MN-Core™ 2 Software Developer Manual

Preferred Networks, Inc.

November 15, 2024

Abstract

This document explains the specification of hardware and assembly instruction necessary for developing MN-Core 2 applications.

Contents

Chapter 1 Overview of MN-Core 2 Architecture 3
1.1 Structure 3
1.2 Overview of Machine Code 3
1.3 Word Length and Operation Precision 5
1.4 Performance of floating point operations 7
1.5 Memory Performance 7

Chapter 2 Tool Chain 9
2.1 Real Device 9
2.2 Assembler 9
2.3 Emulator 9

Chapter 3 Developement with MN-Core 2 Assembly Lamnguage 10
3.1 Statements 10
3.2 Numerical Values 10
3.2.1 Natural Numbers 10
3.2.2 Immediate Values 11
3.2.3 Tags 11
3.3 Description of Instruction Operation in Pseudocode 11
3.4 Control Statements 12
3.4.1 Comments 12
3.4.2 quit Statement 12
3.4.3 d get Statement 13
3.4.4 d set Statement 17
3.5 MV Instruction Statements 20
3.5.1 Overview of Syntax 20
3.5.2 Operands 20
3.5.2.1 p - PDM 21
3.5.2.2 d - DRAM 21
3.5.2.3 c - L2BM in MV Instructions 21
3.5.3 Transfer Word Count Specification and Unit Operation 22
3.5.4 Tag Specification 22
3.5.5 Reduction Operation Specification 22
3.5.6 DRAM Indirection 23
3.5.7 Priority Specification 23
3.5.8 Basic Mode for MV Instructions 24
3.5.8.1 mvnop Instruction 24
3.5.8.2 PDM → DRAM Individual Transfer Instruction 25
3.5.8.3 DRAM → PDM Individual Transfer Instruction 26
3.5.8.4 PDM → L2BM Individual Transfer Instruction 27
3.5.8.5 L2BM → PDM Individual Transfer Instruction 28
3.5.8.6 DRAM → L2BM Individual Transfer Instruction 29
3.5.8.7 L2BM → DRAM Individual Transfer Instruction 30
3.5.8.8 PDM → PDM Individual Transfer Instruction 31

3.5.8.9 PDM → L2BM Parallel Transfer Instruction 32
3.5.8.10 L2BM → PDM Parallel Transfer Instruction 33
3.5.8.11 DRAM → L2BM Parallel Transfer Instruction 34
3.5.8.12 L2BM → DRAM Parallel Transfer Instruction 35
3.5.8.13 DRAM → L2BM Intra-Group Broadcast Instruction 36
3.5.8.14 L2BM → DRAM Intra-Group Broadcast Instruction 37
3.5.8.15 L2BM → PDM Intra-Group Reduction Instruction 38
3.5.8.16 DRAM → L2BM Inter-Group Broadcast Instruction 39
3.5.8.17 L2BM → DRAM Inter-Group Gather Reduction Instruction 40
3.5.8.18 PDM → L2BM Inter-Group Broadcast Instruction 41
3.5.8.19 L2BM → PDM Inter-Group Reduction Instruction 42
3.5.8.20 DRAM → L2BM Inter-Group Broadcast Instruction 43
3.5.8.21 L2BM → DRAM Inter-Group Reduction Instruction 44
3.5.8.22 PDM → L2BM Scatter Instruction 45
3.5.8.23 L2BM → PDM Gather Instruction 46
3.5.8.24 PDM → DRAM Scatter Instruction 47
3.5.8.25 DRAM → PDM Gather Instruction 48
3.5.9 Constraints between Multiple MV Instructions 49
3.6 PE Instruction Statements 49
3.6.1 Operands 49
3.6.1.1 c - L2BM in PE Instructions (Except for l2bmdars Instruction) . . 49
3.6.1.2 c - L2BM in l2bmdars Instruction 50
3.6.1.3 dar - DRAM Address Register 50
3.6.1.4 b - L1BM in L2BM Instructions 50
3.6.1.5 b - L1BM in L1BM Instructions 50
3.6.1.6 m - LM0 (Excluding Base Address Register Writes) 51
3.6.1.7 m - LM0 (Base Address Register Write) 52
3.6.1.8 n - LM1 (Excluding Base Address Register Writes) 52
3.6.1.9 n - LM1 (Base Address Register Write) 52
3.6.1.10 r - GRF0 52
3.6.1.11 s - GRF1 53
3.6.1.12 t - T-register 53
3.6.1.13 omr - Mask Register Write 53
3.6.1.14 x, y - Matrix Register 54
3.6.1.15 mauf - MAU Operation Result Forwarding 54
3.6.1.16 aluf - ALU Operation Result Forwarding 55
3.6.1.17 lbf - L1BM → PE Direction Transfer Forwarding 55
3.6.1.18 mreadf - Transposed Matrix Register Read Forwarding 55
3.6.1.19 nowrite - Dummy Output for Forwarding 56
3.6.1.20 Constant Input Operand 56
3.6.2 Mask Register 56
3.6.2.1 Write Mask Application 57
3.6.2.2 Applying Zero-flush Mask 59
3.6.3 Avoiding hazards 59
3.6.3.1 L1BM → L2BM Transfer ⇒ MV Instruction That Reads from L2BM 60
3.6.3.2 L1BM →L2BM Transfer ⇒ L2BM → L1BM Transfer 60
3.6.3.3 L2BM → L1BM Transfer ⇒ L1BM → L2BM Transfer / Internal Multicast 60
3.6.3.4 Internal Multicast ⇒ L1BM → L2BM Transfer / Internal Multicast 61
3.6.3.5 Internal Multicast ⇒ L1BM → PE Transfer 61
3.6.3.6 L2BM → L1BM Transfer ⇒ L1BM → PE Transfer 61
3.6.3.7 PE → L1BM Transfer ⇒ L1BM → L2BM Transfer / Internal Multicast 61

3.6.3.8 PE → L1BM Transfer ⇒ L1BM → PE Transfer 62
3.6.3.9 PE Memory Write ⇒ PE Memory Read 62
3.6.4 Simultaneous execution condition 63
3.6.5 nop - NOP 65
3.6.6 Nonforward - not updating forwarding and turnaround register 65
3.6.7 L2BM Instruction Expressions 66
3.6.7.1 L1BM Subset Specification 66
3.6.7.2 l2bmb - L2BM → L1BM Broadcast 68
3.6.7.3 l2bmb2 - L2BM → L1BM Distributing Broadcast 69
3.6.7.4 l2bmd - L2BM → L1BM Distribution 70
3.6.7.5 l2bm@<l1badr> - L1BM → L2BM Transfer 71
3.6.7.6 l2bmr<rrn_opcode> - L1BM → L2BM Reduction 72
3.6.7.7 l2bmr2<rrn_opcode> - L1BM → L2BM Comibing Reduction . . . 73
3.6.7.8 l2bmd - L1BM → L2BM Gather 74
3.6.7.9 l2bmi - Inter-L1BM Multicast 75
3.6.7.10 l2bmdars/l2bmdarw - DAR Write 76
3.6.8 L1BM Instruction Expressions 77
3.6.8.1 About 4x4 Mode 77
3.6.8.2 Types of L1BM Instruction Expressions and Turnaround Operations 77
3.6.8.3 Word Length of PE Side Operands 78
3.6.8.4 Precision Extension of Inputs 78
3.6.8.5 Precision Shortening of Reduction Results 78
3.6.8.6 l1bmp - Long-word PE Broadcast 80
3.6.8.7 l1bmp - Double-long-word PE Broadcast 81
3.6.8.8 l1bmm - Long-word 16x1MAB Broadcast 82
3.6.8.9 l1bmm - Double-long-word 16x1MAB Broadcast 83
3.6.8.10 l1bmm@<mabadr> - Long-word 16x1 Transfer 84
3.6.8.11 l1bmm@<mabadr> - Double-long-word 16x1 Transfer 85
3.6.8.12 l1bmr<rrn_opcode> - Long-word 16x1 Reduction 86
3.6.8.13 l1bmr<rrn_opcode> - Double-long-word 16x1 Reduction 87
3.6.8.14 l1bmm4 - Long-word 4x4MAB Broadcast 88
3.6.8.15 l1bmm4 - double-long-word 4x4MAB Broadcast 89
3.6.8.16 l1bmm4@<mabadr> - Long-word 4x4 Transfer 90
3.6.8.17 l1bmm4@<mabadr> - Double-long-word 4x4 Transfer 91
3.6.8.18 l1bmr4<rrn_opcode> - Long-word 4x4 Reduction 92
3.6.8.19 l1bmr4<rrn_opcode> - Double-long-word 4x4 Reduction 93
3.6.8.20 l1bmd - Distribution 94
3.6.8.21 l1bmd - Gather 95
3.6.9 MAU instruction Expressions 96
3.6.9.1 Basic Operations 96
3.6.9.2 dmfma - Basic Operation of Matrix-Vector Multiply-Add 97
3.6.9.3 dmmul - Double-precision Matrix-Vector Product 97
3.6.9.4 fmfma - Basic Operation of Single-precision Matrix-Vector Multiply-Add 98
3.6.9.5 fmmul - Single-Precision Matrix-Vector Product 98
3.6.9.6 gmfma - Pseudo-Single-Precision Matrix-Vector Multiply-Add . . . 99
3.6.9.7 gmmul - Pseudo-Single-Precision Matrix-Vector Product 99
3.6.9.8 hmfma - Basic Operation of half-precision Matrix-Vector Multiply-Add 100
3.6.9.9 hmmul - Half-Precision Matrix-Vector Product 100
3.6.9.10 dvmul - Double-Precision Vector Multiply-Add 101
3.6.9.11 dvmul - Double-Precision Vector Product 101
3.6.9.12 dvadd - Basic Operation of Double-Precision Vector Sum 102

3.6.9.13 dvpassa - Double Precision Vector Copy . . . 102
3.6.9.14 fvfma - Single-Precision Vector Multiply-Add . . . 103
3.6.9.15 fvmul - Single-Precision Vector Product . . . 103
3.6.9.16 fvadd - Single-Precision Vector Sum . . . 103
3.6.9.17 fvpassa - Single-Precision Vector Copy . . . 103
3.6.9.18 hvfma - Basic Operation of Half-Precision Vector Multiply-Add . . 104
3.6.9.19 hvmul - Half-Precision Vector Product . . . 104
3.6.9.20 hvadd - Half-Precision Vector Sum . . . 104
3.6.9.21 hvpassa - Half-Precision Vector Copy . . . 104
3.6.9.22 Input Sign Inversion . . . 105
3.6.9.23 Precision Extension of Inputs . . . 105
3.6.9.24 Input Shortening of Inputs . . . 106
3.6.9.25 Precision Shortening of Output . . . 106
3.6.9.26 Mask Flags Generated by MAU Instruction Expressions . . . 106
3.6.10 Matrix Register Write Instruction Expressions . . . 106
3.6.10.1 dmwrite - Double-Precision Matrix Register Write . . . 106
3.6.10.2 mwrite/gmwrite - Single-Precision/Pseudo-Single-Precision Matrix Register Write . . . 108
3.6.10.3 hmwrite - Half-Precision Matrix Register Write . . . 109
3.6.11 Transposed Matrix Register Read Instruction Expressions . . . 110
3.6.11.1 dmread - Double-Precision Transposed Matrix Register Read . . . 110
3.6.11.2 fmread/gmread - Single-Precision Transposed Matrix Register Read 111
3.6.11.3 hmread - Half-Precision Transposed Matrix Register Read . . . 112
3.6.12 ALU Instruction Expressions . . . 113
3.6.12.1 Mask Flags Generated by ALU Instruction Expressions . . . 114
3.6.12.2 zero - Zero Output Instruction . . . 116
3.6.12.3 imm - Immediate Value Instructions . . . 116
3.6.12.4 msl, msr - Inter-PE Circular Shift Instruction . . . 116
3.6.12.5 passa - Copy Instruction . . . 117
3.6.12.6 inc, dec - Increment / Decrement Instruction . . . 117
3.6.12.7 not - Bit Reverse Instruction . . . 117
3.6.12.8 lnot - Logical Not Instruction . . . 117
3.6.12.9 rsqrt - Approximate Reciprocal Square Root Instruction . . . 118
3.6.12.10 floor - Floor Instruction . . . 118
3.6.12.11 ftoi - Conversion from Floating-Point to Integer . . . 118
3.6.12.12 bfe, bfn - Block Floating Point Conversion Instructions . . . 119
3.6.12.13 max, min - Maximum, Minimum Instructions . . . 120
3.6.12.14 packbit - Packing Sign Bit Instruction . . . 120
3.6.12.15 and, or, xor - Bitwise Logical AND, OR, Exclusive OR Instructions 121
3.6.12.16 add, sub - Addition, Subtraction Instructions . . . 121
3.6.12.17 lsl, lsr, bsl, bsr - Shift Instructions . . . 121
3.6.12.18 ReLU-related Instructions . . . 122
3.6.12.19 Precision Shortening from Single-Precision to Half-Precision for the Inputs of the ALU . . . 123
3.6.13 wait - Make PE instructions Wait for MV Instructions . . . 124
3.7 Examples of MN-Core 2 Assembly Programs . . . 124
3.7.1 Conversion from Integer to Floating-point Value . . . 124
3.7.2 Multiple Input to Computation Unit from PE Memory Read Operand . . . 124
3.7.3 Double-long-word Output from Computation Unit to PE Memory Write Operand . . . 125

Chapter 4 Details of MN-Core 2 Floating-Point Operations 126
4.1 Outpu Normalization . . . 126

4.2 RNN . . . 126
4.2.1 RRN Floating Point Addition . . . 126
4.2.2 RNN Floating Point Maximum Value and Minimum Value . . . 127
4.3 Vector Multipyl-Add . . . 127
4.4 Block-floating Conversion . . . 129
4.5 Matrix-Vector Multiply-Add . . . 131

List of Tables

3.1 Data format interpretation specification of the d get statement 13
3.2 Combinations of memory elements and word lengths that can be read by the d get statement 14
3.3 Combinations of memory elements and word lengths that can be written by d set statements 18
3.4 Notations for long-word data written by d set statements 18
3.5 List of constant input operands 56
3.6 Division of PE instruction expressionss (describing simultaneous execution condition) . . 63
3.7 Correspondance of the pair of l1badr and immode to L1B set 67
3.8 The list of L1BM instruction expr. 77
3.9 Table of ALU Instruction Opcodes 114
3.10 Correspondance between ALU operation types and acceptable precisions 114
3.11 Mask Flags Generated by ALU Instruction Expressions 115
3.12 The definition of ReLU-related Instructions 122

4.1 The block size of block-floating point format and source precision of floating-point number129

Chapter 1

Overview of MN-Core 2 Architecture

1.1 Structure

MN-Core 2 is a SIMD accelerator board composed of a tree-shaped hierarchical memory structure with support for mutual data transfers between memory units and a large number of matrix-vector product dedicated circuits at the leaves of the tree.These operations are performed concurrently using VLIW instructions, allowing for high performance and low power consumption.

There are no hardware-managed caches in MNCore 2, and all the intra-board data transfers are explicitly managed by using machine code instructions.Instructions are sent to the board in a single stream and there are no mechanisms to manage the control flow of the execution.

Instead of caches, the leaves of the tree have large local memories (SRAM) in addition to the computational units. By placing the data so that the data movement remains as close to the leaves of the tree as possible, high-bandwidth data transfers can be achieved at low cost, and the efficiency of the operations can be improved.

A single board consists of a single chip and peripheral circuits. The chip is composed of a top level, which corresponds to the root of the tree, and eight L2Bs (Level 2 Blocks) as its children.

The structure below the L2B is as follows:

- Each L2B has 8 **L1B**s (Level 1 Blocks) as its children.
- Each L1B has 16 **MAB**s (Matrix Arithmetic Blocks) as its children.
- Each MAB has 4 **PE**s (Processing Elements) as its children, and a **MAU** (Matrix Arithmetic Unit) as well.

Thus, for example, there are 4096 PEs per board.

L2Bs and L1Bs each have local SRAMs, called **L2BM** and **L1BM**, respectively. PEs consist of several types of local memories and an **ALU** (Arithmetic Logic Unit).

L2Bs are divided into 4 **group**s, each containing 2 L2Bs. Each group has an SRAM called **PDM** (PIU Data Memory, PIU stands for PCIe Interface Unit) and a DRAM. The top level can transfer data on the PDM, DRAM, and L2BM memories (**upper memories**) between its own group and other groups. The PDM of group 0 is connected to the host via a PCIe interface, and all input/output data communication with the host goes through the PDM. The DRAM acts as the main memory.

The upper memories and L1BM and local memories inside PEs form the hierarchical memory structure mentioned at the beginning, and the MAU corresponds to the large number of matrix-vector product dedicated circuits at the leaves of the tree.

1.2 Overview of Machine Code

Machine code instructions consist of data transfer instructions (MV instructions) that control data transfers between upper memories and PE instructions that control L2B and below*1.

*1 PE instructions control elements above the hardware element ”PE” but are conventionally called this way.

PE instructions control the entire L2Bs and below in 4 cycles*2. This 4-cycle unit is called a **step**.

PE instructions are in VLIW format, allowing multiple memory units and arithmetic units to be controlled simultaneously in a single instruction. Software pipelining is the key to improving performance by packing as many operations as possible into a single instruction and minimizing the time the units are idle.

Instruction execution is pipelined, and the results cannot be used immediately after the instruction is issued. It is the programmer's responsibility to leave cycles until the value written to one of the memories can be read.

Inside the PE, there are 5 memory elements: GRF (General Register File) 0, GRF1, T-register (Temporary Register), LM (Local Memory) 0, and LM1. These are collectively referred to as **PE memories**.

PE memory is connected to 3 **arithmetic units**: ALU, MAU, and L1BM*3.

Figure 1.1 shows the structure of a PE instruction.

PE memory can read or write up to 2 long words per cycle. The data path between the arithmetic units and PE memory is always 2 long words per cycle. For results shorter than 2 words, they are always placed on the MSB side of the 2 long words. Here, MSB refers to the Most Significant Bit. We will use LSB (Least Significant Bit) and MSB without further explanation. The storage format is big-endian. That is, if it is addressable, the MSB side corresponds to the smaller address.

Examples to understand the relationship between memory access word length and the placement of values in the data path will be shown later in Section 3.7.2.

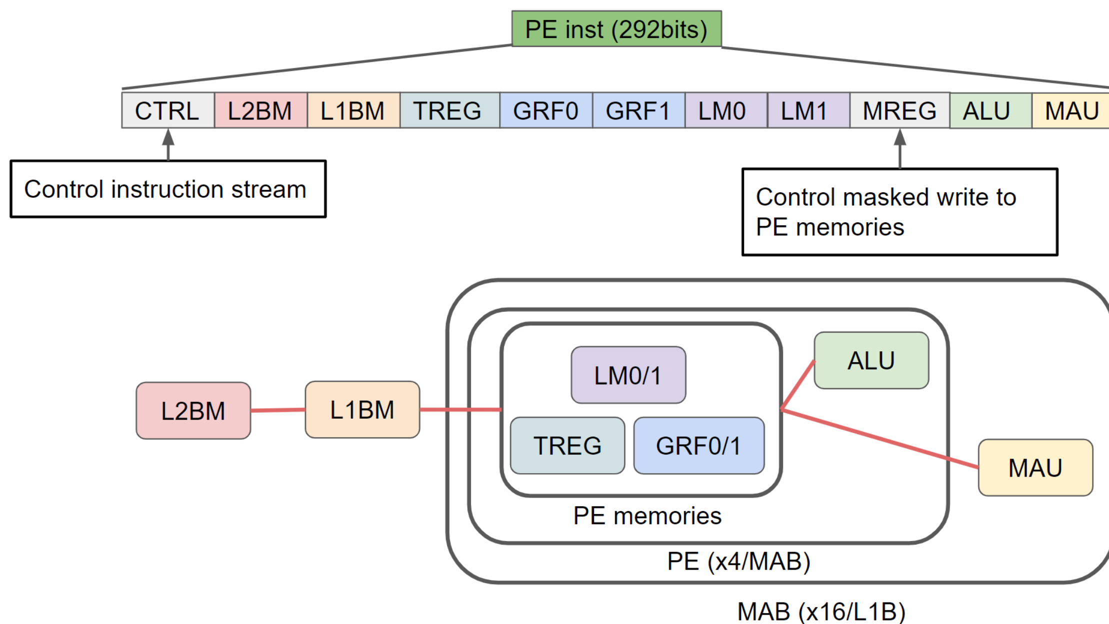

Fig. 1.1 Structure of PE instruction

The instruction stream has two modes: the **auto-stride mode** and the **flat mode**.

In the auto-stride mode, an instruction stream unit consists of 3 PE instructions and 1 MV instruction. In the flat mode, an instruction stream unit consists of 2 PE instructions and 1 MV instruction. The instruction stream is **packed** if it is unified in either the auto-stride or flat mode. The instructions to be sent to the actual machine must be packed. The emulator can execute instruction sequences that are not packed, i.e., sequences where PE and MV instructions appear in any order. Details will be described

*2 This is because 1 cycle per instruction would saturate the PCIe bandwidth between the host and the board.

*3 L1BM is also the name of the memory in L1B, but here it is also called an arithmetic unit as it is a unit that exchanges data with the PE. It also has a circuit that performs reduction operations.

in Section 2.3.

The instruction sequences that can be expressed in the auto-stride mode are a subset of the instruction sequences that can be expressed in the flat mode.

In the auto-stride mode, the address value of the PE instruction is incremented by a certain value every cycle within a step. This allows different operations to be performed in multiple cycles within a single instruction. Furthermore, the combination of 3 PE instructions and 1 MV instruction forms an instruction stream unit, which is repeated.

The flat mode is a mode that allows flexible address specification in exchange for a larger instruction bandwidth per step. Specifically, in each cycle within a step, the address values of GRF0, GRF1, LM0, and LM1 are specified. The addresses of other memories are determined by adding an automatically determined increment value to the address of the 0th cycle, as in the auto-stride mode. Furthermore, the combination of 2 PE instructions and 1 MV instruction forms an instruction stream unit, which is repeated.

It is necessary to specify whether to use the auto-stride or flat mode at assembly time. Since PE instructions that can be expressed in auto-stride mode can also be expressed in the flat mode, an assembly that mixes PE instruction expressions from both modes can be processed in the flat mode. Conversely, if an assembly that includes flat-mode PE instruction expressions is assembled in the auto-stride mode, it will result in an error, even if it was possible to rewrite the instruction expressions in the auto-stride mode without changing the meaning.

After an MV instruction is issued, it is executed asynchronously with respect to subsequent instructions. However, PE instructions can include a wait instruction[*4], in which case the instruction stream waits until just before the issuance of that PE instruction for the specified MV instruction to complete.

In other words, if you want to issue an MV instruction after issuing a PE instruction, you can simply write the instruction stream in that order, and if you want to issue a PE instruction after the completion of an MV instruction, you can use the wait instruction.

In both PE and MV instructions, data transfers are basically possible in the opposite direction of the transfer. For example, if there is a mode for distribution (sending data by dividing it equally) from an upper layer to a lower layer, there is basically a mode for combining (reading and sending data of equal size and writing it by connecting it) from a lower layer to an upper layer, and the layout is such that the distributed data is joined as it is to complete the same data in the upper layer, including the layout. This allows data layout consistency to be maintained.

Figure 1.2 shows the structure of the instruction stream and the control range of PE and MV instructions.

1.3 Word Length and Operation Precision

The minimum access unit for PDM and DRAM is 1 word = 64 bits. Due to alignment constraints for each instruction type, the address cannot always be specified in this unit.

For L2B and below, the basic access unit is 1 long word = 64 bits. In PE, some accesses are possible in 1 single word = 32 bits and 4 words = 2 long words = 128 bits.

In PE instructions, the following three types of operations are supported for both integers and floating-point numbers:

- Half word (half precision): 16 bits
- Single word (single precision): 32 bits
- Long word (double precision): 64 bits

For example, if a half-precision operation is performed on a value accessed in long words in PE, SIMD operations are executed on the 4 half words contained in 1 long word.

Integers can be either unsigned or signed. Signed integers use two's complement representation.

The bit lengths of each part of the floating-point number format are as follows:

[*4] Since instructions have a fixed bit width, the wait instruction is always included in the PE instruction, and it is possible to specify not to wait for any MV instruction as an option.

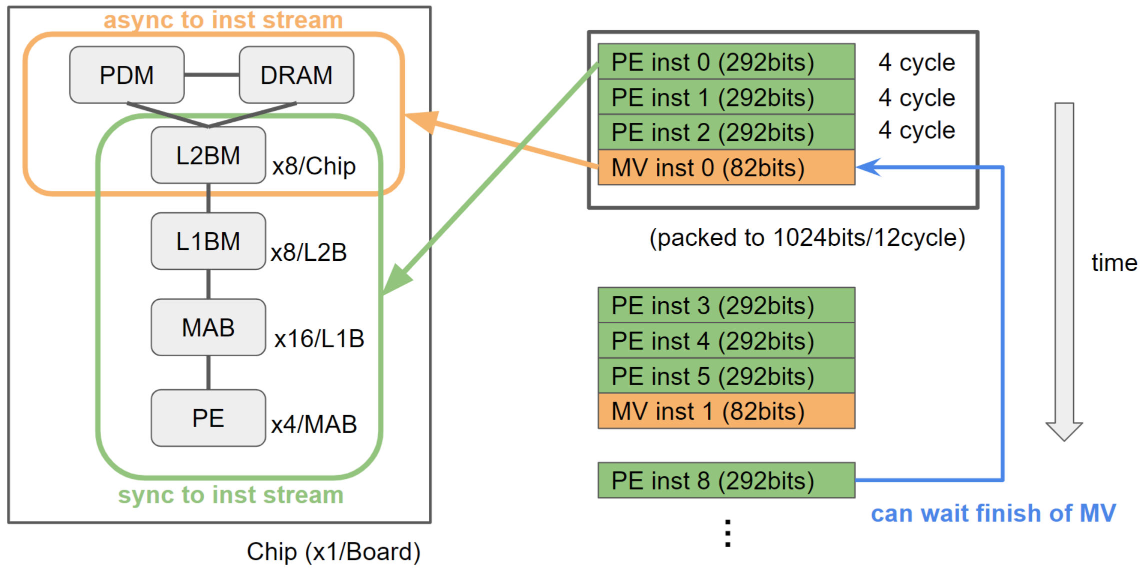

Fig. 1.2 Structure of instruction stream and control range. The right side shows the instruction stream in the auto-stride mode.

- Half precision: 1-bit sign, 6-bit exponent, 9-bit mantissa
- Single precision: 1-bit sign, 8-bit exponent, 23-bit mantissa
- Double precision: 1-bit sign, 11-bit exponent, 52-bit mantissa

Floating-point numbers consist only of normalized numbers, positive and negative zeros, and positive and negative infinities. There are no denormalized numbers or NaNs.

If the exponent part is all 0, it represents zero regardless of the value of the mantissa part, and if it is all 1, it represents infinity regardless of the value of the mantissa part. In each case, the sign bit determines whether it is positive or negative zero or infinity. All other values are normalized numbers and are interpreted according to the IEEE 754 format. However, for half precision, note that the number of bits in the exponent and mantissa parts is different from the binary16 format specified in IEEE 754.

Note that while integers do not have a -0, floating-point numbers do.

The mantissa part uses a hidden bit representation. For example, the half-precision `1.0` is represented as `0x3e00` in bits.

One word of a double-precision value stores the sign bit, the upper bits of the exponent, and the upper bits of the mantissa from the MSB to the LSB of the long word. Two words of a single-precision value store the MSB side and LSB side of the long word, respectively, in the same order. Four words of a half-precision value store 16 bits each from the MSB side of the long word in the same order. The same applies to single-precision 4 words within 2 long words = 128 bits and integer values.

When accessing in single words in the PE, the lower side (even side) of the single-word address corresponds to the MSB side of the long-word access, and the upper side (odd side) corresponds to the LSB side of the long-word access.

When performing matrix-vector product operations, it is necessary to perform a block-floating-point conversion in advance, and the block-floating-point numbers have a different format from the one described above*5. Specifically, while the number of bits in each part is the same as in normal floating-point numbers, no hidden bit representation is used in the mantissa part.

There are no types such as floating-point or integer types in the assembly. That is, it is possible to

*5 In MN-Core, block floating point was used only for half precision, but in MN-Core 2, it is used for all precisions.

use a floating-point value output from the MAU as an input for integer operations in the ALU and vice versa, and there is no software check for this.

1.4 Performance of floating point operations

MAU (Matrix Arithmetic Unit) performance is described. MAU has two modes: matrix-vector multiply-add and vector multiply-add mode.

In matrix-vector multiply-add mode, one MAU can perform the operation $y = A \times b + c$ in one cycle for an $m \times n$ matrix A and n-dimensional vectors b and c. Here, m, n, and the operation precision are one of the following:

- Double precision: $m = 2, n = 4$, A and b are block-floating-point double precision, c and y are normal double precision
- Single precision: $m = 8, n = 4$, A and b are block-floating-point single precision, c and y are normal single precision
- Pseudo-single precision: $m = 8, n = 8$, A and b are block-floating-point pseudo-single precision, c and y are normal single precision
  - Here, block-floating-point pseudo-single precision is a block-floating-point format with a shorter effective mantissa length than single precision. Pseudo-single precision only appears in block-floating-point format.
- Half precision: $m = 16, n = 16$, A and b are block-floating-point half precision, c and y are normal single precision

Therefore, the peak FLOPS value per board in double precision for matrix-vector multiply-add mode is $2 \times 4 \times 2 \times 1024$ multiplied by the frequency. Here, the last 2 counts one multiply-add operation as 2 FLOPs.

In data transfer from PE to L1BM and other upper layers, reduction by floating-point addition is possible, but since the nature of the arithmetic unit is significantly different from MAU, it is not counted in the official peak performance value.

In vector multiply-add mode, one MAU can perform the operation $y = a \times b + c$ in one cycle for m-dimensional vectors a, b, and c. Here, $\times$ and $+$ are element-wise independent operations, and m and the operation precision are one of the following:

- Double precision: $m = 4$, a, b, c, and y are all normal double precision
  - However, only the 0th and 1st elements or the 2nd and 3rd elements of the product can be enabled. The disabled side will have a product term of 0.
  - To execute the entire 4-element FMA, this operation should be executed twice. In that case, the result of the first operation is passed to c in the second operation.
- Single precision: $m = 8$, a, b, c, and y are all normal single precision
- Half precision: $m = 16$, a and b are normal half precision, c and y are normal single precision

Therefore, the peak FLOPS value per board in double precision for vector multiply-add mode is $2 \times 2 \times 1024$ multiplied by the frequency. Here, the 2 counts one multiply-add operation as 2 FLOPs again.

Note that there is no equivalent to pseudo-single precision in vector multiply-add mode since $m = 8$ single precision FMA is possible.

1.5 Memory Performance

Describes the size and throughput of each memory type. In practice, throughput is subject to various constraints such as available transfer types and read/write switching overhead, which will be explained sequentially.

- PDM: Capacity of 4 MiB per group.
- DRAM: Capacity of 4 GiB per group, bandwidth approximately 128 GB/s.

- L2BM: Capacity of 32 Ki long words per L2BM.
- L1BM: Capacity of 8 Ki long words per L1BM.
- LM0: Capacity of 2 Ki long words per PE. 1RW at 2 long words/cycle.
- LM1: Same as LM0.
- GRF0: Capacity of 256 long words per PE. 1R1W at 2 long words/cycle.
- GRF1: Same as GRF0.
- T-register: Capacity of 8 long words per PE. 1R1W at 2 long words/cycle.

Chapter 2

Tool Chain

2.1 Real Device

There is an API for data input/output between the host and DRAM and instruction execution for assembly implementations starting from DRAM. The explanation of this API is left to another document.

2.2 Assembler

The MN-Core 2 assembler converts programs implemented in the assembly language described in Chapter 3 into machine code. The standard binary name is `assemble3`.

When `assemble3` is in the PATH, the following command will output the assembly result to `pass.asm`.

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

2.3 Emulator

There is an emulator for the MN-Core 2 board. It is suitable for behavior confirmation because it does not require the instruction stream to be in packed format (Section 1.2) and can run control statements that output the contents of major memory elements in the assembly language (`d get` statement, Section 3.4.3).

The standard binary name is `gpfn3_package_main`.

The following is an example of assembling a handwritten instruction sequence in the file `sample.vsm` and running it in the emulator. It is assumed that `assemble3` and `gpfn3_package_main` are in the PATH.

The first line `lpassa` instruction writes the number of the PE to which it belongs to LM0 (0 to 3), and the second line `d get` instruction reads the contents of a specific MAB that was written. The value read by the `d get` instruction is written to the file specified by the `-d` option. Therefore, the file `sample.dmp` will contain values from 0 to 3 corresponding to the PE numbers.

For details on the `lpassa` instruction in the first line, refer to Sections 3.6.12.5, 3.6.1.20, and 3.6.1.6, and for details on the `d get` instruction in the second line, refer to Section 3.4.3.

```
$ 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
```

Chapter 3

Developement with MN-Core 2 Assembly Lamnguage

This chapter shows the syntax and effects of assembly instructions.

The following conventions are used to show syntax in the following sections:

- Parts enclosed in `<>` should be replaced with actual instruction statements as appropriate.
- `(A|B)` selects either `A` or `B`.
- Parts enclosed in `[]` are optional.

3.1 Statements

In the assembly language, one statement is written per line.

There are the following types of statements:

- Control statements (Section 3.4)
- Machine language instruction statements
  - MV instruction statements (Section 3.5)
  - PE instruction statements (Section 3.6)

3.2 Numerical Values

The following numerical values can be written directly in machine language instruction statements:

- Natural numbers (Section 3.2.1)
- Immediate values (Section 3.2.2)
- Tags (Section 3.2.3)

3.2.1 Natural Numbers

Natural numbers are used for addresses, etc. They are written in decimal by default, and can be written in binary, octal, and hexadecimal by using the prefixes `0b`, `0o`, and `0x`, respectively. Negative numbers cannot be specified.

The unit of addresses depends on the instruction. In PE memory (GRF0, GRF1, LM0, LM1), it is a word, and in other memories (L1BM, L2BM, PDM, DRAM), it is a long word. This is independent of the access word length specification.

3.2.2 Immediate Values

Immediate values are values output by the ALU with the `imm` instruction. The format of immediate values is as follows.

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

Note that double quotes are mandatory.

The first part specifies the numerical type. `(f|h)` represents single and half-precision floating-point numbers, `(i|s)` represents signed single and half-precision integers, and `(ui|us)` represents unsigned single and half-precision integers. Long word numerical types `d, l, ul` cannot be specified due to insufficient immediate value bits.

The actual value output by the immediate value instruction is a single word value specified by the payload literal, arranged as described in Section 3.6.12.3. Regardless of the numerical type specified above, the payload literal determines a single word value. If the numerical type is half-precision, the same value is repeated twice to form the single word value.

The interpretation of the payload literal is as follows.

- For floating-point numbers (`<floating-point-number-literal>`)
  - First, it is converted to a float value using C's `strtof`.
  - If single precision is specified, it is used as is; if half precision is specified, rounding is performed.
- For integers (`<integer-number-literal>`)
  - If it is a signed integer `i, s`, a sign `+/-` is allowed at the beginning.
  - The value after the sign is treated as a natural number.
  - If the value is outside the range of the numerical type, it results in an error.

Examples are shown below. The detailed definition of the result actually output by the ALU is left to the section on immediate value instruction expressions (Section 3.6.12.3).

Example 1
Outputs -1.0 as a single-precision floating-point number to LM1.

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

Example 2
Outputs 0x8000 as a half-precision unsigned integer to the T-register.

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

Note that if `imm s"0x8000" $t` is used in this example, it results in an error because it is outside the range of half-precision signed integers.

3.2.3 Tags

Tags are values used for synchronization between MV instructions and PE instructions. Tags must always start with `i` for identification, followed by a two-digit fixed hexadecimal tag value with leading zeros. Although it is in hexadecimal, it does not have the prefix `0x`.

Examples are left to the section on wait instruction expressions (Section 3.6.13).

3.3 Description of Instruction Operation in Pseudocode

In this document, the operation of some instructions is described in pseudocode. The syntax of the pseudocode is not strictly defined, but it follows the rules below.

- `MEM` is a structure that contains memory elements at each level of the MN-Core 2 board.

- `LongWord` represents a long word (64-bit word), `SingleWord` represents a single word (32-bit word), and `HalfWord` represents a half word (16-bit word).
- `for` is a normal for statement, but follows the rules below.
  - Index notation with `:` is inclusive at the start and exclusive at the end. If `for cycle = 0:4` is written, `cycle` takes `0,1,2,3`.
  - When `for cycle` is written, it indicates what happens in each cycle of the PE instruction for each value of `cycle`.
  - `forall group`, `forall l2b`, `forall l1b`, `forall mab`, `forall pe` are abbreviations for `for group = 0:4`, `for l2b = 0:2`, etc.
  - Direct nesting of `forall` is abbreviated as `forall chip,l2b,l1b`, etc.
- The following functions are used.
  - `refer_pemem(mem, cycle)` returns a reference to the PE memory from the operand notation of the PE memory and the cycle number.
  - `refer_matreg(side, wl)` returns a reference to the matrix register from the side and word length of the element.
    * The side of the matrix register is either `x` or `y`.
    * The word length of the element is either `LongWord`, `SingleWord`, or `HalfWord`.
    * Details of the matrix register are described in Section 3.6.1.14.
  - `get_unit_value(rrn_opcode)` returns the identity element of the reduction operation specified by `rrn_opcode`. Specifically, if the operation is fadd, iadd, bor, or lor, it returns all 0; if the operation is max, band, or land, it returns all 1; if the operation is min, it returns the maximum value of the signed integer for that precision.

3.4 Control Statements

Control statements control operations other than machine language instruction execution. Specifically, they command data communication between the board and the outside world, and the end of emulation.

Some control statements are only effective in the emulator.

3.4.1 Comments

Statements starting with `#` are comment statements and are ignored during translation.

3.4.2 quit Statement

All the instructions after the `quit` statement are ignored including itself when outputting the corresponding machine language.

It is useful for debugging because it allows you to effectively comment out the following text with a small edit.

Syntax

```
quit
```

Effects

All the instructions after this statement are ignored including itself when outputting the corresponding machine language.

Example

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

Only the first instruction statement is translated.

3.4.3 d get Statement

The d get (Debug Get) statement is a control statement for outputting the contents of PDM, DRAM, L2BM, L1BM, GRF0/1, LM0/1, T-register, matrix register, and mask register during emulator execution. It cannot be executed on the actual machine.

Note that the emulator does not operate cycle-accurately, and any instruction is completed immediately after being issued. In other words, the d get statement results in the same as reading the memory after waiting for a sufficient number of cycles.

The syntax and output format of the d get statement may change in future emulator updates.

Syntax

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

Here, <dtype> and <memory> are as follows, and the rest are natural numbers.

```
<dtype> ::= d|bd|f|bf|bg|h|bh
<memory> ::= $p<addr>
           | $d<addr>
           | $lc<addr>
           | $(l|ll)b<addr>
           | $[(l|ll)](r|s|m|n)<addr>
           | $[(l|ll)]t
           | $l(x|y)<addr>
           | $omr<addr>
```

Effects

The data of the specified memory element and address is dumped to the specified output file during emulator execution, with interpretation as a floating-point number.

The effects of each element of the syntax are explained in order below.

<dtype> specifies the data interpretation format. Regardless of the specification, the results are output as both floating-point numbers and integers. The numerical format selected by each specification is shown in Table 3.1. If any block-floating-point precision is specified, the result is interpreted within the range of one block, and if it becomes an invalid block-floating-point value (Section 4.4), an error occurs.

Table 3.1 Data format interpretation specification of the d get statement

<dtype>	Floating-point value	Integer
empty	Double precision	Half word and Long word
d	Double precision	Long word
bd	Block-floating-point double precision	Long word
f	Single precision	Single word
bf	Block-floating-point single precision	Single word
bg	Block-floating-point pseudo-single precision	Single word
h	Half precision	Half word
bh	Block-floating-point half precision	Half word

The part of <memory> before <addr> specifies the type of memory element and word length. The possible combinations are listed in Table 3.2. This is the same as the combinations that can be specified in the operands of actual MV and PE instructions. For the T-register, the actual instruction always accesses 2 long words, but in the d get statement, long-word access is possible by writing $t or $lt. Note that writing $t does not result in single-word access.

<addr> specifies the address. For PDM, DRAM, L2BM, and L1BM, it is in long words; for GRF0, GRF1, LM0, and LM1, it is in single words; for matrix registers, it is in rows; and for mask registers, it is in entries. This is also based on the rules of the operands of actual MV and PE instructions, as in

Table 3.2 Combinations of memory elements and word lengths that can be read by the d get statement

<memory>	Memory unit	Word length
$p	PDM	Long word
$d	DRAM	Long word
$lc	L2BM	Long word
$lb	L1BM	Long word
$llb	L1BM	2 long words
$r, $s, $m, $n	GRF0/GRF1/LM0/LM1	Single word
$lr, $ls, $lm, $ln	GRF0/GRF1/LM0/LM1	Long word
$llr, $lls, $llm, $lln	GRF0/GRF1/LM0/LM1	2 long words
$t, $lt	T-register	Long word
$llt	T-register	2 long words
$lx, $ly	Matrix register	Row line
$omr	Mask register	Entry

the part before <addr>. For the T-register, there is no address specification, and it always starts reading from the entry accessed in the first cycle.

[n<group_id>][c<l2b_id>][b<l1b_id>][m<mab_id>][p<pe_id>] limits the memory element number to output for each hierarchy of group, L2B, L1B, MAB, and PE. For grammatical reasons, if c or b is specified, n must also be specified. If not specified, all elements of that hierarchy are targeted. For example, if n0b1m2 is specified, values other than group 0, L1B 1, and MAB 2 are not displayed. Specifications for lower hierarchies than the target memory, such as c<l2b_id> for PDM output, are ignored.

<num_of_words> specifies in decimal how many words to read from <addr> based on the word length specified by <memory>. For the T-register, specifying a range from 1 to 4 allows reading multiple cycles starting from the first cycle. Note that when <memory> is $lt and <num_of_words> is 2, the second word read is the MSB side of the second cycle, not the LSB side of the first cycle.

Regarding the output format of matrix registers (Section 3.6.1.14). The <addr> and <num_of_words> for matrix registers are specified in rows, and the contents of one row of the matrix register are output in one line. For matrix registers, one of the <dtype> must be specified. This is because the actual access destination for matrix-vector multiply-add operations and matrix register writes changes depending on the specified precision. Depending on the <dtype>, <addr>, and <num_of_words> specified, it may exceed the number of rows in the matrix register for that precision, but unlike the mwrite instruction, wraparound does not occur in such cases.

Regarding the output format of mask registers (Section 3.6.2). Note that one entry of the mask register consists of 16 bits, with 4 cycles in the cycle direction and 4 bits in the word direction. The <addr> and <num_of_words> for mask registers are specified in entries, and the contents of one entry of the mask register are output in 4 lines, one for each cycle. Within a line, the 4 bits in the word direction are output as a single integer value. In this integer value output, the bits corresponding to the upper word are interpreted as the higher digits. For example, a mask written only to the smaller side of the word address in a long word is output as 0b1100, i.e., 12. If <num_of_words> is 2 or more, multiple entries are output continuously in the above format. The <dtype> specification for mask registers is ignored.

Errors

- If the word length specification by <memory> is shorter than the data format interpretation specification by <dtype>, an error occurs because it cannot be interpreted at that precision.

Example 1

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

The half precision `1.5` written to LM1 is read for the specified 1PE, with four of them lined up in a long word. The output is as follows.

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

Example 2

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

The long word integer L1B number written to GRF0 is read for each L1B, i.e., a total of 8PE. The output is as follows. Since `<dtype>` is not specified, the interpretation as a double-precision floating-point number for `f:`, as a half-word integer for `i:`, and as a long-word integer for `v:` are displayed.

```
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
```

Example 3

```
imm f"1.5" $nowrite
fmwrite $aluf $lx0
d getf $lx0n0c0b0m0 8
```

Writes single-precision data to the first four rows of the matrix register and read it. The output is as follows. The matrix register is 8 rows by 8 columns of single precision, and it can be confirmed that four rows have been written.

```
DEBUG-MRx(n0c0b0m0,0):{(1.5, 1.5) (0x3fc00000, 0x3fc00000), (1.5, 1.5) (0x3fc00000, 0x3fc00000
    ), (1.5, 1.5) (0x3fc00000, 0x3fc00000), (1.5, 1.5) (0x3fc00000, 0x3fc00000)} #d getf
    $lx0n0c0b0m0 8
DEBUG-MRx(n0c0b0m0,1):{(1.5, 1.5) (0x3fc00000, 0x3fc00000), (1.5, 1.5) (0x3fc00000, 0x3fc00000
    ), (1.5, 1.5) (0x3fc00000, 0x3fc00000), (1.5, 1.5) (0x3fc00000, 0x3fc00000)} #d getf
    $lx0n0c0b0m0 8
DEBUG-MRx(n0c0b0m0,2):{(1.5, 1.5) (0x3fc00000, 0x3fc00000), (1.5, 1.5) (0x3fc00000, 0x3fc00000
    ), (1.5, 1.5) (0x3fc00000, 0x3fc00000), (1.5, 1.5) (0x3fc00000, 0x3fc00000)} #d getf
    $lx0n0c0b0m0 8
DEBUG-MRx(n0c0b0m0,3):{(1.5, 1.5) (0x3fc00000, 0x3fc00000), (1.5, 1.5) (0x3fc00000, 0x3fc00000
    ), (1.5, 1.5) (0x3fc00000, 0x3fc00000), (1.5, 1.5) (0x3fc00000, 0x3fc00000)} #d getf
    $lx0n0c0b0m0 8
DEBUG-MRx(n0c0b0m0,4):{(0, 0) (0x00000000, 0x00000000), (0, 0) (0x00000000, 0x00000000), (0, 0)
     (0x00000000, 0x00000000), (0, 0) (0x00000000, 0x00000000)} #d getf $lx0n0c0b0m0 8
DEBUG-MRx(n0c0b0m0,5):{(0, 0) (0x00000000, 0x00000000), (0, 0) (0x00000000, 0x00000000), (0, 0)
     (0x00000000, 0x00000000), (0, 0) (0x00000000, 0x00000000)} #d getf $lx0n0c0b0m0 8
DEBUG-MRx(n0c0b0m0,6):{(0, 0) (0x00000000, 0x00000000), (0, 0) (0x00000000, 0x00000000), (0, 0)
     (0x00000000, 0x00000000), (0, 0) (0x00000000, 0x00000000)} #d getf $lx0n0c0b0m0 8
DEBUG-MRx(n0c0b0m0,7):{(0, 0) (0x00000000, 0x00000000), (0, 0) (0x00000000, 0x00000000), (0, 0)
     (0x00000000, 0x00000000), (0, 0) (0x00000000, 0x00000000)} #d getf $lx0n0c0b0m0 8
```

Example 4

```
d set $lm0n0c0b0m0 1 3FF0000000000000 # 1.0
d set $lm2n0c0b0m0 1 4000000000000000 # 2.0
d set $lm4n0c0b0m0 1 4008000000000000 # 3.0
d set $lm6n0c0b0m0 1 4010000000000000 # 4.0
```

```
dbfn $lm0v $nowrite
dmwrite $aluf $lx0

d getbd $lx0n0c0b0m0 4
```

Reads the double-precision block-floating-point values written to the matrix register. Refer to Section 3.4.4 for the `d set` statement.

The output is as follows. Although the bit sequence is different from the input, it can be confirmed that the value interpreted as a double-precision block-floating-point matches the value as a normal double-precision floating-point number before block-floating-point conversion[*1].

```
DEBUG-MRx(n0c0b0m0,0):{(1) (0x3ff8000000000000), (1) (0x3ff8000000000000), (1) (0
    x3ff8000000000000), (1) (0x3ff8000000000000)} #d getbd $lx0n0c0b0m0 4
DEBUG-MRx(n0c0b0m0,1):{(2) (0x4008000000000000), (2) (0x4008000000000000), (2) (0
    x4008000000000000), (2) (0x4008000000000000)} #d getbd $lx0n0c0b0m0 4
DEBUG-MRx(n0c0b0m0,2):{(3) (0x400c000000000000), (3) (0x400c000000000000), (3) (0
    x400c000000000000), (3) (0x400c000000000000)} #d getbd $lx0n0c0b0m0 4
DEBUG-MRx(n0c0b0m0,3):{(4) (0x4018000000000000), (4) (0x4018000000000000), (4) (0
    x4018000000000000), (4) (0x4018000000000000)} #d getbd $lx0n0c0b0m0 4
```

Example 5

```
imm i"0" $lr0
imm i"1" $lr2
imm i"2" $lr4
imm i"3" $lr6
nop
isub $subpeid $lr0v $omr1
d get $omr1n0c0b0m0 1
```

Refer to Section 3.6.12.3 for the `imm` instruction, Section 3.6.12.1 for the mask flags generated by the `isub` instruction, and Section 3.6.1.20 for `$subpeid`. The `isub` instruction subtracts the cycle number from its own PE number. Note that the PE number and cycle number are both 0-origin, and that the mask flag generated by the `isub` instruction is 1 when the result word integer is non-negative, so the flag becomes 1 up to the *i*-th cycle for the *i*-th PE. The output is as follows. Since there is no PE number specified in the element number specification `n0c0b0m0`, the values of all PEs are output. In this assembly sequence, the same value is written in the word direction, so when the flag is 1, `0b1111`, i.e., 15, is output.

```
DEBUG-OMR(n0c0b0m0p0,1):Mask{15} #d getf $omr1n0c0b0m0 1
DEBUG-OMR(n0c0b0m0p0,1):Mask{0} #d getf $omr1n0c0b0m0 1
DEBUG-OMR(n0c0b0m0p0,1):Mask{0} #d getf $omr1n0c0b0m0 1
DEBUG-OMR(n0c0b0m0p0,1):Mask{0} #d getf $omr1n0c0b0m0 1
DEBUG-OMR(n0c0b0m0p1,1):Mask{15} #d getf $omr1n0c0b0m0 1
DEBUG-OMR(n0c0b0m0p1,1):Mask{15} #d getf $omr1n0c0b0m0 1
DEBUG-OMR(n0c0b0m0p1,1):Mask{0} #d getf $omr1n0c0b0m0 1
DEBUG-OMR(n0c0b0m0p1,1):Mask{0} #d getf $omr1n0c0b0m0 1
DEBUG-OMR(n0c0b0m0p2,1):Mask{15} #d getf $omr1n0c0b0m0 1
DEBUG-OMR(n0c0b0m0p2,1):Mask{15} #d getf $omr1n0c0b0m0 1
DEBUG-OMR(n0c0b0m0p2,1):Mask{15} #d getf $omr1n0c0b0m0 1
DEBUG-OMR(n0c0b0m0p2,1):Mask{0} #d getf $omr1n0c0b0m0 1
DEBUG-OMR(n0c0b0m0p3,1):Mask{15} #d getf $omr1n0c0b0m0 1
DEBUG-OMR(n0c0b0m0p3,1):Mask{15} #d getf $omr1n0c0b0m0 1
DEBUG-OMR(n0c0b0m0p3,1):Mask{15} #d getf $omr1n0c0b0m0 1
DEBUG-OMR(n0c0b0m0p3,1):Mask{15} #d getf $omr1n0c0b0m0 1
```

Example 6

[*1] Note that this exact match with the pre-conversion value is a special case in block-floating-point conversion

```
d set $lm0n0c0b0m0p0 1 h0000_1111_1111_0000
d set $lm2n0c0b0m0p0 1 h0000_0000_1111_1111
d set $lm4n0c0b0m0p0 1 h1111_0000_0000_0000
d set $lm6n0c0b0m0p0 1 h0000_0000_0000_0000

spassa $lm0v $omr1
lpassa $lm0v $omr2
d get $omr1n0c0b0m0p0 1
d get $omr2n0c0b0m0p0 1
d get $omr1n0c0b0m0p0 2
```

The output is as follows. To determine which of the three `d get` statements the output came from, check the part after `#`. Blank lines have been added for readability.

The mask flags output by `passa` are 1 if all bits are 0 for the specified operation precision, as described in Section 3.6.12.1. For `spassa`, i.e., the half-precision `passa`, the flag is set if all bits are 0 for each half word, so the flag value for `h0000_1111_1111_0000` is 9 when converted to an integer value. For `lpassa`, i.e., the double-precision `passa`, the flag is set if all bits are 0 for each long word except for `h0000_0000_0000_0000`, so all flag values for values other than this are 15 when converted to an integer value.

```
DEBUG-OMR(n0c0b0m0p0,1):Mask{9} #d get $omr1n0c0b0m0p0 1
DEBUG-OMR(n0c0b0m0p0,1):Mask{12} #d get $omr1n0c0b0m0p0 1
DEBUG-OMR(n0c0b0m0p0,1):Mask{7} #d get $omr1n0c0b0m0p0 1
DEBUG-OMR(n0c0b0m0p0,1):Mask{15} #d get $omr1n0c0b0m0p0 1

DEBUG-OMR(n0c0b0m0p0,2):Mask{0} #d get $omr2n0c0b0m0p0 1
DEBUG-OMR(n0c0b0m0p0,2):Mask{0} #d get $omr2n0c0b0m0p0 1
DEBUG-OMR(n0c0b0m0p0,2):Mask{0} #d get $omr2n0c0b0m0p0 1
DEBUG-OMR(n0c0b0m0p0,2):Mask{15} #d get $omr2n0c0b0m0p0 1

DEBUG-OMR(n0c0b0m0p0,1):Mask{9} #d get $omr1n0c0b0m0p0 2
DEBUG-OMR(n0c0b0m0p0,2):Mask{0} #d get $omr1n0c0b0m0p0 2
DEBUG-OMR(n0c0b0m0p0,1):Mask{12} #d get $omr1n0c0b0m0p0 2
DEBUG-OMR(n0c0b0m0p0,2):Mask{0} #d get $omr1n0c0b0m0p0 2
DEBUG-OMR(n0c0b0m0p0,1):Mask{7} #d get $omr1n0c0b0m0p0 2
DEBUG-OMR(n0c0b0m0p0,2):Mask{0} #d get $omr1n0c0b0m0p0 2
DEBUG-OMR(n0c0b0m0p0,1):Mask{15} #d get $omr1n0c0b0m0p0 2
DEBUG-OMR(n0c0b0m0p0,2):Mask{15} #d get $omr1n0c0b0m0p0 2
```

3.4.4 d set Statement

The `d set` (Debug Set) statement is a control statement for writing the specified value to L2BM, L1BM, GRF0/1, LM0/1, and the T-register during emulator execution. It cannot be executed on the actual machine.

There is a possibility that the syntax and effects of the `d set` statement may change in future emulator updates.

Syntax

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

The syntax from `<memory>` to `<num_of_words>` is the same as that of the `d get` statement (Section 3.4.3). However, PDM and DRAM cannot be specified for `<memory>`. Unlike the `d get` statement, there is no specification for `<dtype>`.

`<payload>` is written by arranging the contents of the data to be written in long words, each long word represented by 16 hexadecimal digits. There are several syntax sugars for representing long-word data. Details on the required number of long words and syntax sugars will be described later.

Effects

Writes the specified data to the specified memory element and address.

The number of long words to be written in `<payload>` is the product of the number of words `num_of_words` and the payload word length determined by `<memory>`. An error occurs if the actual number of long words written is different. The possible memory elements and word lengths for access, as well as the corresponding payload word lengths, are listed in Table 3.2. As shown here, when the access word length is single word, the payload word length is longer than the access word length. In this case, the single word on the LSB side is ignored for each long word, and the single word on the MSB side is written.

When writing two or more long words, the addressing is big-endian. In other words, the smaller address is at the beginning of the payload.

Table 3.3 Combinations of memory elements and word lengths that can be written by d set statements

`<memory>`	Memory unit	Word length	Payload length
`$lc`	L2BM	Long word	1
`$lb`	L1BM	Long word	1
`$llb`	L1BM	Double long word	2
`$r, $s, $m, $n`	GRF0/GRF1/LM0/LM1	Short word	1
`$lr, $ls, $lm, $ln`	GRF0/GRF1/LM0/LM1	Long word	1
`$llr, $lls, $llm, $lln`	GRF0/GRF1/LM0/LM1	Double long word	2
`$t, $lt`	T-register	Long word	1
`$llt`	T-register	Double long word	2

Each long word in `<payload>` can be described using one of four notations: **16-digit hexadecimal integer**, **hexadecimal long-word integer**, **hexadecimal single-word integer**, and **hexadecimal half-word integer**. Table 3.4 explains each notation and provides examples. While hexadecimal long-word, single-word, and half-word integers can be mixed in a single line, a 16-digit hexadecimal integer cannot be mixed with other notations.

`<num_of_words>` specifies the number of words to be written from `<addr>` in decimal, using the access word length specified by `<memory>` as the unit. Note that the range of access to the T-register is the same as for the `d get` statement.

Table 3.4 Notations for long-word data written by d set statements. The examples are the same for all notations. In the examples of hexadecimal long-word, single-word, and half-word integers, the leading 0 is unnecessary because they are not fixed-length. Also, signed integers are not allowed in any case.

Notation name	Notation	Example
16-digit hex. integer	Fixed-length 16-digit hex. number	`0123456789abcdef`
Hex. long-word integer	1 max 16-digit hex. number after `l`	`l123456789abcdef`
Hex. single-word integer	2 max 8-digit hex. numbers joined by `_` after `s`	`s1234567_89abcdef`
Hex. half-word integer	4 max 4-digit hex. numbers joined by `_` after `h`	`h123_4567_89ab_cdef`

Errors

- An error occurs if there is a part of `<payload>` that does not match any of the notations shown in Table 3.4 when interpreted from the beginning
- An error occurs if the number of long words in the payload determined by `<memory>` and `num_of_words` does not match the actual length of `<payload>`
- An error occurs if a number in `<payload>` exceeds the fixed or maximum number of digits in that notation

Example 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
```

An example of hexadecimal long-word integer notation, hexadecimal half-word integer notation, and a mix of the two. The output is as follows.

```
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
```

Example 2

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

An example of hexadecimal single-word integer notation. This also serves as an example of the relationship between the payload position and the address. The output is as follows.

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

In the first line, `$lr0` writes two long words, with `1_2` in the first long word and `3_4` in the second long word. In the second line, `$lr2`, i.e., the address one long word ahead of `$lr0`, is read, so the result is a value with the words 3 and 4.

Example 3

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

An example where the payload word length is longer than the access word length, and the LSB side of each long word is discarded. The output is as follows. Since the access word length is a single word, only two single words = one long word are written. Therefore, the second long word is 0.

```
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
```

Example 4

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

An example of writing to the T-register using 16-digit hexadecimal integer notation. In the first line, the MSB side of one long word of the 2-long-word entry for one cycle is written, and in the third line, the entire 2-long-word entry for two cycles is written (although the third line is displayed with a line break, the `<payload>` is actually written in a single line with 64 hexadecimal digits). Reading by the `d get` statement is the same in the second and fourth lines, reading the entire 2-long-word entry × four cycles.

```
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
    x5555666677778888)} #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:0xDDDDEEEEFFFF0000
    )} #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 Instruction Statements

This section describes the syntax and effects of MV instructions.

A data transfer (MV) instruction statement is a unit that is translated into an MV instruction that is executed once every four steps. Unlike PE instructions, MV instructions do not have a format that separates expressions with semicolons, and only a single statement terminated by a newline is allowed.

3.5.1 Overview of Syntax

The syntax of MV instruction statements is complex, so we first provide an overview of the syntax and effects of each part using several examples.

Example 1

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

`mvp/n64i01` is the opcode, `$p0@0` is the source operand, and `$lc0@2.1` is the destination operand.

The opcode is divided into the basic mode part and the parameter part by the `/`.

Regarding the basic mode part of the opcode, `mv` is a fixed string that indicates that the instruction is an MV instruction, and `p` indicates that the basic mode is individual transfer.

Regarding the parameter part of the opcode, `n64` indicates that the number of transfer words is 64 long words, and `i01` indicates that the tag number is `0x01`.

Example 2

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

Unlike Example 1, the tag is not specified.

Regarding the tag, it can be omitted in the syntax because the completion timing of MV instructions may be determined without waiting. For example, the following is valid assembly.

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

Example 3

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

The `r` following `mv` in the basic mode part indicates that this instruction is a reduction transfer instruction. In reduction transfer, precision specification is required, and the content of the reduction operation is determined by the subsequent operation specification. In this case, it performs double-precision floating-point addition, indicated by `d` and `fadd`, respectively.

3.5.2 Operands

Since the possible MV instructions differ depending on the combination of memory types (PDM, DRAM, L2BM), we first show the syntax of the operands, and then show the syntax and effects of possible transfers for each combination of memory types.

For all memory types, the address wraps around when it exceeds the memory size.

3.5.2.1 p - PDM

The syntax for operands of PDM is as follows.

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

<addr> is a natural number specifying the address of a long word inside the PDM. The size of the PDM is 4 MiB per group, so the address wraps around when it reaches 512 Ki. Also, <addr> cannot be greater than 512 Ki.

<group> is a value from 0 to 3.

If <group> is not specified, $p<addr> accesses the same address in all groups' PDM.

Example

Reads 128 long words from the address 64 of group 1's PDM and writes them starting address 32 in group 2's DRAM.

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

3.5.2.2 d - DRAM

The syntax for operands of DRAM is as follows.

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

The first two are direct access by address. The latter two starting with $di enable DRAM indirection.

<addr> is a natural number specifying the address of a long word inside the DRAM. The size of the DRAM is 4 GiB per group, so the address wraps around when it reaches 512 Mi.

<group> is a group number from 0 to 3.

If <group> is not specified, $d<addr> accesses the same address in all groups' DRAM.

<dar_addr> is the start address for reading the DAR (DRAM Address Register) required for DRAM indirection. DRAM indirection is described in detail in Section 3.5.6.

Example

Reads 128 long words from the address 32 of group 2's DRAM and write them starting address 64 in group 1's PDM.

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

3.5.2.3 c - L2BM in MV Instructions

The syntax for operands of L2BM is as follows.

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

<addr> is a natural number specifying the address of a long word inside the L2BM. The size of the L2BM is 32 Ki long words, so the address wraps around when it reaches 32 Ki. Also, <addr> cannot be greater than 32 Ki.

<group> and <l2b> after @ are group numbers from 0 to 3 and L2B numbers of 0 or 1, respectively. If <group> is not specified, it accesses all groups, and if <l2b> is not specified, it accesses both L2BMs in the group.

Example

In parallel for all groups, read 64 long words from the address 32 of the group's DRAM and write them starting address 0 in the group's 1st L2BM.

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

3.5.3 Transfer Word Count Specification and Unit Operation

The transfer word count must be specified for MV instructions. In the syntax, n<size> is added to the parameter part of the opcode. Here, <size> is the number of long words counted in L2BM for transfers involving L2BM, and the number of long words counted in PDM for transfers between PDM and DRAM.

For any MV instruction, except for when DRAM indirection is used or address wraparound, it accesses a contiguous region in both the source and destination.

Except when DRAM indirection is used, increasing the transfer word count of an MV instruction results in the same as the original instruction, plus the remaining words transferred starting from the address after the region accessed by the instruction. That is, the following two examples are equivalent.

Example 1

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

Example 2

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

The transfer word count is determined by the basic mode, and only multiples of this minimum value can be specified. The operation at this minimum transfer word count is called the **unit operation**.

The address alignment of the operands is also determined by the unit operation. That is, if the unit operation accesses 64 words in the read or write operand, the address of that operand must be a multiple of 64 words.

3.5.4 Tag Specification

Tags can be specified for MV instructions to be used for waiting with the wait instruction. To specify a tag, add a tag (3.2.3 section) such as i01 to the parameter part of the opcode.

3.5.5 Reduction Operation Specification

In reduction transfers, the precision and type of operation are specified after mvr in the basic mode part. The specific syntax is as follows. d, f, h are double, single, and half precision floating-point, and l, i, s are double, single, and half precision integer, respectively. RRN in rrn_opcode is an abbreviation for Result Reduction Network.

```
  <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
```

The details of the fadd/max/min operations are described in Chapter 4.

The `rrn_opcode` is also used in reduction instructions below L2B.

3.5.6 DRAM Indirection

Any MV instruction involving DRAM can enable DRAM indirection mode.

In DRAM indirection mode, the address values written in advance to the **DAR (DRAM Address Register)** by the two L2BM instructions `l2bmdars`/`l2bmdarw` (Section 3.6.7.10) are used. The DAR exists for each group. Therefore, it is possible to access different DRAM addresses for each group. The DAR consists of 1024 entries, each containing a 32-bit address word. The unit of the address word is 16 long words*2.

The unit of indirection is 16 long words. That is, regardless of the basic mode of the MV instruction, the next address word is used every 16 long words accessed in DRAM.

In MV instructions with DRAM indirection mode enabled, two parameters are specified: the DAR read start address M and the number of consecutive DAR entries used N. In this case, for the number of 16-long-word accesses i, the final value of the DRAM address in 16 long words is $\mathrm{DAR}[(M+i/N)\%1024]+i\%N$. That is, for N times, the DRAM is accessed continuously using the value of a certain entry in the DAR as an offset, and then the value of the next entry in the DAR is used as a new offset, and so on.

Until the cycle immediately before the data written to the DAR by the `l2bmdarw` instruction becomes valid, the previous data can be read from that DAR entry.

M is specified in the DRAM operand (Section 3.5.2.2). N is specified in the option part of the MV instruction in the form `nd<N>`. If `nd<N>` is omitted, N is infinite, that is, the address becomes $\mathrm{DAR}[M]+i$.

In the explanation of the basic modes of MV instructions described in Section 3.5.8, the description for DRAM indirection mode is omitted.

Example

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

This is the DRAM indirection mode version of the PDM → DRAM Individual Transfer Instruction described in Section 3.5.8.2. The effect is as follows.

```
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 Priority Specification

Any MV instruction other than `mvnop` can set a **priority** from 0 to 3.

Details are left to the chip specification, but this means that if multiple accesses originating from multiple MV instructions are executable in each of the PDM, DRAM, and L2BM, the access with the higher priority value is executed first.

This is an option for optimizing the execution time of MV instructions, and if the assembly sequence is correct with appropriate waits, the priority setting has no effect on the calculation results.

The default priority is 0.

In the explanation of the basic modes of MV instructions described in Section 3.5.8, the description of priority is omitted.

Example

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

*2 It does not mean that DRAM with a capacity of 16 long words × 32 bits is implemented. Bits corresponding to unimplemented areas are ignored.

Issues an MV instruction with priority 3.

3.5.8 Basic Mode for MV Instructions

Here, we explain the basic modes of MV instructions, including syntax and effects. The throughput listed is the value when issued alone, that is, the value when not considering the influence of other MV instructions.

3.5.8.1 mvnop Instruction

`mvnop` is an MV instruction statement that does nothing. There are no parameters or operands. It is not necessary for users to write directly, and it may appear when disassembling packed instructions.

Syntax

```
mvnop
```

Effects
Does nothing.

3.5.8.2 PDM → DRAM Individual Transfer Instruction

Copies data from the specified group's PDM to the specified group's DRAM.

The unit operation is 64 long words.

The source and destination groups can be the same or different.

The throughput is 16 long words per cycle within the group and 8 long words per cycle between groups.

Syntax

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

Effects

```
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]
```

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

Copies 64 long words from group 0's PDM to group 1's DRAM.

3.5.8.3 DRAM → PDM Individual Transfer Instruction

Copies data from the specified group's DRAM to the specified group's PDM.

The unit operation is 64 long words.

The source and destination groups can be the same or different.

The throughput is 8 long words per cycle within the group and 4 long words per cycle between groups.

Syntax

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

Effects

```
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]
```

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Examples

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

Copies 64 long words from group 1's DRAM to group 0's PDM.

3.5.8.4 PDM → L2BM Individual Transfer Instruction

Copies data from the specified group's PDM to the specified group and L2B number's L2BM.

The unit operation is 64 long words.

The source and destination groups can be the same or different.

The throughput is 16 long words per cycle within the group and 8 long words per cycle between groups.

If the same address is used for all groups for intra-group transfers, it is advantageous in terms of latency to use the parallel version described in Section 3.5.8.9.

Syntax

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

Effects

```
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
```

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

Copies 64 long words from group 0's PDM to group 2's 1st L2BM.

3.5.8.5 L2BM → PDM Individual Transfer Instruction

Copies data from the specified group and L2B number's L2BM to the specified group's PDM.

The unit operation is 64 long words.

The source and destination groups can be the same or different.

The throughput is 16 long words per cycle within the group and 8 long words per cycle between groups.

If the same address is used for all groups for intra-group transfers, it is advantageous in terms of latency to use the parallel version described in Section 3.5.8.10.

Syntax

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

Effects

```
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
```

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

Copies 64 long words from group 2's 1st L2BM to group 0's PDM.

3.5.8.6 DRAM → L2BM Individual Transfer Instruction

Copies data from the specified group's DRAM to the specified group and L2B number's L2BM.

The unit operation is 64 long words.

The throughput is 16 long words per cycle within the group and 8 long words per cycle between groups.

If the same address is used for all groups for intra-group transfers, it is advantageous in terms of latency to use the parallel version described in Section 3.5.8.11.

Syntax

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

Effects

```
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
```

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

Copies 64 long words from group 0's DRAM to group 2's 1st L2BM.

3.5.8.7 L2BM → DRAM Individual Transfer Instruction

Copies data from the specified group and L2B number's L2BM to the specified group's DRAM.

The unit operation is 64 long words.

The throughput is 16 long words per cycle within the group and 8 long words per cycle between groups.

If the same address is used for all groups for intra-group transfers, it is advantageous in terms of latency to use the parallel version described in Section 3.5.8.12.

Syntax

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

Effects

```
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
```

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

Copies 64 long words from group 2's 1st L2BM to group 0's DRAM.

3.5.8.8 PDM → PDM Individual Transfer Instruction

Copies data from the specified group's PDM to the specified different group's PDM.

The unit operation is 64 long words.

The throughput is 8 long words per cycle.

Syntax

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

Effects

```
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]
```

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

Copies 64 long words from group 0's PDM to group 1's PDM.

3.5.8.9 PDM → L2BM Parallel Transfer Instruction

Copies data from the PDM to the specified L2B number's L2BM in the same group for all groups.

The unit operation is 64 long words.

The throughput is 16 long words per cycle per group.

Syntax

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

Effects

```
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
```

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

Copies 64 long words from the PDM to the 1st L2BM for all groups.

3.5.8.10 L2BM → PDM Parallel Transfer Instruction

Copies data from the specified L2B number's L2BM to the PDM in the same group for all groups.

The unit operation is 64 long words.

The throughput is 16 long words per cycle per group.

Syntax

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

Effects

```
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
```

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

Copies 64 long words from the 1st L2BM to the PDM for all groups.

3.5.8.11 DRAM → L2BM Parallel Transfer Instruction

Copies data from the DRAM to the specified L2B number's L2BM in the same group for all groups.

The unit operation is 64 long words.

The throughput is 16 long words per cycle per group.

Syntax

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

Effects

```
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
```

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

Copies 64 long words from the DRAM to the 1st L2BM for all groups.

3.5.8.12 L2BM → DRAM Parallel Transfer Instruction

Copies data from the specified L2B number's L2BM to the DRAM in the same group for all groups.

The unit operation is 64 long words.

The throughput is 16 long words per cycle per group.

Syntax

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

Effects

```
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
```

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

Copies 64 long words from the 1st L2BM to the DRAM for all groups.

3.5.8.13 DRAM → L2BM Intra-Group Broadcast Instruction

Copies data from the DRAM to two L2BMs in the same group for all groups.

The unit operation is 64 long words.

The throughput is 16 long words per cycle.

Syntax

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

Effects

```
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
```

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

Broadcasts 64 long words from the DRAM to both L2BMs for all groups.

3.5.8.14 L2BM → DRAM Intra-Group Broadcast Instruction

Reads from two L2BMs in the same group, reduces in the L2B direction, and writes to the DRAM in the same group for all groups.

The unit operation is 64 long words.

The throughput is 16 long words per cycle.

Syntax

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

For the reduction operation specification `<op>`, refer to Section 3.5.5.

Effects

```
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
```

Note: The reduction is not actually performed internally in this manner.

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

For all groups, reads 64 long words from two L2BMs, performs double-precision floating-point addition, and writes the resulting 64 long words to the DRAM.

3.5.8.15 L2BM → PDM Intra-Group Reduction Instruction

Reads the same size from two L2BMs in the specified group, reduces in the L2B direction, and writes to the PDM in the same group.

The unit operation is 64 long words.

The throughput is 16 long words per cycle.

Note that there is no corresponding PDM → L2BM intra-group broadcast instruction for this instruction.

Syntax

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

For the reduction operation specification <op>, refer to Section 3.5.5.

Effects

```
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
```

Note: The reduction is not actually performed internally in this manner.

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

For group 1, reads 64 long words from two L2BMs, performs double-precision floating-point addition, and writes the resulting 64 long words to PDM 1.

3.5.8.16 DRAM → L2BM Inter-Group Broadcast Instruction

This instruction is complex, so first, the unit operation of writing 64 long words to the L2BM side is explained. Reads 32 long words from the DRAM of each group, broadcasts 64 long words arranged in group order to the 0th L2BM of all groups, and broadcasts 64 long words arranged in the same way to the 1st L2BM of all groups.

If the size is not a unit operation, the unit operation is repeated.

The throughput is 8 long words per cycle on the DRAM side and 16 long words per cycle on the L2BM side.

Syntax

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

Effects

```
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]
```

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

The unit operation described at the beginning of this section.

3.5.8.17 L2BM → DRAM Inter-Group Gather Reduction Instruction

This instruction is complex, so first, the unit operation of reading 64 long words from the L2BM side is explained. Reads 64 long words from all L2BMs and reduces in the group direction. Divides the reduced 64 long words of the 0th L2B of all groups into 16 long words each and writes them to each DRAM. Similarly, divides the reduced 64 long words of the 1st L2B of all groups into 16 long words each and writes them to each DRAM. This results in 32 long words being written to each DRAM.

If the size is not a unit operation, the unit operation is repeated.

The throughput is 16 long words per cycle on the L2BM side and 8 long words per cycle on the DRAM side.

Syntax

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

For the reduction operation specification `<op>`, refer to Section 3.5.5.

Effects

```
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]
```

Note: The reduction is not actually performed internally in this manner.

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

The unit operation described at the beginning of this section. Double-precision floating-point addition is used for the reduction.

3.5.8.18 PDM → L2BM Inter-Group Broadcast Instruction

Broadcasts data from the PDM of the specified group to all L2BMs for all groups.

The unit operation is 64 long words.

The throughput is 8 long words per cycle.

Syntax

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

Effects

```
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
```

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

Broadcasts 64 long words from the PDM of group 0 to all L2BMs.

3.5.8.19 L2BM → PDM Inter-Group Reduction Instruction

Reads the same size from all L2BMs, reduces in the L2B direction, and writes to the PDM of the specified group.

The unit operation is 64 long words.

The throughput is 8 long words per cycle.

Syntax

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

For the reduction operation specification `<op>`, refer to Section 3.5.5.

Effects

```
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
```

Note: The reduction is not actually performed internally in this manner.

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

Reads 64 long words from all eight L2BMs, reduces in the L2B number direction by double-precision floating-point addition, and writes to the PDM of group 0.

3.5.8.20 DRAM → L2BM Inter-Group Broadcast Instruction

Reads the same size from the DRAM of all groups, joins, and broadcasts to all L2BMs.

The unit operation is 16 long words on the DRAM side and 64 long words on the L2BM side.

The throughput is 32 long words per cycle on the L2BM side.

Syntax

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

Effects

```
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]
```

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

Reads 16 long words from the DRAM of all groups, joins into 64 long words, and broadcasts to all L2BMs.

3.5.8.21 L2BM → DRAM Inter-Group Reduction Instruction

Reads the same size from all L2BMs, reduces in the L2B direction, and distributes equally to four DRAMs.

The unit operation is 64 long words on the L2BM side and 16 long words on the DRAM side.

The throughput is 32 long words per cycle on the L2BM side.

Syntax

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

For the reduction operation specification `<op>`, refer to Section 3.5.5.

Effects

```
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]
```

Note: The reduction is not actually performed internally in this manner.

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

Reads 64 long words from all eight L2BMs, reduces in the L2B direction, divides the resulting 64 long words into 16 long words for each group, and writes to the DRAM.

3.5.8.22 PDM → L2BM Scatter Instruction

Reads from the PDM of the specified group, divides into eight parts, and writes to the L2BM of each group.

The unit operation is 512 long words on the PDM side and 64 long words on the L2BM side. The division method is round-robin in the L2B number direction from 0 to 7 in 16 long word units. Refer to the effects part for details.

The throughput is 8 long words per cycle on the PDM side and 1 long word per cycle on the L2BM side.

Syntax

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

Effects

```
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]
```

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

Reads 512 long words from the PDM of group 0 and writes 64 long words to each L2BM.

3.5.8.23 L2BM → PDM Gather Instruction

Reads from all L2Bs, joins, and writes to the PDM of the specified group.

The unit operation is 64 long words on the L2BM side and 512 long words on the PDM side. The gathering method is round-robin in the L2B number direction from 0 to 7 in 16 long word units. Refer to the effects part for details.

The throughput is 1 long word per cycle on the L2BM side and 8 long words per cycle on the PDM side.

Syntax

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

Effects

```
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]
```

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

Reads 64 long words from each L2BM, joins into 512 long words, and writes to group 0's PDM.

3.5.8.24 PDM → DRAM Scatter Instruction

Reads from the PDM of the specified group, divides into four parts, and writes to the DRAM of each group.

The unit operation is 64 long words on the PDM side and 16 long words on the DRAM side.

The throughput is 8 long words per cycle on the PDM side and 2 long words per cycle on the DRAM side.

Syntax

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

Effects

```
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]
```

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

Reads 64 long words from group 0's PDM and writes 16 long words to each DRAM.

3.5.8.25 DRAM → PDM Gather Instruction

Reads from the DRAM of each group, joins, and writes to the PDM of the specified group.

The unit operation is 16 long words on the DRAM side and 64 long words on the PDM side.

The throughput is 2 long words per cycle on the DRAM side and 8 long words per cycle on the PDM side.

Syntax

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

Effects

```
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]
```

Errors

- An error occurs if `size` is not a multiple of the unit operation 64.

Example

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

Reads 16 long words from each DRAM, joins, and writes 64 long words to group 0's PDM.

3.5.9 Constraints between Multiple MV Instructions

In addition to the constraints on individual MV instructions described above, there are constraints that only become a problem when issuing multiple MV instructions at the same time.

The assembler checks these constraints and reports an error if the appropriate options are specified, so the details are described in a separate document.

3.6 PE Instruction Statements

A PE instruction statement is a unit that is translated into PE instructions executed in one step. A PE instruction statement consists of several PE instruction expressions separated by semicolons and terminated by a newline.

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

The number of input operands an expression has is determined by the opcode.

The number of output operands can be specified as any number as long as it meets other constraints. If multiple output operands are specified, all of them will be written to. Specifically, for MAU and ALU outputs, if both the mask register and other PE memory are specified as output operands, the mask register will receive the mask flags, while the other PE memory will receive the normal output values. An example is shown later in Section 3.7.3.

The `nop`, `noforward`, `l2bmdarw`, and `wait` instructions do not have input or output values, so they do not fit the above syntax.

By listing multiple expressions in one line separated by semicolons, they can be issued as a single instruction under the condition that the bit fields of the translation results do not overlap.

For all memory types, the address wraps around when it exceeds the memory size.

Below are some examples. The detailed syntax of opcodes and operands will be described later.

Example 1

`dvpassa` is a one-input operand instruction that copies data using the MAU. In the example below, it copies from LM0 (`$lm0v`) to LM1 (`$ln0v`). Note that the values are interpreted as double-precision floating-point numbers and may not be copied as bit sequences.

```
dvpassa $lm0v $ln0v
```

Example 2

The following is an example with two-output operands. It copies from LM0 to LM1 and GRF0 (`$lr0v`).

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

Example 3

`linc` is a one-input operand instruction that increments double-precision integers with an ALU. In the example below, in addition to the effects of Example 2, it increments the data read from the T-register (`$t`) and stores it in GRF1 (`$ls0v`).

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

3.6.1 Operands

3.6.1.1 c - L2BM in PE Instructions (Except for l2bmdars Instruction)

The syntax of the L2BM operand in PE instructions is as follows, except for the `l2bmdars` instruction (Section 3.6.7.10).

```
$lc<addr>
```

<addr> is the address in long words in the L2BM. The size of the L2BM is 32 Ki long words, so the address wraps around when it reaches 32 Ki. Also, <addr> cannot be specified as 32 Ki or more.

Unlike PE memories such as LM, L2BM cannot specify address incrementation, and it is automatically determined to access continuous regions without overlap depending on the contents of the L2BM instruction.

The SRAM ports that constitute L2BM are divided into the side accessed by MV instructions and the side accessed by L2BM instructions, and can be accessed in parallel. Each side is 1R/1W, so it is not possible to read from the transfer to L1BM and write to the transfer from L1BM at the same time.

3.6.1.2 c - L2BM in l2bmdars Instruction

When used as an input operand for the l2bmdars instruction (Section 3.6.7.10), the L2BM differs from the operands of other L2BM instructions and uses the following syntax.

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

A field for the L2B number is added to specify which L2B's data in the group is transferred to the DAR (Section 3.6.1.3).

3.6.1.3 dar - DRAM Address Register

DAR (DRAM Address Register) is an operand used only in the output of the l2bmdars instruction (Section 3.6.7.10). The syntax is as follows.

```
$dar<addr>
```

<addr> is the write start entry number of the DAR.

There is one DAR per group, each with 1024 entries. When the write entry number reaches 1024, it wraps around. Also, <addr> cannot be specified as 1024 or more.

3.6.1.4 b - L1BM in L2BM Instructions

The syntax of the L1BM operand in L2BM instructions is as follows.

```
$lb<addr>
```

<addr> is the address in long words in the L1BM. The size of the L1BM is 8 Ki long words, so the address wraps around when it reaches 8 Ki. Also, <addr> cannot be specified as 8 Ki or more.

Unlike PE memories such as LM, L1BM cannot specify address incrementation, and it is automatically determined to access continuous regions without overlap depending on the contents of the L2BM instruction. This is the same for L1BM in L1BM instructions, as described in Section 3.6.1.5.

The SRAM ports that constitute L1BM are divided into the side accessed by L2BM instructions and the side accessed by L1BM instructions, and can be accessed in parallel. Each side is 1R/1W, so it is not possible to read from the transfer to L2BM and write to the transfer from L2BM at the same time. However, the internal multicast instruction (Section 3.6.7.9) is realized using only the ports of the side accessed by L2BM instructions, as the reading L1BM and writing L1BM are separate.

3.6.1.5 b - L1BM in L1BM Instructions

The syntax of the L1BM operand in L1BM instructions is as follows.

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

(l|ll) is the access word length specification. l is for long words, and ll is for 2 long words. The latter specifies the address in long words in the L1BM with <addr> or specifies the turnaround register with i. The size of the L1BM is 8 Ki long words, so the address wraps around when it reaches 8 Ki if <addr> is specified. Also, <addr> cannot be specified as 8 Ki or more.

Due to the reasons described in Section 3.6.1.4, it is not possible to read from the transfer to the PE and write to the transfer from the PE at the same time. However, with the turnaround register described in Section 3.6.8.2, these simultaneous issuances are possible with restrictions.

3.6.1.6 m - LM0 (Excluding Base Address Register Writes)

The syntax of the LM0 operand is as follows, except for writes to the Base Address Register (BAR).

```
  $[(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
```

The first syntax is for the auto-stride mode, and the second syntax is for the flat mode (see Section 1.2 for both modes).

Writing to the base address register of LM0 is described in Section 3.6.1.7.

Additionally, input sign inversion (Section 3.6.9.22), precision extension (Section 3.6.9.23), precision shortening (Section 3.6.9.24), and output mask application (Section 3.6.2.1) are omitted here. Refer to each section noted in parentheses.

[(l|ll)] is the access word length specification. If empty, it is a single word; if l, it is a long word; if ll, it is 2 long words.

The latter part is divided into two patterns: normal addressing (the auto-stride mode <addr>[v[<adri>]][j<madpe>] or the flat mode <flat_addrs>[j<madpe]) and T-register indirection (t<addr>[v[<adri>]] or t<flat_addrs>). First, each part such as <addr> is described.

In the auto-stride mode, <addr> specifies the address offset in single words in LM0. This value is added to the final address values during the 1 instruction 4 cycles.

v specifies the address increment between cycles. If v is not attached, the increment width is 0. <adri> is the increment width in single words. If the value is omitted, it is set to 1 word in the access word length, which accesses continuous regions without overlap.

In the flat mode, <flat_addrs> directly specifies the addresses in single words for each cycle in the form of [<addr0>,<addr1>,<addr2>,<addr3>]. Here, the square brackets ([]) mean that this symbol is actually described, not an option.

j<madpe> enables MAB internal address modification. This increments the address by 1 word in the access word length only for PEs with numbers less than or equal to the number specified by madpe.

t enables T-register indirection. The value of the MSB side 1 word of the 2 long words read from the T-register in each cycle is added to the final address value in single words.

The part after t, <addr>[v[<adri>]] or <flat_addrs>, can be omitted. In that case, the address in these parts is considered 0 for all cycles. That is, the value read from the T-register in each cycle becomes the address value as it is.

T-register indirection and MAB internal address modification cannot be enabled simultaneously.

Additionally, the value read from the base address register is always implicitly added to the final address value in single words.

If the address value becomes 4096 single words or more, which is the size of LM0, it wraps around.

In summary, the final address in single words in the auto-stride mode is determined as follows.

$$(\text{BAR}+(\text{T[C] if TI else }0)+\text{ADDR}+\text{ADRI}\times\text{C}\times\text{WL}+(\text{WL if MAADJ and PE} \leq \text{MADPE else }0))\%4096$$

Here,

- BAR: The value read from the base address register
- C: Cycle number from 0 to 3
- T[C]: The MSB side 1 long word of the 2 long words read from the T-register in that cycle
- TI: True if T-register indirection is enabled
- ADDR: The value of <addr>
- ADRI: The value of <adri> converted to the access unit, i.e., <adri>/WL
- WL: 1, 2, and 4 for single-word access, long-word access, and 2-long-word access, respectively
- MAADJ: True if MAB internal address modification is enabled

- MADPE: The value of madpe
- PE: PE number from 0 to 3

In the flat mode, you can directly specify the addresses in single words for each cycle by reinterpreting the part ADDR + ADRI × C × WL.

The address must be aligned to the access word length in each cycle. Therefore, in the auto-stride mode, both addr and adri must be multiples of 2 for long-word access and 4 for 2-long-word access. These are checked by the assembler, but the values of the T-register in T-register indirection and the base address register are determined at runtime, so there may be misalignment. In that case, the remainder is truncated.

Example

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

Copy 4 long words from LM0 to LM1. Since the flat mode is used in LM0, this can only be assembled in the flat mode. At that time, the address of LM1 is also replaced with an equivalent flat-mode specification. In other words, this is equivalent to the following.

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

3.6.1.7 m - LM0 (Base Address Register Write)

The syntax of the operand for writing to the base address register (BAR) of LM0 is as follows.

```
$[l]mb
```

The output mask application (Section 3.6.2.1) is omitted here. Refer to the section noted in parentheses.

This is an output-only operand.

If l is attached, it is for long-word access; otherwise, it is for single-word access.

When this operand is taken as the output destination of the MAU, the LSB side 12 bits of the MSB side 1 word are written to the base address register of LM0 according to the access word length.

3.6.1.8 n - LM1 (Excluding Base Address Register Writes)

The syntax of the LM1 operand is as follows, except for writes to the base address register (BAR).

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

The first syntax is for the auto-stride mode, and the second syntax is for the flat mode (see Section 1.2 for both modes).

Writing to the base address register of LM1 is described in Section 3.6.1.9.

Additionally, input sign inversion (Section 3.6.9.22), precision extension (Section 3.6.9.23), precision shortening (Section 3.6.9.24), and output mask application (Section 3.6.2.1) are omitted here. Refer to each section noted in parentheses.

The effects are the same as in the case of LM0 described in Section 3.6.1.6, except that T-register indirection cannot be used in LM1.

3.6.1.9 n - LM1 (Base Address Register Write)

The syntax of the operand for writing to the base address register (BAR) of LM1 is as follows.

```
$[l]nb
```

The output mask application (Section 3.6.2.1) is omitted here. Refer to the section noted in parentheses.

The effects are the same as those of LM0 (Section 3.6.1.7).

3.6.1.10 r - GRF0

The syntax of the GRF0 operand is as follows.

```
  $[(l|ll)]r<addr>[v[<adri>]] # auto-stride mode
| $[(l|ll)]r<flat_addrs> # flat mode
```

The first syntax is for the auto-stride mode, and the second syntax is for the flat mode (see Section 1.2 for both modes).

Input sign inversion (Section 3.6.9.22), precision extension (Section 3.6.9.23), precision shortening (Section 3.6.9.24), and output mask application (Section 3.6.2.1) are omitted here. Refer to each section noted in parentheses.

`[(l|ll)]` is the access word length specification. If empty, it is a single word; if `l`, it is a long word; if `ll`, it is 2 long words.

In the auto-stride mode, `<addr>` specifies the address in single words in GRF0. This value is added to the final address values during the 1 instruction 4 cycles.

`v` specifies the address increment between cycles. If `v` is not attached, the increment width is 0. `<adri>` is the increment width in single words. If the value is omitted, it is set to 1 word in the access word length, which accesses continuous regions without overlap.

In the flat mode, `<flat_addrs>` directly specifies the addresses for each cycle in the form of `[<addr0>,<addr1>,<addr2>,<addr3>]`. Here, the square brackets (`[]`) mean that this symbol is actually described, not an option.

The size of GRF0 is 256 long words, so it wraps around when the address reaches 512. Also, `<addr>` and others cannot be specified as 512 or more.

The address must be aligned to the access word length in each cycle. Therefore, in the auto-stride mode, both `addr` and `adri` must be multiples of `2` for long-word access and `4` for 2-long-word access.

3.6.1.11 s - GRF1

The syntax of the GRF1 operand is as follows.

```
  $[(l|ll)]s<addr>[v[<adri>]] # auto-stride mode
| $[(l|ll)]s<flat_addrs> # flat mode
```

The first syntax is for the auto-stride mode, and the second syntax is for the flat mode (see Section 1.2 for both modes).

Input sign inversion (Section 3.6.9.22), precision extension (Section 3.6.9.23), precision shortening (Section 3.6.9.24), and output mask application (Section 3.6.2.1) are omitted here. Refer to each section noted in parentheses.

The effects are the same as those of the GRF0 operand described in Section 3.6.1.10.

3.6.1.12 t - T-register

The syntax of the T-register operand is as follows.

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

Input sign inversion (Section 3.6.9.22), precision extension (Section 3.6.9.23), precision shortening (Section 3.6.9.24), and output mask application (Section 3.6.2.1) are omitted here. Refer to each section noted in parentheses.

`[(l|ll)]` can be described but is ignored. The T-register is always accessed in 2 long words.

There is no address specification for the T-register. It has a size of 2 long words x 4 cycles and automatically accesses the area corresponding to the current cycle.

3.6.1.13 omr - Mask Register Write

The syntax of the mask register operand is as follows.

```
$omr<addr>
```

This is an output-only operand.

The output mask application (Section 3.6.2.1) is omitted here. Refer to the section noted in parentheses.

Note that the mask can also be applied when writing to the mask register.

The mask register can only be specified as the output of a MAU or ALU instruction expression. The details of the generated mask value are explained in the sections of each instruction expression.

3.6.1.14 x, y - Matrix Register

The syntax of the matrix register operand is as follows.

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

(1) is the syntax for a matrix register write (Section 3.6.10) and a transposed read (Section 3.6.11), and (2) is the syntax for matrix-vector multiplication execution (Section 3.6.9).

`(l|ll)` specifies the access word length. `l` is for a long word, and `ll` is for 2 long words. The access word length can be set to 2 long words only when the opcode precision specification is half-precision in (1) (i.e., `hmwrite` or `hmread`), and in the case of half-precision transposed read (`hmread`), it must be set to 2 long words.

`(x|y)` specifies which of the two matrix register sides to access.

`<addr>` is the starting row number for the write and the starting column number for the transposed read. In (2), the entire side of the matrix register is used, so there is no address specification. Depending on `<addr>`, the address may exceed the matrix size after the first cycle. In that case, it wraps around.

Unlike PE memories such as LM, address incrementation cannot be specified, and it is incremented in each cycle according to the word length to access consecutive rows (columns) without overlap. If the row number (column number) exceeds the number of rows (columns) for each precision, it wraps around. The row number (column number) must be aligned to the access word length in each cycle. That is, in the case of 2-long-word access, `<addr>` must be a multiple of `2`.

Matrix-vector multiply-add can be executed from the step immediately after writing to the matrix register.

The values read from the matrix register in matrix-vector multiply-add must be block-floating-point values generated by the block-floating-point instruction (Section 3.6.12.12). However, there are no restrictions on the values read from the matrix register in transposed read, and the bit sequence is copied as is.

The precision of the matrix data stored in the matrix register can be double-precision, single-precision, pseudo-single-precision, or half-precision, and the number of rows in the matrix is logically 4, 8, 8, and 16 rows, respectively. On the other hand, the physical side of the matrix register consists of 16 rows. In the case of double-precision, the physical rows 0, 4, 8, and 12 correspond to the logical 4 rows. That is, in commands that access the matrix register in double-precision, such as `dmwrite` (Section 3.6.10.1), the physical rows 1, 2, 3, 5, ..., 15 are not accessed. Similarly, in the case of single-precision and pseudo-single-precision, the physical rows 0, 2, ..., 14 correspond to the logical 8 rows. This correspondence does not need to be considered as long as the precision of the write and read matches. Also, since there are few opportunities to intentionally write assembly code where the precision of the write and read does not match, it is rarely a problem in most cases.

3.6.1.15 mauf - MAU Operation Result Forwarding

The syntax of the operand for reading the data output by the MAU in the previous step, excluding `nop` and `noforward`, is as follows.

```
$mauf
```

More specifically, in the `i`th cycle, the data output by the MAU in the `i`th cycle of the previous step can be read.

The word length is always 2 long words, and the word length cannot be specified. That is, the value read from `$mauf` is equal to the data output by the MAU once written to `$llr0v` and then read from the

same operand after an appropriate number of steps.

`$mauf` can appear multiple times in the same step.

The value of `$mauf` is not updated in steps where there is a `nop` or `noforward` instruction. This is the same for `nop` instructions explicitly written and `nop` instructions automatically inserted by the instruction issue unit.

This is a read-only operand.

Example

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

The result of the single-precision matrix-vector product FMA is written to GRF0, and the result of adding them between MABs is written to L1BM in the next step. For example, the sum of the values written to `$lr0` is written to `$lb0` through `$lb3`.

3.6.1.16 aluf - ALU Operation Result Forwarding

The syntax of the operand for reading the data output by the ALU in the previous step, excluding `nop` and `noforward`, is as follows. The characteristics other than the data source are the same as those of the MAU operation result forwarding (Section 3.6.1.15).

```
$aluf
```

Example

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

The result of double-precision block-floating-point conversion is written to GRF0, and the result is written to the matrix register in the next step.

3.6.1.17 lbf - L1BM → PE Direction Transfer Forwarding

The syntax of the operand for reading the data transferred from L1BM to PEs in the previous step, excluding `nop` and `noforward`, is as follows.

```
$lbf
```

Example

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

The MAB broadcast from L1BM to GRF0 is performed, and the double of it is calculated in the next step and written to GRF1.

3.6.1.18 mreadf - Transposed Matrix Register Read Forwarding

The syntax of the operand for reading the data output by the transposed matrix register read in the previous step, excluding `nop` and `noforward`, is as follows.

This can only be used with the first input operand of the ALU.

The characteristics other than the data source and operand position constraints are the same as those of the MAU operation result forwarding (Section 3.6.1.15).

```
$mreadf
```

Errors

An error occurs if used not as the first input operand of the ALU. For example, `drelu $lr0v $mreadf $ls0v` causes an error.

Example

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

Writes the result of double-precision matrix register transposed read to GRF0 and writes the value to GRF1 after block-floating-point conversion in the next step.

3.6.1.19 nowrite - Dummy Output for Forwarding

The existence of the forwarding path may result in an operation being performed in a step without writing the result to any memory. In that case, the following must be specified as the only output operand.

```
$nowrite
```

Errors

An error occurs if another output operand is specified with `$nowrite` in a single opcode.

Example

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

Reads the immediate value `1.0` generated by the ALU instruction from the forwarding path in the next step, adds it to the value read from LM0, and writes it to GRF0.

3.6.1.20 Constant Input Operand

A few constants can be used for the first input operand of the ALU. The list is shown in Table 3.5.

Table 3.5 List of constant input operands

Operand	Value
`$l2bid`	Own group number ×2+ L2B number (3 bits)
`$l1bid`	Own L1B number (3 bits)
`$mabid`	Own MAB number (4 bits)
`$peid`	Own MAB number ×4+ own PE number (6 bits)
`$subpeid`	Own PE number (2 bits)
`$msb1`	Value with only the MSB as 1 and the rest as 0

All constant values are input as 2 long words according to the precision specification of the ALU instruction.

Example

```
ipassa $msb1 $llm0
```

`$llm0` contains 2 long words with 4 words of `0x80000000` each.

3.6.2 Mask Register

A mask register is a special PE memory that holds flags, not ordinary data. When writing to the PE memory or outputting the result of an operation from the operator, it allows skipping writes or zero-clearing results at dynamically determined locations. This makes it possible to achieve pseudo-conditional branching.

There are 32 mask register entries, of which 15 are variable and the rest have fixed values that are read-only when applying masks.

One entry consists of 16 bits, divided into four cycles in the cycle direction and four bits in the word direction.

Masks can be applied to PE memories: LM0, LM0 base address registers, LM1, LM1 base address registers, GRF0, GRF1, T-register, and mask registers. When applying a write mask to PE memories other than the mask register, if the corresponding entry is 1, the write is performed, and if it is 0, the write is not performed. When applying a write mask to a mask register, a special operation is performed in which the logical AND of the flag value of the corresponding entry and the destination entry is written.

When applying a zero-flush mask, if the corresponding entry is 1, nothing happens during the output of operation results, and if it is 0, the output is set to all zeros.

Applying a zero-flush mask to MAU or ALU does not affect the mask flags that they output.

The term "cycle direction" means that different entries are accessed depending on which cycle of a step it is, both when writing to and reading from the mask register.

The meaning of "word direction" varies depending on the chosen **mask application word length** during mask application.

When the mask-application word length is a long word, the mask is applied to the 4 bits of the MSB side of 2 long words that pass through the PE internal data path in one cycle, treating them as 4 half words and associating 4-bit flags with them, without interfering with the LSB side long word. When the mask-application word length is 2 long words, the mask is applied to the same 2 long words, treating them as 4 words. The mask application word length has no effect on applying masks to mask register writes.

The addresses and the contents of the fixed-value entries are as follows [*3]:

- Address 0: All 1 (always writes when used as a write mask, never performs a zero-flush for operation units)
- Addresses 16 and later: The lower four bits of the address value become the mask flags for each cycle, in the order from upper bits to lower bits. The same value is used for the word direction.

Addresses 1 through 15 are variable entries. Only the ALU and MAU can write mask flags to variable entries. The definitions of output flag values are described in Section 3.6.12.1 for ALU and Section 3.6.9.26 for MAU. Written mask flags become available from the next step immediately.

Both a write mask and a zero-flush mask can be specified simultaneously. However, it is not possible to read different entries for each.

3.6.2.1 Write Mask Application

The following must be specified when applying a mask:

- Mask application word length
- Which PE memories to apply the mask to (GRF0, GRF1, T-register, LM0 and its base address register, LM1 and its base address register; multiple specifications are possible)
- The address of the entry to read from

There are two types of syntax for applying write masks: multi-row application, which specifies a single mask for multiple rows at once, and single-row application, which specifies a separate mask for each row individually

Syntax (multi-line application)

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

This syntax specifies the write mask settings for all subsequent lines. It is a control statement that changes the behavior of subsequent PE instruction expressions, and is not a PE instruction expressions itself.

`[l|ll]` specifies the mask application word length. `l` represents a long word and `ll` represents 2 long words. If omitted, it defaults to a long word.

[*3] By this definition, address 0 and 31 of fixed value entry hold the same contents

[r][s][t][m][n][k] enables the write mask for the corresponding memory elements in the order of GRF0, GRF1, T-register, LM0 and its base address register, LM1 and its base address register, and mask registers. If a memory element is not specified, its mask will be disabled. [r][s][t][m][n][k] can be specified in any order.

<addr> specifies the address of the entry in the mask register from 0 to 31.

At the start of assembly, the implicit setting is mask 0, which means that no write mask is applied.

Syntax (single-line application)

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

This syntax cancels the multi-line mask specification for this step only and applies the mask to the write operation for <dst>.

<dst> is any of the destination PE memory operands.

[ll]<mask-pattern> specifies a fixed value entry, while $[ll]imr<adr> specifies a variable entry.

ll specifies that the mask application word length is 2 long words.

<mask-pattern> specifies the flag values for each cycle in the fixed value entry as a sequence of 4 0 or 1 repeats, starting from the first cycle.

<adr> specifies the address of the variable entry as an integer between 1 and 15.

The suffixes [t|p] are used to prevent bugs caused by mismatched PE memory access word lengths and mask application word lengths. They do not change the behavior of the instruction. t is short for "truncate", and must be appended when the access word length of <dst> is not double-long word, but the mask application word length is double-long words. p is short for "pad", and must be appended when the access word length of <dst> is double-long words, but the mask application word length is not double-long words.

Errors

- Within the same step, if multiple single-row mask applications are specified with different mask application word lengths or entry addresses that do not match, it will result in an error.
- If the word length of the destination PE memory operand and the mask application word length do not match (i.e., one is 2 long words while the other is not), and neither t nor p is properly set, it will result in an error.
- If unnecessary t or p suffixes are present, it will also result in an error.

Example: multi-line application

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

Copy from LM0 to GRF0 in each two steps, but only update in the 4th cycle. That is, only $lr6,$lr14 will be updated. After that, return to the default state where no mask is applied.

Example: Single-line application fixed value entry

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

Copy from LM0 to GRF0, but only update in the 4th cycle for the first step, and only update in the 1st cycle for the next step. That is, only $lr6,$lr8 will be updated. After that, return to the default state where no mask is applied.

Example: Single-line application of variable entry

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

Read flags from the first variable entry and write the result of half-precision matrix-vector product operation to GRF0 with mask application. Here, since the basic operation of the half-precision MAU results in 2 long words, the mask application word length is also set to 2 long words.

Example: Mask application to a mask register write

```
sinc $peid $omr1/1100

d get $omr1n0c0b0m0p0 1

spassa $lm0v $ln0v/$imr1
```

Create a mask that writes only in the first 2 cycles and apply it to LM1 write. This is not a practical example, but for illustration purposes. The same result can be achieved with a single line: `spassa $lm0 v $ln0v/1100`. The flags generated by `sinc $peid` are always 1, but the logical AND operation with the write masking `/1100` results in the following output for the `d get` command:

```
DEBUG-OMR(n0c0b0m0p0,1):Mask{15} #d get $omr1n0c0b0m0p0 1
DEBUG-OMR(n0c0b0m0p0,1):Mask{15} #d get $omr1n0c0b0m0p0 1
DEBUG-OMR(n0c0b0m0p0,1):Mask{0} #d get $omr1n0c0b0m0p0 1
DEBUG-OMR(n0c0b0m0p0,1):Mask{0} #d get $omr1n0c0b0m0p0 1
```

3.6.2.2 Applying Zero-flush Mask

Data that can be written to PE memory (MAU output, matrix register transposed read, ALU output, L1BM transfer to PE) can have some values zeroed out using mask flags.

When applying a mask, use the mask flags written in advance to the mask register. If the mask flag is 0, the corresponding part will be zeroed before writing.

To apply a mask, specify the following:

- Mask application word length
- Which operation (MAU output, matrix register transposed read, ALU output, L1BM transfer to PE) to apply the mask to (multiple selections not allowed)
- The address of the entry to read from

Note a that applying a a zero-flush mask to MAU or ALU does not affect the mask flags they produce.

Syntax

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

`<opcode>` is an opcode for MAU operation, matrix register transposed read, ALU operation, or L1BM transfer to PE.

The syntax after the `/` is the same as for single-row application of a write mask.

Unlike write masks, there is no syntax for multi-row application.

Errors

- Within the same step, if multiple single-row mask applications are specified with different mask application word lengths or entry addresses that do not match, it will result in an error.
- Within the samea step, if a zero-flush masks are applied to multiple opcodes, it will result in an error.

Example

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

This code writes the result of half-precision matrix-vector product operation to GRF0, with all values in the first cycle set to zero. Here, since the basic operation of the half-precision MAU results in a double-long-word result, the mask application word length is also set to double-long word.

3.6.3 Avoiding hazards

There are cases where it is necessary to leave an adequate number of steps between instructions. The two reasons for this are: **data races**, which occurs when a read operation needs to wait until the previous write operation has completed, and **port conflicts**, which occurs when hardware resources, such as address signal lines, need to be released before the next instruction can access them. Note that data races only occur when reading and writing to the same address, whereas port conflicts do not depend on the address value. In the following, the number of steps required between instructions will be described for each combination of instructions:

These conditions are all checked by the assembler, and if they are violated, an error occurs.

3.6.3.1 L1BM → L2BM Transfer ⇒ MV Instruction That Reads from L2BM

At least 1 step must be inserted between an instruction transferring L1BM → L2BM and an MV instruction that reads from L2BM, due to data race.

Example

```
l2bm@0 $lb0 $lc4096
nop
mvp/n4160 $lc0@.0 $d0
```

In this example, removing the nop instruction would result in an error.

However, in reality, the MV instruction accesses its operands from the smaller address to the larger one. Therefore, even without the nop, by the time the MV instruction reads from addresses 4096 and above of L2BM, the write operation from the previous L2BM instruction would have already completed.

But, the assembler's checker is simple and doesn't take this into account. Instead, it assumes that the MV instruction accesses all necessary regions in the same cycle it is issued, which is why removing the nop would result in an error.

3.6.3.2 L1BM →L2BM Transfer ⇒ L2BM → L1BM Transfer

At least 3 steps must be inserted between a L1BM → L2BM instruction → and a L2BM → L1BM instruction.

Example

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

In this example, substituting `nop/3` with `nop/2` would result in an error.

3.6.3.3 L2BM → L1BM Transfer ⇒ L1BM → L2BM Transfer / Internal Multicast

At least 2 steps must be inserted between L2BM → L1BM transfer instruction and L1BM → L2BM transfer instruction or internal multicast instruction, due to port conflict. It is only the case where the L1B number read by the second instruction is included in the L1B numbers written by the first instruction.

Example 1

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

In this example, substituting `nop/2` with `nop` would result in an error.

Example 2

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

In this example, only the 0th L1BM is written to and the 1st L1BM is read out in the following instruction, there are no need to insert one step between them.

3.6.3.4 Internal Multicast ⇒ L1BM → L2BM Transfer / Internal Multicast

At least 3 steps must be inserted between Internal multicast instruction and L1BM → L2BM transfer or internal multicast instruction, due to port conflict.

This is only the case where the L1B number read by the second instruction is included in the L1B numbers written by the first instruction.

Example 1

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

In this example, if you change the `nop/3` to `nop/2`, it would result in an error.

Example 2

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

In this example, as there are no overlaps between L1BM written to (i.e. All L1BM except the 0th) and L1BM read out (i.e. only the 0th L1BM), two sequential instructions can be issued immediately.

3.6.3.5 Internal Multicast ⇒ L1BM → PE Transfer

At least 10 cycles must be inserted between internal multicast instruction and L1BM → PE transfer instruction that reads from the same address, due to data race. However, if minimizing latency is not a concern, then inserting at least 3 steps would be sufficient to avoid hazards.

Example

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

In this example, if you change the `$lb52` in the `l1bmm` instruction to `$lb56`, it will result in an error. This is because the write operation that writes to `$lb64` is issued at cycle 0, and the read operation that reads from `$lb64` (assuming the `l1bmm` instruction reads 4 long words per cycle) is issued at cycle 10. This means there are only 9 cycles between the two operations, which is not enough to avoid a data race.

3.6.3.6 L2BM → L1BM Transfer ⇒ L1BM → PE Transfer

At least 6 cycles must be inserted between from L2BM →L1BM transfer instruction and L1BM →PE transfer instruction that reads from the same address, due to data race. However, if minimizing latency is not a concern, then inserting at least 2 steps would be sufficient to avoid hazards.

Example

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

In this example, if you change the `$lb52` in the `l1bmm` instruction to `$lb56`, it will result in an error. This is because the write operation that writes to `$lb64` is issued at cycle 0, and the read operation that reads from `$lb64` (assuming the `l1bmm` instruction reads 4 long words per cycle) is issued at cycle 6. This means there are only 5 cycles between the two operations.

3.6.3.7 PE → L1BM Transfer ⇒ L1BM → L2BM Transfer / Internal Multicast

At least 10 cycles must be inserted between PE → L1BM instruction and from L1BM → L2BM instruction or internal multicast instruction that reads from the same address, due to data race.

However, if minimizing latency is not a concern, then inserting at least 3 steps would be sufficient to avoid hazards

Example

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

In this example, if you change the `$lb48` in the `l1bmr4dfadd` instruction to `$lb32`, it will result in an error.

This is because the write operation that writes to `$lb64` is issued at (counted from cycle 0,) cycle 2 (since 1 word length 1x4 contraction command writes 16 long words per cycle), and the read operation that reads from `$lb64` is issued at cycle 12. This means there are only 9 cycles between the two operations.

3.6.3.8 PE → L1BM Transfer ⇒ L1BM → PE Transfer

At least 2 steps must be inserted between PE → L1BM instruction and L1BM → PE instruction, due to port conflict.

Example 1

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

In this example, if you change the `nop/2` to `nop`, it will result in an error.

3.6.3.9 PE Memory Write ⇒ PE Memory Read

After writing to LM0/LM1, you must wait at least 2 steps due to port conflict before starting a read operation.

After writing to GRF0/GRF1, you must wait at least 1 step before starting a read operation from the same address. However, if the read and write operations access different addresses, they can be issued simultaneously and independently.

After writing to the T-register, you must wait at least 1 step due to data race before starting a read operation.

It is not a problem to perform reads and writes in the same step for LM, GRF, or T-register. In such cases, the read operation will return the data that was already written.

More precisely, any instruction that writes to PE memory (excluding mask registers) takes 6 cycles to complete. The assembler's hazard detection mechanism can be used to verify this. The following assembly sequence is assembled correctly:

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

The `/1000` is a write mask specification. The write mask specification is described in Section 3.6.2.1. The reason why this code works is that the write operation to `$r0` only occurs in the first cycle of the `imm` instruction, and then there are 7 cycles available until the start of the `dvadd` instruction (including the subsequent `nop`). Similarly, if the mask specification is `/0100` (write only in the second cycle), there will still be 6 cycles available, so it is not a problem.

However, if you change it to `/0010` (write only in the third cycle) or `0001` (write only in the fourth cycle), the number of available cycles will drop below 6, and the assembler will detect this and report an error.

Table 3.6 Grouping of PE instruction expressions to describe simultaneous execution condition

Group Name	PE Instruction Opcodes
nop	`nop`
noforward	`noforward`
l2bm	Any L2BM instruction expr. other than `l2bmdarw`
l2bmdarw	`l2bmdarw`
l1bm	Any non-turnaround L1BM instruction expr.
l1bm-turnaround	Any turnaround L1BM instruction expr.
mau-calc	`mfma`, `mmul`, `vfma`, `vmul`, `vadd`, `vpassa`
mau-mwrite	`mwrite`
mau-mread	`mread`
alu	Any ALU instruction expr.
wait	`wait`

The result of writing to the mask register is available for use in the next step. For example, the following assembly sequence is valid:

```
lpassa $lm0v $omr1
lpassa $ln0v $lr0v/$imr1
```

3.6.4 Simultaneous execution condition

Multiple PE instruction expressions can be issued simultaneously within a single step by separating those instructions by semicolon (`;`) if they meet certain conditions.

These conditions are checked by the assembler, and any violations will result in an error.

This section describes these conditions in detail. First, we divide the PE instruction expressions into groups as shown in Table 3.6.

The conditions for issuing multiple PE instructions within a single step are as follows: Here, a PE operand refers to GRF0, GRF1, LM0, LM1, T-register, mask registers, or various forwarding methods.

- At most one instruction expression is issued per group [*4].
- If a `nop` instruction is issued, no other instruction expressions except `wait` is issued.
- Among the mau-calc, mau-mwrite, and mau-mread groups, at most two instruction expressions are issued. If two are issued, they must also meet the following conditions:
  - They have the same precision specification.
  - If `vfma` or `vmul` is issued with `mwrite`, the second input of `vfma` or `vmul` and the input of `mw rite` must have the same PE operand read from, sign inversion (Section 3.6.9.22), precision expansion (Section **??**), and precision shortening (Section 3.6.9.24) specifications.
- The same matrix register operand (`$x` or `$y`) appears at most once in the same line.
- Multiple instruction expressions do not write to the same PE operand.
- If multiple instruction expressions read from the same PE operand, they all access the same region across all cycles.
  - In other words, for addressable PE memory, it is not possible to perform multiple reads from different addresses in a single cycle.
  - This includes reading mask flags during mask application (Sections 3.6.2.1 and 3.6.2.2).
- If LM0 is read from and written to simultaneously, the accessed region must be the same across all cycles. The same applies to LM1.
- If an immediate instruction is issued, LM0 is not accessed.
- a Zero-flush mask application (Section 3.6.2.2) occurs at most once.

[*4] with some exceptions for the `l2bmdars` instruction, which can be issued simultaneously with other l2bm instructions. However, since the situation is complicated and impractical, we treat it within a group

- If multiple mask applications occur, they all have the same mask register width (Section 3.6.2).

In addition to these conditions, there may be other limitations due to hazards (Section 3.6.3) that prevent adding instruction expressions.

The order in which instructions are listed separated by semicolons does not affect the generated machine code. However, error messages may vary depending on the order.

Example

As an extreme example, the following is a valid assembly sequence:

(Note that the second line appears to be wrapped, but in reality, everything from `noforward;` onwards is on a single line.)

```
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

Do nothing for 4 cycles.

This may be necessary to insert in order to avoid data hazards. The NOP instruction can also be automatically inserted by the chip's instruction issuance unit under certain circumstances.

The only PE command that can be issued simultaneously with `nop` is the `wait` command. The `nop` command implies `noforward` (Section 3.6.6).

Syntax

```
nop
```

As a syntax sugar, `nop/<n>` can be used to insert `n` instances of `nop`.

Effects

Do nothing for 4 cycles.

Errors

Attempting to issue any command other than the `wait` command simultaneously with `nop` will result in an error.

Examples

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

Copy 4 long words from LM0 to LM1, wait until reads from LM1 are possible, and then copy another 4 long words to GRF0.

3.6.6 Nonforward - not updating forwarding and turnaround register

In a step where the `noforward` instruction expression is written, do not update the forwarding registers (`$mauf`, `$aluf`, `$mreadf`, `$lbf`) and the L1BM loopback register (`$lbi`).

The `noforward` instruction expression can be issued simultaneously with any other PE instruction expression except `nop`.

Syntax

```
noforward
```

3.6.7 L2BM Instruction Expressions

This section explains the L2BM instruction expressions. These instructions are primarily used for transferring data between L2BM and L1BM. The only instruction that transfers data between L1BMs is internal multicast.

3.6.7.1 L1BM Subset Specification

In some L2BM instruction expressions, it is possible to partially select the target L1Bs for transfer. This subset will be referred to as an "L1B set" in the following sections.

The syntax for specifying an L1B set is as follows:

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

The `<l1badr>` and `immode` are integers from 0 to 7, while the `<l1blist>` is a comma-separated list of L1B numbers from 0 to 7.

In the first notation, let `<l1badr>` be b_0, `immode` be i, and the constant 0b111 be m. Then, an L1B with number b is included in the L1B set if it satisfies the condition: $b\&(m \oplus i) = b_0\&(m \oplus i)$. Here, & represents a logical AND operation, and $\oplus$ represents an exclusive OR operation. In other words, the L1Bs with numbers that match `<l1badr>` when all bits set in `immode` are masked out will be included in the set.

The second notation using `<l1badr>` is a shorthand for setting `immode` to 0 in the first notation. This results in an L1B set consisting of only the L1B with number `<l1badr>`.

The `<l1blist>` allows direct specification of the L1B set. However, only lists that can be represented using the first notation are allowed. For details, see Table 3.7.

Note that even if `<immode>` is the same, different values for `<l1badr>` may sometimes result in the same L1B set, but different values of `immode` will always result in different L1B sets.

Table 3.7 Correspondance of the pair of l1badr and immode to L1B set

[<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 Broadcast

Copy a contiguous region from an L2BM to all or some of the subordinate L1BMs at a rate of 16 long words per cycle.

Syntax

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

The part following the `@` symbol specifies an L1B set according to Section 3.6.7.1, and only the L1Bs included in this set will be written. If omitted, all L1Bs will be written to.

Both the read address `<addr_c>` and the write address `<addr_b>` must be 16-long-word aligned.

Effects

```
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]
```

Example

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

Broadcast a total of 64 long words from L2BM to only 0th, 1st, 2nd, and 3rd L1BMs.

3.6.7.3 l2bmb2 - L2BM → L1BM Distributing Broadcast

Read 64 long words per cycle from L2BM and divide the subordinate L1Bs into 4 groups of 2 each, starting from the beginning. Within each group, broadcast the data, and between groups, distribute the data, writing 16 long words per cycle to each L1BM.

Syntax

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

The part following the `@` symbol specifies an L1B set according to Section 3.6.7.1, and only the L1Bs included in this set will be written. If omitted, all L1Bs will be written to.

Both the read address `<addr_c>` must be 64-long-word aligned, and the write address `<addr_b>` must be 16-long-word aligned.

Effects

```
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]
```

Example

```
l2bmb2 $lc0 $lb0
```

Reads a total of 256 long words from L2BM, and for the L1Bs, broadcasts within groups and distributes between groups, writing 64 long words to each L1BM.

3.6.7.4 l2bmd - L2BM → L1BM Distribution

Reads a contiguous region from L2BM at 64 long words per cycle, and distributes 8 long words per cycle to each of the subordinate L1Bs, writing to a contiguous region.

Syntax

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

The part following the `@` symbol specifies an L1B set according to Section 3.6.7.1, and only the L1Bs included in this set will be written. If omitted, all L1Bs will be written to.

Both the read address `<addr_c>` must be 64-long-word aligned, and the write address `<addr_b>` must be 8-long-word aligned.

Effects

```
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
```

Read a total of 256 long words from L2BM and write different 32-long-word data to 0th, 1st, 2nd, and 3rd L1BMs (data corresponding to 4th, 5th, 6th, and 7th L1BMs will be discarded).

3.6.7.5 l2bm@<l1badr> - L1BM → L2BM Transfer

Copy a contiguous region from the specified L1BM to L2BM at a rate of 16 long words per cycle.

Syntax

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

The `<l1badr>` is an L1B number between 0 and 7.
Both the read address `<addr_b>` and the write address `<addr_c>` must be 16-long-word aligned.

Effects

```
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]
```

Example

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

Copy a total of 64 long words from the 1st L1BM to L2BM.

3.6.7.6 l2bmr<rrn_opcode> - L1BM → L2BM Reduction

Read a contiguous region from all or some of the subordinate L1BMs to L2BM at a rate of 16 long words per cycle, perform a specified reduction operation in the L1B direction, and write the result to L2BM.

This is the counterpart to the L2BM → L1BM broadcast described in Section 3.6.7.2.

Syntax

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

The `<rrn_opcode>` is a reduction operation specifier defined in Section 3.5.5.

The part following the `@` symbol specifies an L1B set according to Section 3.6.7.1, and reads are performed only from the L1Bs included in this set. If omitted, reads are performed from all L1Bs. L1Bs not included in the L1B set transmit the identity value for the reduction operation.

Both the read address `<addr_b>` and the write address `<addr_c>` must be 16-long-word aligned.

Effects

```
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]
```

Note: The reduction is not actually performed internally in this manner.

Example

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

Read 64 long words from 0th and 4th L1BMs, interpret them as double-precision floating-point numbers, add them in the L1B direction, and write the resulting 64 long words to L2BM.

3.6.7.7 l2bmr2<rrn_opcode> - L1BM → L2BM Comibing Reduction

For each L2BM, read a contiguous region from the subordinate L1BMs at a rate of 16 long words per cycle, divide the L1Bs into 4 groups of 2 starting from the beginning, perform the specified reduction operation within each group, join the results between groups, and write the result to the L2BM at a rate of 64 long words per cycle.

This is the counterpart to the L2BM → L1BM distributed broadcast described in Section 3.6.7.3.

Syntax

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

The `<rrn_opcode>` is a reduction operation specifier defined in Section 3.5.5.

Unlike the standard reduction (Section 3.6.7.6), which is not a combining reduce, it is not possible to specify the source L1B set for this instruction.

The read address `<addr_b>` must be 16-long-word aligned, and the write address `<addr_c>` must be 64-long-word aligned.

Effects

```
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]
```

Note: The reduction is not actually performed internally in this manner.

Example

```
l2bmr2dfadd $lb0 $lc0
```

Read 64 long words from each L1BM, interpret them as double-precision floating-point numbers, add them within the group, and join the results between groups to write a total of 256 long words to the L2BM.

3.6.7.8 l2bmd - L1BM → L2BM Gather

For each L2BM, read a contiguous region from all subordinate L1BMs at a rate of 8 long words per cycle each, join the results, and write to a contiguous region in the L2BM at a rate of 64 long words per cycle.

This is the counterpart to the L2BM → L1BM distributed instruction described in Section 3.6.7.4.

Syntax

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

The read address `<addr_b>` must be 8-long-word aligned, and the write address `<addr_c>` must be 64-long-word aligned.

Effects

```
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]
```

Example

```
l2bmd $lb0 $lc0
```

Read 32 long words from each L1BM, join them in units of 8 long words, and write a total of 256 long words to the L2BM.

3.6.7.9 l2bmi - Inter-L1BM Multicast

Using the L2BM-L1BM path, transfer data between different L1BMs within the same L2BM at a rate of 16 long words per cycle.

Since multiple sources and destinations are possible, this is referred to as multicast.

Syntax

```
l2bmi@<l1bset> $lb<addr0> $lb<addr1>
```

The part following the `@` symbol specifies an L1B set according to Section 3.6.7.1, and reads are performed only from the L1Bs included in this set. Let `<immode>` be i and the L1B number be b. Then, writes are performed to the L1Bs where $b\&i$ is the same as read L1B's. Here, & denotes a logical AND operation. For example, if `<immode>` is 0, then $b\&i$ is always 0, so the value read from the L1B specified by `<l1badr>` is sent to all other L1Bs. Alternatively, if `<l1badr>` is 0 and `<immode>` is 6 (=0b110), then the L1Bs are grouped based on their upper two binary digits. The values read from 0th, 2nd, 4th, and 6th L1Bs are sent to 1st, 3rd, 5th, and 7th L1Bs, respectively. Note that `<immode>` cannot be 7, and it is also not possible to specify a set consisting of all L1Bs using the direct specification format for the L1B set.

Both the read address `<addr0>` and the write address `<addr1>` must be 16-long-word aligned.

Effects

```
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 != l1b_src && l1b_dst & immode == l1b_src & immode
          MEM[group][l2b][l1b_dst].l1bm[dst_addr:dst_addr+16] = data[0:16]
```

Example

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

Read a total of 64 long words from the 0th L1BM and broadcast them to the same address in the 1st, 2nd, and 3rd L1BMs. Perform the same operation for the 4th L1BM and 5th, 6th, and 7th L1BMs.

3.6.7.10 l2bmdars/l2bmdarw - DAR Write

The **DAR (DRAM Address Register)** is a memory that stores the DRAM address for indirect referencing of DRAM. The DAR is written to through the **DARBUF (DAR write buffer)**, which allows copying of address values from L2BM *5.

The DRAM indirect reference address word is 32 bits wide, meaning that one long word read from L2BM corresponds to two address words.

The `l2bmdars` instruction copies data from L2BM to DARBUF at a rate of 128 address words per cycle. The `l2bmdarw` instruction copies data from DARBUF to DAR at a rate of 1 address word per cycle. These two instructions are used in tandem to copy address values from L2BM to DAR.

There is one DARBUF for each group, and it has a size of 512 address words.

There is one DAR for each group, with a size of 1024 address words.

The `l2bmdars` instruction reads 64 long words from L2BM per cycle and writes them to DARBUF as 128 address words. The MSB side (32 bits) of each long word corresponds to the lower address in DARBUF (even address), while the LSB side (32 bits) corresponds to the higher address (odd address). Since L2BM is read at a rate of 64 long words per cycle, one step can fill the entire DARBUF. There is no explicit specification for the starting write address of DARBUF. Instead, the first cycle of `l2bmdars` writes the first 128 address words to DARBUF, and subsequent cycles increment by 128 address words.

Additionally, the first cycle of `l2bmdars` resets the DARBUF read address to zero and specifies the starting write address for DAR. The `l2bmdarw` instruction can be issued at the same step as `l2bmdars`, and it transfers data from DARBUF to DAR. During execution of `l2bmdarw`, the DARBUF read address and DAR write address are incremented by 1 each cycle, wrapping around when they reach the end.

The `l2bmdarw` instruction can be issued concurrently with any other L2BM instructions.

Syntax of `l2bmdars`

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

Effects of `l2bmdars`

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

Syntax of `l2bmdarw`

```
l2bmdarw
```

Effects of `l2bmdarw`

```
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
```

Example

*5 This is done to reduce the time spent occupying L2BM and transferring address values between L2BM and DRAM.

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

Copy address values from the 1st L2BM to DARBUF, and simultaneously start writing to DAR from address 16. Thereafter, continue writing to DAR in parallel with distributing data from L2BM to L1BM.

3.6.8 L1BM Instruction Expressions

This section explains the L1BM instruction expressions.

A list of L1BM instructions, including their word length distinctions, is shown in Table 3.8.

Table 3.8 The list of L1BM instruction expr. The speed unit at L1BM and PE is long words per cycle. LW, 2LW, BC, TF, RDC stand for long-word, 2-long-word, broadcast, transfer, and reduction, respectively.

Name	Direction	L1BM rate	PE rate	Example
LW PE BC (3.6.8.6)	L1BM → PE	1	1	`l1bmp $lb0 $lr0v`
2LW PE BC (3.6.8.7)	L1BM → PE	2	2	`l1bmp $llb0 $llr0v`
LW 16x1MAB BC (3.6.8.8)	L1BM → PE	4	1	`l1bmm $lb0 $lr0v`
2LW 16x1MAB BC (3.6.8.9)	L1BM → PE	8	2	`l1bmm $llb0 $llr0v`
LW 16x1 TF (3.6.8.10)	PE → L1BM	4	1	`l1bmm@0 $lr0v $lb0`
2LW 16x1 TF (3.6.8.11)	PE → L1BM	8	2	`l1bmm@0 $llr0v $llb0`
LW 16x1 RDC (3.6.8.12)	PE → L1BM	4	1	`l1bmrdfadd $lr0v $lb0`
2LW 16x1 RDC (3.6.8.13)	PE → L1BM	8	2	`l1bmrffadd $llr0v $llb0`
LW 4x4MAB BC (3.6.8.14)	L1BM → PE	16	1	`l1bmm4 $lb0 $lr0v`
2LW 4x4MAB BC (3.6.8.15)	L1BM → PE	32	2	`l1bmm4 $llb0 $llr0v`
LW 4x4 TF (3.6.8.16)	PE → L1BM	16	1	`l1bmm4@0 $lr0v $lb0`
2LW 4x4 TF (3.6.8.17)	PE → L1BM	32	2	`l1bmm4@0 $llr0v $llb0`
LW 4x4 RDC (3.6.8.18)	PE → L1BM	16	1	`l1bmr4dfadd $lr0v $lb0`
2LW 4x4 RDC (3.6.8.19)	PE → L1BM	32	2	`l1bmr4ffadd $llr0v $llb0`
Distribution (3.6.8.20)	L1BM → PE	64	1	`l1bmd $lb0 $lr0v`
Gather (3.6.8.21)	PE → L1BM	64	1	`l1bmd $lr0v $lb0`

3.6.8.1 About 4x4 Mode

In the 4x4 transfer modes listed in Table 3.8, when MAB number is viewed as a 4-bit binary number, broadcasts and reductions are performed on the lower 2 bits, while distributions and gather are performed on the upper 2 bits.

The access speed of L1BM is four times that of the corresponding 16x1 mode instruction. In terms of L1BM address space, this four-fold increase corresponds to the outermost part. For example, in a double-long-word 4x4 MAB broadcast, the 32 long words read from L1BM in one cycle can be thought of as having a multidimensional array structure in C like [4 (upper 2 bits of MAB number)][2 (long word)][4 (PEs)].

3.6.8.2 Types of L1BM Instruction Expressions and Turnaround Operations

Transfer commands with the same access speed on both the L1BM side and the PE side are considered to be of the same type. In transfers of the same type, the layout is preserved. In other words, even if you transfer data from L1BM to PE, perform some element-wise operation, and then transfer it back to L1BM, the layout will not be disrupted.

Additionally, within the same type of transfer, **turnaround** operations are possible between PE → L1BM transfers and L1BM → PE transfers. A **turnaround** operation refers to the following:

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

This is equivalent to the following:

```
l1bmm@2 $lr0v $lb0
l1bmm@2 $lr8v $lb16
nop
nop
l1bmm $lb0 $lm0v
l1bmm $lb16 $lm8v
```

In other words, by reading from the turnaround register `$lbi` (Section 3.6.1.5), you can immediately read out the data written to L1BM in the previous step and issue it simultaneously with the subsequent PE $\rightarrow$ L1BM transfer.

Additionally, by setting the write destination of the PE $\rightarrow$ L1BM transfer to `$lbi`, you can update only the turnaround register without writing to L1BM.

The turnaround register is not updated while no PE $\rightarrow$ L1BM transfers or turnaround transfers are issued.

In the following pseudo-code descriptions, the turnaround register will be ignored for simplicity.

3.6.8.3 Word Length of PE Side Operands

Even for instructions with a "PE-side speed" of 1 long word per cycle, as shown in Table 3.8, it is possible to specify 2 long words as operands on the PE side. In this case, if it is an L1BM $\rightarrow$ PE transfer, the MSB side will contain valid values, and the LSB side will be all zeros. If it is a PE $\rightarrow$ L1BM transfer, the LSB side of the 1st long word will be discarded.

For L1BM $\rightarrow$ PE transfers with a "PE-side speed" of 2 long words per cycle, specifying a PE operand with a length smaller than 2 long words will result in an error.

In the following pseudo-code descriptions, it is assumed that the "PE-side speed" and the length of the PE-side operands match.

3.6.8.4 Precision Extension of Inputs

For the instructions shown in Table 3.8 with a "PE-side speed" of 2 long words, specifically the reduction commands that specify single-precision floating-point arithmetic, it is possible to add an `e` suffix to the PE-side operand. This causes the value read from the PE-side operand to be interpreted as half-precision floating-point numbers and converted to single-precision floating-point numbers with increased precision.

The arrangement of data within a cycle during this precision extension is as follows: For the MSB side 1 long word of the 2 long words sent from the p-th PE, assigning the number $4 \times p + j$ (in hexadecimal notation) to the j-th half-word starting from the MSB side, and converting these half-words to single-precision words, results in an arrangement on L1BM that is equivalent to arranging them in the order of `014589cd2367abef`, starting from the MSB side. This arrangement was chosen so that if 2 long word broadcasts are performed on the data written to L1BM and rounding to half-precision is done on PE, the resulting arrangement will match the original arrangement on PE.

Example

```
l1bmrffadd $lr0ve $llb0
```

Read 4 half-precision words per cycle and PE from GRF0, convert them to single precision, perform addition reduction, and write the result as 16 single-precision words (8 long words) to L1BM per cycle.

3.6.8.5 Precision Shortening of Reduction Results

For the reduction commands shown in Table 3.8, specifically when the reduction operation specifies single-precision floating-point arithmetic, adding an `r` suffix to the opcode allows for rounding of the

result from the reduction circuit to half-precision floating-point numbers before writing it to L1BM or the turnaround register. This is referred to as **precision shortening**.

The arrangement of data within a cycle during precision shortening is as follows: Let the 16 single words output by the reduction circuit be represented in order from MSB side as `0123456789abcdef`. For example, values from `0` to `3` originate from PE0. In this case, the arrangement of 4 long words on L1BM is obtained by rounding them to half-words and arranging them in order from MSB side as `018923ab45 cd67ef`.

This arrangement corresponds to the inverse transformation of the reordering used for precision expansion described in Section 3.6.8.4. Half-precision floating-point reduction is actually implemented using this process: precision expansion → single-precision reduction → precision shortening.

In other words, `l1bmrhfadd $lr0v $lb0` is an alias for `l1bmrffaddr $lr0ve $lb0`.

Example

```
l1bmrffaddr $llr0v $lb0
```

Read 4 single-precision words per cycle and PE from GRF0, perform addition reduction, round to 16 half-precision words, and write the result to L1BM per cycle.

3.6.8.6 l1bmp - Long-word PE Broadcast

Read 1 long word from L1BM per cycle and broadcast it to all subordinate 64 PEs.

There is no symmetric transfer instruction in the PE → L1BM direction.

Syntax

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

The syntax `<dst_0> [<dst_1>..]` represents the destination PE operand(s) for writing.

For example, like `$lr0`, `$lm0v/$imr1`, and `$lt` in the instruction `l1bmp $lb0 $lr0 $lm0v/$imr1 $lt`, multiple operands can be specified for different memory units. However, it is not allowed to specify the same memory unit multiple times as an output operand, such as in `l1bmp $lb0 $lm0v $lm100v`. For simplicity, only a single destination operand `dst` is used in the following effect examples. This convention applies to all subsequent L1BM instruction notation.

There are no alignment constraints on the L1BM side.

Effects

```
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 - Double-long-word PE Broadcast

Read 2 long words from L1BM per cycle and broadcast them to all subordinate 64 PEs.

There is no symmetric transfer instruction in the PE → L1BM direction.

Syntax

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

The syntax `<dst_0> [<dst_1>..]` represents the destination PE operand(s) for writing.

There are no alignment constraints on the L1BM side, but the lower 6 bits of the address must have a value between 0 and 56 (inclusive).

Effects

```
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]
```

Note that the read region on L1BM is non-contiguous within a cycle.

3.6.8.8 l1bmm - Long-word 16x1MAB Broadcast

Read 4 long words from L1BM per cycle, broadcast them to all subordinate 16 MABs, and distribute the 4 long words among 4 PEs to write 1 long word each.

Syntax

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

The syntax `<dst_0> [<dst_1>..]` represents the destination PE operand(s) for writing.
The L1BM address `<addr_b>` must be 4-long-word aligned.

Effects

```
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 - Double-long-word 16x1MAB Broadcast

Read 8 long words from L1BM per cycle, broadcast them to all subordinate 16 MABs, and distribute the 8 long words among 4 PEs to write 2 long words each.

Syntax

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

The syntax `<dst_0> [<dst_1>..]` represents the destination PE operand(s) for writing.
The L1BM address `<addr_b>` must be 8-long-word aligned.

Effects

```
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> - Long-word 16x1 Transfer

For each L1BM, read 1 long word from each of the 4 PEs under the specified MAB number per cycle, join them, and write the result to L1BM as 4 long words per cycle.

Syntax

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

The syntax `<mabadr>` represents the MAB number, which is a value between 0 and 15. The syntax `<src>` represents the source PE operand for reading.

The L1BM address `<addr_b>` must be 4-long-word aligned.

Effects

```
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> - Double-long-word 16x1 Transfer

For each L1BM, read 2 long words from each of the 4 PEs under the specified MAB number per cycle, join them, and write the result to L1BM as 8 long words per cycle.

Syntax

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

The syntax `<mabadr>` represents the MAB number, which is a value between 0 and 15. The syntax `<src>` represents the source PE operand for reading.

The L1BM address `<addr_b>` must be 8-long-word aligned.

Effects

```
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> - Long-word 16x1 Reduction

For each L1BM, read 1 long word from each subordinate 4 PEs of each MAB per cycle, perform reduction in the MAB direction and gather in the PE direction, and write the result to L1BM as 4 long words per cycle.

Syntax

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

The syntax `<rrn_opcode>` specifies the reduction operation, as defined in Section 3.5.5.
The syntax `<src>` represents the source PE operand for reading.
The L1BM address `<addr_b>` must be 4-long-word aligned.

Effects

```
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]
```

Note: In the real device, the reduction is not actually carried out using the above procedure.

3.6.8.13 l1bmr<rrn_opcode> - Double-long-word 16x1 Reduction

For each L1BM, read 2 long words from each subordinate 4 PEs of each MAB per cycle, perform reduction in the MAB direction and gather in the PE direction, and write the result to L1BM as 8 long words per cycle.

This double-long-word operation is only possible for single-precision floating-point operations or arbitrary-precision `bor` operations.

This instruction can also specify the operation with input value conversion from half-precision to single-precision, as described in Section 3.6.8.4.

Syntax

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

The syntax `<rrn_opcode>` specifies the reduction operation, as defined in Section 3.5.5.

The syntax `<src>` represents the source PE operand for reading.

The L1BM address `<addr_b>` must be 8-long-word aligned.

Effects

```
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]
```

Note: The reduction is not actually performed internally in this order.

3.6.8.14 l1bmm4 - Long-word 4x4MAB Broadcast

Read 16 long words from L1BM per cycle, distribute and broadcast them to the MABs according to the mapping described in Section 3.6.8.1, further distribute the resulting 4 long words among 4 PEs, and write 1 long word each.

Syntax

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

The syntax `<dst_0> [<dst_1>..]` represents the destination PE operand(s) for writing.
The L1BM address `<addr_b>` must be 16-long-word aligned.

Effects

```
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 - double-long-word 4x4MAB Broadcast

Read 32 long words from L1BM per cycle, distribute and broadcast them to the MABs according to the mapping described in Section 3.6.8.1, further distribute the resulting 8 long words among 4 PEs, and write 2 long words each.

Syntax

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

The syntax `<dst_0> [<dst_1>..]` represents the destination PE operand(s) for writing.
The L1BM address `<addr_b>` must be 32-long-word aligned.

Effects

```
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> - Long-word 4x4 Transfer

For each L1BM, read 1 long word from each subordinate 4 PEs of MAB $0 + i, 4 + i, 8 + i, 12 + i$ (where i is one of the specified values: $0, 1, 2, 3$) per cycle, join them, and write the result to L1BM as 16 long words per cycle.

Syntax

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

The syntax `<mabadr>` represents the MAB number, which is a value between 0 and 3. The syntax `<src>` represents the source PE operand for reading.

The L1BM address `<addr_b>` must be 16-long-word aligned.

Effects

```
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> - Double-long-word 4x4 Transfer

For each L1BM, read 2 long words from each 4 PEs of MAB $0+i, 4+i, 8+i, 12+i$ (where i is one of the specified values: $0, 1, 2, 3$) per cycle, join them, and write the result to L1BM as 32 long words per cycle.

Syntax

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

The syntax `<mabadr>` represents the MAB number, which is a value between 0 and 3. The syntax `<src>` represents the source PE operand for reading.

The L1BM address `<addr_b>` must be 32-long-word aligned.

Effects

```
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> - Long-word 4x4 Reduction

For each L1BM, read 1 long word from each subordinate 4 PEs of MAB per cycle, join them in the PE direction, perform reduction and gather with respect to the MABs according to the mapping described in Section 3.6.8.1, and write the result to L1BM as 16 long words per cycle.

Syntax

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

The syntax `<rrn_opcode>` specifies the reduction operation, as defined in Section 3.5.5.
The syntax `<src>` represents the source PE operand for reading.
The L1BM address `<addr_b>` must be 16-long-word aligned.

Effects

```
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]
```

Note: In the real device, the reduction is not actually carried out using the above procedure.

3.6.8.19 l1bmr4<rrn_opcode> - Double-long-word 4x4 Reduction

For each L1BM, read 2 long words from each subordinate 4 PEs of MAB per cycle, join them in the PE direction, perform reduction and gather with respect to the MABs according to the mapping described in Section 3.6.8.1, and write the result to L1BM as 32 long words per cycle.

This double-long-word operation is only possible for single-precision floating-point operations or arbitrary-precision `bor` operations.

This instruction can also specify the operation with input value conversion from half-precision to single-precision, as described in Section 3.6.8.4.

Syntax

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

The syntax `<rrn_opcode>` specifies the reduction operation, as defined in Section 3.5.5.

The syntax `<src>` represents the source PE operand for reading.

The L1BM address `<addr_b>` must be 32-long-word aligned.

Effects

```
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]
```

Note: In the real device, the reduction is not actually carried out using the above procedure.

3.6.8.20 l1bmd - Distribution

Read 64 long words from L1BM per cycle and distribute them to the subordinate 64 PEs, writing 1 long word each.

In terms of L1BM addresses, MAB numbers correspond to the outer dimension and PE numbers correspond to the inner dimension. In C-like notation, this corresponds to `[16(MABnumber)][4(PEnumber)]`.

As a result, data on L1BM is sent to different MABs in blocks of 4 long words, but it is possible to rotate the destination MAB number in a round-robin manner.

Note that there are no distribution instructions for double-long-word operations.

Syntax

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

The syntax `<dst_0> [<dst_1>..]` represents the destination PE operand(s) for writing.

The syntax `<mabdiff>` specifies the amount to rotate the MAB number, which is a value between 0 and 15 preceded by "+" or "-". The sign is mandatory even for positive values. This creates a correspondence where the data that would have been sent to an MAB without rotation is instead sent to the MAB rotated by `<mabdiff>` positions. If `<mabdiff>` is not specified, the data would be sent to the intended MAB.

The MAB rotation feature also applies to turnaround transfers. To use this feature correctly, the amount of MAB rotation specified in this instruction must match the amount specified in the corresponding gather instruction (Section 3.6.8.21). Since this feature does not apply when writing to a turnaround register using a gather instruction, the amount wouldn't be doubled.

The L1BM address `<addr_b>` must be 64-long-word aligned.

Effects

```
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 dst_mab = (mab + mabdiff) % 16
      MEM[group][l2b][l1b][dst_mab][pe].refer_pemem(dst, cycle) = data[mab * 4 + pe]
```

Example

```
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
```

As an example to illustrate turnaround transfers, the gather instruction described in Section 3.6.8.21 is also included. Suppose that the registers `$ls0, $ls8, $ls16, $ls24` of MAB 0 contain the values $15, 1, 15, 1$, respectively.

In this case, when performing turnaround transfers, the first two registers have the MAB rotation feature applied during the transfer and the last two registers have the MAB rotation feature applied when performing the join write to `$lb0, $lb256`.

3.6.8.21 l1bmd - Gather

Read 1 long word from each PE per cycle, join the values from all 64 PEs under L1BM, and write the result to L1BM as 64 long words per cycle.

The correspondence between L1BM addresses and MAB numbers/PE numbers is the same as that of the distribution instruction.

When writing, values sent from different MABs are written in blocks of 4 long words. However, it is possible to rotate the source MAB number in a round-robin manner. Note that this rotation feature does not apply when writing to turnaround registers.

Also, there are no gather instructions for double-long-word operations.

Syntax

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

The syntax `<src>` represents the source PE operand for reading.

The syntax `<mabdiff>` specifies the amount to rotate the MAB number, which is a value between 0 and 15 preceded by "+" or "-". The sign is mandatory even for positive values. This creates a correspondence where the data that would have been sent to an address without rotation is instead sent to the address rotated by `<mabdiff>` positions. If `<mabdiff>` is not specified, the data would be sent to the intended MAB.

For notes on turnaround transfers, refer to the item in the distribution instruction (Section 3.6.8.20).

The L1BM address `<addr_b>` must be 64-long-word aligned.

Effects

```
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]
```

The syntax `l1bm_turnaround` represents the turnaround registers.

Write operations to the turnaround registers occur not only with this instruction, but also with other PE → L1BM transfer instructions. However, gather instructions have a special behavior regarding MAB rotation when writing to turnaround registers, so it is noted separately here.

Example

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

In this example, the values written to the 4-long-word range starting from `$ls0` and the 4-long-word range starting from `$ls8` will be equal.

This means that it is defined that specifying a certain value for `<mabdiff>` during gather or distribution is equivalent.

3.6.9 MAU instruction Expressions

3.6.9.1 Basic Operations

The MAU has two operation modes: matrix-vector multiply-add mode and vector multiply-add mode. The precision modes are double precision, single precision, pseudo-single precision, and half precision.

In half-precision mode, accumulation is performed in single-precision. In other words, the input and output of the sum part of the multiply-add operation are both single-precision.

The opcodes for matrix-vector multiply-add mode are as follows:

- `mfma` - Takes three inputs and performs a matrix-vector product of the matrix from the first input and the vector from the second input, and then computes an element-wise sum with the vector from the third input.
- `mmul` - A 2-input version of `mfma`, equivalent to setting the third input to 0.

The opcodes for vector multiply-add mode are as follows:

- `vfma` - Takes three inputs and performs an element-wise product of the first and second inputs, and then computes an element-wise sum with the third input.
- `vmul` - A 2-input version of `vfma`, equivalent to setting the third input of to 0.
- `vadd` - A 2-input version of `vfma`, equivalent to setting the second input of to 0.
- `vpassa` - A 1-input version of `vfma`, equivalent to setting the third input of to 0 and the second input to 1.

First, we describe the **basic operation** of `mfma` and `vfma` without input sign inversion, precision extension, and precision shortening. We will then explain the basic operation for each arithmetic mode and precision mode.

3.6.9.2 dmfma - Basic Operation of Matrix-Vector Multiply-Add

Assuming the matrix register A contains matrix data A, performs a 4-dimensional double-precision matrix-vector product FMA (Ax+y).

The operation uses a 2x4 sub-matrix extracted from the 4x4 block of floating-point double-precision matrix data read from the matrix register. The specific rows used are determined by the following specification, and can be either 0, 1 or 2, 3.

Syntax

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

The (u|d) syntax specifies whether to use the upper half (rows 0 and 1) or lower half (rows 2 and 3) of the 4x4 matrix for the operation. If u is specified, the operation uses rows 0 and 1 of matrix A, multiplies them by x, and stores the result in elements 0 and 1 of Ax. Elements 2 and 3 of Ax are set to 0, and then added to y (with offset=0). If d is specified, the operation uses rows 2 and 3 of matrix A, multiplies them by x, and stores the result in elements 2 and 3 of Ax. Elements 0 and 1 of Ax are set to 0, and then added to y (with offset=2).

The first input $l(x|y) is the source matrix register, referred to as side in the following effect.

The second input <src_x> and third input <src_y> are read from PE operands. <src_x> is a block float double-precision value with an access length of long word. In the basic operation, <src_y> is a normal double-precision value with an access length of long word.

The output <dst_0> [<dst_1>..] is a write destination PE operand. Although multiple PE memories can be written to simultaneously, for simplicity, a single write destination dst is specified in the following effects. In the basic operation, the result is a normal double-precision value with an access length of long word.

Effects

```
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 dmmul - Double-precision Matrix-Vector Product

The dmmul instruction is equivalent to the basic operation of dmfma with the third input set to 0, i.e., it computes a double-precision matrix-vector product.

Like dmfma, the (u|d) specification is required.

Example

```
dmmulu $lx $lr0v $nowrite
dmfmad $lx $lr0v $mauf $ls0v
```

Using the data in the matrix register as A and the data in GRF0 as x, first compute the 0,1 rows of Ax using the dmmulu instruction. Then, use the dmfmad instruction to compute the 2,3 rows of Ax while simultaneously forwarding the multiplication result of the 0,1 rows using MAU result forwarding (Section 3.6.1.15). This combination of two instructions performs a 4x4 double-precision matrix-vector product (Ax) and writes the result to GRF1.

3.6.9.4 fmfma - Basic Operation of Single-precision Matrix-Vector Multiply-Add

Takes the matrix data stored in the matrix register `A`, performs a 4-dimensional single-precision matrix-vector product FMA (`Ax+y`).

The operation uses an 8x4 submatrix extracted from the 8x8 block-floating-point single-precision matrix data read from the matrix register. Specifically, columns 0, 2, 4, and 6 are selected for the operation.

Syntax

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

The first input `$l(x|y)` is the source matrix register, referred to as `side` in the following effect.

The second input `<src_x>` and third input `<src_y>` are read from PE operands. `<src_x>` is a block-floating-point single-precision value with an access length of word. In the basic operation, `<src_y>` is a normal single-precision value with an access length of long word.

The output `<dst_0> [<dst_1>..]` is a write destination PE operand. Although multiple PE memories can be written to simultaneously, for simplicity, a single write destination `dst` is specified in the following effects. In the basic operation, the result is a normal single-precision value with an access length of long word.

Effects

```
for cycle = 0:4
  forall chip,l2b,l1b,mab
    SingleWord src_data_A[8][8] = MEM[chip][l2b][l1b][mab].refer_matreg(side, SingleWord)
    SingleWord src_data_x[4] = MEM[chip][l2b][l1b][mab][0:4].refer_pemem(src_x, cycle)
    SingleWord src_data_y[4][2] = MEM[chip][l2b][l1b][mab][0:4].refer_pemem(src_y, cycle)
    SingleWord 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]
```

3.6.9.5 fmmul - Single-Precision Matrix-Vector Product

The `fmmul` instruction is equivalent to the basic operation of `fmfma` with the third input set to 0, i.e., it computes a single-precision matrix-vector product.

3.6.9.6 gmfma - Pseudo-Single-Precision Matrix-Vector Multiply-Add

Takes the matrix data stored in the matrix register as `A` performs a 8-dimensional pseudo-single-precision matrix-vector multiply-add FMA (A x + y).

Syntax

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

The first input `$l(x|y)` is the source matrix register, referred to as `side` in the following effect.

The second input `<src_x>` and third input `<src_y>` are read from PE operands. `<src_x>` is a block-floating pseudo-single-precision value with an access length of long word. In the basic operation, `<src_y>` is a normal single-precision value with an access length of long word.

The output `<dst_0> [<dst_1>..]` is a write destination PE operand. Although multiple PE memories can be written to simultaneously, for simplicity, a single write destination `dst` is specified in the following effect. In the basic operation, the result is a normal single-precision value with an access length of long word.

Effects

```
for cycle = 0:4
  forall chip,l2b,l1b,mab
    SingleWord src_data_A[8][8] = MEM[chip][l2b][l1b][mab].refer_matreg(side, SingleWord)
    SingleWord src_data_x[4][2] = MEM[chip][l2b][l1b][mab][0:4].refer_pemem(src_x, cycle)
    SingleWord src_data_y[4][2] = MEM[chip][l2b][l1b][mab][0:4].refer_pemem(src_y, cycle)
    SingleWord 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]
```

3.6.9.7 gmmul - Pseudo-Single-Precision Matrix-Vector Product

The `gmmul` instruction is equivalent to the basic operation of `gmfma` with the third input set to 0, i.e., it computes a pseudo-single-precision matrix-vector product.

3.6.9.8 hmfma - Basic Operation of half-precision Matrix-Vector Multiply-Add

Takes the matrix data stored in the matrix register as `A` performs a 16-dimensional half-precision matrix-vector multiply-add FMA (`Ax+y`).

Syntax

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

The first input `$l(x|y)` is the source matrix register, referred to as `side` in the following effect.

The second input `<src_x>` and third input `<src_y>` are read from PE operands. `<src_x>` is a block-floating half-precision value with an access length of long word. In the basic operation, `<src_y>` is a normal single-precision value with an access length of double-long words.

The output `<dst_0> [<dst_1>..]` is a write destination PE operand. Although multiple PE memories can be written to simultaneously, for simplicity, a single write destination `dst` is specified in the following effect. In the basic operation, the result is a normal single-precision value with an access length of double-long words.

Effects

```
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)
    SingleWord src_data_y[4][4] = MEM[chip][l2b][l1b][mab][0:4].refer_pemem(src_y, cycle)
    SingleWord 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]
```

3.6.9.9 hmmul - Half-Precision Matrix-Vector Product

The `hmmul` instruction is equivalent to the basic operation of `hmfma` with the third input set to 0, i.e., it computes a half-precision matrix-vector product.

3.6.9.10 dvmul - Double-Precision Vector Multiply-Add

Performs a double-precision vector FMA (x*y+z) within the MAB.

Syntax

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

The (u|d) specifier determines whether the multiplication is performed on the elements of PE0 and PE1 or PE2 and PE3. When u is specified, the product of x and y is computed in PE0 and PE1, and the result is treated as an intermediate value x*y. In PE2 and PE3, this intermediate value is considered to be 0, and only the addition with z is performed (in the following effect, offset=0). When d is specified, the product of x and y is computed in PE2 and PE3, and the result is treated as an intermediate value x*y. In PE0 and PE1, this intermediate value is considered to be 0, and only the addition with z is performed (in the following effect, offset=2).

The first input <src_x>, second input <src_y>, and third input <src_z> are read from PE operands. In the basic operation, they are all normal double-precision values with an access length of long word.

The output <dst_0> [<dst_1>..] is a write destination PE operand. Although multiple PE memories can be written to simultaneously, for simplicity, a single write destination dst is specified in the following effect. In the basic operation, the result is a normal double-precision value with an access length of long word.

Effects

```
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]
```

3.6.9.11 dvmul - Double-Precision Vector Product

The dvmul instruction is equivalent to the basic operation of dvfma with the third input set to 0, i.e., it computes a double-precision vector product.

Like dvfma, (u|d) specification is required.

Example

```
dvmulu $lm0v $lr0v $nowrite
dvfmad $lm0v $lr0v $mauf $ls0v
```

The data in LM0 is denoted as x and the data in GRF0 is denoted as y. First, x*y for PE0 and PE1 is computed using dvmulu, and then dvfmad is used to compute x*y for PE2 and PE3 while forwarding the multiplication result from PE0 and PE1 through MAU result forwarding (Section 3.6.1.15). This process takes two instructions to compute the double-precision vector product (x*y) across all PEs, which is then written to GRF1.

3.6.9.12 dvadd - Basic Operation of Double-Precision Vector Sum

Performs a double-precision vector sum (x+y) within the MAB (Matrix ALU Block).

Syntax

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

The first input `<src_x>` and second input `<src_y>` are read from PE operands. In the basic operation, they are both normal double-precision values with an access length of long word.

The output `<dst_0> [<dst_1>..]` is a write destination PE operand. Although multiple PE memories can be written to simultaneously, for simplicity, a single write destination `dst` is specified in the following effect. In the basic operation, the result is a normal double-precision value with an access length of long word.

Effects

```
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
```

3.6.9.13 dvpassa - Double Precision Vector Copy

The `dvpassa` instruction is equivalent to the basic operation of `dvadd` with the second input set to 0, i.e., it performs a copy operation.

Note that although no multiply-add occurs, normalization as a floating-point number does occur, so it is not a perfect copy.

3.6.9.14 fvfma - Single-Precision Vector Multiply-Add

Performs a single-precision vector FMA (x*y+z) within the MAB.

Syntax

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

The first input <src_x>, second input <src_y>, and third input <src_z> are read from PE operands. In the basic operation, they are all normal single-precision values with an access length of long word.

The output <dst_0> [<dst_1>..] is a write destination PE operand. Although multiple PE memories can be written to simultaneously, for simplicity, a single write destination dst is specified in the following effect. In the basic operation, the result is a normal single-precision value with an access length of long word.

Effects

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

3.6.9.15 fvmul - Single-Precision Vector Product

The fvmul instruction is equivalent to the basic operation of fvfma with the third input set to 0, i.e., it computes a single-precision vector product.

3.6.9.16 fvadd - Single-Precision Vector Sum

The fvadd instruction is equivalent to the basic operation of fvfma with the second input set to 1, i.e., it computes a single-precision vector sum.

3.6.9.17 fvpassa - Single-Precision Vector Copy

The fvpassa instruction is equivalent to the basic operation of fvfma with the second input set to 1 and the third input set to 0, i.e., it performs copy.

Note that although no multiply-add occurs, normalization as a floating-point number does occur, so it is not a perfect copy.

3.6.9.18 hvfma - Basic Operation of Half-Precision Vector Multiply-Add

Performs a half-precision vector FMA (`x*y+z`) within the MAB.

Syntax

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

The first input `<src_x>`, second input `<src_y>`, and third input `<src_z>` are read from PE operands. In the basic operation, `<src_x>` and `<src_y>` are normal half-precision values with an access length of long word. In the basic operation, `<src_z>` is a normal single-precision value with an access length of double-long words.

The output `<dst_0> [<dst_1>..]` is a write destination PE operand. Although multiple PE memories can be written to simultaneously, for simplicity, a single write destination `dst` is specified in the following effect. In the basic operation, the result is a normal single-precision value with an access length of double-long words.

Effects

```
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)
    SingleWord src_data_z[4] = MEM[chip][l2b][l1b][mab][pe].refer_pemem(src_z, cycle)
    SingleWord dst_data[4] = src_data_x[:] * src_data_y[:] + src_data_z[:]
    MEM[chip][l2b][l1b][mab][pe].refer_pemem(dst, cycle) = dst_data[:]
```

3.6.9.19 hvmul - Half-Precision Vector Product

The `hvmul` instruction is equivalent to the basic operation of `hvfma` with the third input set to 0, i.e., it computes a single-precision vector product.

3.6.9.20 hvadd - Half-Precision Vector Sum

The `hvadd` instruction is equivalent to the basic operation of `hvfma` with the second input set to 1, i.e., it computes a single-precision vector sum.

3.6.9.21 hvpassa - Half-Precision Vector Copy

The `hvpassa` instruction is equivalent to the basic operation of `hvfma` with the second input set to 1 and the third input set to 0, i.e., it performs copy.

Note that although no multiply-add occurs, normalization as a floating-point number does occur, so it is not a perfect copy.

3.6.9.22 Input Sign Inversion

For the input operands of `mfma,vfma,mwrite` instructions, except for matrix operands, it is possible to reverse the sign of each element of the input value. To reverse the sign, a minus sign (`-`) is added to the beginning of that operand. If you want to reverse the sign of the first input matrix in the `mfma` instruction, you can write it with the reversed sign using the `mwrite` instruction beforehand.

Example 1

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

Assuming the data in the matrix register is `A`, GRF0 is `x`, and LM0 is `y`, performs a double-precision matrix-vector product FMA with the sign of `y` reversed (`Ax-y`) and writes the result to LM1.

Example 2

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

Assuming the data in GRF0 is `x` and LM0 is `y`, performs a double-precision vector FMA with the signs of both `x` and `y` reversed (`-x-y`) and writes the result to LM1.

3.6.9.23 Precision Extension of Inputs

For the input operands of `mfma,vfma,mwrite` instructions that require normal single-precision or double-precision values in the basic operation, it is possible to indicate a conversion from a one step lower precision value, i.e., from half-precision to single-precision or from single-precision to double-precision. To indicate precision extension, an `e` is appended to the end of that operand.

Note that precision extension cannot be indicated for operands that require block float single-precision or block float double-precision. Precision extension must be done before block-floating-point conversion.

The number of words used does not change from the basic operation, and the word length per word becomes half. Therefore, if the input length is double-long words in the basic operation, it will read from PE memory in long word format, and if it is a long word, it will read in single word format.

Example 1

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

Assuming the data in the matrix register is block-floating pseudo-single-precision `A`, LM0 is block-floating pseudo-single-precision `x`, and GRF0 is normal half-precision `y`, performs a pseudo-single-precision matrix-vector product FMA with `y` extended to single-precision (`Ax+y`) and writes the result to LM1.

Example 2

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

Assuming the data in the matrix register is block float half-precision `A`, LM0 is block float half-precision `x`, and GRF0 is normal half-precision `y`, performs a half-precision matrix-vector product FMA with `y` extended to single-precision (`Ax+y`) and writes the result to GRF0.

Example 3

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

Assuming the data in LM0 is single-precision `x`, GRF0 is single-precision `y`, and LM1 is single-precision `z`, performs a double-precision vector FMA with `x,y,z` extended to double-precision (`x*y+z`) and writes the result to GRF0.

Example 4

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

Assuming the data in LM0 is normal half-precision `x`, GRF0 is normal half-precision `y`, and LM1 is normal half-precision `z`, performs a half-precision vector FMA with `z` extended to single-precision (`x*y+z`) and writes the result to GRF0.

3.6.9.24 Input Shortening of Inputs

For the input operands of `vfma,mwrite` instructions that require normal half-precision values in the basic operation, it is possible to indicate a conversion from single-precision values. The operations that require normal half-precision values are half-precision vector multiply-add and (matrix transpose for) half-precision matrix write. These require long word half-precision values, so 2-long-word single-precision values will be input after rounding. This operation is called **precision shortening**. Precision shortening can be indicated by adding an `r` to the end of a double-long-word input operand.

Note that it is not possible to indicate precision shortening from one step higher precision for operands that require block-floating half-precision or (regardless of block-floating) single precision. To convert from single-precision values to block-floating half-precision values, you can use the `bf` input operand in ALU instructions described later. To convert from double-precision values to single-precision values, you need to add precision shortening to the output of MAU instructions – which will be described later– using `vpassa`, and then perform block floating-point conversion as needed.

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

Assuming the data in GRF0 is normal single-precision `x` and LM0 is normal half-precision `y`, performs a half-precision vector product (`x*y`) with `x` reduced to half-precision and writes the result to the T-register.

3.6.9.25 Precision Shortening of Output

By appending an `r` to the end of the MAU instruction opcode, it is possible to round the output value to one precision level lower than in the basic operation. This is also called **precision shortening** as in the case of inputs.

When precision shortening is applied, double-precision operations are reduced from double-precision to single-precision, and half-precision operations are reduced from single-precision to half-precision. The number of words used remains the same as in the basic operation, but the word length per word becomes half. Therefore, the output length for double-precision operations becomes a single word, and for half-precision operations, it becomes a long word.

Here is an example of precision shortening from double-precision to single-precision output.

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

Here is an example of precision shortening from half-precision to half-precision output. This example also uses precision extension for the C-port input.

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

3.6.9.26 Mask Flags Generated by MAU Instruction Expressions

The mask flag generated by the MAU instruction is the result of inverting the sign bit of the operation result. Therefore, when applying the mask, writing is performed where the operation result is non-negative and not performed where it is negative.

3.6.10 Matrix Register Write Instruction Expressions

3.6.10.1 dmwrite - Double-Precision Matrix Register Write

Performs write to matrix register for double-precision data.

Syntax

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

The operand `<src>` is a source PE operand with an access length of long word. When writing for matrix-vector multiply-add calculation, it holds a block-floating-point double-precision value; when writing for matrix transpose readout, it holds an ordinary double precision value. Note that there are no readout instructions without transposition.

The operand `$l(x|y)<addr>` is a destination matrix register operand. The `(x|y)` part specifies which matrix register to write to, referred to as `side` in the following effects. The `<addr>` part is the row number where writing starts and is incremented cycle by cycle to access consecutive rows without duplication.

Since double precision matrix data has 4 rows and 4 columns, it is possible to write an entire matrix data into a matrix register with one instruction.

Effects

```
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 mwrite/gmwrite - Single-Precision/Pseudo-Single-Precision Matrix Register Write

Writes single-precision data or pseudo-single-precision data to a matrix register. These are distinguished during matrix-vector multiply-add calculation, but not during read/write to matrix registers.

Syntax

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

There is no functional difference between `fmwrite` and `gmwrite`. Use `fmwrite` when issued concurrently with single precision matrix multiply-add instructions or single precision vector multiply-add instructions, and use `gmwrite` when issued concurrently with pseudo-single-precision matrix-vector multiply-add instructions.

The operand `<src>` is a source PE operand with an access length of either word or long word. In the case of word access, the second element of `src_data` in the following effects is zero-filled. When writing for matrix-vector multiply-add calculation, it holds a block floating-point single precision or pseudo-single-precision value; when writing for matrix transpose readout, it holds an ordinary single precision value.

The operand `$l(x|y)<addr>` is a destination matrix register operand. The `(x|y)` part specifies which matrix register to write to, referred to as `side` in the following effects. The `<addr>` part is the row number where writing starts and is incremented cycle by cycle to access consecutive rows without duplication.

Since single precision and pseudo-singles-precision matrix data have 8 rows and 8 columns, it is possible to write an entire matrix data into a matrix register with two instructions.

Effects

```
for cycle = 0:4
  forall chip,l2b,l1b,mab
    SingleWord 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, SingleWord)[(addr+cycle)%8][j] = src_data[j/2][
          j%2]
```

3.6.10.3 hmwrite - Half-Precision Matrix Register Write

Performs write to matrix register for half-precision data.

Syntax

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

The operand `<src>` is a source PE operand with an access length of either long word or double-long words. The access length must match that of the matrix register described below. When writing for matrix-vector multiply-add calculation, it holds a block-floating-point half-precision value; when writing for matrix transpose readout, it holds an ordinary half-precision value.

The operand `$(l|ll)(x|y)<addr>` is a destination matrix register operand. The `(l|ll)` part specifies the access length to the matrix register. `l` writes 1 row per cycle, and `ll` writes 2 rows per cycle; it is referred to as `wl` in the following effects. The `(x|y)` part specifies which matrix register to write to, referred to as `side` in the following effects. The `<addr>` part is the row number where the write starts and is incremented cycle by cycle to access consecutive rows without duplication. If `wl` is `ll`, then `<addr>` must be a multiple of 2. Since half-precision matrix data has 16 rows and 16 columns, it is possible to write an entire matrix data into a matrix register with two instructions by writing 2 rows per cycle.

Effects

```
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 Transposed Matrix Register Read Instruction Expressions

3.6.11.1 dmread - Double-Precision Transposed Matrix Register Read

Performs a transpose read from the matrix register for double-precision data.

Syntax

```
dmread $l(x|y)<addr> <dst_0> [<dst_1>..]
```

The syntax `$l(x|y)<addr>` represents the source matrix register operand. `(x|y)` specifies which matrix register to read from, referred to as `side` in the following explanation. `<addr>` is the column number where the read starts, and it is incremented cycle by cycle to access consecutive columns without duplication. Since double-precision matrix data is 4x4, it is possible to read the entire matrix data from the matrix register with a single instruction. Unlike `dmwrite`, the PE destination number corresponds to the row number, so that transposition can be performed by using `dmread` on data written by `dmwrite`.

The syntax `<dst_0> [<dst_1>..]` represents the PE destination operand. Although multiple PE memories can be written simultaneously, for simplicity, a single write destination is specified as `dst`. The read word length is long word. In other words, when writing to PE memory with a double-long-word access, the LSB-side 1 long word becomes 0.

No normalization or interpretation of floating-point numbers is performed during output. Instead, copying occurs as bit sequences. In particular, transposition of integer matrices with a width of long words can be performed using `dmwrite` and `dmread`.

Effects

```
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 - Single-Precision Transposed Matrix Register Read

Performs a transpose read from the matrix register for single-precision data.

Syntax

```
(f|g)mread $l(x|y)<addr> <dst_0> [<dst_1>..]
```

There is no functional difference between `fmread` and `gmread`. `fmread` should be used when issued simultaneously with single-precision matrix multiply-add instructions or single-precision vector multiply-add instructions, while `gmread` should be used when issued simultaneously with pseudo single-precision matrix-vector multiply-add instructions.

The syntax `$l(x|y)<addr>` represents the source matrix register operand. `(x|y)` specifies which matrix register to read from, referred to as `side` in the following explanation. `<addr>` is the column number where the read starts, and it is incremented cycle by cycle to access consecutive columns without duplication. Single-precision matrix data is 8x8, so it is possible to read the entire matrix data from the matrix register with two instructions. Unlike `(f|g)mwrite`, the PE destination number corresponds to the row number, so that transposition can be performed by using `(f|g)mread` on data written by `(f|g)mwrite`.

The syntax `<dst_0> [<dst_1>..]` represents the PE destination operand. Although multiple PE memories can be written simultaneously, for simplicity, a single write destination is specified as `dst`. The read word length is long word. In other words, when writing to PE memory with a double-long-word access, the LSB-side 1 long word becomes 0.

No normalization or interpretation of floating-point numbers is performed during output. Instead, copying occurs as bit sequences. In particular, transposition of integer matrices with a width of single words can be performed using `(f|g)mwrite` and `(f|g)mread`.

Effects

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

3.6.11.3 hmread - Half-Precision Transposed Matrix Register Read

Performs a transpose read from the matrix register for half-precision data.

Syntax

```
hmread $ll(x|y)<addr> <dst_0> [<dst_1>..]
```

The syntax `$ll(x|y)<addr>` represents the source matrix register operand. Unlike `hmwrite`, it always reads 2 columns per cycle. `(x|y)` specifies which matrix register to read from, referred to as `side` in the following explanation. `<addr>` is the column number where the read starts, and it is incremented cycle by cycle to access consecutive columns without duplication. Note that `<addr>` must be a multiple of 2. Since half-precision matrix data is 16x16, it is possible to read the entire matrix data from the matrix register with two instructions. Unlike `hmwrite`, the PE destination number corresponds to the row number, so that transposition can be performed by using `hmread` on data written by `hmwrite`.

The syntax `<dst_0> [<dst_1>..]` represents the PE destination operand. Although multiple PE memories can be written simultaneously, for simplicity, a single write destination is specified as `dst`. The read word length is 2 long words. In other words, when writing to PE memory with a long-word access, the LSB-side 1 long word of the value read from the matrix register is discarded.

No normalization or interpretation of floating-point numbers is performed during output. Instead, copying occurs as bit sequences. In particular, transposition of integer matrices with a width of half words can be performed using `hmwrite` and `hmread`.

Effects

```
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 Instruction Expressions

The ALU exists one per PE, and for most opcodes, it processes one long word per cycle.

At this time, when the precision is specified as single-precision, parallel operations are performed on two elements within a long word. When the precision is specified as half-precision, parallel operations are performed on four elements within a long word. Therefore, in terms of MAB, for half-precision, parallel operations can be performed on 16 elements per cycle.

In particular, for some opcodes that require high throughput, 2 long words can be processed per cycle.

The generalized syntax for ALU instructions is as follows.

```
  [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> represents the opcode, excluding zero output instruction zero and immediate value commands imm.

<dst_0> [<dst_1>..] is the write destination PE operand. This can include mask register writes (Section 3.6.1.13).

Zero output instruction and immediate value instruction do not take a read source PE operand. Instead, immediate value commands use an immediate value payload (Section 3.2.2) as an input operand. The [u] option for immediate value commands is described in Section 3.6.12.3.

Opcodes other than zero output and immediate value instructions are as follows: They are either 1-input or 2-input. <src_x> is the first input read source PE operand, and <src_y> is the second input read source PE operand. <src_x> and <src_y> can be the same. Only <src_x> can specify a fixed value input operand (Section 3.6.1.20). The leading [u] is an unsigned operation option, and [(d|f|g|h|l|i|s)] is a precision specifier. d, f, g, h represent floating-point double, single, pseudo-single, and half precision, respectively. l, i, s represent integer double, single, and half precision, respectively. The opcode determines whether the unsigned operation option is accepted and which precision specifications are allowed. For opcodes that accept the unsigned operation option, specifying the option is called **unsigned mode**, while not specifying it is called **signed mode**.

The list of opcodes is shown in Table 3.9.

The "Operation Type" column indicates the possible precision specifications. The actual precision specifications allowed for each operation type are listed in Table 3.10. For example, msl has an untyped operation type and does not allow precision specifications, while inc has an integer operation type and only allows l, i, s. The output of instructions that copy inputs, such as passa, is not affected by the precision specification. However, note that the generated mask flag is affected.

The "Double-long-Word Operation" column indicates whether the instruction can operate on double-long-word data. If it is "no", then the LSB side of the 2-word output from the ALU will be the same as the LSB side of the first input. If precision shortening (Section 3.6.12.19) is enabled for the first input, this LSB side will be zero. The behavior when it is "yes" is described in each instruction's section.

The "Unsigned Option" column indicates whether the u option is accepted. For imm instructions, the meaning of the u option is different from unsigned operations, so it is marked as "no". The differences in behavior between signed and unsigned modes for opcodes that accept the u option are described in each instruction's section.

Table 3.9 Table of ALU Instruction Opcodes. In output colum, x is the first input operand and y is the second input operand.

Opcode	Calc. Type	#Inputs	DLW Op.	Unsigned Opt.	Output
`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) $= 0$ else -0
`relu0`	float	2	no	no	same as `relu`
`relu1`	float	2	no	no	y if (2nd MSB of x) $= 0$ else -0
`relu2`	float	2	no	no	y if (3rd MSB of x) $= 0$ else -0
`relu3`	float	2	no	no	y if (4th MSB of x) $= 0$ else -0
`lrelud`	float	2	no	no	y if $x \geq 0$ else $y/2$
`lreluo`	float	2	no	no	y if $x \geq 0$ else $y/8$
`ilrelud`	float	2	no	no	y if $x \geq 0$ else increment exp of y

Table 3.10 Acceptable precisions to ALU operation types that appear in the Table 3.9. None means no precision is acceptable

Operation Type	Acceptable Precisions
int	`l, i, s`
float	`d, f, h`
both	`d, f, h, l, i, s`
bfn	`d, f, g, h`
bfe	`h`
untyped	none

3.6.12.1 Mask Flags Generated by ALU Instruction Expressions

The mask flags generated by the ALU instruction syntax are shown in Table 3.11.

The "Unsigned" column indicates whether the opcode has an unsigned option. "`-`" indicates that there is no unsigned option for this opcode. "false" means that the mask flag generation applies to signed mode and "true" means that the mask flag generation applies to unsigned mode. "both" means that the unsigned option exists, but it does not affect the generation of mask flags.

The "Condition for flag becomes 1" column indicates that the mask flag becomes 1 when the condition is met for each word in the output long word, and 0 otherwise. Note that the mask flags are always generated at a rate of 4 bits per cycle, and if the precision specification is long word, one word's result is duplicated into 4 bits, while for single word precision, two words' results are each duplicated into 2 bits.

The `passa` instruction and floating-point mode `max/min` instructions have slightly more complex conditions, which are described in detail in Sections 3.6.12.5 and 3.6.12.13, respectively. Note that each entry in the mask register is 16 bits in size, which can accommodate the number of bits generated by an immediate value command. However, it is not possible to directly write arbitrary values into the mask register using an immediate value command.

Table 3.11 Mask flags generated by ALU instructions and whether the instruction accepts opcode and sign options.

Opcode	Unsigned	Condition for flag becomes 1
`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.

The following sections will provide detailed explanations for each operand.

3.6.12.2 zero - Zero Output Instruction

The `zero` instruction takes no input and always outputs double-long-word zero.

Example

```
zero $llr0v
```

Zero-fills registers `$r0` through `$r15`, a total of 8 long words.

3.6.12.3 imm - Immediate Value Instructions

The `imm` instruction outputs a 2-long-word value composed of the word immediate literals described in Section 3.2.2, arranged in the following manner:

If the `u` option is not present, four literal words are concatenated to form 2 long words. If the `u` option is present, the 2nd and 4th words from the MSB side are replaced with zero. If we denote a single-word literal as `x` and the single-word of zero as `o`, then the output will be `xxxx` without the `u` option, and `xox o` with the `u` option.

One use case for the `u` option is directly generating a double-precision word with the lower 32 bits of the mantissa set to zero.

It is not possible to issue an immediate value command simultaneously with an instruction that accesses LM0 *6.

Example

```
immu s"1" $llr0
```

The data in `$llr0` can be written as 0x0001_0001_0000_0000_0001_0001_0000_0000 when divided into half-precision units starting from the MSB side.

3.6.12.4 msl, msr - Inter-PE Circular Shift Instruction

Each PE transfers one long word to the adjacent numbered PE per cycle. The `msl` instruction transfers input from PE0 to output at PE1, and similarly for subsequent PEs: $1 \to 2$, $2 \to 3$, $3 \to 0$. The `msr` instruction performs this transfer in the reverse direction.

Example

```
msl $lr0v $lr0v
```

Copies the 4 long words in GRF0 of PE0, 1, 2, and 3 to GRF0 of PE1, 2, 3, and 0, respectively.

*6 This is because the address of LM0 is shared with the immediate value in the instruction bits

3.6.12.5 passa - Copy Instruction

The `passa` instruction copies the input 2 long words unchanged to the output.

Due to the 2-long-word operation, not only the MSB-side long word but also the LSB-side long word is copied from the input, but the LSB side is ignored in mask flag generation, and only the 1 long word, 2 single words, or 4 half words included in the MSB-side long word are referenced.

Example

```
lpassa $llr0v $lls0v
```

Copies a total of 8 long words from GRF0 to GRF1.

3.6.12.6 inc, dec - Increment / Decrement Instruction

The `inc` and `dec` instructions perform integer increment and decrement operations, respectively.

Both input and output are treated as signed integers in signed mode or unsigned integers in unsigned mode.

In both modes, overflow results in wrapping around.

Example

```
zero $nowrite
sdec $aluf $lr0v
```

A wrap-around occurs in the negative direction, and GRF0 is written with four longwords of all ones.

3.6.12.7 not - Bit Reverse Instruction

The `not` instruction performs a bitwise NOT operation.

Example

```
zero $nowrite
lnot $aluf $lr0v
```

Four long words of all ones are written to GRF0. Note that this `lnot` is not the logical NOT instruction (Section 3.6.12.8), but rather a bitwise NOT operation with long precision specification.

3.6.12.8 lnot - Logical Not Instruction

The `lnot` instruction returns the logical NOT of the input. In other words, if the input is all zeros, it returns 1; otherwise, it returns 0.

Example

```
ilnot $subpeid $lr0
```

The value in `$lr0` is two single-precision integers with a value of 1 concatenated together, but only for PE 0. For all other PEs, the value is all zeros.

3.6.12.9 rsqrt - Approximate Reciprocal Square Root Instruction

The `rsqrt` instruction returns an approximate value of the reciprocal square root. The sign bit of the input is ignored. The precision is approximately 5 bits. If a more accurate value is required, this result can be used as an initial value for further refinement using methods such as Newton's method.

3.6.12.10 floor - Floor Instruction

The `floor` instruction interprets the input as a floating-point number and rounds it to the nearest floating-point number with zero mantissa part. The rounding is done towards negative infinity. If infinity or zero is input, the input is output unchanged. In other cases where the result is zero, the mantissa is zero-flushed.

3.6.12.11 ftoi - Conversion from Floating-Point to Integer

The `ftoi` instruction interprets the input as a floating-point number and rounds it to the nearest integer. The rounding is done towards zero. In unsigned mode, the absolute value of the input is rounded to an unsigned integer. If the result exceeds the maximum integer value, including when the input is infinity, the result is clipped to that value.

Note that there is no instruction for converting an integer to a floating-point number (`itof`). Such conversions can be achieved using a sequence of instructions as described in Section 3.7.1.

3.6.12.12 bfe, bfn - Block Floating Point Conversion Instructions

For the matrix-vector multiply-add operation, all inputs for the product part must be **block floating point (BF)** normalized using a BF normalization instruction corresponding to the precision of the operation. This is roughly equivalent to aligning the exponents within each of the two inputs in the inner product circuit. Therefore, unlike other ALU instructions, the BF normalization instruction does not perform an independent operation for each element. The opcode syntax is as follows:

Syntax of Opcode

```
  dbfn
| fbfn
| gbfn
| hbfn/<n>
| hbfe/<n>
```

The prefix `d, f, g, h` represents double precision, single precision, pseudo single precision, and half precision, respectively.

The size of the inner product circuit is 4 words for double precision and single precision, 8 words for pseudo single precision, and 16 words for half precision. BF normalization is performed in units of these word counts, referred to as **blocks** in this section.

For double precision, BF normalization is performed on a block consisting of one long word from the MSB side of each PE.

For single precision, BF normalization is performed on two blocks for a long word of the MSB side: one word consisting of the MSB-side and the other word consisting of the LSB-side. Note that in single-precision MAU operations, only the MSB-side word is used, so when using the LSB-side word, it must be written to LM or another register and read back as a word.

For pseudo single precision, BF normalization is performed on a block consisting of one long word from the MSB side of each PE.

Half-precision BF normalization instructions `hbfn` and `hbfe` both read 2 long words per cycle from each PE, totaling 32 half-precision words as input. BF normalization is performed on two blocks: the long word on the MSB side and the long word on the LSB side.

The throughput of BF normalization per PE is for all precisions except half precision: 1 long word/-cycle, and for half precision: 2 long words/cycle.

The `hbfe` instruction uses an **extended representation** to prevent underflow for numbers with relatively small exponents within a block. This extended representation is similar to the denormalized number in standard floating-point representation. In contrast, the `hbfn` instruction does not use the extended representation. The <n> parameter specifies the length of the mantissa after conversion and can take values from 6 to 9. Using a smaller value reduces the precision but is expected to result in lower power consumption.

Note that for all precisions except half precision, there are no extended representation or mantissa length specification features.

3.6.12.13 max, min - Maximum, Minimum Instructions

The `max` and `min` instructions are both two-input operations, with the `max` instruction outputting the not smaller value and the `min` instruction outputting the not larger value. In this section, we will refer to the first input as x and the second input as y.

The mask flag is set to 1 when x is selected. This includes cases where x and y are equal.

Both floating-point and integer precisions can be specified, resulting in different behavior.

When an integer precision is specified, a standard integer comparison is performed. There is also an unsigned option that allows the input to be interpreted as either signed or unsigned.

When a floating-point precision is specified, both the `max` and `min` instructions behave similarly to signed integer comparisons but 0 is treated the same value.

The following describes the details:

Unless otherwise noted, the mask flag is set to 1 when x is output. If all bits of x and y are equal, that value is output and the mask flag is set to 1. If x and y do not have all identical bits but are both floating-point zeros (i.e., their exponent parts are all zero), x is output. If x and y do not have all identical bits but are both infinite with the same sign, the mantissa parts are compared to determine the output. In all other cases, a standard floating-point comparison is performed to determine the output.

3.6.12.14 packbit - Packing Sign Bit Instruction

The `packbit` instruction takes the first input as x and the second input as y, and returns the bit-wise logical OR of $x \ll 1$ and the MSB of y.

This instruction can be used to reduce data volume in cases where only the sign information of a value is necessary, such as with ReLU-related instructions that require only the first argument's sign.

Example

```
hpackbit $msb1 $lr0v $nowrite
hpackbit $aluf $lr8v $nowrite
hpackbit $aluf $lr16v $nowrite
hpackbit $aluf $lr24v $ls0v
```

Packs the sign bits of four half-precision numbers into one half-word. The `$msb1` value can be effectively treated as a zero initial value by shifting it to the left.

3.6.12.15 and, or, xor - Bitwise Logical AND, OR, Exclusive OR Instructions

The `and`, `or`, and `xor` instructions perform bit-wise logical AND, OR, and exclusive OR operations, respectively.

Since these are bit-wise operations, the output is independent of the precision specification. However, the mask flag is affected by the precision specification.

3.6.12.16 add, sub - Addition, Subtraction Instructions

The `add` and `sub` instructions perform integer addition and subtraction operations, respectively.

Both inputs and outputs are treated as signed integers in signed mode and unsigned integers in unsigned mode.

In both modes, if an overflow occurs, the value wraps around.

3.6.12.17 lsl, lsr, bsl, bsr - Shift Instructions

The `lsl` instruction is logical left shift, the `lsr` instruction is arithmetic right shift or logical right shift, the `bsl` instruction is barrel left shift, and the `bsr` instruction is barrel right shift.

The second input is used as the shift amount.

The `lsr` instruction performs an arithmetic right shift in signed mode and a logical right shift in unsigned mode. The other shift instructions do not have an unsigned option.

For logical shifts, the bits shifted out are replaced with zeros. For arithmetic right shifts, the bits shifted out are replaced with the sign bit of the input. For barrel shifts, the bits shifted out wrap around to fill in the empty bits.

When the shift amount exceeds the word length, the following behavior occurs: Let n be the word length (16, 32, or 64 depending on precision), and s be the original shift amount. First, compute $s_0 := s \bmod 2n$, effectively ignoring any bits above the number of bits required to represent a shift amount equal to the word length. Then, if $s_0 < n$, perform the shift as defined above. If $n \leq s_0$, for barrel shifts, shift by $s_0 - n$. For arithmetic and logical shifts, shift by n, effectively shifting out all bits (resulting in either all zeros or all ones).

3.6.12.18 ReLU-related Instructions

The ReLU-related instructions are all two-input operations, defined in Table 3.12.

All input and output values are floating-point numbers.

The `relu0` instruction is an alias for the `relu` instruction.

Table 3.12 The definition of ReLU-related Instructions. x is the first input and y is the second output.

Operand	Definition
`relu/relu0`	Returns y when the MSB of x is 0, and -0 when it is 1.
`relu1`	Returns y when the second bit from the top of x is 0, and -0 when it is 1.
`relu2`	Returns y when the third bit from the top of x is 0, and -0 when it is 1.
`relu3`	Returns y when the forth bit from the top of x is 0, and -0 when it is 1.
`lrelud`	Returns y when the MSB of x is 0, and $y/2$ when it is 1.
`lreluo`	Returns y when the MSB of x is 0, and $y/8$ when it is 1.
`ilrelud`	Returns y when the MSB of x is 0, and the result of incrementing the exponent part of y when it is 1.

The `l` in `lrelud`, `lreluo`, and `ilrelud` instructions is stands for ”leaky”. The `i` in the `ilrelud` instruction stands for ”inverse”.

The `lrelud` and `lreluo` instructions may result in underflow. In this case, they return a value with a negative sign bit and zero exponent and mantissa.

The `ilrelud` instruction may result in overflow. In this case, it returns a value with all bits of the exponent set to 1 and the same sign and mantissa as the input. Note that when x has an MSB of 1, the exponent of y is incremented even if the original exponent was zero, resulting in a non-zero output for $y = 0.0$.

These definitions are designed to compute both the forward and backward passes of ReLU-like functions. Since the MSB of a floating-point number represents the sign bit, the `relu` instruction can be approximated as returning y if x is non-negative and 0 otherwise. Let w be the result of applying the ReLU function to an input v. This can be computed using the `relu` instruction as $w =$`relu`(v, v). To compute the backward pass of the ReLU function, i.e., to compute the gradient with respect to v given the gradient with respect to w, denoted w', one can use either `relu`(v, w') or `relu`(w, w').

The latter is possible because the result of `relu`(v, v) preserves the sign of v.

3.6.12.19 Precision Shortening from Single-Precision to Half-Precision for the Inputs of the ALU

When an ALU instruction expects half-precision floating-point numbers as input operands, it is possible to indicate that single-precision floating-point numbers should be rounded and used as input. This is referred to as **precision shortening**.

By appending the `r` option to the end of an operand, 2 long words are interpreted as four single-precision floating-point numbers, which are then rounded to half-precision floating-point numbers. The resulting four values are placed in the MSB side long word of double-long-word, and the LSB side long word is set to zero.

For 2-input opcodes, it is possible to independently decide whether to append the `r` option to the first input or the second input.

Note that precision shortening occurs at each input port of the ALU, so appending the `r` option to one operand does not affect inputs to other ports. For example, the following assembly code is valid, and rounding only occurs for the second input to the ALU:

```
sor $llm0v $llm0vr $nowrite; hvfma $llm0v $llm0v $llm0v $nowrite; l1bmm@0 $llm0v $lb0
```

Example

```
hrsqrt $llr0vr $lm0v
```

Each cycle, read 2 long words from GRF0, interpret them as 4 single-precision numbers, shorten the precision of them to 4 half-precision numbers, and then execute the approximate reciprocal of square root instruction.

3.6.13 wait - Make PE instructions Wait for MV Instructions

The number of cycles required for an MV instruction is not fixed and can vary, so PE instructions are designed to wait for the corresponding MV instruction using a tag.

In the step where a `wait` instruction is written, the issuing of subsequent instructions is halted until the completion notification from the corresponding MV instruction is received.

The `wait` instruction cannot stand alone as a PE command and must be co-issued with another PE instruction.

Note that since PE and MV instructions share a single instruction stream, not only subsequent PE instructions but also subsequent MV instructions will not be issued until the wait condition is resolved.

Syntax

```
wait <tag>
```

The `<tag>` refers to a tag as defined in Section 3.2.3.

Effects

Halt the issuance of instructions at the current point, and continue sending NOPs in place of subsequent instructions until the completion notification from an MV instruction with tag `<tag>` is received.

Errors

- If no other PE instruction is co-issued with the `wait` instruction, that results in an error
- If a tag number of 0 is specified, that results in an error.

Examples

```
mvp/n64i01 $lc0@.0 $d0
l2bmrdfadd $lb0 $lc0; wait i01
```

Start L2BM → DRAM parallel individual transfer and pause write from L1BM to L2BM until the transfer is completed.

3.7 Examples of MN-Core 2 Assembly Programs

3.7.1 Conversion from Integer to Floating-point Value

The following assembly code converts eight single-word integer values (in this case, 100 to 107) to single-precision floating-point numbers.

```
# Set input integers `n` for itof
imm i"100" $s0/1000
imm i"101" $s1/1000
(...omitted...)
imm i"107" $s7/1000
imm f"8388608" $lr0/1000 # 8388608 is 2**23. Its weight for the LSB of mantissa is 1.0
ior $ls0v $aluf $nowrite # Create 2**23 + 1.0 * n = 2**23 + n in float
fvadd $aluf -$lr0 $ls0v # Calculate (2 ** 23 + n) - 2**23, equals to n
```

3.7.2 Multiple Input to Computation Unit from PE Memory Read Operand

The following assembly code reads double-long-word per cycle from LM0 and uses the data as input for two operations: integer subtraction and copying to L1BM.

Since the copy operation to L1BM is a double-long-word, 16x1 individual transfer (as described in Section 3.6.8.11), all 32 long words read from 0th MAB in one step are written to L1BM. On the other

hand, the integer subtraction operation is a long-word operation (as shown in Section 3.6.12), so only the MSB side of the 2-word data path between LM0 and ALU is used. In terms of single word access, this means that the following words are accessed: `$m0,$m1` in cycle 0, `$m4,$m5` in cycle 1, `$m8,$m9` in cycle 2, and `$m12,$m13` in cycle 3. This is equivalent to accessing the region denoted by the operand notation `$lm0v4`. In other words, for the `isub` instruction, `isub $lr0v $lm0v4 $ln0v` would write the same value to `$ln0v`. However, if we apply this change to the following assembly code, the parallel execution condition " If multiple instruction expressions read from the same PE operand, they all access the same region across all cycles" (described in Section 3.6.4) is not satisfied between the `l1bmm@0` instruction's LM0 read operand `$llm0v` and the other instructions, resulting in an error.

```
isub $lr0v $llm0v $ln0v; l1bmm@0 $llm0v $llb0
```

3.7.3 Double-long-word Output from Computation Unit to PE Memory Write Operand

The following assembly code reads data from LM0 at a rate of 2 long words per cycle, copies it to LM1 and GRF1, and writes the flag value output by ALU to entry 1 of the mask register.

The `passa` instruction (described in Section 3.6.12.5) is a double-long-word operation, so data from `$m0` to `$m15` is completely copied to `$s0` to `$s15`. On the other hand, LM1 is specified as a long word, so only the MSB side of ALU's 2-word output is written in each cycle. This corresponds to the read region `$lm0, $lm4, $lm8, $lm12`. In mask flag generation, the MSB side of the long word data is also referenced, so in each cycle, if all bits of `$lm0, $lm4, $lm8, $lm12` are zero, `0b1111` is written to the corresponding location in `$omr1`, otherwise `0b0000` is written.

Here, `0b1111` and so on represent 4 bits in the word direction of a mask register entry (described in Section 3.6.2).

```
lpassa $llm0v $ln0v $lls0v $omr1
```

Chapter 4

Details of MN-Core 2 Floating-Point Operations

This chapter provides details on the floating-point operations of MN-Core 2.

With the information provided in this chapter, it is possible to implement an emulator that matches the actual RRN and MAU at the bit level, except for half-precision matrix-vector multiply-add with extended representation (Section 3.6.12.12).

4.1 Outpu Normalization

The behavior when normalizing the output is the same for both RRN and MAU. This means that if the output is infinity or zero, i.e., all bits of the exponent are 1 or all bits are 0, the mantissa is set to all zeros. Furthermore, if the output is zero, the sign bit is also set to 0.

4.2 RNN

MN-Core 2 can execute reduction instructions in three places: MV instructions (Section 3.5), L2BM instructions (Section 3.6.7), and L1BM instructions (Section 3.6.8).

L1BM instructions have two types of reductions: 4x4 reduction, which performs a 4-element reduction, and 16x1 reduction, which performs a 16-element reduction. The 16-element reduction is implemented by performing the 4-element reduction twice. Floating-point addition (fadd) and floating-point maximum/minimum (max/min) have different handling, so they are explained separately.

As mentioned in Section 3.6.8.5, L1BM instructions' half-precision floating-point reductions are implemented using single-precision floating-point reductions. Therefore, L1BM instructions essentially only have single-precision and double-precision floating-point reductions. Note that precision shortening is only applicable to single-precision floating-point operations.

In contrast to L1BM instructions, MV instructions and L2BM instructions have separate circuits for half-precision, single-precision, and double-precision floating-point operations. L2BM instructions do not use smaller-element-number reduction circuits in multiple stages to perform larger-element-number reductions. MV instructions have separate circuits for 2-element reductions within a group and 4-element reductions between groups.

The L2BM $\rightarrow$ DRAM inter-group reduction instruction (Section 3.5.8.21) and the L2BM $\rightarrow$ PDM inter-group reduction instruction (Section 3.5.8.19) use these circuits to perform two reductions, achieving 8-element reductions.

4.2.1 RRN Floating Point Addition

For the floating-point addition instruction, we first describe the behavior of L1BM instructions for 4-element reduction. For each element's mantissa, a hidden bit is appended to the MSB, and three zeros are appended to the LSB (least significant bit). Then, the exponent parts are compared, and

the mantissas of elements with non-maximum exponents are right-shifted by their differences. If the difference in exponents exceeds 4 and shifts out the three bits added at the LSB, rounding to nearest even is performed. Next, the mantissas are added and reduced, followed by rounding to nearest even to an appropriate precision. Here, "appropriate precision" means single-precision when precision shortening is specified for 16-element reduction in the first stage, or the output precision otherwise. The result of rounding is then normalized to produce the output of the 4-element reduction. Note that rounding to nearest even may cause a carry, resulting in a final exponent that can be one more than maximum exponent added two found during comparison (accounting for four elements).

For L1BM instructions with 16-element reduction, this process is performed twice, producing the output. Note that even for 16-element outputs, intermediate calculations involve normal floating-point numbers of double precision or single precision. The complexity in rounding precisions implies that when precision shortening is specified, results are rounded directly to half-precision in a single step rather than first being rounded to single precision and then half-precision.

MV instructions and L2BM instructions behave similarly to L1BM's 4-element reduction, except for differences in element numbers, the existence of half-precision shortening, and the lack of precision shortening. Specifically, they append hidden bits and three zeros at the LSB, align exponents, round, add, round again, and normalize.

In MV instructions with 8-element reductions, a two-stage reduction is performed similar to L1BM's 16-element reduction. Again, intermediate calculations involve normal floating-point numbers of some precision.

4.2.2 RNN Floating Point Maximum Value and Minimum Value

For floating-point maximum and minimum instructions, inputs are treated as normalized numbers without special handling for infinity or zero. This is equivalent to interpreting the input values as sign-magnitude integers, combining the exponent and mantissa parts, without distinction. Here, positive zero is considered greater than negative zero.

In L1BM instructions, when precision shortening is specified, the result of the shortening is rounded to half-precision using round-to-nearest-even. For 16-element reductions obtained by performing 4-element reductions twice, rounding occurs only once in the second stage.

Unlike floating-point addition reductions or MAU instructions, no normalization is performed on the result. This holds even when shortening reduction is applied.

MV instructions and L2BM instructions behave similarly to L1BM's 4-element reduction, except for differences in element numbers, the existence of half-precision reductions, and the lack of precision shortening. Specifically, they interpret inputs as sign-magnitude integers, compare them, and output the maximum or minimum without normalization.

4.3 Vector Multipyl-Add

For MAU vector multiply-add operations, we describe the basic operation (Section 3.6.9.1) in detail.

To generalize, we assume a floating-point number format with mantissa length m, as described in Section 4.3. For half, single, and double precision, $m = 9, 23, 52$ respectively.

We denote the multiply-add operation for one output element as $ab + c$. The inputs to the product a, b are floating-point numbers represented as follows:

$$a = 2^{e^{(a)}}(-1)^{s^{(a)}}\left(1 + \sum_{j=1}^{m} 2^{-j} A_j\right) \tag{4.1}$$

$$b = 2^{e^{(b)}}(-1)^{s^{(b)}}\left(1 + \sum_{j=1}^{m} 2^{-j} B_j\right). \tag{4.2}$$

where we use notation for the exponent $e^{(a)}, e^{(b)}$, sign bit $s^{(a)}, s^{(b)} \in \{0,1\}$, and mantissa bits $A_j, B_j \in \{0,1\}$. The 1 before the sum of mantissas correspond to hidden bit.

The exact product is given by:

$$ab = 2^{e^{(a)}+e^{(b)}}(-1)^{s^{(a)}+s^{(b)}}\left(1+\sum_{j=1}^{m}2^{-j}A_j+\sum_{j=1}^{m}2^{-j}B_j+P\right) \tag{4.3}$$

$$P := \sum_{1\leq j,k\leq m} p_{j,k} \tag{4.4}$$

$$p_{j,k} := 2^{-(j+k)}A_jB_k \tag{4.5}$$

For half precision, P is computed exactly. For single and double precision, a partial multiply-add $P' := r + \sum_{(j,k)\in D} p_{j,k}$ is used instead of adding $p_{j,k}$. Here, D is the range of partial product for each precision, r is the correction term determined by digits that are not added A_j, B_k s.t. $(j,k) \notin D$

In single precision:

$$D := \{(j,k) \mid 1 \leq j \leq 18 \vee 1 \leq k \leq 18\} \tag{4.6}$$

$$r := \begin{cases} 0 & \text{if } \sum_{(j,k)\notin D} p_{j,k} = 0 \\ 2^{-38} & \text{otherwise} \end{cases} \tag{4.7}$$

In double precision:

$$D := \{(j,k) \mid 1 \leq j \leq 36 \vee 1 \leq k \leq 36\} \tag{4.8}$$

$$r := \begin{cases} 0 & \text{if } \sum_{(j,k)\notin D} p_{j,k} = 0 \\ 2^{-74} & \text{otherwise} \end{cases} \tag{4.9}$$

In other words, some bits of products of input mantissas are omitted, namely $23 - 18 = 5$ bits for single-precision and $52 - 36 = 16$ bits for double-precision, and if the output is zero then that is treated as zero, otherwise the output is treated as if the top bit of the omitted mantissa was 1.

Regardless of the precision, after calculating the product, the following steps produce the same result:

1. Shift c by the exponent difference with the product and add it with an infinite mantissa length.
2. Round the result to the nearest even value to fit the output floating-point format's mantissa length. Note that this rounding may cause the exponent to change due to carry-over.
3. If the exponent falls within the range of infinity or zero in the output floating-point format, adjust the exponent accordingly.
4. Normalize and output the result.

Depending on the options used, precision expansion (lower input precision than the basic operation) and precision shortening (higher output precision than the basic operation) may occur.

Precision expansion is handled simply by converting to the input precision of the basic operation and then performing the basic operation. Since the conversion to a higher precision can be done exactly, there's no need for special consideration.

Precision shortening, on the other hand, involves rounding directly to the reduced precision instead of first rounding to the output precision of the basic operation and then reducing the precision.

In single and double precision, since replacing part of the product with the correction term only affects bits far below the top bit (due to the product of hidden bits), it would not cause any problems practically. "Bits far below the top bit" means $38 - 23 = 15$ bits down for single precision and $74 - 52 = 22$ bits down for double precision. The difference from the exact value can be confirmed in cases where large cancellations occur.

```
imm f"1099511627776.0" $lr0/1000 # 2**40
imm f"1048577.0" $nowrite # 2**20+1
fvfma $aluf $aluf -$lr0 $ls0/1000 # exact: 2**21+1
d get $ls0n0c0b0m0p0 1 # printed: 0x4a000010
```

The result of the `fvfma` calculation is $(2^{20}+1)^2 - 2^{40}$. The exact value is $2^{21}+1$, which in single precision is `0x4a000004`. However, what is actually written to `$ls0` is `0x4a000010`, resulting in a difference from the exact value.

4.4 Block-floating Conversion

The matrix-vector multiply-add operation requires both inputs of the product to be in block-floating-point format. Therefore, we will first describe the details of the block-floating-point format and the instructions for converting to this format.

The block-floating-point format is an intermediate format that aligns the exponents within a fixed number of elements per precision, eliminating the need for exponent difference shifts during multiply-add operations. Table 4.1 shows the block size and the precision of the original floating-point numbers for each precision in the block-floating-point format.

Table 4.1 The block size of block-floating point format and source precision of floating-point number

precision	block size	precision of normal floating point number before conversion
half precision	16	half precision
pseudo single precision	8	single precision
single precision	4	single precision
double precision	4	double precision

If the exponent values within a block do not match, it becomes an invalid block-floating-point value, except for the case of half-precision extended representation described later. The matched exponent value is hereinafter referred to as the **common exponent**. The behavior when using an invalid block-floating-point value as input to the product in matrix-vector multiply-add operations is undefined.

The interpretation of a valid block-floating-point value, excluding hidden bit representation, is the same as that of the original normal floating-point number before conversion. This includes the length of the exponent and mantissa. For example, if the sign is 0, the unbiased exponent is 0, and the mantissa has only its MSB set to 1 with all other bits being 0, it represents 1.0. Additionally, representing 0.0 is possible by setting all bits of the mantissa to 0. However, in the interpretation of pseudo-single-precision block-floating-point values, the LSB side 5 bits of the mantissa are ignored and treated as 0.

The half-precision block-floating-point format has an extended representation similar to denormalized numbers in normal floating-point numbers. In this extended representation, a value with an exponent that is not the common exponent but all zeros is permitted. Such a value is considered to have an exponent that is -6 from the common exponent. For example, when the block's common exponent is 0x1f (unbiased exponent of 0), a half-precision block-floating-point value with a sign of 0, all-zero exponent, and mantissa of 1 represents 2^{-14}. The purpose of the extended representation is to suppress underflow during conversion to block-floating-point values. The extended representation is not supported in pseudo-single precision, single precision, or double precision.

Since the half-precision block-floating-point format without an extended representation is completely contained within and has the same interpretation as that with an extended representation, there is no need for mode switching based on the presence of an extended representation during matrix-vector multiply-add operations.

The following describes the specific steps for block-floating-point conversion. Let n be the block size.

Pseudo-single precision, single precision, and double precision block-floating-point conversions are performed according to the following steps:

1. Identify the maximum exponent among the n input values and the element that has this maximum exponent.
2. If an element with the maximum exponent has all bits set in its mantissa, increment the maximum exponent by 1. This is due to carry from round-to-nearest-even rounding.
3. Perform exception handling for infinity and zero:

(a) If the common exponent is greater than or equal to the value representing infinity, output all elements with their signs unchanged, exponents set to infinity, and mantissas set to all zeros.
(b) Regardless of the common exponent, if an input element has an all-zero exponent, output that element with its sign unchanged, exponent set to the common exponent, and mantissa set to all zeros.
(c) If all input elements have all-zero exponents, output all elements with their signs unchanged, exponents set to all zeros, and mantissas set to all zeros.

4. Calculate the difference between the common exponent and each element's exponent.
5. For each element, add a hidden bit to its mantissa, downshift by the exponent difference, and perform round-to-nearest-even rounding. The mantissa length will always be shorter after conversion, so rounding is always performed. Additionally, if the exponent difference is zero, align the bits so that the added hidden bit becomes the MSB of the mantissa. And for pseudo-single precision, the MSB side 18 bits out of the 23 bits in the format will be used. The rest of 5 bits will be 0.
6. For elements whose mantissas do not underflow (i.e., the rounded result is not all zeros), output them with their signs unchanged, exponents set to the common exponent, and mantissas set to the rounded results.
7. For elements whose mantissas underflow, output them with their signs unchanged, exponents set to the common exponent, and mantissas set to all zeros.

Next, we describe the conversion to half-precision block-floating-point values. As described in Section 3.6.12.12, half-precision block-floating-point conversion can specify the length of the mantissa after conversion to be between 6 and 9 bits, regardless of whether an extended representation is used or not. Let b be the difference between 9 and the specified mantissa length after conversion, i.e., the amount by which the mantissa length will decrease.

Half-precision block-floating-point conversion without extended representation is performed according to the following steps:

1. Identify the maximum exponent among the n input values and the element that has this maximum exponent.
2. If an element with the maximum exponent has all bits set in its mantissa excluding the LSB-side b bits, increment the maximum exponent by 1. This is due to carry from round-to-nearest-even rounding.
3. Add b to the maximum exponent to obtain the common exponent.
4. Perform exception handling for infinity and zero as in the case of pseudo-single precision, single precision, and double precision block-floating-point conversion.

Half-precision block-floating-point conversion with extended representation is performed according to the following steps:

1. Calculate the common exponent and perform exception handling for infinity and zero up to this point as in the case of half-precision block-floating-point conversion without extended representation.
2. Calculate the difference between the common exponent and each element's exponent.
3. For each element, if the exponent difference is greater than or equal to $6 + b$, set a flag. However, do not set this flag if the exponent difference is exactly $6 + b$ and the mantissa excluding the LSB-side b bits is all ones.
4. For elements with an unset flag, add a hidden bit to their mantissas, downshift by the exponent difference, and perform round-to-nearest-even rounding.
5. For elements with a set flag, add a hidden bit to their mantissas, downshift by the exponent difference minus 6, and perform round-to-nearest-even rounding.
6. Output elements whose mantissas do not underflow with their signs unchanged, mantissas set to the rounded results, and exponents set to either the common exponent (if the flag is unset) or all zeros (if the flag is set).
7. Output elements whose mantissas underflow with their signs unchanged, exponents and mantissas set to all zeros.

4.5 Matrix-Vector Multiply-Add

This section provides a detailed description of the basic operation of matrix-vector multiply-add calculation for MAU (described in Section 3.6.9.1).

For generalization, we assume a block-floating-point format with a block size of n and a mantissa length of m, as described in Section 4.4. The block sizes and mantissa lengths for half-precision, pseudo-single precision, single precision, and double precision are $16, 8, 4, 4$ and $m = 9, 18, 23, 52$, respectively. We denote the multiply-add calculation for one output element as $(\sum_{i=1}^{n} a_i b_i) + c$, where the input products a_i, b_i are block-floating-point values represented as follows:

$$a_i = 2^{e^{(a)}}(-1)^{s_i^{(a)}} \sum_{j=1}^{m} 2^{1-j} A_{i,j} \tag{4.10}$$

$$b_i = 2^{e^{(b)}}(-1)^{s_i^{(b)}} \sum_{j=1}^{m} 2^{1-j} B_{i,j}. \tag{4.11}$$

Here, we use the notation for common exponents $e^{(a)}, e^{(b)}$, signs $s_i^{(a)}, s_i^{(b)} \in \{0, 1\}$, and mantissa digits $A_{i,j}, B_{i,j} \in \{0, 1\}$. The exact value of the dot product is then:

$$\sum_{i=1}^{n} a_i b_i = 2^{e^{(a)}+e^{(b)}+2} \sum_{i=1}^{n} (-1)^{s_i^{(a)}+s_i^{(b)}} P_i \tag{4.12}$$

$$P_i := \sum_{1 \le j,k \le m} p_{i,j,k} \tag{4.13}$$

$$p_{i,j,k} := 2^{-(j+k)} A_{i,j} B_{i,k} \tag{4.14}$$

For half-precision and pseudo-single precision without extended representation, P_i is calculated exactly as above.

For single precision and double precision, similar to vector multiply-add calculation (Section 4.3), we replace P_i with ${P'}_i$, which adds correction terms instead of summing some partial products. This correction is performed identically for each i, just like in the case of single precision and double precision for vector multiply-add calculation. Specifically, for single precision, we omit multiplication of the lower 5 bits of the input mantissas, and for double precision, we omit multiplication of the lower 16 bits, then detect cases where the result would be zero and set it to zero; otherwise, we treat the omitted partial products as if only the most significant bit of the mantissa was set. For additions in the i direction, there is no rounding or truncation, and signed integer addition is performed with sufficient precision.

For half-precision with extended representation, before adding in the i direction, we right-shift and round each P_i by 6 bits if only one of a_i and b_i uses an extended representation and by 12 bits if both do. The exact definition of this operation requires delving into the details of the multiplier output and is omitted.

After calculating the dot product, the subsequent steps are identical to those for vector multiply-add calculation, including precision extension and reduction for each precision level.

It is also the case that replacing partial products with correction terms in single precision and double precision has little practical impact, similar to vector multiply-add calculation.