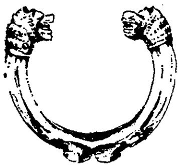
連載講座



計算機の記憶システム—V

スーパコンピュータの記憶システム†

村　上　和　彰††

1. はじめに

スーパコンピュータ (*supercomputer*)[3),7),9),13),18)]とは，その時代における最高性能の計算機を指して言う場合の総称である．その性能基準は時代とともに変化しているが，1980年代で100 MFLOPS (=10^{11} 浮動小数点演算/秒) 以上，現在では 1 GFLOPS (=10^{12} 浮動小数点演算/秒) 以上のピーク性能を有する計算機をスーパコンピュータと呼んでいる．

スーパコンピュータのアーキテクチャは，1976年に登場した米 Cray Research 社 (CRI) の**ベクトル・プロセッサ** CRAY-1[25)] の成功以後，ベクトル処理方式が主流となっている．**ベクトル処理** (*vector processing*)[16)] とは，たとえば，64要素からなるベクトル2個の加算を行い，結果として64要素のベクトル1個を出力するようなものであり，これらの操作をベクトル命令1個で行う．一方，**スカラ処理** (*scalar processing*) でこれと同様のことを行うには，複数のスカラ命令からなるループを実行することになる．つまり，ループの各イタレーションにおいて，①64要素のうちの1要素分の加算，②インデックス更新，そして，③ループの先頭への戻り分岐，ということを繰り返す．このように，ベクトル命令1個は，1個のスカラ・ループにまるまる相当している．

さらに，ベクトル処理は，上記の加算などの算術論理演算操作，および，メモリとの間のロード/ストア操作を**パイプライン処理** (*pipelining*) している点に特徴がある．一般に，命令実行過程をパイプライン処理する方式を**命令パイプライン** (*instruction pipeline*) 方式と，また，その実行ステージをさらにパイプライン処理する方式を**演算パイプライン** (*arithmetic pipeline*) 方式とそれぞれ呼ぶ．命令パイプライン処理とは，命令の実行過程をたとえば，①命令フェッチ (IF)，②命令デコード (ID)，③実行 (E)，および，④書込み (WB) の4ステージに分割し，それらをオーバラップ処理するものである．本方式はプロセッサ高速化のための基本技術であり，プロセッサの種類 (スカラ vs. ベクトル) を問わず広く採用されている．一方，演算パイプライン方式のほうは，スカラ・プロセッサでの採用は稀であり☆，ベクトル・プロセッサでの採用が主となっている☆☆．演算パイプライン処理により，たとえば浮動小数点加算過程は，①指数部比較，②仮数部桁合せ，③仮数部加算，および，④正規化の4ステージに分割され，それらがオーバラップ処理される．

このように，ベクトル・プロセッサは，複数のスカラから構成されるベクトルに対して演算パイプライン処理を施すことにより，演算スループットをスカラ・プロセッサに比べて飛躍的に向上させている．それでは，ベクトル・プロセッサの記憶システムは，スカラ・プロセッサのそれとどのように異なるのだろうか？一般に，スカラ・プロセッサでは，キャッシュを用いて実効的なメモリ・アクセス・レイテンシの低減化に努めている (本連載講座-II参照)．一方，ベクトル・プロセッサは大規模科学技術計算分野において特に有用だが，プログラム・サイズが大きく，かつ，実行時間が長い科学技術計算プログラムにおいては，非常に大きなデータへのアクセス，しかも，参照の局所性の低いアクセスを行う場合が多い．参照

† Memory Architectures of Vector Supercomputers by Kazuaki MURAKAMI (Department of Information Systems, Kyushu University).

†† 九州大学大学院総合理工学研究科情報システム学専攻

☆ これは，まず，従来のCISCプロセッサでは命令パイプライン処理自身が完全に実装されていなかったこと，第2に，RISCプロセッサではEステージに複数サイクルを要する命令 (浮動小数点演算命令など) 自身を定義していなかったことによる．最近のスーパスカラ・プロセッサなどの高性能RISCプロセッサでは，演算パイプライン処理も当り前になっている．

☆☆ そのため，ベクトル処理方式と演算パイプライン方式とを誤って同一視することもある．

の局所性が低いとメモリの階層化の効果が薄れ，結果としてキャッシュが役に立たないという事態を招くことになる．すなわち，ベクトル・プロセッサでは，この種の問題に対処する必要がある．

本稿では，**ベクトル型スーパコンピュータ**の記憶システムに関して，アーキテクチャおよび構成方式を中心に述べる．なお，近年台頭著しい**マルチプロセッサ型スーパコンピュータ**の記憶システムについては，本連載講座-Ⅳに譲る．まず，**2.** でベクトル・アーキテクチャを概観する．**3.** および **4.** で，ベクトル・レジスタと主記憶の一般的な構成法について紹介する．**5.** では，主記憶上の配列へのアクセス方式を述べる．

2. ベクトル・アーキテクチャ

ベクトル・アーキテクチャと記憶システムとは，非常に密接な関係にある．ベクトル・アーキテクチャは，ベクトル演算のオペランドであるベクトルデータをメモリおよびレジスタのいずれに置くかで，次の2方式に分類される☆．

1. メモリ-メモリ演算方式 (*memory-memory architecture*)：最初のベクトル・プロセッサである CDC STAR-100[20] および TI ASC[28]，STAR-100 の後継機である CDC CYBER-205[23]，などに代表される方式で，二つのソース・オペランドと一つのデスティネーション・オペランドいずれもメモリ上に置いてベクトル演算を行う．

2. レジスタ-レジスタ演算方式 (*vector-register architecture*)：CRAY-1 で初めて採用された方式で，二つのソース・オペランドと一つのデスティネーション・オペランドいずれもレジスタ（これを**ベクトル・レジスタ**と呼ぶ）上に置いてベクトル演算を行う．ベクトル演算の前後に，ベクトルデータのベクトル・レジスタへのロード，および，ベクトル・レジスタからのストアが必要で，いわゆる**ロード／ストア・アーキテクチャ**のベクトル・プロセッサ版に相当する．

一般に，長さ n のベクトルを一連のベクトル命令で処理した場合の実行時間 $T(n)$ は下式で表せる[19]．

$$T(n)=T_{base}+\left\lceil\frac{n}{MVL}\right\rceil\times(T_{strip}+T_{start})+n\times T_{element} \quad (1)$$

- T_{base}：ベクトルデータの先頭アドレスの計算，各種制御情報（ベクトル長，ストライド，など）の設定，などに要する時間．
- MVL：最大許容ベクトル長，レジスタ-レジスタ演算方式では，ベクトル・レジスタ長に等しい．一方，メモリ-メモリ演算方式の場合，ベクトル・レジスタが存在しないので，実際上制限がないに等しい．よって，$\left\lceil\frac{n}{MVL}\right\rceil=1$ となる．
- T_{strip}：ストリップ・マイニングに要する時間．**ストリップ・マイニング** (*strip mining*) とは，元のベクトル長がベクトル・レジスタ長よりも長い場合に行う操作で，ベクトル・レジスタ長をセグメント長としてベクトルのセグメント化を行い，セグメント単位でベクトル演算を繰り返す手法である．メモリ-メモリ演算方式では，ストリップ・マイニングが実際上不要なので，$T_{strip}=0$ となる．
- T_{start}：ベクトル命令のスタートアップ時間．つまり，ベクトル命令の実行を開始してから，結果ベクトルの最初の要素を出力するのに要する時間．
- $T_{element}$：スタートアップ・コストは無視して，結果ベクトルの1個の要素を出力するのに要する時間．

メモリ-メモリ演算方式およびレジスタ-レジスタ演算方式それぞれに関して，式1を書き直すと次のようになる．なお，以下の式で，

- T_s：実効的なスタートアップ・コスト．
- T_e：結果ベクトルの1個の要素を出力するのに要する実効的な時間．

である．

1. メモリ-メモリ演算方式：

$$T(n)^{MM}=T^{MM}_{base}+T^{MM}_{start}+n\times T^{MM}_{element}=T^{MM}_{s}+n\times T^{MM}_{e} \quad (2)$$

$$T^{MM}_{s}=T^{MM}_{base}+T^{MM}_{start} \quad (3)$$

$$T^{MM}_{e}=T^{MM}_{element} \quad (4)$$

2. レジスタ-レジスタ演算方式：

$$T(n)^{RR}=T^{RR}_{base}+\left\lceil\frac{n}{MVL}\right\rceil\times(T^{RR}_{strip}+T^{RR}_{start})+n\times T^{RR}_{element}$$
$$\approx T^{RR}_{base}+\frac{T^{RR}_{strip}+T^{RR}_{start}}{2}+n$$

☆ 両方式を折衷したもので，一つのソース・オペランドと一つのデスティネーション・オペランドはベクトル・レジスタ上に，もう一つのソース・オペランドはメモリ上に置いてベクトル演算を行う**レジスタ-メモリ演算方式**もある．IBM 3090 VF[24] が本方式を採用している．

$$\times\left(\frac{T^{RR}_{strip}+T^{RR}_{start}}{MVL}+T^{RR}_{element}\right)$$

$$=T^{RR}_{s}+n\times T^{RR}_{e} \quad (5)$$

$$T^{RR}_{s}=T^{RR}_{base}+\frac{T^{RR}_{strip}+T^{RR}_{start}}{2} \quad (6)$$

$$T^{RR}_{e}=\frac{T^{RR}_{strip}+T^{RR}_{start}}{MVL}+T^{RR}_{element} \quad (7)$$

CRAY-1 の後継機 CRAY X-MP, CRAY Y-MP, CRAY-2[27] をはじめとして，富士通 VP[10]，日立 S[2),4)]，日電 SX[11]，などの今日のスーパコンピュータのほとんどがレジスタ-レジスタ演算方式を採用している．一方，メモリ-メモリ演算方式は，CYBER-205 の後継機 ETA-10[17] を最後に姿を消した．なぜ，メモリ-メモリ演算方式は，レジスタ-レジスタ演算方式に比べて成功しなかったのか？　以下，その理由を簡単にみてみよう．

1. 必要とするメモリ・バンド幅が大きい：1回の演算に対して通常，3個のオペランド(ソース・オペランド2個とデスティネーション・オペランド1個）が必要である．これは，メモリ-メモリ演算方式においては，1回のベクトル演算当り3回のメモリ・アクセス（2回のベクトル読出しと1回のベクトル書込み）を行うことを要求する．したがって，実効演算スループットは，$\frac{\text{ピーク・メモリ・バンド幅}}{3}$で抑えられる．一方，レジスタ-レジスタ演算方式では，ベクトル演算に先だってベクトルデータをメモリからベクトル・レジスタにロードしておき，すべての演算はレジスタ-レジスタ間で行う．中間結果はベクトル・レジスタ上に置いておき，最終結果のみをメモリにストアする．また，1個のソース・オペランドを複数の演算で用いることも可能である．このように，レジスタ-レジスタ演算方式ではメモリ・トラフィックを減少させ，実効演算スループットを$\frac{\text{ピーク・メモリ・バンド幅}}{3}$より大きくすることが可能となる．上記のことを換言すれば，所望の**実効演算スループット**（ST: Sustained Throughput）が与えられた場合にシステムに要求される**要求メモリ・バンド幅**（MB: Memory Bandwidth）が，メモリ-メモリ演算方式（MB^{MM}）とレジスタ-レジスタ演算方式（MB^{RR}）とでは，下式のように異なる．

$$ST\leq MB^{RR}\leq ST\times 3\leq MB^{MM}$$

つまり，メモリ-メモリ演算方式は，レジスタ-レジスタ演算方式以上に高いメモリ・バンド幅を必要とする．

2. スタートアップ・コストが大きい：メモリ-メモリ演算方式のスタートアップ・コスト T^{MM}_{s}（式3）は通常，レジスタ-レジスタ演算方式のスタートアップ・コスト T^{RR}_{s}（式6）よりも大きい．これは主に，ベクトル演算ごとにソース・オペランドをメモリから読み出さなければならないため，各ベクトル命令のスタートアップ時間 T^{MM}_{start} が T^{RR}_{start} より大きくなるためである[21]．

3. チェイニングが難しい：チェイニング（*chaining*）とは，フロー依存関係にある二つのベクトル命令（たとえば，ベクトル・ロード→ベクトル演算，ベクトル演算→ベクトル演算，ベクトル演算→ベクトル・ストア）をオーバラップ実行することである[7]．一般に，メモリ-メモリ演算方式ではチェイニングは不可能である☆．これは，メモリ上のベクトルデータに関するフロー依存関係を動的に解析するのが，きわめて困難なことによる．一方，レジスタ-レジスタ演算方式の場合，フロー依存関係の解析対象はベクトル・レジスタであり，実行時でも容易に解析が行える．つまり，ベクトル・レジスタを介した暗黙的なチェイニングが容易に実施できる．このチェイニングにより，レジスタ-レジスタ演算方式の T^{RR}_{e}（式7）はメモリ-メモリ演算方式の T^{MM}_{e}（式4）よりも通常小さくなる．

以上の理由により，今日のスーパコンピュータの大部分は，レジスタ-レジスタ演算方式を採用している．本稿では，以下，レジスタ-レジスタ演算方式のみを議論の対象とする．**表-1** に，代表的なレジスタ-レジスタ演算方式のベクトル型スーパコンピュータの諸元を示す．

3. ベクトル・レジスタの構成法

ベクトル・レジスタの構成に当たっては，次の点を考慮に入れる必要がある．

● ベクトル・レジスタとパイプライン・ユニット（演算パイプラインおよびロード／ストア・パ

☆ 専用命令により，2個のベクトル演算を明示的にチェイニングすることは可能である．たとえば，CDC のマシンでは，ベクトル＋スカラ×ベクトル，(ベクトル＋スカラ)×ベクトル，などの3項演算をサポートしている．

表-1　レジスタ-レジスタ演算方式のベクトル型スーパコンピュータの諸元

マシン	ベクトル・レジスタ (要素数×本数)	演算スループット			メモリ・バンド幅				主記憶	
		加算/乗算パイプ数	パイプ当りスループット†	スループット合計†	ロード/ストアパイプ数	パイプ当りバンド幅††	ロード・バンド幅計††	バンド幅合計††	アクセス・レイテンシ†††	バンク数 (最大)
Cray Y-MP	64×8	加算×1 乗算×1	1 1	2	ロード×2 ストア×1	1 1	2	3	17	256
Cray-2	64×8	加算×1 乗算×1	1 1	2	ロード/ストア×1	1	1	1	45	128
富士通 VP-200	32×256～ 1024×8††††	加算×1 乗算×1	2 2	4	ロード/ストア×2	2	4	4	?	256
富士通 VP-2600	64×256～ 2048×8††††	加算/乗算×2	4	8	ロード/ストア×2	4	8	8	?	512
日立 S-810/20	256×32	加算×2 乗算&加算×2	2 2	8	ロード×3 ロード/ストア×1	2 2	8	8	?	?
日立 S-820/80	512×32	加算×1 乗算&加算×1	4 4	8	ロード×1 ロード/ストア×1	4 4	8	8	?	?
NEC SX-2	256×40††††	加算×1 乗算×1	4 4	8	ロード×1 ストア×1	8 4	8	8	?	512
NEC SX-3/14	256×72††††	加算×2 乗算×2	4 4	16	ロード×2 ストア×1	4 4	8	12	?	1024

† 倍精度浮動小数点演算/クロック・サイクル
†† 64ビット倍語/クロック・サイクル
††† クロック・サイクル数
†††† 可変構成

イプライン）とをどのように対応付けるか？　すなわち，いかにして，並列動作する複数のパイプライン・ユニットが各ベクトル・レジスタへ同時アクセスするのを可能にするか？

● ベクトル・レジスタをいくつ備えるか？　ベクトル・レジスタ長（ベクトル・レジスタ当りのベクトル要素数）をいくらにするか？

● チェイニングをベクトル・レジスタ経由で行うか？　もしそうなら，チェイニングは任意のタイミングで可能とするか？

● レジスタ・アクセスがラップアラウンド（wraparound）するのを許すか？☆

図-1 に，標準的なベクトル・レジスタ1個の論理的な構造を示す．レジスタ経由でチェイニングを行わせる場合，通常，読出しポートと書込みポートを1個ずつ，また，読出し用ポインタと書込み用ポインタを1個ずつ備える．

これに対して，ベクトル・レジスタの物理的な構成法としては，**図-2** に示すように，次の二つの代表的な選択肢が存在する[7]．

☆ ラップアラウンド可能にすると，各ベクトル・レジスタはリングFIFOバッファとして動作するので，ストリップ・マイニングが不要となる．

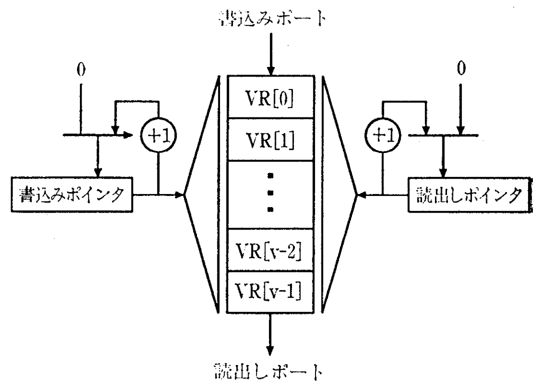
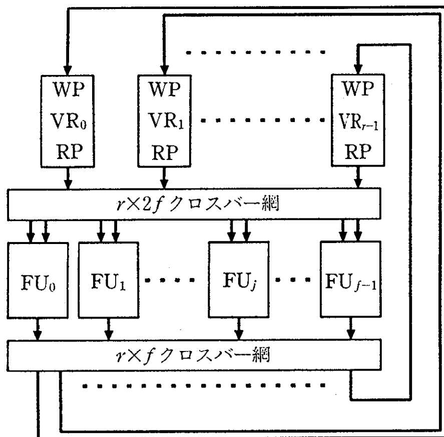


図-1　ベクトル・レジスタの論理構造

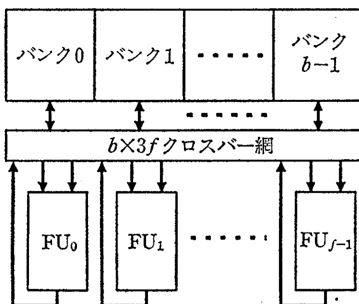
1. 論理-物理一致型（**図-2**(a)）；ベクトル・レジスタの論理構造をそのまま物理構造に反映させた構成とする．アーキテクチャ上 r 個のベクトル・レジスタが定義されている場合，ハードウェアとしても r 個の個々に完全に独立したベクトル・レジスタを設ける．各レジスタは論理構造にならい，読出しポート（読出し用ポインタを含む）と書込みポート（書込み用ポインタを含む）を1個ずつ備えたデュアルポートRAMとなる．

レジスタ長は，一般に固定長である (Cray-1 では64, S-810 では 256). f 個のパイプライン・ユニットが任意のレジスタに対して同時にアクセス可能であるためには，ソース・ベクトル読出し用に r-to-1 のマルチプレクサ $2f$ 個 ($r\times 2f$ のクロスバー網に相当)，結果ベクトル書込み用に f-to-1 のマルチプレクサ r 個($r\times f$ のクロスバー網に相当) がそれぞれ必要である (全体としては，$r\times f$ のクロスバー網3個に相当する). 複数のベクトル命令が一時期に同一のベクトル・レジスタに対して読出し (あるいは，書込み) を行うことを通常許していないので，レジスタ・コンフリクト (正確には，ポート・コンフリクト) が発生しない.

2. マルチバンク・メモリ型 (図-2(b)): ベクトル・レジスタの論理構造とは無関係に，ベクトル・レジスタ全体を1個のマルチバンク・メモリ (**4.** 参照) として構成する. パイプライン・ユニットごとに，2個の読出しポート (読出し用ポインタを含む) と1個の書込みポート (書込み用ポインタを含む) を備える. 各バンクは，シングルポート RAM でよい. バンク数 b は，ベクトル・レジスタ数 r とは独立に決めることができる. 逆に言えば，バンク数 b は固定されているが，レジスタ数 r とレジスタ長 l は「$r\times l=$レジスタ容量」を満たす範囲内で可変とすることが可能である (VP-200 では，$r\times l=8\times 1024=16\times 512=32\times 256=64\times 128=128\times 64=256\times 32$). f 個のパイプライン・ユニットが任意のバンクに対して同時にアクセス可能であるためには，$3f$-to-1 のマルチプレクサが b 個必要である (全体としては，$b\times f$ のクロスバー網3個に相当する). ただし，複数のポートが同一バンクに対して同時にアクセス要求を出し得るので，バンク・コンフリクトの可能性がある.



(a) 論理-物理一致型



FU：機能ユニット
VR：ベクトル・レジスタ
RP：読出しポート
WP：書込みポート
(b) バンク・メモリ型

図-2 ベクトル・レジスタの構成法

4. 主記憶の構成法

ロード／ストア・パイプラインが1クロック・サイクル当りにメモリとの間で転送可能なデータ量，すなわちピーク・メモリ・バンド幅は，

ロード／ストア・パイプライン本数
×パイプライン多重度

で与えられる (表-1 参照). パイプライン多重度は，パイプライン当りのバンド幅に相当する. このピーク・メモリ・バンド幅を維持するには，メモリはそれに見合った速度でデータを読み書きできねばならない. これは，通常，複数のメモリ・バンク (*memory bank*) を備えたマルチバンク・メモリ (*multibank memory*) 構成とすることで実現する. 各バンクは小容量の独立したメモリであり，他のバンクと並列に異なるアドレスへアクセスすることができる.

マルチバンク・メモリの構成に当たっては，次の点を考慮に入れる必要がある.

- バンクをいくつ設けるか？
- アドレス写像をどのように行うか？ すなわち，メモリ・アドレスと，バンク番号 i とバンク内オフセット j の組 (i, j) とをどのように対応付けるか？
- バンクへのアクセスを全バンク同時に同期して行うか，それとも，各バンクごとに独立に行うか？

4.1 バンク数

ピーク・メモリ・バンド幅 MB を維持するためには，バンク数 B は下式を満足しなければならない．

$$B \geq \frac{MB \times T_c}{BW} \quad (8)$$

ここで，T_c はメモリ・サイクル時間（クロック・サイクル），BW は各バンクのデータ幅（bank width）である．

なぜ上記の関係が成立するのか？ その理由を，ピーク・メモリ・バンド幅 MB がクロック・サイクル当り1語（すなわち，ロード／ストア・パイプラインが1本で，その多重度もクロック・サイクル当り1語）で，バンク幅 BW も1語（通常，8バイト）であるような単純な場合でみてみよう．このとき，上式は次のように簡単化できる．

バンク数 $B \geq$ メモリ・サイクル時間 T_c

アドレスが連続するデータを $n(\gg B)$ 語ロードしよう．ある1個のバンクに注目する．1語／クロック・サイクルのバンド幅を維持しようとした場合，当該バンクは B クロック・サイクルごとにデータ1語を読み出せねばならない．そのためには，当該バンクのメモリ・サイクル時間 T_c はバンク・アクセス周期 B 以下でなければならない．

次に述べるアドレス写像を簡略化するため，通常は，バンク数 B を2のべき乗とする（ただし，**4.4** で述べるプライム・メモリのような例外もある）．

4.2 インタリービング

メモリ・アドレスと，バンク番号 i とバンク内オフセット j の組 (i, j) との間の対応の付け方には，ここで述べる**インタリービング**（*interleaving*）方式と，**4.4** で述べる**スキューイング**（*skewing*）方式の二つがある．

インタリーブ方式では，メモリのバイト・アドレス $a(0 \leq a \leq B \times M-1)$ と，バンク番号 $i(0 \leq i \leq B-1)$ とバンク内バイト・オフセット $j(0 \leq j \leq M-1)$ の組 (i, j) とを次のように対応付ける（**図-3** 参照）．ただし，バンク数を B 個，バンク・サイズを M バイト，インタリーブ幅（チャンク・サイズ）を $C(1 \leq C \leq M)$ バイトとする．

$$a = (i + u \times B) \times C + v \quad (9)$$

$$i = \left\lfloor \frac{a}{C} \right\rfloor \bmod B \quad (10)$$

$$u = \left\lfloor \frac{\left\lfloor \frac{a}{C} \right\rfloor}{B} \right\rfloor$$

$$v = a \bmod C$$

$$j = u \times C + v \quad (11)$$

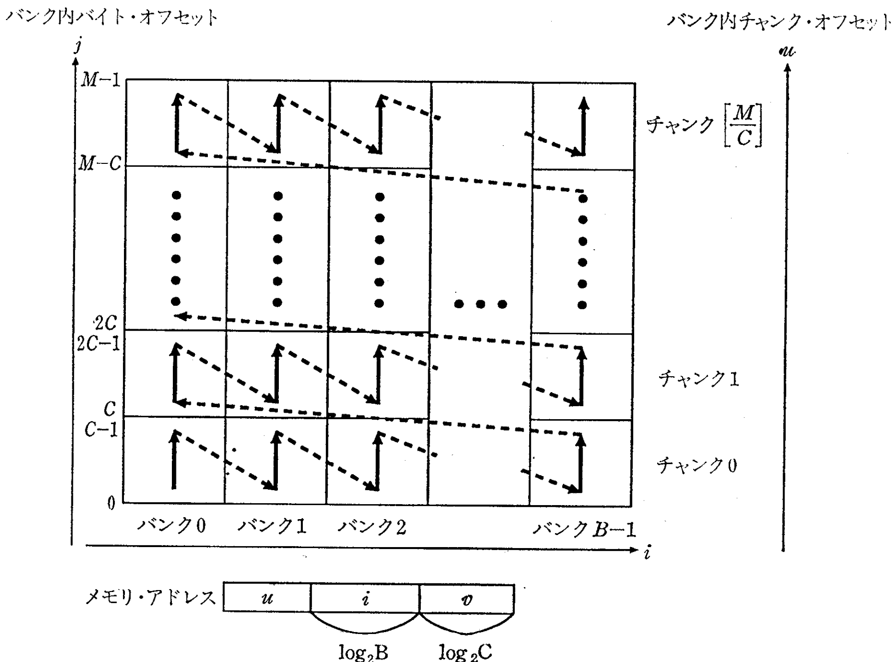


図-3 インタリーブド・メモリ

u はバンク内チャンク・オフセット $(0 \leq u \leq \left\lfloor \frac{M}{C} \right\rfloor)$, v はチャンク内バイト・オフセット $(0 \leq v \leq C-1)$ である.

インタリーブ方式のマルチバンク・メモリをインタリーブド・メモリ (*interleaved memory*) と呼ぶ. このとき, バンク数 B のことをウェイ (way) 数と呼ぶこともある. さて, インタリーブ幅 C の設定次第で種々のインタリーブド・メモリの構成が可能だが, 通常は, C を基本アクセス単位である語長(たとえば, 8バイト)とする**ワード・インタリーブド・メモリ** (*word-interleaved memory*) 構成を採る. これは, C が語長より小さいと (極端な例だと, $C=1$ のバイト・インタリーブド・メモリの場合) 1語をアクセスするのに複数バンクを同時にアクセスする必要が生じるし, 逆に, 語長より大きいと (極端な例だと, $C=M$ の場合) 連続する語をアクセスする際にバンク・コンフリクトを必ず起こすからである.

ワード・インタリーブド・メモリの場合, バイト・アドレス a およびバンク内バイト・オフセット j をそれぞれ語アドレスおよびバンク内語オフセットとみなせば, 式 9〜11 は次のように簡単になる.

$$a = i + j \times B \quad (12)$$

$$i = a \bmod B \quad (13)$$

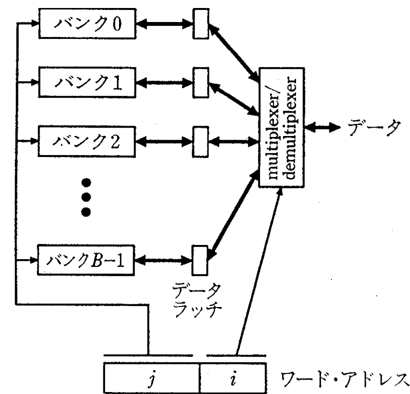
$$j = \left\lfloor \frac{a}{B} \right\rfloor \quad (14)$$

4.3 バンク・アクセス法

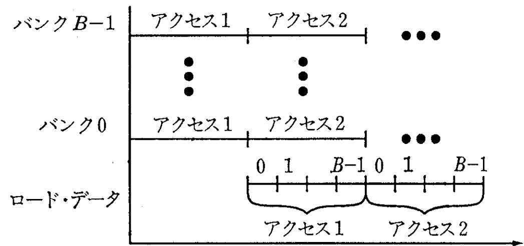
マルチバンク・メモリへのアクセス方法には, 大きく分けて次の2種類がある[22]. なお, ワード・インタリーブド・メモリを仮定する.

1. 全バンク一斉アクセス法 (**図-4**(a)): すべてのバンクを同期して一斉にアクセスする. メモリ・アドレスの上位アドレスがバンク内ワード・オフセットとして, すべてのバンクで用いられる. データは各バンクにラッチしておき, データ転送をしている間に次のアクセスを開始可能とする. **図-4**(b) に, ロード・アクセスの際のタイミングを示す.

2. 各バンク独立アクセス法 (**図-5**(a)): バンクへのアクセスを各バンク独立に行う. データはラッチせず, データ転送は一時に1語ずつ行

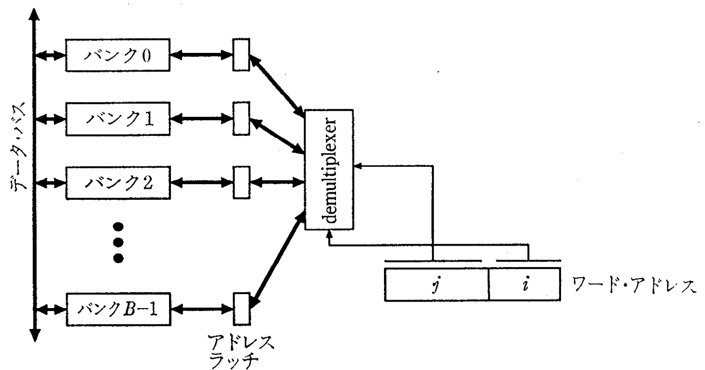


(a) 構成

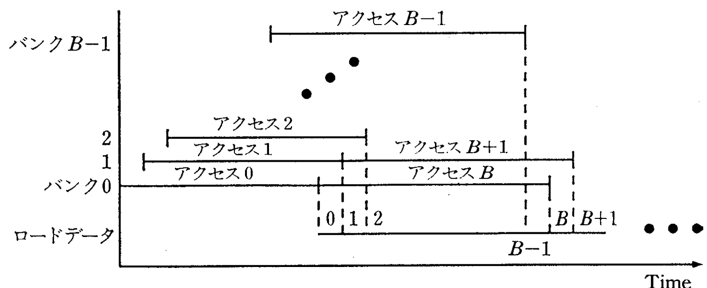


(b) タイミング (ロード・アクセス)

図-4 全バンク一斉アクセス法



(a) 構成



(b) タイミング (ロード・アクセス)

図-5 各バンク独立アクセス法

う．各バンクは，データを転送すると，次のアクセスをただちに開始する．この間，各バンクではアドレスをラッチしておく．**図-5**(b) に，ロード・アクセスの際のタイミングを示す．

全バンク一斉アクセス法では，不連続アクセスの場合，あるいは，連続アクセスでも開始アドレスがバンク0に揃っていない場合，不要なデータへのアクセスを頻発し効率が悪い．また，各バンク独立アクセス法では，データ・バスが込んでいると，各バンクのアクセスが完了していても次のアクセスが開始できないという問題がある．そこで，現実には，両方法のハイブリッド法，すなわち，各バンク独立アクセス法において各バンクにデータ・ラッチを設けることが多い．

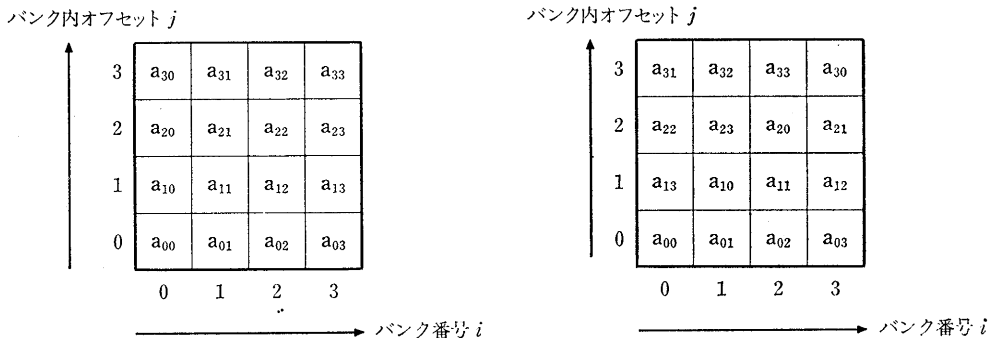
4.4 コンフリクトフリー・メモリ

これまでに述べたインタリーブド・メモリでは，ロード／ストア・パイプラインが1本で，かつ，アクセスが連続したアドレスに対して行われる場合，バンク・コンフリクトが生じない．しかし，ロード／ストア・パイプラインが1本の場合でも，アクセス・アドレスが連続していないとバンク・コンフリクトが発生する．この様子を**図-6** (a) で見てみよう．図のメモリは4バンク構成で，4×4 の行列が行優先 (row-major) で格納されている．このとき，同一行要素および対角要素へのアクセスはバンク・コンフリクトなしで行えるが，同一列要素はすべて同一バンクに格納されているのでアクセスのたびに毎回バンク・コンフリクトが発生する．

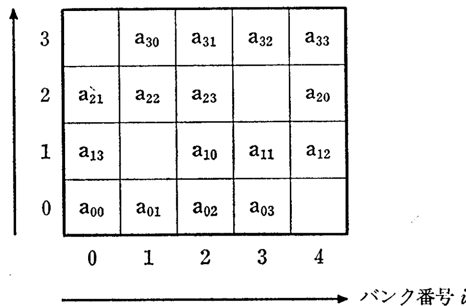
スキューイング (*skewing*) 方式は，$N \times N$ 行列のある部分（同一行，同一列，対角要素，部分行列，など）に対するアクセスがバンク・コンフリクトなしで行えるように，行列要素 $a_{s,t}(0 \leq s, t \leq N-1)$ とバンク番号 $i(0 \leq i \leq B-1)$ とを対応付ける方法である[15]．スキューイング方式のマルチバンク・メモリをスキュード・メモリ (*skewed memory*) と呼ぶ．これには，以下の分類項目に応じて種々の方式が提案されている．

● 周期的 (periodic)/非周期的 (aperiodic)：ある一定のアドレス周期ごとに同一バンクへアクセスするような対応付けを行う方式を周期的スキューイングと呼ぶ[26]．

● 線形 (linear)/非線形 (nonlinear)：バンク番号 i が $i = p \cdot s + q \cdot t \pmod{B}$ で与えられるような周期



(a) インタリーブド・メモリ ($i=t$)　　(b) スキュード・メモリ ($i=s+t \pmod 4$)



(c) プライム・メモリ ($i=2s+t \pmod 5$)

図-6 コンフリクトフリー・メモリ

的スキューイングを線形スキューイング（**図-6**（b）参照）と呼ぶ[15]．そうでないものを非線形スキューイング[26]，または，スクランブルド・メモリ（*scrambled memory*）[14]と呼ぶ．

● バンク数 B：線形スキューイングでは，B, p. q の定め方でコンフリクト・フリーでアクセス可能な行列部分が決まる．よく知られているのは，バンク数 B を N より大きい素数とする**プライム・メモリ**（*prime memory*：**図-6**（c）参照）で，行，列，対角要素，$\sqrt{N}\times\sqrt{N}$ の部分行列のいずれに対してもコンフリクト・フリーでアクセス可能である[15]．

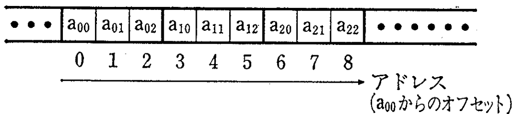
5. 主記憶アクセス方式

ベクトル演算の対象となるベクトル・データは，メモリに格納されている多次元配列のある要素（同一行要素，同一列要素，対角要素，部分行列要素，任意要素，など）の集合体である．ところで，多次元配列（**図-7**（a）参照）をメモリに格納するには，それを1次元に並び換えて，行優先（row-major）あるいは列優先（column-major）のいずれかの順序で配置する必要がある．

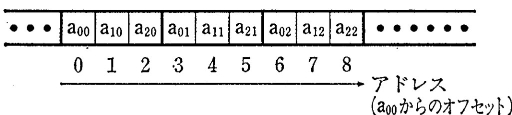
● **行優先配置**（**図-7**（b）参照）：FORTRAN を除く大部分のプログラミング言語で採用されている．2次元配列の場合，要素 $a_{s,t}$ と要素 $a_{s,t+1}$ とがメモリ上で隣接するように行を優先して配列を配置する．

a_{00}	a_{01}	a_{02}
a_{10}	a_{11}	a_{12}
a_{20}	a_{21}	a_{22}

（a） 2次元 3×3 配列



（b） 行優先配置



（c） 列優先配置

図-7 2次元配列のメモリ内配置

● **列優先配置**（**図-7**（c）参照）：FORTRAN で採用されている．2次元配列の場合，要素 $a_{i,j}$ と要素 $a_{i+1,j}$ とがメモリ上で隣接するように配列を配置する．

それでは，上記のように配置された配列要素からどのようにしてベクトル・データを形成するのであろうか？ 次の（FORTRAN ふうに記述した）例題を見てみよう．

```
do 10 i=1,64
  do 10 j=1,64
    do 10 k=1,64
10          A(IX(k),j)=A(i,IX(k))
                +B(i,k)*C(k,j)
```

上記コードの最内ループにおいては，行列Bの行と行列Cの列との乗算，および，その乗算結果と行列Aの列との加算がベクトル化可能である．つまり，行列Bの第 i 行 B(i, k) ($1\leq k\leq 64$)，行列Cの第 j 列 C(k, j) ($1\leq k\leq 64$)，および，行列Aの第 j 列 A(IX(k), j) ($1\leq k$, IX(k)≤ 64) が，それぞれベクトル $V_B(k)$, $V_C(k)$, $V_A(k)$ ($1\leq k\leq 64$) ということになる．これらのベクトル・データを形成するには，メモリ上の配列要素を以下のアクセス方式によりベクトル・レジスタにロードしてくる必要がある．なお，各行列はメモリに列優先配置されているものとする．

1. 連続アクセス（*unit-stride access*）：**図-8**（a）に示すように，行列Cの第 j 列要素からなるベクトル V_C の隣接する要素 $V_C(k)$=C(k, j) と $V_C(k+1)$ =C(k+1, j) は，メモリ上でも隣接している．よって，メモリ上の配列要素を単に連続してアクセスすればよい．これを連続アクセス方式といい，ベクトル要素 $V_c(k)$ のメモリ・アドレス（語アドレス）a_k を下式で与える．

$$a_k=a_1+(k-1)$$

2. 非連続アクセス（*nonunit-stride access*）：**図-8**（b）に示すように，行列 B の第 i 行要素からなるベクトル V_B の隣接する要素 $V_B(k)$=B(i, k) と $V_B(k+1)$=B(i, k+1) は，メモリ上では隣接していない．隣接ベクトル要素間のメモリにおける距離をストライド（*stride*）と呼ぶ．ベクトル V_B のストライドは64である．このようにストライドが1より大きい場合（nonunit-stride），メモリ上の配列要素を非連続かつ等間隔でアクセスする必要がある．これを非連続アクセス（または，等間

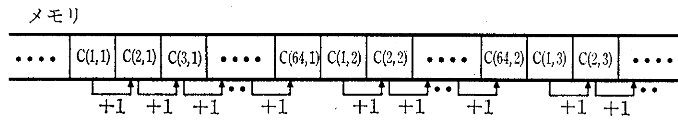
隔アクセス，ストライド付きアクセス)方式といい，ベクトル要素 $V_B(k)$ のメモリ・アドレス(語アドレス) a_k を下式で与える．

$$a_k = a_1 + (k-1) \times \text{ストライド}$$

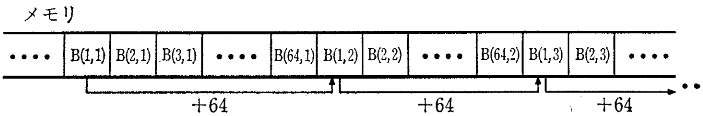
なお，先のベクトル V_C のストライドは1であった．すなわち，連続アクセス方式は本方式におけるストライド＝1 (unit-stride) の場合に相当する．

3. インデックス付きアクセス (*indexed access*)：行列Aの第 *j* 列要素 A(IX(k), j) とベクトル要素 $V_A(k)$ との対応関係は，インデックス・ベクトル (index vector) $V_{IX}(k)$ $(1 \leq k \leq 64)$ で与えられる．よって，メモリ上の配列要素へのアクセスは**図-8** (c) に示すように，インデックス・ベクトル $V_{IX}(k)$ を介して間接的に行う．これをインデックス付きアクセスといい，ベクトル要素 $V_A(k)$ のメモリ・アドレス (語アドレス) a_k を下式で与える．

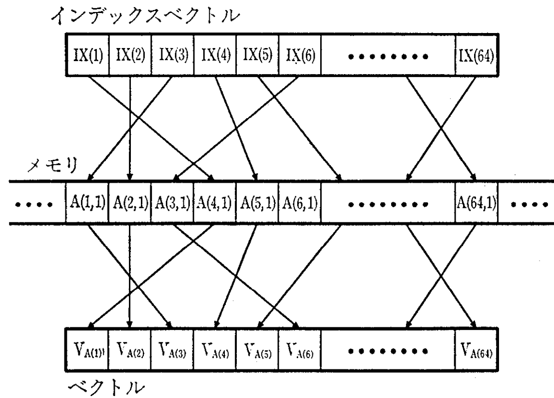
$$a_k = a_1 + (V_{IX}(k) - 1)$$



(a) 連続アクセス (ストライド=1)



(b) 非連続アクセス (ストライド=64)



(c) インデックス付きアクセス

図-8 メモリ・アクセス方式

大部分のベクトル・プロセッサは，これら3種のアクセス方式をすべてサポートしている．特に，インデックス付きアクセス方式は，条件付きベクトル演算や疎行列計算を実現するうえで重要な働きをする．また，**4.1** で，

バンク数＜メモリ・サイクル時間

のとき連続アクセスを行うとバンク・コンフリクトが起きることを示した．非連続アクセスは連続アクセスに比べてバンク・コンフリクトを起こしやすいので，バンク数の決定には注意が必要である．すなわち，非連続アクセスでは，

$$\frac{\text{バンク数とストライドの最小公倍数}}{\text{ストライド}} < \text{メモリ・サイクル時間}$$

のときバンク・コンフリクトを起こすことになる．

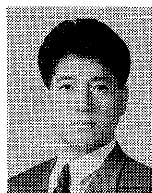
6. おわりに

以上，ベクトル型スーパコンピュータのベクトル・レジスタと主記憶に関して，アーキテクチャおよび構成方式を中心に述べた (拡張記憶，実装技術，などについては紙面の都合で割愛した)．その特徴を一言で言うと，「メモリ性能にベクトル処理性能が大きく依存している」ということである．今日のスーパコンピュータは，プロセッサとメモリ (主記憶) との間に大量のハードウェアを惜しみなくつぎこんでいる．しかし，近年のダウンサイジングの波はベクトル・プロセッサにまで押し寄せ，遂にシングル・チップ構成のベクトル・プロセッサ (**マイクロベクトルプロセッサ**[12]) が登場するに至っている．このようなマイクロベクトルプロセッサにおいては，プロセッサとメモリとの間に，今日のスーパコンピュータのような大量のハードウェアをつぎこむことはできない．あるのは，普通のマイクロプロセッサ同様，数百本の入出力ピンだけである．**ピン・ボトルネック**の存在は歴然としている．今後のベクトル・プロセッサ技術は，このピン・ボトルネックをいかに解消するかを中心に発達していくものと予想する．

参 考 文 献

1) 内田啓一郎：スーパコンピュータのハードウェア，電子情報通信学会誌，Vol. 75, No. 2, pp. 114-119 (1992).
2) 小高，小林，河辺，長島：最大性能が630 MFLOPSで1Gバイトの半導体拡張記憶が付くスーパコンピュータ HITACS-810，日経エレクトロニクス，No. 314, pp. 159-184 (1983).
3) 唐木幸比古（編）：特集/スーパーコンピュータの現在，コンピュートロール，第20号 (1987).
4) 河辺，小林，村山，桐生，半田，田上，後藤，青山：シングル・プロセッサで最大性能 2 GFLOPSのS-820，日経エレクトロニクス，No. 437, pp. 111-125 (1987).
5) 島崎眞昭：スーパーコンピュータとプログラミング，共立出版 (1989).
6) 富田眞治：並列計算機構成論，第6章，pp. 143-194 (1986).
7) 長島，田中：スーパコンピュータ，オーム社 (1992).
8) 名取，野寺（編）：スーパーコンピュータと大型数値計算，共立出版 (1987).
9) 日本物理学会（編）：スーパーコンピュータ，培風館 (1985).
10) 平栗，田畑，槌本，田口：マシン・サイクル 7.5 ns を達成した並列パイプライン処理方式のスーパコンピュータ FACOM VP，日経エレクトロニクス，No. 314, pp. 131-155 (1983).
11) 古勝，渡辺，近藤：最大性能 1.3 GFLOPS，マシン，サイクル 6 ns のスーパコンピュータ SX システム，日経エレクトロニクス，No. 356, pp. 237-272 (1984).
12) 村上，橋本，弘中，安浦：マイクロベクトルプロセッサ・アーキテクチャの検討，情報処理学会研究会報告，ARC-94-3 (1992).
13) 村田，小国，唐木（編）：スーパーコンピューター科学技術計算への適用一，pp. 1-66，丸善 (1985).
14) Batcher, K. E.: The Multidimensional Access Memory in STARAN, *IEEE Trans. Comput.*, Vol. C-26, No. 2, pp. 174-177 (1977).
15) Budnik, P. and Kuck, D. J.: The Organization and Use of Parallel Memories, *IEEE Trans. Comput.*, Vol. C-20, No. 12, pp. 1566-1569 (1971).
16) Ercegovac, M. D. and Lang, T.: ベクトル処理，スーパーコンピュータ (Fernbach, S.（編），長島（訳）)，pp. 37-74，パーソナルメディア (1988).
17) Fazio, D.: It's Really Much More Fun Building a Supercomputer Than It Is Simply Inventing One, *COMPCON, IEEE*, pp. 102-105 (1987).
18) Fernbach, S.（編）：スーパーコンピュータ（長島（訳））：パーソナルメディア (1988).
19) Hennessy, J. L. and Patterson, D. A.: *Computer Architecture: A Quantitative Approach*, Chap. 7, Morgan Kaufmann Publishers, Inc. (1990)；富田，村上，新實（訳）：ヘネシー＆パターソン コンピュータ・アーキテクチャー設計・実現・評価の定量的アプローチー，第7章，日経 BP 社 (1992).
20) Hintz, R. G. and Tate, D. P.: Control Data STAR-100 Processor Design, *COMPCON, IEEE*, pp. 1-4 (1972).
21) Hockney, R. W. and Jesshop, C. R.: *Parallel Computers 2: Architecture, Programming and Algorithms*, Adam Hilger (1988).
22) Hwang, K. and Briggs, F. A.: *Computer Architecture and Parallel Processing*, McGraw-Hill (1984).
23) Lincoln, N. R.: Technology and Design Trade Offs in the Creation of a Modern Supercomputer, *IEEE Trans. Comput.* Vol. C-31, No. 5, pp. 363-376 (1982).
24) Padegs, A., Moore, B. B., Smith, R. M. and Buchholz, W.: The IBM System/370 Vector Architecture: Design Considerations, *IEEE Trans. Comput.* Vol. 37, No. 5, pp. 509-520 (1988).
25) Russell, R. M.: The CRAY-1 Computer System, *Comm. of the ACM*, Vol. 21, No. 1, pp. 63-72 (1978).
26) Shapiro, H. D.: Theoretical Limitations on the Efficient Use of Parallel Memories, *IEEE Trans. Comput.*, Vol. C-27, No. 5, pp. 421-428 (1978).
27) Thompson, J. R.（編）：CRAY-1, CRAY X-MP, CRAY-2 とその将来，Cray Research のスーパーコンピュータ，スーパーコンピュータ (Fernbach, S.（編），長島（訳）)，pp. 89-116，パーソナルメディア (1988).
28) Watson: W. J.: The TI ASC—A Highly Modular and Flexible Super Computer Architecture, *Proc. AFIPS Fall Joint Computer Conf.*, pp. 221-228 (1972).

（平成4年12月14日受付）



村上　和彰（正会員）

1960年生．1982年京都大学工学部情報工学科卒業．1984年同大学院修士課程修了．同年富士通(株)本体事業部に入社，汎用計算機Mシリーズのアーキテクチャ開発に従事．1987年九州大学工学部助手，1992年同大学院総合理工学研究科講師，現在に至る．計算機アーキテクチャ，並列処理，性能評価などの研究に従事．著書「計算機システム工学（共著）」，「ヘネシー＆パターソン：コンピュータ・アーキテクチャ（共訳）」，本学会研究賞（平成3年度），本学会論文賞（平成3年度）受賞．電子情報通信学会，日本応用数理学会，ACM，IEEE，IEEE-CS 各会員．