計算機アーキテクチャ 101−12
（1993. 8. 20）

ハイパースカラ・プロセッサ・アーキテクチャ
— 実現上の課題 —

斎藤靖彦　　村上和彰

九州大学 大学院総合理工学研究科 情報システム学専攻

〒816 福岡県春日市春日公園 6-1

E-mail: {saitoh, murakami}@is.kyushu-u.ac.jp

命令レベル並列性を活用するハイパースカラ・プロセッサの実現に際して，いくつか解決すべき課題がある．特に，ハードウェア・コストを低く抑え，なおかつ性能的に遜色のないレジスタ・ファイルを構成することは重要である．ハイパースカラ・プロセッサでは，擬似ベクトル処理またはソフトウェア・パイプライニングにより科学技術計算のループに内在する命令レベル並列性を活用する．そのためには，ベクトル・レジスタを備えることが望ましい．ところが，従来のベクトル・プロセッサで用いられているベクトル・レジスタはベクトル処理を行うことを前提に構成されたものであり，擬似ベクトル処理およびソフトウェア・パイプライニング向きではない．よって，新たにハイパースカラ・プロセッサ向きの構成を検討する必要がある．本稿では，ベクトル・レジスタ構成の種々の選択肢を挙げ，ソフトウェア・パイプライニングとの適応性を調べる．

Hyperscalar Processor Architecture
— Implementation Issues —

Yasuhiko Saitoh　Kazuaki Murakami

Department of Information Systems
Interdisciplinary Graduate School of Engineering Sciences
Kyushu University
Kasuga-shi, Fukuoka 816 Japan

E-mail: {saitoh, murakami}@is.kyushu-u.ac.jp

Some implementation issues must be solved on implementing hyperscalar processors, which exploit instruction-level parallelism by self-creating VLIW instructions in its instruction registers. Especially, one of the most important implementation issue is how to implement high-performance register-files at low hardware-costs. In hyperscalar processors, loops are executed by pseudovector processing and software-pipelining. So it is very effective to implement vector-registers. Such vector register that are provided in conventional vector processors are not good for the use in hyperscalar processors. This is because of the different constrains given by the pseudovectorization and software-pipelining. So we need to consider the design issues of vector-registers for hyperscalar processors. This paper presents some possible alternatives for implementing vector-registers, and evaluates its adaptability to software-pipelining.

1 はじめに

ハイパースカラ・プロセッサ・アーキテクチャ(hyper-scalar processor architecture)は，スーパースカラ方式，VLIW方式，およびベクトル処理方式のそれぞれの長所を包含するアーキテクチャである[1].

ハイパースカラ方式とは，簡約すれば，

- 命令長および命令フェッチ巾はスーパースカラ方式と同程度だが，
- 機能ユニット対応に(1個以上の)ユーザ可視の命令レジスタを設け，それに（解読済みの）命令をロードすることでVLIWプログラムをプロセッサ内部に自己形成し，あたかもVLIWプロセッサの如く振舞い，
- さらに，ベクトル・レジスタを設け，自己形成したVLIW命令のループにより，ベクトル・データに対して擬似ベクトル処理(ベクトル命令の処理内容をスカラ/VLIW命令のループで模擬する)あるいはソフトウェア・パイプライニング処理を施す，

ことを目的としたプロセッサ・アーキテクチャである.

ハイパースカラ・プロセッサの実現に際して，解決しなくてはならない課題がいくつか残っている．レジスタ・ファイルの大容量化への対策も重要な課題の1つである．ハイパースカラ・プロセッサにおいて擬似ベクトル処理やソフトウェア・パイプライニングを効果的に行うためには，大容量かつ十分なポート数を備えたレジスタ・ファイル(特にベクトル・レジスタ)を備えることが望ましい．しかし，そのようなベクトル・レジスタは極めて高いハードウェア・コストを必要とする．将来，ハイパースカラ・プロセッサのマイクロプロセッサ化を目指す上では，ハードウェア・コストを抑えつつ，性能的に見劣りしないベクトル・レジスタを構成することが重要である.

本稿では，ベクトル・レジスタの構成法について検討する．まず，2章でハイパースカラ方式の概要を述べる．3章で大容量レジスタ・ファイルの実装における課題を提示し，4章でベクトル・レジスタの構成法，および各構成法がソフトウェア・パイプライニングに与える影響を調べる．最後に，5章で今後の課題について述べる.

2 ハイパースカラ方式

ハイパースカラ方式は，従来の命令レベル並列処理の長所を取り入れ短所を解決する.

2.1 命令レジスタ

図1に，ハイパースカラ・プロセッサの基本構成例を示す.

ハイパースカラ・プロセッサの構成は，基本的にスーパースカラ・プロセッサと変わらない．命令長および命令フェッチ巾は，スーパースカラ方式と同程度とする．たとえば，命令長は32ビットで命令フェッチ巾は2〜4命令程度で構わない．これより，VLIW方式の短所であるコード・サイズの増加および命令キャッシュの低使用効率といった問題を解決する.

スーパースカラ方式を含めた他方式との本質的な相違点は，並列動作可能な個々の機能ユニット(FU)毎に，1個以上のユーザ可視の命令レジスタ(IR)を設けた点である．機能ユニット数をf，各機能ユニット当りの命令レジスタ数をrとすると，計$f \times r$個の命令レジスタが存在する.

命令レジスタが構成するアドレス空間は，$f \times r$の2次元配列となる．各命令レジスタを$IR[i,j](0 \leq i \leq r-1, 0 \leq j \leq f-1)$と記す．また，2次元配列の第$i$番目の行全体を$\boldsymbol{IR}^i = \{IR[i,0], IR[i,1], \ldots, IR[i,f-1]\}$と，第$j$番目の列全体を$\boldsymbol{IR}_j = \{IR[0,j], IR[1,j], \ldots, IR[r-1,j]\}$とそれぞれ記す．このとき，各行$\boldsymbol{IR}^i$はあたかも，$f$個のフィールドから成る1個のVLIW命令のように見える．また，命令レジスタ全体では，r命令から成る1個のVLIWプログラムのように見える.

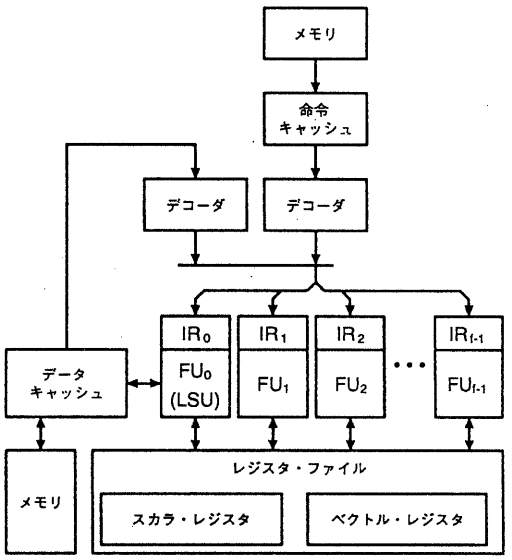


図1: ハイパースカラ・プロセッサの基本構成

命令レジスタ$\boldsymbol{IR}_j$は，対応する機能ユニットFU_jにディスパッチ可能な解読済み命令をr命令まで格納する．これら解読済み命令のロード方法として，次の2方法が考えられる.

- **専念ロード**：専用ロード命令により，メモリ(データ・キャッシュ)から解読前の命令を読み出し，それをデコーダで解読して指定された命令レジスタ$IR[i,j]$に格納する.
- **並行ロード**：一般命令の通常の実行過程において，デコーダで解読した命令を機能ユニットFU_jにディスパッチすると同時に，指定された命令レジスタ$IR[i,j]$に格納する.

上記の2方法のうち，少なくとも専念ロードが備わっていれば十分である.

2.2 動作原理

命令レジスタを用いない場合(これをNormalモードと呼ぶ)の命令実行過程は，一般のスーパースカラ方式とまったく同じである(図2(a)参照)．すなわち，デコーダで解読した各命令を対応する機能ユニットにディスパッチする.

ハイパースカラ方式は，命令レジスタ$\boldsymbol{IR}_j$からも対応する機能ユニットFU_jへ命令をディスパッチすることを可能とする．このディスパッチ方法として，次の2方法が考えられる.

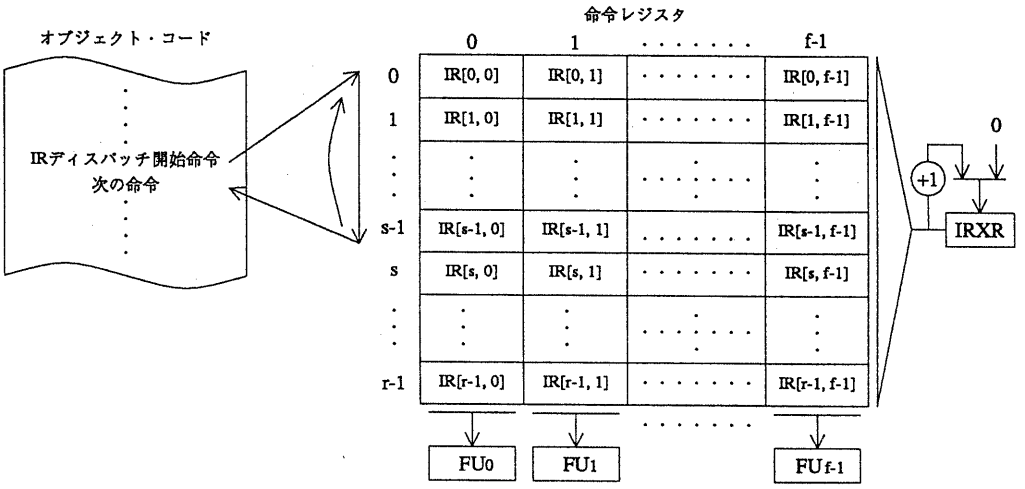
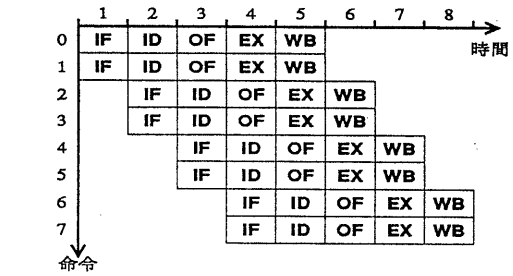
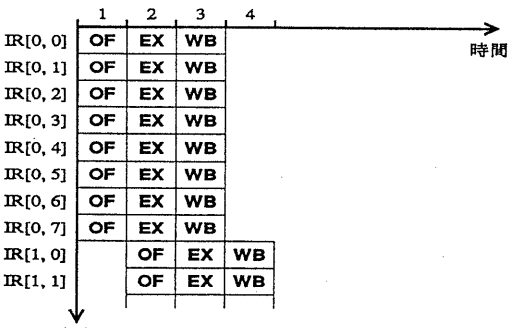


図 3: 専念ディスパッチ



(a)命令レジスタを用いない場合の命令実行過程
(スーパースカラ度 n = 2)



(b)命令レジスタを用いた場合の命令実行過程
(機能ユニット数 $f = 8$)

図 2: 命令実行過程

- **専念ディスパッチ**：IRディスパッチ開始命令により命令レジスタからの命令ディスパッチを開始し，IRディスパッチ終了条件を満たすまで続ける．
- **並行ディスパッチ**：通常の命令実行と並行して，命令レジスタからも命令をディスパッチする．

上記の 2 方法のうち，少なくとも専念ディスパッチが備わっていれば十分である．以下，専念ディスパッチにおけるプロセッサ動作 (この動作モードを **Turbo** モードと呼ぶ) を述べる (図 2(b)，図 3参照)．

①IRディスパッチ開始命令により，IRインデックス・レジスタ $IRXR$を 0 にリセットし，Turbo モードへ移行する．

②Turbo モードの間，プロセッサはあたかも s 命令から成る VLIW プログラム$\boldsymbol{IR}^i = \{IR[i,0], IR[i,1], \ldots, IR[i,f-1]\}(0 \le i \le s-1 \le r-1)$ を実行しているかのように見える．すなわち，毎サイクル，IRインデックス・レジスタ $IRXR$で指定される VLIW 命令$\boldsymbol{IR}^{IRXR}$をすべての機能ユニット FU_jに対してディスパッチする．そして，$IRXR$を+1 インクリメントする．

③任意の VLIW 命令$\boldsymbol{IR}^i$中に，IRディスパッチ終了命令を含むことが出来る．本命令は IRディスパッチ終了条件が成立するか否かを判定し，以下のいずれかの動作を行う．

- 成立する場合：Turbo モードを抜けて，Normal モードへ移行する．制御 (プログラム・カウンタ) は，IRディスパッチ開始命令の次の命令に移る．
- 成立しない場合：そのまま Turbo モードを続行する．続行方向には，次の 2 方向がある．
  - 前方向：$IRXR$を+1 インクリメントする．
  - 後方向：$IRXR$を 0 にリセットする．

このように，

- s 個の VLIW 命令$\boldsymbol{IR}^i(0 \le i \le s-1 \le r-1)$ は『手続き』として，
- IRディスパッチ開始命令は『手続き呼出し』として，また，

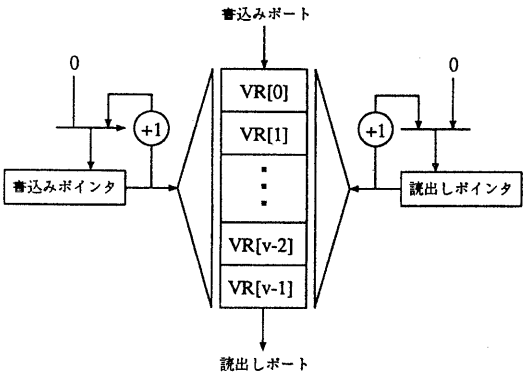


図 4: ベクトル・レジスタの論理構造

- *IR*ディスパッチ終了命令は「手続きからの戻り」および「ループの戻り分岐」として,

それぞれ動作する.

3 大容量レジスタ・ファイル

ハイパースカラ方式の活用方法としては,次の2つの手法が考えられる [1].

- 擬似ベクトル処理
- ソフトウェア・パイプライニング処理

これらの手法の効果を高めるには大容量のレジスタ・ファイルを備えるのが望ましい.

レジスタ・ファイルの論理構造としては一般に次の2種類が用いられる.

- スカラ・レジスタ (SR) : 一次元構造のレジスタ・ファイルであり,レジスタ番号のみでレジスタ内データ (スカラ・データ) を特定する.
- ベクトル・レジスタ (VR) : 二次元構造のレジスタ・ファイルであり,各レジスタは一定のレジスタ長を有する. (レジスタ番号, レジスタ内インデックス) の組によりレジスタ内データ (ベクトル・データの要素) を特定する (図4参照).

また,命令としては,スカラ命令とベクトル命令の2種類がやはり存在する.したがって,どの命令がどのレジスタにアクセス可能とするかで,以下の4通りのアクセス機能が存在する [2].

- **SISR** (*Scalar Instructions – Scalar Registers*) 機能:スカラ命令がスカラ・レジスタにアクセスして演算等を行う.通常のスカラ処理形態.
- **SIVR** (*Scalar Instructions – Vector Registers*) 機能:スカラ命令がベクトル・レジスタにアクセスして演算等を行う.一般には,未活用.
- **VISR** (*Vector Instructions – Scalar Registers*) 機能:ベクトル命令がスカラ・レジスタにアクセスして演算等を行う.従来のベクトル・プロセッサでも使用している.
- **VIVR** (*Vector Instructions – Vector Registers*) 機能:ベクトル命令がベクトル・レジスタにアクセスして演算等を行う.通常のベクトル処理形態.

ハイパースカラ方式の命令セット・アーキテクチャとしてはスカラ命令のみから成っていることから **SISR** および **SIVR** が活用可能なアクセス機能となる.まず,**SISR** は通常のスカラ処理形態であるが,SRを大容量化してベクトル・データを格納することも可能である.ただし,このときのアクセス方法はメモリへのアクセス方法と等価となり,それほど検討の余地はない.よって,本稿では **SIVR** を検討の対象とする.

大容量のVRの実装に際して,以下の2つの課題を解決する必要がある.

- 並列に動作可能なすべてのFUが一つのVRを共有し,同時読出し,および同時書込みができることが理想である.しかし,マルチポートVRは極めて高いハードウェア・コストを必要とする.そこで,ハードウェア・コストを抑えつつ,十分な数のポートを確保し,なおかつ性能的にさほど見劣りしないVRの構成方式を検討する必要がある.
- 大容量のVRを装備した場合,コンテキスト量が多くなる.割込み発生に伴うコンテキスト・スイッチに要するオーバヘッドを短縮,または隠蔽する方法を検討する必要がある.

本稿では,上記2つの課題のうち前者のVRの構成方式について議論する.

4 ベクトル・レジスタの構成法

ベクトル・レジスタ (VR) の構成において,次のような選択肢が存在する.

- メモリ構成
  - –論理-物理一致型 vs 不一致型
  - –R & W 完全共有 vs R/W 完全共有 vs 部分共有 (分割)
- ポインタ構成
  - –各命令専有 vs 各VR専有 vs 全VR共有
  - –暗黙インクリメント vs 明示インクリメント
  - –ラップアラウンド不可 vs 可

以下の節では,上記選択肢の比較,および選択肢がソフトウェア・パイプライニングに与える影響に関する考察を行う.図5に例題プログラム,およびその非ソフトウェア・パイプライニング時の命令コードを示す.仮定したFU構成は,浮動小数点 (FP) 加算器 × 1 , FP乗算器 × 1 , LOAD/STORE × 2 である.

4.1 論理-物理一致型 vs 不一致型

VRの論理構造と物理構造の対応関係について,以下の選択肢がある [4].

- **論理-物理一致型** : 図6(a)のように,VRの論理構造をそのまま物理構造に反映させた構成である.個々のVRは,読出しポートと書込みポートを1個ずつ備えた2ポートRAMで構成される.構造上,レジスタ長は固定長となる.
- **論理-物理不一致型** : VRの論理構造とは無関係に物理構造を決定する.たとえば,図6(b)のように,複数のVRを含むRFを1個のマルチバンク・メモリとして構成する方式 (マルチバンク・メモリ型) がある.各バンクは1ポートRAMで構成される.バンク数は固定である.総レジスタ容量は一定だが,レ

```
DO 1 K = 1, N
X(K) = Q + Y(K)*(R*Z(K+10) + T*(Z+11))
DOEND
```

(a)r例題プログラム

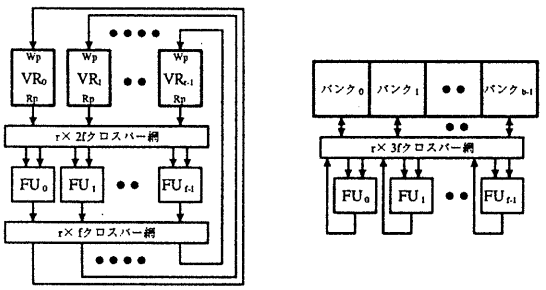
```
LOAD S1, R      LOAD S2, T
LOAD S3, Q      LOAD V0, Z(K+10)      } プロローグ(Normalモード)
```

Turboモード開始

TIME	Load/Store	Load/Store	FP乗算器	FP加算器	整数系
1	LOAD V2, Y(K)	LOAD V1, Z(K+11)	NOP	NOP	NOP
2	NOP	NOP	MUL V3, S1,V0	NOP	NOP
3	NOP	NOP	MUL V4, S2,V1	NOP	NOP
4	NOP	NOP	NOP	NOP	MOVE V0, V1
5	NOP	NOP	NOP	ADD V5, V3,V4	NOP
6	NOP	NOP	NOP	NOP	NOP
7	NOP	NOP	MUL V6,V2,V5	NOP	NOP
8	NOP	NOP	NOP	NOP	NOP
9	NOP	NOP	NOP	ADD V7,S3,V6	NOP
10	NOP	NOP	NOP	NOP	NOP
11	STORE X(K) , V7	NOP	NOP	NOP	NOP

(b)非ソフトウェア・パイプライニング時のコード

図 5: 例題プログラム，およびその非ソフトウェア・パイプライニング時の命令コード



(a) 論理-物理一致型　　(b) 論理-物理不一致型

VR：ベクトル・レジスタ
FU：機能ユニット
r ：ベクトル・レジスタ数
f ：機能ユニット数

図 6: 論理物理一致型 vs 不一致型

ジスタ数とレジスタ長が可変であるような構成にすることが可能である．この構成を取る場合，同時に複数のVRをアクセスした場合にバンク・コンフリクトが発生する可能性がある．

上記 2 つの選択肢がソフトウェア・パイプライニングに与える影響は以下のとおりである．

- 論理-物理一致型では，同一番号のVRから/への読出し/書込み競合 (レジスタ・コンフリクト) を避けるようにコード・スケジューリングする必要がある．このようなレジスタ・コンフリクトの回避はコンパイラにより容易に実現可能である．
- 論理-物理不一致型では，同一番号のバンクから/への読出し/書込み競合 (バンク・コンフリクト) が発生する可能性がある．このようなバンク・コンフリクトを回避するためにはまず，各VRの各要素がメモリバンク上にどのように分布しているのかをコンパイラが認識している必要がある．しかし，与えられた情報を用いてバンク・コンフリクトを回避するソフトウェア・パイプライニングは非常に複雑であり，汎用的なアルゴリズムの考案が難しい．

以上から，ハイパースカラ・プロセッサには論理-物理不一致型はあまり適さず，論理-物理一致型のほうが望ましいと言える．

4.2 R&W 完全共有 vs R/W 完全共有 vs 部分共有 (分割)

VRの共有形態に関して，以下の選択肢がある．

- **R&W 完全共有** (図 6(a)) : すべての FU 間で VR 全体を共有する．各 FU から全 VR に対して直接，読出しおよび書込みの両方 (**R&W**) を行うことが可能である．ハードウェア・コストは 3 つの選択肢の中で最も高い．

- **R/W 完全共有** (図 7(a),(b)) : すべての FU 間で VR 全体を共有する．ただし，各 FU からの読出し/書込み (**R/W**) は，いずれか一方が全 VR に対して可能だが，他方は一部の VR に対してのみ可能である．

- **部分共有 (分割)**(図 7(c)) : 各 FU から直接，読出しおよび書込み可能な VR が全部でなく一部に限定されている．直接アクセスが不可能な VR 内データに対しては，VR 間データ転送を行う必要がある．このとき，転送用 FU が性能上のボトルネックにならないような FU のグループ化を検討する必要がある．VR 間転送方式としては，ハードウェアの責任で暗黙的に転送を行う方式と，コンパイラの責任で明示的に転送を行う方式がある．部分共有方式のハードウェア・コストは 3 つの選択肢の中で最も低い．また，分割数が多くなればなるほどハードウェア・コストはさらに下がる．SR の分割については文献 [5] で，VR の分割は文献 [3] で提案されている．

上記 3 つの選択肢がソフトウェア・パイプライニングに与える影響は以下のとおりである．

- R&W 完全共有と R/W 完全共有では，通常は大きな性能差はない．必要な VR 数が多いプログラムをスケジューリングする場合には，R/W 完全共有では MOVE 命令の挿入等により VR 上のデータを RF 間で再配置する必要があり，このため性能が低下する．このため，R/W 完全共有の場合，R&W 完全共有に比べてコンパイラの負担が大きいといえる．図 8 に R&W 完全共有の場合においてスケジューリングした例を示す．

- 部分共有の場合，VR 間データ転送のオーバヘッドのためクリティカル・パスが長くなる．しかしながら，VR 間データ転送能力が十分にあれば，R&W 完全共有の場合と同程度のスループットが得られる．図 9 に転送用ユニットを 2 つ設けた場合におけるスケジューリング例を示す．

4.3 各命令専有 vs 各 VR 専有 vs 全 VR 共有

VR のポインタの所有者に関して，以下の選択肢がある．

- **各命令専有** : 各命令ごとに，ソース・レジスタ (Rs) 用に 2 個，ディスティネーション・レジスタ (Rd) 用に 1 個，計 3 個のポインタを所有する．従来ベクトル・プロセッサにおけるベクトル処理を行う場合，必要なポインタ数は FU 数に比例して決まる．しかしながら，ソフトウェア・パイプライニングを行う場合は，ループ・ボディを構成する命令数に比例したポインタ数が必要となる．

- **各 VR 専有** : 各 VR ごとに，読出し用ポインタ (Rp)，および書込み用ポインタ (Wp) を 1 個ずつ所有する．必要なポインタ数は VR 数に比例して決まる．

- **全 VR 共有** : 全 VR に共通に有限個のポインタを設ける．必要なポインタ数は一定であり，3 つの選択肢の中では必要なポインタ数が最小である．本方式の 1 種であるレジスタ・ウィンドウ方式が文献 [6] で提案されている．

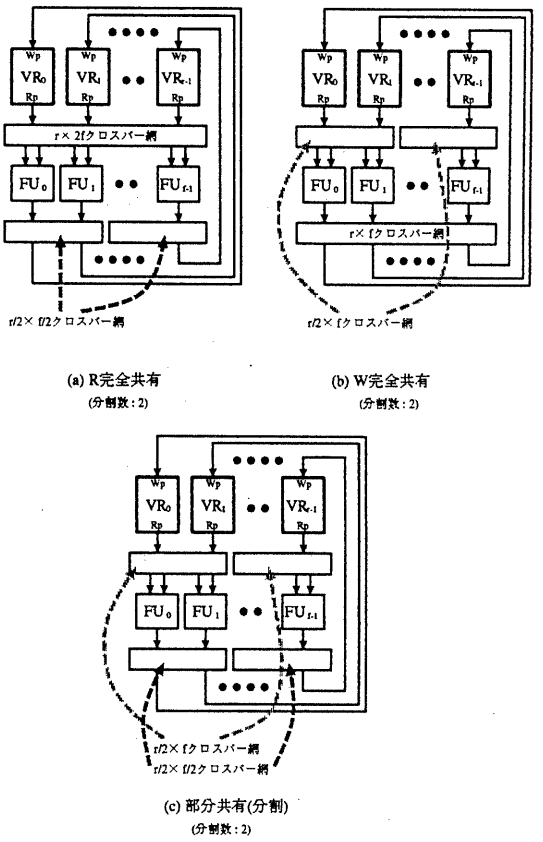


図 7: VR の共有形態

各選択肢が，ソフトウェア・パイプライニングに与える影響は次節で記す．

4.4 暗黙インクリメント vs 明示インクリメント

VR のポインタのインクリメント指定に関して，以下の選択肢がある．

- **暗黙インクリメント** : 対応する命令が実行されたり，あるいは，対応する VR がアクセスされるごとに暗黙的にポインタを自動インクリメントする．

- **明示インクリメント** : 対応する命令が実行されたり，あるいは，対応する VR がアクセスされるごとに，明示的な指定 (インクリメントの可否) に従ってポインタを操作する．VR を SR として使用する場合は，インクリメントしないように指定すればよい [7]．

前節の 3 つの選択肢および上記の 2 つの選択肢が，ソフトウェア・パイプライニングに与える影響は以下のとおりである．

- 各命令専有の場合，各命令が実行されるごとに該当するポインタを暗黙的にインクリメントすればよい．

Turboモード

TIME	Load/Store	Load/Store	FP乗算器	FP加算器	整数系
1	LOAD V2, Y(K)	LOAD V1, Z(K+11)	MUL V6,V2,V5	NOP	MOVE V0, V1
2	STORE X(K) , V7	NOP	MUL V3, S1,V0	ADD V5, V3,V4	NOP
3	NOP	NOP	MUL V4, S2,V1	ADD V7,S3,V6	NOP

定常状態

図 8: ソフトウェア・パイプライニング時の命令コード (R&W 完全共有-各命令専有-暗黙インクリメント)

Turboモード

TIME	Load/Store_0	FP乗算器	転送用FU	転送用FU	Load/Store_1	FP加算器
1	LD V0,Z(K+10)	MUL V4,S2,V1	TRNS01 V5	TRNS10 V6	LD V1,Z(K+11)	ADD V5,V3,V4
2	LD V2,Y(K)	MUL V6,V2,V5	NOP	TRNS10 V3	ST X(K) ,V7	ADD V7,S3,V6
3	NOP	MUL V3,S1,V0	TRNS01 V1	TRNS10 V4	NOP	NOP

定常状態

* (FP乗算器と Load/Store_0) , (FP加算器とLoad/Store_1)はそれぞれ別々のVR群を共有する

図 9: ソフトウェア・パイプライニング時の命令コード (部分共有 (分割)- 各命令専有-暗黙インクリメント)

明示インクリメントはVRをSRとして使用する場合に効果的である．ただし，命令所有-明示インクリメントの組合せはスケジューリング上の利点は特になく，命令所有-暗黙インクリメントと等価である．いずれにせよ，他の二つの選択肢に比べ，スケジューリングは容易である．

- 各VR専有の場合，同一VRの内容を複数回参照する時に問題が生じる．暗黙インクリメントの場合は，同一レジスタ内容を複数回参照することができない．明示インクリメントの場合は，同一番号のレジスタの異なるベクトル要素に対して，1, 1, 2, 2, 3, 3, のように昇順にアクセスする場合には対処できる．しかし，2, 1, 3, 2, 4, 3, というような順の参照に対しては対処できない．したがって，ソフトウェア・パイプライニングに対する制約が大きいといえる．

- 全VR共有の場合，例えば先行するイタレーションからの値の読出し用，実行中のイタレーションでの中間計算値の一時保存用，および後続イタレーションへの値の引き渡し用のポインタを区別して設ける．全VR共有-暗黙インクリメントでは，ポインタ数が少ない場合にはコード・スケジューリングに大きな制約となる．そのため，自由なコード・スケジューリングには各命令専有-暗黙インクリメントと同程度のポインタ数が必要となる．よって，ハードウェア・コストが小さく，コード・スケジューリングが容易な全VR共有-明示インクリメントの組合せが望ましい．

4.5 ラップアラウンド不可 vs 可

VRのポインタのラップアラウンドに関して，以下の選択肢がある．

- ラップアラウンド不可：VRのポインタのラップアラウンドを許さない．常に$Wp \geq Rp$の関係が成り立つ．

- ラップアラウンド可：VRのポインタのラップアラウンドを許し，VRをリングバッファ$FIFO$として使用する．初期状態では$Wp \geq Rp$の関係にある．次に，Wpがラップアラウンドすると$Wp < Rp$の関係になる．続いてRpがラップアラウンドすると，$Wp \geq Rp$の関係に戻る．

上記2つの選択肢がソフトウェア・パイプライニングに与える影響は，以下のとおりである．

- ラップアラウンド不可の場合，ストリップ・マイニングが必要である．ソフトウェア・パイプライニングの場合，定常状態でVRのレジスタ長に等しい回数だけループを実行するごとにストリップ・マイニング操作が必要となる．よって，VRのレジスタ長が小さい場合は，ストリップ・マイニング操作の回数が多くなり，総実行時間中に占めるストリップ・マイニングによるオーバヘッドの割合が相対的に大きくなる．

- ラップアラウンド可の場合，ストリップ・マイニングが不要である．

ハイパースカラ・プロセッサの場合，ベクトル・プロセッサに比べて，VRのレジスタ長はそれほど大きい必要はなく，VR数が多い方が適している．したがって，ラップアラウンド可の方が望ましい．

5 おわりに

本稿では，ハイパースカラ・プロセッサを構成する際に重要となるベクトル・レジスタ (VR) の構成法について検討した．さらに，各構成法がソフトウェア・パイプライニングに与える影響について検討した．

VRの構成においては，メモリ構成とポインタ構成について検討する必要がある．選択肢として次のものを検討した．

- メモリ構成
  - –論理-物理一致型 vs 不一致型
  - –R & W 完全共有 vs R/W 完全共有 vs 部分共有 (分割)
- ポインタ構成
  - –各命令専有 vs 各VR専有 vs 全VR共有
  - –暗黙インクリメント vs 明示インクリメント
  - –ラップアラウンド不可 vs 可

以上の選択肢について検討した結果，ソフトウェア・パイプライニングを効果的に行うことが可能で，かつ，ハードウェア・コストが小さい選択肢は，以下のものであることがわかった．

- 論理-物理一致型であることが望ましい．
- R&W 完全共有，R/W 完全共有，部分共有 (分割) のいずれでもよい．
- ラップアラウンド可であることが望ましい．

ポインタの所有者，インクリメント法は各選択肢とも一長一短がある．現在認識している点は，以下のとおりである．

- VRとは別にスカラ・レジスタ (SR) が存在する場合，各命令専有-明示インクリメントの組合せは無意味である．
- 各VR専有の場合，同一VR内容を複数回参照する時にうまく処理できない場合が多い．
- 全VR共有-暗黙インクリメントでポインタ数が少ない場合，コード・スケジューリングがかなり不自由である．少ないポインタ数でコード・スケジューリングする場合，全VR共有-明示インクリメントの方がよい．

今後，さらに検討が必要である．また，各選択肢とハードウェア・コストとの関係を定量的に評価して，ハイパースカラ・プロセッサで採用するレジスタ・ファイル構成を決定する必要がある．

謝辞

日頃ご討論頂く九州大学 大学院総合理工学研究科 安浦寛人教授，ならびに，宮嶋浩志氏をはじめとする安浦研究室の諸氏に感謝致します．

本稿の執筆にあたり御助力頂いた九州大学 工学部の弘中哲夫助手に感謝致します．

参考文献

[1]村上和彰，“ハイパースカラ・プロセッサ・アーキテクチャ — 命令レベル並列処理への第 5 のアプローチ —，” 並列処理シンポジウム JSPP '91 論文集，pp.133–140，1991 年 5 月．

[2]村上和彰，橋本隆，弘中哲夫，安浦寛人，“マイクロベクトルプロセッサ·アーキテクチャの検討”，情報処理学会研究報告，ARC-94-3，pp.17–24，1992 年 6 月．

[3]斎藤靖彦，ハイパースカラ・プロセッサ・プロトタイプの設計，九州大学卒業論文，1993 年 2 月．

[4]村上和彰，“スーパコンピュータの記憶システム”，情報処理，vol.34,no.3，pp361-371，1993 年 3 月．

[5]丸島敏一，西直樹，“スーパスカラプロセッサにおけるレジスタ分割方式”，情処 39 全大論文集，7X-1，1989 年 10 月．

[6]中村宏，位守弘充，伊藤元久，中澤喜三郎，“レジスタウィンドウとスーパースカラ方式による擬似ベクトルプロセッサの提案，” 並列処理シンポジウム JSPP'92 論文集，pp.367-374，1992 年 6 月．

[7]藤井啓明，稲上泰弘，“命令並列処理機構を意識したスケジューリングを支援するレジスタ構成とその評価，” 並列処理シンポジウム JSPP'93 論文集，pp.307-314，1993 年 5 月．