを述べる。

第3章ではベクトルプロセッサの構成を述べ、先行アーキテクチャの問題点を分析し、問題点を指摘する。

第4章では、ベクトル処理の適用範囲を拡張するベクトルアーキテクチャとして、特に条件文ベクトル処理やデータ編集機能等を提案する。

第5章では、出来るだけ多くのベクトル命令を多重発信し、非同期並列処理を実現するアーキテクチャを提案し、情報の連鎖による同期逐次処理を実行する方式を、コンパイラとハードウェア制御との関連について考察する。またベクトルレジスタの構成方式と動的構成変更方式について提案する。さらに、ベクトル命令の並列実行時における、アドレス変換および割り込み処理アーキテクチャを提案する。

第6章では上記のアーキテクチャを実現するハードウェアを設計し、その命令の非同期的並列実行における制御方式を述べる。パイプライン演算器をさらにパイプライン制御する方式を考案した。物理的に多重にベクトルレジスタや演算器を構成して性能向上を図る方式を述べる。命令の発信タイミングの制御方式を記述し、命令の連結実行方式を提案する。さらに、ベクトルプロセッサで特に大事なメモリ制御方式を述べ、キャッシュコヒーレンシとチャネルバッファについての工夫と実現方式について述べる。さらにメモリアクセスを多重並列に動作させる方式を実現し、逆に同期をとるための命令の逐次実行方式を記述する。

第7章では本研究で構築したベクトルアーキテクチャとその実現方式について、総合的な評価を行い、目標に対する成果を示す。

第8章に結言を述べる。本論文をまとめ、今後の展望にも触れる。

1. 4 言葉の定義

以下に本論文中使用している言葉のうち、その意味を明確にすべきものを特に解説をしておく。狭義よりもさらに限定した使い方をしている場合がある。

(1) アーキテクチャ

広義には電子計算機のハードウェアの構成方式をいうが、狭義では計算機の論理的な仕様について言うことがある。本論文では後者の意味で使うことが多い。

(2) ベクトル

スカラに対応して使うことが多く、複数のデータを配列状に並べたデータ構成を言う。個々のベクトルデータはその配列同志を演算すると、それぞれの配列の同一順番の要素同志を演算する。

(3) テクノロジ

本論文ではハードウェアを構成する素子実装技術のことを言う。

(4) 独立／従属

独立と従属は反対の意味で使われる。特にデータ間でのつながりのことを言い、従属関係とはあるデータが別のデータに連鎖されることを言う。ある命令でレジスタに作られた結果を別の命令の被演算データとして使用するような場合、両者の命令間は従属関係にあると言う。一方従属関係にない関係を独立関係もしくは独立であると言う。

(5) 並列／逐次

並列とは同時に複数の事象が並行して実行されることである。ハードウェアを複数個配置すること、命令を複数個同時に実行することなどはそれぞれ並列回路、命令の並列実行と言う。これに対し、逐次は並列に反する言葉であり、ものごとを1つずつ順番に行うことで、たとえば命令の逐次実行とは命令を1つずつ処理することを言う。

(6) SIMD/MIMD

SIMDはSingle Instruction stream Multiple Data stream の略であり、MIMDはMultiple Instruction stream Multiple Data stream のことである。

単一命令で多数のデータを実行することがSIMDであり、多数の命令が多数のデータを処理するのがMIMDである。一般的には多数のプロセッサで多数のデータを実行することをMIMDと言う。

(8) ベクトルプロセッサ

ベクトル機能をアーキテクチャとして持つプロセッサのことをいう。

(9) パラレルプロセッサ (並列プロセッサ)

並列にプロセッサを並べた計算機システムを言う。

(10) クロック (サイクル) タイム

計算機の同期用クロックの時間間隔を示す。クロックあるいはサイクルタイムと言う場合もある。

(11) RISC

Reduced Instruction Set Computerの略である。単純な命令を用い、レジスタを出来るだけ多数配置し、命令の並列実行をし易くするアーキテクチャを言う。CISC (Complex Instruction Set Computer) においては複雑な制御をハードウェアで実行していたが、RISCでは出来るだけ複雑な処理はソフトウェア (特にコンパイラ) に任せ、ハードウェアは単純処理に専念する。命令の同時並列処理によって高速化を実現する。

(12) VLIW

並列RISCの一種であり、長い命令の中に複数の動作コードを配置し、複数動作の並列実行を実現するアーキテクチャである。

(13) データフロー

データが使用されていく経路を示す。途中ではデータの加工が行われるとしても、データが入力され、演算され、次の入力となり、最終的には出力されるそのデータの流れのことを言う。コンパイラでは木構造で示すことが多い。

第2章 コンピュータの高速化アーキ テクチャと本研究の位置づけ

2.1 コンピュータの高速化の歴史

(1) コンピュータの高速化

スカラコンピュータの性能は平均命令実行サイクル数と構成素子の速度（クロックサイクルタイム）の積の逆数で表現できる。

$$P = \frac{1}{T_c \times E_i}$$

P : 性能
E_i : 平均命令実行サイクルタイム
T_c : クロックサイクルタイム

コンピュータに使用された素子は真空管、リレー、パラメトロン、Geトランジスタ、Siトランジスタ、TTL、ECL、CMOS、BiCMOS、GaAsと発展し、クロックサイクルタイムを減少させた。

実装容積の減少は素子単体速度と同様にクロックサイクルタイムの低減に寄与した。真空管やリレーに比べると、固体デバイスでは物理的な素子の大きさが大幅に小さくなった。Si（シリコン）デバイス時代になると、高速素子として、電流切替え型のECL素子が高速素子を独占するようになり、1960年代の後半には1940年代のデバイスに比べて10の6乗倍のクロックサイクルタイム短縮が実現された。

一方、コンピュータ開発のごく初期に、演算器の並列化により平均命令実行サイクルタイムE_iは、数10倍改善された。さらにその後は時間的並列処理（パイプライン処理）により約10倍の改善があった。しかしその後は小さな改良程度だったが、1980年代の後半より、並列RISCアーキテクチャの採用により、1サイクル内で複数の命令が実行できるようになり、平均命令実行サイクルタイムE_iの値を1以下にすることができるようになった。ここでも1桁弱の改良がなされた。従ってテクノロジーに頼らないでアーキテクチャ（計算機構成方式）による高速化は、合計10の3乗程度の改良があったことにな

る。

この両者を併せると、コンピュータの歴史において、約10の9乗倍の性能向上が達成されたといえる。言い換えると、秒のオーダの命令実行時間がナノ秒オーダになったことになる。

(2) 高速化としての並列処理方式

この項では、上記の2つの高速化要因のうち、平均命令実行サイクルタイムの改良に注目して述べる。

平均命令実行サイクルタイムの短縮はコンピュータの設計方式、広い意味でのアーキテクチャ上の高速化の帰結である。以下にのべるように、アーキテクチャ上での高速化とは、すなわち時間的、空間的両方の並列化であったと言えよう。

初期の逐次処理では最少のハードウェアを繰り返し有効に使って、時間はかかっても、費用は安い処理方式であった。ハードウェア価格が急激に低下することにより、逐次処理から並列処理への移行していくが、その並列化の程度はその時代の回路実現規模と回路価格とによって決まった。

初期の計算機では真空管やリレーなど物理的にも大きく、価格的にも高かったため、できるだけ最少の部品回路を構成するか最大に関心事であった。従って回路構成も逐次処理方式を採用せざるを得なかった。

空間的な並列化による高速化の例としては加算器をあげることができる。

最も初期の段階でのコンピュータ設計において、図2. 1に示すように加算回路は1ビットの演算器でシリアルに加算を行い、結果として多重桁の結果を得る方式であった。性能向上のためには、高価格ではあるが図2. 2のように多重桁の並列加算回路を構成し、キャリールックアヘッド回路を設置し、桁上げを高速化し、全加算器を各ビットに配置して、数十倍の高速化を実現した。

図 2. 1 逐次型加算器

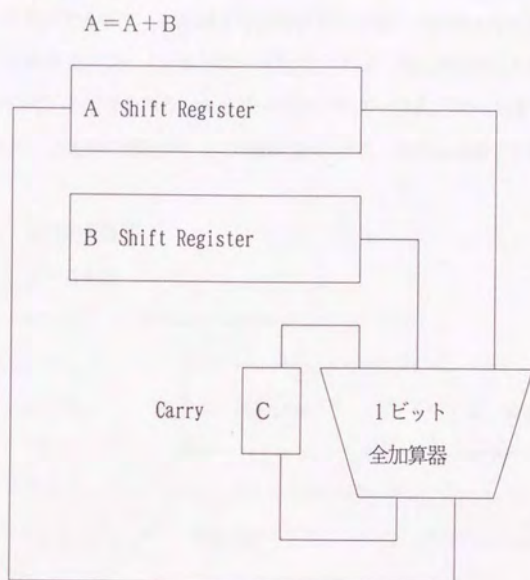
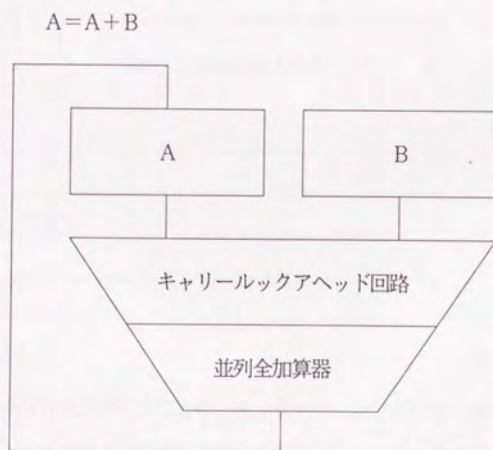


図 2. 2 並列型加算器

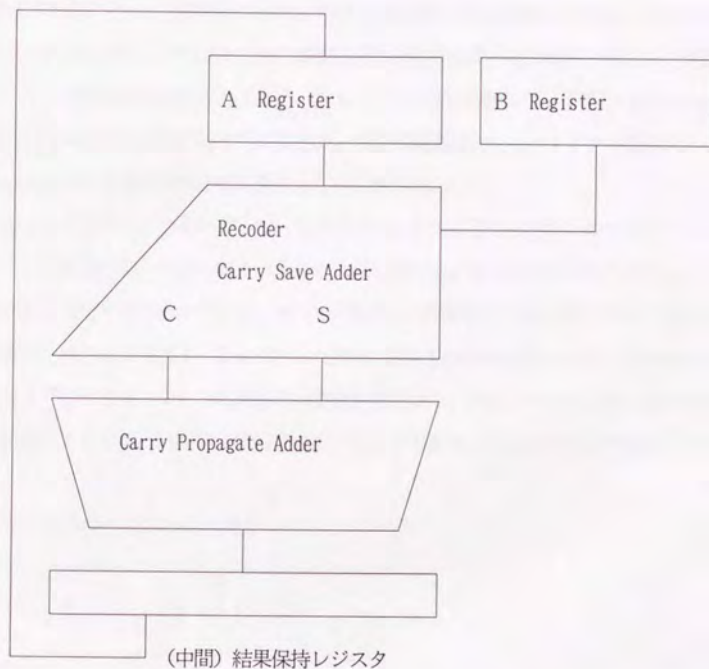


同様に、並列乗算器として図 2. 3 に示すようなに並列化による高速化の例がある。乗算

は加算と桁シフトの繰り返しで実行できるが、この複数演算を空間的に配置した桁シフト回路と、それぞれの桁ごとに加算する回路とを組み合わせたキャリーセーブアダによって構成する。乗数をデコード・エンコードして（リコーダ）ある桁ごとの被乗数を加算するかどうかのゲート信号を作り、キャリーセーブアダの入力ゲート信号とする。キャリーセーブアダは数段の論理ゲートで構築できるので、それを通過する遅延時間は小さく、

図2. 3 並列型乗算器

$$A = A * B$$



システムサイクルタイムごとに、キャリーセーブアダから出力を出すことができる。このとき、ハードウェアの量的な制限により、リコーダ・キャリーセーブアダの部分を節約し、乗数を分割してたとえば1バイトごとに各サイクルタイム毎に複数回実行することがある。この場合でも、上記リコーダ・キャリーセーブアダのバイナリ演算の特徴を生

かし、中間的な組み合わせ演算結果を高速に（数サイクル以内）で出力させ、最後にシリアルな加算動作をキャリープロパゲートアダーにより、一回のみに限定させることができる。本方式の採用により、結果的に一桁以上の乗算の高速化が実現できた。この方式の乗算器構成は最近のスカラプロセッサでは常識化されている。さらに、ベクトル乗算器には必須である。

一方、もう一つのアーキテクチャ上の高速化は時間的な並列化、つまりパイプライン方式によって実現された。

パイプライン方式とは、機能回路を小さな時間で実行可能な単位機能に分解し、小さな時間毎にその回路に異なるイベントをその機能回路に投入するように制御し、結果として各イベントが多数繰り返して実行されるため、立ち上がり時間が無視でき、前記の小さな時間で実行されるのと同価に見させる方式である。本方式は通常のフォンノイマン型のスカラプロセッサにおいて命令制御の基本方式として定着した。

図2. 4に示すように、命令の実行は、命令のフェッチI—デコードD—オペランドのフェッチF—命令実行E—結果の書き込みS等に細分化でき、命令は次の命令の実行とオーバラップして実行することができる。命令1の処理に時間間隔 τ の後すぐに命令2が続いて実行されるが、それぞれI、D等で示した機能回路に順次命令が投入され、命令が次々にちょうどパイプラインのように連続的に処理されていく。図2. 4では命令を別の流れのように書いてあるが、時間的な経緯を表現するためであり、見方を変えると図2. 5

図2. 4 命令のパイプライン制御

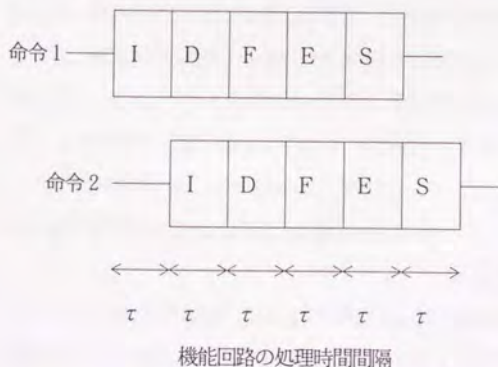
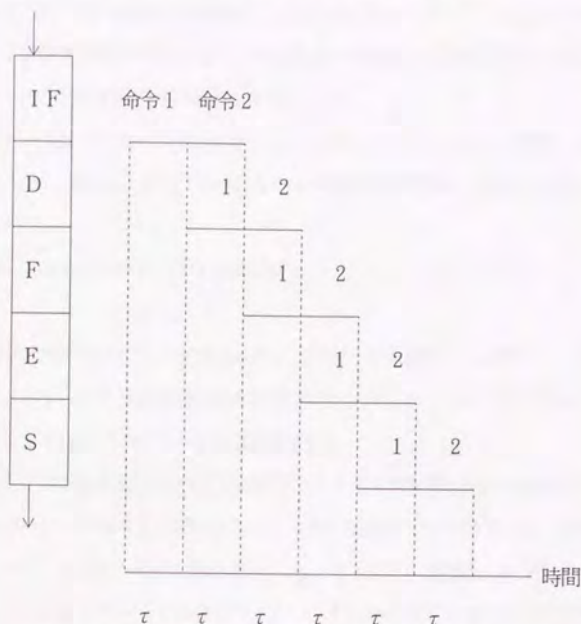


図2. 5 ハードウェアイメージでのパイプライン動作



に示すように実際のハードウェアでは、命令パイプラインに命令が連続的に投入される。さらに、後に述べるように、パイプライン処理方式はベクトルプロセッサにおいて、演算（加算など）の細分化された部分演算を多重に実行するベクトルパイプライン演算器に発展した。

現在では時間、空間的並列処理の両方を組み合わせて高速化していることが多い。

例えば、最近のRISCプロセッサでは、命令実行は複数の多重パイプライン処理が行われており、最高6個の並列命令実行数を達成したものもある。

他の例としては、ベクトルプロセッサでは、時間的な並列機能を持つパイプライン演算器をさらに空間的に多重に配備して並列に同時実行させることができ、さらに異種類のパイプライン演算器をさらに並列同時実行させることができるようになっている。

このような高度な並列化は究極の姿に近づいている。

テクノロジーの発展に伴い、固体電子素子はSi素子による特にLSI化への大変な発展を遂げた。素子技術もTTL、ECL、さらにGaAs系の素子へと発展した。こういう単

体素子の高速化競争時代では、高価格素子であるため、時間的並列処理が重視され、空間的な並列化（同一装置の多数配置）はあまり進展がなかった。最近のCMOSでは特にLSIの高密度化が極端に進み、それに伴う低価格化が大変な勢いで進行している。

このため空間的並列化も進展している。

論理ゲートレベルでの最少化は単一LSI内では必須であり、時間的な並列化が実現され、システム全体としては、同一LSIの空間的並列配置の可能性が強まると思われる。

（3） 命令アーキテクチャと高速化

当初のスカラコンピュータではメモリ上のデータを演算し、結果データをメモリに戻す言わばメモリメモリ演算を実行する機械であった。CPUには暗黙のレジスタ（累算器）があり、それとメモリデータの演算を実行した。

これを1アドレス命令といい、暗黙のレジスタ（累算器）を一つのオペランドとみなし、他のデータはメモリ上のデータをフェッチして他のオペランドとし、結果は累算器へ一時的に保持することが命令の動作であった。そしてこの累算器上のデータをメモリの該当番地にストアすることも1命令動作であり、その動作完了を待って、次の命令に移ることになる。

命令形式は図2. 6のように命令コードでロード/ストアも加算も指定する。Bでベースレジスタを、Xでインデックスレジスタを示す。この両者の内容を加算し、さらにMを加算してメモリアドレスを作る。このアドレスからデータを取り込んで、演算し、累算器に

図2. 6 1アドレス命令形式

OP	B	X	M
----	---	---	---

OP : 命令コード

B : ベースレジスタ番号（指定がない場合もある）

X : インデックスレジスタ番号（指定が無い場合もある）

M : メモリアドレス指定

保持する。ストア命令では、累算器のデータを上記と同じように計算して、あるメモリアドレスへストアする。

この方式は、命令の実行が完全に逐次的に処理されることが前提のアーキテクチャであり、個々の命令機能は順次逐次的に実行された。

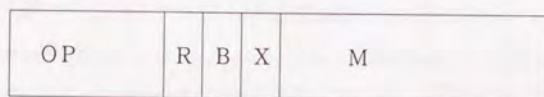
次に2アドレス方式について述べる。本方式は上記1アドレスと後で述べる3アドレス方式の中間に行く方式である。命令形式は図2. 7に示すように2種類に大別できる。メモリーレジスタ形式の命令ではRで示されるレジスタとB, X, Mで指定されるメモリアドレスのデータが演算され、Rで示されるレジスタに結果が保持される。ストア命令ではRで示されるレジスタの内容がメモリにストアされる。

レジスタ-レジスタ形式の命令ではR1とR2で示されるレジスタの内容が演算され、結果がR1で示されるレジスタの中に保持される。

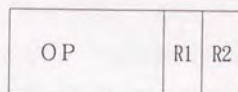
R2で示されるレジスタの内容がメモリアドレスを示す時は、該当するメモリの内容が被演算データとなる。

図2. 7 2アドレス命令形式

メモリーレジスタ形式



レジスタ-レジスタ形式



OP, B, X, M: 図2. 6と同じ

R: オペランドを示すレジスタ番号

R1: 第一オペランドを示すレジスタ番号

R2: 第二オペランドを示すレジスタ番号

本方式ではレジスタが複数個指定され、演算データのバッファリングが可能となる。ただし、1命令内でレジスタは被演算オペランドが読み出されたあと演算結果を格納するために書き換えられる。従って命令単体としては、上記の1アドレス方式と同等な命令の逐次実行形態となる。しかし、複数の命令で異なるレジスタを結果レジスタとして使用するような命令シーケンスを指定すると、複数のデータフローを指定することができる。

2アドレス方式では折角の並列実行性を持ちながら、1アドレス的なメモリデータともう一つのオペランドの同期が必要な命令内逐次処理が必要であり、ハードウェアの実現には1アドレス的な要素と3アドレス的な要素を合わせ持つ必要があり、複雑な回路構成が必要となる。

さらにデータフローの独立性を利用した並列実行のためには、多くのレジスタ数が必要となり、少数のレジスタアーキテクチャではリネーミングが必要（ハードウェア用に別置された中間データ用の別レジスタに退避）となる。

特にIBMの360・91/95等は中間的に3アドレス中間コードを生成すのほどの複雑な実現方式を取ったが高速化には限度があった。大きな原因はプログラムレジスタの数（浮動小数点レジスタは4個）が不足していたことが大きい。

次に3アドレス方式について述べる。この命令形式を図2. 8に示す。やはり2つの命令形式に大別され、メモリアクセス命令と演算命令とに分離される。メモリーレジスタ形式には演算命令はなく、メモリとのロード/ストア命令に限定される。演算命令はもっぱらレジスターレジスタ命令に限定される。

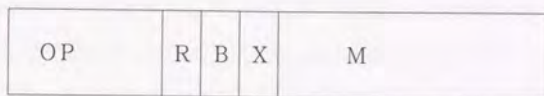
このアーキテクチャを前記2アドレス方式と比較してみる。2アドレス方式ではメモリーレジスタ命令が存在しており、オペランドを読出したレジスタそのものに結果を格納するため、メモリデータを直接、演算器のオペランドとして、上記のレジスタオペランドと同期して供給しなければならない。さらにレジスタオペランドのみの演算命令もあるためレジスタへからの読出しバスも必要であり、さらにメモリからのデータはレジスタへの書き込みも必要である。従ってレジスタの書き込みルート回避するバイパスするルートを設置する必要がある等、非常に冗長な演算バスが必要になる。

一方、3アドレス方式では、演算はレジスタデータ同志の間のみ実行されるので、メモリアクセスと演算が分離されるため、レジスタはメモリアクセスと演算のタイミングのずれ

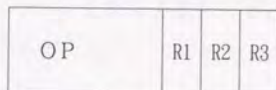
を許容するバッファになることもできる。従って、3アドレス方式は2アドレス方式に比べると、逐次的命令実行のレイテンシーの短縮に対する要求が少なく、非同期的な多重命令を並列実行するハードウェアを実現し易い方式である。

図2. 8 3アドレス命令形式

メモリーレジスタ形式



レジスターレジスタ形式



R3: 第3 オペランド

その他の記号は図2. 7と同じ

従って、最近のRISCでは3アドレス方式が常識になっている。

並列RISCアーキテクチャは並列同時命令実行を行わせるのに適しているが、実際にはコンパイラが命令のスケジューリングを最適化する必要がある。

(1) レジスタ競合への考慮

レジスタ演算命令では、同一レジスタを読出オペランドと結果格納レジスタとに共用すると、データのアクセス順序関係逐次実行が必要である。連続する命令間でレジスタ共用を行うと、クロックサイクル毎のパイプライン的な命令実行ができず、各命令実行時間を逐次的に費してしまう。コンパイラは意識して、レジスタコンフリクトチェックにかからないように、相互の命令を引き離し、間に別の命令を詰め込む必要がある。

(2) メモリ競合への考慮

同一アドレスへのメモリ書き込みが生じた場合、同様にメモリアクセスの同期処理を実行する必要がある。ストアとロードの論理的な順序関係はハードウェアでチェックし、命令の実行順序に従って逐次処理されるため、メモリアクセス待ち時間が生じる。コンパイラはメモリアクセス命令相互間に、出来るだけ並列同時実行可能な命令を詰め込んで、待ち

時間の軽減を図る必要がある。

スカラプロセッサのキャッシュ上にデータが存在すれば、同一アドレスのデータの読出し書き込みは上記レジスタアクセス程度の影響で済むので大きな問題にはならない。しかし、同一アドレスへのアクセスが繰り返し数多く実行されたりして、メモリアクセスの影響が出てくる程度になると、特にストアスルー方式のキャッシュといえども、性能低下の原因となりうる。

スカラ命令アーキテクチャは以上のように、1アドレス方式から3アドレス方式まで発展したが、並列同時実行の可能性を高めるには3アドレス方式が優れている。今後の高速スカラプロセッサアーキテクチャは3アドレス方式に限られていくと思われる。最近の並列RISCアーキテクチャではもっぱらこの方式が採用されている。

3アドレス方式はベクトル命令でも採用され、ベクトル命令相互間の並列同時実行にも大きく貢献している。

2. 2 ベクトルアーキテクチャと高速化

(1) スーパーコンピュータの方式

科学技術計算を高速実行するコンピュータのことをスーパーコンピュータと言う。スーパーコンピュータと言う言葉は必ずしも、はっきりした定義があるわけでは無いが、その時代で比較して格段に高速性を発揮する機械のことを言う。1970年代の初頭までは超高速スカラコンピュータもスーパーコンピュータと言われていた。最近までは商用機ではベクトルプロセッサがスーパーコンピュータの代名詞であったが、近頃はベクトルプロセッサの並列システムはもちろん、スカラプロセッサの並列システムもスーパーコンピュータと言うことがある。

初期のスーパーコンピュータの方式はFORTRANのDO LOOP文を高速化することを目指して開発された。そこでは、従属的關係(DO LOOPのあるサイクルで生成した結果が後のサイクルで使われること)にない、つまり独立な演算を繰り返し実行することを仮定し、同時並列実行による高速化を図ることを目指していた。この方式は多くのデータに対して同一演算を施すと言う意味で、いわばSIMD (Single Instruction stream Multiple Data stream)方式とすることができる。

スーパーコンピュータの基本方式にはベクトル方式と並列方式がある。

ベクトル方式は、時間的な並列処理としてのパイプライン演算器によって、SIMD処理をベクトル命令によって同一演算を繰り返し多数データを処理することにより高速化を実現した。商用機としてのスーパーコンピュータはTI ASCから、CRAY-1、富士通 F230/75APU、最近のCRAY C90、NEC SX-4、日立 S3800、富士通 VP2000、VPP300までベクトル方式が主流である。ベクトルプロセッサ方式では自動ベクトル化コンパイラの技術が進んでおり、ベクトル化によってスカラのみ性能に比べて数倍の性能が容易に達成でき、ベクトル化が原因で性能低下することは殆ど有り得ないため、商用機として成功したと思われる。

一方、並列方式は、空間的にプロセッサを多重並列に設置する高速性実現方式のことを言う。初期の研究的並列処理システム（ILLIAC IV）は、SIMD方式の並列システムであり、中央制御プロセッサから命令が放送され、各プロセッサはこの指令にもとづき、自分が担当するデータの処理を行う。この方式ではSIMDの範囲であり、ベクトル方式と同様な範囲で有効であった。

並列プロセッサシステムは、SIMD方式から脱却し、個々のプロセッサはそれぞれの命令を実行する独立プロセッサとして動作するようになった。これをSIMDに対してMIMD（Multiple Instruction stream Multiple Data stream）方式と言っている。MIMDにもとづく並列方式はプログラミング技術およびコンパイラ技術が未熟であり、場合によってはプロセッサ間のデータ通信性能の限界により、単一プロセッサ性能より悪化することもあり、製品化に苦勞した。しかし、最近では標準マイクロプロセッサの性能が極端に向上し、大幅な価格低下が生じており、価格性能比としてのメリットを生かす方向での製品化が進んでいる。さらに並列処理のソフトウェア技術も大幅に改良がすすんでいる。最近の超並列商用機では殆どの機械がMIMD方式を採用している。

（2）ベクトル処理方式

1990年代の当初までは、素子の高速化に期待が強く、高速素子ECLの性能改善やさらにGaAs等のテクノロジーによる高速化を多くの人が期待していた。いわば単体プロセッサ性能に対する楽観的な改善期待感が強かったと言える。

しかし、高速素子の価格は高く、従って、できるだけ少量のハードウェアで効率よく回路を使用し、使用効率を上げることによって、価格性能比を向上させることが、有利であった。ハードウェアの使用効率の向上による価格性能比の向上と言う観点からは、ベクトルアーキテクチャ型のパイプライン演算器は非常に優れており、スーパーコンピュータとしてはもっぱらベクトルプロセッサがその代名詞であった。

少なくとも1980年代から、1990年始めでは素子テクノロジーの高速化の割合も現状より急峻であったため、空間的並列よりも時間的並列（ベクトル処理）型のベクトルプロセッサのスーパーコンピュータが歓迎され、本研究のように徹底的なアーキテクチャの高度化を行い、超高密度の実装技術を併せて、プロセッサの高速化を図り、絶対性能の向上とともに実効効率の向上を図ってきた。その結果、価格性能比が向上し、世の中に広く受入

れられている。

独立並列処理が有効な分野では、ベクトル型パイプライン演算器は極めて効率がよく、使用回路規模を最少にしてLSI上に高集積化し、価格性能比として効率の高いプロセッサに内蔵して使用されていくと思われる。

(3) ベクトルアーキテクチャの原理

先の述べたFORTRANのDO LOOPを高速化することがスーパーコンピュータの第一の目標であった。DO LOOPとは次のような処理を言う。

```
DO 1000 I=1, 100  
A(I)=B(I)+C(I)*D(I)      --①  
END DO
```

1から100番目までの100個の異なるデータ同士に対し、それぞれ①の計算を実行し結果を求める。この処理は別々の独立したデータを演算するが、それぞれ同一動作を行うことに特徴がある。従って、A, B, C, Dはそれぞれ100個の配列データとして定義しておくことが便利であり、これを配列（またはアレイ）と呼び、こういう複数データの並びをベクトルと呼ぶことがある。

個々のデータ同士は独立であるため並列同時実行が可能であり、かつ同一計算の繰り返しでのSIMD方式ではパイプライン処理が可能である。同様に多重プロセッサによる並列処理も可能である。

スカラー命令でのコーディング例は仮想的なマシンコードで②のようになる。

C, D, Bのデータをメモリからロードし、掛け算、足し算を実行して、結果をAにストアする。さらにインデックスIを繰り返し上げて、分岐命令BRCで先頭のLOAD命令に戻る。これをインデックスIを変化させて1から100まで100回繰り返す。この他、こ

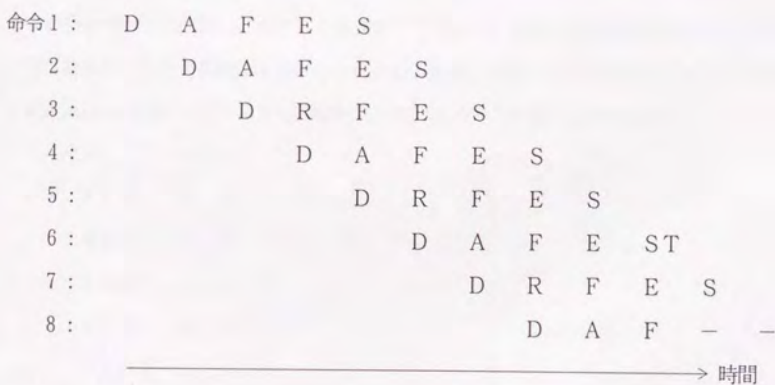
LOAD	0, C (I)	--1	}	--②
LOAD	2, D (I)	--2		
MULT	4, 0, 2	--3		
LOAD	6, B (I)	--4		
ADD	8, 4, 6	--5		
STORE	6, A (I)	--6		
ADD	INDEX	--7		
BRC	*-8	--8		

のループの外にアドレス計算等の初期化命令が必要である。

②の命令実行は5段の命令パイプラインを仮定すると図2. 9のようになる。

各命令は順次Dフェイズに次々と投入され、それぞれ5つの命令パイプラインセグメント

図2. 9 スカラ命令の命令実行タイムチャート



D : デコード

E : 実行

A/R : アドレス計算/レジスタ読出

S : 結果のレジスタへのストア

F : 命令/オペランドのフェッチ

ST : 結果のメモリへのストア

で部分的に実行され、最後に結果をレジスタ等にストアする。命令8はブランチ命令であ

り、アドレス計算後命令フェッチを行い、命令1にブランチする。従って第9番目および10番目の命令とは実は、インデックス変更後最初に戻る命令1、2のことである。

このシーケンス（命令1から8まで）を100回繰り返すことになる。

本命令シーケンスの実行には、平均命令実行時間を2クロックとすれば、式③のクロック数が必要となる。

$$8 \text{ 命令} \times 2 \text{ クロック} \times 100 \text{ 回} = 1600 \text{ クロック} \quad \text{---③}$$

一方、ベクトルでは④のようなベクトル命令シーケンスになる。

ここではA, B, C, Dは長さ100のアレイデータを示し、数字の0, 2, 4, 6, 8はベクトルレジスタを示す。各レジスタは100個以上の要素数をもつものと仮定する。スカラ命令と同様に、メモリデータをベクトルレジスタにロードし、レジスタ上のデータを演算し、その結果をベクトルレジスタに保持し、最終結果はメモリ上にストアする。スカラと違うのは、それぞれの命令で100個（ベクトル長に一致）のデータをまとめて処理することである。

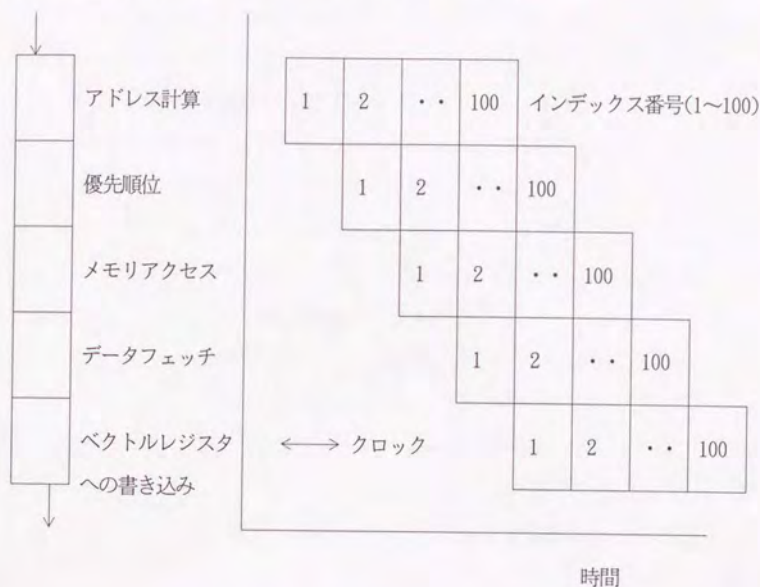
この命令シーケンスの前に、ベクトル長指定、アドレス計算等の命令が初期化命令として必要であるが、ここでは省略する。さらにスカラ命令シーケンスとは異なり、インデックス修正用の処理はハードウェアで自動化されているので、命令には現れない。

VLD	0, C	}	---④
VLD	2, D		
VMD	4, 0, 2		
VLD	6, B		
VAD	8, 6, 4		
VSTD	8, A		

VLD（ベクトルロード）命令は仮定マシン上では図2. 10のように実行される。

図2. 10のぎざぎざのある菱形を1つの命令実行単位として表す。

図2. 10 ベクトルロードパイプラインのタイムチャート



ロード、ストアパイプライン、掛け算、足し算パイプラインに対してそれぞれ同様な菱形で表現すると、ベクトル命令列④は複数のベクトルパイプラインによって、図2. 11のように連続して動作する。

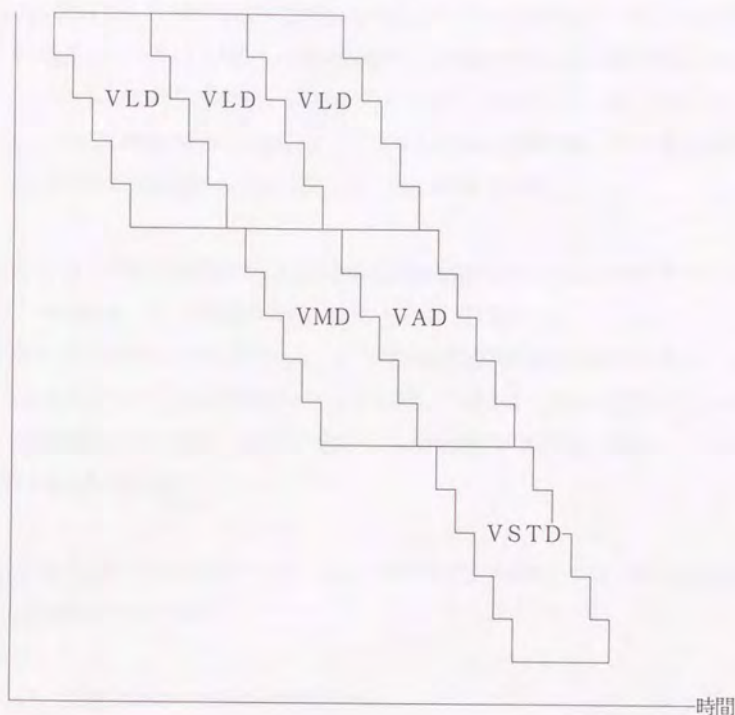
VLDパイプラインは、次の命令が来ると前の命令のすぐ次のタイミングで連続して投入できる。VMD命令は、同様にVLD命令の第1番目のデータがベクトルレジスタに書き込まれるとすぐに命令が投入できる。

この時の実行時間は次の式で表される。

$$\begin{aligned} \text{必要クロック数} &= 100 \times 3 + (\text{VLD, VAD, VSTDの立ち上がり}) \quad \text{---⑤} \\ &= \text{約} 350 \end{aligned}$$

以上のような原始的なベクトル命令制御でのタイミングで比較してもスカラとベクトルの性能差は約4.5倍となる (④/⑤)。

図2. 11 ベクトルパイプラインのタイムチャート



さらに物理的なパイプラインの数をたとえば8本にして、8個の連続インデックスデータを同時に実行すれば、さらに8倍の性能になる。ロード/ストアパイプラインを2組用意すればさらに最大2倍の性能が得られる。上記の例では3個のVLDの一つが他の一つと重なり、ベクトル長が長い場合には次のような性能が得られる。

$$(1600/200) \times 8 \times 2 = 64 \text{ 倍}$$

ベクトル命令の立ち上がり時間のデメリットを考慮すれば、この倍率までは単純に性能が上がるわけではない。一方、スカラ命令の実行時間の方も、キャッシュミス等のペナルティもあるため、互いに相殺して50倍程度までの性能を目標とすることができる。

本研究ではこのような高速性能を達成するベクトルアーキテクチャを構成することが、目標である。

スカラ命令ではインデックス増分の繰り返し回数分の命令数の実行が必要であり、命令フェッチ用のメモリアクセスが必要である。しかし、ベクトル命令ではベクトル命令1つでベクトル長（インデックス増分）の動作を実行するため命令読出しは1回で済む。つまり、DOループ1回分が1つのベクトル命令のハードウェア制御シーケンスにリダクションされる。従って命令の読出しに必要なメモリアクセスが大幅に削減でき、メモリ転送性能をデータアクセスに殆ど使うことができ、コスト面でも有利である。

一方、ベクトル命令を動作させるための初期化用の命令が必要でこれがスカラ命令として実行されるため、ベクトル方式の性能へのオーバーヘッドになる。

しかし、スカラプロセッサの場合でも、ループ制御初期化用の命令が必要であるし、さらにキャッシュのミスヒット等のオーバーヘッドもあり、ベクトル・スカラ双方ともオーバーヘッド要因があるため、ベクトル化の効果はベクトル長が非常に短い場合を除いて、それほど落ちることはない。

以上のようにベクトルアーキテクチャはスカラアーキテクチャに比べて、約50倍の性能を目標とすることができる。

2. 3 高速化技術の中での本研究の位置づけ

(1) ベクトル処理の歴史と発展

初期のベクトル型アーキテクチャでは、初期の1アドレス方式のスカラ命令アーキテクチャと同様に、メモリデータ同志の演算を行う命令形式を前提としていた。この時代では、ベクトルデータのデータアドレス、要素間の距離、データ種類（浮動小数点単精度、倍精度）、ベクトル長などを指定するディスクリプタによって、メモリ上のデータを指定していた（FACOM 230-75 APU）。従ってメモリ上のデータを演算し、その結果も直接メモリ上へストアするメモリーメモリーアーキテクチャであった。この時代はベクトルプロセッサの第1世代であり、スーパーコンピュータの揺籃期であった。

メモリーメモリー命令形式では結果データをメモリにストアするため、そのデータを次の命令でも使用することが多く、メモリへのストアが完了してからでないと、次の命令を開始

することができない。従って、次の命令は前の命令の完了を待たざるを得ず、立ち上がり時間をオーバーラップして、実行時間を短縮することが困難であった。敢えて、前後の命令オーバーラップを実行するには、メモリアドレスを相互チェックするための大変なマトリックス回路が必要であり、実現することは非常に困難であった。

その後、ベクトルレジスタの導入によりスーパーコンピュータはCRAY-1によって、第2世代へと進んだ。スカラ命令形式と同様にベクトルレジスタを指定する3アドレス命令形式の命令に発展した。ベクトルレジスタにデータを保持するアーキテクチャの採用により、ベクトルレジスタにいったん保持されたデータ同志を演算することができ、メモリアクセス命令も独立に動作可能であるため、命令の並列同時実行がやりやすい。命令の同期関係はオペランドの同期制御に他ならず、命令相互間のレジスタ競合チェックに限定できる。従って、メモリーメモリアーキテクチャのように個別のメモリデータアドレスのチェックの必要はなく、複数のデータのまとまりであるベクトルデータ間、つまり1かまとまりのデータを代表するベクトルレジスタ番号間の前後関係のみの競合チェックをし、命令の制御をすればよい。ベクトルレジスタ番号を扱うことは命令の同期処理の制御回路を小規模に簡略化することを意味している。従って、ベクトルレジスタの導入はベクトル命令の並列実行管理を容易とし、同時実行命令数も大幅に増やす可能性を高めたと言えることができる。

(2) 本研究以前のベクトルアーキテクチャの問題点

1970年代から実用化（製品化）された、いわば第2世代のベクトルアーキテクチャではコンパイラによる自動ベクトル化が可能な範囲は単純な演算のみであり、全体のプログラムの中でのベクトル化による高速化の適用範囲が小さいため、ベクトル化対象以外は依然スカラ処理が残り、スカラ処理性能に比べて、ベクトル化処理性能の効果は相対的に小さかった。例えば、ベクトル化が半分しか出来なければ、いくらベクトル性能が高くてもプログラム全体の性能向上はたかずか2倍でしかない。

従って初期のベクトルプロセッサではベクトルパイプライン用のハードウェアを大量に投入しても意味がなく、適当な投資量に限定されたため、ベクトル化による高速化の効果も

平均化するとスカラ性能に比べて10倍以下程度に限定されることになり、ベクトルプロセッサの実効的な性能向上率を低下させていた。

(3) 本研究の位置づけ

上記のように第2世代のベクトルプロセッサは一部のプログラムへの適合性はあったが、例えば、IF文を含むような一般的なプログラムでは高速化率として問題があった。単体プロセッサ性能の高速化限界を超えるためには、大量のベクトルハードウェアを投入する事が必要であり、物理的に多重並列にパイプライン演算器を構成して、並列ベクトル処理を実現してそれを効率良く動作させることが必要である。そのためにはベクトルアーキテクチャを工夫することにより、ベクトルそのものの性能を大幅に向上させたい、ベクトル演算の適用範囲を大幅に拡大し、スカラ処理部分を縮小させ、プログラム全体の性能を向上させることが必要である。このために、いわば第3世代の新規高速化ベクトルアーキテクチャを構築して、超高速単一プロセッサの実現をすることが本研究の目標である。

本研究では第2世代のベクトルアーキテクチャの問題点を明らかにし、より広範囲に高速化を実現するためのアーキテクチャを考察した。本アーキテクチャの構築に当たり、ハードウェアの実現方式を考案し、コンパイラ技術の可能性と限界を充分に考慮した。

コンパイラの自動ベクトル化機能により、ベクトルプロセッサは高速スカラプロセッサの自然な発展系として位置付けることができる。つまり、既存のスカラプログラムとソースプログラム互換性を維持しながら、もう一度だけリコンパイルすることによって高速の性能を達成できることに意味がある。従って、本研究では自動ベクトルコンパイラ技術の動向に合わせてベクトルアーキテクチャを構築した。

本研究の他の目的は、最高の性能を最少のハードウェア資源で実現することである。言い換えると投資効果の最も高い製品を供給するための、アーキテクチャの構築とハードウェアの実現である。

本研究で実現したベクトルアーキテクチャは、単一プロセッサの究極的な高性能化を目的としており、同一テクノロジーでのスカラプロセッサに比べれば、数倍から50倍の性能を

実現することが研究の目標である。

自動ベクトル化を実現したコンパイラ技術は、最近、広く採用されている並列RISCアーキテクチャのコンパイル技術の発展にも貢献しているため、本研究は高速並列処理技術の発展に大きな刺激を与えたと言うことが出来よう。

現在の最新RISCプロセッサではデータおよび命令キャッシュがその高速化に大きく貢献しているが、ベクトル処理になると、ベクトルプロセッサには性能的にまだまだ追いつくことは出来ない。つまり、もともとピーク性能差が大きく、さらにスカラプロセッサではベクトルデータアクセス時にキャッシュペナルティが大きく、折角の高速スカラ性能が発揮できないことがあるからである。特にベクトル長が大きいようなプログラムでは、キャッシュからデータがオーバフローして、キャッシュミスの回数が多くなるためである。特に最近の並列RISCアーキテクチャでは平均命令実行時間に占めるキャッシュミスのペナルティの影響が大きくなっていることがさらに相対的速度低下を引き起す。

VLIWアーキテクチャはスーパスカラ方式に比べると命令の並列度を大きくすることが可能であり、並列処理されるデータの数もベクトル処理と同等にすることが可能である。その意味では、VLIW方式はベクトル方式に近づく可能性を持っている。

ベクトル方式と異なるのは、あくまでデータはそれを伴う命令に1対1であり、データの数と同数の命令（VLIWでは動作指令部）が必要である。従って命令をフェッチする分の最終的にはメモリアクセス量が、ベクトル命令のベクトルデータに対応して1つだけで済むのに比べて、どうしても多いことになる。また、スカラプロセッサとしてはキャッシュは不可避と思われ、上記で述べたように、大規模問題でのペナルティは避けられない。

以上述べたように、アーキテクチャ的には、ベクトルアーキテクチャは並列RISCおよびVLIWに比べて高速性からみたメリットを持つ。

今後の標準マイクロプロセッサの開発競争により、スカラプロセッサの高速化がさらに進む可能性はある。しかし同一素子テクノロジーによってベクトルプロセッサを実現すれば、必ずスカラプロセッサ以上の性能が実現できるはずである。ベクトルプロセッサの開発にはスカラプロセッサ開発に比べて開発コストは多少大きいと考えられる。しかしプロセッ

サ絶対性能の差は明らかにベクトルプロセッサの方が優位であり、開発の意義は大きいと考えられる。

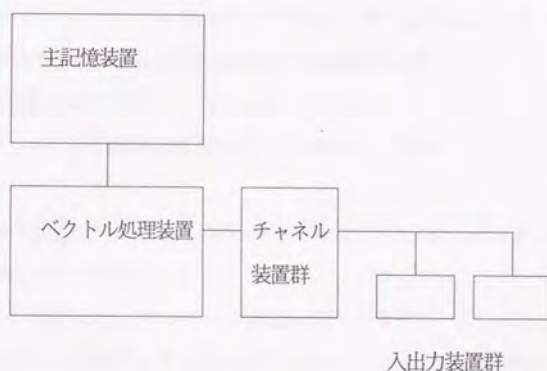
今後、ベクトルプロセッサの開発が継続されている限り、その優位は保たれ、本研究の成果も活用され続けられると思われる。

第3章 ベクトルプロセッサの分析と問題点

3.1 ベクトルプロセッサのシステム構成

本分析の前提として、図3.1に示すように、ベクトルプロセッサシステムの最も単純な構成要素を想定した。

図3.1 想定したシステム



ベクトル処理装置は単一のプロセッサではあるが、スカラー処理を実行するユニットとベクトル処理を実行するユニットからなる。スカラーアーキテクチャに加えて、ベクトルアーキテクチャを拡張し、単一命令シーケンスを実行する。命令実行制御は通常のフォンノイマン型のコンピュータと同様に命令フェッチ、データフェッチ、実行、結果の格納を実行し、それを繰り返す。割り込み機能等も通常のコンピュータと同様に処理する。

主記憶装置は両方のユニットから共有される単一空間としてアクセスされ、さらにチャンネル装置からも同一空間としてアクセスされる。チャンネル装置はベクトルプロセッサからは従属的なプロセッサである。ベクトルプロセッサから指令を受け、それぞれの命令シーケンスを実行し、その終了をベクトルプロセッサに割り込みで報告する。チャンネル装置は入出力装置を制御し、主記憶装置と入出力装置との間のデータ転送を管理する。

プログラムから見えるベクトルプロセッサに内在するアーキテクチャ上の資源として、次のものを想定する。

制御レジスタ

整数スカラレジスタ

浮動小数点スカラレジスタ

ベクトルレジスタ

マスクレジスタ

制御レジスタはプロセッサの動作モードの設定、割り込み状態の設定、各種タイマ、割り込み時の状態表示、障害時の状態表示等の情報設定表示を行う。

ベクトル長を指定できるベクトル長レジスタを持つ。

整数スカラレジスタは固定小数点のデータを保持し、整数データやアドレスの計算に使用される。

浮動小数点レジスタは、浮動小数点データを格納するためのレジスタで、単精度、倍精度、4倍精度データを保持することができる。

ベクトルレジスタはベクトルデータを保持するレジスタで、固定小数点データ、単精度および倍精度浮動小数点データを保持する。本レジスタの構成は本研究の対象であり、動的に構成を変えることができる。その詳細は後述する。

マスクレジスタは条件ベクトル処理を行うためのレジスタであり、各要素のデータは1ビットのデータである。本アーキテクチャではベクトルレジスタと同数の要素数をもつ。

3. 2 ベクトル化の可能性の分析と問題点

ベクトルプロセッサの性能を単純化したモデルに基づいて議論する。

ベクトル命令はスカラ命令に比べて常に α 倍の性能の性能を達成するとし、スカラ命令とベクトル命令は逐次的に実行されるとする。

ベクトルプロセッサの性能をスカラ命令で実行していたときと比較して、ベクトル命令も併せて実行するときに達成できる性能向上率を P とする。スカラ命令で実行していたときにかかった時間で測定して、ベクトル化できる部分のプログラム実行時間の割合をベクト

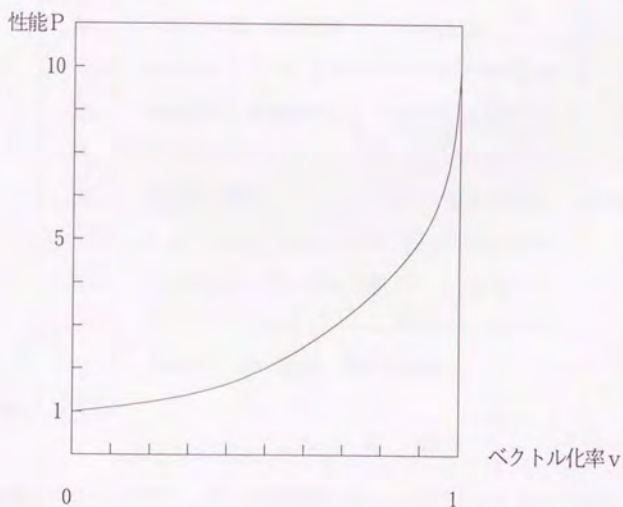
ル化率と定義し、 v とする。 α を変数とする v と P との関係は①のような式で表現される。これをアムダールの式ということがある。

$$P = \frac{1}{v/\alpha + (1-v)} = \frac{\alpha}{v + (1-v)\alpha} \quad \text{---①} \quad v=0 \sim 1$$

図3. 2には加速率 $\alpha=10$ の場合のベクトル化率とプロセッサ性能を示す。

v が1に近く、殆どベクトル化されたプログラムでは、性能 P はベクトル加速率 α に近い性能を達成できるが、 v が0に近く、殆どスカラで実行されるプログラムでは、 α がいくら大きくても、 P は1に近くスカラ性能と殆ど変わらない。また α が無限大であったとしても、ベクトル化率 $v=0.5$ であれば、全体性能 P はたかだか2倍でしかない。さらに $P=5$ 、つまりピーク性能の半分を達成する場合でも $v=0.89$ でなければならない。

図3. 2 ベクトル化率とベクトルプロセッサ性能



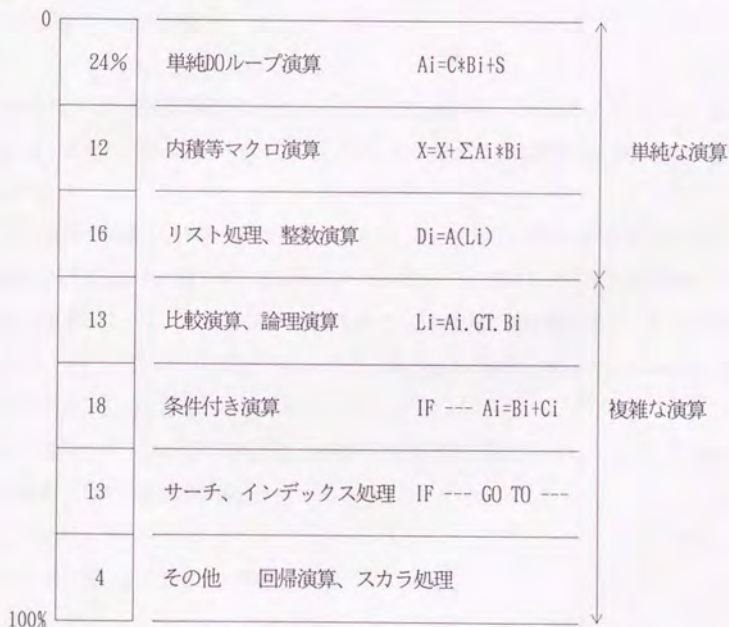
第2世代のベクトルプロセッサでは $v=0.5$ 程度であったため、ベクトル性能を十分に発揮できなかった。

以上の結果、ベクトル性能によって世界最高の単一プロセッサ性能を実現するためには、ベクトル化率を向上させることが、必須である。従って、第2世代の機械で何がベクトル化を高める障害になっているのかを分析することにした。

この目的で、我々は現実に科学技術計算ユーザが実際に使用されている一般的な科学技術計算プログラム、数100本を統計的に分析した。DO LOOPやそれに相当する処理を選び出し、理論的にベクトル処理が可能となるはずの機能を分析した。

分析結果を図3.3に示す。

図3.3 プログラム分析結果



各演算はDOループなど、繰り返し演算を含むプログラムカーネルであり、演算の意味を分析して、分類をした。

この分析により、ベクトル処理は四則演算、内積演算、リスト/整数演算等の単純ベクト

ル演算と、条件付き処理、比較演算、インデックス処理、サーチ等の複雑なベクトル演算とがあり、それぞれ約半分ずつ存在することが分かった。

単純ベクトル演算のベクトル化は第2世代アーキテクチャでも可能であったが、複雑な演算のベクトル化を実現するためには、DO LOOP内の意味を解釈してベクトル化できるコンパイラが必要であることが分かった。さらにそのためには、コンパイラが複雑なベクトル処理を行うことができるオブジェクトコードを、容易に生成できる機能を備えたベクトルアーキテクチャを構築しなければならない。

上記の複雑なベクトル処理を分類、整理した結果、次のような機能に分類でき、これらの機能をハードウェア命令の機能として実現しなければならないことが分かった。

(1) 条件ベクトル処理

上記で述べた複雑な演算の内容を調べると、IF文がDO LOOPに内在することが非常に多く、それに伴う、論理演算や、インデックス処理用の演算が沢山出現することが分かった。

IF文の実行結果は、YES/NOまたはTRUE/FAULSEで示されるが、これを論理データで1/0に置き換えて表現し、その論理データを用いてベクトル処理化することができる。ここで、IF文に係わる処理をベクトル処理に置き換えることを、条件処理のベクトル化と言い、IF文をベクトル化して実行される処理を条件ベクトル処理と呼ぶことにする。

さらに条件ベクトル処理に付随して、論理データ処理や下記に述べるインデックス処理などが組み合わせられることが多い。

(2) インデックス/ビット処理

FORTRANではインデックスの概念がある。インデックスとは配列データの何番目かを指示する数または変数であり、評価をすれば1から始まって配列の長さNまでの数で示される。一方、配列の相対的メモリアドレスは0から始まり、N-1で終わる。従って、インデックスと相対的メモリアドレスとは1つずれており、この変換も必要である。

インデックスに従って配列データの演算を施すこと、インデックスそのものを処理、演算することも多いことが分かった。図3. 3の中では、リスト処理、比較演算、論理演算、およびインデックス処理などがこの処理に関係している。

(3) データ編集処理

何らかのデータの操作を行う処理であり、次のような機能を含むことが分かった。

- ①検索処理（最大値／最小値の検索等）は、0 次再帰演算機能であり、結果はスカラデータとインデックスをその結果として伴うことが多い。
- ②変数のデータ変換や、データの移動（ムーブ）など、四則演算を伴わない操作も結構多く、単純機能ではあるが無視できない。
- ③ビット配列やインデックスとその配列を制御データとするデータの入替え、並べ換えなど編集機能も重要であり、ビット配列とインデックスの変換、該当データに対するインデックスの抽出など制御データ相互の編集機能も必要である。

以上をまとめると、単一ベクトルプロセッサの平均的な性能向上率が、第2世代のベクトルプロセッサのようにかたかだか2倍と言う状況を脱して、ベクトル加速率 α に近づけるためには、図3. 3に示した「複雑な演算」に相当する演算をベクトル化することが必須である。

3. 3 命令の並列実行可能性に関する分析と問題点

ベクトルプロセッサはベクトル長に応じた多数のクロックでベクトル命令を実行することと、ベクトル演算器はパイプライン構成をとり、かつ複数のベクトル演算器を持つため、多重演算器による命令の物理的並列実行とパイプライン演算器による時間的な命令並列実行（パイプライン制御）を両方実行でき、スカラ命令に比べてはるかに多くのベクトル命令同士の多重並列同時実行が可能となる。つまり、ベクトルプロセッサの命令実行アーキテクチャは、物理的、空間的な命令多重処理を前提において、実効命令処理数（実効性能）を大幅に向上させる命令発信が可能である。しかしながら、第2世代のベクトルプロセッサでは、せいぜい4～5命令の並列実行しか可能でなかった。

(1) ベクトルレジスタの数と容量の不足

第2世代ベクトルプロセッサ(CRAY-1)ではベクトルレジスタの数が8個である。同一の命令では全てのレジスタ指定部で別のレジスタを指定しなければならない、かつ命令の並列実行時は全て異なるベクトルレジスタでのみ可能である。例外的に、同一レジスタを指定して、命令が同時に実行できるのは、先行命令の結果レジスタと後続する命令のオペランドレジスタが同一だったとき、いわゆるチェイニングのときだけである。さらに複数のチェイニングを行う場合は、別のチェイニングレジスタを指定しなければならない。従って、1演算命令あたりに3オペランドが必要であり、ロードストア命令では1オペランド使用するため、例えば2演算命令と2ロードストア命令など、命令の同時実行可能数はたかだか4命令である。以上のようにCRAY-1における命令の並列同時実行は、チェイニングを前提として、たかだか数命令の間の並列実行しか考慮していなかったことを示すものであり、多数命令を並列実行するアーキテクチャではなかった。

一方、ベクトルプロセッサの最大の長所の一つは、ベクトルパイプラインを物理的に多重に(たとえば4個の多重化)設置して、配列をベクトル長の $1/4$ に折り畳んで命令実行時間を $1/4$ にし、実行性能を向上させることができることである。しかし、この機能はベクトル長を短くするのと等価であり、立ち上がり時間の影響度が高くなる。従って単一レジスタ内の要素数を増強する必要がある。第2世代アーキテクチャにおける、指定できるレジスタ数とベクトルレジスタの容量では、性能要求に対して大幅に不足している。さらに応用プログラムではベクトル変数が多数必要なものやそれほど必要でないものがある。ベクトルの長さも多様である。従って、ユーザプログラムに合った変数の数と配列の要素数を指定できると良い。しかし、レジスタ構成は固定の方が設計は容易であり、可変構成は実現されなかった。本研究では後述するように、独自の機能としてベクトルレジスタの可変構造を実現した。

(2) メモリアクセス量の不足

CRAY-1のメモリアクセスパイプラインは1本しかなく、加算/乗算と同等の本数である。単純な加算における2つのメモリオペランドを足して、結果をメモリに返すような

演算では、メモリパイプラインネックとなり、演算性能の1/3以下の性能しかでない。さらに、たかだか8個のレジスタでは、多数の演算がDO LOOP中にある場合には、オペランドをベクトルレジスタに保存することができず、メモリへの退避が必要となり、その分のメモリアクセスが余分に必要になる。従って、メモリアクセスに対する高いデータ転送能力が必要であり、ただでさえ負荷の高いメモリアクセス回路の負担をさらに増加させることになる。

これとは別に、前記のユーザプログラムの分析において、メモリアクセスの必要量を、四則演算に対する相対比率として測定分析した。その結果を表3. 1に示す。加算、乗算、を同等の頻度とすれば、それぞれの演算に対して同等のアクセス頻度のロード演算が必要であり、さらにその半分頻度のストア演算が必要であることが分かった。

表3. 1 メモリアクセス頻度

演算	出現頻度
四則演算	57%
ロード演算	30%
ストア演算	13%

平均的なスカラ命令出現頻度

この分析では、プログラムで使用されたスカラ命令の出現頻度の統計であり、正確にはベクトル命令に対する数値と一致しないが、等価であるとした。その結果、複数のベクトルメモリアクセスパイプラインが必要であることが分かった。

従って、複数のロード/ストア命令を同時に実行可能なアーキテクチャを作る必要があることが分かった。

(3) 順不同な命令の並列実行

多くの命令が並列実行できる環境では、各命令が独立であれば、起動順に命令が終了する

必要はない。従って、命令の実行は順不同である方が演算器を有効に使用することができる。第2世代のアーキテクチャでは順不同の命令制御までは考慮されていなかったが、特に短いベクトル長のベクトル命令を実行するときには、演算器の使用効率が低下する可能性が強く、複数命令実行により演算器の使用効率を高めることが必要である。そのためには、できる限り命令の先行実行を行うアウトオブオーダー（順不同）の命令実行制御が必要となる。しかしプログラム実行論理を保つには、逆に論理的逐次実行を陽に指定しなければならない。従って、逐次実行命令シーケンスを多重に実行する必要がある。

この考え方をメモリアクセス命令の多重実行まで拡大するために、逐次実行命令シーケンスの多重実行指定を実現する独自のアーキテクチャを導入する必要がある。

ベクトル命令においては、アドレス変換において動的アドレス変換を実現して、命令の再開可能性を追求すると大量の冗長な回路が必要となる。性能重視の観点からの発想の転換が必要であり、静的なアドレス変換によってプログラム容易性を制限するかどうかの検討が必要である。

さらにアウトオブオーダーの命令実行を実現すると、割り込み時の状態保管方式を、逐次実行方式の場合と大幅な変更が必要である。さらに、例外が多重発生することを前提としてアーキテクチャを考慮する必要がある。

（4）メモリアクセス方式とキャッシュの整合性

ベクトルプロセッサのハードウェア設計上で最も重要なポイントはメモリアクセスの効率である。大規模なベクトルプログラムではとくにプロセッサからメモリまでの間のデータ転送率は強大であるため、スカラプロセッサのキャッシュのような資源はベクトルアクセスに整合性が悪いという問題がある。

データを一時蓄積しても、ベクトル処理では大規模なデータをメモリの広い範囲に渡ってアクセスするため、常にデータの入替えが生じ、データの再使用が出来ないため無駄である。従って、ベクトル命令におけるメモリアクセスはメモリ直接アクセスが有利である。この方式は、極端にベクトル長が短い場合のメモリアクセスの立ち上がり時間のペナルティがあるので、上記の大規模処理の要望とのトレードオフが必要である。

一方、スカラプロセッサにはキャッシュが必須であり、ベクトルストアによるデータのストアに対して、キャッシュのコヒーレンシを保つ必要がある。この機能は純粹にハードウェア機能で実現し、ユーザからはトランスペアレントにすることが好ましい。

第4章 ベクトルアーキテクチャの機能拡張

前章で分析した問題点を解決するために、本章ではベクトル化の範囲を拡張するためのベクトルアーキテクチャを提案する。条件処理をベクトル化する方式とデータ編集に関連する複雑なベクトル処理をベクトル化する機能と命令について、その必要理由とその効果を含めて述べる。

4.1 条件処理のベクトル化

条件処理とはFORTRANで言うIF文を含む処理のことである。前章の繰り返し演算の分析からIF文を含むDO LOOPの割合は全体の殆ど20%であり、複雑な演算の中に限って言えば約40%にもなるため、条件処理のベクトル化が必須である。

条件ベクトル処理を実行するには、ハードウェア機能として3つの基本方式（マスク演算方式、集散命令方式、リストベクトル方式）があり、それぞれの方式の特長と欠点を示し、それらの方式の中からコンパイラが最適な機能を選択する方法を提案する。

例えば②のような文を例に取って説明する。

```
DO 30 I=1, 50
  IF A(I).GT. 3
    E(I)=F(I)
  ELSE
    E(I)=B(I)+C(I)*D(I)
```

-----②

```
30 END DO
```

この例ではA(I)の値が3より大きければ、E(I)にはF(I)をいれ、さもなければE(I)にはB(I)+C(I)*D(I)（この評価結果をG(I)とする）を入れる。

これをスカラ演算で実行すると、IF文の条件判定によって、E(I)にF(I)もしくはG(I)を入れるためのそれぞれの命令列に分岐し、別個の命令列を実行する。従って

条件に合致しない命令列は実行しないで、スキップすることになる。

一方、ベクトル演算でこのループを実行する場合は、上記で述べた3つの方式がある。

以下にこれらを説明する。

(1) 演算マスク方式

この方式で条件ベクトル処理を行う場合は、それぞれの命令を一通り全部実行して、個々の要素データに対する制御情報（ビット配列）によって、それぞれの要素ごとに条件設定にもとづいた動作を行う。

まず、IF文の実行はベクトル比較命令を実行し、条件判定をするための制御データとしての論理データ配列をマスクレジスタ上に作る。そのマスク配列にもとづき、条件合致命令列と非合致命令列を演算マスク機能付きで実行し、両方とも全てのインデックスIについて結果のベクトル配列を作る。最終結果は、2つ出来た結果ベクトル配列の各要素を、マスクデータに従って一方の結果に他方の結果を上書きすることになる。

図4. 1にベクトル命令形式を示し、これにもとづいて、詳細に説明する。第4オペランド(R4)としてマスクレジスタの指定し、マスク機能の制御情報として使う。

図4. 1 演算命令形式

FE	OP	R2	R1	R3	R4
0	8	16	24	36	48 63

FEはベクトル命令を示すコードであり、OPはベクトル命令の動作を示す。たとえば加算、乗算と言った演算の種類である。図4. 2に演算マスク機能の動作を示す。命令形式ではベクトルレジスタR2とR3で示されるベクトルレジスタのそれぞれの同一インデックス(番号)の要素を演算し、R4で示されるマスクレジスタの同一インデックスの内容が1の時はR1で示されるベクトルレジスタの同一インデックスの要素を演算結果で置き換え、R4で示される同一レジスタの内容が0の時はR1の要素はそのままに保持する。

このマスクレジスタの内容による、要素入替え制御をマスク機能と呼ぶ。

この方式の基本的なアイデアは既に実現されているが、本研究では演算命令の中で同時にマスク機能を実行してしまうことに独自性がある。

図4. 2 マスク機能



R 4 指定部を0にする、つまり、マスクレジスタ0を指定することにより、マスク機能を無効にすることができる。この場合は、マスクレジスタの内容はすべて1と等価であり、全てのR 1ベクトルレジスタの内容が演算結果で置き換えられる。

このように、マスク機能を指定することにより、通常の演算命令の実行と同時に条件ベクトル処理が実行できるため、R 1の入替え操作分を別命令で実行する必要がなく、ベクトル処理としては性能的にはI F文の有る無しで同一である。しかし、ベクトル演算では合否の両方の動作を実行する必要がある、実行時間が掛かる。

一方、スカラ演算では条件によって命令をスキップするための分岐命令を実行する必要がある。逆に、スカラ演算では条件合致か否かにより、一方の命令列のみの実行で済む。

以上のように、スカラ演算とベクトル演算の優劣は両面があるが、ベクトル加速率が非常に大きいため、ベクトル演算の方が有利である。

ベクトル比較命令でのマスク機能は、I F文が入れ子になっているときに現れる。OP部

で比較命令を指定したとき、R 4 部で指定したマスクレジスタの内容とのAND機能を果たす。つまり、R 4 で示されるマスクレジスタには多重のIF文の一つ前の条件ベクトルデータが入っている。従って、マスクレジスタの該当インデックスの内容が0のときは、結果は0が入る。また、マスクレジスタの内容が1の時は比較演算結果の値がR 1 で示されるマスクレジスタに入る。従って、R 4 部でマスクレジスタ0を指定しなければ（演算マスク機能を有効にすれば）、比較演算のマスク機能が実行される。それは、実は両論理データのAND演算が実行されているのと等価である。

比較演算におけるマスク機能でも演算とマスク機能が同一命令内で実行されるため、実行時間の短縮となり、命令実行性能の向上に寄与する。

以上のように、IF文の入れ子処理ではベクトル比較命令自身でマスク機能を有効にするため、コンパイラはベクトル処理に付加的なベクトル命令が要らないため、スカラ処理で実行する場合のように、付加的なブランチ命令の挿入に比べて有利である。しかし、マスク機能によって実行する条件ベクトル処理では、多重のIF文の入れ子の数が多くなるほど、冗長実行の確率が高まり、性能の限界が出てくることが普通である。実アプリケーションプログラムでは3重程度以内のIF文の入れ子が普通であり、ベクトル化による性能効果は高い。

マスク演算処理では、スカラ演算の時は演算をスキップしてしまうデータをベクトル命令で演算してしまうが、最後にマスク情報で演算しなかったように見せ掛ける。しかし、マスクすべき演算で例外条件が発生したときは、条件発生自身も抑止する必要がある。詳細は割り込みの項で述べる。

(2) 集散命令方式

マスク方式とは異なり、本方式では本来の配列から条件に合致した要素だけを取り出した新しい配列で演算を行い、その結果を本来の配列に格納することになる。条件に合致する要素の取り出し、格納にはそれぞれに対応するベクトル集散命令が必要である。比較命令により作り出した条件を示すビット配列はマスクレジスタ上に置かれており、ベクトル集散命令の制御データとして使われる。

ビット配列に基づいて該当する要素を取り出す命令を用意した。

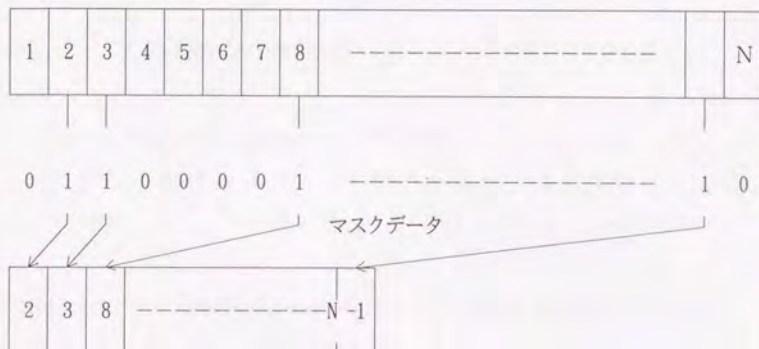
この命令をベクトル収集命令 (Vector Compress)と言う。

本命令の動作を図4. 3に示す。まず、ベクトルレジスタ上の入力データとマスクレジスタ上の制御データを読み出し、制御データが1である要素のみを抽出して結果のレジスタに前から順番に詰め込む。

この命令を実行した結果のベクトルデータには、マスク (制御) データが1のみのデータが集まっている。従って、ベクトルの長さは元の入力データの長さ以下になる。こうしてベクトル長の小さなデータ同士で演算できるため、演算時間はその分だけ短縮される。

図4. 3 ベクトル収集命令

入力ベクトルデータ



結果ベクトルデータ

この処理では、マスク用の制御データの数を勘定する命令が必要であり、本研究では Vector Sum Mask と言う命令を用意した。この命令により、マスクデータ配列のなかの1の総数を求めることができる。この値が結果ベクトルデータのベクトル長となる。

最終的に、条件成立時の演算の結果を元の入力データと同一の配列の当該場所に戻してやる場合には、上記の収集命令と逆の動作を行う分散命令 (Vector Expand) も用意した。

この集散方式では、いったん条件成立したデータのみ取り出して演算し、該当要素のみをまたもとの配列に戻す条件ベクトル処理を、ベクトルレジスタ上のみで実行できることが特徴であり、次に述べるリストベクトル方式との大きな違いである。

(3) リストベクトル方式

この方式はもう1つのデータ集散にもとづく条件ベクトル処理方式である。リストベクトルとはインデックスを集めた配列に従って、メモリ上のデータ配列に対し、データを集めたり、しまいこむことを言う。この操作を行うために、間接アクセスベクトルロード／ストア命令を用意した。この命令は、ベクトルレジスタ上にあるインデックス配列を利用して、メモリ上に分散しているデータをベクトルレジスタにロードする。逆にメモリにストアするのが間接ストア命令である。条件ベクトル処理では条件に合致したインデックスを取り出してきて、その集合体としてのインデックス配列を使用する。

ベクトル間接ロード命令の命令形式を図4. 4に示す。

図4. 4 ベクトル間接ロード命令形式

FF	OP	R 2	R 1	X 2	B 2	D 2
----	----	-----	-----	-----	-----	-----

FFはメモリアクセス用のベクトル命令を示し、OPは具体的な命令コードを示す。

R 2で示されるベクトルレジスタの内容はそれぞれの要素のインデックスを示している。ベクトル間接アクセスロード命令のアドレス指令部X 2, B 2の各汎用レジスタ上のアドレスデータを加算し、さらにD 2部を加算する。この値は、配列の仮想的な0番目の要素のメモリアドレスを指定している（FORTRANとしてはインデックスは1から始まるがここでは0番目の要素を仮定している）。

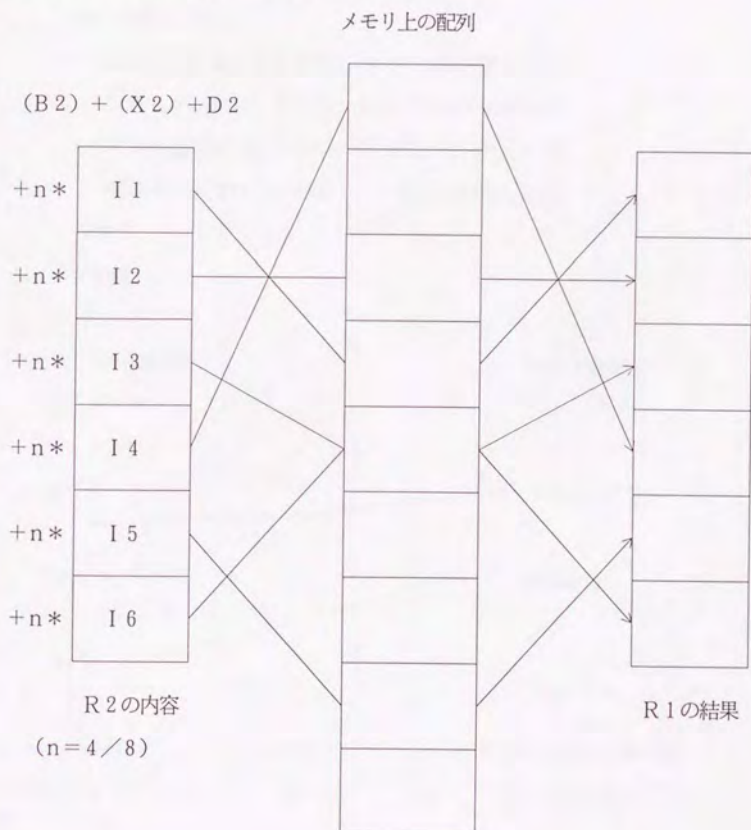
さらに本研究対象ハードウェアはバイトアドレス方式であるため、アクセスするデータのサイズに応じて、上記ベクトルレジスタ上のインデックスに4バイトまたは8バイトを乗じた値（2または3ビットシフトして）でベクトルデータのメモリアドレスをランダムにアクセスする。結果のベクトルレジスタには該当するデータが収集されて保管される。

ベクトル間接ロード命令の動作を図4. 5に示す。

この条件に合致するインデックスを収集してベクトルレジスタR2上のインデックス配列を作るには以下の通りを行う。

まず等差級数(1、2、・・・、N)をベクトルレジスタ上に作る(Vector Generate Arithmetic Series命令による)。ベクトル比較命令が生成した条件判定ベクトルをマスクデータとして、この等差数列即ちインデックス配列データから、該当するインデックスをVector Compress 命令によって収集し、結果のインデックス配列をベクトルレジスタ上に生成する。結果インデックス配列データのベクトル長さはビット配列方式で述べたのと同じく、Vector Sum Mask 命令を用いて求める。

図4. 5 ベクトル間接ロード命令の動作説明



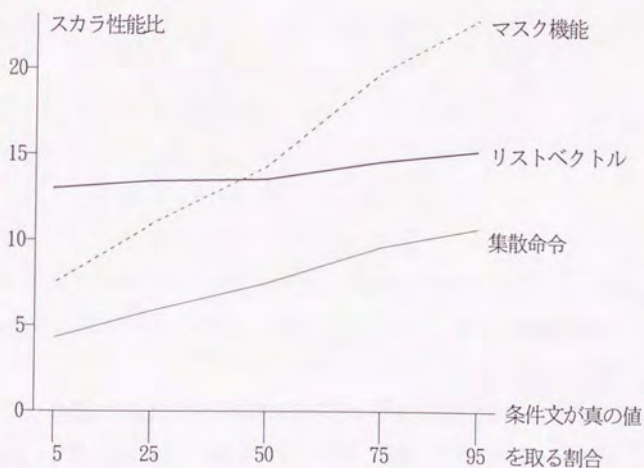
条件成立データ同志での演算が終了すると、ベクトル分散命令で行ったのと同様に、同一インデックス配列データを利用して、ベクトルレジスタ上のデータを、メモリ上の配列の当該部分に分散格納する（ベクトル間接ストア命令による）。

(4) 3方式の評価と選択方法

上記3方式に対して、例題による性能評価を行った。この結果を図4.6および図4.7

図4.6 条件ベクトル処理プログラム例1

```
DO 10 I=1, N
  IF ( M(I) ) THEN
    A(I)=0.25*(B(I+1, J)+B(I-1, J)+B(I, J+1)+B(I, J-1))
    C(I)=0.25*(D(I+1, J)+D(I-1, J)+D(I, J+1)+D(I, J-1))
    E(I)=0.25*(F(I+1, J)+F(I-1, J)+F(I, J+1)+F(I, J-1))
    G(I)=0.25*(H(I+1, J)+H(I-1, J)+H(I, J+1)+H(I, J-1))
  ENDIF
CONTINUE
```

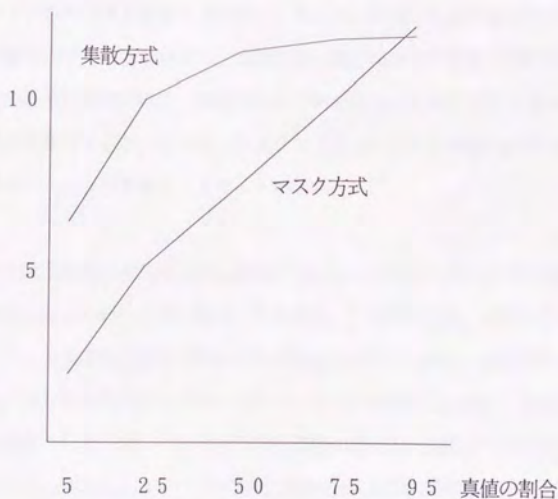


に示す。図4. 6はオペランドのメモリアクセスが比較的多い場合、図4. 7は演算密度が高く、メモリアクセスが少ない例を示す。

図4. 7 条件ベクトル処理プログラム例2

```
DO 10 I=1,N
  S1 = A(I)
  S2 = B(I)
  IF ( M(I) ) THEN
    S2 = DLOG((1.0 + S1)/(1.0 - S1)) * 0.5
  ENDIF
  C(I) = S1 + S2
10 CONTINUE
```

スカラー比性能



マスク、集散、リスト方式の実行時間を T_m , T_g , T_l で表し、次の式で示す。

サフィックスの m , g , l はそれぞれマスク、集散、リスト方式を示す。

$$T_m = T_{sm} + T_{em}$$

$$T_g = T_{sg} + t * T_{eg} + T_{xg} + T_{vg}$$

$$T_l = t * T_{sl} + t * T_{el} + T_{xl} + T_{vl}$$

T_s はメモリアクセス時間、 T_e は演算実行時間、 T_x はデータ集散に関する時間、 T_v はベクトル長の変更に必要な時間、 t は真値の割合を示す。

マスク方式では条件処理がベクトル命令に内在しているためロードストア命令と演算命令実行時間のみが必要である。場合によっては、さらにロード命令、演算命令の同時実行ができ、式の表現よりも命令実行時間が短縮される。

集散命令方式では一度メモリからロードされたデータはベクトルレジスタ内で集散が行われ、演算は真値の割合分だけの実行時間で済む。つまり、 t を T_{eg} に乗じた演算時間となる。ベクトル長変更時間は必要である。

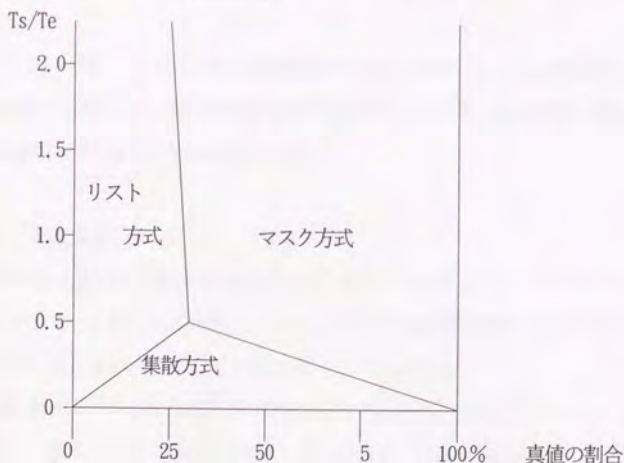
リスト方式では真値の割合 t に応じてロード/ストアを実行して集散機能を兼用する。従って、 t を T_{sl} 、 T_{el} ともに乗じて良い。しかし、あらかじめインデックス配列をつくり出す時間が必要であり、その時間は T_{xl} 、 T_{vl} の中に含まれる。

図4. 6ではロードストアデータが多く、マスク/リスト方式が有利である。真値の割合が小さい場合はリスト方式が良く、真値が高い場合はマスク方式が有利である。一方、

図4. 7では演算密度が高く、集散方式が有利である。リスト方式ではロードストア命令が集散機能を兼用するが、インデックスリストをつくり出す時間が必要であり、この例ではそのオーバーヘッドが大きく、メリットがない。

コンパイラは上記の3方式の中から最適なものを、パラメータによって選択する機能を持つ必要がある。その選択方法を図4. 8に示す。この方式では、本来は T_s 、 T_e 、 T_x 、 T_v 、ベクトル長等の変数で決めるのが適当である。しかし、選択式の簡単化のために、ロード/ストア命令の数と演算オペレーションの数の比 (T_s/T_e)、条件式での真値を持つ割合を代表とした。コンパイラはこの2変数によって、上記3つの方式で最適なものを選択することに決めた。コンパイル時間との競合もあり、厳密な解よりも簡略化を重視した。

図 4. 8 条件ベクトル処理の選択



(5) マスクレジスタの構成

条件ベクトル処理に関連して、マスクレジスタの構成要件を検討した。

マスクレジスタはベクトルレジスタの各データと1対1の関係で対応させると、各マスクデータはベクトルレジスタを直接制御する情報として位置づけられ、単純な対応関係を作ることができる。

しかし、マスクレジスタの数は、主に条件ベクトル処理のIF文の数に余裕として幾つかを足した数でも良く、ベクトルレジスタよりも少ない数にすることもできる。

一方、マスクレジスタの各要素のデータは1ビットの長さなので、ベクトルレジスタの64分の1の容量であり、ハードウェアの使用量は小さい。従って容量よりも、ベクトルレジスタとの統一性を重視する方を優先し、ベクトルレジスタと同一の構成をとることにした。従って、ベクトルレジスタと同一の容量をもつマスクレジスタを用意した。また本研究で構築したアーキテクチャではベクトルレジスタの可変構成が可能であり、マスクレジスタも同一構成にした方が良い。最大のベクトルの長さを指定するとマスクレジスタは8本となり、多重IF文で取り扱う数との関係で適当な数と判断した。

4. 2 ベクトル処理拡張機能

ここでは、図3. 3で示された複雑な演算のうち、条件ベクトル処理以外の演算について基本機能に分解し、これを実現する命令を設置した。それら基本機能の組合せによって、上記複雑な演算を実行する方法も示す。

(1) 検索機能

検索命令は最大値/最小値の検出をして、該当データ及びそのインデックスを浮動小数点レジスタもしくは、及び汎用レジスタに入れる。複数の該当データがある時にはインデックスの一番小さいインデックスを汎用レジスタに入れる。

浮動小数点レジスタまたは汎用レジスタが0であると、該当するレジスタには結果を入れない。つまりレジスタ番号0は特別な意味を持ち、結果の格納を行わないことを示す。

検索命令は線型一次方程式の計算におけるピボットの検出等に有効である。

検索命令は従属的な演算を含んでおり、配列に含まれているデータ同志の比較を全演算の何処かで実行する必要がある。従って全てのデータ同志が独立であると言う本来のベクトル演算の仮定には反しているため、バイナリ選択等の余分の処理時間が必要である。

このように少なくとも演算結果の一つが、他の演算に関係する従属関係をもつ演算のことを回帰的演算と呼ぶ。検索命令の場合のように、最も単純な回帰演算を0次の回帰演算と呼ぶ。

(2) インデックス/ビット操作機能

インデックスとマスク論理データの変換命令や、これらを制御データとして演算動作を行う基本機能を命令として用意した。

条件ベクトル演算で述べたように、マスクデータは論理データとして0/1のビット配列情報である。一方、インデックスでの要素番号指定も必要である。従って、前記のリスト方式の条件ベクトル処理で述べたように、ビット配列とインデックス情報は互いに変換しなければならないことがある。もちろん、一方の制御データのみで閉じた演算方法（ビット配列のみ等）では、変換の必要はない。

この範疇に入る機能を持つ命令は次のとおりである。

- ①マスクレジスタ上のデータの中から1の値をもつ最少のインデックスを探す命令
- ②同じく最大のインデックスを探す命令
- ③等差級数を発生する命令 (1、4、7、10、13・・・)
- ④あるインデックスに対応するビットを1にして他を0にするビット配列を生成する命令
- ⑤ビット配列のうち1の数を勘定する命令

それぞれの命令単体では単純な演算ではあるが、これらの命令の単体動作も再帰的演算を含むことがある。

上記の命令はその他の命令、例えばデータ転送命令、との組み合わせで、複雑な演算機能を実現することができる。

(3) 論理演算機能

次に論理データ配列処理演算命令がある。

ベクトル比較命令で作られた、マスクレジスタ上のビット配列データを、例えばIF文の入れ子の各条件に応じた論理演算が必要となる。

そのためには、次のような命令が必要である。

AND/OR/NOT/EXOR他の基本的な論理演算命令。

さらに、上記とは異なる論理演算命令がある。ベクトルレジスタ上の64ビットのデータを個々の論理データ要素とし、上記のAND/OR等の論理演算を施す命令を持たせた。

(4) データ編集機能

データ転送やデータ取り出し、データの放送等、データを加工、入替えなどの編集を行う命令を用意した。おもな命令の機能は次のように分類できる。

- ①ベクトルレジスタ間でのデータ転送する。
- ②指定されたインデックスに該当するベクトルレジスタの要素を取り出し、汎用レジスタまたは浮動小数点レジスタに入れる。
- ③スカラデータをベクトルレジスタの各要素にブロードキャストする。
- ④該当するベクトルレジスタ上の論理データ(64ビット)をシフトする。
- ⑤入力ベクトルレジスタの各要素毎にそれぞれ指定されたシフト数に従って、シフト

して、結果レジスタのそれぞれの要素に格納する。

②の命令の逆の動作、つまり汎用レジスタまたは浮動小数点レジスタ上のスカラデータを該当するベクトルレジスタの各要素に入れる動作は次のような、組合せ動作で実行することができる。つまり、上記③の命令に、マスク機能を使うことにより、適当なインデックスの要素のみスカラデータを挿入することができる。このマスクデータを作るために、スカラレジスタで指定したインデックスに対応して、1つだけマスクデータを1とする命令を作った。

同様に③のブロードキャスト命令で、マスクデータに複数ビット1を立てておけば、複数のスカラデータを該当するインデックス番号に書き込むこともできる。

上記の④、⑤の命令は64ビットの論理データを論理操作(AND, OR等)とも組み合わせ、ライブラリの中で関数評価を実行する際の、数表の代用として使われることがある。

(5) 収集分散命令

データの集散機能を持つ収集分散命令を用意した。これらは条件ベクトル処理のところですでに述べた。マスクデータに従って適当なデータを集めたり、分散させたりする。これらの命令では、入力データのインデックスと出力データのインデックスがずれるため、間にデータのバッファリングが必要であり、実現方式は複雑である。しかし基本的なデータ操作として本命令を採用した。

(6) 1次回帰関数用命令

次に示す関数を1次回帰演算(1st Order Recurrence)と言う。本演算はインデックス間での依存関係があり、ベクトル演算にはならない。しかし、線形1次方程式の解法で、逆方向入替えの時など、線形計算で良く現れる関数である。

$$A(I) = A(I-1) * B(I) + C(I)$$

本関数は、本質的に通常のベクトル命令ほどに高速化するのが困難である。

ただし、式の展開によって、多重項の演算の順序を工夫することによって、最多重項の演

算シーケンスのタイミングの中に、他の演算項を埋没させることができ、スカラ処理の約2倍の性能が達成される。

式④ではインデックス順に3つのデータを求める場合に展開した式を示す。

演算開始にはインデックス0に相当する数が必要である。これを $A(0)$ とする。命令ではこの定数をスカラレジスタ上に S として定義する。

演算の繰り返しはバイナリトリート方式で演算をスケジューリングすれば、演算器への投入回数は $\log_2 n$ となる。

$$\begin{array}{l}
 A(0)=S \\
 A(1)=A(0)*B(1)+C(1) \\
 A(2)=A(0)*B(1)*B(2)+C(1)*B(2)+C(2) \\
 A(3)=A(0)*B(1)*B(2)*B(3)+C(1)*B(2)*B(3)+C(2)*B(3)+C(3) \\
 \cdot \\
 \cdot \\
 \cdot \\
 A(N)=A(N-1)*B(N)+C(N)
 \end{array}
 \quad \left. \vphantom{\begin{array}{l} A(0)=S \\ A(1)=A(0)*B(1)+C(1) \\ A(2)=A(0)*B(1)*B(2)+C(1)*B(2)+C(2) \\ A(3)=A(0)*B(1)*B(2)*B(3)+C(1)*B(2)*B(3)+C(2)*B(3)+C(3) \\ \cdot \\ \cdot \\ \cdot \\ A(N)=A(N-1)*B(N)+C(N) \end{array}} \right\} \text{---④}$$

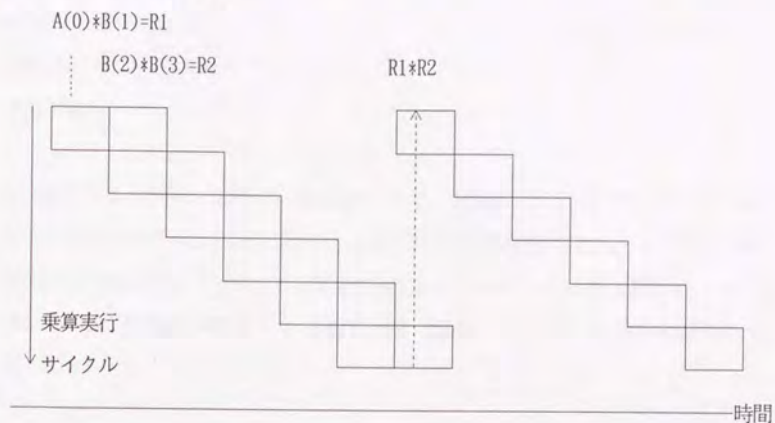
図4. 9のように $A(0)*B(1)*B(2)*B(3)$ を得るためには、連続したクロックタイムに $A(0)*B(1)$ と $B(2)*B(3)$ を演算し、それぞれの結果をさらに乗算器にループバックして乗算を行えばよい。シリアルに演算実行すると3回の乗算サイクルが必要であるが、この例では演算を分解して乗算を並列実行することにより、演算サイクルは2回分と等価になる。

さらに乗算器の空きクロックに $R2 * C(1)$, $R1*B(2)$, $C(1)*B(2)$, $C(2)*B(3)$ 等の乗算を実行し、各加算を乗算結果が出た時点で順次、加算器パイプラインに投入すれば、乗算サイクル2回と加算サイクル2回分の時間で $A(1)$, $A(2)$, $A(3)$ の結果を得ることができる。シリアルな演算実行では、乗算および加算の組を各3回の演算サイクル数の合計6回のサイクルの実行時間が必要である。したがって本方式では $6/4=3/2$ 倍の性能が達成される。さらに分解する項数を $A(4)$, $A(5)$, 等増やしていくと、上記 $l=\log_2 n$ の効果は上がるが、項の増加による演算数も増え、項の順序関係制御も複雑になるわりに、2倍を超える性能

は達成できず、有効でない。

以上のように、1次回帰的演算の性能向上は大きくはない。

図4. 9 連続乗算の実行



ただし、スカラキャッシュを持つベクトルプロセッサでは、意外な盲点があり、この演算をベクトル命令として持つことは意義がある。

もしこの命令をベクトル命令として持たない時、スカラ命令で1次回帰演算を実行しようとする、ベクトル演算を続けてきた結果をメモリにストアし、そのデータをメモリからキャッシュ経由でスカラ演算をすることになる。メモリにストアすることにより、キャッシュの該当データは無効となり、新データをメモリからロードしなければならない。このキャッシュペナルティの発生による性能低下が大きい。1次回帰演算ベクトル命令を持たない時、局所的な性能低下は、本命命令が存在する場合の3～4倍以上にも達していることが実測結果として明らかになった。従って、このキャッシュペナルティを避けるためにも1次回帰演算ベクトル命令をベクトル命令の1つとして、加えることにした。

(7) データ変換機能

最後にデータ変換命令を用意した。データ変換はFORTRANのオブジェクトとして頻繁に出現する。また、変換命令をサポートするハードウェアの増加分は小さいため、次のような、データ変換命令を設置することにした。

- ①浮動小数点データから整数へ
- ②整数から浮動小数点データへ
- ③倍精度から単精度浮動小数点データへ
- ④正のデータへ
- ⑤負のデータへ
- ⑥絶対値へ

以上のような、命令およびそれらの組合せにより、広範囲のDO LOOPのベクトル化が出来るようになった。従ってプログラム全体のベクトル化率が向上し、ベクトル処理の適用範囲が拡大する。さらにプログラマから見ると、コンパイラの持つ自動ベクトル化機能によって、容易に自動的にベクトル化が出来、性能も向上するため、大いに歓迎すべきことになる。

第5章 並列命令実行アーキテクチャ

本章では第3章で分析した問題点に応じて、命令の多重並列実行関連するアーキテクチャを提案する。データ転送能力を確保するためのベクトルレジスタの構成、命令の多重発信方式、データフロー解析にもとづく逐次実行制御方式、命令の並列実行にに適するアドレス変換方式、順不同命令実行に伴う割り込み方式について述べる。

5.1 ベクトルレジスタの構成と機能

(1) ベクトルレジスタの要件と構成

ベクトルレジスタはベクトルデータのベクトルプロセッサ内での一時保管領域である。つまり、演算パイプラインへのベクトルデータの供給源と結果の保管倉庫、メモリからのデータの保管倉庫となる。従って、できるだけ多くのデータを保留できるのが好ましい。

メモリへのデータの格納および演算器へのデータ供給はこのベクトルレジスタ倉庫の中からデータを取り出して、これをそれぞれの装置へ供給する。さらに演算器からの結果データを格納して保管しておかなければならない。従って、ベクトルレジスタはこれらの資源との間でデータ供給格納を行う資源として、ベクトルプロセッサの中で最も大きなデータ転送能力が必要な資源である。

第3章で述べたように第2世代アーキテクチャ(CRAY-1)では8個のレジスタしかなく並列命令実行数は小さい。従って、レジスタ指定数の拡張が必要である。

本研究では、命令の並列実行数を高めることができるアーキテクチャの構築を目的としており、最低でもロードストア命令を4命令、演算命令を4命令、合計8命令を同時稼働させることが必要であると判断した。そのためには局所的にはそれぞれを6命令ずつとマスク命令を2命令、合計14命令以上を並列実行できる必要がある。さらに命令の待合わせに対処するためのバッファリングも考慮して、20命令程度の同時実行を目標とした。

これを実現するためには、ロードストア命令では1つ、演算命令では3つのレジスタを1命令で指定するため、 $6 + 1 \times 8$ つまり24レジスタ以上のレジスタ指定が可能になるようにしたい。従って、ベクトルレジスタ数は最低32、データの倉庫としての機能を想定す

れば、64以上を指定できるようにすべきである。

ベクトルプロセッサでは、ベクトルレジスタのそれぞれが多数の要素をもち、そのオペランドを連続的に演算パイプラインに投入することで、性能を向上させた。従って1つのベクトルレジスタの容量はなるべく多くの要素数を確保すべきではある。ベクトル命令の立ち上がり時間の影響を軽減できるような要素数(64)を確保することが必要である。

64個のベクトルレジスタを仮定すると、それぞれが64要素を持ち、1要素のデータは64ビット(8バイト)とすれば、合計の容量は32KB(K:1024, B:バイト)の容量となる。さらに物理的なパイプラインを2/4組、多重設置する場合には、それに応じてデータ転送能力を確保するために多重並列にデータ転送することが必要であり、ベクトルレジスタの容量も2/4倍にすべきである。こういう理由により、実機ではベクトル性能比が1:2:4の3つのモデルに従って、ベクトルレジスタの容量を32/64/128KBとした。通常の論理素子ではこの容量は実現不可能であったが、幸い超高速RAMを開発し、使用することができたため、上記の大容量を持つベクトルレジスタを実現するアーキテクチャを作ることができた。

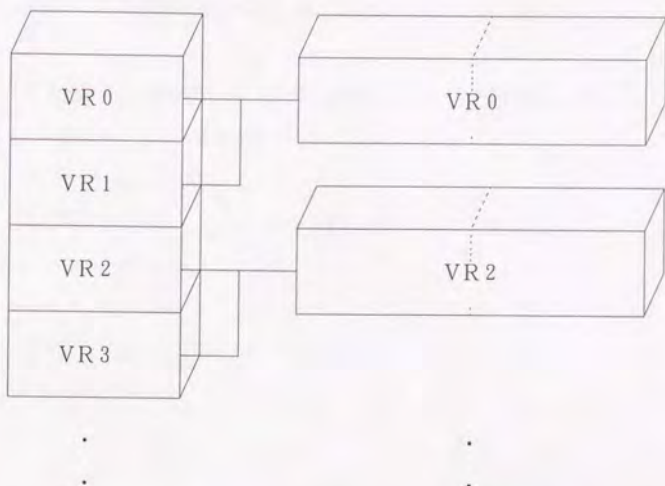
ベクトルレジスタの数の指定は上記で述べたように、64個は必要であるが、多くのプログラムを分析した結果、128個以上ではその数を増やす効果はあまり大きくないことが分かった。従って、命令の中でのベクトルレジスタ指定部は6ビットあれば、目標を達成できるが、本研究の命令体系では、命令指定部の区切りが4または8ビット単位であるため、ベクトルレジスタ指定部を8ビット即ち、256のベクトルレジスタの指定が可能とするアーキテクチャとした。こうすると、128KBの容量の場合、1ベクトルレジスタの要素数は64となり適当ではあるが、32KBの容量では要素数が16となり、小さすぎる。このため、レジスタの数を可変にする工夫を考えた。複数のベクトルレジスタをまとめて連結し、1つのベクトルレジスタとして指定できるようにした。図5.1に示すように、2の中乗の数のレジスタを連結し、それを1つのレジスタとして、再定義できるようにした。

図のように2つのベクトルレジスタを連結すると、ベクトルレジスタの番号VR0とVR1が併せて、新VR0となり、VR2とVR3が新VR2となる。

これを続けて、4、8、16、最大32個のレジスタを連結して、1つのレジスタにまと

めることができるようにした。従って5通りの指定が可能である。

図5. 1 ベクトルレジスタの連結



命令のレジスタ指定部は8ビットであり、その内の下位1ビットを0のみに指定し、上位7ビットで半分の数のレジスタを指定することが、2つのレジスタの連結指定となる。

図5. 2にベクトル長レジスタの内容とベクトル命令のベクトルレジスタ指定部との関係を示す。VP-200ではベクトルレジスタ容量が64KBであり、単一ベクトルレジスタは32の要素数を持つ。2個のベクトルレジスタを連結すると、64までのベクトル長を指定することが許される。ベクトル長が33から64を示していると、レジスタ指定部の最下位ビットが0でなければならない。ベクトル長が0から32までの場合は最下位ビットが0であっても一向に差し支えない。従って、ベクトル長は0から64まで指定できるわけで、しかも、ベクトル長が33から64までの時に、最下位ビットが0でないと、命令指定例外として命令実行を禁止させる仕様とした。

同様にベクトル長が65以上の時、命令のベクトルレジスタ番号指定部の最下位2ビットが00でないと命令指定例外とする。以下順に、513以上で1024以下の時は最下位5ビットが00000でないと命令指定例外とする。

このように、レジスタ長に大きな値を指定する場合は、その値以上でかつ最も小さな2の巾乗の数をベクトル長として、それに応じたベクトルレジスタ連結構成を指定する。これ

に対応してベクトル命令のベクトルレジスタ指定部の指定をすればよい。

図5. 2 命令のレジスタ指定部とベクトルレジスタ

ベクトルレジスタ指定部	レジスタ数	ベクトル長
(図4. 1のR1, R2, R3等)		
XXXXXXXX	256個指定	32以下
XXXXXXXX0	128個指定	64以下
.		
.		
.		
XXX00000	8個指定	1024以下

この時の違反検出条件は非常に簡単であり、命令のレジスタ指定部とベクトル長レジスタの内容チェックによって、容易に実現できる。

ベクトルレジスタの構成変更は、ベクトルレジスタ番号指定部とベクトル長レジスタの内容を変更することにより、コンパイル時にプログラム内で動的に変更できるように指定できることは自明である。

なお、このベクトルレジスタ変更機能は他に例を見ない本研究独自の機能である。

(2) ベクトルレジスタの構成変更の評価

一般的にベクトルレジスタの長さ指定が大きければ大きいほど、ベクトル命令の性能は向

上するため、上記のレジスタ連結で最大限のベクトル長指定をすることが有利ではある。その一方、最大のベクトル長指定時にはレジスタの数が少なくなるため、ベクトル命令の並列実行数が少なくなる恐れがある。変数と演算数が少ない時は、ベクトル長が大きいメリットを生かし、変数が多いときは、命令の並列実行数を上げるにより、性能を上げることができる。本アーキテクチャでは、各アプリケーションプログラムにより最適なベクトルレジスタの構成を選ぶことができ、さらにプログラム内で動的に構成変更できる。

次にこの効果を分析した結果を示す。

図5. 3にループ1及び図5. 4にループ2のサンプルプログラムを示す。これらの性能測定をベクトルレジスタの構成を変えて測定する。図5. 5がその結果である。

図5. 3 プログラム例 (ループ1)

```
DO 10 I=1, NX
      R (I, J) = (A (I, J) +B (I, J) *C (I, J) +
                  D (I, J) ) /6
10 CONTINUE
```

図5. 4 プログラム例 (ループ2)

```
DO 10 IX=2, NX-1
      DU0 (IX) =U (IX, J, K) -U (IX, J-1, K)
      DU1 (IX) =V (IX, J, K) -V (IX, J-1, K)
      DU2 (IX) =W (IX, J, K) -W (IX, J-1, K)
      DGR (IX) =U (IX, J, K) -U (IX, J, K-1)
      DRX (IX) =V (IX, J, K) -V (IX, J, K-1)
      MIU (IX) =W (IX, J, K) -W (IX, J, K-1)
      .
      .
      .
```

```

DRX (IX) =R (IX, J, K) -R (IX-1, J, K)
MIU (IX) =1. 0/R (IX, J, K)
DU0 (IX) =B11 (IX) *U (IX, J, K) +B12 (IX) . .
DU1 (IX) =RCP* (U (IX, J, K) *OUX (IX) + . . .
DU2 (IX) =B11 (IX) *OUX (IX) +B12 (IX) . . .
DGR (IX) = (RCP* (M (IX, J, K) -0. 5* . . . .
F11 (IX) =F11 (IX) -F11 (IX+1)
F12 (IX) =F12 (IX) -F12 (IX+1)
F13 (IX) =F13 (IX) -F13 (IX+1)
F22 (IX) =F22 (IX) -F22 (IX+1)
F23 (IX) =F23 (IX) -F23 (IX+1)
F33 (IX) =F33 (IX) -F33 (IX+1)
N11 (IX) =N11 (IX) -N11 (IX+1)
N21 (IX) =N21 (IX) -N21 (IX+1)
N31 (IX) =N31 (IX) -N31 (IX+1)
N12 (IX) =N12 (IX) -N12 (IX+1)
N22 (IX) =N22 (IX) -N22 (IX+1)
N32 (IX) =N32 (IX) -N32 (IX+1)
N13 (IX) =N13 (IX) -N13 (IX+1)
N23 (IX) =N23 (IX) -N23 (IX+1)
N33 (IX) =N33 (IX) -N33 (IX+1)
RB (IX, J, K) =R (IX, J, K) -OLTTX* (DUO (IX)
UB (IX, J, K) =U (IX, J, K) -OLTTX* (B11 (IX)

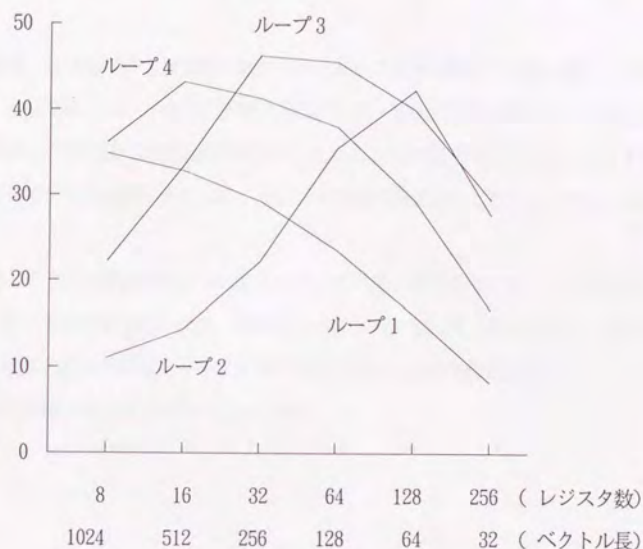
```

.
.
.

10 CONTINUE

ループ1のように単調に変化するものとループ2のように性能に山のあるものがある。
さらにループ3およびループ4のプログラム例は省略するが、それらの性能測定結果を示す。ともに極大値を示す例である。
最適なベクトルレジスタ構成を選ぶことが必要であり、コンパイラは構成選択の最適化を実行することが必要である。

図5. 5 ベクトルレジスタの構成と性能



VPシリーズのコンパイラでは、有効になってから使い終わるまでのベクトル変数の数とベクトル長によって、最適なベクトルレジスタ構成を選択している。

選択の方法はこの有効ベクトル変数の数と、ロード/ストア命令数と演算命令数によって決めている。ループ毎のこれらの値を表5. 1に示す。

表5. 1 サンプルプログラムの分析

プログラム例	有効ベクトル変数	ロード／ストア数	演算数
ループ1	9	5	4
2	7 6	9 4	2 4 3
3	3 2	2 0	8 1
4	1 0	1 0	1 0

一般的には有効ベクトル変数の数がベクトルレジスタの数以下になる最少の分割数の場合に、性能が最もよいとすることができる。VP-200の場合ではベクトル長が64を超えれば、この方法で分割を決めてよい。しかし、それ以下の場合にはベクトル長を長くしておいて、有効変数以下のベクトルレジスタ数の構成とした方が良い場合もある。

上記のように、本研究ではベクトルレジスタにRAMを使用することを前提とした、アーキテクチャの構築を行ったが、実際にハードウェアを実現する場合には、ベクトルレジスタの複数のバンク構成とそのアクセス方式に対する工夫が必要である。

その詳細は、第6章で述べることにする。

5. 2 命令の多重発信と並列実行

ここでは、指定がない限り、無制限に命令の先行並列実行を行うアーキテクチャを提案する。そのためには、逆にデータの従属関係にもとづいた逐次性が確保されていなければならない。従ってプログラムで陽に従属関係を指定することができる機能や逐次化のための方式を提案する。とくにメモリアクセスの多重並列実行にともなう、新規アーキテクチャが必要となった。その結果、アドレス変換と割り込みに対して工夫した方式を提案する。

(1) 命令の多重並列実行の必要性

プロセッサは演算器が多数あれば、命令を多重に発信することができる。

命令制御部が命令を一つずつ処理する場合でも、命令実行時間が複数クロック必要な場合は演算器が複数あることを前提に命令を多重に発信することができる。

同様に単一命令が複数のクロックで実行される場合でも、演算器がパイプライン制御を可能とする構成を取っていれば、同種の命令の多重発信が可能である。つまり、パイプライン制御により、該当演算器は時間的な並列実行が可能であるからである。この場合は、演算器がステージに分かれており、その各ステージがそれぞれ一組のオペランドを受容できるように構成されている(セグメンテーション)ため、その演算器に対して次々にオペランドを投入することができる。

ベクトルプロセッサは少なくともベクトル長に応じた多数のクロックでベクトル命令を実行することに加え、ベクトル演算器はパイプライン構成をとり、かつ複数のベクトル演算器を持つため、上記で述べた多重演算器による命令の並列実行およびパイプライン演算器による時間的な命令並列実行を両方実行でき、スカラ命令に比べてはるかに多くのベクトル命令同士の多重並列同時実行が可能となる。

従って、本研究ではスカラ比50倍を超える性能向上を図るため、ベクトルプロセッサの命令実行アーキテクチャは、物理的、空間的な命令多重処理を前提にして、実効命令処理数(実効性能)を大幅に向上させる命令発信を可能とし、前に述べたように命令発信数は20以上を目標とした。

本研究のアーキテクチャは、プログラムの実行論理を保つ制限が必要な部分を除いて、論

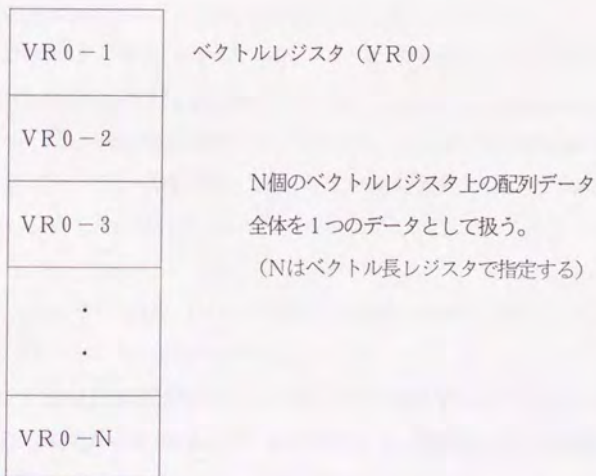
理的には無制限の数の命令をベクトル命令制御機構が並列発信する事が可能となるように構築した。

(2) データフロー解析と従属関係の検出

上記のように命令の先行発信と多重命令実行を前提とする場合は、プログラムで陽に実行論理を指定しなければならない。

プログラム実行論理(実行順序制御)は、データフロー解析によって、その順序を保証する制御が必要である。スカラ命令の実行では、個々のデータを対象にデータフローを解析し、それぞれのデータ同志の実行順序を規定する。ベクトル命令の実行論理では、データを個別データに分解せずに、ベクトル(配列)をひとかたまりとして、そのかたまり単位での、データフロー解析を行う(図5. 6)。このことは、1つのベクトルデータを1つのベクトル命令に対応させて扱うことができ、スカラデータと同様に命令単位にデータフローを管理することができるので、制御方式を容易にすることができる。

図5. 6 ベクトルデータ



従って、ベクトルデータ相互間のデータフローに応じて、ベクトル命令による命令の順序制御を行うことが必要である。

今回構築したアーキテクチャでは、レジスタ上のベクトルデータはプログラムアクセシブルなレジスタ（スカラレジスタ、ベクトルレジスタ及びマスクレジスタ）に置かれたベクトル長レジスタで指示される長さのデータを指定しており、そのベクトルデータ相互間のデータフローに従ってプログラム実行順序を規定していくことになる。このベクトルデータ全体はベクトルレジスタ番号によって代表される。従って、ベクトルデータ間で規定されたデータフローの順序関係はベクトル命令間で同一レジスタ番号の連鎖によって、指定することができる。

スカラデータはスカラ命令だけでなく、ベクトル命令でも指定することができる。スカラデータによるデータの連鎖はスカラレジスタ番号によって指定する。ベクトル命令でスカラレジスタをオペランドとして指定すると、同一データをベクトル長まで拡張したベクトルデータとして展開することがあり、この場合はベクトルデータとしてひとかたまりのデータをデータフロー解析の対象とする。

一方、メモリ上にあるベクトルデータもデータフロー制御ではベクトルレジスタ上の扱いと同様にひとかたまりのデータとして扱うことにした。

メモリ上でのベクトルデータはベクトルロード／ストア命令でメモリ上にベクトル長レジスタで指示される長さのひとかたまりのデータのことを言い、命令に従ってレジスタ（スカラ、ベクトル、マスクレジスタ）上へロードされ、またはレジスタからメモリ上へストアされる。データフロー解析には、このメモリ上のひとかたまりのベクトルデータを使用する。

図5. 7ではA (I) 同志、DO LOOPの内外でのSはデータフローとして従属関係にあり、プログラムでの指定が必要である。

2つのA (I) は従属関係にあり、2つのメモリアクセスするベクトルデータ変数は同一データフロー指定が必要である。ベクトル命令のフロー指定部に同一の指定値を指定する。本指定部では複数個のデータフローを指示することができる。

一方、同一のデータフロー以外のデータ間では実行順序を保証しないことにした。言い換えると、同一のフロー番号の指定がない異なるデータフロー間では後続する命令は無制限に命令発信が可能となることにした。さらに、ベクトル命令相互間では命令の同時並列実行を順不同 (OUT OF ORDER) で可能とした。従って、コンパイラはデータフ

図5. 7 メモリデータの順序関係

```
DO I=1, 99
  A(I)=B(I)*C(I)+D(I)
  D(I)=A(I)*E(I)+F(I)
  .
  S=D(I)+S
  .
  (A(I), Sはメモリ上のデータとする)
END DO
.
.
X=S+QX
.
.
```

ロー解析によって、一方では同一データフロー内での命令順序関係指定を行わなければならないのに加えて、他方では出来るだけ多くの他のデータフローを検出し、同時並列実行の可能性を追求しなければならない。つまり、コンパイラは、できるだけ多くのデータフローを管理し、それらのデータフローを同時多重に実行させる制御を行う必要がある。

メモリデータ上のスカラデータ同志では個別のデータ単位で逐次処理を前提としてデータフローに応じた命令の順序制御を行う。

メモリ上のスカラデータとベクトルデータとの間では、順序関係を規定するための、特別の命令を用意した。詳細は後で述べる。

このように、本研究のアーキテクチャはコンパイラの高度なデータフロー解析技術を前提として構築されている。コンパイラはデータ間での従属関係と独立関係の両立を図って、プログラム論理を維持しながら、できる限り多数の命令の多重発信を行うような、相反する要求を同時に実現するような高度な技術を必要とする。

以下の項では場合分けをして、命令実行順序の規定の詳細を述べる。

(3) レジスタデータに関する命令の逐次実行方式

本研究では命令のレジスタ（スカラ、ベクトル、マスクレジスタ）指定番号が、同一番号であることにより従属関係を指定し、ベクトル命令の実行順序を以下のように規定した。

命令間で指定されるレジスタの番号に同じものがなければ、それらの命令同士は独立に並列同時実行が可能である。レジスタ番号が一致しても、ともに読出オペランドとして使われていれば、レジスタの内容が変わるわけではないので、それらのレジスタ番号を指定している命令間では並列同時実行が可能である。

レジスタ番号が異なるか、読出オペランドとして使われている場合を依存関係がないといい、その場合は命令間での実行順序は追越しも自由である。両者の実行が完了していれば、結果は同一であるからである。

特定のある例外割り込みや、ハードウェア障害が発生した時には強制的な中断が生じる場合があるが、その時の状態は一定に決定できないことがある。例えば、命令が完了していても、順不同な命令制御が行われるため、割り込み時の状態は定義できない場合があり、別途割り込みの項で、これらの詳細を述べる。

一方、データの更新が生ずると更新前のデータと更新後のデータとの間での依存関係（従属関係）を規定する必要がある。コンパイラはコンパイル時にデータフロー解析を実行し、あらかじめこの従属関係を検出しておかねばならない。

ハードウェア的に表現すると従属関係とはレジスタやメモリのデータの書き込みが生じ、その同一データの読出しがその前後に生ずる場合である。

さらに細かく言うと、ハードウェアとしての命令間で、結果レジスタとしての指示がある命令と読出オペランドとして使われるレジスタ番号の一致する場合であり、それらの命令相互間には、命令の実行順序をプログラムで命令の出現順序どおりに行う必要がある。

読出しと書き込みの順序如何によらず、それぞれの命令の出現順序通りにデータの読出し、書き込みを行う必要がある。

従って、既に発信された命令よりも、プログラム上で後で発信されようとしている命令は必ず後で実行することにする。ある命令の発信タイミングでその命令よりもプログラム上で先行して発信された実行中の全命令のレジスタ番号のチェックを行い、もし一致する番号が検出されると、先行すべき命令が発信された後で該命令が発信されるように制御しなければならない。トリビアルな場合もあるが、同一レジスタへの書き込みが前後の命令に現れる場合でも、プログラムの実行順序に従って命令を発信することとする。

この順序関係はベクトル命令とスカラ命令間でも存在する。スカラデータがベクトル命令でも指定されることがあるためである。

特にベクトル命令の結果レジスタとスカラ命令の演算対象レジスタが一致する場合は、ベクトル命令の終了を待って、スカラ命令の実行が開始される。

ベクトルレジスタの構成変更を実行するときにはベクトル命令の逐次化が必要である。ベクトル長レジスタの内容を書き換える命令が実行されたときには、レジスタの構成変更が行われると言う前提での逐次化を実行する。すなわち、ベクトル長レジスタを変更した命令以前の命令は、ベクトル長を変える命令を実行する以前に終了させ、ベクトル長を変更する命令以降の命令はベクトル長を変える命令実行後に起動させる必要がある。

これをレジスタ構成変更時の逐次化と呼ぶ。

(4) メモリデータに関する逐次命令実行方式

メモリデータをアクセスする命令も順不同の命令実行を行うため、従属関係にあるデータ同志を逐次実行させるアーキテクチャを作った。

メモリ上のベクトルデータフローを指定して、命令実行順序を規定するために、ロードストア命令コードの中に、プログラムによって指定することができるデータフロー指定部を

作った。図5. 8に示すように、これをメモリアクセスID部と呼ぶ。

同一ID番号の指定のある命令間では、プログラムで現れたのと同じ順序でメモリデータを読み書きできるように命令を制御することにした。

ハードウェア資源との対応で言うと、順序動作の保証された論理的なメモリアクセスパイプラインを定義し、同期的逐次処理が必要なメモリアクセス同志には、同一のIDを指定して同一のパイプラインで命令を実行し、逐次処理して、データフローの順序関係を保証することにした。

一方、異なるデータフローに属する非同期並列処理が可能なメモリデータ同志には異なるIDを指定し、異なるパイプラインにおいてメモリアクセスが非同期並列同時実行できるようにした。そして複数のメモリアクセスパイプラインを設置できるように、ID部で複数(本研究では4ビット、16本)のパイプラインを指定できるようにした。

図5. 8 メモリアクセスID

OP	EP	ID	R2	R1	X2	B2	D2
----	----	----	----	----	----	----	----

同一IDを持つ命令群の中でも、結果レジスタR1部が異なればロード命令同志では幾らでも先行実行は可能である。ストア命令が入り込むと論理的にその前後の命令(他のストア命令も含む)は実行順序を保証する。つまり、プログラム上で先行する命令を実行できるタイミングのあとにストア命令を発信することとした。または、ストア命令を実行できるタイミングの後に、プログラム上で後続する命令を発信するアーキテクチャにした。

ハードウェアの実現方式としては、最後のロードメモリアクセスを発信するタイミングの直後から、次のストア命令のストアメモリアクセスの起動を開始するようにし、順序は保証しながらできるだけレイテンシーが小さくなるように工夫した。

単一のベクトル命令の中のメモリアクセス間でも、論理的に要素番号順にメモリアクセスを行うように、ハードウェア制御を行う必要がある。たとえば間接アクセス命令では、複数個の要素番号で同一アドレスをアクセスする可能性があるからである。このときでも最

後の要素番号のデータが結果のメモリアドレスにストアされているように制御する必要がある。

以上のように、逐次処理を実行するメモリアクセスパイプラインを複数個もつアーキテクチャとして、それぞれのロードストアパイプライン内ではシリアル性が保証され、相互間は何の規定もなく独立に命令が実行される。

この複数パイプラインの間では、それぞれのパイプライン間で同時並列命令実行が可能となり、コンパイラは依存関係の無いことが保証されるデータ間のみ、別々のパイプラインを割り当てる必要がある。論理的なメモリパイプラインと物理的なメモリパイプラインの割当はハードウェア自身が行う。従って、ハードウェアがうまく制御すれば、同一物理的パイプラインに複数の論理的パイプラインを割当てることもできる。

さらに特定のIDの値を指定すると（例えば0）その命令は、逐次同期制御からは解放される。ID=0は非同期先行実行を許すことを意味する。従って、コンパイラは依存関係がない限り、このIDを指定し、データフロー制御から解放させて、先行実行させることが出来る。ハードウェアは空いているパイプラインにその命令を割りつける。

（5）同期処理を行う逐次化命令

メモリアクセスの逐次化とは、すべての先行するメモリアクセスが終了していることを言う。プログラムでは陽には指定しないが、割り込みや、制御命令の後ではメモリアクセスの逐次化を実行することにした。

先行実行命令としてメモリアクセスするスカラ／ベクトル命令があると、全てのデータのメモリアクセス動作完了を待って、割り込みや制御命令動作が開始される。従って、割り込みや制御命令の実行時点ではメモリデータの順序関係は保証され、プログラム実行順序は保たれる。プログラマは割り込みや制御命令実行後は、命令実行と同期してデータ順序が正しいと解釈することができる。

上記のベクトル命令によるID指定のほか、プログラムで陽に逐次化を行うことができる機能をもうひとつ設置した。POST/WAIT命令がその機能を実現する。

POST命令とWAIT命令との組合わせて、プログラム上で意識した同期化を実行する

だけでなく、さらに命令の先行実行を意識的に行うことができる。

POST命令以前での命令実行とメモリアクセスが全て終了したあとWAIT命令が実行されるため、POST命令とWAIT命令の間でプログラムから見た逐次処理を行うことができる。そのときにただ単純に待つのではなく、待ち時間の無駄をできるだけ小さくするために、データフローと独立な命令群を命令列の中に挿入し、先行してそれらの命令を並列実行できるようにした。

図5. 9に示すように、POST命令②からWAIT命令④までの命令③は、逐次処理とは独立な命令であり、先行制御の対象となる。このタイミングでできるだけ同期関係のない命令を実行して、上記待ち時間を有効に利用することがPOST命令の意味である。

図5. 9 POST/WAIT命令

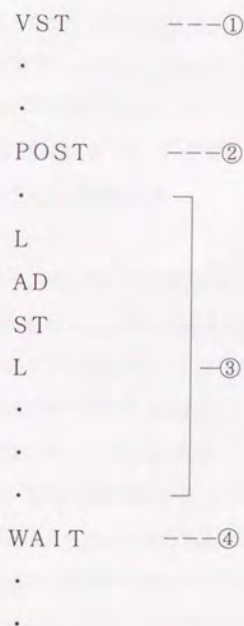


図5. 9の例ではVST命令①は、WAIT命令④を実行する以前に終了している。

従って、VST命令で格納したデータはWAIT命令を実行した後で、スカラ命令でメモ

りから読出して使用することができる。

なお、POST命令が無い時には、WAIT命令以前のすべての命令の終了を待って、WAIT命令を実行するため、WAIT命令は単独でも逐次化処理用に使うこともできる。

5. 3 命令書換えの禁止

本研究では、命令のメモリアクセスに対して次のような制限をおいた。

スカラおよびベクトルデータを後に実行する命令部分に書き込むことを禁止した。つまり命令は読出し専用とし、命令自身を書換えることはないことにした。

ただし、この禁止事項に違反した場合は、つまり命令を書き換えた場合は、ハードウェアの書き込み動作は実行されてしまうので、古い命令のまま実行されるか書換え後の命令が実行されるか分からないので、プログラムの動作は保証できないことにした。この場合、ハードウェアによるチェック機構を用意していない。

この制限事項は通常のフォンノイマンアーキテクチャには違反することになり、これは大きな制限になる。従って、この意味では、本研究アーキテクチャはフォンノイマン型の計算機ということは出来ない。

命令書換えの禁止をしたのは次のような理由からである。

ベクトル計算機でこの命令の書換えを許すと仮定すると、命令の先読み時に読み込んだ多くの命令のアドレス全てと、クロックサイクル中に同時並列的に実行されている多数のベクトル書き込みデータのアドレスとが一致しているかどうかをチェックするハードウェアが必要になる。もし一致していれば、その命令をキャンセルし、すでに実行されていた命令に対して、実行以前の状態に戻してやらなければならない。そしてもとに戻すことができれば、書き換えられた命令を再読出して、命令を再実行することになる。現実にはベクトルプロセッサではベクトル命令単独でも、再試行は相当困難であり、敢えてこの一連の作業を実行するとしても、そのためのハードウェアコストは莫大であり、命令の再読出しとすでに読み出していた命令をキャンセルする制御も殆ど不可能に近い。

一方、通常のプログラムでは、命令とデータは分離させることが可能である。コンパイラ

が意識して命令とデータに分離する作業を実施すれば、通常では起こりえないアドレス一致チェックのために無駄なハードウェアを浪費しないで済ませることができる。

従って、本ベクトルプロセッサアーキテクチャでは、フォンノイマン方式に固執せず、命令の書換えは禁止とした。

この結果、命令領域はプログラムのローディングを完了したあとは、読出し専用であり、かついくらでも時間的に過去の時点での命令を読出してもその命令は変更されていないことが保証されることになった。従って、ハードウェア的には命令の大幅な先読みが可能となった。

なお、命令書換え禁止は最近のRISCアーキテクチャでも同様に採用されるようになっている。

5. 4 アドレス変換方式

ベクトル命令では一度命令実行の起動がわかってしまうと、同一命令でも再実行をするとは、困難である。さらに、世界最高性能の単一プロセッサを達成するために、多重命令実行を実現したため、下記割り込みとの関係で命令の再実行はさらに困難となった。通常では常識化された動的アドレス変換もベクトルプロセッサでは実現することが出来ず、その代替機能として静的アドレス変換を実現する必要がある。

通常のスカラプロセッサでは、バーチャルアドレスによる動的アドレス変換が常識である。その場合にはページフォルト割り込み、つまりアクセスしたメモリアドレスがページ変換テーブルに登録されていないとき、そのことを通知する割り込みをして、その命令からもう一度プログラムを再開することができるように制御する。

本研究のベクトルプロセッサではその実現が困難である。その理由は命令の並列非同期実行を行うため、割り込みが起こった時の命令実行状態は一律でなく、先行して発信されている命令はタイミングによって、実行されていることもあり、実行されていないこともある。入出力割り込み等であらかじめ割り込み原因が発生したことにより、停止信号を発信し、ある程度の時間を経過してから割り込む場合は、再開可能な一律に定義できる状態で割り込めるが、ページフォルトでは、一律のプロセッサ状態を割り込み時に規定することができない。従って、ページフォルトを起こさないハードウェア実現方式が必要であり、その結果、バーチャルアドレス方式は適用できない。

このため、ベクトルプロセッサがアクセスできる範囲は実装されているメモリ容量以内とし、アドレス変換可能な範囲は実メモリ実装範囲内に限ることに決めた。一方、アドレス変換の高速化を達成するために、アドレス変換テーブルはレジスタによって保持することにした。アドレス変換レジスタのページエントリ数は、ハードウェア量としてそれほど多くを望めず、たとえば256エントリを実装することにした。ベクトルプログラムの多重処理数はそれほど多くする必要はないため、エントリ数は256の制限でも運用上問題はない。

一方、実装されているメモリ容量一杯をアクセス可能とするためには、ページサイズとし

では比較的大きな空間をカバーする必要がある。たとえば、256MB（バイト）の実装容量の機械では、エントリ数が256の時は、1MBのページサイズとしなければならない。ただし、スカラ用のアドレス方式は通常のバーチャルアドレス機能を持っており、ページサイズもたとえば4KBと小さな単位である。

上記のように、スカラ用のアドレス変換とベクトル用のそれが異なっているため、ベクトルプログラムではその一致をとる必要がある。ベクトル用の大きなページに相当するスカラ用の多くのページに対し、連続アドレスの変換テーブルを用意し、スカラアドレス変換とベクトルのそれをプログラムからみたとときに一致させる。そのような作業はOSのメモリ管理機能によって、これを実現している。

この結果、プログラム論理上ではページフォルト割り込みが起らないアーキテクチャを作ることができ、ベクトル命令の先行並列実行が可能となった。

5. 5 割り込みと命令並列実行

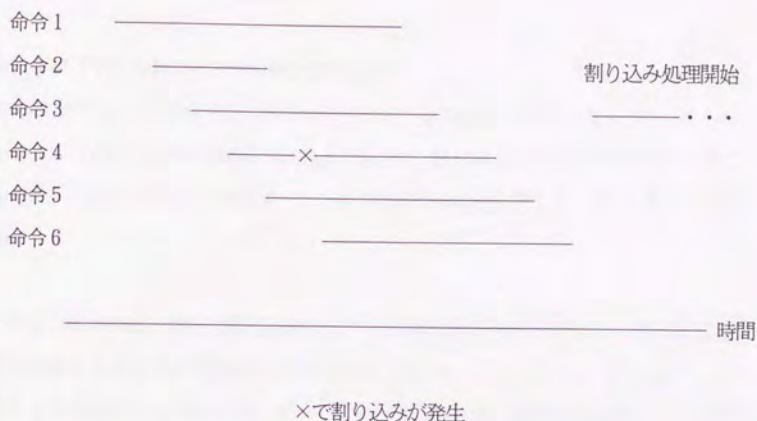
（1）割り込み原因の発生と命令の終了

割り込みの原因発生タイミングと割り込み動作の開始までには時間的なずれがある。特にベクトル命令では複数の命令が同時に実行される可能性が高く、さらに極端な先行制御を許すアーキテクチャであるため、論理的に後で実行される命令も既に実行を開始している場合や、先行して実行を終了している場合すらある。従って割り込み動作は逐次的アーキテクチャとは異なるものが必要である。

図5.10の例では命令4のあるインデックスで演算例外が発生し、割り込みを起こす必要がある。しかし命令は1から5まで、並列に実行中であり、すでに命令6もすでに命令の起動がわかっているとすれば、すべての命令の動作が完了した時点でハードウェアとしての割り込み動作を開始させる。ベクトルプロセッサでは、いったんベクトル命令を実行し始めると、ベクトル長まで一連の動作を連続してひとかたまりとして実行するため、命令の途中で停止・中断することは困難である。従って、すでに起動中の命令は全て終了さ

せたあと、例外の種類等の割り込み情報を格納して、割り込むことにした。

図5. 10 割り込み発生と実行中の命令



その場合、命令は一応完了してしまうので、演算例外の発生した命令4の該当インデックスで示される結果レジスタには、異常データを格納してしまうことになる。従って、この場合は、その異常データを後に読み出した時に、そのデータはやはり例外データとなり、その命令を実行すると、さらに例外になってしまう。

これを避けるために、その結果データが異常にならないように、例外条件に対応して異常にならないような意味のあるデータを格納する手段を作った。例えば、浮動小数点指数アンダーフローであれば、浮動小数点の絶対0を結果データとして格納するなどである。これらの指定は割り込み制御用の例外制御レジスタの内容で指定できるようにした。

割り込み状態情報として割り込みアドレスは、その例外が発生した命令以降の命令のアドレスを示すが、機械の先行制御状態により予測出来ない。

一方、プログラムデバッグのために、本研究では命令の並列実行禁止モードを指定でき、1命令ずつ逐次実行することもできる。このモードでは性能は犠牲にし、論理的に先行命

令実行を禁止する。従って、割り込み時の状態情報は一律に決まる。図5. 10で言えば、命令4で割り込みが起こったときには、命令5のアドレスを割り込み情報としてプログラムに示すことができる。この場合は例外発生命令と命令の実行状態は一律に定義される。

(2) 命令の中断と完了

命令が終了するときに、2つの場合を想定した。

1つは完了で、全ての命令実行を該当命令として全て終わらせることをいう。

もう1つは中断で、命令の途中で終わってしまい、残りは未実行のままで終わらせることを言う。上記のようにある特定データを仮に格納することも含めて、完了と言うことにする。

ベクトル命令では、現在、同時実行されている命令の全てを完了にして、実行中の最後の命令の次の命令アドレスを表示して割り込む。

割り込み時の割り込み情報が完了を示していれば、割り込み原因と該当命令、インデックス等を検索して、スカラ命令による再試行が成功する場合もないとは言えない。

しかし一般的にいうと、演算例外を含めて、例外が生じたときは、プログラム論理が保証されないので、プログラムの続行は行わない。何らかのロギングの情報を取って、プログラムは中断する場合が多い。

割り込み原因が多数同時に発生していれば、できればすべての割り込み原因を表示したいが、それには数多くの原因情報の格納が必要であり、ハードウェアの節約のため、最も優先順位の高い割り込みのみを表示して割り込む。同一優先順位のなかでは時間的に最も早く発生した原因を表示する。

勿論、全ての原因ごとにそれぞれの原因が1回でも発生すると、そのそれぞれを表示しても構わないが本実現案では1つの原因だけを表示するに止めた。本案では、割り込み発生時にはプログラムは中断し、データの再設定により、再開することとし、原因表示も1つで良いとした。

(3) 例外発生時のデータ保証

ベクトル命令では例外が発生しても、命令を中断せず、完了する。

例えば、データ例外の場合はそのデータが、被演算対象データが意味の無い形式である場合は演算を実行すると、演算結果は保証できない。

その場合は割り込み制御レジスタの指定に従って、制御レジスタの指定が0なら演算結果をそのまま書き込む場合と、同指定が1なら固定のデータを書き込む（例えば0）ことがある。以下にはそれぞれの例外条件と制御ビットがオンのときの固定データの関係性を記述する。さらに⑤、⑥では割り込み原因通知を抑止して、割り込みそのものを起こさないようにすることが出来る。

①データ例外

浮動小数点演算対象のオペランドで、ゼロまたは正規化データ以外のデータを認識すると、結果にゼロを書き込んで、命令を終了させる。

② 浮動小数点除算例外

浮動小数点除算において除数がゼロのときこの例外が発生する。

被除数がゼロのときはゼロ、正規化数のときは被除数の符号に最大の絶対値をいれて演算を実行する。

③ 固定小数点オーバーフロー例外（制御レジスタで制御可能）

固定小数点演算で最上位ビットからの繰り上げが生じたとき、正の数でオーバーフローしたときには正の最大値、負の数でオーバーフローしたときには負の最小値を入れる。

④ 指数オーバーフロー例外（制御レジスタで制御可能）

浮動小数点の演算で指数部がオーバーフローし、小数部がゼロでない場合にこの例外が検出され、符号はそのまま最大の絶対値をいれる。

⑤ 指数アンダフロー例外（制御レジスタで制御可能）

浮動小数点演算で指数部が0以下になり、小数部がゼロでないときに検出される。

結果にはゼロを入れる。

⑥ 有効数字例外（制御レジスタで制御可能）

浮動小数点演算で小数部がゼロで、指数部がゼロでないとき検出される。結果にはゼロが入る。このとき割り込み自体も無視される。制御ビットがゼロのときは本割り込み原因が検出されたデータがそのままの結果データとして入る。

（４） マスク処理と割り込みの優先順位

—スカラプログラムとの互換性の維持—

ある命令においてマスク機能が有効になると、マスクデータが0である個々のインデックスに対応し、割り込み原因発生そのものをマスクすることにした。

もともと、マスク機能の意味は、マスクデータが0のときはそれに対応するインデックスの演算を実行せず、結果のデータはもとのまま保持することである。演算を実行しないのであるから、例外も発生することはない。従って、例外を抑止するためには、例外原因を発生させないように制御するのが良い。

マスク機能において、演算を実行しないのと等価とする理由は、次の通りであり、スカラプログラミング（コンパイル）と密接な関係がある。

スカラプログラムにおいて、IF文の条件判定により、条件に合致しないため、プログラムのある部分が実行されないことがある。

ベクトルプログラムでは、IF文のベクトル化によって条件に合致しないときはマスクビットを0として、マスク機能付演算を実行するが、最終的なデータは書き込まず、元のデータのまま維持する。この状態では演算を実行しなかったのと同じ状態になる。さらに該当データに例外が発生するとスカラプログラムでは演算を実行しないのだから、例外も発生しないことになる。従ってベクトル命令ではマスク機能によって、実行しなかった状態と等価な状態をつくり出す必要があり、マスク機能によって例外条件の発生を抑止してしまわなければならない。ベクトルロード／ストア命令の場合も同様で、該当データに保護例外や領域侵害が発生しても、その原因の発生を抑えこんでしまう必要がある。

この機能は、スカラプログラミングとの互換性の維持として必要である。もしもスカラプ

プログラムにこの演算実行しない部分にプログラムミス（たとえば領域侵害）があっても発覚しないが、ベクトル演算実行時にマスクによる例外条件発生を許すと、上記のプログラムミスが発覚してしまう。潜在的にはバグがあっても、そのプログラムはバグを避けて実行されるため、正常なプログラムといわざるを得ない。プログラマの責任ではあるものの、システムとしては、同一プログラムには同一結果を保証すべきである。

同一結果を保証するためには、マスク機能は例外条件より、強い割り込み抑止機能を持って優先的に処理しなければならない。

（５） 多重割り込み条件の発生とその優先順位

本アーキテクチャではスカラー命令もベクトル命令も複数の命令が同時並列に実行されているため、同一命令の中でも、命令相互間でも複数の割り込み原因が生ずることがある。割り込み原因が発生すると、割り込み準備を開始し、できるだけ早い時点で割り込み動作に入る。その最終タイミングで複数の割り込み原因の中から、決められた優先順位に従って、割り込み原因を示すコードを表示する。同一優先順位の複数個の割り込み原因では時間的に最も早く報告されたものを表示する。

多重割り込み原因が発生する場合は、同一プログラムを別の時点で実行したときに、同一原因で割り込むとは限らない。

表5. 2に割り込み条件の優先順位を示し、表5. 3にプログラム例外とその優先順位を示す。優先順位は1が最も高く、数字が大きくなるに従って優先順位が下がる。同一数字は同一優先順位を示し、例外条件発生順に報告する。

表 5. 2 多重割り込み条件と優先順位

優先順位	割り込み条件
1	緊急マシンチェック割り込み
2	スーパバイザコール割り込み
3	プログラム割り込み
4	抑止可能なマシンチェック割り込み
5	外部割り込み
6	入出力割り込み
7	再起動割り込み

表 5. 3 ベクトル命令におけるプログラム例外とその優先順位

優先順位	プログラム割り込み条件
1	オペレーション例外
2	特権命令例外
3	指定例外
4	アドレス境界指定例外
5	記憶保護例外
6	アドレス指定例外
7	データ例外
8	浮動小数点除算例外
9	固定小数点オーバフロー例外
9	指数オーバフロー例外
9	指数アンダフロー例外
9	有効数字例外

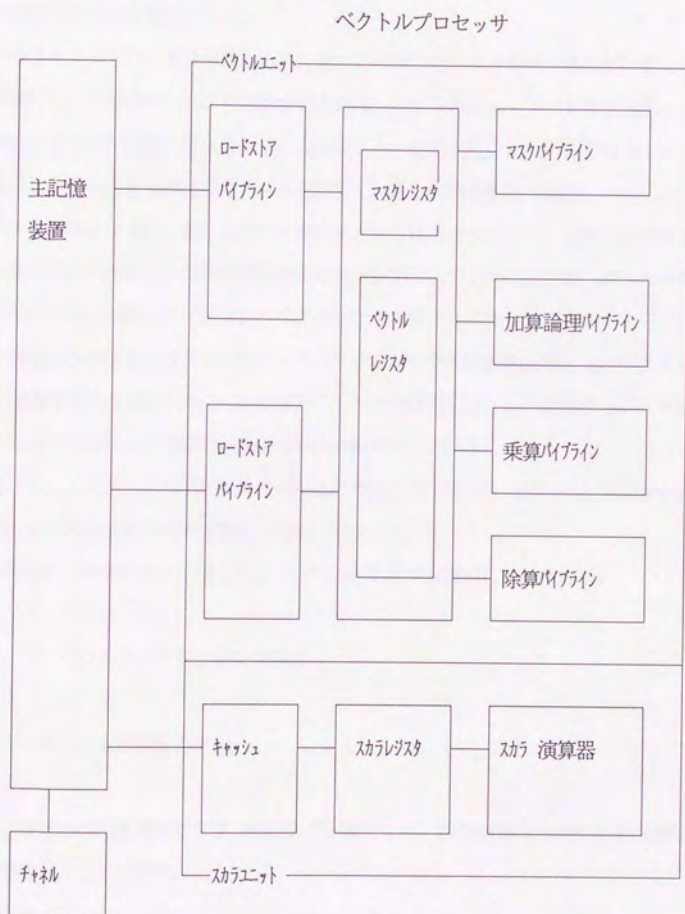
優先順位 9 は同一順位なので発生順に報告する。

第6章 ベクトル命令の制御方式とその実装

6.1 ベクトルプロセッサの構成

ベクトルプロセッサのシステム構成をFACOM VP-200のシステムを例にとって図6.1に示す。

図6.1 FACOM VP-200のシステム構成



VP-200は主記憶装置、チャンネル装置、ベクトルプロセッサからなる。ベクトルプロセッサはさらにスカラユニットとベクトルユニットに分割した。

スカラユニットはおもにスカラ演算の実行とともにシステム全体の制御を行う。

VP-200ではスカラ命令アーキテクチャはFACOM Mシリーズの仕様に準拠しており、ベクトルアーキテクチャを拡張した。

スカラユニットは命令の取り込み、スカラ命令の実行、ベクトル命令をベクトルユニットへの発信、終了処理、割り込み等の処理を実行する。さらにチャンネルに対して入出力命令の発信と割り込み処理を行う。

スカラユニットは、命令を読み出した後、その命令がスカラ命令であれば、自らの演算器で実行し、その命令がベクトル命令であれば、ベクトルユニットに命令を送出し、併せてスカラデータが必要な場合はベクトルユニットへ送出する。また入出力命令をチャンネルに発信し、チャンネルは独立にコマンドを解釈し、自ら命令を実行する。

ベクトルユニットは、あくまでプロセッサとしてはスカラユニットと同一の命令ストリームを実行するユニットの立場で命令を処理するが、スカラユニットはベクトル命令をベクトルユニットに渡ししまうと、ベクトル命令の終了とスカラレジスタへのデータの格納を管理するのみで、ベクトルユニットがベクトル命令の制御を管理する。ベクトルユニットは論理的にはスカラユニット配下での1つの命令ストリームを受理するが、ベクトル命令に関しては自らの制御で多重命令の同時並列実行を行う。

従って、ベクトルユニットはチャンネルほどの独立性は持たないが、ベクトル命令の実行に関しては大幅な独自制御権を持つように構成した。

本研究のアーキテクチャはこのようなハードウェアを想定して構築した。

6. 2 ベクトルパイプラインの構成

(1) 命令の出現頻度の測定

第3章で述べた応用プログラムの調査の一環として、四則演算の出現確率を別途計算した結果を表6. 1に示す。

表6. 1 演算の出現確率

演算	出現確率
加算	24%
減算	14%
乗算	34%
除算	6%
比較演算	7%
マクロ演算	7%
関数	8%

加算と減算はともに加算用ベクトルパイプライン演算器で実行できる。これらの命令出現確率を合計すると38%であり、乗算の34%とほぼ同等である。従って、ベクトル加算器とベクトル乗算器はそれぞれ1:1の割合で装備するのが適当である。除算命令の実行頻度は加算乗算の1/6であることが分かった。

上記の調査と同時に、演算に必要なメモリアクセス量の分析もした。結果はすでに表3. 1に示されている。この数値を分析の結果、加算パイプライン、乗算パイプラインに対応して同等性能のロードパイプラインが必要であり、さらにその半分の性能のストアパイプラインが必要ことが分かった。つまり、複数のメモリアクセスパイプラインが必要である。

(2) ベクトル演算パイプラインの構成

上記の分析に基づき、主な演算パイプラインとして加算、乗算、除算、ロード、ストア用を設置した。さらに第4章で述べた条件ベクトル処理を主目的としたマスク演算用のマスクパイプラインを用意した。

ベクトル除算器は、加算と同等の性能を満足する回路の実現には大量の回路量が必要である。概算では5~7倍の回路量が見積もられた。しかし除算命令の出現確率も低いいため、多少物量投入を加減できる。従って性能が7分の1になる回路量に限定した。

その結果コンパイラとしては逆数の乗算をできるだけ使用し、出来るだけ除算の出現を抑えることにした。

加算、乗算等のそれぞれの主演算はそれぞれの演算パイプラインで実行するが、上記の分析の中で現れた他の演算の実行形態を次に述べる。

比較演算は加算器による減算とマスクパイプラインの組合せで実現し、マクロ演算はやはり加算パイプラインや、乗算パイプライン、マスクパイプラインとの組合せで実現した。

関数はソフトウェアとして加算、乗算、論理演算等の組合せで実現した。

演算マスク機能は、ベクトル演算命令の実行時に合わせて、マスクレジスタを読み出し、そのマスクデータを演算パイプラインに供給して、実現した。

次にメモリアクセスパイプラインの構成方式について述べる。ベクトルユニットの性能を確保する最重要ポイントは複数のメモリアクセスパイプラインである。従って、2本のメモリアクセスパイプラインを設置することにした。ただし、そのパイプラインを単純にはロード用とストア用にしなかった。その理由は、通常のFORTRAN文では、①のような単純な演算で全ての変数がメモリ上にあれば、B、Cオペランドはメモリから持ってくる必要があることや、DO LOOPに含まれる演算数が多い場合でも、演算開始の初期にはロード対象オペランドを先行して取り込んでおくことが有利なため、ロード命令を複数個同時実行できるように構成したいからである。従って、2本のメモリアクセスパイプラインはともにロード実行することができるようにし、対称的な動作を可能とすべく、ストアも実行できるよう構成した。ロード命令の複数同時実行により、演算全体の実行開始時の立ち上がり時間を削減することができる。

$$A(I) = B(I) + C(I) \quad \text{---①}$$

以上のような各演算の出現割合などの考察に基づき、図6. 1に示すように、ロードストアパイプラインを2本、加算、乗算パイプラインを各1本、マスクパイプラインを1本、除算パイプラインを1本設置することにした。但し、除算パイプラインの性能は他のものより低くすることとした。

6. 3 ベクトル命令のパイプライン制御

以上のように、ベクトル演算パイプラインの構成を決めたので、それらのパイプラインを効率良く動作させる制御方式を構築する。第5章で述べたように、ロードストア命令、演算命令がそれぞれ2パイプラインに2命令ずつ、計8命令を並列実行させ、局所的には2パイプラインに3命令ずつ、さらにマスク命令を2命令合計14命令を並列実行させ、バッファリングも合わせて20命令同時制御を目標とした。その目標を達成するため以下のような制御回路を構成することにした。

スカラユニットは図6. 2に示すようにスカラ命令を命令制御パイプラインのIからSまでの5段階のパイプラインで処理する。即ち、命令フェッチI、デコード及びアドレス計算D、データフェッチF、演算実行E、及び結果の格納Sの五段階である。

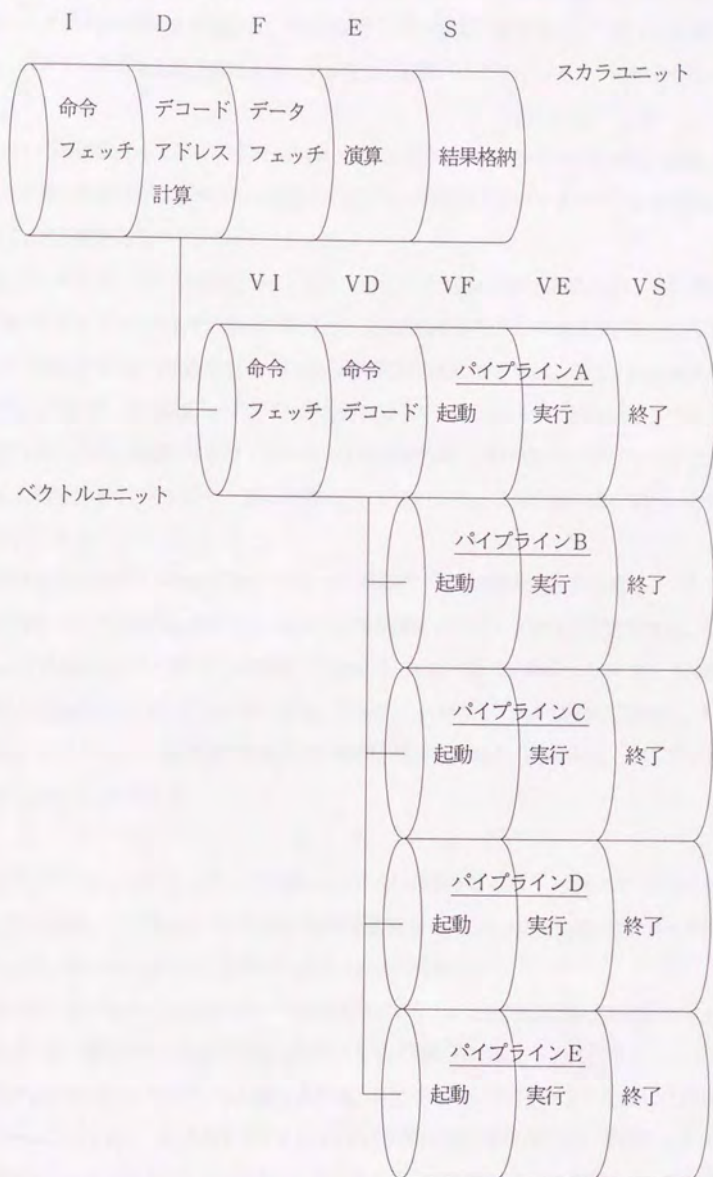
スカラユニットはベクトル命令も同一制御パイプラインで実行するが、途中からベクトル命令情報をベクトルユニット側に送出し、主な制御はベクトルユニット側に委託する。スカラユニットはベクトル命令をデコードすると、ベクトル命令制御情報と必要ならスカラデータを組みにして、ベクトルユニットに送出する。

図6. 2にはスカラユニットの命令制御パイプラインとそれに連結したベクトル命令の命令制御パイプラインを示す。さらにスカラユニットとベクトルユニット間のスカラデータの受渡しの概念図を図6. 3に示す。

ベクトル命令ではスカラパイプラインの主にIとD、すなわち命令フェッチおよび命令デコードフェイズで命令解釈とベクトルユニットへの起動がかけられる。スカラデータが必要な時は、デコードフェイズでデータフェッチが起動され、そのデータはベクトルユニットへ送信される。

一方、ベクトル命令制御ユニットはスカラユニットから送信されたベクトル命令を命令フェッチフェイズVIで受取り命令バッファに蓄積する。また、スカラデータをVIフェイズで受信し、ベクトル制御ユニットのスカラデータバッファに蓄える。このデータはVDフェイズで読み出され、VFフェイズで演算パイプライン側に送出される。デコードフェイズVDで命令解釈ならびにベクトル命令で使用するベクトルレジスタ(VR)、及びマスキングレジスタ(MR)のレジスタコンフリクトチェック(競合解析)を行う。つまりベクトル命令制御パイプラインAからEまでのあらゆる実行段階のVRおよびMRが依存関係にあるかどうかのチェックを行う。もし依存関係がなく、VDフェイズの制御機構は該当

図6. 2 命令制御パイプライン



パイプラインが空になる事を検知したら、デコードされた待機中のベクトル命令を取り出し、命令発信状態にあればベクトル命令を該当制御パイプラインに投入する。同時に演算パイプラインの動作が起動され、ベクトルデータが演算パイプラインで次々と処理される。ベクトル命令を新規発信しようとする5つの制御パイプラインの1つに命令を投入する。

この5つの制御パイプラインは2つのロードストア用のパイプラインA及びB、加算/乗算/除算の演算用のうちの2つの演算パイプラインを制御するパイプラインC及びD、およびマスク演算用のパイプラインEがある。

VDフェイズで、これらの制御パイプラインに1つずつ命令が投入されるが、それぞれの制御パイプラインはそれぞれ独自に動作し、全体の命令制御としては順序不同 (OUT OF ORDER) で制御される。順序不同制御を可能にするには、それぞれの演算パイプラインに対応した制御パイプラインが必要になるため上記の5つの制御パイプラインを設置した。各命令制御パイプラインに投入された命令は、それぞれのパイプラインで定義された制御フェイズを経由し、最終段階のパイプラインフェイズを通り過すと、命令実行が終了する。

VFフェイズはベクトル命令のオペランドの先頭のデータの読み込みを起動し、ベクトル長の最終データの読み込みまでのタイミングを制御するステートマシンを管理する。VEフェイズは命令がパイプライン演算器で実行されている状態を制御し、VFフェイズの起動により起動され、終了により終了する。VSフェイズではデータの格納を制御し、VE起動からパイプライン演算器の立ち上がり時間を経由したあと、起動され、最終データが書き込まれると終了する。

スカラーデータは、ベクトルデータの個々のデータに作用させるもう一方のオペランドとして使用される。この場合はベクトル命令制御回路は読み出された同一のスカラーデータをベクトル長の長さまで繰り返し演算パイプラインに送出する。

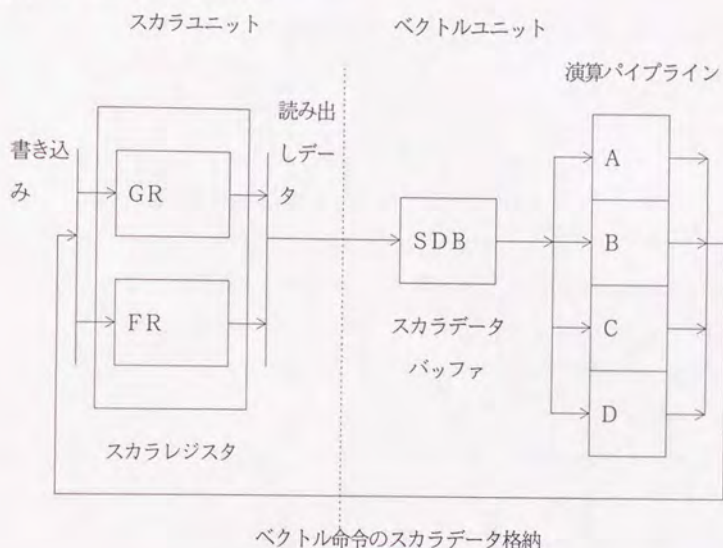
スカラーデータの書き込みは結果データを演算パイプラインが出力した後、スカラーユニットへ送出する。結果データを送信するとVSフェイズは終了する。

VSフェイズでのスカラーデータの書き込みは、スカラーユニットにデータを受渡したのち、スカラーユニット側からのスカラーデータ書き込み受け付け信号を受信して、そのフェイズを終了する。もしもスカラーユニット側で、ベクトル命令での書き込みスカラーレジスタ番号と

同一のスカラレジスタ番号を後続のスカラ命令がオペランドとして使う場合、スカラユニットはデコードフェイズで検出したスカラレジスタのぶつかり合いを認識しており、スカラレジスタへのベクトル命令からの書き込みを待つ。後続の該当命令は該当スカラレジスタへの書き込みが終了するまで待ったあと、実行される。

このような命令の順序制御により、スカラレジスタデータの順序性は保証されている。

図 6. 3 スカラデータの送受信



6. 4 ステージング処理とパイプライン制御

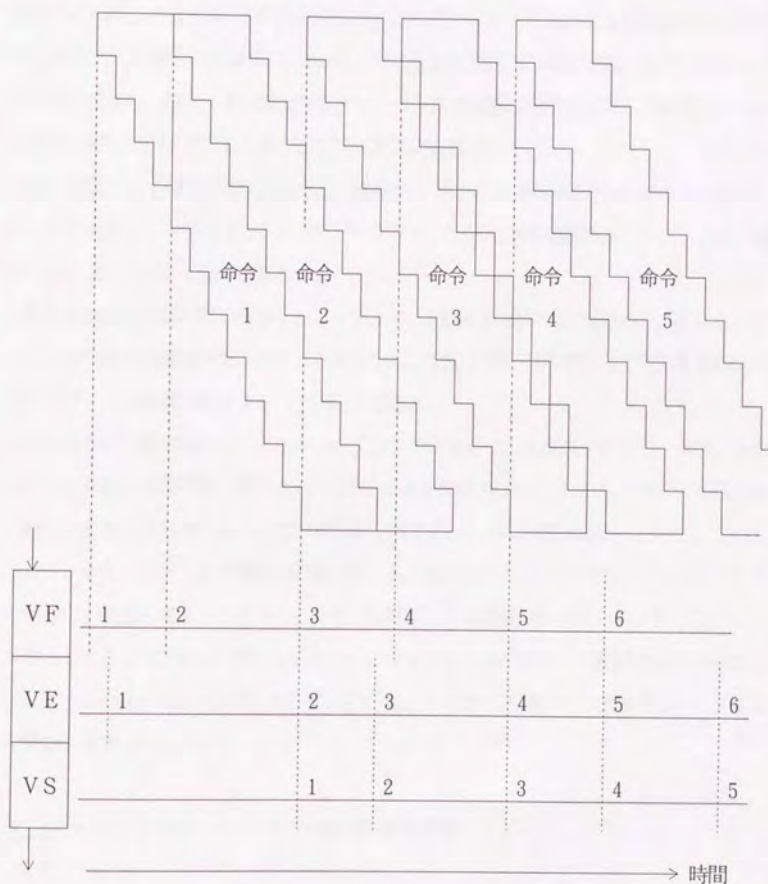
ベクトル演算器はセグメンテーションとして細かいステージに分割され、それぞれのステージは毎サイクルごとに次のデータを受入れられる構造になっている。

本実現案では1つの演算パイプラインを管理する制御パイプラインを3ステージとした。その理由を図6. 4によって説明する。

ここでは、演算パイプラインのステージが10段から構成されるとする。これを3段の制御パイプラインステージで管理する。非常にベクトル長が長い場合は立ち上がり時間中の前後の命令を制御するだけなので、2命令の情報を3段のステージで管理することになり

問題はない。しかしベクトル長が小さい場合は、演算パイプラインの中を多数の命令が通過することが必要である。ベクトル長が16程度の時に効率良く処理できればよく、逆に多すぎるパイプライン制御ステージは必要ない。命令1及び命令2はベクトル長が16と仮定すると、1サイクルで4個のインデックスを処理するので、最低4サイクルの間、制御パイプラインのVF、VE、VSの各フェイズに留まる。

図6. 4 演算パイプラインと制御段数



VDフェイズで待ち状態にある命令はVFが空いているか次のサイクルで空くことが分か

っている場合にVFに投入する。同様にVEが空いているか次のサイクルで空くことが分かっている時にVEに投入する。VSへの投入も同様である。

命令1はVF、VEに連続して投入されたあと、命令の立ち上がり時間を經由してVSに投入される。従ってVEへの命令2の投入およびVFへの命令3の投入はその同一タイミングまで待たされる。以下同様に下に位置するフェイズが空けば、上に位置するフェイズに命令が投入できるので、同時に2ないし3フェイズの命令が入れ代わることがある。この例でみれば、パイプラインの使用率はベクトル長が16の場合に75%程度になり、ベクトル長が20を超えればほぼ100%となるため投資目標を満足する。もしVEフェイズを省略すると、図6.4の場合の命令シーケンスで演算器使用率は35%程度となってしまう、使用率が低すぎるため、VEフェイズは必須である。

逆に、制御フェイズ数をあげればさらに演算パイプラインの使用効率を上げることができる。VP-2000ではロードストアパイプラインの使用効率を高めるために、4段の管理フェイズを設置して効率を高めた。

これだけの命令制御回路を投資したことにより、局所的に使用効率を高めることによって、平均的な命令処理効率を高めることができた。上記3段のパイプライン制御を各物理的パイプライン演算器に配置することが必要である。

ベクトル長の大きな場合はベクトルロードストア命令2つ、演算命令が2つ、マスク命令を1つを有効に使用率高く動作させることは比較的容易である。しかし、ベクトル長が短い場合に命令の並列実行性としての有効度を実測することは意義がある。しかし、このようなマイクロなハードウェア制御の評価には、通常動作中のハードウェアからモニタリングの信号を引き出してマイクロなタイミングで起動停止する機能が必要である。残念ながら、VP-200ではプローブするためのハードウェアが十分でなく、評価は出来なかった。プログラム全体の総合的評価を第7章に述べる。その後、開発された機械ではモニタリング機能が考慮されている。

6.5 ベクトル演算パイプラインの物理的多重構成

ベクトルアーキテクチャの一つの重要なメリットは、ベクトルパイプラインを物理的に多重配置し、その性能をその多重度まで高めることができることである。VP-200では

1 システムサイクルに4多重配置したのと等価な性能に向上させた。実際の実装としては、ハーフサイクルで動作するベクトル演算パイプラインを2個配置したため、システムサイクルでは4個の物理的パイプライン配置と等価である。従ってシステムクロックサイクルごとに4個のベクトルインデックスに相当するデータがベクトル演算パイプラインで処理される。

つまり、例えば1、2、3、4のインデックス対応の2組のデータが演算パイプラインの入力にそれぞれ投入され、同時に1組の4つのデータが出力される。さらに2つのロードストア命令および2つの演算命令が常時実行できるだけのデータ転送能力が必要である。この演算性能を満足させるためにはレジスタのデータの供給、格納も同じ転送能力が必要があり、ベクトルレジスタ及びマスクレジスタは1システムサイクルの間に、ロードストア用として2組の4個のデータを読み出すまたは書き込むことができ、演算用としてそれぞれ4個のオペランドを同時に4組読み出し、かつ2組の4個のデータを格納しなければならぬ。

6. 6 ベクトルレジスタの構成

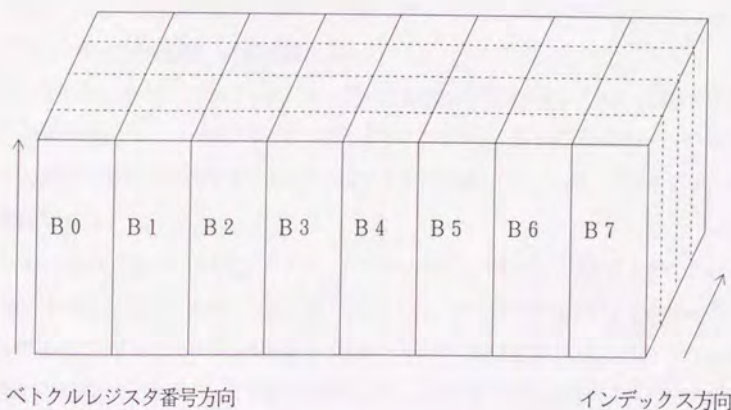
上記のようにベクトル演算パイプラインを物理的に多重化すると、同一性能を出すためにベクトルレジスタのデータ転送性能も強化する必要がある。

一方、ベクトルレジスタの容量は大きい程良いが、経験的には64個のレジスタ数で個々のレジスタはそれぞれ64要素数以上、すなわち全体容量として32KB（キロバイト）は必要である。VP200では256個のレジスタでそれぞれ32要素つまり64KBの容量を実現したが、ECLテクノロジーでこの大容量を満足するための素子は高速RAMが適当であり、かつロジックとRAMが同一LSIチップ上に混在できるテクノロジー（ロジックインRAM）を開発した。

上記のように、ベクトルレジスタに対するデータ転送能力の要求は、1システムクロック当たり、4個のデータを同時に動作させるオペランドを、2つの演算用パイプラインに対し、3オペランドを2組同時動作、2つのメモリアクセス用に1オペランドを2組同時動作させることである。従って、合計 $8 \times 4 = 32$ 個のデータを読み書きする能力を供給しなければならない。

ロジックインRAM方式を採用したことにより、RAMは読み書きの動作が1システムク

図6. 5 ベクトルレジスタのバンク構成



ロックサイクルで実現できた。従って、ベクトルレジスタは8 Bのデータ幅で32インターリーブの構成が必要である。

図6. 5に示すようにB0からB7までの各バンクは、それぞれ4つの8 Bのデータを保持しており、インデックス順に4個のデータを同時に読み出しまたは書き込むことができるように構成した。B0からB7までを逐次スキャンすると合計32個の要素をアクセスすることができる。

一方、バンク内でのオフセットアドレス方向はベクトルレジスタの番号を示す。例えば、256個のベクトルレジスタ構成の場合は、1個のベクトルレジスタは32個の要素から構成されるため、B0からB7までで1つのベクトルレジスタを構成する。従ってベクトルレジスタ番号方向で0から255までの各相対アドレスがそれぞれ1つのベクトルレジスタを示す。ベクトル長を32以下に指定したときは、1つのベクトルレジスタのアクセスはB0からB7までの1回以内のスキャンで終了する。

ベクトルレジスタの個数を128個、64個等に指定する構成ではそれぞれ2回、4回のサイクリックなスキャンまで続くことになる。

ベクトル命令のベクトル演算パイプラインの1つのオペランドとして、データを読み出すときには、インデックス順にB0の1、2、3、4の要素を同時に読み出し、次のサイクルでB1の5、6、7、8の要素を読み出し、以降B2、B3、・・・B7と読み出し、B0に戻るが、この時は33、34、35、36に相当する要素を読み出す。このことはインデックスが32だけ加算されていることを意味する。以降、B1の37から40までを読み、以後これを繰り返す。各バンクで読んだ4つのデータは同時に演算ベクトルパイプラインの4つの物理的入力に供給する。

ベクトルレジスタのアクセスはタイミングが決まれば最初の起動をB0から起動すればよく、連続的にクロックサイクルごとにB7までアクセスし、B0に戻る時にバンクのアドレスを相対的に1つ加算する。このようなアクセス制御をベクトル長までサイクリックに継続させる。

もう一つのオペランドは上記よりクロックサイクルを1つずらして先行するオペランドを追いつけるようにアクセスさせる。先行するオペランドを1クロックサイクルバッファリングさせて、2つのオペランドが揃うと演算パイプラインはスタートし、ベクトルレジスタから供給されるオペランドを次々と演算する。立ち上がり時間が経過すると結果データが演算パイプラインの出力に現れるので、これを別のタイミングでやはりB0から順次各

バンクに書き込んで行く。

上記の読み出し、または書き込みはベクトル長まで達すると完了する。最終アクセスバンクではベクトル長を超えた要素は無効データが処理されるため、結果的には書き込みも実行されない。つまり、マスク機能により演算が無効になると等価な制御を行う。

6. 7 ベクトル命令の発信タイミング

VDフェイズに命令起動する条件が揃っていることと(6. 3に記述)、さらにベクトルレジスタへのアクセス起動条件が満足された時に、ベクトル命令が演算パイプラインへ発信される。ベクトルレジスタの連続アクセス用の起動タイミングを規定するためには、バンクB0のアクセスが可能かどうかの検査が必要である。

ベクトル演算命令では3つのオペランドアクセスに伴い、3つのベクトルレジスタアクセス起動タイミングを専有することになる。もしもベクトル命令の発信を任意のタイミングで行うと、任意のタイミングでバンクアクセスが起動され、後続する命令の起動タイミングを競合しないように上手く選択しなければならない。タイミング割当が適当でないと、使用していないバンクがあっても競合して、うまくアクセス出来ない場合もあり、結果的には、ベクトルレジスタの使用効率を下げる危険性もある。

このようなことが生じないようにするためには、命令発信のための論理を単純化して特定のタイミングのみに命令発信をするように規定すると、発信タイミングは1~3クロックずれる場合もあるが、バンク使用効率を高めることが出来る。ここでは、この起動タイミングをバンクスロットと名前をつけた。バンクスロットとはある時点からスタートする論理的なタイミングニックネームである。一度起動されるとある条件が発生しない限りサイクリックに巡回し続け、8クロックごとに同一バンクスロットに戻る。

図式化すると図6. 6に示すようにバンクスロット決定用のインディケータがあって、これがクロックごとに回転し、現状のバンクスロットが分かる。この表示に従ってベクトル命令の起動タイミングが決定される。

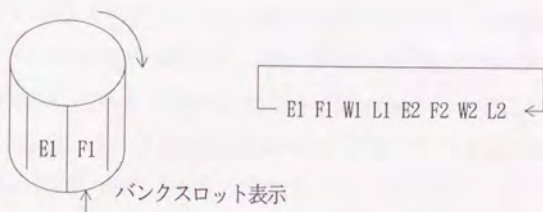
今回の実装ではこのバンクスロットにE1 F1 W1 L1 E2 F2 W2 L2 と言うニックネームを付けた。E1/F1 またはE2/F2 は2つの演算命令のオペランド読み出しのタイミングであり、W1/W2 は結果の書き込みタイミング、L1/L2 はメモリアクセス用のベクトルレジスタの読み出し、書き込みタイミングである。従って、ベクトル演算命令はE1またはE2から起動す

るようにし、同時にF1, W1またはF2, W2も専有してしまうように制御する。

マスクレジスタもベクトルレジスタのバンクスロットと同期しており、マスク機能が有効なときは、マスクレジスタの読みだしをE1またはE2から起動する。メモリアクセスベクトル命令はL1またはL2から起動する。マスク命令はE1またはE2より起動する。

この結果、ベクトル演算命令同志、メモリアクセス命令同志は4サイクルに1回の割合で

図6. 6 バンクスロット



しか発信できないが、演算とメモリアクセス命令は交互に連続発信が可能である。

以上のように、命令発信のための論理を簡略化し、命令実行の効率を上げるためにバンクスロット方式が寄与している。

上記のように、バンクスロットによって命令発信が規定されるが、ベクトル長が長い場合は、次に命令が用意できていても先行する命令のオペランドアクセスが終了していないため、命令発信は待たされる。さらにその間はベクトル演算器は連続的に使用される。

ベクトル長が小さい場合は、4クロックサイクルごとに巡回してくるバンクスロットへ次々に命令を投入することができる。

6. 8 メモリアクセスと状態制御

メモリアクセスを伴うベクトル命令の実行時には、メモリアクセス競合が生ずるため、必ずしも一定の時間でベクトル命令が終了しない。先に起動されたメモリアクセスが終了するまで、同一メモリバンクへのアクセスを待たなければならない。つまりベクトル命令実行中に待ち状態が生ずる。この待ち時間の間、メモリアクセス命令のみを待たし、他の命令は終了させることが可能であるが、本実現方式では異なる方式を採用した。

その理由は次のとおりである。上記のバンクスロット方式を採用したため、Lフェイズのタイミングがメモリ待ちによって、擾乱を受ける。従って、メモリ待ちが生ずることにLフェイズへの再同期が必要になる。同期化のためには、最大7サイクルのバッファリングが必要である。こうすると待ち時間が1サイクルでもタイミングによっては大きなサイクルのずれを吸収しなければならない、最終的にはメモリアクセス命令の実行終了がメモリ待ち時間よりさらに大きく遅れることになる。

そこで、演算命令には性能面で多少不利ではあるが、バンクスロットタイミング制御とそれに準拠する命令発信論理には全く影響を与えない方式を採用することにした。

本方式ではメモリ待ち時間に相当する時間のあいだ、ベクトル命令発信回路およびベクトル演算パイプライン全体をクロック停止状態におき、命令制御状態を凍結して、状態遷移を禁止してしまう。その結果上記バンクスロットも凍結状態になり、命令発信論理は凍結状態のまま存続する。この方式を採用すると、停止再開処理はメモリアクセス待ちによる部分にのみ限定でき、メモリロード/ストアに関する多少のデータバッファリングのみを実現することによって、制御論理が簡単化され、設計の簡易化が実現される。

性能面での分析をすると、演算命令の実行ができる時間を停止させてしまう分のペナルティは生ずる。このペナルティは、演算命令実行時間よりもメモリアクセス命令の実行時間が少ない時に陽に影響が出るが、メモリアクセス命令実行時間の方が長い時には、演算命令は影に隠れて影響しない。さらにメモリロード命令のベクトルレジスタの内容は後続する演算命令のオペランドとして使われる場合も多く（次項の連結実行を参照）、その場合はロード命令リミットで演算命令も動くため、ペナルティは生じない。

一方、上記で述べたように、バンクスロットを停止しないで、演算命令を優先して実行させる方式では、かえってメモリアクセス回路で1/12のタイミングまでロードしてきたデータをバッファリングしなければならないため、その待ち時間が、後続する連結動作をお

こなう演算命令まで巻き込んで待ち時間の影響を与える可能性もあり、性能面でのペナルティがかわって生ずる危険もある。

従って、本ハードウェア実装においては、性能ペナルティはメモリアクセス待ちに対応してバンクスロット停止の方式が有利と判断し、さらに制御の簡単化もあわせて実現可能な本方式を採用した。

6. 9 ベクトル命令の連結実行

従属データを持つベクトル命令間の並列実行を命令の連結実行と呼ぶ。連結実行を行うためには、連結して動作する命令相互間での追越しがないことが命令実行の条件となる。

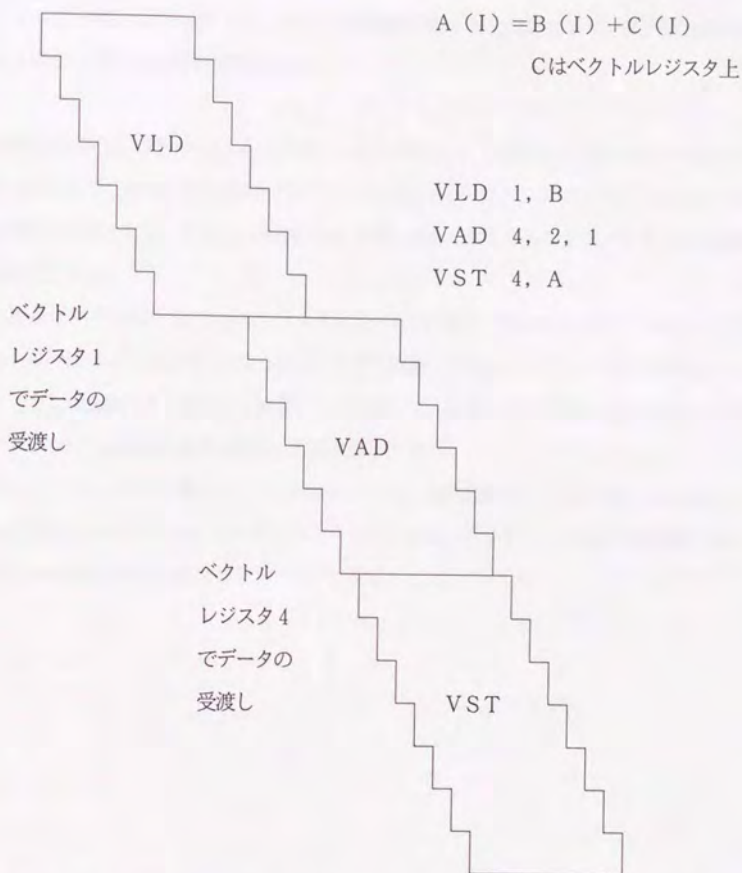
本ハードウェア実装では、ベクトル命令の命令起動のタイミングで後続する命令が実行待ちであれば、先行する命令を後続する命令が追い越さないように制御する。上記のメモリ待ち時間で命令制御全体をクロック停止する制御方式にすることで、命令の前後関係の状態を維持することが出来るので、命令の前後関係は保証される。

従って、図6. 7に示すように、ベクトル命令相互間で先行命令の結果を後続命令のオペランドとして使用する場合、命令実行のオーバーラップが可能であり、結果として、命令実行時間の短縮が可能である。

図6. 7で示すように、ベクトルロード命令VLDの結果はベクトルレジスタ1に入り、そのデータがベクトル加算命令VADのオペランドとして使用される。VADのもう一方のオペランドがベクトルレジスタ上であれば、ベクトルレジスタ1のデータはVLDで書かれたあと、VADのオペランドとして読み出される。もしも何らかの理由でVADの起動タイミングが遅れても、すでにベクトルレジスタ1にはデータが書かれているため、そのタイミングからVAD命令の起動が可能である。この図ではVADが4クロックサイクル遅れた場合を示す。

従って、ベクトル命令の連結実行は先行する命令のレジスタにデータが書かれたタイミング以降であれば、どのタイミングでも可能である。最良の起動タイミングは前記のバンクスロットでいえば、L1の直後のE2もしくはL2の直後のE1のタイミングである。同様に上記のVADに後続するベクトルストア命令VSTも互いに連結動作可能である。

図6. 7 ベクトル命令の連結実行



このベクトル命令の連結機能は、CRAY-1のチェイニング機構と良く似ている。しかし、CRAY-1の場合はベクトルレジスタに通常のECLロジックを使用しており、レジスタへのアクセスは同一レジスタに対し、1つしか許されていなかった。つまり同一レジスタへのアクセスは1つのアドレスを指定して、読み書きを同時に1サイクルタイムで行う必要があった。従って、命令発信のタイミングは先行するベクトルレジスタへの結果書き込みのタイミングがただ1回だけ存在し、そのタイミングを逃すとチェイニング機構は無効となる。さらに同一ベクトルレジスタへは多重のアクセスは許されないので、チェ

イニング機構が有効な時以外では、ベクトルレジスタの多重アクセス、もしくはベクトル命令の多重発信は出来ない。つまり、従属関係のある命令間での同時並列実行はチェイニングの場合だけしか許されない。また、従属関係のない独立な異なるレジスタ間でのみベクトル命令間の並列実行が有効である。

本実装方式では、同一ベクトルレジスタへのアクセスは、最高バンクの数つまり8個まで許されるので、複数の命令で同一ベクトルレジスタを使用していても、ベクトル命令の並列実行は可能となる。ただし、連結機能を使用してもしなくてもレジスタアクセスの個数は同じである。

ある命令シーケンスでベクトルレジスタ指定が互いに独立であれば、ベクトル命令はその命令実行環境がそろえば、任意のタイミングで起動してよい。しかし、同様の命令シーケンスでも連結動作する場合は、先行する命令のレジスタ書き込みが開始された以降のタイミングでのみ、後続の命令を起動しなければいけない。

従って、コンパイラの命令コード生成においては、連結機能になるべく避け、出来るだけ独立関係の命令を含む命令シーケンスとすべきである。つまり、できるだけ多種類のデータフローを実行する命令スケジューリングが望ましい。

6. 10 ベクトルデータのメモリ直接アクセス

スカラプロセッサではいわゆるキャッシュは等価的にメモリの実効的な高速化に寄与しており、現在ではキャッシュ関連技術は基本的な不可欠の技術となった。しかしベクトル処理ではこの常識は、必ずしも正しくはない。特に大規模な配列を取り扱うアプリケーションプログラムではキャッシュペナルティ（データがキャッシュ上にないためにメモリへ取りに行くこと）が大きく、実質的にキャッシュの効果が発揮できなかったり、効果が小さいことが多い。

まず第2章2. 2②で述べたようにスカラプロセッサではDO LOOPを命令シーケンスに従って1回転するため、キャッシュに取り込んだ該当ブロックは次の1回転でも使用できる。従って、もし配列データが連続アドレスに配置されていれば、キャッシュミスはキャッシュに取り込むブロックの大きさ（たとえば8個）ごとに1回の割合で発生するが、ブロック内の他のデータ（たとえば8個のうち7個）ではキャッシュ内アクセスが可能である。従って該当LOOP内でアクセスする配列がキャッシュ内に収まれば、キャッシュミスは12. 5%であり、キャッシュの効果は発揮できる。しかし、このように理想的な場合は少なく、配列データのメモリアccessが飛び飛びのデータだったり、間接アドレスアクセスの場合には、キャッシュミスが多発し、特にキャッシュを超えるような大規模配列では大きく性能を落とすことがある。

ベクトルアーキテクチャとキャッシュの整合性を検討する。

仮にベクトル命令によるメモリアccessにキャッシュを採用することを仮定する。VLD命令ではスカラの場合のようにブロックサイズのデータのみを取り込むのではなく、配列全体を取り込み、さらに複数の配列をアクセスすることになる。キャッシュ上には複数の配列をその全体として保持する必要がある。大規模配列の場合は、データ量がキャッシュ容量をこえることがあり、その場合は折角取り込んだデータが、再使用することもなく追いつき出されてしまい、キャッシュに保持しておく動作は完全に無駄になる場合が生ずる。従ってこの場合はキャッシュの効果は皆無であり、却って性能劣化に加担することになる。配列が小さい場合でも、読みだしたデータを繰り返し使わないときにはキャッシュミス検出から実際にデータが届くまでの立ち上がり時間はキャッシュ無しの場合よりも大きく、基本的には無駄である。つまりベクトル処理では、同一データを繰り返し使う確率が小さ

く、キャッシュ上のデータを繰り返し使うと言う本来の目的が注かされない。従ってCPUの非常に近くの場合に高速メモリを配置して中間データとして使用する時には、ハードウェアが自動的にデータを入れ換えるよりも何らかのプログラム指定による、データの入替えをした方が良い場合がある。FACOM 230-75 APUでは高速メモリを配置して、そのアドレスをプログラム指定可能なベクトルレジスタと呼び、計算の中間データや定数領域として使用したことがあった。この場合はベクトルレジスタはプログラムアクセス可能なベクトルデータ用の中間データ保持領域と見なして使用し、多少の効果があった。

以上のように、通常のキャッシュでのハードウェア制御による自動的なキャッシュ置き換えアルゴリズムはベクトルアクセスとは必ずしも整合せず、本研究のハードウェア実装では、ベクトル用のメモリアクセスはメモリとベクトルレジスタが直接アクセスする方式を採用した。従って、大規模配列では特に性能を発揮し、演算時間に占める立ち上がり時間オーバーヘッドも相対的に影響が少なくなるので、性能はさらに向上する。ただし、極端にベクトル長が短い場合は、たとえば要素数5以下では、スカラに比した性能向上率は小さい。

6. 1.1 スカラキャッシュ コヒーレンシ

上記のように、ベクトル命令ではキャッシュとの整合性が悪く、ベクトル命令でのアクセスにキャッシュを介在させるのはかえって性能的に不利である。したがって、メモリを直接アクセスすることにした。しかしスカラユニットとしての性能を確保するためにはキャッシュは必要であり、スカラユニットはこれを保持することにした。

その結果、スカラユニットのキャッシュの内容がベクトルメモリアクセスと一致していなければならない。すなわち、ベクトル命令によるメモリ内容の変更をキャッシュに反映しなければならない。スカラキャッシュの内容とメモリの内容を一致させることをキャッシュコヒーレンシと言う。

例えば、入出力装置からのメモリアクセスでも、メモリへの直接アクセスが生ずるため、該当するメモリアクセスのアドレスをスカラキャッシュに送り、キャッシュディレクトリに該当アドレスが存在すれば、そのディレクトリのエントリを無効化する。

スカラキャッシュはストアスルー方式を採用しており、スカラユニット以外のストアアク

セスにより、もしキャッシュにそのデータが存在すれば、そのデータを無効化する。

本研究で提案したアーキテクチャは、制限のない限り、出来るだけ多くの命令を先行して発信できることを特徴としている。したがって、何らかの指示がないかぎり命令はどんどん先の命令を実行していく。第5章で述べたとおり、逐次実行を行ういずれかの手段を講じるにより、ベクトル命令とスカラ命令との連携を取ることができる。たとえばスカラレジスタの情報連鎖によるベクトル命令とスカラ命令の逐次化、制御命令の実行、POST/WAIT命令による、メモリアクセスの逐次化などにより相互連携が可能である。この逐次化のタイミングでキャッシュコヒーレンシが保証されていなければならない。

ベクトル命令によるメモリ内容の変更をキャッシュに反映する方法は、1) ハードウェアが自動的にキャッシュコヒーレンシの管理をすること、2) キャッシュ内容をプログラムが管理すること、の2つがある。

2) のキャッシュ内容をプログラムが管理する方法には、①キャッシュ全体の内容を消去すること、②プログラム管理によるベクトル命令によるメモリ変更アドレス群をサーチし、そのアドレスのみ消去すること、③同じく該当アドレスに対応するデータでキャッシュ内容も変更すること等がある。プログラム管理の方法では、逐次化のタイミングで情報のアップデートのための多くの時間的オーバーヘッドを伴う。さらにベクトルストア命令の出力するアドレスやデータを保持しておく必要がある。2) の範囲ではいずれの方法も現実的な解は簡単には見つかりそうにもなく、ベクトル命令によるメモリ内容の変更に対処するには1) のハードウェアによる自動管理しかないと考えた。

さらに、もともとキャッシュそのものは全て、ハードウェアが管理するものであり、プログラムの負担を期待すべきでないという理由もあった。

この結果、スカラ命令側ではベクトルストアのプログラムへの影響はすべてハードウェアが実行するため、キャッシュの内容の管理はプログラムで意識することはなくなった。

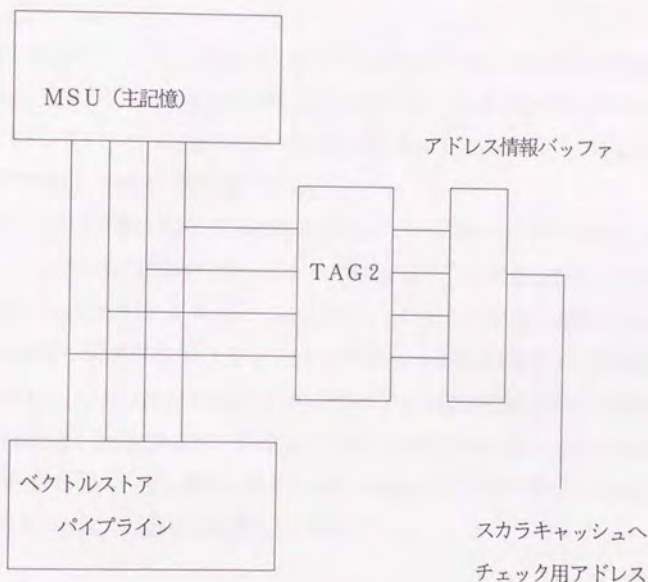
ただし、逐次化のタイミングだけはプログラムから起動をかける必要があり、上記のいくつかの手段により、逐次化を行う。したがって、逐次化のタイミング管理はすべて、コンパイラに任せることにした。

以上のように、ベクトルストア命令ではその書換えられたデータをスカラキャッシュへ反映するアーキテクチャとした。ハードウェアはベクトルストアアドレスをチェックし、そ

の同一アドレスデータがスカラキャッシュ上にあれば、それを無効化する。

図6. 8に示すように、VP-200ではストアアクセスは1サイクルに4本同時に動く。従って、ベクトルストアアクセスはそれぞれ4個のアドレスのチェックを同時に実施する。

図6. 8 ベクトルアクセスとキャッシュコヒーレンシ



ベクトルストアアクセスからMSU（主記憶装置）へ送出する4つのアドレスを毎サイクルごとにTAG 2と呼ばれるアドレスアレイでチェックする。一致が検出されると1サイクル当たり最大4個のアドレスがアドレスバッファに一時保留される。スカラキャッシュはディレクトリの一致チェックを1サイクルに1個のみしか受け付けないため、アドレスバッファで収容できる制限個数のアドレスを超えて、アドレスバッファに溜まるとストアアクセスパイプラインを停止させる。アドレスバッファからスカラキャッシュへアドレスが送出されて、バッファに空きができれば、またベクトルストアが再開できる。

一方、ベクトルとスカラで使用するメモリ領域はできるだけ分離するように、コンパイラ

が考慮するため、TAG 2でアドレスが一致する確率を小さくすることができる。さらに、TAG 2で一致が検出されたアドレスのみが、キャッシュに送られるため、送出されるアドレスの数は少なく、キャッシュへの影響も小さい。

逆に、コンパイラはスカラ命令で取り扱うデータと、ベクトル命令で取り扱うデータを、できるだけ共有化しないように割り当てなければならない。その理由は、第4章4.2で述べたように、ベクトルで作られた結果をすぐにスカラ命令で演算するようなことを、メモリデータを介して行くと、そのオーバーヘッドは非常に大きいからである。

TAG 2にはスカラユニットのキャッシュのアドレスアレイ（ディレクトリとかTAGとか呼ばれることもある）と同等のコピー情報が入っている。さらに上記ベクトルストア用の4本のアドレスそれぞれにそれぞれ同一情報のコピーが必要である。従ってチェック回路も含めて4組のハードウェアが必要である。

実際のハードウェアではTAG 2の構成はスカラユニットのキャッシュTAG部と構成が異なる。ハードウェアの節約のため、エントリ数を少なくし、その代わりエントリがカバーする大きさを大きくした。スカラユニットのキャッシュにはTAG 2の全てのエントリでカバーされている範囲以内しか置くことが出来ないようにしておくため、最悪の場合は本来スカラユニットに保持出来るはずのエントリがTAG 2の制限のおかげで使用出来ないことも起こる。実際にはそういった場合が発生する確率は小さい。エントリ未使用を実測したデータはないが、間接的にはスカラユニット性能の低下は殆ど起こらないことから、TAG 2の構成による妨害は殆どないと予測している。

一方、ベクトルロード命令でもスカラユニットのストアアクセスも逐次化の時点でメモリに反映されていなければならない。通常ではスカラ命令のストアアクセスは先行して実行されているため、ベクトルロード命令の起動はオーバーヘッドなく、即座に行うことが望ましい。本ハードウェア実現方式ではストアスルー方式のキャッシュ制御方式を採用しているためこの要望に沿った動作が実現できた。もしもスワップ方式を採用した場合は、前記のオーバーヘッドが大きく、この観点からもスワップ方式は採用出来ない。

以上のようなハードウェア実装方式によって、スカラユニットのキャッシュコヒーレンスを確保することができた。

6. 12 チャンネルバッファ

IO系のメモリアクセスは頻度が低いが、アクセス順位は高いため、ベクトルやスカラーメモリアクセスに影響があると考え、IO系からのメモリアクセスに伴うベクトル命令メモリアクセスの妨害を最小にする工夫を考えた。ベクトルアクセスは頻度が高く、いったん起動がかかるとできれば途中ストップをかけたくない。しかしIO系からのメモリアクセスはIO装置のオーバラン等をさけるために受け付けておかなければならない。

しかしIOアクセスは細かいデータを頻繁に転送することが多く、一端バッファに蓄えて、データ転送タイミングまで待たせて、一括転送する方が他のシステム内装置への影響を減少させることができて有利である。

今回の実装ではこのバッファを簡単なキャッシュの制御方式により実現した。理論的には単純なバッファで問題ないため、キャッシュ制御にするとキャッシュを無効化するタイミングに問題が残るのではないかと議論も有った。しかしチャンネルを統括する制御装置からのメモリアクセス方式が、キャッシュ方式に向いていると判断し、最終的にはキャッシュ方式を採用した。

キャッシュの中でのデータブロック長はスカラーユニットキャッシュとあわせて64バイトとしIO側からのデータがブロック単位で満杯になれば、データをメモリに書き込んだあと、該当ブロックを無効化する。逆にメモリからデータを読み出す時はブリフェッチにより、データを予め用意しておく。

最後に、ブロックの途中でIOからのメモリデータ送出が終了したら、IO割り込みの時点で、残りのデータをメモリへ格納し、逐次化を実行する。

チャンネルからのメモリストアアクセスでスカラーキャッシュの無効化は上記のブロックストアアクセスの時点で行われる。

以上のように、チャンネルバッファの実現により、チャンネルからのアクセスをブロック単位に転送することが可能になり、IOからのメモリアクセスの細切れでサイズの小さいデータアクセスによるベクトルメモリアクセス等への妨害を抑止することが出来た。

6. 1 3 メモリアクセスの同期制御

本研究のアーキテクチャではベクトル命令相互間でのメモリアクセスにおいて、同一IDを持つ命令間では逐次実行が保証されている。(第5章(5. 2))

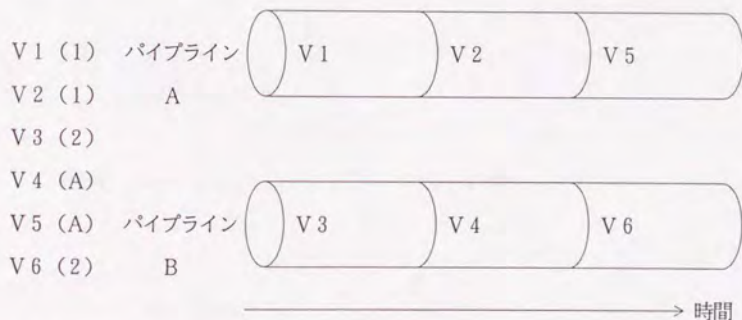
異なるIDのベクトル命令間では並列同時実行を前提にしているので、任意のタイミングで命令の起動ができる。2本のメモリアクセスパイプラインにおいて、一方に同一IDの命令群を割り当てると、他のパイプラインには異なるIDもしくは任意のパイプライン指定をもつ命令を割り当てて、パイプライン使用効率を高める。

図6. 9に示すように同一IDを持つ命令群は物理的にも同一メモリアクセスパイプラインに割り当て、他の命令は他のパイプラインに割り当てて、

従って同一パイプラインでは逐次実行が保たれる。

図6. 9 IDによる命令パイプライン割当

プログラム例



() は ID

A は ID 指定なし (どちらのパイプラインでも可)

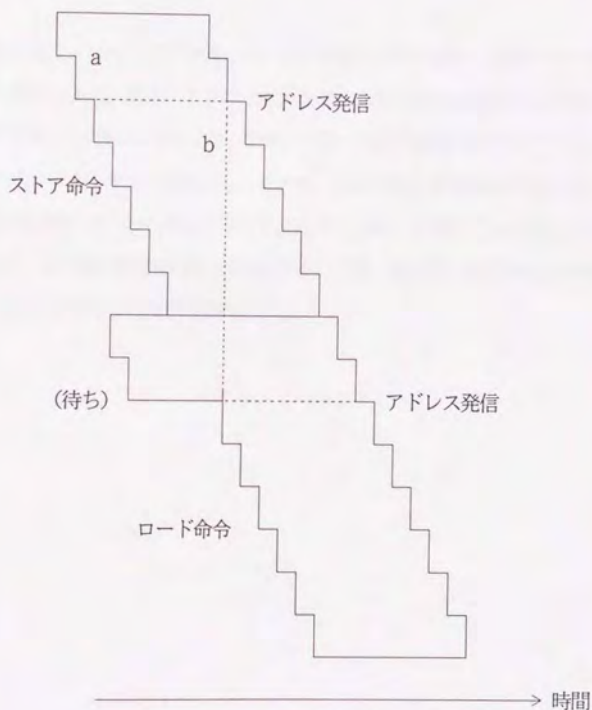
一方、同一命令内のメモリアクセスでは、インデックス順にアクセスが行われ、これが逆転することはない。当然ベクトルレジスタはインデックス順でアクセスされるため、両者

が一致して順番にアクセスできる。リスト処理の伴うメモリアクセスベクトル命令においてもこの順番は確保するように制御する。

従って、リスト処理の伴うベクトルストア命令において、同一アドレスへのストアが生じる場合でも、最大のインデックスに対応するデータが該当アドレスの結果として残ることになり、論理的な順序関係が保証される。

図6. 10は先行するストア命令と、後続するロード命令との間での逐次的な命令実行を記述している。a点ではアドレスをメモリ制御回路へ送出してメモリパイプラインを専有し、先行命令のリソース解除b点、の直後に後続命令の発信を行うことを示している。

図6. 10 メモリアクセス命令の逐次実行

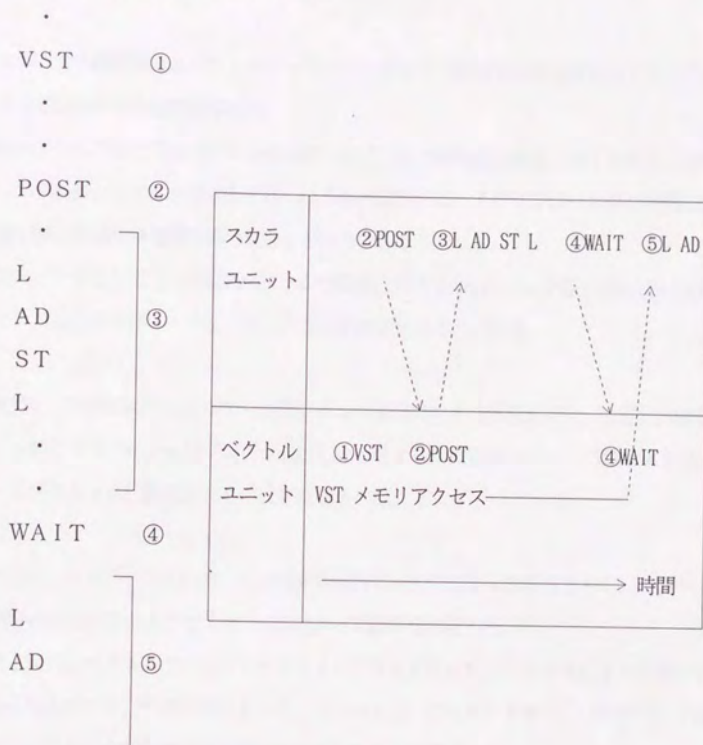


この図ではストア命令のリソース解除が遅れ、後続ロード命令のリソース専有が遅れているが、すでにストア命令のリソースが解除されていれば、後続ロード命令はすぐに発信できる。

実は上記の同一命令内でのインデックス順序が保たれる機能を利用すると、前後の命令間で全く同一配列データを利用している場合にさらに後続命令の発信を早める手段がある。即ち、a点の直後にすぐ後続ロード命令を発信可能である。しかし、異なる配列データをアクセスする場合は、ストア命令の後半のタイミングでアクセスするアドレスとロード命令の前の方でアクセスするデータがたまたま同一アドレスをアクセスした場合は、ロードの後でストア動作が実行され論理的に矛盾が生じる危険性があり、一般化は出来ない。従って、同一配列であることを敢えて指示すれば、この制御は成り立つが、アーキテクチャ上の指定の増加、パイプラインリソース管理の複雑さを増大することになることに比べてメリットが少ないこと等を考え、アーキテクチャとしては採用しなかった。

POST/WAIT命令は、ベクトル命令とスカラー命令との間での同期を主目的に設置した命令である。図6. 11に示すように、③のスカラー命令群は、POST命令以前の命令①VSTと並列に実行してしまうが、①のVST命令のメモリアccessが全部終了してから④のWAIT命令を実行し、その後、後続の⑤の命令群が実行される。従ってPOST命令以前の全ての命令はメモリアccessを含めて全部終了したあと、⑤以降の命令が実行され、命令間での逐次実行が行われる。一方、非同期に実行可能な命令群③は逐次実行命令群とは関係なく独立に実行できる。

図6. 11 POST/WAIT命令



第7章 総合性能評価

本章では本研究によるベクトルアーキテクチャとその実現方式を実現したVP-200によって総合的な性能評価を行う。

ハードウェアとしてはベクトル命令のパイプライン演算器の構成、パイプライン制御方式、ベクトルレジスタの構成とそれに伴う命令起動方式、メモリアクセス待ち時間と制御回路の停止制御及び連結制御の組合せを評価することになる。

コンパイラとしては、自動的ベクトル化機能、ベクトル命令の並列実行制御方式と命令シーケンスの最適化スケジューリング方式を評価することになる。

最初に、本研究のベクトルアーキテクチャの拡張に対する評価を行う。定量的な評価として、実アプリケーションプログラムにおけるベクトル化可能なループの数を第2世代のアーキテクチャと比較して、その改善度を示した。

つぎに、VP-200における並列命令実行アーキテクチャの評価として、コンパイラ能力も含めて以下のようなプログラムによって性能を評価した。

第1世代のベクトルプロセッサF230-75APUとVP-200によって同一プログラムを走行させ、その比較によって、コンパイラ、アーキテクチャ、ハードウェア実現方式における命令並列実行が非常に有効であることを確認した。

典型的なベンチマークプログラムである、livermore Loopを最も簡単なプログラム例として性能を評価し、実応用プログラムも実測した。最後に、システムソフトウェアとしては最大限性能を発揮できるシステムライブラリの性能を実測した。

単純な小ループ（カーネルと呼ぶ）から、本格的な実アプリケーションまで実測した。

本研究で意図した目標については性能目標を達成したことを実証した。

本章では、ベクトル化プログラムの性能をスカラプログラム性能との性能向上倍率によって示し、VP-200システムの性能を計る目安とした。その理由は、ベクトル化プログラム性能の性能向上倍率を示せば、ベクトル処理による性能改善の程度を知る基準とすることができ、さらに、当時の単体プロセッサとして、VP-200のスカラ性能は世界のトップクラスにあったため、等価的に絶対性能の比較にもなったことによる。

7. 1 ベクトル化可能性の評価

表7. 1に実アプリケーションプログラムに現れるループに対し、それがベクトル化できるかどうかの評価を示す。

ベクトル化可能性は、コンパイラのループの意味解釈能力と、ハードウェアのアーキテクチャのベクトルサポート機能の評価になる。

本実測では全194ループ中134ループがベクトル化でき、第2世代アーキテクチャでは111ループに停まった。従って57%から69%への1.21倍のベクトル化能力の改善があり、アーキテクチャの拡張の有効性が確認できた。

しかしこの比較対象は正確には第2世代アーキテクチャ(CRAY-1)ではなく、言わば2.5世代とも言うべきクレイX-MPにより測定したため、本来の改善度はもう少し良いのではないかと思われる。

第3章での分析では最大2倍の改善が期待でき、VOLTEXでは約2倍のベクトル化ループ数が測定され、目標を達成した。

3本のプログラム例ではベクトル化出来ないループもあり、アプリケーションプログラムそれぞれでいわゆる単純ベクトル演算しかない場合もあり、平均では1.2倍の効果に止まった。

表7. 1 ループのベクトル化

プログラム	プログラム中の ループの数	VP-200でベクトル 化されたループ数	X-MPでベクトル 化されたループ数
VORTEX	33	19	11
EULER	73	51	42
BARO	88	64	58
合計	194	134	111

7. 2 第1世代アーキテクチャとの性能評価

本研究で実現した第3世代のVP-200と、第一世代のベクトルプロセッサとしてのF230-75APUの性能測定結果を、表7. 2に示す。

共に同一プログラム(ABOLUS-R1)を使用して、各ループを比較した。ここではAPUでもベクトル化できたプログラムであり、この性能評価はベクトルアーキテクチャの拡張を伴わない範囲、つまり単純ベクトル処理での比較になる。

それぞれの実行時間は、SCALARはプログラム全体をスカラ処理のみで実行した時間を示し、VECTORはベクトル化を実施した時のプログラム実行時間を示す。比はスカラ実行時間とベクトルプログラム実行時間の比を示す。つまり、ベクトル性能のスカラ性能に対する性能向上率である。

またAPU/VPの項ではF230-75APUとVP-200での実行時間の比を示す。スカラおよびベクトル化プログラムの性能向上率を記述した。

スカラ/ベクトル性能比はAPUでは平均6. 5倍であるが、VP-200では4. 4倍である。従って、VP-200のベクトル性能の向上率が非常に高く、ほぼ目標性能(5. 0倍)を達成したことを実証した。

VP-200は64ビット演算、APUは36ビット演算の差があり、一律には比較はできないが、ベクトルパイプラインのピーク性能はVP-200は570MFLOPS、APUは22MFLOPSであり、性能比は2. 6倍である。一方、ベクトルプログラムのAPU/VP比は31. 6倍であり、ベクトルのみのピーク性能以上に性能が向上した。

プログラムにはスカラ実行部分が必ず存在するため、同一アーキテクチャでは性能比がベクトルピーク性能比以上に成りえないが、VP-200では演算ピーク性能値に隠れた、ロードストア命令や、データ編集命令などアーキテクチャの改善や、大幅な命令並列実行などが、実効的なベクトルプログラムの性能を引き上げていると思われる。

さらに、別の観点でみると、ハードウェア構成上、F230-75APUはベクトルパイプラインは1本であるが、VP-200はベクトルパイプラインが物理的に4本の構成で

ある（4多重インデックス番号のベクトル要素同時実行）。このため、本来なら、等価的にベクトル長が短くなることに相当し、オーバーヘッドが増加し、ピーク性能比よりもベクトル化プログラム性能は下回ってもおかしくない。しかし、逆の数値になっているのは、上記と同様に、スカラ命令とベクトル命令の並列実行、およびベクトル命令同志の徹底的な並列実行によりオーバーヘッドが軽減したからであると思われる。さらにAPUがメモリーメモリ命令アーキテクチャであることもAPUのオーバーヘッドを高めている原因と思われる。

表7. 2 F230-75APUとの比較 (AEOLUS-R1 による)

ループ名	F230-75APU 時間			VP-200 時間			APU/VP比	
	SCALAR	VECTOR	比	SCALAR	VECTOR	比	SCALAR	VECTOR
LOOP200	94	10	9.4	16.7	0.44	38.1	5.6	22.7
LOOP300	700	93	7.5	120.2	2.94	40.9	5.8	31.6
LOOP400	31	3	10.3	3.6	0.11	34.3	8.6	27.3
LOOP500	734	133	5.5	197.8	3.28	60.4	3.7	40.5
LOOP600	78	13	6.0	7.0	0.98	7.1	11.1	13.3
LOOP700	59	7	8.4	14.8	0.44	33.4	4.0	15.9
TOTAL	1696	259	6.5	360.2	8.19	44.0	4.7	31.6

7. 3 実アプリケーションプログラムによる評価

(1) 単純な演算の評価

リバモアループという非常に有名なベンチマークコードを例に取って性能評価を行った。このベンチマークプログラムでは、比較的小規模のkernelと呼ばれる14本のループプログラムによって構成されている。従って典型的な単純な演算パターンに対しての評価が可能である。

なお、この評価では通常のベクトルFORTRANコンパイラを使用し、そのプログラムを実機上で走行させた。コンパイラは最高の最適化レベルを指定してコンパイルを行っているが、測定用の特別なチューニングは行っていない。

ループ3では最高の倍率として44倍が達成され、ハードウェアおよびコンパイラの組合せとして、設計目標値としての50倍に近い、満足の行く数値が得られた。平均性能比として、16倍の性能が達成され、過去の機械に比べると大幅な前進があった。しかし、ループ5、6、11は本質的に逐次処理のためベクトル化できない部分が多く、自動ベクトル化には限界がある。これらのループではベクトル化の効果が殆どなく、スカラ性能比は殆ど1である。またループ13、14では部分的なベクトル化が可能ではあるが、ベクトル化出来ない部分が残されており、性能向上率は2倍強に止まった。この試算プログラムは、殆どがいわゆる単純ベクトル処理の範囲であるにもかかわらず、本質的にベクトル化出来ないものが含まれており、それはベクトル処理の限界でもある。

しかし、ここまで小さなループになれば工夫のしようがないものでも、実プログラム全体としては例えばDO LOOPの入替え等の、ベクトル化への工夫もあり得る訳で、マクロな分析も必要である。さらにアルゴリズムの見直しも必要である。

表7. 3 リバモアループの性能

Kernel	VP-200 MFLOPS	M380 MFLOPS	性能比 倍
1	331.4	10.0	33.1
2	180.4	11.3	16.0
3	338.2	7.7	43.7
4	88.1	5.8	15.3
5	10.0	9.6	1.0
6	9.5	9.3	1.0
7	331.0	13.9	23.8
8	90.4	12.4	7.3
9	260.8	12.6	20.8
10	85.9	7.9	10.9
11	4.8	4.6	1.0
12	115.3	4.7	24.5
13	6.2	2.4	2.6
14	13.8	5.7	2.4
平均	133.3	8.4	15.8

MFLOPS:

Million Floating

Operations Per Second

VP200 のスカラユニットは性能的にはM380のCPU と同一と言える。

従って性能比はVP200 のスカラ性能対ベクトル性能と等価である。

(2) 実アプリケーションプログラムでの評価

表7. 4に、実アプリケーションプログラムと同規模のプログラムでの性能測定結果を示す。本アプリケーションプログラムは、7. 1で述べたベクトル化の可能性を分析したプログラムも含まれており、大規模科学技術分野における典型的な分野から5本のプログラムが選ばれた。この性能測定においては、特に流体計算、プラズマ物理に関するプログラムで、性能向上率が高い。(VORTEX, 2DMHD, BARO)

VORTEXとBAROは、自動ベクトルコンパイラでプログラムをそのままを走行させた結果であり、実アプリケーションプログラムでも性能向上率が6. 1倍、27倍と高い。

EULER, 2DMHD, SHEAR はFFT ルーチンを内在しているが、そのままのプログラムを利用している。本来、FFT はライブラリで大幅に性能向上が可能ではあるため、これらのプログラムは性能が低いチューニングの可能性が十分にある。

EULER では3つのコンパイラ指示文を挿入して測定しているが、その2つはベクトル化の禁止であり、さらに全体の45%がFFT 走行時間で占められている。つまり、本質的にスカラ処理が多く、ベクトル化の性能向上は1. 3倍と限度がある。本測定では、コンパイラの自動ベクトル化による性能を実測することが主目的であった。さらに性能向上のためには、主にFFT のチューニングが必要である。

表7. 4 実アプリケーションプログラムの性能(1)

プログラム	VP200	VP200	SCALAR
	SCALAR	VECTOR	VECTOR
VORTEX	217.2 秒	35.4 秒	6.1 倍
EULER	6.3	4.8	1.3
2DMHD	43.4	2.6	16.7
SHEAR	164.4	83.6	2.0
BARO	1107.8	41.1	27.0

別の実応用プログラムについての測定結果を表7. 5に示す。

表7. 5 実アプリケーションプログラムの性能(2)

番号	計算分野	スカラ	ベクトル	比
1	流体	655 秒	36 秒	22.8 倍
2	流体	607	112	5.4
3	高エネルギー	205	26	7.8
4	高エネルギー	31	2.4	13.0
5	粒子	168	32	5.2
6	プラズマ	101	43	2.4
7	プラズマ	96	26	3.7
8	天気予報	413	109	3.8
9	核物理	25	3.5	7.3
10	エネルギー交換	76	3.8	19.9
平均		—		9.1

本測定値では平均倍率は、9.1 倍の性能であり、実アプリケーションプログラムで最高2.3 倍を示したことは価値がある。特に流体や高エネルギー物理学等で高性能が得られた。本測定では多少のチューニングを実施した。ただし、FORTRANプログラム字面上でのチューニングであり、アルゴリズムの変更はない。

自動コンパイルと多少の最適化指示により、スカラに比べて大体5 倍程度の性能向上が期待でき、良い場合では20 倍程度の性能がでることが分かった。
この結果は満足すべき値であり、ベクトルプロセッサ型のスーパーコンピュータの中ではトップクラスの性能を示すことができた。

7. 4 ライブラリの性能評価

システムライブラリは最大限の性能を達成して、ユーザに提供する。

専門家が十分に最適化を行い、ハードウェアの命令スケジューリングも徹底的に追求した結果である。従って、特別に性能向上率は高い。

例として線形1次方程式の解法と、高速フーリエ級数 (FFT) の性能を示す。

元数の増加に従ってスカラ性能比は高くなり、最高7.3倍を示している。チューニングにより目標の50倍以上の性能が達成できた。

FFTではアドレスアクセスパターンが連続アドレスでなく、性能は線型1次方程式よりも低い。

表7. 6 線形1次方程式の性能

(スカラ比性能)

元数	非対称	正対称
50	5.5 倍	4.5 倍
100	13.5	11.1
250	39.8	36.0
500	73.3	61.0

表7. 7 FFTの性能

元数	スカラ比の性能
64	7.7 倍
128	13.3
256	20.5
512	23.2
1024	32.0

上記の性能での7.3倍の値は非常に大きいが、その理由は元数が増大するのにしたがって、ベクトルの性能が相対的に高くなるのと同時にスカラ性能が低下することも効いていると思われる。従って、スカラ性能は一定でベクトル処理の立ち上がり時間の影響のみをその原因とするアムダールの法則だけで全てを解釈することは危険である。

7. 5 まとめ

本章ではベクトルプロセッサの総合的な性能評価を示した。

第2世代の機械(クレイX-MP)に比べて自動ベクトル化能力の向上が、実測により評価できた。第1世代の機械(F230-75)に比べて大幅なアーキテクチャ改良を達成したことを実証し、ベクトル処理の拡張機能および命令並列実行の効果を実証した。

個々の評価では、それぞれ本研究での性能目標を達成した。

一方、製品としての評価は使い易いことも大きな要因となる。本研究で実現されたVP-200は、自動ベクトル化コンパイラの寄与により、通常の素直な科学技術プログラムならばスカラ性能比5倍、非常にベクトル化に向いたプログラムでチューニングを実行すれば、スカラ性能比10~20倍の性能がでることが実証された。

アプリケーションプログラムは多種多様ではあるため、FORTRANコンパイラの自動ベクトル化には限界があるとは言え、平均的には性能向上が大きく、上記のようにリコンパイルのみで5倍、チューニングで10~20倍という使い安さが、製品としてベクトルプロセッサが世の中に受け入れられる大きな要因であろう。

第 8 章 結 言

8. 1 結 言

世界最高速の単一プロセッサを開発する計画の中で、本研究は最高速ベクトルアーキテクチャを構築し、その実現方式を考案した。本論文では、第2世代ベクトルアーキテクチャ(CRAY-1)の問題点を分析し、プログラムの高速実行を行うために、さらにベクトル化範囲を拡張し、ベクトルパイプライン演算器の使用効率を高め、大幅に命令の並列同時実行が可能となる言わば第3世代の新規アーキテクチャを構築した。

新規ベクトル範囲の拡張に関わるアーキテクチャでは、条件ベクトル処理を実現するために命令と機能を追加して、3つの条件処理方式を実現した。その中から最適な方式を選択する方式を考えた。その他、命令機能の拡張により、複雑なベクトル処理を実行するデータ編集命令を設置し、ベクトル処理効率を高めた。

ベクトル命令の並列実行に関する新設アーキテクチャとしては、ベクトルレジスタの動的構成変更を可能とし、コンパイラによる自動的な構成選択方式を実現した。

さらに、命令を順序不同に実行できることを許すために必要な命令制御方式を考案した。コンパイラによるプログラムのデータフロー解析結果にもとづき、データ間で逐次実行を実現する論理的パイプラインを定義できるアーキテクチャを構築した。さらにそれらを多重に動作させる方式を考案した。とくにメモリアクセス用の複数のパイプラインを定義し、それらを多重並列に動作させる方式を独自機能として実現した。

順序不同に命令を非同期並列実行する状態で、アドレス変換を高速に実行するための方式を考察し、静的アドレス変換の必然性を示し、これを実現した。

例外状態が発生した時の割り込み処理および例外情報保管方式も、命令の多重並列実行を行う上で、必然的な方式を実現した。

上記アーキテクチャにもとづいてハードウェア制御を設計した。

高速RAM(ランダムアクセスメモリ)を素子として使い、大容量ベクトルレジスタに最大限のデータ転送能力を実現する方式を考案し、ベクトルレジスタのインターリーブ方式

として実現した。この資源をベクトルユニットの中心に置いて、ベクトル命令の並列実行を実現する制御方式を考案した。パイプライン演算器をさらにパイプライン制御する方式を考案し、実現した。さらにそのパイプライン制御回路によって複数のパイプライン演算器を制御する方式を案出した。ベクトルレジスタへのデータアクセスを制御するバンクスロットという概念を導入し、命令発信のためのタイミング制御を大幅に簡略化した。バンクスロット制御のもとで、メモリアクセス待ち時間に対応して、クロック停止をとまなう命令制御状態の凍結制御方式を考案し、実現した。これも命令制御論理の簡略化に大きく貢献した。

ベクトルプロセッサのメモリアクセス方式を考案し、ベクトル命令のメモリアクセスはキャッシュを介在させないで、直接アクセスにすることが有利であることを示し、これを実現した。スカラユニットのキャッシュの内容とをメモリの内容を一致させる方式を考案し、実装した。チャンネルからのメモリアクセスの軽減を図るため、一種のキャッシュを実現し、ベクトルアクセスへの影響を最小限に抑える方式を実現した。

本研究の総合評価を行った。

新規アーキテクチャを利用して、コンパイラが自動ベクトル化能力を向上させていることを検証するために、実アプリケーションプログラムにおけるベクトル化可能ループの数を実測した。その結果、本研究のアーキテクチャの有効性が確かめられた。

簡単なベンチマークテストプログラムから、本格的なアプリケーションプログラムまでの性能評価を行った。

簡単なカーネルレベルプログラムでも平均1.6倍、最大で4.4倍の性能向上が得られた。実アプリケーションプログラムでは、ベクトル/スカラ性能比は最大2.7倍を達成しており、平均的に9倍の性能向上が達成された。従って、所期の性能目標は達成された。最大限の最適化を行ったシステムライブラリではさらに高い性能が得られた。ベクトルプログラムの性能はスカラ性能に比べて最大7.3倍にも達した。

一方、スカラ性能と比べて1~2倍程度の向上しか達成出来ないものもあるが、その原因はスカラ走行比率が非常に多いものである。もしくは本質的に並列実行による性能向上が出来ない演算である。局所的なベクトル化技術のみならず、大きな範囲での最適化によって、さらにベクトル化率が向上することもある。そこでは応用プログラムの解法アルゴリズムとの関係が大きな影響を与える。しかし、ベンダとしては、プログラムをいじらない

範囲で、コンパイル技術によって改良を進めるにとどめなければならないという限界がある。

本アーキテクチャ構築の研究は1978年ころからスタートし、1980年代の始めに完成した。さらに次々と製品化が進むに従って、改良修正され今日に至っている。

実現された機械は、富士通からFACOM VP-100/VP-200として製品化され、VP-400/50/30等の後続機に同一アーキテクチャで実現された。さらにVP100Eシリーズ、VP2000シリーズ、VPP500、VPP300等に受け継がれ、富士通のベクトルプロセッサのアーキテクチャのベースとなった。

特に言語、FORTRAN、C等コンパイラにおける、コンパイル技術の進歩に呼応して、むしろその影響を受けながら本アーキテクチャの構築が行われ、かつ改良された。

これらのハードウェア、OS、言語全体の製品が、国内外の市場で広く受け入れられ、ベクトルプロセッサ市場に広く受け入れられたことは、本アーキテクチャとその実現方式がそのベースとして貢献したと思われる。

本アーキテクチャを実現したスーパーコンピュータが、科学技術計算のための1ツールとして、科学技術発展の一翼を担うことができたことが、本研究の成果であると信じている。

8. 2 今後の展望

本研究を実施した時代では、高速素子による単体プロセッサ性能の向上を最大限の目標としてスーパーコンピュータの開発が行われてきた。

しかし、ECL、GaAs等の超高速テクノロジーの限界が見え、CMOSテクノロジーの急激な速度改良と高集積化によるコストダウンの両面から、CMOSの地位がスーパーコンピュータでも著しく向上している。

今後はCMOSに代表される超高密度LSIが一般化され、旧来に比べて非常に激しくハードウェア価格が低下していくと思われる。単一プロセッサでの高速化よりも、多重に低速プロセッサを設置してもシステムとしてのスループットに対する価格性能比が変わらない、または却って安いこともあり得る状態になっている。

このため、物理的な並列プロセッサシステムは現実性を高めている。

富士通が製品化した、VPP（ベクトルプロセッサの並列処理方式）は今後のスーパーコンピュータの発展への一つの解であると思われる。

一方、標準マイクロプロセッサチップによる低価格性の利点は大きく、超並列機いわゆるMPP方式による、ベクトル処理の効率が高まれば、価格性能比としてベクトルプロセッサとの競合もあり得る可能性があり、今後も注目していく必要がある。

2000年にはシステムとして、実効性能として1TFLOPS実現の可能性は大きく、そのためには最大性能として、10TFLOPSに近い性能が必要とされると思われる。我々はこの目標に対して、積極的に挑戦して行く必要があろう。

本論文はFACOM VP-100/200の開発にあたり、ベクトルアーキテクチャおよびハードウェアの実現方式に関する研究開発の成果を、東京大学工学部電子情報工学科田中英彦教授のご指導のもとでまとめたものである。

本論文の作成に当たっては高木幹雄教授、井上博允教授、武市正人教授、喜連川優助教授の方々からきわめて有益なご助言をいただいた。

筆者は会社の職務に基づいて行った研究を今回整理をして論文としてまとめた。

現職場の上司の小坂義裕氏の励ましがなければ、本論文の成立は無かったと思われる。感謝する次第である。

さらに、筆者の同窓の友人である長島重夫君の勧めもあり、論文としてまとめることを思い立った次第であり、彼の助言に感謝するものである。たまたま彼が実施した研究と私のそれとは多くの分野でオーバーラップしている。従って、氏の博士論文とは、背景として共通のスーパーコンピュータ分野であるがため、共通の記述内容がある。しかし、研究内容としてはアーキテクチャの構成概念やハードウェアの実現技術として全く異なるものであり、互いにオリジナリティを発揮した結果である。なぜなら、後日になって、研究当時は同一分野で独立にそれぞれが研究を行っていたことが判明したからである。

当時航空宇宙研究所の三好甫室長殿には幾つかの仕様決定にあたり、貴重なご助言をいただき、そのうちのいくつかはアーキテクチャおよびハードウェア実現上に採用させていただいたものがある。深く感謝する次第である。

本研究は、富士通内の多くの人々の研究と努力が総結集したものである。

とくにコンパイラ部隊の人達には基礎資料として、数々のプログラム分析にもとづく、ベクトルアーキテクチャの構築に多大な支援を受けた。性能の評価に対しても、コンパイラ部隊とシステムエンジニア部隊の人々の成果であり、本論文ではこれを参照している。

さらに当時米国海軍大学院大学のDr. Mendezによるコンパイラと性能評価に対して重要な寄与があり、本論文もこれを参照している。同様に感謝する次第である。

筆者は当時のハードウェア設計のリーダーであった。

本プロジェクトの推進は当時の山本、二宮氏ほかトップ経営陣の経営判断により実施されたものであり、プロジェクト推進の許可を受けられたことは幸運であった。

当時の事業部長であった平栗、田畑氏が特に開発推進に責任者であり、三輪、鈴木、秋山、田口、槌本氏の指導のもとで、プロジェクトが推進された。

担当の技術者集団としては、浦川、新開、黒羽氏を始めとするOS部隊、野崎、棚倉、磯辺、神谷氏他を始めとする言語部隊、および田村、千葉、茂木、岡本、追永、奥谷、中谷、伊藤、栗林、坂本氏等ハード開発部隊がいた。いずれの部隊も心身を削って開発に心血を注ぎ、製品化を達成した。

この他にも、間接的に製品化に参加した共通技術部門、たとえば回路開発部門、半導体部門、製造・検査部門の人達の努力のおかげで製品開発が成功したものである。

さらにシステムエンジニア部隊においては田子、松浦、田原、折居氏を始めとする人々により、実際のアプリケーションの解法の研究が進んだ。これも製品の販売拡大に大きな力であり、現場のアプリケーションからの開発部門へのフィードバックにも貢献があった。

富士通アメリカの三浦氏は特に海外での製品紹介、学会活動、アプリケーションアルゴリズム開発等に貢献した。

以上のように名前を上げた人々に加え、さらに実質的に製品開発および製品化への沢山の人の参加により、VPシリーズプロジェクトが達成されたわけであり、全ての人々に心からの感謝を表明する次第である。

幸丸筆論文

- 1 内田啓一郎
スーパーコンピュータの開発
FACOMジャーナル VOL. 11 NO. 10・11 (1985) pp. 95-98
- 2 内田啓一郎
超高速論理素子を用いた超高速コンピュータの可能性
技術予測シリーズ 第2巻: エレクトロニクス技術 (産業電子) pp. 23-30
- 3 内田啓一郎
スーパーコンピュータ
電気学会誌 108巻10号、昭63 pp. 977-980
- 4 内田啓一郎
スーパーコンピュータのハードウェア
スーパーコンピュータとその応用小特集
電子情報通信学会誌 VOL. 75 NO. 2 pp. 114-119
1992年2月
- 5 内田啓一郎
スーパーコンピュータ
FJITSU VOL. 42, NO. 1 (1991) pp. 95-100
- 6 内田啓一郎、坂本一志
スーパーコンピュータ FACOM VP
情報処理 命令セットアーキテクチャ特集 Vol. 29 No. 12 pp. 1527-1529
- 7 三輪修、乾範男、久米宣明、内田啓一郎
FACOM 230-75 アレイプロセッサ
情報処理 VOL. 18 No. 4 (4月 1977) pp. 410-415
- 8 三輪修、久米宣明、内田啓一郎、鈴木滋、棚倉由行、磯辺文雄
FACOM 230-75 アレイプロセッサシステム
雑誌FJITSU VOL. 29 NO. 1 (1978) pp. 94-128

- 9 K. Uchida, Y. Seta, Y. Tanakura
The FACOM 230-75 ARRAY PROCESSOR SYSTEM
3rd USA-JAPAN Computer Conference, 1978 pp. 369-373
- 10 岡本哲郎、田村宏、内田啓一郎
スーパーコンピュータFACOM VPのハードウェア
FUJITSU VOL. 35 No. 4 (1984) pp. 465-476
- 11 K. Miura, K. Uchida
FACOM VECTOR PROCESSOR VP-100/VP-200
NATO ASI Series, VOL. F7 High-Speed Computation (1984) pp. 127-138
- 12 FUJITSU VP/VPXシリーズ ハードウェア機能説明書
- 13 M. Motegi, K. Uchida, T. Tsuchimoto
The Architecture of the FACOM Vector Processor
PARALLEL COMPUTING 83 (1984) pp. 541-546
- 14 K. Uchida, M. Itoh
HIGH SPEED VECTOR PROCESSORS IN JAPAN
Computer Physics Communications 37(1985) pp. 7-13
North-Holland, Amsterdam
- 15 内田啓一郎、奥谷茂明
命令の並列実行方式
情報処理学会第25回(昭和57年後期)全国大会 6F-2 pp. 137-138
- 16 内田啓一郎、岡本哲郎
多重データ処理とその同期化方式
情報処理学会第25回(昭和57年後期)全国大会 6F-8 pp. 149-150
- 17 清水和之、内田啓一郎
FUJITSU VP2000シリーズのシステム概要
FUJITSU VP2000特集号
雑誌FUJITSU Vol. 41, NO. 1 PP. 3-11

参考文献

第1章

- 1-1 内田啓一郎
スーパーコンピュータの開発
FACOMジャーナル VOL. 11 NO.10-11 (1985) pp.95-98
- 1-2 内田啓一郎
超高速論理素子を用いた超高速コンピュータの可能性
技術予測シリーズ 第2巻: エレクトロニクス技術 (産業電子) pp.23-30
- 1-3 内田啓一郎
スーパーコンピュータ
電気学会誌 108巻10号、昭63 pp.977-980
- 1-4 内田啓一郎
スーパーコンピュータのハードウェア
スーパーコンピュータとその応用小特集
電子情報通信学会誌 VOL. 75 NO. 2 pp.114-119
1992年2月
- 1-5 内田啓一郎
スーパーコンピュータ
FJITSU VOL42, NO. 1 (1991) pp.95-100
- 1-6 長島重夫
ベクトルプロセッサの高速化方式に関する研究
平成3年3月、東京大学工学部電子工学科博士論文
- 1-7 立花隆
世界一のスピードを生むスパコンの仕掛け
科学朝日 5月 1991年 pp.124-131
- 1-8 内田啓一郎、坂本一志
スーパーコンピュータ FACOM VP
情報処理 命令セットアーキテクチャ特集 Vol.29 No.12 pp.1527-1529

- 1-9 Thornton
Parallel Operation in the Control Data 6600
AFIPS 1964, FJCC, Vol. 26 Pt2, Spartan Book, N. Y. pp. 33-40
- 1-10 D. W. Anderson, F. J. Sparacio, R. M. Tomasulo
The IBM System/360 Model 91
Machine Philosophy and Instruction Handling
IBM Journal R&D Vol. 11, No. 1, Jan. 1967 pp. 8-24
- 1-11 R. M. Tomasulo
An efficient algorithm for exploiting multiple arithmetic units
IBM Journal, R&D Vol. 11, No. 1, Jan. 1967, pp. 25-33
- 1-12 M. J. Flynn
Some Computer Organization and Their Effectiveness
IEEE Transaction on Computer C-12, 1972, pp. 948-960
- 1-13 R. G. Hintz, D. P. Tate
Control Data STAR-100 Processor Design
Proc. of COMPCON, 1972, pp. 1-4
- 1-14 Texas Instrument INC.
Processing and Arithmetic pipelines of the TI/ASC System
A Description of the Advanced Scientific Computer System
1973
- 1-15 小高俊彦 他
HITAC M180 内蔵アレイプロセッサ
日立評論 VOL. 21. NO. 1, 1978 pp. 451-456
- 1-16 元岡 達
スーパーコンピュータの現状と展望
情報処理 Vol. 22, 12, 1981 pp. 1103-1110
- 1-16 G. Radin
The 801 Minicomputer
Symposium on Architecture Support for Programming, Language and
operation, VOL. 10, No. 2, Mar. 1982, pp. 39-47

第2章

- 2-1 A. Groves, R. Oehler
An IBM Second Generation RISC Processor Architecture
Proceedings of ICCD' 89 IEEE PP.134-137
- 2-2 G. Amdahl
Architecture of IBM System 360
IBM Juornal Res. and Dev. Vol.8, No.2 pp.87-101 Apr. 1964
- 2-3 三輪修、乾範男、久米宣明、内田啓一郎
FACOM 230-75 アレイプロセッサ
情報処理 VOL. 18 No. 4 (4月 1977) pp.410-415
- 2-4 三輪修、久米宣明、内田啓一郎、鈴木滋、棚倉由行、磯辺文雄
FACOM 230-75 アレイプロセッサシステム
雑誌FUJITSU VOL. 29 NO. 1 (1978) pp.94-128
- 2-5 K.Uchida, Y. Seta, Y. Tanakura
The FACOM 230-75 ARRAY PROCESSOR SYSTEM
3rd USA-JAPAN Computer Conference, 1978 pp.369-373
- 2-6 岡本哲郎、田村宏、内田啓一郎
スーパーコンピュータFACOM VPのハードウェア pp.465-475
FUJITSU VOL. 35 No. 4 (1984)
- 2-7 磯部文雄、神谷幸男、黒羽法男
スーパーコンピュータFACOM VPのソフトウェア
FUJITSU VOL. 35 No. 4 (1984) pp.477-488
- 2-8 平栗俊男、他
マシンサイクル7.5nsを達成した並列パイプライン処理
方式のスーパーコンピュータFACOM VP
日経エレクトロニクス 1983. 4. 11 pp.131-184
- 2-9 Kenichi Miura, Keiichiro Uchida
FACOM VECTOR PROCESSOR VP-100/VP-200
NATO ASI Series Vol. F7 High-Speed Computation(1984) pp.127-138

- 2-10 小高俊彦、他
 最大性能が630MFLOPSで1Gバイトの半導体拡張
 記憶が付くスーパーコンピュータHITAC S-810
 日経エレクトロニクス 1983. 4. 11 pp.159-184
- 2-11 R. M. Russel
 CRAY-1 Computer System
 Communication of the ACM, Vol.21, No. 1, 1978, pp.63-72
- 2-12 清水和之、内田啓一郎
 FUJITSU VP2000シリーズのシステム概要
 FUJITSU VP2000特集号
 雑誌FUJITSU Vol. 41, NO. 1 pp.3-11
- 2-13 G.H. Barnes, etc
 The ILLIAC IV Computer
 IEEE Transaction on Computer Vol. C-17 No.8 (AUGUST 1968) pp.746-757
- 2-14 スーパーコンピュータが超並列型へ
 分散共有メモリを採用
 日経エレクトロニクス 3-14, 1994 pp.119-127
- 2-15 J. A. Fiser
 Very Long Instruction Word Architecture and Eli-512
 Proc. of 10th international Symposium on Computer Architecture
 pp.140-150, 1983

第3章

- 3-1 神谷幸男 他
 FORTRANプログラムの特徴とベクトルプロセッサアーキテクチャ
 情報処理学会第25回(昭和57年後期)全国大会 6F-1 pp.135-136
- 3-2 T. Matsuura, S. Kamiya, M. Takiuchi
 Design Concept of the FACOM VP Based on extensive Analyses of
 Applications
 IEEE International Conference on Computer design ICCD'94 pp.232-237

第4章

- 4-1 FUJITSU VP/VPXシリーズ ハードウェア機能説明書
富士通
- 4-2 FACOM Mシリーズハードウェア機能説明書Ⅰ（命令編） 富士通
- 4-3 FACOM Mシリーズハードウェア機能説明書Ⅱ（機能編） 富士通
- 4-4 平林俊弘、青木正樹
ベクトルプロセッサの柔軟性をもたせたベクトルレジスタ方式
情報処理学会第25回（昭和57年後期）全国大会 6F-4 pp.141-142
- 4-5 茂木正徳
マスク機能付ベクトル処理
情報処理学会第25回（昭和57年後期）全国大会 6F-5 pp.143-144
- 4-6 滝内政昭 他
ベクトルプロセッサの効率的条件文ベクトル化方式
情報処理学会第25回（昭和57年後期）全国大会 6F-6 pp.145-146
- 4-7 田村宏、追永勇次
ベクトル集散命令制御
情報処理学会第25回（昭和57年後期）全国大会 6F-7 pp.147-148
- 4-8 神谷幸男 他
FORTRANプログラムのベクトル化要素
情報処理学会第26回（昭和58年前期）全国大会 6N-4 pp.171-172
- 4-9 高島秀夫 他
FORTRANプログラムのベクトルデータ構造
情報処理学会第26回（昭和58年前期）全国大会 6N-5 pp.173-174
- 4-10 滝内政昭 他
ベクトル計算機の条件ベクトル化方式の効果
情報処理学会第27回（昭和58年後期）全国大会 5P-7 pp.173-174
- 4-11 平林俊弘 他
ベクトルプロセッサのベクトルレジスタ可変構成の効果
情報処理学会第27回（昭和58年後期）全国大会 7P-3 pp.195-196

- 4-12 M. Motegi, K. Uchida, T. Tsuchimoto
The Architecture of the FACOM Vector Processor
PARALLEL COMPUTING 83 (1984) pp. 541-546
- 4-13 K. Miura, K. Uchida
FACOM VECTOR PROCESSOR VP-100/VP-200
NATO ASI Series, VOL. F7 High-Speed Computation (1984) pp. 127-138
- 4-14 K. Uchida, M. Itoh
HIGH SPEED VECTOR PROCESSORS IN JAPAN
Computer Physics Communications 37(1985) pp. 7-13
North-Holland, Amsterdam
- 4-15 S. Kamiya, F. Isobe, H. Takashima, M. Takiuchi
Practical vectorization techniques for the FACOM VP
INFORMATION PROCESSING 83, R. E. A. Mason(ed) 1983 pp. 389-394
- 4-16 H. Tamura, S. Kamiya, T. Ishigai
FACOM VP=100/200: Supercomputers with ease of use
Parallel Computing 2 (1985) North-Holland pp. 87-107
- 4-17 伊藤賢一
複号計算機システムにおけるスーパーコンピュータ
情報処理学会第25回(昭和57年後期)全国大会 7F-1 pp. 151-152

第5章

- 5-1 内田啓一郎、奥谷茂明
命令の並列実行方式
情報処理学会第25回(昭和57年後期)全国大会 6F-2 pp. 137-138
- 5-2 高嶋秀夫 他
ベクトルプロセッサの並列ベクトル演算方式
情報処理学会第25回(昭和57年後期)全国大会 6F-3 pp. 139-140
- 5-3 内田啓一郎、岡本哲郎
多重データ処理とその同期化方式
情報処理学会第25回(昭和57年後期)全国大会 6F-8 pp. 149-150

第6章

6-1 坂本一志 他

ベクトルプログラムにおける並列実行制御方式

情報処理学会第27回(昭和58年後期)全国大会 5P-1 pp.161-162

6-3 伊藤幹雄 他

ベクトル・プロセッサにおけるメモリ・アクセスの制御と性能

情報処理学会第27回(昭和58年後期)全国大会 5P-2 pp.163-164

6-4 中谷彰二 他

FACOM-VPにおけるアライン制御方式

情報処理学会第27回(昭和58年後期)全国大会 5P-3 pp.165-166

6-5 千葉隆、栗林

ベクトルプロセッサにおけるバッファインバリデーション方式

情報処理学会第27回(昭和58年後期)全国大会 5P-4 pp.167-168

6-6 千葉隆、伊藤幹雄

ベクトルプロセッサにおけるチャンネル専用バッファの制御方式

情報処理学会第27回(昭和58年後期)全国大会 5P-5 pp.169-170

第7章

7-1 H. Ina, S. Kamiya, J. Mikami

Language and Software Development Tools for Supercomputers

Computer Physics Communications 38(1985) pp.211-219

North-Holland

7-2 T. Matsuura, K. Miura, M. Makino

Supercomputer Performance without Toil: FORTRAN Implemented

Vector Algorithms on the VP-100/200

Computer Physics Communications 37(1985) pp.101-107

North-Holland

7-3 K. Miura, T. Tanakura, S. Kamiya

Software Oriented Approach for Supercomputer Design

Fall Joint Computer Conference, 1986 pp.1020-1025

7 - 4 R. H. Mendez

Supercomputer Benchmarks Gives Edge to Fujitsu

SIAM News 3, 1984

7 - 5 R. H. Mendez

Benchmarks on Japanese and American Supercomputers

--Preliminary Results--

IEEE Transactions on Computers, Vol. C-33, No. 4, April 1984 pp. 374

