MN-Core™ 2 ソフトウェア開発者マニュアル

株式会社 Preferred Networks

2024 年 2 月 21 日

概要

この文章は、MN-Core 2 で動作するアプリケーションを開発するために必要な、ハードウェアとアセンブリ命令の仕様を説明する。

目次

第 1 章

MN-Core 2 アーキテクチャ概観

1.1 構成

MN-Core 2 は、ツリー状に階層化されたメモリ間での集団通信と、そのツリーの葉にあたる多数の行列ベクトル積専用回路付き演算ユニットでの浮動小数点数演算を、VLIW 形式の命令により並列動作させることで、高い実効性能・電力性能を実現する SIMD 並列方式のアクセラレータボードである。

キャッシュは存在せず、すべてのボード内データ転送は機械語命令で明示的に指定される。機械語命令は制御構造の存在しない、1 ボードに対して単一のストリームである。

キャッシュの代わりに、ツリーの葉には演算ユニットに加えて大容量のローカルメモリ (SRAM) が存在する。データの移動をできるだけツリーの葉側に留めるように並列演算を配置することで、高帯域なデータ移動を低コストに実現し、演算効率を高められる。

1 ボードは 1 チップと周辺回路からなる。

1 チップはツリーの根にあたる**トップレベル**と、その子である 8 つの L2B (Level 2 Block) からなる。

L2B 以下は次のようなツリーになっている。

- 1 つの L2B は 8 個の L1B (Level 1 Block) を子として持つ
- 1 つの L1B は 16 個の MAB (Matrix Arithmetic Block、行列演算ブロック) を子として持つ
- 1 つの MAB は 4 個の PE を子として持ち、また 1 つの MAU (Matrix Arithmetic Unit、行列演算ユニット) を持つ

よって例えば PE はボードあたり 4096 個あることになる。

L2B と L1B はそれぞれローカルに SRAM を持ち、L2BM および L1BM と呼ばれる。PE はいくつかの種類のローカルメモリと ALU (Arithmetic Logic Unit、整数演算ユニット) からなる。

L2B は 2 つごと、計 4 つの**グループ**に分かれており、グループごとに 1 つの PDM (PIU Data Memory、PIU は PCIe Interface Unit) という SRAM と、DRAM が付属する。トップレベルは自グループおよび他グループの間で、PDM、DRAM、L2BM の 3 種のメモリ (**上位記憶**) 間のデータ転送を行える。第 0 番グループの PDM はホストと PCIe インターフェースで接続され、ホストとの入出力データ通信はすべて PDM を経由する。DRAM はメインメモリの役割を果たす。

上位記憶と L1BM および PE 内ローカルメモリが冒頭で述べた『ツリー状に階層化されたメモリ』、MAU が『ツリーの葉にあたる多数の行列ベクトル積専用回路付き演算ユニット』にあたる。

1.2 機械語命令概観

機械語命令は上位記憶間データ転送を制御するデータ移動命令 (以下 MV 命令) と、L2B 以下を制御する PE 命令からなる[*1]。

PE 命令は 1 命令で L2B 以下全体を 4 サイクル制御する[*2]。この 4 サイクルの単位を**ステップ**と呼ぶ。

PE 命令は VLIW 形式であり、1 命令で複数のメモリユニット・演算ユニットを同時に制御できる。ソフトウェアパイプライニングにより 1 命令になるべく多くの動作を詰め込み、ユニットが停止している時間を最小化することが性能向上の鍵となる。

命令実行はパイプライン化されており、命令発行直後に結果を利用することはできない。命令発行後、いずれかのメモリに書き込んだ値を読み出せるようになるまでのサイクルを空けるのはプログラマの責任である。

PE 内には GRF(General Register File)0、GRF1、T レジスタ (Temporary Register)、LM(Local Memory)0、LM1 の 5 つのメモリ要素が存在する。これらをまとめて PE **メモリ**と呼ぶことにする。

PE メモリには ALU、MAU、L1BM の 3 つの**演算器**が接続されている[*3]。

図 1.1 に PE 命令の構造を示す。

演算器と PE メモリの間のデータパスは常に 2 長語/サイクルである。2 長語より短い結果については常に 2 長語の MSB 側に置くようになっている。ここで MSB とは最上位ビット (Most Significant Bit) を指す。LSB=最下位ビット (Least Significant Bit) と合わせて以降説明無しで用いる。格納形式はビッグエンディアンである。すなわちアドレッシング可能なら MSB 側が小さいアドレスに対応する。

命令ストリームは Auto stride **モード**と Flat **モード**の 2 つのモードを持つ。

Auto stride モードでは、命令ストリームの 1 単位は 2 つの PE 命令と 1 つの MV 命令から構成される。Flat モードでは、命令ストリームの 1 単位は 3 つの PE 命令と 1 つの MV 命令から構成される。Auto stride モードか Flat モードのいずれかに統一された命令列を**パックされている** (packed) と呼ぶ。実機に流す命令はパックされていなければならない。エミュレータはパックされていない、つまり PE 命令と MV 命令が任意の順序で現れる命令列も実行できる。詳細は第 2.3 節で述べる。

Auto stride モードで表現できる命令列は、Flat モードで表現できる命令列のサブセットになっている。

Auto stride モードでは、ステップ内では毎サイクル、PE 命令のアドレス値に、ある増分値が加算される。これにより 1 命令で複数サイクルの異なる内容の演算を実現する。さらに PE 命令 3 つと MV 命令 1 つを合わせたものが命令ストリームの 1 単位となり、これが繰り返される。

Flat モードはステップあたりの命令帯域が大きくなる代わりにアドレス指定を柔軟にしたものである。具体的には、ステップ内の各サイクルで、GRF0、GRF1、LM0、LM1 のアドレス値をすべて指定する。それ以外のメモリのアドレスは Auto stride モード同様、自動的に決まる増分値を第 0 サイクルのアドレスに加えることで決定される。さらに PE 命令 2 つと MV 命令 1 つを合わせたものが命令ストリームの 1 単位となり、これが繰り返される。

アセンブル時には Auto stride モードと Flat モードどちらを用いるかを指定しなければならない。Auto stride モードで表現可能な PE 命令は Flat モードでも表現可能なので、Flat モードであれば両モードの PE

[*1] PE 命令はハードウェア要素としての「PE」より上位の要素の制御も行うが、慣例的にこう呼ばれる。

[*2] これは 1 命令 1 サイクルだとホスト-ボード間の PCIe 帯域が不足するためである。

[*3] L1BM は L1B に備わったメモリの名前でもあるが、PE とデータをやり取りするユニットという意味でここでは演算器としてもこう呼ぶこととする。実際に縮約演算を行う回路も備わっている。

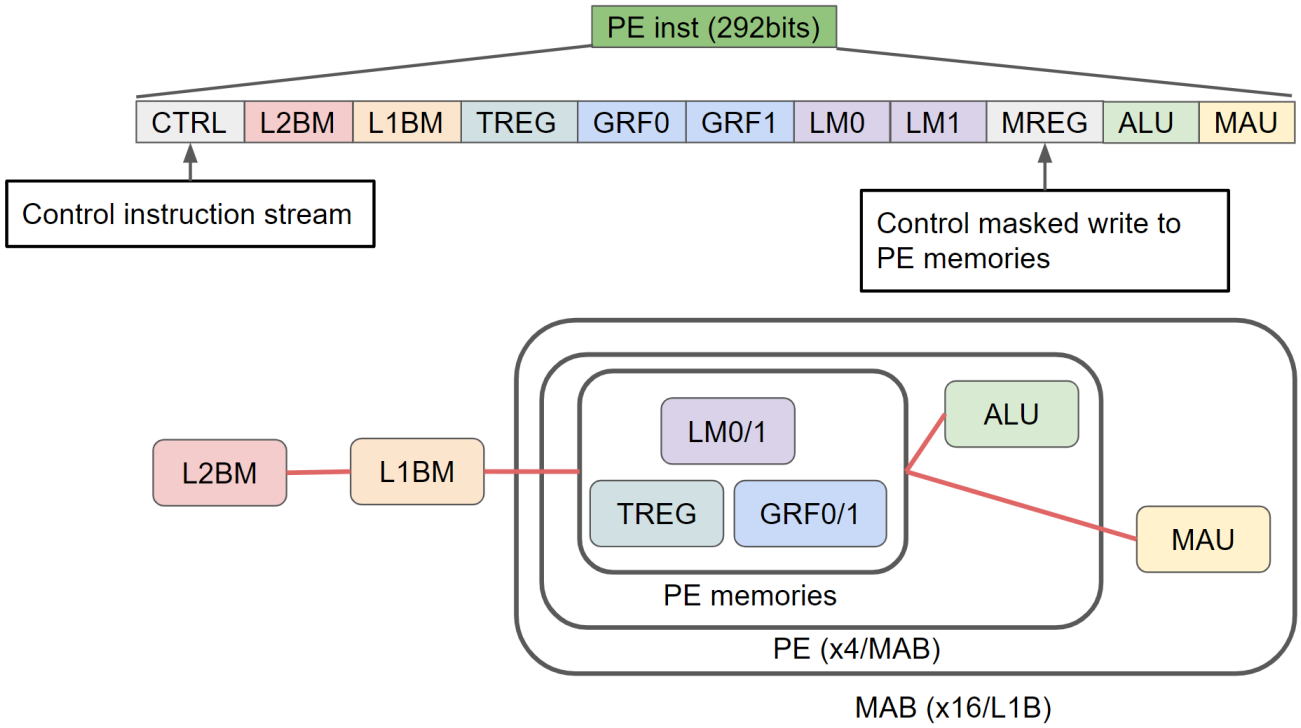

図 1.1　PE 命令の構造

命令式が混在したアセンブリを処理できる。逆に、Flat モードの PE 命令式が含まれるアセンブリを Auto stride モードでアセンブルしようとすると、たとえ命令式を Auto stride モードに等価に書き換えることが可能だったとしてもエラーになる。

MV 命令は発行後、後続の命令とは非同期に実行される。ただし、PE 命令は wait 命令を含むことができ[*4]、その場合指定した MV 命令の完了まで、その PE 命令の発行直前で命令ストリームを待機させることができる。

つまりある PE 命令の発行後にある MV 命令を発行したい場合はその順番に命令ストリームを記述すればよく、逆に MV 命令の完了後に PE 命令を発行したい場合は wait 命令を用いればよい。

PE 命令と MV 命令いずれでも、データ転送には基本的に、それと対称な逆方向の転送が可能である。例えば上位階層から下位階層への分配 (データを等分割して送信) のモードがある場合、基本的には下位から上位への結合 (データを等サイズで読み出して送信し、繋げて書き込み) のモードが存在し、分配したデータをそのまま結合するとレイアウトを含めて同一のデータが上位階層に完成するようになっている。これによりデータレイアウトの一貫性を保つことができる。

図 1.2 に命令ストリームの構造と、PE 命令および MV 命令の制御範囲を示す。

[*4] 命令は固定ビット幅なので、正確には wait 命令は常に PE 命令に含まれており、オプションでどの MV 命令に対しても待機しないことを指定できる。

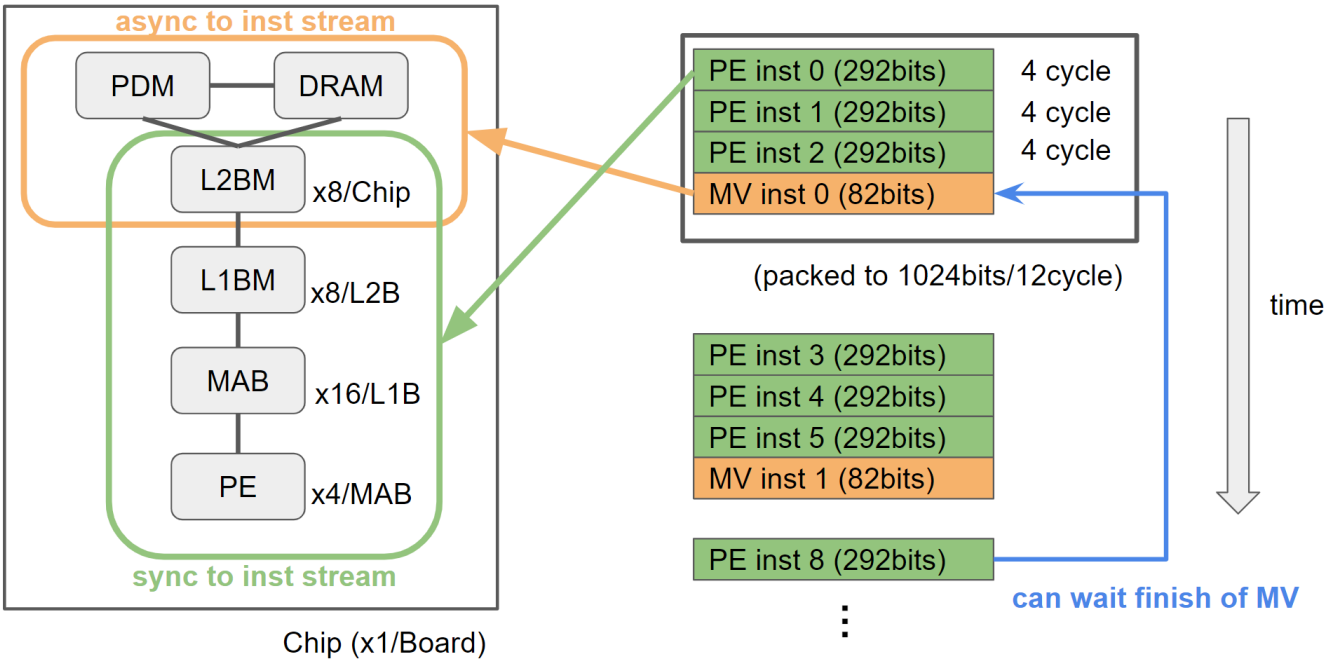

図 1.2　命令ストリームの構造と制御範囲。右側は Auto Stride モードの場合の命令ストリームを示している。

1.3　語長と演算精度

PDM と DRAM では最小のアクセス単位は 1 語=64bit である。命令種別ごとにアラインメント制約があるため常にアドレスをこの単位で指定できるわけではない。

L2B 以下は 1 長語=64bit のアクセス単位を基本とする。PE では一部 1 短語=32bit 単位および 4 短語=2 長語=128bit でのアクセスが可能である。

PE 命令の演算においては整数・浮動小数点数ともに以下の 3 種類がサポートされる。

- 半語 (半精度): 16bit
- 短語 (単精度): 32bit
- 長語 (倍精度): 64bit

例えば PE で長語アクセスした値に対して ALU で半精度演算を行った場合、1 長語に含まれる 4 半語に対して SIMD 演算が実行される。

整数は符号なしと符号ありがある。

浮動小数点数フォーマットの各部ビット数はそれぞれ以下である。

- 半精度: 符号 1bit、指数部 6bit、仮数部 9bit
- 単精度: 符号 1bit、指数部 8bit、仮数部 23bit
- 倍精度: 符号 1bit、指数部 11bit、仮数部 52bit

浮動小数点数は正規化数、正負のゼロ、正負の無限大のみからなり、非正規化数と NaN は存在しない。

整数には -0 はないが浮動小数点数にはあることに注意する。

仮数部にはけち表現を用いる。例えば、半精度の `1.0`はビット列としては `0x3e00`である。

倍精度値 1 語は、長語 64 ビットの MSB から LSB に向かって、符号ビット、指数部の上位ビットから下位ビット、仮数部の上位ビットから下位ビットの順に格納される。単精度値 2 語は、長語 64 ビットの MSB 側と LSB 側それぞれ 32 ビットについて、同様の順に格納される。半精度値 4 語は、長語 64 ビットの MSB 側から 16 ビットずつ、同様の順に格納される。これらは 2 長語 128 ビット内の単精度値 4 語や整数値についても同様である。

PE で短語アクセスした際は、短語アドレスの下側 (偶数側) が長語アクセスの MSB 側、上側 (奇数側) が長語アクセスの LSB 側となる。

行列ベクトル積演算を行う場合には事前にブロックフロート化という変換を行っておく必要があり、ブロックフロート化浮動小数点数は上で述べたものとは異なるフォーマットになる[*5]。具体的には、各部ビット数は通常の浮動小数点数と同じであるが、仮数部にはけち表現を用いない。

1.4 浮動小数点数演算性能

MAU (行列演算ユニット) による演算性能について述べる。MAU は行列ベクトル積和演算モードとベクトル積和演算モードのふたつのモードを持つ。

行列ベクトル積和演算モードでは、ひとつの MAU は 1 サイクルで、$m \times n$ 行列 A と n 次元ベクトル b, c に対し $y = A \times b + c$ の演算を行える。ここで m, n と演算精度は次のいずれかである。

- 倍精度: $m = 2, n = 4$、A, b はブロックフロート倍精度、c, y は通常の倍精度
- 単精度: $m = 8, n = 4$、A, b はブロックフロート単精度、c, y は通常の単精度
- 疑似単精度: $m = 8, n = 8$、A, b はブロックフロート疑似単精度、c, y は通常の単精度
  - ここでブロックフロート疑似単精度とは単精度よりも有効な仮数部長が短いブロックフロート形式である。疑似単精度はブロックフロート形式でしか現れない
- 半精度: $m = 16, n = 16$、A, b はブロックフロート半精度、c, y は通常の単精度

よって行列ベクトル積和演算モードでの倍精度のボードあたりピーク FLOPS 値としては $2 \times 4 \times 2 \times 1024$ に周波数をかけたものとなる。ここで後ろの 2 は積和演算 1 回を 2FLOP と数えている。

PE から L1BM など上位階層への転送において、浮動小数点数加算による縮約が可能であるが、演算器としての性質が MAU と大きく異なるため、公式のピーク性能値においてはカウントしていない。

ベクトル積和モードでは、ひとつの MAU は 1 サイクルで、m 次元ベクトル a, b, c に対し $y = a \times b + c$ の演算を行える。ここで $\times, +$ は要素ごとの独立な演算であって、m と演算精度は次のいずれかである。

- 倍精度: $m = 4$、a, b, c, y はすべて通常の倍精度
  - ただし積は 0,1 番目の要素か、2,3 番目の要素のどちらかしか有効にできない。有効にしなかった側は積の項は 0 になる
  - 4 要素の FMA 全体を実行するには、この演算を 2 回実行すればよい。その際、2 回目の c には 1 回目の結果を渡す

[*5] MN-Core ではブロックフロートを用いるのは半精度のみだったが、MN-Core 2 では全精度で用いる

- 単精度: $m = 8$、a, b, c, y はすべて通常の単精度
- 半精度: $m = 16$、a, b は通常の半精度、c, y は通常の単精度

よってベクトル積和演算モードでの倍精度のボードあたりピーク FLOPS 値としては $2 \times 2 \times 1024$ に周波数をかけたものとなる。ここでも 2 は積和演算 1 回を 2FLOP と数えている。

$m = 8$ の単精度 FMA が可能なので、ベクトル積和モードには疑似単精度に相当するものは存在しないことに注意する。

1.5 メモリ性能

各メモリ種別のサイズとスループットを述べる。スループットについては実際は、使用可能な転送種別や読み書き切り替えオーバーヘッドの存在など様々な制約があり、順次解説する。

- PDM: グループあたり容量 4MiB。
- DRAM: グループあたり容量 4GiB、帯域約 128GB/s。
- L2BM: 1L2BM あたり容量 32Ki 長語。
- L1BM: 1L1BM あたり容量 8Ki 長語。
- LM0: 1PE あたり容量 2Ki 長語。2 長語/サイクルで 1RW。
- LM1: LM0 と同じ。
- GRF0: 1PE あたり容量 256 長語。2 長語/サイクルで 1R1W。
- GRF1: GRF0 と同じ。
- T レジスタ: 1PE あたり容量 8 長語。2 長語/サイクルで 1R1W。

第 2 章

ツールチェーン

2.1 実機

DRAM を起点としたアセンブリ実装に対し、ホスト-DRAM 間のデータ入出力と命令実行を行うための API が存在する。この API の説明は別文書に譲る。

2.2 アセンブラ

MN-Core 2 アセンブラは、第 3 章で述べるアセンブリ言語で実装されたプログラムを機械語に変換する。標準のバイナリ名は `assemble3`である。

`assemble3`に PATH が通っている状況で以下を実行すると pass.asm にアセンブル結果が出力される。

```
echo 'lpassa $lm0v $ln0v' > pass.vsm
assemble3 pass.vsm > pass.asm
```

2.3 エミュレータ

MN-Core 2 ボードのエミュレータが存在する。命令ストリームがパック形式 (第 1.2 節) になっている必要がなく、またアセンブリ言語内で主要なメモリ要素の内容を出力する制御文 (Debug get 文、第 3.4.3 節) が記述できるため挙動の確認に向く。

標準のバイナリ名は `gpfn3_package_main`である。

以下はファイル `sample.vsm`に手書きした命令列をアセンブルし、エミュレータで実行する例である。`assemble3`および `gpfn3_package_main`に PATH が通っている前提とする。

1 行目の `lpassa`命令は LM0 にそれが属する PE の番号 (0 から 3) を書き込み、2 行目の `d get`命令はあるひとつの MAB について、書き込んだ場所の内容を読み出している。`d get`命令で読み出した値は`-d`オプションで指定したファイルに出力される。よってファイル `sample.dmp`には、PE 番号に対応して 0 から 3 の値が書き込まれることとなる。

1 行目の `lpassa`命令についての詳細は第 3.6.12.4 節、第 3.6.1.20 節、第 3.6.1.6 節を、2 行目の `d get`命令についての詳細は第 3.4.3 節を参照のこと。

```
$ cat sample.vsm
lpassa $subpeid $lm0
```

```
d get $lm0n0c0b0m0 1
$ assemble3 sample.vsm > sample.asm
$ gpfn3_package_main -i sample.asm -d sample.dmp
$ cat sample.dmp
DEBUG-LM0(n0c0b0m0p0,0):(f:0, i:{{0x0,0x0},{0x0,0x0}}, v:0x0) #d get $lm0n0c0b0m0 1
DEBUG-LM0(n0c0b0m0p1,0):(f:0, i:{{0x0,0x0},{0x0,0x1}}, v:0x1) #d get $lm0n0c0b0m0 1
DEBUG-LM0(n0c0b0m0p2,0):(f:0, i:{{0x0,0x0},{0x0,0x2}}, v:0x2) #d get $lm0n0c0b0m0 1
DEBUG-LM0(n0c0b0m0p3,0):(f:0, i:{{0x0,0x0},{0x0,0x3}}, v:0x3) #d get $lm0n0c0b0m0 1
```

第 3 章

MN-Core 2 アセンブリ言語による開発

本章ではアセンブリ命令の文法と効果を示す。
以降では次の凡例に従って文法を示す。

- <>で囲んだ部分は実際の命令文で適宜置き換えられるべきである
- (A|B)は Aまたは Bを選択する
- []で囲んだ部分はオプショナルである

3.1 文

アセンブリ言語では 1 行に 1 文を記述する。
文には以下の種類がある。

- 制御文 (第 3.4 節)
- 機械語命令文
  - MV 命令文 (第 3.5 節)
  - PE 命令文 (第 3.6 節)

3.2 数値

機械語命令文内に直接書かれる数値には以下がある。

- 自然数 (第 3.2.1 節)
- 即値 (第 3.2.2 節)
- タグ (第 3.2.3 節)

3.2.1 自然数

自然数はアドレス等の記述に用いる。10 進整数をデフォルトとし、0b，0o，0xのプレフィックスによりそれぞれ 2 進、8 進、16 進での表記が可能である。負数の指定はできない。

アドレスの単位は命令によって異なる。アドレス可能な PE メモリ (GRF、LM0、LM1) では短語、L1BM

と L2BM では長語、PDM と DRAM では 64 ビットワード単位である。これらはアクセス語長指定に依らない。

3.2.2 即値

即値は ALU が imm命令により出力する値である。即値のフォーマットは以下である。

```
  (f|h)"<floating-point-number-literal>"
| (i|s|ui|us)"<integer-number-literal>"
```

ダブルクオートが必須なので注意する。

先頭は数値型を指定する。(f|h)は単精度および半精度浮動小数点数、(i|s)は符号ありの単精度および半精度整数、(ui|us)は符号なしの単精度および半精度整数である。長語の数値型 d, l, ulは即値指定ビット数が不足しているため指定できない。

即値命令が実際に出力する値は、ペイロードリテラルが指定する短語の値を、第 3.6.12.2 節で述べる方法で並べたものである。上記の数値型指定に関わらず、ペイロードリテラルは短語の値を定める。数値型が半精度ならば、同じ値を 2 つ並べて短語の値とする。

ペイロードとなるリテラルの解釈は以下の通りである。

- 浮動小数点数 (<floating-point-number-literal>) の場合
  - まず C 言語の strtofで float の値に変換される。
  - その後単精度指定であればそのまま扱われ、半精度指定であれば丸めが行われる。
- 整数 (<integer-number-literal>) の場合
  - 符号あり整数 i, sであれば、先頭に符号 +/-をつけることを許す。
  - 符号以降は自然数として扱う。
  - 値が数値型の範囲外であればエラーになる。

以下に例を示す。ALU が実際に出力する結果の詳細な定義は即値命令式の節 (第 3.6.12.2 節) に譲る。

例 1

単精度浮動小数点数の-1.0 を LM1 に出力する。

```
imm f"-1.0" $ln0
```

例 2

半精度符号なし整数の 0x8000 を T レジスタに出力する。

```
imm us"0x8000" $t
```

なおこの例で imm s"0x8000" $tとすると、半精度符号あり整数の範囲外なのでエラーとなる。

3.2.3 タグ

タグは MV 命令と PE 命令の間の待ち合わせのための値である。タグは識別のため必ず先頭に iを付け、その後 0 埋めありで 2 桁固定の 16 進数でタグ値を記述する。16 進数であるがプレフィックス 0xは付かない。

実例は wait 命令式の節 (第 3.6.13 節) に譲る。

3.3 疑似コードによる命令動作の記述

本文書では一部の命令の動作を疑似コードで記述してある。疑似コードの文法を厳密には定義しないが以下のルールに従って記述する。

- `MEM`は MN-Core 2 ボードの各階層のメモリ要素をまとめた構造体である
- `LongWord`は長語 (64 ビットワード)、`ShortWord`は短語 (32 ビットワード)、`HalfWrod`は半語 (16 ビットワード) の型を表す
- `for`は通常の for 文だが、次のルールに従う
  - `:`によるインデックス表記は始端がインクルーシブ、終端がエクスクルーシブである。`forcycle=0:4`と書いたら `cycle`は `0,1,2,3`を取る
  - `for cycle`とあるとき `cycle`の値それぞれで PE 命令の各サイクルで起きることを示す
  - `forall group`, `forall l2b`, `forall l1b`, `forall mab`, `forall pe`はそれぞれ `for group = 0:4`, `for l2b = 0:2`, ... などの略記である
  - `forall`の直接のネストは `forall chip,l2b,l1b`などと略記する
- 次の関数を用いる
  - `refer_pemem(mem, cycle)` PE メモリのオペランド表記とサイクル番号から PE メモリへの参照を返す
  - `refer_matreg(side, wl)` 行列レジスタの面と要素の語長から行列レジスタへの参照を返す
    - 行列レジスタの面は `x`か `y`のいずれか
    - 要素の語長は `LongWord`、`ShortWord`、`HalfWord`のいずれか[*1]
    - 行列レジスタの詳細は 3.6.1.14 節で述べる
  - `get_unit_value(rrn_opcode)` 縮約演算指定 `rrn_opcode`の単位元を返す。具体的には演算が fadd, iadd, bor, lor なら all 0、演算が max, band, land なら all 1、演算が min ならその演算精度における符号あり整数の最大値である

3.4 制御文

制御文は機械語命令実行以外の制御を行う。具体的には、ボードと外部とのデータ通信や、エミュレーションの終了などを司令する。

一部の制御文はエミュレータでのみ有効である。

3.4.1 コメント文

`#` で始まる文はコメント文となり、翻訳時に無視される。

[*1] 行列レジスタのひとつの面について、書き込みと読み出しで語長指定が一致していない場合は未定義動作である

3.4.2 Quit 文

これ以降の命令を無視し、自身を含めて対応する機械語を出力しない。

アセンブリのテキストに対して小さな編集で以降を実質的にコメントアウトできるのでデバッグ時に有用である。

文法

```
quit
```

効果

これ以降の命令を無視し、自身を含めて対応する機械語を出力しない。

例

```
lpassa $lm0v $lr0v
quit
lpassa $ln0v $lr8v
```

最初の命令文のみが翻訳される。

3.4.3 Debug get 文

Debug get 文は PDM、DRAM、L2BM、L1BM、GRF0/1、LM0/1、T レジスタの内容を出力するエミュレータ実行時専用の制御文である。実機で実行することはできない。

エミュレータはサイクルアキュレートではなく、任意の命令は発行後即時完了することに注意する。言い換えるなら Debug get 文は直前で十分なサイクル数を待ってからメモリを読んだのと同等の結果になる。

文法

```
d get[<dtype>] <memory>[n<group_id>][c<l2b_id>][b<l1b_id>][m<mab_id>][p<pe_id>] <
    num_of_words>
```

ここで<dtype>と<memory>は以下であり、それ以外は自然数である。

```
<dtype> ::= d|f|h
<memory> ::= $p<addr>
           | $d<addr>
           | $lc<addr>
           | $(l|ll)b<addr>
           | $[(l|ll)](r|s|m|n)<addr>
           | $[(l|ll)]t
```

効果

指定したメモリ要素・アドレスのデータをエミュレータ実行時に指定した出力ファイルにダンプする。その際浮動小数点数としての解釈が付加される。

以下、構文の各要素の効果を順に解説する。

dtypeはデータをどの数値フォーマットで解釈するかを指定する。どの指定でも浮動小数点数と整数の両方で解釈した結果が出力される。各指定で選択される数値フォーマットを表 3.1 に示す。

表 3.1　Debug get 文のデータフォーマット解釈指定

<dtype>	浮動小数点数	整数
無指定	倍精度	半語および長語
d	倍精度	長語
f	単精度	短語
h	半精度	半語

<memory>の<addr>以前の部分はメモリ要素の種類と語長の指定である。可能な組み合わせを表 3.2 に列挙する。これは実際の MV 命令文や PE 命令文のオペランドで指定可能な組み合わせと同じである。T レジスタについて、実際の命令では常に 2 長語アクセスだが、Debug get 文では$tまたは$ltの記述により長語アクセスが可能である。$tと書いても短語アクセスにはならないことに注意する。

表 3.2　Debug get 文で読み出し可能なメモリ要素と語長の組み合わせ

<memory>	メモリ要素	語長
$p	PDM	長語
$d	DRAM	長語
$lc	L2BM	長語
$lb	L1BM	長語
$llb	L1BM	2 長語
$r, $s, $m, $n	GRF0/GRF1/LM0/LM1	短語
$lr, $ls, $lm, $ln	GRF0/GRF1/LM0/LM1	長語
$llr, $lls, $llm, $lln	GRF0/GRF1/LM0/LM1	2 長語
$t, $lt	T レジスタ	長語
$llt	T レジスタ	2 長語

<addr>はアドレスを指定する。単位は GRF0、GRF1、LM0、LM1 では短語、PDM、DRAM、L2BM、L1BM では長語である。これも実際の MV 命令文や PE 命令文のオペランドのルールに従っている。T レジスタについてはアドレス指定がなく、常に先頭、すなわち第 1 サイクルでアクセスされるエントリから読み出しが開始される。

[n<group_id>][c<l2b_id>][b<l1b_id>][m<mab_id>][p<pe_id>]は順に、グループ、L2B、L1B、MAB、PE の階層について、出力する対象のメモリ要素番号を限定する。文法の都合上、cや bの指定を行う場合は、nの指定が行われていなければならない。指定しなかった場合、その階層の全ての要素が対象となる。例えば n0b1m2とした場合、0 番グループ、1 番 L1B、2 番 MAB 以外の値は表示されない。対象のメモリより下の階層の指定、例えば PDM 出力時の c<l2b_id>指定などは無視される。

<num_of_words>は<memory>で指定した語長を単位として<addr>から何語を読み出すかを 10 進数で指定する。T レジスタにおいては、これを 1 から 4 の範囲で指定することで第 1 サイクルから始めて複数サイクル

分を読み出せる。<memory>を$lt、<num_of_words>を 2 としたときに 2 語目として読み出されるのは、第 1 サイクル分の LSB 側 1 長語ではなく、第 2 サイクル分の MSB 側 1 長語であることに注意する。

エラー

- <memory>による読み出し語長指定が<dtype>によるデータフォーマット解釈指定より短いと、その精度での解釈ができないのでエラーとなる

例 1

```
imm h"1.5" $ln0
d geth $ln0n0c0b0m0p0 1
```

LM1 に書き込んだ半精度の 1.5が 4 つ並んだ長語を、指定した 1PE について読み出す。出力は以下となる。

```
DEBUG-LM1(n0c0b0m0p0,0):(1.5, 1.5, 1.5, 1.5) (0x3f00, 0x3f00, 0x3f00, 0x3f00) #d geth
     $ln0n0c0b0m0p0 1
```

例 2

```
lpassa $l1bid $lr0
d get $lr0n0c0m0p0 1
```

GRF0 に書き込んだ長語整数の L1B 番号を、L1B ごとに 1PE、すなわち計 8PE について読み出す。出力は以下となる。<dtype>が無指定なので、f:に倍精度浮動小数点数、i:に半語整数、v:に長語整数としての解釈がそれぞれ表示されている。

```
DEBUG-GREG0(n0c0b0m0p0,0):(f:0, i:{{0x0,0x0},{0x0,0x0}}, v:0x0) #d get $lr0n0c0m0p0 1
DEBUG-GREG0(n0c0b1m0p0,0):(f:0, i:{{0x0,0x0},{0x0,0x1}}, v:0x1) #d get $lr0n0c0m0p0 1
DEBUG-GREG0(n0c0b2m0p0,0):(f:0, i:{{0x0,0x0},{0x0,0x2}}, v:0x2) #d get $lr0n0c0m0p0 1
DEBUG-GREG0(n0c0b3m0p0,0):(f:0, i:{{0x0,0x0},{0x0,0x3}}, v:0x3) #d get $lr0n0c0m0p0 1
DEBUG-GREG0(n0c0b4m0p0,0):(f:0, i:{{0x0,0x0},{0x0,0x4}}, v:0x4) #d get $lr0n0c0m0p0 1
DEBUG-GREG0(n0c0b5m0p0,0):(f:0, i:{{0x0,0x0},{0x0,0x5}}, v:0x5) #d get $lr0n0c0m0p0 1
DEBUG-GREG0(n0c0b6m0p0,0):(f:0, i:{{0x0,0x0},{0x0,0x6}}, v:0x6) #d get $lr0n0c0m0p0 1
DEBUG-GREG0(n0c0b7m0p0,0):(f:0, i:{{0x0,0x0},{0x0,0x7}}, v:0x7) #d get $lr0n0c0m0p0 1
```

3.4.4 Debug set 文

Debug set 文は L2BM、L1BM、GRF0/1、LM0/1、T レジスタに指定した値を書き込むエミュレータ実行時専用の制御文である。実機で実行することはできない。

文法

```
d set <memory>[n<group_id>][c<l2b_id>][b<l1b_id>][m<mab_id>][p<pe_id>] <num_of_words>
    <payload>
```

ここで<memory>から<num_of_words>までの文法は Debug get 文 (第 3.4.3 節) と同様である。ただし、<memory>に PDM や DRAM を指定することはできない。また、Debug get 文と異なり、<dtype>の指定はない。

<payload>は書き込むデータの内容を、16 進 16 桁で表記した 1 長語を単位として必要な長語数分並べて書く。長語データの表記にはいくつかのシンタックスシュガーがある。これら必要な長語数およびシンタックスシュガーについての詳細は後述する。

効果

指定したデータを、指定したメモリ要素・アドレスに書き込む。

<payload>に書くべき長語の数は、<memory>によって決まるペイロード語長に語数 num_of_wordsを掛けたものである。実際に書かれた長語の数がそれと異なっているとエラーになる。可能なメモリ要素とアクセス語長の指定、またそれに対応するペイロード語長を表 3.2 に列挙する。ここに示したとおり、アクセス語長が短語の際にはペイロード語長がアクセス語長より長くなる。このとき、ペイロードは長語ごとに LSB 側の短語は無視され、MSB 側の短語が書き込まれる。

2 長語以上を書き込むときのアドレッシングはビッグエンディアンである。つまり、ペイロードの先頭側が小さいアドレスに入る。

表 3.3　Debug set 文で書き込み可能なメモリ要素と語長の組み合わせ

<memory>	メモリ要素	アクセス語長	ペイロード語長
$lc	L2BM	長語	1
$lb	L1BM	長語	1
$llb	L1BM	2 長語	2
$r, $s, $m, $n	GRF0/GRF1/LM0/LM1	短語	1
$lr, $ls, $lm, $ln	GRF0/GRF1/LM0/LM1	長語	1
$llr, $lls, $llm, $lln	GRF0/GRF1/LM0/LM1	2 長語	2
$t, $lt	T レジスタ	長語	1
$llt	T レジスタ	2 長語	2

<payload>のそれぞれの長語の記述は 16 **進** 16 **桁整数**、16 **進長語整数**、16 **進短語整数**、16 **進半語整数**の 4 つの記法から選ぶことができる。表 3.4 にそれぞれの記法の説明と例を示す。1 行の中で 16 進長語・短語・半語整数は混在できるが、16 進 16 桁整数は他の記法とは混在できない。

<num_of_words>は<memory>で指定したアクセス語長を単位として<addr>から何語を書き込むかを 10 進数で指定する。T レジスタのアクセス範囲に関する注意は Debug get 文と同様である。

エラー

- <payload>を先頭から解釈していき、表 3.4 に示した記法のいずれにも当てはまらないものがあるとエラーとなる
- <memory>と num_of_wordsによって決まるペイロードの長語数と実際の<payload>の長さが異なる場合エラーとなる
- <payload>に現れる数がその記法における固定または最大の桁数を超えている場合エラーとなる

表 3.4　Debug set 文で書き込む長語データの記法。例は全ての記法で同一の値となる。16 進長語・短語・半語整数の例では固定長でないため先頭に 0 は不要となっている。また、いずれも符号あり整数は使用不可である

記法名	記法	例
16 進 16 桁整数	固定長 16 桁の 16 進数	0123456789abcdef
16 進長語整数	lの後に 1 つの最大 16 桁の 16 進数	l123456789abcdef
16 進短語整数	sの後に_区切りの 2 つの最大 8 桁の 16 進数	s1234567_89abcdef
16 進半語整数	hの後に_区切りの 4 つの最大 4 桁の 16 進数	h123_4567_89ab_cdef

例 1

```
d set $lm0n0c0b0m0p0 2 h1_2_3_4h5_6_7_8
d set $lm4n0c0b0m0p0 2 laabblccdd
d set $lm8n0c0b0m0p0 2 l4321hf_e_d_c
d get $lm0n0c0b0m0p0 6
```

16 進長語整数記法と 16 進半語整数記法、およびそれらの混在の例。出力は以下となる。

```
DEBUG-LM0(n0c0b0m0p0,0):(f:0, i:{{0x1,0x2},{0x3,0x4}}, v:0x1000200030004) #d get
    $lm0n0c0b0m0p0 6
DEBUG-LM0(n0c0b0m0p0,2):(f:0, i:{{0x5,0x6},{0x7,0x8}}, v:0x5000600070008) #d get
    $lm0n0c0b0m0p0 6
DEBUG-LM0(n0c0b0m0p0,4):(f:0, i:{{0x0,0x0},{0x0,0xAABB}}, v:0xAABB) #d get
    $lm0n0c0b0m0p0 6
DEBUG-LM0(n0c0b0m0p0,6):(f:0, i:{{0x0,0x0},{0x0,0xCCDD}}, v:0xCCDD) #d get
    $lm0n0c0b0m0p0 6
DEBUG-LM0(n0c0b0m0p0,8):(f:0, i:{{0x0,0x0},{0x0,0x4321}}, v:0x4321) #d get
    $lm0n0c0b0m0p0 6
DEBUG-LM0(n0c0b0m0p0,10):(f:0, i:{{0xF,0xE},{0xD,0xC}}, v:0xF000E000D000C) #d get
    $lm0n0c0b0m0p0 6
```

例 2

```
d set $lr0n0c0b0m0p0 2 s1_2s3_4
d get $lr2n0c0b0m0p0 1
```

16 進短語整数記法の例。ペイロード位置とアドレスの関係の例示にもなっている。出力は以下となる。

```
DEBUG-GREG0(c0b0m0p0,2):(f:0, i:{{0x0,0x3},{0x0,0x4}}, v:0x300000004) #d get
    $lr2n0c0b0m0p0 1
```

命令 1 行目では$lr0から 2 長語分書き込んでおり、最初の 1 長語には 1_2が、次の 1 長語には 3_4が書き込まれる。命令 2 行目では$lr2、すなわち$lr0から 1 長語進んだアドレスから読み出しているので、結果は短語の 3 と 4 が並んだ値となる。

例 3

```
d set $m0n0c0b0m0p0 2 h1_2_3_4h5_6_7_8
d get $lm0n0c0b0m0p0 2
```

ペイロード語長がアクセス語長より長く、各長語の LSB 側が捨てられる例。出力は以下となる。アクセス語長は短語のため、2 短語=1 長語しか書き込みは行われない。よって、2 長語目は 0 となる。

```
DEBUG-LM0(n0c0b0m0p0,0):(f:0, i:{{0x1,0x2},{0x5,0x6}}, v:0x1000200050006) #d get
    $lm0n0c0b0m0p0 2
DEBUG-LM0(n0c0b0m0p0,2):(f:0, i:{{0x0,0x0},{0x0,0x0}}, v:0x0) #d get $lm0n0c0b0m0p0 2
```

例 4

```
d set $tn0c0b0m0p0 1 123456789abcdef0
d get $lltn0c0b0m0p0 4
d set $lltn0c0b0m0p0 2 1111222233334444555566667777888899999
    aaaabbbbccccddddeeeeffff0000
d get $lltn0c0b0m0p0 4
```

16 進 16 桁整数記法による T レジスタ書き込みの例。1 行目では 1 サイクル分の 2 長語エントリの MSB 側 1 長語に、3 行目では 2 サイクル分の 2 長語エントリ全体に、それぞれ書き込んでいる (3 行目は改行して表示されているが実際は<payload>は 16 進 64 桁が 1 行に書かれている)。Debug get 文による読み出しは 2 行目と 4 行目で同一で、2 長語エントリ × 4 サイクル分全体を読み出している。出力は以下となる。

```
DEBUG-TREG(n0c0b0m0p0,0):{(f:5.62635e-221, i:{{0x1234,0x5678},{0x9ABC,0xDEF0}}, v:0
    x123456789ABCDEF0), (f:0, i:{{0x0,0x0},{0x0,0x0}}, v:0x0)} #d get $lltn0c0b0m0p0 4
DEBUG-TREG(n0c0b0m0p0,1):{(f:0, i:{{0x0,0x0},{0x0,0x0}}, v:0x0), (f:0, i:{{0x0,0x0
    },{0x0,0x0}}, v:0x0)} #d get $lltn0c0b0m0p0 4
DEBUG-TREG(n0c0b0m0p0,2):{(f:0, i:{{0x0,0x0},{0x0,0x0}}, v:0x0), (f:0, i:{{0x0,0x0
    },{0x0,0x0}}, v:0x0)} #d get $lltn0c0b0m0p0 4
DEBUG-TREG(n0c0b0m0p0,3):{(f:0, i:{{0x0,0x0},{0x0,0x0}}, v:0x0), (f:0, i:{{0x0,0x0
    },{0x0,0x0}}, v:0x0)} #d get $lltn0c0b0m0p0 4
DEBUG-TREG(n0c0b0m0p0,0):{(f:1.80811e-226, i:{{0x1111,0x2222},{0x3333,0x4444}}, v:0
    x1111222233334444), (f:1.19826E+103, i:{{0x5555,0x6666},{0x7777,0x8888}}, v:0
    x5555666777778888)} #d get $lltn0c0b0m0p0 4
DEBUG-TREG(n0c0b0m0p0,1):{(f:-2.35957e-185, i:{{0x9999,0xAAAA},{0xBBBB,0xCCCC}}, v:0
    x9999AAAABBBBCCCC), (f:-1.46007E+144, i:{{0xDDDD,0xEEEE},{0xFFFF,0x0}}, v:0
    xDDDDEEEEFFFF0000)} #d get $lltn0c0b0m0p0 4
DEBUG-TREG(n0c0b0m0p0,2):{(f:0, i:{{0x0,0x0},{0x0,0x0}}, v:0x0), (f:0, i:{{0x0,0x0
    },{0x0,0x0}}, v:0x0)} #d get $lltn0c0b0m0p0 4
DEBUG-TREG(n0c0b0m0p0,3):{(f:0, i:{{0x0,0x0},{0x0,0x0}}, v:0x0), (f:0, i:{{0x0,0x0
    },{0x0,0x0}}, v:0x0)} #d get $lltn0c0b0m0p0 4
```

3.5 MV 命令文

本節では MV 命令の文法と効果を述べる。

データ移動 (MV) 命令文は 4 ステップに 1 回実行される MV 命令に翻訳される単位である。MV 命令文には PE 命令文と異なり、式をセミコロンで区切る形式はなく、改行で終端する単一の文のみが許される。

3.5.1 文法の概観

MV 命令文の文法は複雑なため、まず例を数点用いて各部のおおまかな文法と効果を述べておく。

例 1

```
mvp/n64i01 $p0@0 $lc0@2.1
```

mvp/n64i01がオペコード、$p0@0が読み出し元オペランド、$lc0@2.1が書き込み先オペランドである。

オペコードは/の前後で基本モード部とパラメータ部に分かれる。

オペコードの基本モード部は、mvが MV 命令であることを表す固定文字列であり、pが基本モードが個別転送であることを示す。

オペコードのパラメータ部は、n64が転送語数が 64 長語であること、タグ番号が 0x01であることをそれぞれ示す。

例 2

```
mvp/n64 $lc0@2.1 $p0@0
```

例 1 と異なりタグが指定されていない。

タグについては、待ち合わせなしでも MV 命令同士の完了タイミングの前後関係が確定することがあるので、文法としては省略可能である。例えば、次は正しいアセンブリである。

```
mvp/n64 $lc0@2.1 $p0@0
mvp/n64i01 $lc64@2.1 $p64@0
nop; wait i01
```

例 3

```
mvrdfadd/n128 $lc0 $d0
```

縮約転送では精度指定が必要で、その後の演算指定と合わせて縮約演算の内容を決める。この場合は dすなわち倍精度の faddすなわち浮動小数点数加算を行う。

3.5.2 オペランド

可能な MV 命令はメモリ種別 (PDM、DRAM、L2BM) の組み合わせによって異なるため、まずオペランドの文法を示し、その後メモリ種別の組み合わせごとに可能な転送の文法と効果を示す。

3.5.2.1 p - PDM

PDM オペランドの文法は次の通りである。

```
  $p<addr>
| $p<addr>@<group>
```

<addr>は PDM 内の長語単位のアドレスを指定する自然数である。

<group>は 0 から 3 のグループ番号である。

<group>の付かない$p<addr>はすべてのグループの PDM の同じアドレスにアクセスすることを示す。

例

1 番グループの PDM の 64 番地から 128 長語を読み、2 番グループの DRAM の 32 番地以降に書き込む。

```
mvp/n0x80 $p0x40@1 $d0x20@2
```

3.5.2.2 d - DRAM

DRAM オペランドの文法は次の通りである。

```
  $d<addr>
| $d<addr>@<group>
| $di<dar_addr>
| $di<dar_addr>@<group>
```

最初の 2 つはアドレス直接指定によるアクセスである。$diから始まる後ろの 2 つは DRAM 間接参照を有効にする。

<addr>は DRAM 内の長語単位のアドレスを指定する自然数である。

<group>は 0 から 3 のグループ番号である。

<group>の付かない$d<addr>はすべてのグループの DRAM の同じアドレスにアクセスすることを示す。

<dar_addr>は DRAM 間接参照に必要な、DAR (DRAM アドレスレジスタ) の読み出し開始アドレスである。DRAM 間接参照については第 3.5.6 節で詳述する。

例

2 番グループの DRAM の 32 番地から 128 長語を読み、1 番グループの PDM の 64 番地以降に書き込む。

```
mvp/n0x80 $d0x20@2 $p0x40@1
```

3.5.2.3 c - MV 命令における L2BM

L2BM オペランドの文法は次の通りである。

```
  $lc<addr>
| $lc<addr>@.<l2b>
| $lc<addr>@<group>.<l2b>
```

<addr>は L2BM 内の長語単位のアドレスを指定する自然数である。

@以降の<group>と<l2b>はそれぞれ、0 から 3 のグループ番号と、0 か 1 の L2B 番号である。

<group>が付かない場合はすべてのグループに、<l2b>が付かない場合はグループ内の両方の L2BM にアクセスすることを示す。

例

全グループについて並行に、そのグループの DRAM の 32 番地から 64 長語を読み、そのグループの 1 番 L2BM の 0 番地以降に書き込む。

```
mvp/n0x40 $d0x20 $lc0@.1
```

3.5.3 転送語数指定と単位動作

MV 命令には転送語数を指定しなければならない。文法としてはオペコードのパラメータ部に n<size>を追加する。ここで<size>は L2BM が関わる転送では L2BM で数えた長語ワード数、PDM-DRAM 間転送では PDM で数えた長語ワード数である。

どの MV 命令も、DRAM 間接参照がオンの場合とアドレスのラップアラウンドを除き、読み込み元と書き込み先いずれでも連続の領域にアクセスする。

DRAM 間接参照がオンの場合を除き、MV 命令の転送語数を大きくした場合の結果は、元の命令に加えて、その命令でアクセスした領域の次をアドレスとして残りの語数を転送した場合に等しくなる。すなわち、次のふたつの例は等価である。

例 1

```
mvp/n192 $p0@0 $d0@1
```

例 2

```
mvp/n128 $p0@0 $d0@1
mvp/n64 $p128@0 $d128@1
```

転送語数は基本モードごとに最小の値が決まっており、その倍数しか指定できない。この最小の転送語数での動作を**単位動作**と呼ぶこととする。

3.5.4 タグ指定

wait命令を用いた待ち合わせのため、MV 命令にはタグを指定できるようになっている。文法としてはオペコードのパラメータ部に i01などのタグ (3.2.3 節) を追加すればよい。

3.5.5 縮約演算指定

縮約転送では、基本モード部の mvrの後ろに演算精度と演算種別を指定する。具体的な文法は以下である。d，f，hが浮動小数点数の倍・単・半精度、l，i，sが整数の倍・単・半精度である。

```
<rrn_opcode> ::=
| (d|f|h)fadd # floating-point add
| (d|f|h)max # max of floating-point values
| (d|f|h)min # min of floating-point values
| (l|i|s)iadd # integer add
| (l|i|s)band # bitwise logical and
| (l|i|s)and # logical and
| (l|i|s)bor # bitwise logical or
| (l|i|s)or # logical or
```

この指定は L2B 以下の縮約命令においても用いられる。

3.5.6 DRAM 間接参照

DRAM が関わる任意の MV 命令は DRAM 間接参照モードをオンにできる。

DRAM 間接参照モードでは、`l2bmdars`/`l2bmdarw`の 2 つの L2BM 命令 (第 3.6.7.10 節) によってあらかじめ DAR (DRAM **アドレスレジスタ**) に書き込んでおいたアドレス値を用いる。DAR はグループごとに存在する。よって、グループごとに異なる DRAM アドレスにアクセスすることが可能である。DAR は 32 ビットのアドレス語を 1 エントリとして 1024 エントリからなる。アドレス語の単位は 16 長語である[*2]。

間接参照の単位は 16 長語である。すなわち、MV 命令の基本モードに関わらず、DRAM に 16 長語アクセスするごとに次のアドレス語を用いる。

DRAM 間接参照モードをオンにした MV 命令では、DAR 読み出し開始アドレス M と DAR エントリ連続使用回数 N の 2 つのパラメータを指定する。このとき、16 長語アクセスの回数 i に対し、16 長語単位 DRAM アドレスの最終的な値は $\mathrm{DAR}[(M + i/N)\%1024] + i\%N$ となる。すなわち、N 回の間は DAR のあるエントリの値をオフセットとして DRAM を連続アクセスし、その後 DAR の次のエントリの値を新たなオフセットとし、以降繰り返す。

`l2bmdarw`命令により DAR に書き込んだデータが有効になる直前のサイクルまではその DAR のエントリからは前のデータが読める。

M は DRAM オペランドで指定する (第 3.5.2.2 節)。N は MV 命令のオプション部で `nd<N>`の形式で指定する。`nd<N>`を省略した場合 N は無限大、すなわちアドレスは $\mathrm{DAR}[M] + i$ となる。

第 3.5.8 節で述べる MV 命令の基本モードの解説では DRAM 間接参照モード時の記述は省略する。

例

```
mvp/n256nd4 $p1600@1 $di512@2
```

これは第 3.5.8.2 節で述べる PDM → DRAM 単独個別転送命令の DRAM 間接参照モード版である。効果は以下のようになる。

```
for i = 0:16
  uint_t src_addr = 1600 + 16 * i
  uint_t dst_addr = 16 * (MEM[2].dar[512+i/4] + (i % 4))
  LongWord data[16] = MEM[1].pdm[src_addr:src_addr+16]
  MEM[2].dram[dst_addr:dst_addr+16] = data[0:16]
```

3.5.7 優先度指定

`mvnop`以外の任意の MV 命令には 0 から 3 の**優先度**を設定できる。

詳細はチップ仕様書に譲るが、これは PDM、DRAM、L2BM のそれぞれにおいて、複数の MV 命令に由来するアクセスが実行可能ならば、優先度の数値の大きい方のアクセスが先に実行されるというものである。

これは MV 命令実行時間の最適化のためのオプションであり、待ち合わせ等が適切に行われている正しいアセンブリ列であれば、優先度の指定は計算結果には一切影響しない。

[*2] 16 長語 × 32 ビット分の容量の DRAM が実装されているわけではない。実装されていない領域に相当するビットは無視される。

デフォルトの優先度は 0 である。
第 3.5.8 節で述べる MV 命令の基本モードの解説では優先度の記述は省略する。

例

```
mvp/n256p3 $p0@0 $d0@0
```

優先度 3 の MV 命令を発行する。

3.5.8 MV 命令の基本モード

以下では MV 命令の基本モードについて、文法と効果を解説する。併記されているスループットは単独で発行した場合、つまり他の MV 命令の影響を考慮しない場合の数値である。

3.5.8.1 mvnop 命令

mvnopは何もしない MV 命令文である。パラメータやオペランドはない。通常ユーザが直接書く必要はなく、パックされた命令をディスアセンブルした場合などに出現する。

文法

```
mvnop
```

効果
何もしない。

3.5.8.2 PDM → DRAM 単独個別転送命令

指定したグループの PDM から指定したグループの DRAM にデータをコピーする。
単位動作は 64 長語である。
読み出し元と書き込み先のグループは同じでも別でもよい。
スループットはグループ内の場合 16 長語/サイクル、グループ間の場合 8 長語/サイクルである。

文法

```
mvp/n<size>[<tag>] $p<addr_p>@<group_p> $d<addr_d>@<group_d>
```

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_p + 64 * i
  uint_t dst_addr = addr_d + 64 * i
  LongWord data[64] = MEM[group_p].pdm[src_addr:src_addr+64]
  MEM[group_d].dram[dst_addr:dst_addr+64] = data[0:64]
```

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvp/n64 $p0@0 $d0@1
```

0 番グループの PDM から 1 番グループの DRAM に 64 長語コピーする。

3.5.8.3 DRAM → PDM 単独個別転送命令

指定したグループの DRAM から指定したグループの PDM にデータをコピーする。

単位動作は 64 長語である。

読み出し元と書き込み先のグループは同じでも別でもよい。

スループットはグループ内の場合 8 長語/サイクル、グループ間の場合 4 長語/サイクルである。

文法

```
mvp/n<size>[<tag>] $d<addr_d>@<group_d> $p<addr_p>@<group_p>
```

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_d + 64 * i
  uint_t dst_addr = addr_p + 64 * i
  Word64 data[64] = MEM[group_d].dram[src_addr:src_addr+64]
  MEM[group_p].pdm[dst_addr:dst_addr+64] = data[0:64]
```

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvp/n64 $d0@1 $p0@0
```

1 番グループの DRAM から 0 番グループの PDM に 64 長語コピーする。

3.5.8.4 PDM → L2BM 単独個別転送命令

指定したグループの PDM から指定したグループと L2B 番号の L2BM にデータをコピーする。

単位動作は 64 長語である。

読み出し元と書き込み先のグループは同じでも別でもよい。

スループットはグループ内の場合 16 長語/サイクル、グループ間の場合 8 長語/サイクルである。

すべてのグループについて同じアドレスでグループ内転送を行う場合、3.5.8.9 節で述べる並列版を利用した方がレイテンシの面で有利である。

文法

```
mvp/n<size>[<tag>] $p<addr_p>@<group_p> $lc<addr_c>@<group_c>.<l2b_c>
```

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_p + 64 * i
  uint_t dst_addr = addr_c + 64 * i
  LongWord data[64] = MEM[group_p].pdm[src_addr:src_addr+64]
  MEM[group_c][l2b_c].l2bm[dst_addr:dst_addr+64] = data
```

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvp/n64 $p0@0 $lc0@2.1
```

0 番グループの PDM から 2 番グループ・1 番 L2B の L2BM に 64 長語をコピーする。

3.5.8.5 L2BM → PDM 単独個別転送命令

指定したグループと L2B 番号の L2BM から指定したグループの PDM にデータをコピーする。

単位動作は 64 長語である。

読み出し元と書き込み先のグループは同じでも別でもよい。

スループットはグループ内の場合 16 長語/サイクル、グループ間の場合 8 長語/サイクルである。

すべてのグループについて同じアドレスでグループ内転送を行う場合、3.5.8.10 節で述べる並列版を利用した方がレイテンシの面で有利である。

文法

```
mvp/n<size>[<tag>] $lc<addr_c>@<group_c>.<l2b_c> $p<addr_p>@<group_p>
```

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_c + 64 * i
  uint_t dst_addr = addr_p + 64 * i
  LongWord data[64] = MEM[group_c][l2b_c].l2bm[src_addr:src_addr+64]
  MEM[group_p].pdm[dst_addr:dst_addr+64] = data
```

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvp/n64 $lc0@2.1 $p0@0
```

2 番グループ・1 番 L2B の L2BM から 0 番グループの PDM に 64 長語をコピーする。

3.5.8.6 DRAM → L2BM 単独個別転送命令

指定したグループの DRAM から、指定したグループ番号・L2B 番号の L2BM にデータをコピーする。

単位動作は 64 長語である。

スループットはグループ内の場合 16 長語/サイクル、グループ間の場合 8 長語/サイクルである。

すべてのグループについて同じアドレスでグループ内転送を行う場合、3.5.8.11 節で述べる並列版を利用した方がレイテンシの面で有利である。

文法

```
mvp/n<size>[<tag>] $d<addr_d>@<group_d> $lc<addr_c>@<group_c>.<l2b_c>
```

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_d + 64 * i
  uint_t dst_addr = addr_c + 64 * i
  LongWord data[64] = MEM[group_d].dram[src_addr:src_addr+64]
  MEM[group_c][l2b_c].l2bm[dst_addr:dst_addr+64] = data
```

エラー

- `size`が単位動作 64 の倍数でないとエラーになる。

例

```
mvp/n64 $d0@0 $lc0@2.1
```

0 番グループの DRAM から 2 番グループ・1 番 L2B の L2BM に 64 長語をコピーする。

3.5.8.7　L2BM → DRAM 単独個別転送命令

指定したグループ番号・L2B 番号の L2BM から、指定したグループの DRAM にデータをコピーする。

単位動作は 64 長語である。

スループットはグループ内の場合 16 長語/サイクル、グループ間の場合 8 長語/サイクルである。

すべてのグループについて同じアドレスでグループ内転送を行う場合、3.5.8.12 節で述べる並列版を利用した方がレイテンシの面で有利である。

文法

```
mvp/n<size>[<tag>] $lc<addr_c>@<group_c>.<l2b_c> $d<addr_d>@<group_d>
```

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_c + 64 * i
  uint_t dst_addr = addr_d + 64 * i
  LongWord data[64] = MEM[group_c][l2b_c].l2bm[src_addr:src_addr+64]
  MEM[group_d].dram[dst_addr:dst_addr+64] = data
```

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvp/n64 $lc0@2.1 $d0
```

2 番グループ・1 番 L2B の L2BM から 0 番グループの DRAM に 64 長語をコピーする。

3.5.8.8 PDM → PDM 単独個別転送命令

指定したグループの PDM から指定した異なるグループの PDM にデータをコピーする。

単位動作は 64 長語である。

スループットは 8 長語/サイクルである。

文法

```
mvp/n<size>[<tag>] $p<addr0>@<group0> $p<addr1>@<group1>
```

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr0 + 64 * i
  uint_t dst_addr = addr1 + 64 * i
  LongWord data[64] = MEM[group0].pdm[src_addr:src_addr+64]
  MEM[group1].pdm[dst_addr:dst_addr+64] = data[0:64]
```

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvp/n64 $p0@0 $p0@1
```

0 番グループの PDM から 1 番グループの PDM に 64 長語コピーする。

3.5.8.9 PDM → L2BM 並列個別転送命令

すべてのグループについて、PDM から同じグループの指定した L2B 番号の L2BM にデータをコピーする。
単位動作は 64 長語である。
スループットはグループあたり 16 長語/サイクルである。

文法

```
mvp/n<size>[<tag>] $p<addr_p> $lc<addr_c>@.<l2b_c>
```

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_p + 64 * i
  uint_t dst_addr = addr_c + 64 * i
  forall group
    LongWord data[64] = MEM[group].pdm[src_addr:src_addr+64]
    MEM[group][l2b_c].l2bm[dst_addr:dst_addr+64] = data
```

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvp/n64 $p0 $lc0@.1
```

すべてのグループについて、PDM から 1 番 L2B の L2BM に 64 長語をコピーする。

3.5.8.10 L2BM → PDM 並列個別転送命令

すべてのグループについて、指定した L2B 番号の L2BM から同じグループの PDM にデータをコピーする。

単位動作は 64 長語である。

スループットはグループあたり 16 長語/サイクルである。

文法

```
mvp/n<size>[<tag>] $lc<addr_c>@.<l2b_c> $p<addr_p>
```

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_c + 64 * i
  uint_t dst_addr = addr_p + 64 * i
  forall group
    LongWord data[64] = MEM[group][l2b_c].l2bm[src_addr:src_addr+64]
    MEM[group].pdm[dst_addr:dst_addr+64] = data
```

エラー

- `size`が単位動作 64 の倍数でないとエラーになる。

例

```
mvp/n64 $lc0@.1 $p0
```

すべてのグループについて、1 番 L2B の L2BM から PDM に 64 長語をコピーする。

3.5.8.11 DRAM → L2BM 並列個別転送命令

すべてのグループについて、DRAM から同じグループの指定した L2B 番号の L2BM にデータをコピーする。

単位動作は 64 長語である。

スループットはグループあたり 16 長語/サイクルである。

文法

```
mvp/n<size>[<tag>] $d<addr_d> $lc<addr_c>@.<l2b_c>
```

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_d + 64 * i
  uint_t dst_addr = addr_c + 64 * i
  forall group
    LongWord data[64] = MEM[group].dram[src_addr:src_addr+64]
    MEM[group][l2b_c].l2bm[dst_addr:dst_addr+64] = data
```

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvp/n64 $d0 $lc0@.1
```

すべてのグループについて、DRAM から 1 番 L2B の L2BM に 64 長語をコピーする。

3.5.8.12 L2BM → DRAM 並列個別転送命令

すべてのグループについて、指定した L2B 番号の L2BM から、同じグループの DRAM にデータをコピーする。

単位動作は 64 長語である。

スループットはグループあたり 16 長語/サイクルである。

文法

```
mvp/n<size>[<tag>] $lc<addr_c>@.<l2b_c> $d<addr_d>
```

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_c + 64 * i
  uint_t dst_addr = addr_d + 64 * i
  forall group
    LongWord data[64] = MEM[group][l2b_c].l2bm[src_addr:src_addr+64]
    MEM[group].dram[dst_addr:dst_addr+64] = data
```

エラー

- `size`が単位動作 64 の倍数でないとエラーになる。

例

```
mvp/n64 $lc0@.1 $d0
```

すべてのグループについて、1 番 L2B の L2BM から DRAM に 64 長語をコピーする。

3.5.8.13 DRAM → L2BM グループ内放送命令

すべてのグループについて、DRAM から同じグループの 2 つの L2BM にデータをコピーする。
単位動作は 64 長語である。
スループットは 16 長語/サイクルである。

文法

```
mvb2/n<size>[<tag>] $d<addr_d> $lc<addr_c>
```

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_d + 64 * i
  uint_t dst_addr = addr_c + 64 * i
  forall group
    LongWord data[64] = MEM[group].dram[src_addr:src_addr+64]
    forall l2b
      MEM[group][l2b].l2bm[dst_addr:dst_addr+64] = data
```

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvb2/n64 $d0 $lc0
```

すべてのグループについて、DRAM から 2 つの L2BM に 64 長語を放送する。

3.5.8.14 L2BM → DRAM グループ内縮約命令

すべてのグループについて、2 個両方の L2BM から等サイズで読んで L2B 方向に縮約し、同じグループの DRAM に書き込む。

単位動作は 64 長語である。

スループットは 16 長語/サイクルである。

文法

```
mvr2<op>/n<size>[<tag>] $lc<addr_c> $d<addr_p>
```

縮約演算指定<op>については 3.5.5 節を参照のこと。

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_c + 64 * i
  uint_t dst_addr = addr_d + 64 * i
  forall group
    LongWord buf[64] = [0, ..., 0]
    forall l2b
      buf[0:64] = op(buf[0:64], MEM[group][l2b].l2bm[src_addr:src_addr+64])
    MEM[group].dram[dst_addr:dst_addr+64] = buf
```

注意：縮約を内部的に実際にこの手順で行っているわけではない。

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvr2dfadd/n64 $lc0 $d0
```

すべてのグループについて、2 つの L2BM から 64 長語ずつ読んで倍精度浮動小数点数加算を行い、完成した 64 長語を DRAM に書き込む。

3.5.8.15　L2BM → PDM グループ内縮約命令

指定したグループについて、2 個両方の L2BM から等サイズで読んで L2B 方向に縮約し、同じグループの PDM に書き込む。

単位動作は 64 長語である。

スループットは 16 長語/サイクルである。

この命令には対となる PDM → L2BM グループ内放送命令が存在しないことに注意する。

文法

```
mvr2<op>/n<size>[<tag>] $lc<addr_c>@<group> $p<addr_p>@<group>
```

縮約演算指定<op>については 3.5.5 節を参照のこと。

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_c + 64 * i
  uint_t dst_addr = addr_p + 64 * i
  LongWord buf[64] = [0, ..., 0]
  forall l2b
    buf[0:64] = op(buf[0:64], MEM[group][l2b].l2bm[src_addr:src_addr+64])
  MEM[group].pdm[dst_addr:dst_addr+64] = buf
```

注意：縮約を内部的に実際にこの手順で行っているわけではない。

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvr2dfadd/n64 $lc0@1 $p0@1
```

1 番グループについて、2 つの L2BM から 64 長語ずつ読んで倍精度浮動小数点数加算を行い、完成した 64 長語を 1 番 PDM に書き込む。

3.5.8.16　DRAM → L2BM グループ間分配放送命令

この命令は複雑なのでまず具体的に、単位動作である L2BM 側 64 長語書き込みの動作を説明する。各グループの DRAM から 32 長語ずつ読み、前半 16 長語をグループ順に並べた 64 長語を全グループの 0 番目の L2BM に放送する。後半 16 長語を同様に並べた 64 長語を全グループの 1 番目の L2BM に放送する。

単位動作でない場合はこれを繰り返す。

スループットは DRAM 側 8 長語/サイクル、L2BM 側 16 長語/サイクルである。

文法

```
mvb4/n<size>[<tag>] $d<addr_d> $lc<addr_c>
```

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_offset = addr_d + 32 * i
  uint_t dst_addr = addr_c + 64 * i
  LongWord buf[2][64]
  forall group,l2b
    uint_t src_addr = src_offset + l2b * 16
    uint_t buf_addr = group * 16
    buf[l2b][buf_addr:buf_addr+16] = MEM[group].dram[src_addr:src_addr+16]

  forall group,l2b
    MEM[group][l2b].l2bm[dst_addr:dst_addr+64] = buf[l2b][0:64]
```

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvb4/n64 $d0 $lc0
```

本節冒頭で延べた単位動作である。

3.5.8.17　L2BM → DRAM グループ間結合縮約命令

この命令は複雑なのでまず具体的に、単位動作である L2BM 側 64 長語読み出しの動作を説明する。全 L2BM から 64 長語ずつ読み、グループ方向に縮約する。全グループの 0 番 L2B のデータを縮約した 64 長語を 16 長語ずつに分割し、各 DRAM に書き込む。同様に全グループの 1 番 L2B のデータを縮約した 64 長語を 16 長語ずつに分割し、各 DRAM に書き込む。以上で各 DRAM に 32 長語が書き込まれる。

単位動作でない場合はこれを繰り返す。

スループットは L2BM 側 16 長語/サイクル、DRAM 側 8 長語/サイクルである。

文法

```
mvr4<op>/n<size>[<tag>] $lc<addr_c> $d<addr_d>
```

縮約演算指定<op>については 3.5.5 節を参照のこと。

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_c + 64 * i
  uint_t dst_offset = addr_d + 32 * i
  LongWord buf[2][64]
  forall group,l2b
    buf[l2b][0:64] = op(buf[l2b][0:64], MEM[group][l2b].l2bm[src_addr:src_addr+64])

  forall group,l2b
    uint_t buf_addr = 16 * group
    uint_t dst_addr = dst_offset + 16 * l2b
    MEM[group].dram[dst_addr:dst_addr+16] = buf[l2b][buf_addr:buf_addr+16]
```

注意：縮約を内部的に実際にこの手順で行っているわけではない。

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvr4dfadd/n64 $lc0 $d0
```

本節冒頭で延べた単位動作である。縮約には倍精度浮動小数点数加算を用いている。

3.5.8.18 PDM → L2BM グループ間放送命令

0 番グループの PDM から 8 個すべての L2BM にデータを放送する。

単位動作は 64 長語である。

スループットは 8 長語/サイクルである。

文法

```
mvb/n<size>[<tag>] $p<addr_p>@<group_p> $lc<addr_c>
```

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_p + 64 * i
  uint_t dst_addr = addr_c + 64 * i
  LongWord data[64] = MEM[group_p].pdm[src_addr:src_addr+64]
  forall group,l2b
    MEM[group][l2b].l2bm[dst_addr:dst_addr+64] = data
```

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvb/n64 $p0@0 $lc0
```

0 番グループの PDM から 8 個すべての L2BM に 64 長語を放送する。

3.5.8.19　L2BM → PDM グループ間縮約命令

8 個すべての L2BM から等サイズで読んで L2B 方向に縮約し、指定したグループの PDM に書き込む。
単位動作は 64 長語である。
スループットは 8 長語/サイクルである。

文法

```
mvr<op>/n<size>[<tag>] $lc<addr_c> $p<addr_p>@<group_p>
```

縮約演算指定<op>については 3.5.5 節を参照のこと。

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_c + 64 * i
  uint_t dst_addr = addr_p + 64 * i
  LongWord buf[64]
  forall group,l2b
    buf[0:64] = op(buf[0:64], MEM[group][l2b].l2bm[src_addr:src_addr+64])
  MEM[group_p].pdm[dst_addr:dst_addr+64] = buf
```

注意：縮約を内部的に実際にこの手順で行っているわけではない。

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvrdfadd/n64 $lc0 $p0@0
```

8 個すべての L2BM から 64 長語ずつ読んで、L2B 番号方向に倍精度浮動小数点数加算で縮約し、0 番グループの PDM に書き込む。

3.5.8.20 DRAM → L2BM グループ間放送命令

すべてのグループの DRAM から等サイズに読んで結合し、すべての L2BM に放送する。

単位動作は DRAM 側 16 長語、L2BM 側 64 長語である。

スループットは L2BM 側で数えて 32 長語/サイクルである。

文法

```
mvb/n<size>[<tag>] $d<addr_d> $lc<addr_c>
```

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_d + 16 * i
  uint_t dst_addr = addr_c + 64 * i
  LongWord buf[64]
  for group_dram = 0:4
    uint_t buf_addr = group_dram * 16
    buf[buf_addr:buf_addr+16] = MEM[group_dram].dram[src_addr:src_addr+16]

  forall group,l2b
    MEM[group][l2b].l2bm[dst_addr:dst_addr+64] = buf[0:64]
```

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvb/n64 $d0 $lc0
```

すべてのグループの DRAM から 16 長語ずつ読んで結合した 64 長語をすべての L2BM に放送する。

3.5.8.21 L2BM → DRAM グループ間縮約命令

すべての L2BM から等サイズに読んで L2B 方向に縮約し、4 つの DRAM に等分配する。
単位動作は L2BM 側 64 長語、DRAM 側 16 長語である。
スループットは L2BM 側で数えて 32 長語/サイクルである。

文法

```
mvr<op>/n<size>[<tag>] $lc<addr_c> $d<addr_d>
```

縮約演算指定<op>については 3.5.5 節を参照のこと。

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_c + 64 * i
  uint_t dst_oddr = addr_d + 16 * i
  LongWord buf[64]
  forall group
    forall l2b
      buf[0:64] = op(buf[0:64], MEM[group][l2b].l2bm[src_addr:src_addr+64]

  forall group
    uint_t buf_addr = 16 * group
    MEM[group].dram[dst_addr:dst_addr+16] = buf[buf_addr:buf_addr+16]
```

注意：縮約を内部的に実際にこの手順で行っているわけではない。

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvrdfadd/n64 $lc0 $d0
```

8 個すべての L2BM から 64 長語ずつ読んで L2B 方向に縮約した 64 長語を、各グループ 16 長語ずつにして DRAM に書き込む。

3.5.8.22 PDM → L2BM 分配命令

指定したグループの PDM から読み出したデータを 8 分割して各グループの L2BM に書き込む。

単位動作は PDM 側 512 長語、L2BM 側 64 長語である。分割方法は 16 長語単位で 0 から 7 の L2B 通し番号方向にラウンドロビンである。詳細は効果のパートを参照のこと。

スループットは PDM 側 8 長語/サイクル、L2BM 側 1 長語/サイクルである。

文法

```
mvd/n<size>[<tag>] $p<addr_p>@<group_p> $lc<addr_c>
```

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_p + 512 * i
  uint_t dst_offset = addr_c + 64 * i
  LongWord buf[512] = MEM[group_p].pdm[src_addr:src_addr+512]
  for j = 0:4
    forall group,l2b
      uint_t buf_addr = (j * 8 + group * 2 + l2b) * 16
      uint_t dst_addr = dst_offset + j * 16
      MEM[group][l2b].l2bm[dst_addr:dst_addr+16] = buf[buf_addr:buf_addr+16]
```

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvd/n64 $p0@0 $lc0
```

0 番グループの PDM から 512 長語を読んで各 L2BM に 64 長語ずつ書き込む。

3.5.8.23 L2BM → PDM 結合命令

全 L2B から読み出したデータを結合して指定したグループの PDM に書き込む。

単位動作は L2BM 側 64 長語、PDM 側 512 長語である。結合方法は 16 長語単位で 0 から 7 の L2B 通し番号方向にラウンドロビンである。詳細は効果のパートを参照のこと。

スループットは L2BM 側 1 長語/サイクル、PDM 側 8 長語/サイクルである。

文法

```
mvd/n<size>[<tag>] $lc<addr_c> $p<addr_p>@<group_p>
```

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_offset = addr_c + 64 * i
  uint_t dst_addr = addr_p + 512 * i
  LongWord buf[512]
  for j = 0:4
    forall group,l2b
      uint_t src_addr = src_offset + j * 16
      uint_t buf_addr = (j * 8 + group * 2 + l2b) * 16
      buf[buf_addr:buf_addr+16] = MEM[group][l2b].l2bm[src_addr:src_addr+16]
  MEM[group_p].pdm[dst_addr:dst_addr+512] = buf[0:512]
```

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvd/n64 $lc0 $p0@0
```

各 L2BM から 64 長語ずつ読んで結合した 512 長語を 0 番グループの PDM に書き込む。

3.5.8.24 PDM → DRAM 分配命令

指定したグループの PDM から読み出したデータを 4 分割して各グループの DRAM に書き込む。
単位動作は PDM 側 64 長語、DRAM 側 16 長語である。
スループットは PDM 側 8 長語/サイクル、DRAM 側 2 長語/サイクルである。

文法

```
mvd/n<size>[<tag>] $p<addr_p>@<group_p> $d<addr_d>
```

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_p + 64 * i
  uint_t dst_addr = addr_d + 16 * i
  LongWord buf[64] = MEM[group_p].pdm[src_addr:src_addr+64]
  forall group
    uint_t buf_addr = group * 16
    MEM[group].dram[dst_addr:dst_addr+16] = buf[buf_addr:buf_addr+16]
```

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvd/n64 $p0@0 $d0
```

0 番グループの PDM から 64 長語を読み出して各 DRAM に 16 長語ずつ書き込む。

3.5.8.25 DRAM → PDM 結合命令

各グループの DRAM から読み出したデータを結合して指定したグループの PDM に書き込む。
単位動作は DRAM 側 16 長語、PDM 側 64 長語である。
スループットは DRAM 側 2 長語/サイクル、PDM 側 8 長語/サイクルである。

文法

```
mvd/n<size>[<tag>] $d<addr_d> $p<addr_p>@<group_p>
```

効果

```
uint_t n = size / 64
for i = 0:n
  uint_t src_addr = addr_d + 16 * i
  uint_t dst_addr = addr_p + 64 * i
  LongWord buf[64]
  forall group
    uint_t buf_addr = group * 16
    buf[buf_addr:buf_addr+16] = MEM[group].dram[src_addr:src_addr+16]
  MEM[group_p].pdm[dst_addr:dst_addr+64] = buf[0:64]
```

エラー

- sizeが単位動作 64 の倍数でないとエラーになる。

例

```
mvd/n64 $d0 $p0@0
```

各 DRAM から 16 長語ずつ読み出し、結合して 0 番グループの PDM に 64 長語を書き込む。

3.5.9 複数 MV 命令間の制約

以上で述べた単独の MV 命令にかかる制約以外にも、複数の MV 命令を同時に発行する場合に初めて問題になる制約が存在する。

アセンブラは適切なオプションを指定すればこの制約をチェックしてエラーとして報告するので具体的には別文書に譲る。

3.6 PE 命令文

PE 命令文は 1 ステップに実行される PE 命令に翻訳される単位である。PE 命令文は、いくつかの PE 命令式をセミコロンで区切り、改行で終端する形式を持つ。

PE 命令式は、`nop`命令、`noforward`命令、`l2bmdarw`命令、`wait`命令を除き、1 つのオペコード、0 個以上の入力オペランド、1 個以上の出力オペランドが、1 つ以上の空白文字によって区切られた次のような文法を持つ。

```
<opcode> <in-operand-1> ... <in-operand-k> <out-operand-1> [<out-operand-2> ...]
```

ある式がいくつの入力オペランドを持つかはオペコードによって決まる。出力オペランドは他の制限が許すかぎり任意の個数を指定できる。

`nop`命令、`noforward`命令、`l2bmdarw`命令、`wait`命令は入出力値を持たないので上記の文法にあてはまらない。

セミコロン区切りで 1 行に複数の式を列挙することで、翻訳結果のビットフィールドが重ならないなどの条件のもとで、1 命令として発行できる。

以下にいくつかの例を示す。オペコード、オペランドの詳細な文法は後述する。

例 1

dvpassa は MAU によりデータのコピーを行う 1 入力オペランドの命令である。以下の例では LM0 (`$lm0v`) から LM1 (`$ln0v`) にコピーを行う。

```
dvpassa $lm0v $ln0v
```

例 2

以下は出力オペランドが 2 つある例である。LM0 から LM1 と GRF0 (`$lr0v`) にコピーを行う。

```
dvpassa $lm0v $ln0v $lr0v
```

例 3

linc は ALU により倍精度整数のインクリメントを行う 1 入力オペランドの命令である。以下の例では例 2 の効果に加え、T レジスタ (`$t`) から読み出したデータをインクリメントして GRF1 (`$ls0v`) に格納する。

```
dvpassa $lm0v $ln0v $lr0v; linc $t $ls0v
```

3.6.1 オペランド

3.6.1.1 c - PE 命令における L2BM (l2bmdars 命令を除く)

l2bmdars 命令 (3.6.7.10 節) を除き、PE 命令における L2BM オペランドの文法は次の通りである。

```
$lc<addr>
```

addrは L2BM 内の長語単位のアドレスである。

L2BM は LM などの PE メモリと違いアドレスインクリメントを指定できず、L2BM 命令の内容に依存して、連続領域を重複なくアクセスするように自動的に決定される。

L2BM を構成する SRAM のポートは MV 命令によりアクセスする側と L2BM 命令によりアクセスする側で分かれており、並行にアクセスできる。それぞれの側は 1R/1W であり、例えば L1BM への転送の読み出しと L1BM からの転送の書き込みを同時に行うことはできない。

3.6.1.2 c - l2bmdars 命令における L2BM

l2bmdars 命令 (3.6.7.10 節) の入力オペランドにした際の L2BM は、他の L2BM 命令のオペランドとは異なり次の文法を用いる。

```
$lc<addr>@.<l2b>
```

グループ内のどちらの L2B のデータを DAR (3.6.1.3 節) に転送するかの指定のために L2B 番号のフィールドが追加されている。

3.6.1.3 dar - DRAM アドレスレジスタ

DAR (DRAM アドレスレジスタ) は、l2bmdars 命令 (3.6.7.10 節) の出力にのみ用いられるオペランドである。文法は次の通りである。

```
$dar<addr>
```

<addr>は DAR の書き込み開始エントリ番号である。

DAR はグループあたり 1 つ存在し、それぞれ 1024 エントリである。

3.6.1.4 b - L2BM 命令における L1BM

L2BM 命令における L1BM オペランドの文法は次の通りである。

```
$lb<addr>
```

addrは L1BM 内の長語単位のアドレスである。

L1BM は LM などの PE メモリと違いアドレスインクリメントを指定できず、L2BM 命令の内容に依存して、連続領域を重複なくアクセスするように自動的に決定される。これは 3.6.1.5 節で述べる、L1BM 命令における L1BM でも同様である。

L1BM を構成する SRAM のポートは L2BM 命令によりアクセスする側と L1BM 命令によりアクセスする側で分かれており、並行にアクセスできる。それぞれの側は 1R/1W であり、例えば L2BM への転送の読み出しと L2BM からの転送の書き込みを同時に行うことはできない。ただし、内部マルチキャスト命令 (3.6.7.9

節) は読み出しを行う L1BM と書き込みを行う L1BM が分かれているので、L2BM 命令によりアクセスする側のポートのみを用いて実現されている。

3.6.1.5 b - L1BM 命令における L1BM

L1BM 命令における L1BM オペランドの文法は次の通りである。

```
$(l|ll)b(<addr>|i)
```

(l|ll)はアクセス語長指定である。lが長語、llが 2 長語となる。後半は<addr>で L1BM 内の長語単位のアドレスを指定するか、iで折り返しレジスタを指定する。

3.6.1.4 で述べた理由により、PE への転送の読み出しと PE からの転送の書き込みを同時に行うことはできない。ただし、3.6.7.9 節で述べる折り返しレジスタにより、制限付きでこれらの同時発行が可能である。

3.6.1.6 m - LM0 (ベースアドレスレジスタ書き込みを除く)

ベースアドレスレジスタ (BAR) への書き込みの場合を除き、LM0 オペランドの文法は次の通りである。

```
  $[(l|ll)]m(<addr>[v[<adri>]][j<madpe>]|t<addr>[v[<adri>]]) # Auto stride mode
| $[(l|ll)]m(<flat_addrs>[j<madpe>]|t<flat_addrs>) # Flat mode
```

1 番目の文法が Auto stride モード、2 番目の文法が Flat モードである (両モードについては第 1.2 節を参照)。

LM0 のベースアドレスレジスタへの書き込みについては 3.6.1.7 節で述べる。

また、ここでは入力の符号反転 (3.6.9.10 節)、精度拡張 (3.6.9.11 節)、丸め (3.6.9.12 節)、出力のマスク適用 (3.6.2.1 節) については省略する。括弧で付したそれぞれの節を参照のこと。

[(l|ll)]はアクセス語長指定である。空であれば短語、lであれば長語、llであれば 2 長語となる。

後半は通常アドレッシング (Auto stride モード<addr>[v[<adri>]][j<madpe>]または Flat モード<flat_addrs>[j<madpe>]) と T レジスタ間接参照 (t<addr>[v[<adri>]]または t<flat_addrs>) の 2 パターンに分かれる。まず<addr>などの各パートについて述べる。

Auto stride モードでは、<addr>により LM0 内の短語単位のアドレスオフセットを指定する。1 命令 4 サイクルの間この値が最終的なアドレス値に加算される。

vはサイクル間のアドレスインクリメントを指定する。vを付けなかった場合、インクリメント幅は 0 となる。<adri>は短語単位のインクリメント幅で、値を省略するとアクセス語長での 1 語分、すなわち連続領域を重複なくアクセスするように設定される。

Flat モードでは、<flat_addrs>にて [<addr0>,<addr1>,<addr2>,<addr3>]の形式で各サイクルの短語単位アドレスを直接指定する。ここで角括弧 ([]) は実際にこの記号を記述することを意味し、オプションの意味ではない。

j<madpe>は MAB 内アドレス修飾を有効にする。これは、<madpe>で指定した番号以下の PE でのみ、アドレスをアクセス語長での 1 語分インクリメントする。

tは T レジスタ間接参照を有効にする。各サイクルで T レジスタから読み出した短語単位の値が最終的なアドレス値に加算される。

さらに、アドレスは常に暗黙にベースアドレスレジスタから読み出した短語単位のアドレス値が加算される。

以上をまとめると、Auto stride モードでの短語単位アドレスは以下で決定される。

$$\mathrm{BAR} + (\mathrm{T[C]\ if\ TI\ else\ 0}) + \mathrm{ADDR} + \mathrm{ADRI} \times \mathrm{C} \times \mathrm{WL} + (\mathrm{WL\ if\ MAADJ\ and\ PE} \leq \mathrm{MADPE\ else\ 0})$$

ここで、

- BAR: ベースアドレスレジスタから読み出した値
- C: 0 から 3 のサイクル番号
- T[C]: T レジスタからそのサイクルで読み出した値
- TI: T レジスタ間接参照が有効のとき真
- ADDR: `<addr>`の値
- ADRI: `<adri>`の値
- WL: 短語アクセス、長語アクセス、2 長語アクセスそれぞれで 1, 2, 4
- MAADJ: MAB 内アドレス修飾が有効のとき真
- MADPE: `madpe`の値
- PE: 0 から 3 の PE 番号

である。Flat モードでは ADDR + ADRI × C の部分を各サイクルで直接指定すると読み替えればよい。

アドレスは各サイクルでアクセス語長にアラインしていないとならない。よって、Auto stride モードでは `addr`と `adri`はいずれも、長語アクセスなら 2の倍数、2 長語アクセスなら 4の倍数になっていないとならない。これらはアセンブラによりチェックされるが、ベースアドレスレジスタの値のみは実行時に決まるのでアラインしていない可能性がある。その場合は端数は切り捨てられる。

例

```
lpassa $lm[0,4,10,14] $ln0v
```

LM0 から LM1 に計 4 長語をコピーする。LM0 に Flat モードを用いているので、これは Flat モードでしかアセンブルできない。その際、LM1 のアドレスも Flat モードの等価な指定に置き換えられる。つまり、これは次に等しい。

```
lpassa $lm[0,4,10,14] $ln[0,2,4,6]
```

3.6.1.7 m - LM0 (ベースアドレスレジスタ書き込み)

LM0 のベースアドレスレジスタ (BAR) への書き込みを行うためのオペランドの文法は次の通りである。

```
$[l]mb
```

ここではマスク適用 (3.6.2.1 節) については省略する。括弧で付した節を参照のこと。

これは出力専用オペランドである。

`l`が付けば長語アクセス、付かなければ短語アクセスとなる。

このオペランドを演算器の出力先に取ると、アクセス語長に応じて、MSB 側 1 語の LSB 側 12 ビットが LM0 のベースアドレスレジスタに書き込まれる。

3.6.1.8 n - LM1 (ベースアドレスレジスタ書き込みを除く)

ベースアドレスレジスタ (BAR) への書き込みの場合を除き、LM0 オペランドの文法は次の通りである。

```
  $[(l|ll)]n<addr>[v[<adri>]][j<madpe>] # Auto stride mode
| $[(l|ll)]n<flat_addrs>[j<madpe>] # Flat mode
```

1 番目の文法が Auto stride モード、2 番目の文法が Flat モードである (両モードについては第 1.2 節を参照)。

LM1 のベースアドレスレジスタへの書き込みについては 3.6.1.9 節で述べる。

また、ここでは入力の符号反転 (3.6.9.10 節)、精度拡張 (3.6.9.11 節)、丸め (3.6.9.12 節)、出力のマスク適用 (3.6.2.1 節) については省略する。括弧で付したそれぞれの節を参照のこと。

効果については、LM1 では T レジスタ間接参照が使用できないことを除き、3.6.1.6 節で述べた LM0 の場合と同様である。

3.6.1.9 n - LM1 (ベースアドレスレジスタ書き込み)

LM1 のベースアドレスレジスタ (BAR) への書き込みを行うためのオペランドの文法は次の通りである。

```
$[l]nb
```

ここではマスク適用 (3.6.2.1 節) については省略する。括弧で付した節を参照のこと。

効果については LM0 のもの (3.6.1.7 節) と同様である。

3.6.1.10 r - GRF0

GRF0 オペランドの文法は次の通りである。

```
  $[(l|ll)]r<addr>[v[<adri>]] # Auto stride mode
| $[(l|ll)]r<flat_addrs> # Flat mode
```

1 番目の文法が Auto stride モード、2 番目の文法が Flat モードである (両モードについては第 1.2 節を参照)。

ここでは入力の符号反転 (3.6.9.10 節)、精度拡張 (3.6.9.11 節)、丸め (3.6.9.12 節)、出力のマスク適用 (3.6.2.1 節) については省略する。括弧で付したそれぞれの節を参照のこと。

`[(l|ll)]`はアクセス語長指定である。空であれば短語、`l`であれば長語、`ll`であれば 2 長語となる。

Auto stride モードでは、`<addr>`により GRF0 内の短語単位のアドレスを指定する。1 命令 4 サイクルの間この値が最終的なアドレス値に加算される。

`v`はサイクル間のアドレスインクリメントを指定する。`v`を付けなかった場合、インクリメント幅は 0 となる。`<adri>`は短語単位のインクリメント幅で、値を省略するとアクセス語長での 1 語分、すなわち連続領域を重複なくアクセスするように設定される。

Flat モードでは、`<flat_addrs>`にて `[<addr0>,<addr1>,<addr2>,<addr3>]`の形式で各サイクルのアドレスを直接指定する。ここで角括弧 ([]) は実際にこの記号を記述することを意味し、オプションの意味ではない。

アドレスは各サイクルでアクセス語長にアラインしていないとならない。よって、Auto stride モードでは `addr`と `adri`はいずれも、長語アクセスなら 2の倍数、2 長語アクセスなら 4の倍数になっていないとならない。

3.6.1.11 s - GRF1

GRF1 オペランドの文法は次の通りである。

```
  $[(l|ll)]s<addr>[v[<adri>]] # Auto stride mode
| $[(l|ll)]s<flat_addrs> # Flat mode
```

1 番目の文法が Auto stride モード、2 番目の文法が Flat モードである (両モードについては第 1.2 節を参照)。

ここでは入力の符号反転 (3.6.9.10 節)、精度拡張 (3.6.9.11 節)、丸め (3.6.9.12 節)、出力のマスク適用 (3.6.2.1 節) については省略する。括弧で付したそれぞれの節を参照のこと。

効果は 3.6.1.10 節で述べた GRF0 オペランドと同様である。

3.6.1.12 t - T レジスタ

T レジスタオペランドの文法は次の通りである。

```
$[(l|ll)]t
```

ここでは入力の符号反転 (3.6.9.10 節)、精度拡張 (3.6.9.11 節)、丸め (3.6.9.12 節)、出力のマスク適用 (3.6.2.1 節) については省略する。括弧で付したそれぞれの節を参照のこと。

`[(l|ll)]`は記述可能だが無視される。T レジスタは常に 2 長語アクセスとなる。

T レジスタにはアドレス指定はない。2 長語 x 4 サイクル分のサイズがあり、自動的に現在のサイクルに対応する領域にアクセスする。

3.6.1.13 omr - マスクレジスタへの書き込み

マスクレジスタオペランドの文法は次の通りである。

```
$omr<addr>
```

これは出力専用オペランドである。

ここではマスク適用 (3.6.2.1 節) については省略する。括弧で付した節を参照のこと。

マスクはマスクレジスタ書き込み時にも適用できることに注意する。

マスクレジスタを出力に指定できるのは MAU 命令式か ALU 命令式のみである。生成されるマスク値の詳細はそれぞれの命令式の節で解説する。

3.6.1.14 x, y - 行列レジスタ

行列レジスタオペランドの文法は次の通りである。

```
  $(l|ll)(x|y)<addr> # (1) write or transposed read
| $l(x|y) # (2) matvec
```

(1) は行列レジスタ書き込み (第 3.6.10 節) および転置読み出し (第 3.6.11 節)、(2) は行列ベクトル積演算実行 (第 3.6.9 節) のときの文法である。

`(l|ll)`はアクセス語長指定である。`l`が長語、`ll`が 2 長語となる。アクセス語長を 2 長語にできるのは (1) でオペコードの精度指定が半精度の場合 (すなわち hmwrite か hmread) のみで、かつ半精度転置読み出しの

場合 (hmread) では 2 長語にしなければならない。

(x|y) は 2 面ある行列レジスタのどちらにアクセスするかを指定する。

<addr>は書き込みにおいては行番号、転置読み出しにおいては列番号である。(2) においては行列レジスタの 1 面全体が使われるのでアドレス指定はない。

LM などの PE メモリと違いアドレスインクリメントを指定できず、連続する行 (列) を重複なくアクセスするように語長に合わせて各サイクルでインクリメントされる。行番号 (列番号) が精度ごとの行数 (列数) を超える場合はラップアラウンドする。行番号 (列番号) は各サイクルでアクセス語長にアラインしていないとならない。すなわち、2 長語アクセスの場合は<addr>は 2の倍数になっていないとならない。

行列ベクトル積和演算は行列レジスタに書き込んだ直後のステップから実行可能である。

行列レジスタは精度ごとに実体があるわけではなく、また書き込んだ精度の情報は記憶していない。書き込みと読み出しで精度が一貫していなかった場合の動作は未定義である。

3.6.1.15 mauf - MAU 演算結果フォワーディング

次のオペランドからは nopと noforwardを除いた直前のステップで MAU が出力したデータを読み出せる。

```
$mauf
```

より詳しくは、第 iサイクルでは直前のステップの第 iサイクルでの MAU の出力が読み出せる。

語長は常に 2 長語であり、語長指定はできない。すなわち、$maufから読み出される値は、MAU の演算結果を一度$llr0vに書き、適切にステップを空けてから同オペランドから読み出したものに等しい。

$maufは同じステップで複数回現れてもよい。

nop命令か noforward命令があるステップでは$maufの値は更新されない。これは陽に書かれた nop命令でも、命令発行ユニットによって自動的に挿入される nop命令でも同様である。

これは読み出し専用のオペランドである。

例

```
fmfma $lx $lm0v $lr0v $lr0v
l1bmrffadd $mauf $lb0
```

単精度行列ベクトル積 FMA の結果を GRF0 に書き込み、次のステップでそれを MAB 間で足し合わせた結果を L1BM に書き込む。例えば$lb0から$lb3には$lr0に書き込まれた値を足し合わせた結果が書き込まれる。

3.6.1.16 aluf - ALU 演算結果フォワーディング

次のオペランドからは nopと noforwardを除いた直前のステップで ALU が出力したデータを読み出せる。データ生成元以外の性質は MAU 演算結果フォワーディング (3.6.1.15 節) と同じである。

```
$aluf
```

例

```
dbfn $lr0v $lr0v
dmwrite $aluf $lx0
```

倍精度 BF 化の結果を GRF0 に書き込み、次のステップでそれを行列レジスタに書き込む。

3.6.1.17 lbf - L1BM → PE 方向転送フォワーディング

次のオペランドからは nopと noforwardを除いた直前のステップで L1BM から PE に転送されたデータを読み出せる。データ生成元以外の性質は MAU 演算結果フォワーディング (3.6.1.15 節) と同じである。

```
$lbf
```

例

```
l1bmm $lb0 $lr0v
dvadd $lbf $lbf $ls0v
```

L1BM から GRF0 に MAB 放送を行い、次のステップでその 2 倍を計算して GRF1 に書き込む。

3.6.1.18 mreadf - 行列レジスタ転置読み出しフォワーディング

次のオペランドからは nopと noforwardを除いた直前のステップで行列レジスタ転置読み出しで出力されたデータを読み出せる。

これは ALU の最初の入力オペランドでしか使用できない。

データ生成元とオペランド位置制約以外の性質は MAU 演算結果フォワーディング (3.6.1.15 節) と同じである。

```
$mreadf
```

エラー

ALU の最初の入力オペランド以外で用いるとエラーになる。例えば drelu $lr0v $mreadf $ls0vはエラーになる。

例

```
dmread $lx0 $lr0v
dbfn $mreadf $ls0v
```

倍精度行列レジスタ転置読み出しの結果を GRF0 に書き込み、次のステップでその値をブロックフロート化して GRF1 に書き込む。

3.6.1.19 nowrite - フォワーディング用ダミー出力

フォワーディングパスの存在により、あるステップで演算は行うが結果はどのメモリにも書き込まないということがありうる。その際は唯一の出力オペランドとして次を指定する必要がある。

```
$nowrite
```

エラー

ひとつのオペコードに$nowriteと同時に別の出力オペランドを指定した場合、エラーになる。

例

```
imm f"1.0" $nowrite
fvadd $lm0v $aluf $lr0v
```

ALU 命令で生成した即値 1.0 を次のステップでフォワーディングパスから読み出して、LM0 から読んだ値に加算し、GRF0 に書き込む。

3.6.1.20 固定値入力オペランド

ALU の最初の入力オペランドに限り、いくつかの固定値を選択できる。一覧を表 3.5 に示す。

表 3.5 固定値入力オペランドの一覧

オペランド	値
$l2bid	自身のグループ番号 ×2+ L2B 番号 (3bit)
$l1bid	自身の L1B 番号 (3bit)
$mabid	自身の MAB 番号 (4bit)
$peid	自身の MAB 番号 ×4+ 自身の PE 番号 (6bit)
$subpeid	自身の PE 番号 (2bit)
$msb1	MSB だけ 1 で残りは 0 の値

いずれの固定値も、ALU 命令の精度指定に従って 2 長語分を並べたものが入力になる。

例

```
ipassa $msb1 $llm0
```

$llm0には短語 0x80000000が 4 つ並んだ 2 長語が入る。

3.6.2 マスクレジスタ

マスクレジスタは通常のデータではなくフラグを保持する特殊な PE メモリであり、PE メモリ書き込み時または演算器からの演算結果出力時に、動的に決定した箇所について書き込みのスキップまたは演算結果のゼロフラッシュが可能である。これにより擬似的な条件分岐を実現できる。

マスクレジスタは 32 エントリ存在し、15 エントリは可変であり、残りは固定値でマスク作用時の読み出し専用である。

1 エントリはサイクル方向に 4 サイクル分、ワード方向に 4 ビットで計 16 ビットからなる。

書き込みマスク適用時には、対応するエントリが 1 なら書き込みを行い、0 なら書き込みを行わない。ゼロフラッシュマスク適用時には、対応するエントリが 1 なら何もせず、0 なら書き込み前に値を 0 にする。

「サイクル方向」とは、マスクレジスタへの書き込み時と読み出し時ともに、1 ステップのうち何サイクル目かによって異なるエントリにアクセスするということを意味する。

「ワード方向」の意味はマスク作用時に選択できる**マスク作用語長**によって異なる。マスク作用語長が長語のとき、1 サイクルで PE 内データパスを通る 2 長語について、MSB 側 1 長語を 4 半語とみなして 4 ビットのフラグを対応させてマスクを作用させ、LSB 側 1 長語には干渉しない。マスク作用語長が 2 長語のとき、同じ 2 長語について、4 短語とみなしてマスクを作用させる。

固定値エントリのアドレスと内容は以下である[*3]。

- アドレス 0: All 1 (書き込みマスクについては常に書き込みを行う、演算器ゼロフラッシュにおいては一切ゼロフラッシュを行わない)
- アドレス 16 以降: アドレス値の下位 4 ビットがそのまま、上位ビットから順に各サイクルのマスクフラグになる。ワード方向にはすべて同じ値である。

アドレス 1 から 15 が可変エントリとなる。可変エントリにマスクフラグを書き込み可能な演算器は ALU と MAU である。出力フラグ値はそれぞれの演算器の命令式の箇所で解説する。書き込んだマスクフラグは直後のステップから利用可能である。

書き込みマスクとゼロフラッシュマスクは同時に指定可能である。ただし、それぞれで異なるエントリを読み出すことはできない。

3.6.2.1 書き込みマスク適用

PE メモリ (LM0、LM0 ベースアドレスレジスタ、LM1、LM1 ベースアドレスレジスタ、GRF0、GRF1、T レジスタ、マスクレジスタ) の書き込みにはマスクを作用させられる。

マスク作用時にはマスクレジスタにあらかじめ書き込んでおいたマスクフラグを用いる。マスクフラグが 0 のとき、対応する PE メモリの箇所への書き込みは行われない。

マスク作用時には次を指定する。

- マスク作用語長
- PE メモリ (GRF0、GRF1、T レジスタ、LM0 および LM0 のベースアドレスレジスタ、LM1 および LM1 のベースアドレスレジスタ) のどれにマスクを作用させるか (複数指定可能)
- どのエントリを読み出すかのアドレス

文法 (複数行適用)

```
mask[l|ll][r][s][t][m][n][k] <addr>
```

その行以降のすべての書き込みマスク設定を指定する。それ以降の PE 命令式の動作を変更する制御文であって、これ自体は PE 命令式ではない。

`[l|ll]`はマスク作用語長指定である。`l`が長語、`ll`が 2 長語となる。省略すると長語になる。

`[r][s][t][m][n][k]`は順に GRF0、GRF1、T レジスタ、LM0 および LM0 のベースアドレスレジスタ、LM1 および LM1 のベースアドレスレジスタ、マスクレジスタへの書き込みマスクを有効化する。指定がないメモリ要素に対するマスクはオフになる。`[r][s][t][m][n][k]`は実際には順不同である。

`<addr>`はマスクレジスタのエントリのアドレスを 0 から 31 で指定する。

アセンブリ先頭での暗黙の指定は `mask 0`、すなわち書き込みマスクを一切適用しない設定となっている。

文法 (単一行適用)

```
<dst>/([ll]<mask-pattern>|$[ll]imr<adr>)
```

[*3] この定義から、固定値エントリのアドレス 0 とアドレス 31 は同じ内容となる。

そのステップのみで複数行マスク指定をキャンセルし、<dst>への書き込みにマスクを適用する。
<dst>はいずれかの書き込み先 PE メモリオペランドである。
[ll]<mask-pattern>が固定値エントリ、$[ll]imr<adr>が可変エントリの指定である。
llはマスク作用語長指定を 2 長語にする。
<mask-pattern>は固定値エントリにおける各サイクルのフラグ値を、第 1 サイクルから順に 0か 1の 4 回の繰り返しで指定する。
<adr>は可変エントリのアドレスを 1 から 15 の整数で指定する。

エラー

- 同一ステップ内において、マスク作用語長やエントリのアドレスが一致しない複数の単一行マスク適用が指定された場合、エラーになる
- 書き込み先 PE メモリオペランドの語長とマスク作用語長について、片方が 2 長語でもう片方が 2 長語でない場合、エラーになる

例 (複数行適用)

```
maskr 0b10001
lpassa $lm0v $lr0v
lpassa $lm8v $lr8v
mask 0
```

LM0 から GRF に、2 ステップそれぞれにおいて第 4 サイクルでのみコピーを行う。すなわち、$lr6,$lr14のみが更新される。その後、マスクが一切適用されないデフォルトの状態に戻す。

例 (固定値エントリの単一行適用)

```
mask 0
lpassa $lm0v $lr0v/0001
lpassa $lm8v $lr8v/1000
```

LM0 から GRF0 に、最初のステップでは第 4 サイクル、次のステップでは第 1 サイクルでのみコピーを行う。すなわち、$lr6,$lr8のみが更新される。その後、マスクが一切適用されないデフォルトの状態に戻る。

例 (可変エントリの単一行適用)

```
hmmul $lx $lm0v $llr0v/$llimr1
```

1 番の可変エントリからフラグを読み出し、半精度行列ベクトル積演算の結果をマスクを適用しつつ GRF0 に書き込む。ここで半精度 MAU 演算の基本動作の結果は 2 長語になるため、マスク作用語長も 2 長語としている。

3.6.2.2 ゼロフラッシュマスク適用

PE メモリに書き込み可能なデータ (MAU の出力、行列レジスタの転置読み出し、ALU の出力、L1BM から PE への転送) はマスクフラグを用いて一部の値をゼロにできる。

マスク作用時にはマスクレジスタにあらかじめ書き込んでおいたマスクフラグを用いる。マスクフラグが 0 のとき、対応する箇所は書き込み前にゼロになる。

マスク作用時には次を指定する。

- マスク作用語長
- MAU の出力、行列レジスタの転置読み出し、ALU の出力、L1BM から PE への転送のどれにマスクを作用させるか (複数指定不可)
- どのエントリを読み出すかのアドレス

文法

```
<opcode>/([ll]<mask-pattern>|$[ll]imr<adr>)
```

<opcode>は MAU 演算、行列レジスタ転置読み出し、ALU 演算、L1BM から PE への転送のいずれかのオペコードである。

/以降の文法は書き込みマスクの単一行適用と同じである。

書き込みマスクと異なり、複数行適用の文法は存在しない。

エラー

- 同一ステップ内において、マスク作用語長やエントリのアドレスが一致しない複数の単一行マスク適用が指定された場合、エラーになる
- 複数のオペコードにゼロフラッシュマスク適用が指定された場合、エラーになる

例

```
hmmul/ll0111 $lx $lm0v $llr0v
```

半精度行列ベクトル積演算の結果について、第 1 サイクルをすべて 0 としたうえで GRF0 に書き込む。ここで半精度 MAU 演算の基本動作の結果は 2 長語になるため、マスク作用語長も 2 長語としている。

3.6.3 ハザードの回避

命令間で適切にステップ数を空けなければならない場合がある。その原因には、書き込みが完了するまで読み出しを待たなければならないことに起因するデータ競合と、アドレス信号線などのハードウェア資源の解放を待たなければならないことに起因するポート衝突の 2 種類がある。データ競合の場合は同じアドレスの読み書きでなければ起きないが、ポート衝突の場合はアドレス値に依存しない。以下では命令の組み合わせごとに必要なステップ数を述べる。

以下で述べる条件はすべてアセンブラによってチェックされ、違反していればエラーとなる。

3.6.3.1 L1BM → L2BM 転送 ⇒ L2BM → L1BM 転送

L1BM → L2BM 転送命令から L2BM → L1BM 転送命令まではポート衝突のため 3 ステップ空ける必要がある。

例

```
l2bm@0 $lb0 $lc0
nop/3
l2bmb $lc64 $lb64
```

この例で nop/3を nop/2とするとエラーになる。

3.6.3.2 L2BM → L1BM 転送 ⇒ L1BM → L2BM 転送 / 内部マルチキャスト

L2BM → L1BM 転送命令から、L1BM → L2BM 転送命令または内部マルチキャスト命令まではポート衝突のため 2 ステップ空ける必要がある。これは後者で読み出しが行われる L1B 番号が、前者で書き込みが行われる L1B 番号に含まれている場合に限る。

例 1

```
l2bmb $lc0 $lb0
nop/2
l2bm@0 $lb64 $lc64
```

この例で nop/2を nopとするとエラーになる。

例 2

```
l2bmb@0 $lc0 $lb0
l2bm@1 $lb0 $lc64
```

この例では最初の命令では 0 番 L1B にのみ書き込みがあり、次の命令では 1 番 L1B からのみ読み出すので、ステップを空ける必要はない。

3.6.3.3 内部マルチキャスト ⇒ L1BM → L2BM 転送 / 内部マルチキャスト

内部マルチキャスト命令から、L1BM → L2BM 転送命令または内部マルチキャスト命令まではポート衝突のため 3 ステップ空ける必要がある。これは後者で読み出しが行われる L1B 番号が、前者で書き込みが行われる L1B 番号に含まれている場合に限る。

例 1

```
l2bmi@0/0 $lb0 $lb0
nop/3
l2bm@1 $lb64 $lc64
```

この例で nop/3を nop/2とするとエラーになる。

例 2

```
l2bmi@0/0 $lb0 $lb0
l2bmi@0/0 $lb64 $lb64
```

この例では書き込みが起きる L1B (0 番以外すべて) と読み出しが起きる L1B (0 番のみ) に重なりがないので連続して発行できる。

3.6.3.4 内部マルチキャスト ⇒ L1BM → PE 転送

内部マルチキャスト命令から、書き込んだのと同じアドレスを読み出す L1BM → PE 転送命令まではデータ競合のため 10 サイクル空ける必要がある。レイテンシを最小化する必要がなければ、3 ステップ空ければ必ずハザードを回避できる。

例

```
l2bmi@0/0 $lb64 $lb64
nop
l1bmm $lb52 $lr0v
```

この例で l1bmm命令の$lb52を$lb56に変更するとエラーになる。先頭を第 0 サイクルとして、$lb64に書き込む命令が発行されるのは第 0 サイクル、読み出す命令が発行されるのは (l1bmm命令はサイクルあたり 4 長語を読むので) 第 10 サイクルとなり、間に 9 サイクルしかないためである。

3.6.3.5 L2BM → L1BM 転送 ⇒ L1BM → PE 転送

L2BM → L1BM 転送命令から、書き込んだのと同じアドレスを読み出す L1BM → PE 転送命令まではデータ競合のため 6 サイクル空ける必要がある。レイテンシを最小化する必要がなければ、2 ステップ空ければ必ずハザードを回避できる。

例

```
l2bmb $lc0 $lb64
l1bmm $lb52 $lr0v
```

この例で l1bmm命令の$lb52を$lb56に変更するとエラーになる。先頭を第 0 サイクルとして、$lb64に書き込む命令が発行されるのは第 0 サイクル、読み出す命令が発行されるのは (l1bmm命令はサイクルあたり 4 長語を読むので) 第 6 サイクルとなり、間に 5 サイクルしかないためである。

3.6.3.6 PE → L1BM 転送 ⇒ L1BM → L2BM 転送 / 内部マルチキャスト

PE → L1BM 転送命令から、書き込んだのと同じアドレスを読み出す L1BM → L2BM 転送命令または内部マルチキャスト命令までは、データ競合のため 10 サイクル空ける必要がある。レイテンシを最小化する必要がなければ、3 ステップ空ければ必ずハザードを回避できる。

例

```
l1bmr4dfadd $lr0v $lb48
nop/2
l2bmrdfadd $lb64 $lc0
```

この例で l1bmr4dfadd命令の$lb48を$lb32に変更するとエラーになる。先頭を第 0 サイクルとして、$lb64に書き込む命令が発行されるのは (1 長語 4x4 縮約命令はサイクルあたり 16 長語を書くので) 第 2 サイクル、読み出す命令が発行されるのは第 12 サイクルとなり、間に 9 サイクルしかないためである。

3.6.3.7 PE → L1BM 転送 ⇒ L1BM → PE 転送

PE → L1BM 転送命令から、L1BM → PE 転送命令まではポート衝突のため 2 ステップ空ける必要がある。

例 1

```
l1bmrdfadd $lr0v $lb0
nop/2
l1bmm $lb16 $ls0v
```

この例で nop/2を nopとするとエラーになる。

3.6.3.8 PE メモリ書き込み ⇒ PE メモリ読み出し

LM0/LM1 に書き込んだ後は、ポート衝突により 2 ステップ空けてからでないと読み出しを開始できない。

GRF0/GRF1 に書き込んだ後は、データ競合により 1 ステップ空けてからでないと同じアドレスからは読み出しを開始できない。異なるアドレスであれば同時に書き込みと読み出しを独立に発行できる。

T レジスタに書き込んだ後は、データ競合により 1 ステップ空けてからでないと読み出しを開始できない。

LM、GRF、T レジスタのいずれでも、読み出しと書き込みを同じステップに行うことは問題ない。読み出しではすでに書き込まれたデータが読み出される。

正確には、PE メモリへの書き込みを伴う命令は完了まで 6 サイクルを要する。アセンブラのハザード検出機構を使って以下が確かめられる。次のアセンブリ列は正常にアセンブルできる。

```
imm f"1.0" $r0/1000
nop
dvadd $lm0v $r0e $ln0v
```

/1000は書き込みマスク指定である。書き込みマスク指定の仕様は第 3.6.2.1 節を参照のこと。マスク指定により immのサイクルでは最初のサイクルでのみ$r0への書き込みが起こっており、それ以降次ステップの nopと合わせて dvaddの開始までに 7 サイクル空いているためである。同様に、マスク指定を/0100 (第 2 サイクルでのみ書き込み) としても 6 サイクルの空きがあるため問題ない。しかし、これを/0010 (第 3 サイクルでのみ書き込み) または/0001 (第 4 サイクルでのみ書き込み) とすると、サイクルの空きが 6 未満になってしまうため、アセンブラはこれを検出してエラーとする。

3.6.4 並列実行条件

複数の PE 命令式は、ある条件を満たしていればセミコロン (;) で区切って一行に並べることで 1 ステップ内で同時に発行できる。

条件はすべてアセンブラによってチェックされ、違反していればエラーとなる。

本節ではこの条件を詳しく述べる。まず、PE 命令式を表 3.6 に示すグループに分ける。

条件はステップ内の PE 命令式について、以下がすべて満たされていることである。ここで PE オペランドとは、GRF0、GRF1、LM0、LM1、T レジスタ、マスクレジスタ、各種フォワーディングのいずれかである。

表 3.6　並列実行条件を記述するための、PE 命令式のグループ分け

Group Name	PE Instruction Opcodes
nop	`nop`
noforward	`noforward`
l2bm	`l2bmdarw`以外の任意の L2BM 命令式
l2bmdarw	`l2bmdarw`
l1bm	折り返しでない任意の L1BM 命令式
l1bm-turnaround	任意の折り返し L1BM 命令式
mau-calc	`mfma`, `mmul`, `vfma`, `vmul`, `vadd`, `vpassa`
mau-mwrite	`mwrite`
mau-mread	`mread`
alu	任意の ALU 命令式
wait	`wait`

- グループにつき最大 1 つの命令式しか発行されていない[*4]
- `nop`が発行されている場合、`wait`を除く任意の他の命令式が発行されていない
- mau-calc、mau-mwrite、mau-mread の 3 グループうち発行されているのは最大 2 つであり、2 つならばそれらの命令式はさらに次を満たす
  - 精度指定が同一である
  - `vfma`か `vmul`のいずれかと `mwrite`が発行されている場合、`vfma`や `vmul`の第 2 入力と `mwrite`の入力で、読み出し元 PE オペランド、入力の符号反転 (第 3.6.9.10 節)、精度拡張 (第 3.6.9.11 節)、丸め (第 3.6.9.12 節) の指定が同一である
- 同じ行列レジスタオペランド (`$x`または`$y`) が最大 1 回しか出現しない
- 複数の命令式が同一の PE オペランドに書き込んでいない
- 複数の命令式が同一の PE オペランドから読み出している場合、それら全てでアクセスする領域が全サイクルで同一である
  - これにはマスク適用 (第 3.6.2.1 節、第 3.6.2.2 節) におけるマスクフラグの読み出しも含む
- LM0 からの読み出しと LM0 への書き込みが同時に発行されている場合、両方でアクセスする領域が全サイクルで同一である。LM1 についても同様
- 即値命令が発行されているならば LM0 がアクセスされていない
- ゼロフラッシュマスク適用 (第 3.6.2.2 節) は最大 1 回しか行われていない

本節で述べた条件以外にも、ハザード (第 3.6.3 節) などにより命令式を追加できない可能性がある。

セミコロン区切りで命令を並べる順序はアセンブル後の機械語命令には影響しない。ただし、エラー発生時のメッセージは順序によって変わる可能性がある。

例

極端な例としては、次は有効なアセンブリ列になる。(2 行目は改行して表示されているが実際は `noforwar`

[*4] 実際には `l2bmdars`は特殊な条件で他の l2bm グループの命令と同時発行できるが、条件が複雑で実用性も低いと考えられるので同グループとして扱うこととした。

d;以降はすべて 1 行に書かれている)

```
l2bmdars $lc0@.0 $dar0; l2bmdarw; l1bmrdfadd $lr0v $lbi
noforward; l2bmb $lc256 $lb0; l2bmdarw; l1bmm $lbi $lr0v/$imr1; l1bmrdfadd $lr8v $lb256;
      gmmul $lx $lm0v $ln0v/$imr1; gmwrite $ls0v $ly0; hrelu/$imr1 $t $t $t; wait i01
```

3.6.5 nop - NOP

4 サイクルの間なにもしない。
データハザード回避のため挿入する必要があることがある。
NOP はチップ内において命令発行ユニットによっても自動的に挿入される場合がある。
これと同時発行できる PE 命令は wait命令のみである。
nopは noforward (第 3.6.6 節) を含意する。

文法

```
nop
```

シンタックスシュガーとして、nop/<n>で n回 nopが挿入される。

効果

4 サイクルの間なにもしない。

エラー

wait命令以外と同時発行しようとするとエラーとなる。

例

```
lpassa $lm0v $ln0v
nop/2
lpassa $ln0v $lr0v
```

LM0 から LM1 に 4 長語コピーし、LM1 からの読み出しが可能になるまで待ってからさらに GRF0 に 4 長語コピーする。

3.6.6 noforward - フォワーディングと折り返しレジスタの更新をしない

noforward命令式が書かれたステップでは、フォワーディングレジスタ ($mauf、$aluf、$mreadf、$lbf) と L1BM 折り返しレジスタ ($lbi) の更新をしない。
noforward命令式は nopを除く他の任意の PE 命令式と同時発行できる。

文法

```
noforward
```

3.6.7 L2BM 命令式

本節では L2BM 命令式を解説する。これらは基本的に L2BM-L1BM 間の転送を行う。L1BM 間内部マルチキャストのみ、L1BM 同士の間の転送である。

3.6.7.1 L1B 部分集合指定

いくつかの L2BM 命令式で、転送の対象となる L1B を部分的に選択できる。この部分集合を以下では L1B 集合と呼ぶ。

L1B 集合指定の文法は以下である。

```
<l1bset> ::= <l1badr>/<immode>
           | <l1badr>
           | [<l1blist>]
```

<l1badr>および immodeは 0 から 7 の整数、<l1blist>はカンマで区切られた 0 から 7 の L1B 番号のリストである。

1 番目の記法において、<l1badr>を b_0、<immode>を i、定数 0b111 を m と置くと、b 番の L1B が $b\&(m \oplus i) == b_0\&(m \oplus i)$ を満たすならばそれは L1B 集合に含まれる。ここで & は論理積、$\oplus$ は排他的論理和である。

2 番目の<l1badr>による指定は、1 番目の記法で<immode>を 0 とした場合の略記であり、L1B 集合は<l1badr>番 L1B のみになる。

<l1blist>は直接 L1B 集合を指定できる。ただし、<l1blist>として許されるのは 1 番目の記法で記述できるものに限られる。具体的には表 3.7 を参照せよ。

<immode>が同じならば<l1badr>が異なっても同じ L1B 集合を表すことがあるが、<immode>が異なれば必ず異なる L1B 集合になることに注意する。

表 3.7　可能な L1B 集合と、それを実現する l1badr と immode の組み合わせの例

[<l1blist>]	<l1badr>/<immode>
[0]	0/0
[1]	1/0
[2]	2/0
[3]	3/0
[4]	4/0
[5]	5/0
[6]	6/0
[7]	7/0
[0,1]	0/1
[0,2]	0/2
[0,4]	0/4
[1,3]	1/2
[1,5]	1/4
[2,3]	2/1
[2,6]	2/4
[3,7]	3/4
[4,5]	4/1
[4,6]	4/2
[5,7]	5/2
[6,7]	6/1
[0,1,2,3]	0/3
[0,1,4,5]	0/5
[0,2,4,6]	0/6
[1,3,5,7]	1/6
[2,3,6,7]	2/5
[4,5,6,7]	4/3
[0,1,2,3,4,5,6,7]	0/7

3.6.7.2 l2bmb - L2BM → L1BM 放送

L2BM から、配下のすべてまたは一部の L1BM に 16 長語/サイクルで連続領域のコピーをする。

文法

```
l2bmb[@<l1bset>] $lc<addr_c> $lb<addr_b>
```

@以降は第 3.6.7.1 節で定めた L1B 集合の指定であり、これに含まれる L1B のみで書き込みが行われる。省略時は全 L1B に書き込みが行われる。

読み出しアドレス<addr_c>と書き込みアドレス<addr_b>はいずれも 16 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b
    uint_t src_addr = addr_c + cycle * 16
    uint_t dst_addr = addr_b + cycle * 16
    LongWord data[16] = MEM[group][l2b].l2bm[src_addr:src_addr+16]
    for l1b in l1bset
      MEM[group][l2b][l1b].l1bm[dst_addr:dst_addr+16] = data[0:16]
```

例

```
l2bmb@[0,1,2,3] $lc0 $lb0
```

L2BM から計 64 長語を 0,1,2,3 番 L1BM のみに放送する。

3.6.7.3 l2bmb2 - L2BM → L1BM 分配放送

L2BM から 64 長語/サイクルで読み出し、配下の L1B を先頭から 2 つずつ 4 グループに分けてグループ内では放送、グループ間では分配を行い、各 L1BM に 16 長語/サイクルで書き込む。

文法

```
l2bmb2[@<l1bset>] $lc<addr_c> $lb<addr_b>
```

@以降は第 3.6.7.1 節で定めた L1B 集合の指定であり、これに含まれる L1B のみで書き込みが行われる。省略時は全 L1B に書き込みが行われる。

読み出しアドレス<addr_c>は 64 長語アライン、書き込みアドレス<addr_b>は 16 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b
    uint_t src_addr = addr_c + cycle * 64
    uint_t dst_addr = addr_b + cycle * 16
    LongWord data[64] = MEM[group][l2b].l2bm[src_addr:src_addr+64]
    for l1b in l1bset
      uint_t data_addr = 16 * (l1b / 2)
      MEM[group][l2b][l1b].l1bm[dst_addr:dst_addr+16] = data[data_addr:data_addr+16]
```

例

```
l2bmb2 $lc0 $lb0
```

L2BM から計 256 長語を読み出し、L1B についてグループ内では放送、グループ間では分配を行い、各 L1BM に 64 長語を書き込む。

3.6.7.4 l2bmd - L2BM → L1BM 分配

L2BM から 64 長語/サイクルで連続領域を読み、配下のすべての L1BM に 8 長語/サイクルずつ分配をして連続領域に書き込む。

文法

```
l2bmd[@<l1bset>] $lc<addr_c> $lb<addr_b>
```

@以降は第 3.6.7.1 節で定めた L1B 集合の指定であり、これに含まれる L1B のみで書き込みが行われる。省略時は全 L1B に書き込みが行われる。

読み出しアドレス<addr_c>は 64 長語アライン、書き込みアドレス<addr_b>は 8 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b
    uint_t src_addr = addr_c + cycle * 64
    uint_t dst_addr = addr_b + cycle * 8
    LongWord data[64] = MEM[group][l2b].l2bm[src_addr:src_addr+64]
    for l1b in l1bset
      uint_t data_addr = 8 * l1b
      MEM[group][l2b][l1b].l1bm[dst_addr:dst_addr+8] = data[data_addr:data_addr+8]
```

```
l2bmd@[0,1,2,3] $lc0 $lb0
```

L2BM から計 256 長語を読み出し、0,1,2,3 番 L1BM に異なる 32 長語ずつのデータを書き込む (4,5,6,7 番 L1BM に対応するデータは捨てられることになる)。

3.6.7.5 l2bm@<l1badr> - L1BM → L2BM 個別転送

指定した L1B 番号の L1BM から、L2BM に 16 長語/サイクルで連続領域のコピーをする。

文法

```
l2bm@<l1badr> $lb<addr_b> $lc<addr_c>
```

<l1badr>は 0 から 7 の L1B 番号である。
読み出しアドレス<addr_b>と書き込みアドレス<addr_c>はいずれも 16 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b
    uint_t src_addr = addr_b + cycle * 16
    uint_t dst_addr = addr_c + cycle * 16
    LongWord data[16] = MEM[group][l2b][l1badr].l1bm[src_addr:src_addr+16]
    MEM[group][l2b].l2bm[dst_addr:dst_addr+16] = data[0:16]
```

例

```
l2bm@1 $lb0 $lc0
```

1 番 L1BM から計 64 長語を L2BM にコピーする。

3.6.7.6 l2bmr<rrn_opcode> - L1BM → L2BM 縮約

L2BM について、配下のすべてまたは一部の L1BM から 16 長語/サイクルで連続領域を読み、L1B 方向に指定した演算で縮約し、L2BM に書き込む。

第 3.6.7.2 節で述べた L2BM → L1BM 放送と対になっている。

文法

```
l2bmr<rrn_opcode>[@<l1bset>] $lb<addr_b> $lc<addr_c>
```

<rrn_opcode>は第 3.5.5 節で定めた縮約演算指定である。

@以降は第 3.6.7.1 節で定めた L1B 集合の指定であり、これに含まれる L1B のみから読み出しが行われる。省略時は全 L1B から読み出しが行われる。L1B 集合に含まれない L1B からは縮約演算の単位元が送信される。

読み出しアドレス<addr_b>と書き込みアドレス<addr_c>はいずれも 16 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b
    uint_t src_addr = addr_b + cycle * 16
    uint_t dst_addr = addr_c + cycle * 16
    LongWord buf[16]
    buf[:] = get_unit_value(rrn_opcode)
    for l1b in l1bset
      buf[0:16] = rrn_opcode(buf[0:16], MEM[group][l2b][l1b].l1bm[src_addr:src_addr
          +16])
    MEM[group][l2b].l2bm[dst_addr:dst_addr+16] = buf[0:16]
```

注意：縮約を内部的に実際にこの手順で行っているわけではない。

例

```
l2bmrdfadd@[0,4] $lb0 $lc0
```

0,4 番 L1BM からそれぞれ 64 長語を読み出し、倍精度浮動小数点数と解釈して L1B 方向に加算し、結果の計 64 長語を L2BM に書き込む。

3.6.7.7 l2bmr2<rrn_opcode> - L1BM → L2BM 結合縮約

各 L2BM について、配下の L1BM から 16 長語/サイクルで連続領域を読み、L1B を先頭から 2 つずつ 4 グループに分けてグループ内では指定した演算で縮約、グループ間では結合を行い、L2BM に 64 長語/サイクルで書き込む。

第 3.6.7.3 節で述べた L2BM → L1BM 分配放送と対になっている。

文法

```
l2bmr2<rrn_opcode> $lb<addr_b> $lc<addr_c>
```

<rrn_opcode>は第 3.5.5 節で定めた縮約演算指定である。

結合縮約ではない通常の縮約 (第 3.6.7.6 節) とは異なり、読み出し元 L1B 集合を指定することはできない。

読み出しアドレス<addr_b>は 16 長語アライン、書き込みアドレス<addr_c>は 64 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b
    uint_t src_addr = addr_b + cycle * 16
    uint_t dst_addr = addr_c + cycle * 64
    LongWord buf[64]
    buf[:] = get_unit_value(rrn_opcode)
    forall l1b
      uint_t data_addr = 16 * (l1b / 2)
      buf[data_addr:data_addr+16] = rrn_opcode(buf[data_addr:data_addr+16], MEM[group][
          l2b][l1b].l1bm[src_addr:src_addr+16])
    MEM[group][l2b].l2bm[dst_addr:dst_addr+64] = buf[0:64]
```

注意：縮約を内部的に実際にこの手順で行っているわけではない。

例

```
l2bmr2dfadd $lb0 $lc0
```

L1BM からそれぞれ 64 長語を読み出し、倍精度浮動小数点数と解釈してグループ内で加算し、グループ間では結合して計 256 長語を L2BM に書き込む。

3.6.7.8 l2bmd - L1BM → L2BM 結合

各 L2BM について、配下のすべての L1BM から 8 長語/サイクルずつ連続領域を読み出して結合し、L2BM の連続領域に 64 長語/サイクルで書き込む。

第 3.6.7.4 節で述べた L2BM → L1BM 分配命令と対になっている。

文法

```
l2bmd $lb<addr_b> $lc<addr_c>
```

読み出しアドレス<addr_b>は 8 長語アライン、書き込みアドレス<addr_c>は 64 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b
    uint_t src_addr = addr_b + cycle * 8
    uint_t dst_addr = addr_c + cycle * 64
    LongWord data[64]
    forall l1b
      uint_t data_addr = 8 * l1b
      data[data_addr:data_addr+8] = MEM[group][l2b][l1b].l1bm[src_addr:src_addr+8]
    MEM[group][l2b].l2bm[dst_addr:dst_addr+64] = data[0:64]
```

例

```
l2bmd $lb0 $lc0
```

L1BM から各 32 長語を読み出し、8 長語単位で結合して L2BM に計 256 長語を書き込む。

3.6.7.9 l2bmi - L1BM 間内部マルチキャスト

L2BM-L1BM 間のパスを利用して、L2BM 内部の L1BM 同士の間で 16 長語/サイクルの転送を行う。読み出し元も書き込み先も複数にできるのでマルチキャストと呼ばれる。

文法

```
l2bmb@<l1bset> $lb<addr0> $lb<addr1>
```

@以降は第 3.6.7.1 節で定めた L1B 集合の指定であり、これに含まれる L1B のみから読み出しが行われる。L1B 集合指定における<immode>を i、L1B 番号を b として、読み出し元と $b\&i$ が一致する L1B に書き込みが行われる。ここで & は論理積である。例えば<immode>が 0 ならば、$b\&i$ は常に 0 なので、<l1badr>番の L1B から読み出された値が他のすべての L1B に送られる。また、<l1badr>が 0、<immode>が 6 (=0b110) であれば、2 進 L1B 番号の上 2 桁でグループ化され、0,2,4,6 番の L1B から読み出された値が、それぞれ 1,3,5,7 番の L1B に送られる。<immode>は 7 にできず、L1B 集合を直接指定する形式で全 L1B からなる集合を指定することもできない。

読み出しアドレス<addr0>と書き込みアドレス<addr1>はいずれも 16 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b
    uint_t src_addr = addr0 + cycle * 16
    uint_t dst_addr = addr1 + cycle * 16
    LongWord data[16]
    for l1b_src in l1bset
      data[0:16] = MEM[group][l2b][l1b_src].l1bm[src_addr:src_addr+16]
      for l1b_dst = 0:8
        if l1b_dst & immode == l1b_src & immode
          MEM[group][l2b][l1b_dst].l1bm[dst_addr:dst_addr+16] = data[0:16]
```

例

```
l2bmi@0/4 $lb0 $lb0
```

0 番 L1BM から読み出した計 64 長語を 1,2,3 番 L1BM の同じアドレスに放送し、4 番 L1BM と 5,6,7 番 L1BM について同じことをする。

3.6.7.10 l2bmdars/l2bmdarw - DAR への書き込み

DAR (DRAM アドレスレジスタ) は、DRAM 間接参照機能のための DRAM アドレスを保持するためのメモリである。DAR へは、L2BM から DARBUF (DAR 書き込みバッファ) を経由してアドレス値のコピーを行う[*5]。

DRAM 間接参照用アドレス語は 32 ビット幅である。すなわち、L2BM から 1 長語読み出すと 2 アドレス語になる。

l2bmdars命令は L2BM から DARBUF へのコピーをサイクルあたり 128 アドレス語の速度で行う。l2bmdarw命令は DARBUF から DAR へのコピーをサイクルあたり 1 アドレス語の速度で行う。これら 2 段階の命令で L2BM から DAR にアドレス値のコピーを行う。

DARBUF はグループにつきひとつ存在し、サイズは 512 アドレス語である。

DAR はグループにつきひとつ存在し、サイズは 1024 アドレス語である。

l2bmdars命令は、サイクルあたり 64 長語を L2BM から読み出し、128 アドレス語として DARBUF に書き込む。この際、長語につき MSB 側 32 ビットが小さい方 (偶数アドレス)、LSB 側 32 ビットが大きい方 (奇数アドレス) の DARBUF アドレスに対応するようにする。L2BM からはサイクルあたり 64 長語で読み出すので、1 ステップでちょうど DARBUF 全体に書き込むことになる。DARBUF に対する書き込み開始アドレス指定は存在せず、l2bmdars命令を発行したステップの最初のサイクルは先頭から 128 アドレス語を書き込み、以降 128 アドレス語ずつインクリメントする。

また、l2bmdars命令の最初のサイクルでは、DARBUF 読み出しアドレスのゼロリセットと、DAR 書き込み開始アドレスの指定も行う。l2bmdars命令を発行したのと同じステップから、l2bmdarw命令による DARBUF から DAR への転送を開始可能である。l2bmdarw命令が発行されている間、DARBUF 読み出しアドレスと DAR 書き込みアドレスは毎サイクル 1 ずつインクリメントされる。いずれもアドレスが終端に到達したらラップアラウンドする。

l2bmdarwは他の任意の L2BM 命令式と同時発行できる。

l2bmdarsの文法

```
l2bmdars $lc<addr_c>@.<l2b_c> $dar<addr_dar>
```

l2bmdarsの効果

```
MEM.darw_addr = addr_adr
MEM.darbuf_read_addr = 0
for cycle = 0:4
  forall group
    uint_t src_addr = addr_c + cycle * 64
    LongWord data[64]
    data[0:64] = MEM[group][l2b_c].l2bm[src_addr:src_addr+64]
    for i = 0:64
      uint_t dst_addr = cycle * 128 + i * 2
      MEM[group].darbuf[dst_addr] = (data[i] >> 32)
```

[*5] これはアドレス値読み出しのために L2BM を占有する時間と、L2BM から DRAM へのアドレス値転送の帯域幅をともに節約するためである。

```
    MEM[group].darbuf[dst_addr+1] = (data[i] & ((1 << 32) - 1))
```

l2bmdarwの文法

```
l2bmdarw
```

l2bmdarwの効果

```
for cycle = 0:4
  forall group
    MEM[group].dar[MEM.dar_write_addr] = MEM[group].darbuf[MEM.darbuf_read_addr]
  MEM.dar_write_addr = (MEM.dar_write_addr + 1) % 1024
  MEM.darbuf_read_addr = (MEM.darbuf_read_addr + 1) % 512
```

例

```
l2bmdars $lc0@.1 $dar16; l2bmdarw
l2bmd $lc256 $lb0; l2bmdarw
l2bmd $lc512 $lb32; l2bmdarw
l2bmd $lc768 $lb64; l2bmdarw
```

1 番 L2BM から DARBUF にアドレス値をコピーし、同時に DARBUF から DAR 16 番地以降への書き込みを開始する。以降 L2BM から L1BM への分配と並行して DAR への書き込みを続ける。

3.6.8 L1BM 命令式

本節では L1BM 命令式を解説する。
L1BM 命令式の語長の区別も含めたリストを表 3.8 に示す。

表 3.8　L1BM 命令式のリスト。L1BM 側速度と PE 側速度の単位は長語/サイクルである

名称	転送方向	L1BM 側速度	PE 側速度	サンプル
1 長語 PE 放送 (3.6.8.6)	L1BM → PE	1	1	`l1bmp $lb0 $lr0v`
2 長語 PE 放送 (3.6.8.7)	L1BM → PE	2	2	`l1bmp $llb0 $llr0v`
1 長語 16x1MAB 放送 (3.6.8.8)	L1BM → PE	4	1	`l1bmm $lb0 $lr0v`
2 長語 16x1MAB 放送 (3.6.8.9)	L1BM → PE	8	2	`l1bmm $llb0 $llr0v`
1 長語 16x1 個別転送 (3.6.8.10)	PE → L1BM	4	1	`l1bmm@0 $lr0v $lb0`
2 長語 16x1 個別転送 (3.6.8.11)	PE → L1BM	8	2	`l1bmm@0 $llr0v $llb0`
1 長語 16x1 縮約 (3.6.8.12)	PE → L1BM	4	1	`l1bmrdfadd $lr0v $lb0`
2 長語 16x1 縮約 (3.6.8.13)	PE → L1BM	8	2	`l1bmrffadd $llr0v $llb0`
1 長語 4x4MAB 放送 (3.6.8.14)	L1BM → PE	16	1	`l1bmm $lb0 $lr0v`
2 長語 4x4MAB 放送 (3.6.8.15)	L1BM → PE	32	2	`l1bmm4 $llb0 $llr0v`
1 長語 4x4 個別転送 (3.6.8.16)	PE → L1BM	16	1	`l1bmm4@0 $lr0v $lb0`
2 長語 4x4 個別転送 (3.6.8.17)	PE → L1BM	32	2	`l1bmm4@0 $llr0v $llb0`
1 長語 4x4 縮約 (3.6.8.18)	PE → L1BM	16	1	`l1bmr4dfadd $lr0v $lb0`
2 長語 4x4 縮約 (3.6.8.19)	PE → L1BM	32	2	`l1bmr4ffadd $llr0v $llb0`
分配 (3.6.8.20)	L1BM → PE	64	1	`l1bmd $lb0 $lr0v`
結合 (3.6.8.21)	PE → L1BM	64	1	`l1bmd $lr0v $lb0`

3.6.8.1　4x4 モードについて

表 3.8 に現れる 4x4 と付いた転送モードでは、2 進 4 桁の MAB 番号で見て、下位 2 ビットに対して放送・縮約を行い、上位 2 ビットに対しては分配・結合を行う。

L1BM 側のアクセス速度は対応する 16x1 モード命令の 4 倍になる。L1BM アドレス上ではこの 4 倍の分は最も外側に来る。例えば、2 長語 4x4MAB 放送で 1 サイクルで L1BM から読み出される 32 長語は、C 言語の多次元配列風に書くと [4 (MAB 番号上位 2 ビット)][2 (長語)][4 (PE)] という並びになる。

3.6.8.2　L1BM 命令式種別と折り返し動作

L1BM 側アクセス速度と PE 側アクセス速度がともに同じである転送命令は同一の種別となる。同一種別の転送ではレイアウトが保たれる。すなわち、L1BM から PE に転送し、何らかの要素ごとの演算を行って L1BM に戻すという操作をしてもレイアウトが崩れない。

また、同一種別中の PE → L1BM 転送と L1BM → PE 転送の間では**折り返し動作**が可能である。折り返し動作とは次のようなものである。

```
l1bmm@2 $lr0v $lb0
l1bmm $lbi $lm0v; l1bmm@2 $lr8v $lb16
l1bmm $lbi $lm8v
```

これは以下と等価である。

```
l1bmm@2 $lr0v $lb0
```

```
l1bmm@2 $lr8v $lb16
nop
nop
l1bmm $lb0 $lm0v
l1bmm $lb16 $lm8v
```

つまり、折り返しレジスタ`$lbi` (第 3.6.1.5 節) からの読み出しとすることで、直前のステップで L1BM に書き込まれたデータをすぐに読み出し、かつ続く PE → L1BM 方向転送と重ねて発行することができる。

また、PE → L1BM 方向転送の書き込み先を`$lbi`とすることで、折り返しレジスタのみを更新し、L1BM への書き込みは行わないようにできる。

PE → L1BM 方向の転送も折り返し転送も発行していない間は、折り返しレジスタは更新されない。

以降の疑似コードによる効果の記述では簡単のため折り返しレジスタは考慮しない。

3.6.8.3 PE 側オペランドの語長

表 3.8 に示した「PE 側速度」が 1 長語/サイクルの命令でも、PE 側に 2 長語のオペランドを指定できる。この際 L1BM → PE 転送であれば MSB 側 1 長語に有効な値が入り、LSB 側は all 0 になる。PE → L1BM 転送であれば LSB 側 1 長語は切り捨てられる。

「PE 側速度」が 2 長語/サイクルの L1BM → PE 転送で、PE 側オペランドに 2 長語より小さい語長のものを指定するとエラーになる。

以降の疑似コードによる効果の記述では、「PE 側速度」と PE 側オペランド語長は一致しているものとする。

3.6.8.4 入力の精度拡張

表 3.8 に示した命令のうち、「PE 側速度」が 2 長語である縮約命令を用い、かつ縮約演算指定が単精度の浮動小数点数演算であるとする。このとき、PE 側オペランドの末尾に `e`を付加することで、PE 側オペランドから読み出した値を半精度浮動小数点数と解釈して、単精度浮動小数点数への精度拡張を行える。

精度拡張の際のサイクル内でのデータの並びは次の通りである。p 番 PE から送られた 2 長語の MSB 側 1 長語について、さらに MSB 側から j 番目の半精度語に $4 \times p + j$ (16 進表記) の番号を与えると、L1BM 上での 8 長語は、それらを単精度語にした上で MSB 側から `014589cd2367abef`の順に並べたものになる。この並びは、これによって L1BM に書かれたデータに対し 2 長語放送を行って PE で半精度への丸めを行うと、元の PE 上での並びと一致するように決められている。

例

```
l1bmrffadd $lr0ve $llb0
```

GRF0 からサイクル・PE あたり 4 半精度語を読み出して単精度へ変換後に加算で縮約し、サイクルあたり 16 単精度語 (=8 長語) を L1BM に書き込む。

3.6.8.5 縮約結果の丸め

表 3.8 に示した命令のうち、縮約命令を用い、かつ縮約演算指定が単精度の浮動小数点数演算であるとする。このとき、オペコードの末尾に `r`を付加することで、縮約演算回路の結果を L1BM や折り返しレジスタに

書き込む前に、単精度浮動小数点数から半精度浮動小数点数への丸めを行える。

丸めの際のサイクル内でのデータの並びは次の通りである。縮約演算回路が出力した 16 短語を MSB 側から `0123456789abcdef`とする。すなわち、例えば `0`から `3`までが PE0 に由来する値である。このとき、L1BM 上での 4 長語は、それらを半語に丸めた上で MSB 側から `018923ab45cd67ef`の順に並べたものになる。

これは第 3.6.8.4 節で述べた精度拡張の際の並べ替えの逆変換になっている[*6]。

例

```
l1bmrffaddr $llr0v $lb0
```

GRF0 からサイクル・PE あたり 4 単精度語を読み出して加算で縮約し、サイクルあたり 16 半精度語に丸めて L1BM に書き込む。

[*6] 半精度縮約は実際にはこれを用いて、精度拡張 → 単精度縮約 → 丸めという実装になっている。すなわち、`l1bmrhfadd $lr0v $lb0`は `l1bmrffaddr $lr0ve $lb0`のエイリアスである

3.6.8.6 l1bmp - 1 長語 PE 放送

L1BM からサイクルあたり 1 長語を読み出し、配下の 64 個の PE 全てに放送する。

対称な PE → L1BM 方向の転送命令は存在しない。

文法

```
l1bmp $lb<addr_b> <dst_0> [<dst_1>..]
```

<dst_0> [<dst_1>..]は書き込み先 PE オペランドである。

演算結果は複数の PE メモリに同時に書き込むことができるが、以下効果では簡単のため単一の書き込み先として dstを指定した例を示している。これは以降のすべての L1BM 命令式の記述について同様である。

L1BM 側のアラインメント制約は存在しない。

効果

```
for cycle = 0:4
  forall group,l2b,l1b
    uint_t src_addr = addr_b + cycle
    LongWord data = MEM[group][l2b][l1b].l1bm[src_addr]
    forall mab,pe
      MEM[group][l2b][l1b][mab][pe].refer_pemem(dst, cycle) = data
```

3.6.8.7 l1bmp - 2 長語 PE 放送

L1BM からサイクルあたり 2 長語を読み出し、配下の 64 個の PE 全てに放送する。

対称な PE → L1BM 方向の転送命令は存在しない。

文法

```
l1bmp $l1b<addr_b> <dst_0> [<dst_1>..]
```

<dst_0> [<dst_1>..]は書き込み先 PE オペランドである。

L1BM 側のアラインメント制約は存在しないが、アドレスの下位 6 ビットの値は 0 から 56 のいずれかでなければならない。

効果

```
for cycle = 0:4
  forall group,l2b,l1b
    uint_t src_addr = addr_b + cycle
    LongWord data[2]
    data[0] = MEM[group][l2b][l1b].l1bm[src_addr]
    data[1] = MEM[group][l2b][l1b].l1bm[src_addr+4]
    forall mab,pe
      MEM[group][l2b][l1b][mab][pe].refer_pemem(dst, cycle) = data[0:2]
```

サイクル内の L1BM の読み出し領域が不連続なので注意する。

3.6.8.8 l1bmm - 1 長語 16x1MAB 放送

L1BM からサイクルあたり 4 長語を読み出し、配下の 16 個の MAB 全てに放送し、4 長語を 4PE に分配して 1 長語を書き込む。

文法

```
l1bmm $lb<addr_b> <dst_0> [<dst_1>..]
```

<dst_0> [<dst_1>..]は書き込み先 PE オペランドである。
L1BM アドレス<addr_b>は 4 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b,l1b
    uint_t src_addr = addr_b + cycle * 4
    LongWord data[4] = MEM[group][l2b][l1b].l1bm[src_addr:src_addr+4]
    forall mab,pe
      MEM[group][l2b][l1b][mab][pe].refer_pemem(dst, cycle) = data[pe]
```

3.6.8.9 l1bmm - 2 長語 16x1MAB 放送

L1BM からサイクルあたり 8 長語を読み出し、配下の 16 個の MAB 全てに放送し、8 長語を 4PE に分配して 2 長語を書き込む。

文法

```
l1bmm $l1b<addr_b> <dst_0> [<dst_1>..]
```

<dst_0> [<dst_1>..]は書き込み先 PE オペランドである。
L1BM アドレス<addr_b>は 8 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b,l1b
    uint_t src_addr = addr_b + cycle * 8
    LongWord data[4][2]
    data[0:4][0] = MEM[group][l2b][l1b].l1bm[src_addr:src_addr+4]
    data[0:4][1] = MEM[group][l2b][l1b].l1bm[src_addr+4:src_addr+8]
    forall mab,pe
      MEM[group][l2b][l1b][mab][pe].refer_pemem(dst, cycle) = data[pe][0:2]
```

3.6.8.10 l1bmm@<mabadr> - 1 長語 16x1 個別転送

各 L1BM について、指定した番号の MAB 配下の 4PE からサイクルあたり 1 長語ずつを読み出し、結合して L1BM にサイクルあたり 4 長語で書き込む。

文法

```
l1bmm@<mabadr> <src> $l1b<addr_b>
```

<mabadr>は 0 から 15 の MAB 番号、<src>は読み出し元 PE オペランドである。
L1BM アドレス<addr_b>は 4 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b,l1b
    LongWord data[4]
    forall pe
      data[pe] = MEM[group][l2b][l1b][mabadr][pe].refer_pemem(src, cycle)
    uint_t dst_addr = addr_b + cycle * 4
    MEM[group][l2b][l1b].l1bm[dst_addr:dst_addr+4] = data[0:4]
```

3.6.8.11 l1bmm@<mabadr> - 2 長語 16x1 個別転送

各 L1BM について、指定した番号の MAB 配下の 4PE からサイクルあたり 2 長語ずつを読み出し、結合して L1BM にサイクルあたり 8 長語で書き込む。

文法

```
l1bmm@<mabadr> <src> $l1b<addr_b>
```

<mabadr>は 0 から 15 の MAB 番号、<src>は読み出し元 PE オペランドである。
L1BM アドレス<addr_b>は 8 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b,l1b
    LongWord data[4][2]
    forall pe
      data[pe][0:2] = MEM[group][l2b][l1b][mabadr][pe].refer_pemem(src, cycle)
    uint_t dst_addr = addr_b + cycle * 8
    MEM[group][l2b][l1b].l1bm[dst_addr:dst_addr+4] = data[0:4][0]
    MEM[group][l2b][l1b].l1bm[dst_addr+4:dst_addr+8] = data[0:4][1]
```

3.6.8.12 l1bmr<rrn_opcode> - 1 長語 16x1 縮約

各 L1BM について、各 MAB 配下の 4PE からサイクルあたり 1 長語ずつを読み出し、MAB 方向には縮約、PE 方向には結合して L1BM にサイクルあたり 4 長語で書き込む。

文法

```
l1bmr<rrn_opcode> <src> $lb<addr_b>
```

<rrn_opcode>は第 3.5.5 節で定めた縮約演算指定である。
<src>は読み出し元 PE オペランドである。
L1BM アドレス<addr_b>は 4 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b,l1b
    LongWord data[4]
    data[:] = get_unit_value(rrn_opcode)
    forall mab,pe
      data[pe] = rrn_opcode(data[pe], MEM[group][l2b][l1b][mab][pe].refer_pemem(src,
          cycle))
    uint_t dst_addr = addr_b + cycle * 4
    MEM[group][l2b][l1b].l1bm[dst_addr:dst_addr+4] = data[0:4]
```

注意：縮約を内部的に実際にこの手順で行っているわけではない。

3.6.8.13 l1bmr<rrn_opcode> - 2 長語 16x1 縮約

各 L1BM について、各 MAB 配下の 4PE からサイクルあたり 2 長語ずつを読み出し、MAB 方向には縮約、PE 方向には結合して L1BM にサイクルあたり 8 長語で書き込む。

2 長語動作は単精度の浮動小数点数演算または任意精度の borについてのみ可能である。

この命令は第 3.6.8.4 節で述べた、半精度から単精度への入力値変換つきの動作を指定可能である。

文法

```
l1bmr<rrn_opcode> <src> $l1b<addr_b>
```

<rrn_opcode>は第 3.5.5 節で定めた縮約演算指定である。

<src>は読み出し元 PE オペランドである。

L1BM アドレス<addr_b>は 8 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b,l1b
    LongWord data[4][2]
    data[:][:] = get_unit_value(rrn_opcode)
    forall mab,pe
      data[pe][0:2] = rrn_opcode(data[pe][0:2], MEM[group][l2b][l1b][mab][pe].
          refer_pemem(src, cycle))
    uint_t dst_addr = addr_b + cycle * 8
    MEM[group][l2b][l1b].l1bm[dst_addr:dst_addr+4] = data[0:4][0]
    MEM[group][l2b][l1b].l1bm[dst_addr+4:dst_addr+8] = data[0:4][1]
```

注意：縮約を内部的に実際にこの手順で行っているわけではない。

3.6.8.14 l1bmm4 - 1 長語 4x4MAB 放送

L1BM からサイクルあたり 16 長語を読み出し、第 3.6.8.1 節で述べた対応で MAB に対して分配と放送を行い、分配後の 4 長語をさらに 4PE に分配して 1 長語を書き込む。

文法

```
l1bmm4 $lb<addr_b> <dst_0> [<dst_1>..]
```

<dst_0> [<dst_1>..]は書き込み先 PE オペランドである。
L1BM アドレス<addr_b>は 16 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b,l1b
    uint_t src_addr = addr_b + cycle * 16
    LongWord data[16] = MEM[group][l2b][l1b].l1bm[src_addr:src_addr+16]
    forall mab,pe
      uint_t idx = (mab >> 2) * 4 + pe
      MEM[group][l2b][l1b][mab][pe].refer_pemem(dst, cycle) = data[idx]
```

3.6.8.15 l1bmm4 - 2 長語 4x4MAB 放送

L1BM からサイクルあたり 32 長語を読み出し、第 3.6.8.1 節で述べた対応で MAB に対して分配と放送を行い、分配後の 8 長語をさらに 4PE に分配して 2 長語を書き込む。

文法

```
l1bmm4 $l1b<addr_b> <dst_0> [<dst_1>..]
```

<dst_0> [<dst_1>..]は書き込み先 PE オペランドである。
L1BM アドレス<addr_b>は 32 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b,l1b
    uint_t src_addr = addr_b + cycle * 32
    LongWord data[16][2]
    for i = 0:4
      data[i*4:(i+1)*4][0] = MEM[group][l2b][l1b].l1bm[src_addr+i*8:src_addr+i*8+4]
      data[i*4:(i+1)*4][1] = MEM[group][l2b][l1b].l1bm[src_addr+i*8+4:src_addr+i*8+8]
    forall mab,pe
      uint_t idx = (mab >> 2) * 4 + pe
      MEM[group][l2b][l1b][mab][pe].refer_pemem(dst, cycle) = data[idx][0:2]
```

3.6.8.16 l1bmm4@<mabadr> - 1 長語 4x4 個別転送

各 L1BM について、$0+i, 4+i, 8+i, 12+i$ 番 MAB (i は $0, 1, 2, 3$ のいずれか指定した値) の配下の 4PE からサイクルあたり 1 長語ずつを読み出し、結合して L1BM にサイクルあたり 16 長語で書き込む。

文法

```
l1bmm4@<mabadr> <src> $lb<addr_b>
```

<mabadr>は 0 から 3 の MAB 番号、<src>は読み出し元 PE オペランドである。
L1BM アドレス<addr_b>は 16 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b,l1b
    LongWord data[16]
    for mab_outer = 0:4
      forall pe
        data[mab_outer * 4 + pe] = MEM[group][l2b][l1b][mab_outer * 4 + mabadr][pe].
            refer_pemem(src, cycle)
    uint_t dst_addr = addr_b + cycle * 16
    MEM[group][l2b][l1b].l1bm[dst_addr:dst_addr+16] = data[0:16]
```

3.6.8.17 l1bmm4@<mabadr> - 2 長語 4x4 個別転送

各 L1BM について、$0+i, 4+i, 8+i, 12+i$ 番 MAB (i は $0, 1, 2, 3$ のいずれか指定した値) の配下の 4PE からサイクルあたり 2 長語ずつを読み出し、結合して L1BM にサイクルあたり 32 長語で書き込む。

文法

```
l1bmm4@<mabadr> <src> $l1b<addr_b>
```

<mabadr>は 0 から 3 の MAB 番号、<src>は読み出し元 PE オペランドである。
L1BM アドレス<addr_b>は 32 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b,l1b
    LongWord data[16][2]
    for mab_outer = 0:4
      forall pe
        data[mab_outer * 4 + pe][0:2] = MEM[group][l2b][l1b][mab_outer * 4 + mabadr][
            pe].refer_pemem(src, cycle)
    uint_t dst_addr = addr_b + cycle * 32
    for i = 0:4
      MEM[group][l2b][l1b].l1bm[dst_addr+i*8:dst_addr+i*8+4] = data[i*4:(i+1)*4][0]
      MEM[group][l2b][l1b].l1bm[dst_addr+i*8+4:dst_addr+i*8+8] = data[i*4:(i+1)*4][1]
```

3.6.8.18 l1bmr4<rrn_opcode> - 1 長語 4x4 縮約

各 L1BM について、各 MAB 配下の 4PE からサイクルあたり 1 長語ずつを読み出し、PE 方向には結合し、第 3.6.8.1 節で述べた対応で MAB に対して縮約と結合を行って L1BM にサイクルあたり 16 長語で書き込む。

文法

```
l1bmr4<rrn_opcode> <src> $lb<addr_b>
```

<rrn_opcode>は第 3.5.5 節で定めた縮約演算指定である。
<src>は読み出し元 PE オペランドである。
L1BM アドレス<addr_b>は 16 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b,l1b
    LongWord data[16]
    data[:] = get_unit_value(rrn_opcode)
    forall mab,pe
      uint_t idx = (mab >> 2) * 4 + pe
      data[idx] = rrn_opcode(data[idx], MEM[group][l2b][l1b][mab][pe].refer_pemem(src,
          cycle))
    uint_t dst_addr = addr_b + cycle * 16
    MEM[group][l2b][l1b].l1bm[dst_addr:dst_addr+16] = data[0:16]
```

注意：縮約を内部的に実際にこの手順で行っているわけではない。

3.6.8.19 l1bmr4<rrn_opcode> - 2 長語 4x4 縮約

各 L1BM について、各 MAB 配下の 4PE からサイクルあたり 2 長語ずつを読み出し、PE 方向には結合し、第 3.6.8.1 節で述べた対応で MAB に対して縮約と結合を行って L1BM にサイクルあたり 32 長語で書き込む。

2 長語動作は単精度の浮動小数点数演算または任意精度の borについてのみ可能である。

この命令は第 3.6.8.4 節で述べた、半精度から単精度への入力値変換つきの動作を指定可能である。

文法

```
l1bmr<rrn_opcode> <src> $llb<addr_b>
```

<rrn_opcode>は第 3.5.5 節で定めた縮約演算指定である。

<src>は読み出し元 PE オペランドである。

L1BM アドレス<addr_b>は 32 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b,l1b
    LongWord data[16][2]
    data[:][:] = get_unit_value(rrn_opcode)
    forall mab,pe
      uint_t idx = (mab >> 2) * 4 + pe
      data[idx][0:2] = rrn_opcode(data[idx][0:2], MEM[group][l2b][l1b][mab][pe].
          refer_pemem(src, cycle))
    uint_t dst_addr = addr_b + cycle * 32
    for i = 0:4
      MEM[group][l2b][l1b].l1bm[dst_addr+i*8:dst_addr+i*8+4] = data[i*4:(i+1)*4][0]
      MEM[group][l2b][l1b].l1bm[dst_addr+i*8+4:dst_addr+i*8+8] = data[i*4:(i+1)*4][1]
```

注意：縮約を内部的に実際にこの手順で行っているわけではない。

3.6.8.20 l1bmd - 分配

L1BM からサイクルあたり 64 長語を読み出し、配下の 64 個の PE に分配し、サイクルあたり 1 長語で書き込む。

L1BM アドレス上では MAB 番号が外側、PE 番号が内側に対応する。つまり、C 言語の多次元配列風に書くと [16 (MAB 番号)][4 (PE 番号)] という並びになる。

この際、L1BM 上のデータは 4 長語ごとに異なる MAB に送られることになるが、行き先の MAB 番号をラウンドロビンにずらすことができる。

なお 2 長語動作の分配命令は存在しない。

文法

```
l1bmd[<mabdiff>] $lb<addr_b> <dst_0> [<dst_1>..]
```

<dst_0> [<dst_1>..]は書き込み先 PE オペランドである。

<mabdiff>は + か-の後に 0 から 15 の整数を付け、MAB 番号をずらす量を指定する。正の場合でも符号は必須である。<mabdiff>番ずれた MAB に、<mabdiff>未指定であれば送信されるはずだったデータが送られるという対応関係になる。

MAB 番号をずらす機能は折り返し転送でも有効である。この際、対になる結合命令 (第 3.6.8.21 節) と分配命令で MAB 番号をずらす量は一致していなければならない。この機能は結合命令で折り返しレジスタに書き込む際には適用されないので、折り返しの結果 2 倍ずれてしまうということにはならない。

L1BM アドレス<addr_b>は 64 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b,l1b
    uint_t src_addr = addr_b + cycle * 64
    LongWord data[64] = MEM[group][l2b][l1b].l1bm[src_addr:src_addr+64]
    forall mab,pe
      uint_t src_mab = (mab + mabdiff) % 16
      MEM[group][l2b][l1b][mab][pe].refer_pemem(dst, cycle) = data[src_mab * 4 + pe]
```

例

```
lpassa $mabid $lr0v
nop
l1bmd+1 $lr0v $lb0
l1bmd-1 $lr0v $lb256; l1bmd+1 $lbi $ls0v
l1bmd-1 $lbi $ls8v
nop
l1bmd $lb0 $ls16v
l1bmd $lb256 $ls24v
```

折り返し転送の解説のため、第 3.6.8.21 節で述べる結合命令も含めた例とした。0 番 MAB の$ls0，$ls8，$ls16，$ls24にはそれぞれ 15, 1, 15, 1 が入る。前者 2 つでは折り返し分配時に MAB 番号をずらす機能

が適用されており、後者 2 つでは`$lb0`, `$lb256`への結合書き込み時に同機能が適用されている。

3.6.8.21 l1bmd - 結合

PE からサイクルあたり 1 長語を読み出し、L1BM 配下の 64 個の PE について結合し L1BM にサイクルあたり 64 長語で書き込む。

L1BM アドレスと MAB 番号・PE 番号との対応関係は分配命令と同じである。

この際、4 長語ごとに異なる MAB から送られた値が書き込まれることになるが、送り元の MAB 番号をラウンドロビンにずらすことができる。この機能は折り返しレジスタへの書き込み時は動作しない。

なお 2 長語動作の結合命令は存在しない。

文法

```
l1bmd[<mabdiff>] <src> $lb<addr_b>
```

<src>は読み出し元 PE オペランドである。

<mabdiff>は + か-の後に 0 から 15 の整数を付け、MAB 番号をずらす量を指定する。正の場合でも符号は必須である。<mabdiff>番分ずれたアドレスに、<mabdiff>未指定であれば送信されるはずだったデータが送られるという対応関係になる。

折り返し転送の際の注意は分配命令の項目 (第 3.6.8.20 節) を参照のこと。

L1BM アドレス<addr_b>は 64 長語アラインである必要がある。

効果

```
for cycle = 0:4
  forall group,l2b,l1b
    LongWord data[64]
    LongWord data_turnaround[64]
    forall mab,pe
      uint_t dst_mab = (mab + mabdiff) % 16
      data[dst_mab * 4 + pe] = MEM[group][l2b][l1b][mab][pe].refer_pemem(src, cycle)
      data_turnaround[mab * 4 + pe] = MEM[group][l2b][l1b][mab][pe].refer_pemem(src,
          cycle)
    uint_t dst_addr = addr_b + cycle * 64
    MEM[group][l2b][l1b].l1bm[dst_addr:dst_addr+64] = data[0:64]
    MEM[group][l2b][l1b].l1bm_turnaround[cycle][0:64] = data_turnaround[0:64]
```

l1bm_turnaroundは折り返しレジスタを表している。折り返しレジスタへの書き込みが起きるのは他の PE → L1BM 転送命令も同様だが、結合命令のみ MAB 番号をずらす機能に関して特殊な動作をするため、特別に記載した。

例

```
l1bmd+1 $lr0v $lb0
l1bmd $lr0v $lb256
nop/2
l1bmd $lb0 $ls0v
l1bmd+1 $lb256 $ls8v
```

この例では$ls0から 4 長語の範囲に書き込まれる値と、$ls8から 4 長語の範囲に書き込まれる値は等しくなる。つまり、ある<mabdiff>の値を結合時に指定するのと分配時に指定するのは等価になるような定義になっている。

3.6.9 MAU 命令式

3.6.9.1 基本動作について

MAU の演算モードは行列ベクトル積和演算モードとベクトル積和演算モードがあり、精度モードは倍精度、単精度、疑似単精度、半精度がある。

半精度モードでは単精度アキュミュレーションが行われる。すなわち、積和演算の和の方の入力と出力はいずれも単精度にできる。

行列ベクトル積和演算モードのオペコードは以下である。

- `mfma` - 3 入力で、第 1 入力の行列と第 2 入力のベクトルの行列ベクトル積を行い、第 3 入力のベクトルとの要素ごとの和を取る
- `mmul` - 2 入力で、`mfma`の第 3 入力を 0 としたもの

ベクトル積和演算モードのオペコードは以下のとおりである。

- `vfma` - 3 入力で、第 1 入力と第 2 入力の要素ごとの積を取り、第 3 入力と要素ごとの和を取る
- `vmul` - 2 入力で、`vfma`の第 3 入力を 0 としたもの
- `vadd` - 2 入力で、`vfma`の第 2 入力を 1 としたもの
- `vpassa` - 1 入力で、`vfma`の第 3 入力を 0、第 2 入力を 1 としたもの

まず、入力符号反転・精度拡張・丸めが起きない場合の `mfma`および `vfma`を**基本動作**として、各演算モード・精度モードでの基本動作を説明する。

3.6.9.2 dmfma - 倍精度行列ベクトル積和演算の基本動作

あらかじめ行列レジスタに書かれた行列データを Aとして 4 次元の倍精度行列ベクトル積 FMA (Ax+y) を行う。行列レジスタから読み出した 4 行 4 列のブロックフロート倍精度行列データのうち、後述する指定に応じて 0,1 行目あるいは 2,3 行目を抽出した 2 行 4 列の行列が使用される。

文法

```
dmfma(u|d) $l(x|y) <src_x> <src_y> <dst_0> [<dst_1>..]
```

(u|d) は 4 行 4 列のうち上半分・下半分どちらの 2 行 4 列を演算に使用するかを指定するために用いられる。uを指定した場合は Aの 0,1 行目を xと掛け合わせたものを Axの 0,1 要素目とし、Axの 2,3 要素目を 0 として yとの加算を行う (以下効果において offset=0)。dを指定した場合は Aの 2,3 行目を xと掛け合わせたものを Axの 2,3 要素目とし、Axの 0,1 要素目を 0 として yとの加算を行う (以下効果において offset=2)。

第 1 入力の$l(x|y) は読み出し元の行列レジスタであり、以下効果において sideとして参照する。

第 2 入力の<src_x>および第 3 入力の<src_y>は読み出し元 PE オペランドである。<src_x>はブロックフロート倍精度の値でアクセス語長は長語である。基本動作において<src_y>は通常の倍精度の値で、アクセス語長は長語である。

<dst_0> [<dst_1>..]は書き込み先 PE オペランドである。演算結果は複数の PE メモリに同時に書き込むことができるが、以下効果では簡単のため単一の書き込み先として dstを指定した例を示している。基本動作において演算結果は通常の倍精度であり、書き込みのアクセス語長は長語である。

効果

```
for cycle = 0:4
  forall chip,l2b,l1b,mab
    LongWord src_data_A[4][4] = MEM[chip][l2b][l1b][mab].refer_matreg(side, LongWord)
    LongWord src_data_x[4] = MEM[chip][l2b][l1b][mab][0:4].refer_pemem(src_x, cycle)
    LongWord src_data_y[4] = MEM[chip][l2b][l1b][mab][0:4].refer_pemem(src_y, cycle)
    LongWord dst_data[4] = {0, 0, 0, 0}
    for i = 0:2
      for j = 0:4
        dst_data[i + offset] += src_data_A[i + offset][j] * src_data_x[j]
    for i = 0:4
      dst_data[i] += src_data_y[i]
    MEM[chip][l2b][l1b][mab][0:4].refer_pemem(dst, cycle) = dst_data[0:4]
```

3.6.9.3 fmfma - 単精度行列ベクトル積和演算の基本動作

あらかじめ行列レジスタに書かれた行列データを Aとして 4 次元の単精度行列ベクトル積 FMA (Ax+y) を行う。行列レジスタから読み出した 8 行 8 列のブロックフロート単精度行列データのうち、0,2,4,6 列目を抽出した 8 行 4 列の行列が使用される。

文法

```
fmfma $l(x|y) <src_x> <src_y> <dst_0> [<dst_1>..]
```

第 1 入力の$l(x|y) は読み出し元の行列レジスタであり、以下効果において sideとして参照する。

第 2 入力の<src_x>および第 3 入力の<src_y>は読み出し元 PE オペランドである。<src_x>はブロックフロート単精度の値でアクセス語長は短語である。基本動作において<src_y>は通常の単精度の値で、アクセス語長は長語である。

<dst_0> [<dst_1>..]は書き込み先 PE オペランドである。演算結果は複数の PE メモリに同時に書き込むことができるが、以下効果では簡単のため単一の書き込み先として dstを指定した例を示している。基本動作において演算結果は通常の単精度であり、書き込みのアクセス語長は長語である。

効果

```
for cycle = 0:4
  forall chip,l2b,l1b,mab
    ShortWord src_data_A[8][8] = MEM[chip][l2b][l1b][mab].refer_matreg(side, ShortWord)
    ShortWord src_data_x[4] = MEM[chip][l2b][l1b][mab][0:4].refer_pemem(src_x, cycle)
    ShortWord src_data_y[4][2] = MEM[chip][l2b][l1b][mab][0:4].refer_pemem(src_y, cycle
        )
    ShortWord dst_data[8] = {0, 0, 0, 0, 0, 0, 0, 0}
    for i = 0:8
      for j = 0:4
        dst_data[i] += src_data_A[i][j*2] * src_data_x[j]
      dst_data[i] += src_data_y[i/2][i%2]
    forall pe
      MEM[chip][l2b][l1b][mab][pe].refer_pemem(dst, cycle) = dst_data[pe*2:(pe+1)*2]
```

3.6.9.4 gmfma - 疑似単精度行列ベクトル積和演算の基本動作

あらかじめ行列レジスタに書かれた行列データを Aとして 8 次元の疑似単精度行列ベクトル積 FMA (Ax+y) を行う。

文法

```
gmfma $l(x|y) <src_x> <src_y> <dst_0> [<dst_1>..]
```

第 1 入力の$l(x|y) は読み出し元の行列レジスタであり、以下効果において sideとして参照する。

第 2 入力の<src_x>および第 3 入力の<src_y>は読み出し元 PE オペランドである。<src_x>はブロックフロート疑似単精度の値でアクセス語長は長語である。基本動作において<src_y>は通常の単精度の値で、アクセス語長は長語である。

<dst_0> [<dst_1>..]は書き込み先 PE オペランドである。演算結果は複数の PE メモリに同時に書き込むことができるが、以下効果では簡単のため単一の書き込み先として dstを指定した例を示している。基本動作において演算結果は通常の単精度であり、書き込みのアクセス語長は長語である。

効果

```
for cycle = 0:4
  forall chip,l2b,l1b,mab
    ShortWord src_data_A[8][8] = MEM[chip][l2b][l1b][mab].refer_matreg(side, ShortWord)
    ShortWord src_data_x[4][2] = MEM[chip][l2b][l1b][mab][0:4].refer_pemem(src_x, cycle
        )
    ShortWord src_data_y[4][2] = MEM[chip][l2b][l1b][mab][0:4].refer_pemem(src_y, cycle
        )
    ShortWord dst_data[8] = {0, 0, 0, 0, 0, 0, 0, 0}
    for i = 0:8
      for j = 0:8
        dst_data[i] += src_data_A[i][j] * src_data_x[j/2][j%2]
      dst_data[i] += src_data_y[i/2][i%2]
    forall pe
      MEM[chip][l2b][l1b][mab][pe].refer_pemem(dst, cycle) = dst_data[pe*2:(pe+1)*2]
```

3.6.9.5 hmfma - 半精度行列ベクトル積和演算の基本動作

あらかじめ行列レジスタに書かれた行列データを Aとして 16 次元の半精度行列ベクトル積 FMA (Ax+y) を行う。

文法

```
hmfma $l(x|y) <src_x> <src_y> <dst_0> [<dst_1>..]
```

第 1 入力の$l(x|y) は読み出し元の行列レジスタであり、以下効果において sideとして参照する。

第 2 入力の<src_x>および第 3 入力の<src_y>は読み出し元 PE オペランドである。<src_x>はブロックフロート半精度の値でアクセス語長は長語である。基本動作において<src_y>は通常の単精度の値で、アクセス語長は 2 長語である。

<dst_0> [<dst_1>..]は書き込み先 PE オペランドである。演算結果は複数の PE メモリに同時に書き込むことができるが、以下効果では簡単のため単一の書き込み先として dstを指定した例を示している。基本動作において演算結果は通常の単精度であり、書き込みのアクセス語長は 2 長語である。

効果

```
for cycle = 0:4
  forall chip,l2b,l1b,mab
    HalfWord src_data_A[16][16] = MEM[chip][l2b][l1b][mab].refer_matreg(side, HalfWord)
    HalfWord src_data_x[4][4] = MEM[chip][l2b][l1b][mab][0:4].refer_pemem(src_x, cycle)
    ShortWord src_data_y[4][4] = MEM[chip][l2b][l1b][mab][0:4].refer_pemem(src_y, cycle
        )
    ShortWord dst_data[16] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
    for i = 0:16
      for j = 0:16
        dst_data[i] += src_data_A[i][j] * src_data_x[j/4][j%4]
      dst_data[i] += src_data_y[i/4][i%4]
    forall pe
      MEM[chip][l2b][l1b][mab][pe].refer_pemem(dst, cycle) = dst_data[pe*4:(pe+1)*4]
```

3.6.9.6 dvfma - 倍精度ベクトル積和演算の基本動作

MAB 内で倍精度ベクトル FMA (x*y+z) を行う。

文法

```
dvfma(u|d) <src_x> <src_y> <src_z> <dst_0> [<dst_1>..]
```

(u|d) は 4 個の PE のうち PE0,PE1 または PE2,PE3 どちらの 2PE の要素について乗算を行うかを指定する。uを指定した場合は PE0,PE1 で xと yと掛け合わせたものを x*yの中間結果とし、PE2,PE3 では x*yの中間結果を便宜的に 0 として、zとの加算を行う (以下効果において offset=0)。dを指定した場合は PE2,PE3 で xと yと掛け合わせたものを x*yの中間結果とし、PE0,PE1 では x*yの中間結果を便宜的に 0 として、zとの加算を行う (以下効果において offset=2)。

第 1 入力の<src_x>、第 2 入力の<src_y>、および第 3 入力の<src_z>は読み出し元 PE オペランドである。基本動作においていずれも通常の倍精度の値で、アクセス語長は長語である。

<dst_0> [<dst_1>..]は書き込み先 PE オペランドである。演算結果は複数の PE メモリに同時に書き込むことができるが、以下効果では簡単のため単一の書き込み先として dstを指定した例を示している。基本動作において演算結果は通常の倍精度であり、書き込みのアクセス語長は長語である。

効果

```
for cycle = 0:4
  forall chip,l2b,l1b,mab
    LongWord dst_data[4] = {0, 0, 0, 0}
    for i = 0:2
      LongWord src_data_x = MEM[chip][l2b][l1b][mab][i + offset].refer_pemem(src_x,
          cycle)
      LongWord src_data_y = MEM[chip][l2b][l1b][mab][i + offset].refer_pemem(src_y,
          cycle)
      dst_data[i + offset] = src_data_x * src_data_y
    forall pe
      LongWord src_data_z = MEM[chip][l2b][l1b][mab][pe].refer_pemem(src_z, cycle)
      dst_data[pe] += src_data_z
      MEM[chip][l2b][l1b][mab][pe].refer_pemem(dst, cycle) = dst_data[pe]
```

3.6.9.7 dvadd - 倍精度ベクトル和の基本動作

MAB 内で倍精度ベクトル和 (x+y) を行う。

文法

```
dvadd <src_x> <src_y> <dst_0> [<dst_1>..]
```

第 1 入力の<src_x>および第 2 入力の<src_y>は読み出し元 PE オペランドである。基本動作においていずれも通常の倍精度の値で、アクセス語長は長語である。

<dst_0> [<dst_1>..]は書き込み先 PE オペランドである。演算結果は複数の PE メモリに同時に書き込むことができるが、以下効果では簡単のため単一の書き込み先として dstを指定した例を示している。基本動作において演算結果は通常の倍精度であり、書き込みのアクセス語長は長語である。

効果

```
for cycle = 0:4
  forall chip,l2b,l1b,mab,pe
    LongWord src_data_x = MEM[chip][l2b][l1b][mab][pe].refer_pemem(src_x, cycle)
    LongWord src_data_y = MEM[chip][l2b][l1b][mab][pe].refer_pemem(src_y, cycle)
    LongWord dst_data = src_data_x + src_data_y
    MEM[chip][l2b][l1b][mab][pe].refer_pemem(dst, cycle) = dst_data
```

3.6.9.8 fvfma - 単精度ベクトル積和演算の基本動作

MAB 内で単精度ベクトル FMA (x*y+z) を行う。

文法

```
fvfma <src_x> <src_y> <src_z> <dst_0> [<dst_1>..]
```

第 1 入力の<src_x>、第 2 入力の<src_y>、および第 3 入力の<src_z>は読み出し元 PE オペランドである。基本動作においていずれも通常の単精度の値で、アクセス語長は長語である。

<dst_0> [<dst_1>..]は書き込み先 PE オペランドである。演算結果は複数の PE メモリに同時に書き込むことができるが、以下効果では簡単のため単一の書き込み先として dstを指定した例を示している。基本動作において演算結果は通常の単精度であり、書き込みのアクセス語長は長語である。

効果

```
for cycle = 0:4
  forall chip,l2b,l1b,mab,pe
    ShortWord src_data_x[2] = MEM[chip][l2b][l1b][mab][pe].refer_pemem(src_x, cycle)
    ShortWord src_data_y[2] = MEM[chip][l2b][l1b][mab][pe].refer_pemem (src_y, cycle)
    ShortWord src_data_z[2] = MEM[chip][l2b][l1b][mab][pe].refer_pemem (src_z, cycle)
    ShortWord dst_data[2] = src_data_x[:] * src_data_y[:] + src_data_z[:]
    MEM[chip][l2b][l1b][mab][pe].refer_pemem (dst, cycle) = dst_data[:]
```

3.6.9.9 hvfma - 半精度ベクトル積和演算の基本動作

MAB 内で半精度ベクトル FMA (x*y+z) を行う。

文法

```
hvfma <src_x> <src_y> <src_z> <dst_0> [<dst_1>..]
```

第 1 入力の<src_x>、第 2 入力の<src_y>、および第 3 入力の<src_z>は読み出し元 PE オペランドである。基本動作において<src_x>および<src_y>は通常の半精度の値で、アクセス語長は長語である。基本動作において<src_z>は通常の単精度の値で、アクセス語長は 2 長語である。

<dst_0> [<dst_1>..]は書き込み先 PE オペランドである。演算結果は複数の PE メモリに同時に書き込むことができるが、以下効果では簡単のため単一の書き込み先として dstを指定した例を示している。基本動作において演算結果は通常の単精度であり、書き込みのアクセス語長は 2 長語である。

効果

```
for cycle = 0:4
  forall chip,l2b,l1b,mab,pe
    HalfWord src_data_x[4] = MEM[chip][l2b][l1b][mab][pe].refer_pemem(src_x, cycle)
    HalfWord src_data_y[4] = MEM[chip][l2b][l1b][mab][pe].refer_pemem(src_y, cycle)
    ShortWord src_data_z[4] = MEM[chip][l2b][l1b][mab][pe].refer_pemem(src_z, cycle)
    ShortWord dst_data[4] = src_data_x[:] * src_data_y[:] + src_data_z[:]
    MEM[chip][l2b][l1b][mab][pe].refer_pemem(dst, cycle) = dst_data[:]
```

3.6.9.10　入力符号反転

mfma,vfma,mwriteの入力オペランドのうち、行列以外のものについて、その入力値の各要素の符号を反転させることができる。符号反転する際はそのオペランドの先頭に-を付加する。mfmaの第 1 入力である行列の符号を反転したい場合は mwriteの際に符号を反転して書き込んでおけばよい。

例 1

```
dmfmau $lx $lr0v -$lm0v $ln0v
```

行列レジスタのデータを A、GRF のデータを x、LM0 のデータを yとして、yの符号を反転した倍精度行列ベクトル積 FMA (Ax-y) を行い、LM1 に書き込む。

例 2

```
dvadd -$lr0v -$lm0v $ln0v
```

GRF のデータを x、LM0 のデータを yとして、x,yの符号を反転した倍精度ベクトル FMA (-x-y) を行い、LM1 に書き込む。

3.6.9.11　入力の精度拡張

mfma,vfma,mwriteの入力オペランドのうち、基本動作において通常の単精度および倍精度の値を要求するものについて、一つ低い精度の値からの変換、すなわち、半精度から単精度、あるいは単精度から倍精度への変換を指示することができる。精度拡張する際はそのオペランドの末尾に eを付加する。

ブロックフロート単精度またはブロックフロート倍精度を要求するオペランドについては精度拡張を指示できないことに注意する。ブロックフロート化する前に精度拡張しておく必要がある。

基本動作から使われる語数は変わらず、一語あたりの語長が半分になる。よって、基本動作における入力語長が 2 長語なら長語で、長語なら短語で PE メモリから読み出すことになる。

例 1

```
gmfma $ly $lm0v $r0ve $ln0v
```

行列レジスタのブロックフロート疑似単精度データを A、LM0 のブロックフロート疑似単精度データを x、GRF の通常の半精度データを yとして、yを単精度に拡張した上で、疑似単精度行列ベクトル積 FMA (Ax+y) を行い、LM1 に書き込む。

例 2

```
hmfma $lx $lm0v $lr0ve $llr8v
```

行列レジスタのブロックフロート半精度データを A、LM0 のブロックフロート半精度データを x、GRF の通常の半精度データを yとして、yを単精度に拡張した上で、半精度行列ベクトル積 FMA (Ax+y) を行い、GRF に書き込む。

例 3

```
dvfmau $m0ve $r0ve $n0ve $lr4v
```

LM0 の単精度データを x、GRF の単精度データを y、LM1 の単精度データを zとして、x,y,zを倍精度に拡張した上で、倍精度ベクトル FMA (x*y+z) を行い、GRF に書き込む。

例 4

```
hvfma $lm0v $lr0v $ln0ve $llr8v
```

LM0 の通常の半精度データを x、GRF の通常の半精度データを y、LM1 の通常の半精度データを zとして、zを単精度に拡張した上で、半精度ベクトル FMA (x*y+z) を行い、GRF に書き込む。

3.6.9.12 入力の丸め

vfma,mwriteの入力オペランドのうち、基本動作において通常の半精度の値を要求するものについて、単精度の値からの変換を指示することができる。通常の半精度の値を要求するのは、半精度ベクトル積和演算および（行列転置を目的とした）半精度行列書き込みである。これらは長語の半精度の値を要求するので、2 長語の単精度の値を丸めて入力することになる。2 長語入力オペランドの末尾に rをつけることで丸めを指示する。

ブロックフロート半精度または（ブロックフロート如何に関わらず）単精度を要求するオペランドについては一つ高い精度から丸めての入力を指示できないことに注意する。単精度の値からブロックフロート半精度の値を得るには、後述する ALU 命令式の bfの入力オペランドで丸めを指示すればよい。倍精度の値から単精度の値を得るには、vpassaに後述する MAU 命令式の出力の丸めを付加して精度を変換したのち、必要に応じてブロックフロート化することになる。

次に例を示す。

```
hvmul $llr0vr $lm0v $llt
```

GRF の通常の単精度データを x、LM0 の通常の半精度データを yとして、xを半精度に丸めた上で、半精度ベクトル積 (x*y) を行い、T レジスタに書き込む。

3.6.9.13 出力の丸め

MAU 命令式オペコードの末尾に rを付加することで、出力値を丸めることができる。

丸める際、倍精度演算なら倍精度から単精度、半精度演算なら単精度から半精度への変換が行われる。基本動作から使われる語数は変わらず、一語あたりの語長が半分になる。よって、出力語長は倍精度演算なら短語に、半精度演算なら長語になる。

次は倍精度演算の出力を単精度に丸める例である。

```
dmfmaur $lx $lr0v $ln0v $m0v
```

次は半精度演算の出力を半精度に丸める例である。C ポート入力の精度拡張も併用している。

```
hvfmar $lm0v $ln0v $lr0ve $lr8v
```

3.6.9.14 MAU 命令の命令式の生成するマスクフラグ

MAU 命令式の生成するマスクフラグは演算結果の符号ビットを反転したものである。よって、マスク適用時には演算結果が非負だったところでは書き込みを行う、負だったところでは書き込みを行わないという動作になる。

3.6.10　行列レジスタ書き込み命令式

3.6.10.1　dmwrite - 倍精度行列レジスタ書き込み

倍精度データの行列レジスタへの書き込みを行う。

文法

```
dmwrite <src> $l(x|y)<addr>
```

<src>は読み出し元 PE オペランドであり、アクセス語長は長語である。行列ベクトル積和演算を目的として書き込む場合はブロックフロート倍精度の値であり、行列転置読み出しを目的として書き込む場合は通常の倍精度の値である。

$l(x|y)<addr>は書き込み先の行列レジスタオペランドである。(x|y) はどちらの行列レジスタに書き込むかの指定であり、以下効果において sideとして参照する。<addr>は書き込みを開始する行番号である。サイクルごとに連続する行を重複なくアクセスするようにインクリメントされる。倍精度行列データは 4 行 4 列であるので、1 命令で行列データ全体を行列レジスタに書き込むことができる。

効果

```
for cycle = 0:4
  forall chip,l2b,l1b,mab
    LongWord src_data[4] = MEM[chip][l2b][l1b][mab][0:4].refer_pemem(src, cycle)
    for j = 0:4
      MEM[chip][l2b][l1b][mab].refer_matreg(side, LongWord)[(addr+cycle)%4][j] =
          src_data[j]
```

3.6.10.2 fmwrite/gmwrite - 単精度・疑似単精度行列レジスタ書き込み

単精度データまたは疑似単精度データの行列レジスタへの書き込みを行う。これらは行列ベクトル積和演算の際は区別されるが、行列レジスタへの読み書きでは区別されない。

文法

```
(f|g)mwrite <src> $l(x|y)<addr>
```

fmwriteと gmwriteに機能的な違いはない。単精度行列積和命令や単精度ベクトル積和命令と同時に発行する場合は fmwriteを、疑似単精度行列ベクトル積和命令と同時に発行する場合は gmwriteを使う。

<src>は読み出し元 PE オペランドであり、アクセス語長は短語または長語である。短語アクセスの場合、以下効果で src_dataの 2 要素目は 0 埋めされる。行列ベクトル積和演算を目的として書き込む場合はブロックフロート単精度またはブロックフロート疑似単精度の値であり、行列転置読み出しを目的として書き込む場合は通常の単精度の値である。

$l(x|y)<addr>は書き込み先の行列レジスタオペランドである。(x|y) はどちらの行列レジスタに書き込むかの指定であり、以下効果において sideとして参照する。<addr>は書き込みを開始する行番号である。サイクルごとに連続する行を重複なくアクセスするようにインクリメントされる。単精度・疑似単精度行列データは 8 行 8 列であるので、2 命令で行列データ全体を行列レジスタに書き込むことができる。

効果

```
for cycle = 0:4
  forall chip,l2b,l1b,mab
    ShortWord src_data[4][2] = MEM[chip][l2b][l1b][mab][0:4].refer_pemem(src, cycle)
    for j = 0:8
      MEM[chip][l2b][l1b][mab].refer_matreg(side, ShortWord)[(addr+cycle)%8][j] =
          src_data[j/2][j%2]
```

3.6.10.3 hmwrite - 半精度行列レジスタ書き込み

半精度データの行列レジスタへの書き込みを行う。

文法

```
hmwrite <src> $(l|ll)(x|y)<addr>
```

<src>は読み出し元 PE オペランドであり、アクセス語長は長語または 2 長語である。アクセス語長は後述の行列レジスタのアクセス語長と一致している必要がある。行列ベクトル積和演算を目的として書き込む場合はブロックフロート半精度の値であり、行列転置読み出しを目的として書き込む場合は通常の半精度の値である。

$(l|ll)(x|y)<addr>は書き込み先の行列レジスタオペランドである。(l|ll) は行列レジスタへのアクセス語長を表す。lではサイクルあたり 1 行、llではサイクルあたり 2 行に書き込む。以下効果で wlとして参照する。(x|y) はどちらの行列レジスタに書き込むかの指定であり、以下効果において sideとして参照する。<addr>は書き込みを開始する行番号である。サイクルごとに連続する行を重複なくアクセスするようにインクリメントされる。wlが llの場合は<addr>は 2 の倍数である必要がある。半精度行列データは 16 行 16 列であるので、サイクルごとに 2 行ずつ書き込めば 2 命令で行列データ全体を行列レジスタに書き込むことができる。

効果

```
for cycle = 0:4
  forall chip,l2b,l1b,mab
    if wl == l
      HalfWord src_data[4][4] = MEM[chip][l2b][l1b][mab][0:4].refer_pemem(src, cycle)
      for j = 0:16
        MEM[chip][l2b][l1b][mab].refer_matreg(side, HalfWord)[(addr+cycle)%16][j] =
            src_data[j/4][j%4]
    elif wl == ll
      HalfWord src_data[4][8] = MEM[chip][l2b][l1b][mab][0:4].refer_pemem(src, cycle)
      for j = 0:16
        MEM[chip][l2b][l1b][mab].refer_matreg(side, HalfWord)[(addr+cycle*2)%16][j] =
            src_data[j/4][j%4]
        MEM[chip][l2b][l1b][mab].refer_matreg(side, HalfWord)[(addr+cycle*2)%16+1][j] =
             src_data[j/4][j%4+4]
```

3.6.11 行列レジスタ読み出し命令式

3.6.11.1 dmread - 倍精度行列レジスタ読み出し

倍精度データの行列レジスタからの読み出しを行う。

文法

```
dmread $l(x|y)<addr> <dst_0> [<dst_1>..]
```

$l(x|y)<addr>は読み出し元の行列レジスタオペランドである。(x|y) はどちらの行列レジスタから読み出すかの指定であり、以下効果において sideとして参照する。<addr>は読み出しを開始する列番号である。サイクルごとに連続する列を重複なくアクセスするようにインクリメントされる。倍精度行列データは 4 行 4 列であるので、1 命令で行列データ全体を行列レジスタから読み出すことができる。

<dst_0> [<dst_1>..]は書き込み先 PE オペランドである。複数の PE メモリに同時に書き込むことができるが、以下効果では簡単のため単一の書き込み先として dstを指定した例を示している。読み出した内容は通常の倍精度であり、書き込みのアクセス語長は長語である。

効果

```
for cycle = 0:4
  forall chip,l2b,l1b,mab
    LongWord src_data[4]
    for i = 0:4
      src_data[i] = MEM[chip][l2b][l1b][mab].refer_matreg(side, LongWord)[i][(addr+
          cycle)%4]
    forall pe
      MEM[chip][l2b][l1b][mab][pe].refer_pemem(dst, cycle) = src_data[pe]
```

3.6.11.2 fmread/gmread - 単精度行列レジスタ読み出し

単精度データの行列レジスタからの読み出しを行う。

文法

```
(f|g)mread $l(x|y)<addr> <dst_0> [<dst_1>..]
```

fmreadと gmreadに機能的な違いはない。単精度行列積和命令や単精度ベクトル積和命令と同時に発行する場合は fmreadを、疑似単精度行列ベクトル積和命令と同時に発行する場合は gmreadを使う。

$l(x|y)<addr>は読み出し元の行列レジスタオペランドである。(x|y) はどちらの行列レジスタから読み出すかの指定であり、以下効果において sideとして参照する。<addr>は読み出しを開始する列番号である。サイクルごとに連続する列を重複なくアクセスするようにインクリメントされる。単精度行列データは 8 行 8 列であるので、2 命令で行列データ全体を行列レジスタから読み出すことができる。

<dst_0> [<dst_1>..]は書き込み先 PE オペランドである。複数の PE メモリに同時に書き込むことができるが、以下効果では簡単のため単一の書き込み先として dstを指定した例を示している。読み出した内容は通常の単精度であり、書き込みのアクセス語長は長語である。

効果

```
for cycle = 0:4
  forall chip,l2b,l1b,mab
    ShortWord src_data[4][2]
    for i = 0:8
      src_data[i/2][i%2] = MEM[chip][l2b][l1b][mab].refer_matreg(side, ShortWord)[i][(
          addr+cycle)%8]
    forall pe
      MEM[chip][l2b][l1b][mab][pe].refer_pemem(dst, cycle) = src_data[pe][:]
```

3.6.11.3 hmread - 半精度行列レジスタ読み出し

半精度データの行列レジスタからの読み出しを行う。

文法

```
hmread $ll(x|y)<addr> <dst_0> [<dst_1>..]
```

$ll(x|y)<addr>は読み出し元の行列レジスタオペランドである。hmwriteと違い常にサイクルあたり 2 行が読み出される。(x|y) はどちらの行列レジスタから読み出すかの指定であり、以下効果において sideとして参照する。<addr>は読み出しを開始する列番号である。サイクルごとに連続する列を重複なくアクセスするようにインクリメントされる。<addr>は 2 の倍数でなければならない。半精度行列データは 16 行 16 列であるので、2 命令で行列データ全体を行列レジスタから読み出すことができる。

<dst_0> [<dst_1>..]は書き込み先 PE オペランドである。複数の PE メモリに同時に書き込むことができるが、以下効果では簡単のため単一の書き込み先として dstを指定した例を示している。読み出した内容は通常の半精度であり、書き込みのアクセス語長は 2 長語である。

効果

```
for cycle = 0:4
  forall chip,l2b,l1b,mab
    HalfWord src_data[4][8]
    for i = 0:16
      src_data[i/4][i%4] = MEM[chip][l2b][l1b][mab].refer_matreg(side, HalfWord)[i][(
          addr+cycle*2)%16]
      src_data[i/4][i%4+4] = MEM[chip][l2b][l1b][mab].refer_matreg(side, HalfWord)[i
          ][(addr+cycle*2)%16+1]
    forall pe
      MEM[chip][l2b][l1b][mab][pe].refer_pemem(dst, cycle) = src_data[pe][:]
```

3.6.12 ALU 命令式

ALU は PE につきひとつ存在し、ほとんどのオペコードでサイクルあたり 1 長語を処理する。

このとき、精度指定が単精度であれば長語内の 2 要素、半精度であれば長語内の 4 要素に対して並列に演算が行われる。よって、MAB あたりでは、半精度であればサイクルあたり 16 要素に並列に演算が行われることになる。

特にスループットが必要とされる一部のオペコードではサイクルあたり 2 長語を処理できる。

ALU 命令式の一般化した文法は次の通りである。

```
  [u][(d|f|g|h|l|i|s)]<opcode> <src_x> [<src_y>] <dst_0> [<dst_1>..]
| zero <dst_0> [<dst_1>..]
| imm[u] <payload> <dst_0> [<dst_1>..]
```

<opcode>はゼロ出力命令 zeroおよび即値命令 immを除くオペコードである。

<dst_0> [<dst_1>..]は書き込み先 PE オペランドである。これにはマスクレジスタ書き込み (第 3.6.1.13 節) を含めることができる。

ゼロ出力命令と即値命令は読み出し元 PE オペランドを取らない。即値命令は代わりに即値ペイロード (第 3.2.2 節) を入力オペランドとする。即値命令の [u]オプションについては第 3.6.12.2 節で述べる。

ゼロ出力命令や即値命令以外について述べる。これらは 1 入力または 2 入力である。<src_x>は第 1 入力、<src_y>は第 2 入力の読み出し元 PE オペランドである。<src_x>と<src_y>は同一でも構わない。<src_x>に限り、第 3.6.1.20 節で述べた固定値入力オペランドが指定可能である。先頭の [u]は**符号なし演算オプション**指定、[(d|f|g|h|l|i|s)]は精度指定である。d，f，g，hが浮動小数点数の倍精度・単精度・疑似単精度・半精度、l，i，sが整数の倍・単・半精度である。<opcode>によって符号なし演算オプションを受け付けるかどうか、およびどの精度指定を受け付けるかが異なる。符号なし演算オプションを受け付けるオペコードについて、オプションを指定する場合を**符号なしモード**、指定しない場合を**符号ありモード**と呼ぶことにする。

オペコードの一覧を表 3.9 に示す。

「演算種別」カラムは可能な精度指定を示している。演算種別ごとに実際に指定可能な精度を表 3.10 に示す。例えば mslの演算種別は untyped なので精度指定は付けられず、incの演算種別は int なので l，i，sのいずれかのみを付けられるということが分かる。入力のコピーを行う passaなどの出力は精度指定に影響されないが、後述の生成されるマスクフラグは影響されることに注意する。

「2 長語動作」カラムは 2 長語幅での動作をする場合があるかを示している。これが no であれば、ALU が出力する 2 長語の LSB 側 1 長語は、第 1 入力の LSB 側 1 長語がそのまま入る。第 1 入力の丸め (第 3.6.12.18 節) が有効であれば、この LSB 側 1 長語は 0 になる。yes の場合の動作はそれぞれの命令の項目で解説する。

「符号なしオプション」は uオプションを受け付けるかを示している。imm命令については uオプションの意味は符号なしとは異なるので no としてある。yes の場合の符号ありなしでの動作の違いはそれぞれの命令の項目で解説する。

表 3.9　ALU 命令オペコードの一覧。出力の列において第 1 入力オペランドの値を x、第 2 入力オペランドの値を y と表記する。

オペコード	演算種別	入力数	2 長語動作	符号なしオプション	出力
`zero`	untyped	0	yes	no	all 0
`imm`	untyped	1	yes	no	immediate value (Sec. 3.2.2)
`msl`	untyped	1	no	no	x of previous PE
`msr`	untyped	1	no	no	x of next PE
`passa`	both	1	yes	no	x
`inc`	int	1	no	yes	$x + 1$
`dec`	int	1	no	yes	$x - 1$
`not`	int	1	no	no	bitwise not x
`lnot`	int	1	no	no	1 if $x == 0$, 0 otherwise
`rsqrt`	float	1	no	no	approx. of $x^{-1/2}$
`floor`	float	1	no	no	floor(x)
`ftoi`	float	1	no	yes	cast x to integer
`bfe`	bfe	1	yes	no	extended half block float of x
`bfn`	bfn	1	yes (half only)	no	block float of x
`max`	both	2	no	yes (int only)	$\max(x, y)$
`min`	both	2	no	yes (int only)	$\min(x, y)$
`packbit`	both	2	no	no	$(x \ll 1) \mid$ (MSB of y)
`and`	int	2	no	no	bitwise x & y
`or`	int	2	no	no	bitwise $x \mid y$
`xor`	int	2	no	no	bitwise x xor y
`add`	int	2	no	yes	$x + y$
`sub`	int	2	no	yes	$x - y$
`lsl`	int	2	no	no	$x \ll y$
`lsr`	int	2	no	yes	$x \gg y$
`bsl`	int	2	no	no	circularly $x \ll y$
`bsr`	int	2	no	no	circularly $x \gg y$
`relu`	float	2	no	no	y if (MSB of x) $== 0$, -0 otherwise
`relu0`	float	2	no	no	same as `relu`
`relu1`	float	2	no	no	y if (2nd MSB of x) $== 0$, -0 otherwise
`relu2`	float	2	no	no	y if (3rd MSB of x) $== 0$, -0 otherwise
`relu3`	float	2	no	no	y if (4th MSB of x) $== 0$, -0 otherwise
`lrelud`	float	2	no	no	y if $x \geq 0$, $y/2$ otherwise
`lreluo`	float	2	no	no	y if $x \geq 0$, $y/8$ otherwise
`ilrelud`	float	2	no	no	y if $x \geq 0$, $2y$ otherwise

表 3.10　表 3.9 に現れる、ALU 命令オペコードの演算種別に対し、指定可能な精度の一覧。none の場合は精度指定を付けてはいけない。

演算種別	指定可能な精度
int	`l, i, s`
float	`d, f, h`
both	`d, f, h, l, i, s`
bfn	`d, f, g, h`
bfe	`h`
untyped	none

また、ALU 命令式が生成するマスクフラグ (第 3.6.2 節) を表 3.11 に示す。

「符号なし」カラムは符号なしオプションによる場合分けを示す。“-” であれば符号なしオプションが存在しないオペコードであることを示す。“false” であれば符号ありモードの場合であること、“true” であれば符号なしモードの場合であることを示す。“both” であれば、符号なしオプションは存在するが、マスクフラグの生成方法には影響しないことを示す。

「フラグが 1 になる条件」カラムは、出力長語中の各語について、それが満たされるときマスクフラグは 1 になり、そうでなければ 0 になることを示す。マスクフラグは必ずサイクルあたり 4 ビット出力され、精度指定が長語なら 1 語の結果が 4 ビットに、短語なら 2 語の結果がそれぞれ 2 ビットずつに複製されることに注意する。

`passa`命令、浮動小数点数モードの `max/min`命令は若干条件が複雑なため、それぞれ第 3.6.12.4 節と第 3.6.12.12 節で詳細を述べる。

表 3.11　オペコードと符号なし指定有無ごとの、ALU 命令式が生成するマスクフラグ

オペコード	符号なし	フラグが 1 になる条件
`zero/imm/msl/msr/floor/ftoi/bfe/bfn`	-	never
`passa`	-	output is all 0 (ignore LSB long-word)
`inc/dec/add/sub`	false	output is non-negative
`inc/dec/add/sub`	true	overflow does not occur
`not/lnot/and/or/xor`	-	output is all 0
`rsqrt/relu/relu0/lrelud/lreluo/ilrelud`	-	MSB of x is 0
`max/min (int)`	both	x is selected or $x == y$
`max/min (float)`	-	x is selected or $x == y$
`packbit`	-	MSB of y is 0
`lsl/bsl/bsr`	-	output is all 0
`lsr`	both	output is all 0
`relu1/relu2/relu3`	-	2nd/3rd/4th MSB of x is 0 resp.

以降では各オペランドの詳細を述べる。

3.6.12.1 zero - ゼロ出力命令

zero命令は入力を取らず、常に 2 長語の 0 を出力する。

例

```
zero $llr0v
```

$r0から$r15までの計 8 長語をゼロ埋めする。

3.6.12.2 imm - 即値命令

imm命令は、第 3.2.2 節で述べた短語の即値リテラルを、次の方法で並べた 2 長語を出力する。

uオプションがない場合は、リテラルを 4 つ並べて 2 長語とする。uオプションがある場合は、MSB 側から 0-origin で数えて 1 番目と 3 番目の短語を代わりに 0 にする。短語リテラルを x、短語の 0 を oと書くと、u オプションがない場合は xxxx、ある場合は xoxoのようになる。

uオプションの用途としては、仮数部の下 32 ビットが 0 であるような倍精度語を直接生成することが挙げられる。

例

```
immu s"1" $llr0
```

$llr0のデータは、MSB 側から半精度ごとに区切って書くと 0x0001_0001_0000_0000_0001_0001_0000_0000 となる。

3.6.12.3 msl, msr - PE 間循環シフト命令

各 PE から隣の番号の PE にサイクルあたり 1 長語を転送する。msl は PE0 の入力を PE1 で出力し、以下同様に 1 → 2、2 → 3、3 → 0 と転送される。msr はこれの逆向きである。

例

```
msl $lr0v $lr0v
```

PE0,1,2,3 の GRF0 にある 4 長語を、それぞれ PE1,2,3,0 の GRF0 にコピーする。

3.6.12.4 passa - コピー命令

passa命令は、入力の 2 長語をそのままコピーして出力する。

2 長語動作により MSB1 長語だけでなく LSB1 長語でも入力のコピーが行われるが、マスクフラグ生成においては LSB 側は無視され、MSB 側の長語に含まれる 1 長語、2 短語、または 4 半語のみが参照される。

例

```
lpassa $llr0v $lls0v
```

GRF0 から GRF1 に計 8 長語のコピーを行う。

3.6.12.5 inc, dec - インクリメント・デクリメント命令

inc命令と dec命令は整数値のインクリメント・デクリメントを行う。

入出力ともに、符号ありモードでは符号あり整数、符号なしモードでは符号なし整数として扱う。

どちらのモードでもオーバーフロー時は値はラップアラウンドする。

例

```
zero $nowrite
sdec $aluf $lr0v
```

マイナス方向へのラップアラウンドが起き、GRF0 には 4 長語の all 1 が書き込まれる。

3.6.12.6 not - ビット反転命令

not命令はビット反転を行う。

例

```
zero $nowrite
lnot $aluf $lr0v
```

GRF0 には 4 長語の all 1 が書き込まれる。この lnotは論理否定命令 (第 3.6.12.7 節) ではなく、精度指定が長語のビット反転命令である。

3.6.12.7 lnot - 論理否定命令

lnot命令は論理否定を返す。すなわち、入力が all 0 なら 1 を返し、そうでなければ 0 を返す。

例

```
ilnot $subpeid $lr0
```

$lr0には 0 番 PE でのみ単精度整数の 1 が 2 つ並んだ値が入り、それ以外の PE では all 0 となる。

3.6.12.8 rsqrt - 近似逆数平方根命令

`rsqrt`命令は逆数平方根の近似値を返す。入力の符号ビットは無視される。精度は 5 ビット程度である。正確な値が必要なときはこれを初期値としてニュートン法などを適用する。

3.6.12.9 floor - floor 命令

`floor`命令は入力を浮動小数点数として解釈し、小数点以下がゼロの最も近い浮動小数点数に丸める。丸めはマイナス無限大方向に行う。無限大はそのまま出力する。結果がゼロのときは仮数部をゼロフラッシュする。

3.6.12.10 ftoi - 整数への変換

`ftoi`命令は入力を浮動小数点数として解釈し、最も近い整数に丸める。丸めはゼロ方向に行う。符号なしモードのときは入力の絶対値を符号なし整数に丸める。入力が無限大の場合を含め、結果が最大の整数値を超えるときはその値にクリップする。

3.6.12.11 bfe, bfn - ブロックフロート化命令

行列ベクトル積和演算の積の部分に対する入力はすべて、その演算精度に対応するブロックフロート (BF) 化命令で BF 化を行っておかなければならない。これはおおまかには、内積回路の 2 入力のそれぞれの中で指数部の値を揃えておく操作である。このため、BF 化命令は他の ALU 命令と異なり、要素ごとに独立な演算ではない。オペコードの文法は次の通りである。

オペコードの文法

```
  dbfn
| fbfn
| gbfn
| hbfn/<n>
| hbfe/<n>
```

先頭の `d`, `f`, `g`, `h`はそれぞれ倍精度、単精度、疑似単精度、半精度を表す。

内積回路のサイズは倍精度と単精度では 4 語、疑似単精度では 8 語、半精度では 16 語なので、BF 化もこれらの語数ごとに行われる。この単位を本節ではブロックと呼ぶ。

倍精度では、各 PE の MSB 側 1 長語からなる 1 ブロックに対して BF 化を行う。

単精度では、各 PE の MSB 側 1 長語の、さらに MSB 側 1 短語からなるブロックと、LSB 側 1 短語からなるブロックの 2 ブロックに分けて BF 化を行う。単精度 MAU 演算では MSB 側短語を利用するため、LSB 側短語を演算に利用するときは LM 等に書き込んだのち短語で読み出すなどする。

疑似単精度では、各 PE の MSB 側 1 長語からなる 1 ブロックに対して BF 化を行う。

半精度 BF 化命令は `hbfn`と `hbfe`のいずれも、サイクルあたり各 PE から 2 長語を読み出して計 32 半精度語を入力とする。各 PE の MSB 側 1 長語からなるブロックと、LSB 側 1 長語からなるブロックの 2 ブロックに分けて BF 化を行う。

以上のように、BF 化の PE あたりでのスループットは、半精度以外では 1 長語/サイクル、半精度では 2 長語/サイクルである。

`hbfe`はブロックの中で相対的に指数部が小さい数について、通常の浮動小数点数フォーマットで言う非正規化数に近い表現を用いてアンダーフローを防ぐ処理を行う。これを拡張表現と呼ぶ。`hbfn`のときは拡張表現を用いない。`<n>`は 6 から 9 までの整数で変換後の仮数部長を指定する。`<n>`を小さくすると、演算精度が低下する代わりに電力消費量の低減が期待される。

半精度以外では拡張表現や仮数部長指定機能は存在しない。

3.6.12.12　max, min - 最大値・最小値命令

max命令と min命令はともに 2 入力で、max命令は小さくない方の値を、min命令は大きくない方の値を出力する。本節ではこれ以降、第 1 入力を x、第 2 入力を y で表す。

マスクフラグは x が選ばれたときに 1 になる。これには x と y が等しかった場合も含まれる。

精度指定に浮動小数点数と整数の両方を指定でき、動作が異なる。

整数指定の場合は整数としての通常の比較を行う。符号なしオプションがあり、入力を符号ありと符号なしのどちらで解釈するかを指定可能である。

浮動小数点数指定の場合は max命令と min命令ともに、符号あり整数比較を基本として、0 を同一視するような動作をする。以下、詳細に述べる。特に言及がなければ、マスクフラグは x が出力される場合に 1 になる。x と y の全ビットが等しい場合はその値が出力され、マスクフラグは 1 になる。x と y が全ビット一致でなく、かつともに浮動小数点数の 0 のとき (すなわち指数部の全ビットが 0 のとき) は、x が出力される。x と y が全ビット一致でなく、かつともに同符号の無限大のときは、仮数部の比較を行って出力を決める。以上のどれにも当てはまらないときは、通常の浮動小数点数の比較を行って出力を決める。

3.6.12.13　packbit - 符号部パック命令

packbit命令は第 1 入力を x、第 2 入力を y として $x \ll 1$ と y の MSB とのビット論理和を返す。

この命令は、ReLU 系命令の第 1 引数のように値の符号の情報のみがあればよい場合に、データ量を節約するために使うことができる。

例

```
hpackbit $msb1 $lr0v $nowrite
hpackbit $aluf $lr8v $nowrite
hpackbit $aluf $lr16v $nowrite
hpackbit $aluf $lr24v $ls0v
```

1 半語につき 4 半精度語の符号ビットをパックする。$msb1は左シフトにより実質的なゼロ初期値として扱うことができる。

3.6.12.14 and, or, xor - ビット論理積・論理和・排他的論理和命令

`and`命令、`or`命令、および `xor`命令はビットごとの論理積・論理和・排他的論理和演算を行う。

ビットごとの演算のため、出力は精度指定によらないが、マスクフラグは精度指定に影響される。

3.6.12.15 add, sub - 加算・減算命令

`add`命令、`sub`命令は整数の加算・減算を行う。

入出力ともに、符号ありモードでは符号あり整数、符号なしモードでは符号なし整数として扱う。

どちらのモードでもオーバーフロー時は値はラップアラウンドする。

3.6.12.16 lsl, lsr, bsl, bsr - シフト命令

`lsl`命令は論理左シフト、`lsr`命令は算術右シフトまたは論理右シフト、`bsl`命令はバレル左シフト、`bsr`命令はバレル右シフトを計算する。

第 2 入力がシフト量となる。

`lsr`命令は符号ありモードでは算術右シフト、符号なしモードでは論理右シフトになる。これ以外のシフト命令に符号なしオプションはない。

論理シフトではシフトして空いたビットには 0 が埋められる。算術右シフトではシフトして空いたビットは入力の符号ビットで埋められる。バレルシフトではシフトアウトされたビットが回り込んで空いたビットを埋める。

シフト量が語長を超える場合の動作について述べる。語長を n (精度指定に従って 16, 32, または 64)、元のシフト量を s と置く。まず $s_0 := s \bmod 2n$ とする。つまり語長と等しいシフト量を表せるビット数より上のビットは無視される。その後、$s_0 < n$ ならば上の定義どおりにシフトを行う。$n \leq s_0$ ならば、バレルシフトでは $s_0 - n$ だけシフトする。算術シフトと論理シフトでは n だけシフトする (すべてシフトアウトされるので、結果は all 0 または all 1 になる)。

3.6.12.17　ReLU 系命令

ReLU 系命令はいずれも 2 入力で、表 3.12 で定義される。

入出力値はすべて浮動小数点数である。

`relu0`命令は `relu`命令へのエイリアスである。

表 3.12　ReLU 系命令の定義。第 1 入力を x、第 2 入力を y で表す。

オペランド	定義
`relu/relu0`	x の MSB が 0 のとき y を、1 のとき -0 を返す
`relu1`	x の上から (1-origin で) 2 番目のビットが 0 のとき y を、1 のとき -0 を返す
`relu2`	x の上から 3 番目のビットが 0 のとき y を、1 のとき -0 を返す
`relu3`	x の上から 4 番目のビットが 0 のとき y を、1 のとき -0 を返す
`lrelud`	x の MSB が 0 のとき y を、1 のとき $y/2$ を返す
`lreluo`	x の MSB が 0 のとき y を、1 のとき $y/8$ を返す
`ilrelud`	x の MSB が 0 のとき y を、1 のとき $2y$ を返す

`lrelud`命令、`lreluo`、および `ilrelud`命令の `l`は leaky の頭文字である。`ilrelud`命令の `i`は inverse の頭文字である。

`lrelud`命令と `lreluo`命令は結果がアンダーフローする可能性がある。その際は符号ビットが負で、指数部と仮数部が 0 の値を返す。

`ilrelud`命令は結果がオーバーフローする可能性がある。その際は指数部の全ビットが 1 で、符号と仮数部は入力と同じである値を返す。

これらの定義は ReLU 系関数の forward と backward の両方を計算できるようにするためのものである。浮動小数点数の MSB は符号ビットであるから、`relu`命令の定義は概ね、x が非負なら y、負なら 0 を返すというものになる。ここで v を入力とした ReLU 関数の forward の結果を w と置く。これは `relu`命令を用いて $w =$`relu`(v, v) で計算できる。ReLU 関数の backward、すなわち w に関する勾配 w' から v に関する勾配を求める計算をしたい場合は、`relu`(v, w') または `relu`(w, w') を実行すればよい。この後者は `relu`(v, v) の結果が v の符号を保存することから可能になっている。

3.6.12.18　ALU への入力の単精度から半精度への丸め

ALU 命令式の入力オペランドが半精度浮動小数点数を期待するとき、単精度浮動小数点数を丸めて入力する指示ができる。

オペランドの末尾に rオプションを付加すると、2 長語を 4 つの単精度浮動小数点数として解釈し、4 つの半精度浮動小数点数に丸めたものを MSB 側 1 長語に置き、LSB 側 1 長語は 0 にした 2 長語が ALU に入力される。

2 入力のオペコードであっても、第 1 入力と第 2 入力に対してそれぞれ独立に、rオプションを付加するかを決められる。

例

```
hrsqrt $llr0vr $lm0v
```

サイクルごとに、GRF0 から読み出した 2 長語を 4 単精度語として解釈し、4 半精度語に丸めてから近似逆数平方根命令を実行する。

3.6.13 wait - PE 命令に MV 命令を待機させる

MV 命令の必要サイクル数は静的には決まらないため、PE 命令はタグによって対応する MV 命令を待機できるようになっている。

`wait`命令式が書かれたステップでは、直前で命令の発行を止め、対応する MV 命令の完了通知を受信するまで待機する。

`wait`命令式はそれのみでは PE 命令文になれず、他の PE 命令式と同時発行する必要がある。

PE 命令と MV 命令は単一の命令ストリームに混載されるため、後続の PE 命令だけでなく後続の MV 命令の発行もなされなくなることに注意する。

文法

```
wait <tag>
```

`<tag>`は第 3.2.3 節で定めたタグである。

効果

直前で命令の発行を止め、タグが`<tag>`である MV 命令の完了通知を受信するまで、後続の命令の代わりに nop を送出し続ける。

エラー

- 他の PE 命令式が同時に発行されていないとエラーになる。
- タグ番号に 0 を指定するとエラーになる。

例

```
mvp/n64i01 $lc0@.0 $d0
l2bmrdfadd $lb0 $lc0; wait i01
```

L2BM → DRAM 並列個別転送を開始し、それが完了するまで L1BM から L2BM への書き込みを待機させる。