Arm China Zhouyi Compass C

Version: 1.14

Programming Guide

Confidential

arm CHINA

Arm China Zhouyi Compass C

Programming Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
0000-00	16 November 2020	Confidential	Release for 1.0
0101-00	28 December 2020	Confidential	Update for 1.1
0102-00	15 March 2021	Confidential	Update for 1.2
0105-00	18 June 2021	Confidential	Update for 1.5
0106-00	30 September 2021	Confidential	Update for 1.6
0107-00	31 December 2021	Confidential	Update for 1.7
0108-00	31 March 2022	Confidential	Update for 1.8
0109-00	30 June 2022	Confidential	Update for 1.9
0110-00	30 September 2022	Confidential	Update for 1.10
0111-00	31 December 2022	Confidential	Update for 1.11
0112-00	31 March 2023	Confidential	Update for 1.12
0113-00	30 June 2023	Confidential	Update for 1.13
0114-00	30 September 2023	Confidential	Update for 1.14

Confidential Proprietary Notice

Confidentiality Status

This document is Confidential. This document may only be used and distributed in accordance with the terms of the agreement entered into by Arm China and the party that Arm China delivered this document to.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

https://www.armchina.com

Contents

Arm China Zhouyi Compass C Programming Guide

Preface

This preface introduces the *Arm China Zhouyi Compass C Programming Guide*.

It contains the following sections:

About this book

This book is intended to provide a single guide for developers writing programs for the Zhouyi *Neural Processing Unit* (NPU) processor, bringing together information from a wide variety of sources useful to C language programmers. Hardware concepts such as scalar and *Direct Memory Access* (DMA) modules are covered where knowledge of the architecture is necessary to understand the principles of application programming. We will also look at ways to take full advantage of the capabilities of the Arm China processor.

This is not an introductory level book. It assumes some knowledge of the C programming language and microprocessors, but not of any Arm China-specific background. We cannot hope to cover every topic in detail. In some chapters, we suggest additional reading (referring either to books or websites) that can give a deeper level of background to the topic in hand, but in this book we focus on the Arm China-specific detail. We do not assume the use of any particular toolchain. We will mention both GNU and Arm China tools in the course of the book. We hope that the book is suitable for programmers who have a desktop PC or x86 background and are taking their first steps into the Arm China processor based world.

The book is meant to complement rather than replace other Arm China documentation available for Zhouyi series processors, such as the Arm China *Technical Reference Manual*s (TRMs) for the processors themselves, and documentation for integration and implementation.

Intended audience

This guide is for developers and programmers writing programs for the Zhouyi NPU processor with C programming language.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction

This chapter provides an overview of Compass C.

Chapter 2 Basic syntax of Compass C

This chapter describes the basic syntax of Compass C.

Chapter 3 Compass C built-in functions

This chapter describes the complete set of Compass C built-in functions.

Chapter 4 Inline assembly

This chapter describes the inline assembly function module of Compass C.

Chapter 5 Compilation process and integration

This chapter describes the Compass C compilation process and integration with other tools.

Chapter 6 Debug

This chapter describes the debug methods of Compass C.

Glossary

The Arm® Glossary is a list of terms used in Arm China documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm China meaning differs from the generally accepted meaning.

See the *Arm® Glossary* for more information.

Typographic conventions

italic
: Introduces special terminology, denotes cross-references, and citations.

bold
: Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
: Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`mono`space
: Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`
: Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`
: Denotes language keywords when used outside example code.

SMALL CAPITALS
: Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Additional reading

This book contains information that is specific to this product. See the following documents for other relevant information.

Arm China publications

- *Arm China Zhouyi Compass Software Technical Overview*.
- *Arm China Zhouyi Compass Assembly Programming Guide*.
- *Arm China Zhouyi Z2 AIPU Technical Reference Manual*.
- *Arm China Zhouyi Z3 AIPU Technical Reference Manual*.
- *Arm China Zhouyi NPU X1 Technical Reference Manual*.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content, send an e-mail to *errata@armchina.com*. Give:

- The title *Arm China Zhouyi Compass C Programming Guide*.
- The number 61010015_0114_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm China also welcomes general suggestions for additions and improvements.

Note

Arm China tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Chapter 1
Introduction

This chapter provides an overview of Compass C.

It contains the following sections:

1.1 NPU machine model

The Zhouyi NPU is a high performance and highly flexible processing unit that is dedicated to *Artificial Intelligence* (AI). It includes three different units:

- *Scalar Processing Unit* (SPU).
- *Tensor Processing Cluster* (TPC), which consists of multiple *Tensor Execution Cells* (TECs).
- Fixed function accelerator (also called AIFF).

The SPU and TPC are used for general-purpose computation, while the AIFF is used for specific processing hardware acceleration.

All the processing units share the same instruction fetch unit, so the TPC uses the *Single Instruction Multiple Data* (SIMD) model.

The following figure shows the components of a typical NPU processor.

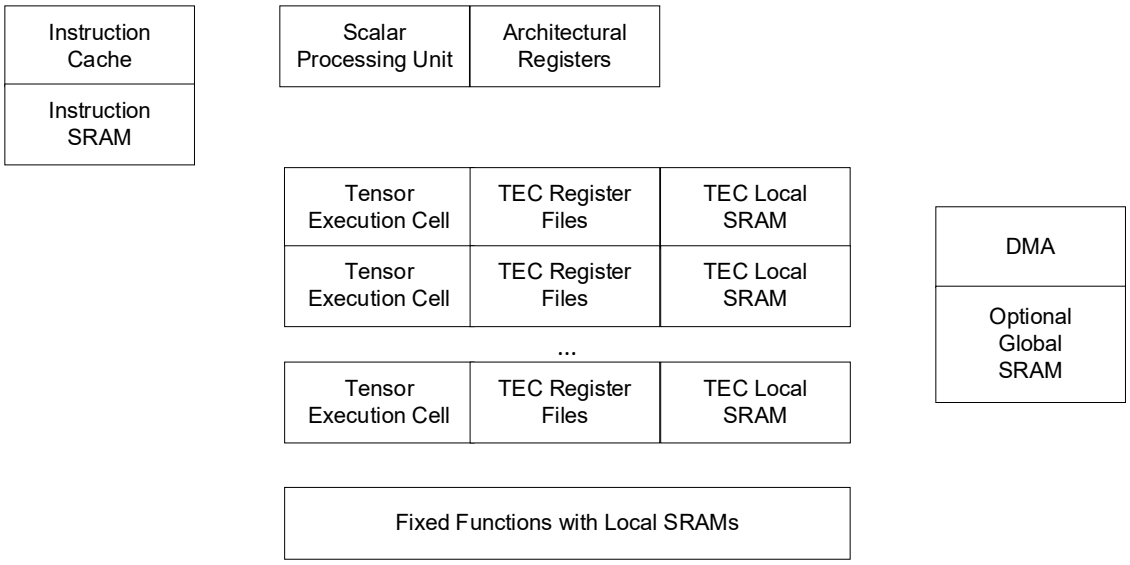

Figure 1-1 Components of a typical NPU processor

1.2 Compass C programming model

To facilitate the programming for the NPU and take full advantage of the hardware features, Compass C is designed based on the C programming language with some NPU extensions.

This section describes the conceptual foundations of Compass C.

1.2.1 Entry function

An entry function defines the entry point of a Compass C program. `Qualifier __entry` is used to define an entry function.

The following example shows how to create an entry function:

```
// entry function definition
__entry void VecAdd(int32_t *A, int32_t *B, int32_t *C, uint32_t size) {
    for (uint32_t i = 0; i < size; i++) {
        C[i] = A[i] + B[i];
    }
}
```

In this example, vector A and vector B of length `size` are added, and the result is stored in vector C.

1.2.2 Data parallel

Compass C supports data parallel programming models. Different from task parallel programming models, SIMD executes the same instruction stream concurrently with each instruction processing different data elements, reflecting the concurrence in the data.

The NPU has multiple TECs, and supports the SIMD model, as shown in the following figure.

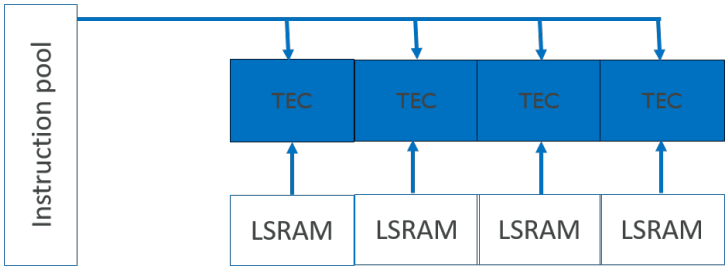

Figure 1-2 Data parallel in the NPU

The length of the vector register in each TEC is 256-bit (32-byte), so the TPC that consists of four TECs can be a virtual vector processing unit that can process 1024 bits (128 bytes) at a time.

1.2.3 Memory model

The NPU has three different memory regions—DDR, LSRAM and GSRAM. Each TEC has private LSRAM. The data in GSRAM is shared data that all TECs can access.

The following figure shows the memory model of the NPU.

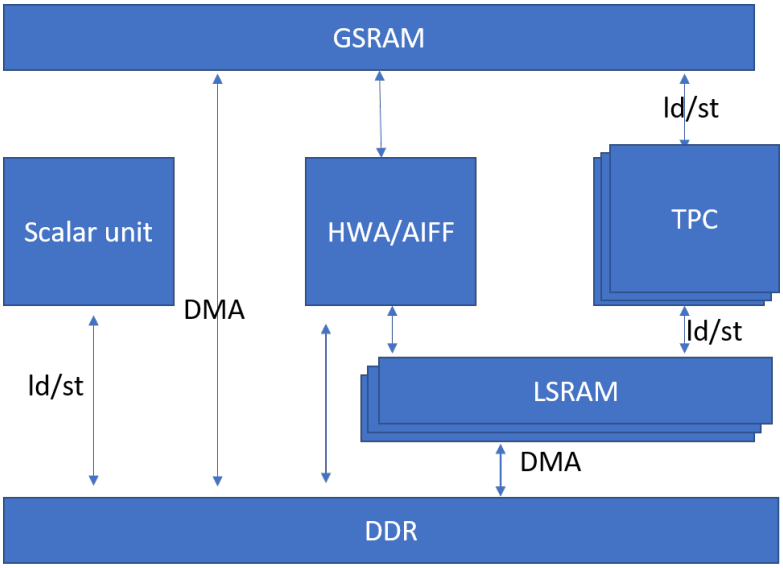

Figure 1-3 Memory model of the NPU

As shown in the figure, the scalar unit can access the data in DDR directly, while the TPC can only access the data in SRAM, but cannot access the data in DDR directly. DMA is used to transfer data between SRAM and DDR.

In Compass C, to distinguish the data in different address regions, address space qualifiers are used to identify the data in SRAM. The address space qualifiers are not needed for the data in DDR. For more information, see *2.4.2 Address space qualifier*.

1.2.4 Programming pattern

In the NPU, the SPU only performs simple arithmetic operations and control flow operations, while most computing depends on the TPC. So, programming for the TPC is critical.

The TPC programming complies with the following pattern:

1. Define data in SRAM.
2. Use the DMA functions to transfer data from DDR to SRAM.
3. Use vector built-in functions to perform vector computing.
4. Use the DMA functions to transfer the result data from SRAM to DDR.

1.2.5 Data slice

Due to the limited size of SRAM, large size of data should be sliced and transferred to SRAM for processing by multiple invocations of DMA functions. In addition, the TPC units require that the data is aligned. Therefore, the unaligned data should be sliced to meet the requirements of SIMD to perform the operation.

Taking four TECs as an example, as described in *1.2.2 Data parallel*, four TECs can be regarded as a virtual processing unit that can simultaneously process 128-byte data at a time. Therefore, it is recommended to slice the data into three parts and perform the operation separately:

- The first part is an integral multiple of 128 bytes. This part is processed in parallel on the four TECs.
- The second part contains less than 128 bytes, but is an integral multiple of 32 bytes. This part can be processed on only one TEC, or on all the four TECs simultaneously through SIMD, with attention to the effective length of the result data.
- The third part is the part less than 32 bytes. This part is processed on one TEC.

The following figure illustrates the data slice in the example.

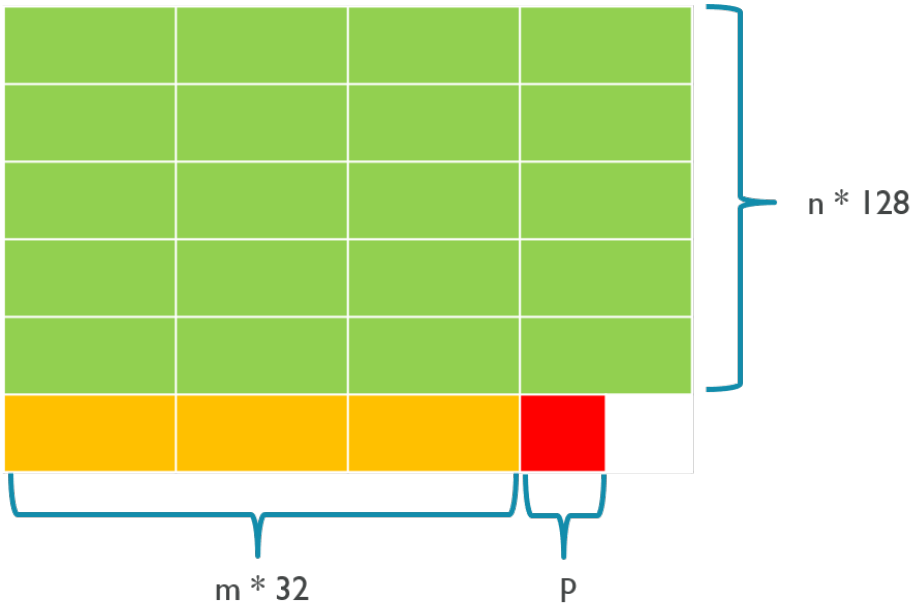

Figure 1-4 Data slice

Chapter 2
Basic syntax of Compass C

This chapter describes the basic syntax of Compass C.

It contains the following sections:

2.1 Data types

2.1.1 Scalar data types

The scalar register in the NPU SPU is 32-bit, so Compass C only supports signed and unsigned 8-bit, 16-bit and 32-bit integer types. The semantics of these types are the same as the standard C semantics.

Table 2-1 Supported data types

Type	Description
int32_t	Signed 32-bit integer.
uint32_t	Unsigned 32-bit integer.
int16_t	Signed 16-bit integer.
uint16_t	Unsigned 16-bit integer.
int8_t	Signed 8-bit integer.
uint8_t	Unsigned 8-bit integer.

2.1.2 Vector data types

The vector data types include:

- Tensor vector, used to place data.
- Predicate vector, used to set mask.

Tensor vector

A single vector register in the NPU TPC is 256-bit, so the tensor vector only supports 256-bit fixed length vector types. The elements of the vector types must be 8-bit, 16-bit or 32-bit integer data.

The following table shows the supported tensor vector types.

Table 2-2 Supported tensor vector types

Type	Description
v32int8_t	32 x int8_t signed 8-bit integer vector.
v32uint8_t	32 x uint8_t unsigned 8-bit integer vector.
v16int16_t	16 x int16_t signed 16-bit integer vector.
v16uint16_t	16 x uint16_t unsigned 16-bit integer vector.
v8int32_t	8 x int32_t signed 32-bit integer vector.
v8uint32_t	8 x uint32_t unsigned 32-bit integer vector.

Predicate vector

The predicate register in the NPU TPC is 32-bit, so the predicate vector only supports 32-bit fixed length vector types. The elements of the vector types must be 1-bit, 2-bit, or 4-bit Boolean data, as shown in the following table.

Table 2-3 Supported predicate vector types

Type	Description
v32bool_t	32 x bool (1-bit) predicate vector.
v16bool_t	16 x bool (2-bit) predicate vector.
v8bool_t	8 x bool (4-bit) predicate vector.

The predicate vector is a data type that is used with the tensor vector to act as the mask in an instruction operation.

The variables of the predicate vector correspond to the variables of the tensor vector. The following table shows the mapping rule.

Table 2-4 Mapping rule for the variables

Vector type	Corresponding predicate vector type
v32int8_t	v32bool_t
v32uint8_t	
v16int16_t	v16bool_t
v16uint16_t	
v8int32_t	v8bool_t
v8uint32_t	

2.1.3 Derived types

Pointer types

All the preceding scalar and vector types have corresponding pointer types. It is recommended that you do not perform complex operations and dereference for pointers.

Compass C only supports the use of simple pointers, but does not support multilevel pointers, such as pointer to pointer.

Arrays and structures

Compass C supports both scalar and vector arrays and structures.

2.2 Operators

This section describes the operators.

2.2.1 Scalar operators

Most scalar operations in Compass C are the same as the operations in C99.

Table 2-5 Scalar supported operations

Operator type	Operator symbol
Arithmetic operator	+
	-
	*
	/
	%
Relational operator	>
	<
	>=
	<=
	==
	!=
Bitwise operator	&
	|
	^
	~
Logical operator	&&
	||
	!
Conditional operator	?:
Shift operator	>>
	<<
Unary operator	+ or -
	++
	--
	sizeof
	& and *
Assignment operator	=, *=, /=, +=, -=, <<=, >>=, &=, ^=, |=

2.2.2 Vector operators

Vector operations are implemented by special instructions that are provided by NPU hardware. Compass C provides a set of vector built-in functions for simplifying programming. For more information, see *3.2 Vector built-in functions.*

2.3 Control statements

The control statements are the same as the statements in the standard C programming language.

2.4 Qualifiers

Compass C supports three types of qualifiers:

- Function qualifier.
- Address space qualifier.
- Type qualifier.

2.4.1 Function qualifier

`Qualifier __entry` is used to specify an entry function.

The following shows an example of the entry function qualifier.

```
__entry void mylayer(uint8_t *input1, uint8_t *input2, uint8_t *output) {
        ...
}
```

The following rules apply to the entry function:

- The return type must be void.
- The entry function parameter cannot have an address space qualifier, and should be 32-bit (int32_t/uint32_t/pointer).
- The entry function can invocate non-entry functions, while the entry function cannot be invocated by other functions.
- There can be at most one entry function in each file.
- There can be at most one entry function in the final linked and generated binary file.

2.4.2 Address space qualifier

The NPU contains three different memory regions that are distinguished by address space qualifiers. The address space qualifiers in Compass C can be the global SRAM qualifier gsram0/gsram1 (or __gsram0/__gsram1) and the local SRAM qualifier lsram0/lsram1 (or lsram0/__lsram1).

If the type of a variable is qualified by an address space qualifier, the variable is allocated in the specified address space. The type of a variable that is qualified with an address space qualifier should be a tensor, a tensor pointer, or a tensor array.

If no address space qualifier is specified, the variable is allocated in the DDR space. The type of a variable without any address space qualifier should be a scalar, a scalar pointer, a scalar array, a structure, or a unit.

For example:

```
__lsram0 v8int32_t va; // valid
__lsram0 int32_t a; // invalid
v8int32_t vb; // invalid
int32_t b; // valid
```

In Compass C, scalar types are stored in DDR and are used in the same manner as in the C programming language. While the vector types are stored in SRAM and the address space must be specified when declaring a vector variable. As needed, the vector address space is divided into global address space and local address space.

Global address space

This address space qualifier is used to specify that variables are allocated in GSRAM and can be accessed by multiple TECs.

Local address space

This address space qualifier is used to specify that variables are allocated in LSRAM and can only be accessed by the corresponding TEC. A TEC can only access its own LSRAM.

2.4.3 Type qualifier

Compass C supports type qualifier const that the C99 specification defines.

Chapter 3
Compass C built-in functions

This chapter describes the complete set of Compass C built-in functions.

It contains the following sections:

3.1 Scalar built-in functions

Compass C provides standard C scalar arithmetic operations and some built-in functions.

Table 3-1 Scalar built-in functions

Function	Description
`uint32_t __abs(int32_t var);`	Gets the absolute value of scalar variable `var`.

`uint32_t __maxu(`
`  uint32_t a,`
`  uint32_t b);`
`int32_t __maxs(`
`  int32_t a,`
`  int32_t b);`
	Gets the maximum value of two variables `a` and `b` of the same sign type. You can also use the ternary selection operator to implement the equivalent function.
	Note: Do not compare two variables of different sign types.

`uint32_t __minu(`
`  uint32_t a,`
`  uint32_t b);`
`int32_t __mins(`
`  int32_t a,`
`  int32_t b);`
	Gets the minimum value of two variables `a` and `b` of the same sign type. You can also use the ternary selection operator to implement the equivalent function.
	Note: Do not compare two variables of different sign types.

`int32_t __bfs(`
`  int32_t value,`
`  uint32_t offset,`
`  uint32_t width);`
	Sets bits of variable `value` to 1, from `offset` to `(offset+width-1)`.

`int32_t __bfc(`
`  int32_t value,`
`  uint32_t offset,`
`  uint32_t width);`
	Sets bits of variable `value` to 0, from `offset` to `(offset+width-1)`.

`int32_t __bfi(`
`  int32_t value,`
`  int32_t offset,`
`  int32_t width);`
	Inverts bits of variable `value`, from `offset` to `(offset+width-1)`.

`uint32_t __bfeu(`
`  int32_t value,`
`  uint32_t offset,`
`  uint32_t width);`
`int32_t __bfes(`
`  int32_t value,`
`  uint32_t offset,`
`  uint32_t width);`
	Extracts bits of variable `value` with unsigned/signed extension, from `offset` to `(offset+width-1)`.

`int32_t __bfd(`
`  int32_t result,`
`  int32_t value,`
`  uint32_t offset,`
`  uint32_t width);`
	Copies bits of variable `value` from `0` to `(width-1)`, to variable `result` bits of the position from `offset` to `(offset + width-1)`.
	Note: The return variable must be the same variable as the first parameter.

`uint32_t __brv(uint32_t var);`	Swaps the higher and lower bits of the variable `var` in sequence, for example, swap the 0th bit with the 31st bit, the 1st bit with the 30th bit.

Function	Description
`int32_t __clz(` `  int32_t value);`	Counts the leading zero of the variable `value`. Leading zero means continuous zero from the highest bit to the lowest bit. When the variable `value` is 0, this function will return 32.
`int32_t __cls(` `  int32_t value);`	Counts the leading sign of the variable `value`. Leading sign means excluding the sign bit (the highest bit) and the continuous bits that are equal with the sign bit from the second highest bit to the lowest bit. When the variable `value` is 0, this function will return 31.

3.2 Vector built-in functions

As an IP for AI, the NPU provides many vector instructions. Accordingly, Compass C provides many vector built-in functions. These vector built-in functions have the same name prefixed with __v (such as __vadd).

The following are the conventions for the argument type and return type of the functions:

- The generic type gentype is used to indicate that functions can take v8int32_t, v8uint32_t, v16int16_t, v16uint16_t, v32int8_t, and v32uint8_t as the type for the arguments, and the type of the arguments should be the same. For example, if x is v8int32_t, y should be v8int32_t.
- The generic type sgentype is used to indicate the signed tensor vector type, such as v8int32_t, v16int16_t, and v32int8_t.
- The generic type ugentype is used to indicate the unsigned tensor vector type, such as v8uint32_t, v16uint16_t and v32uint8_t. The gentype of the arguments should be the same. For example, if x is ugentype, such as v8uint32_t, y is sgentype, which should be v8int32_t.
- The generic type pgentype is used to indicate the corresponding predicate type of gentype. For example, if gentype is v8int32_t, pgentype is v8bool_t. If there is no gentype, you can use the generic type pgentype to indicate that the function can take v8bool_t, v16bool_t, and v32bool_t, and the type of the arguments should be the same.
- The type uimm*x* is used to indicate *x*-bit unsigned immediate value with the range of $[\ 0, 2^{x} - 1\ ]$. The type imm*x* is used to indicate *x*-bit signed immediate value with the range of $[\ -2^{x-1}, 2^{x-1} - 1\ ]$.

3.2.1 Arithmetic built-in functions

__vadd

Table 3-2 __vadd built-in functions

Function	Description

`gentype __vadd(`
`  gentype x,`
`  gentype y);`
	Adds the corresponding elements of tensor variables x and y.

`gentype __vadd(`
`  gentype x,`
`  uimm8);`
	Adds an unsigned immediate value uimm8 to each element of tensor variable x.

`gentype __vadd(`
`  gentype x,`
`  gentype y,`
`  pgentype p);`
	Predicated by the predication variable p, adds the corresponding elements of tensor variables x and y.
	Inactive elements in the return tensor variable are set to zero.

__vaddf

Table 3-3 __vaddf built-in functions

Function	Description

`gentype __vaddf(`
`  gentype x,`
`  uimm8);`
	Shifts the unsigned immediate value uimm8 8 bits to the left, and then adds it to each of the tensor variable x.
	Note: The gentype in this function can take v16int16_t, v16uint16_t, v8int32_t, and v8uint32_t as the type for the arguments only.

__vadds

Table 3-4 __vadds built-in functions

Function	Description

`gentype __vadds(`
`  gentype x,`
`  gentype y,`
`  pgentype p);`
	Adds the corresponding elements of tensor variables x and y.

`sgentype __vadds_s(`
`  ugentype x,`
`  sgentype y,`
`  pgentype p);`
`ugentype __vadds_u(`
`  ugentype x,`
`  sgentype y,`
`  pgentype p);`
	Predicated by the predication variable p, adds the corresponding elements of tensor variables x and y. Inactive elements in the return tensor variable are set to zero.

__vsub

Table 3-5 __vsub built-in functions

Function	Description

`gentype __vsub(`
`  gentype x,`
`  gentype y);`
	Subtracts the corresponding elements of tensor variables x and y.

`gentype __vsub(`
`  gentype x,`
`  uimm8);`
	Subtracts an unsigned immediate value uimm8 to each element of tensor variable x.

`gentype __vsub(`
`  gentype x,`
`  gentype y,`
`  pgentype p);`
	Predicated by the predication variable p, subtracts the corresponding elements of tensor variables x and y. Inactive elements in the return tensor variable are set to zero.

__vsubf

Table 3-6 __vsubf built-in functions

Function	Description

`gentype __vsubf(`
`  gentype x,`
`  uimm8);`
	Shifts the unsigned immediate value uimm8 8 bits to the left, and then subtracts it to each of the tensor variable x.
	Note: The gentype in this function can take v16int16_t, v16uint16_t, v8int32_t, and v8uint32_t as the type for the arguments only.

__vsubs

Table 3-7 __vsubs built-in functions

Function	Description

`gentype __vsubs(`
`  gentype x,`
`  gentype y,`
`  pgentype p);`
	Adds the corresponding elements of tensor variables x and y.

`sgentype __vsubs_s(`
`  ugentype x,`
`  sgentype y,`
`  pgentype p);`
`ugentype __vsubs_u(`
`  ugentype x,`
`  sgentype y,`
`  pgentype p);`
	Predicated by the predication variable p, subtracts the corresponding elements of tensor variables x and y. Inactive elements in the return tensor variable are set to zero.

__vaddh

Table 3-8 __vaddh built-in functions

Function	Description

`gentype __vaddh(`
`  gentype x,`
`  gentype y);`
	Concatenates tensor vector variables x and y as new virtual tensor vector. x is the lower half part, and y is the higher half part.
	For every two adjacent elements in the virtual tensor vector, this function adds elements and places the result in the corresponding element of the return variable.

The following figure shows a simple example in which the element size is word.

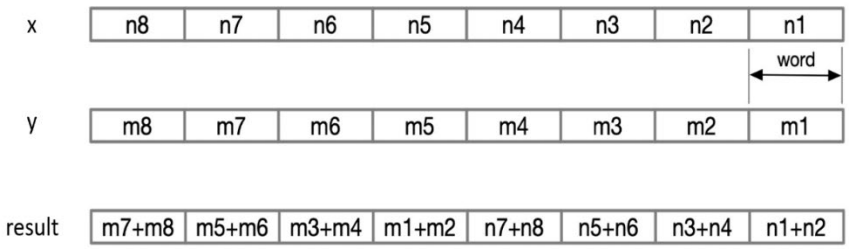

Figure 3-1 __vaddh (v8int32_t x, v8int32_t y) diagram

__vsubh

Table 3-9 __vsubh built-in functions

Function	Description

`gentype __vsubh(`
`  gentype x,`
`  gentype y);`
	Concatenates tensor vector variables x and y as new virtual tensor vector. x is the lower half part, and y is the higher half part.
	For every two adjacent elements in the virtual tensor vector, this function subtracts the higher element from the lower element and places the result in the corresponding element of the return variable.

The following figure shows a simple example in which the element size is word.

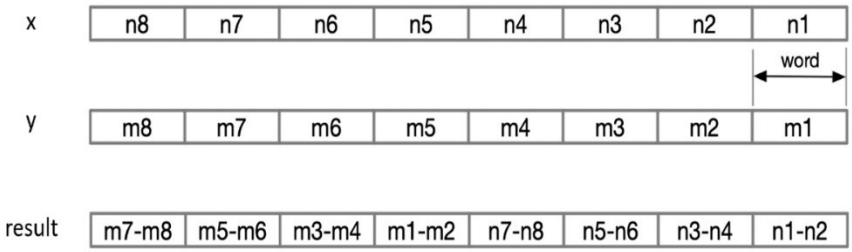

Figure 3-2 __vsubh (v8int32_t x, v8int32_t y) diagram

__vabs

Table 3-10 __vabs built-in functions

Function	Description

`ugentype __vabs(`
`  gentype x,`
`  pgentype p);`
	Predicated by the predication variable `p`, computes the absolute value of every element of tensor variable `x`. Inactive elements in the return tensor variable are set to zero.

__vmul

Table 3-11 __vmul built-in functions

Function	Description

`v16int16_t __vmul(`
`  v32int8_t x,`
`  v32int8_t y,`
`  v32bool_t p);`
	Predicated by the predication variable `p`, multiplies every active element of the low half of tensor variable `x` with the corresponding elements of `y`. Inactive elements in the return tensor variable are set to zero.
	When multiplying a 32-bit element by a 32-bit element, the result of each element pair (higher 32 bits and lower 32 bits, actually) will be put into the adjacent element positions.

`v8int32_t __vmul(`
`  v16int16_t x,`
`  v16int16_t y,`
`  v16bool_t p);`	

`v16uint16_t __vmul(`
`  v32uint8_t x,`
`  v32uint8_t y,`
`  v32bool_t p);`	

`v8uint32_t __vmul(`
`  v16uint16_t x,`
`  v16uint16_t y,`
`  v16bool_t p);`	

`v16int16_t __vmul(`
`  v32int8_t x,`
`  v32uint8_t y,`
`  v32bool_t p);`	

`v8int32_t __vmul(`
`  v16int16_t x,`
`  v16uint16_t y,`	

Function	Description
`v16bool_t p);`	

`v8int32_t __vmul(`
`v8int32_t x,`
`v8int32_t y,`
`v8bool_t p);`	

`v8uint32_t __vmul(`
`v8uint32_t x,`
`v8uint32_t y,`
`v8bool_t p);`	

`v8int32_t __vmul(`
`v8int32_t x,`
`v8uint32_t y,`
`v8bool_t p);`	

The following figure shows a simple example in which the return element size is word and the input is of the half-word element size.

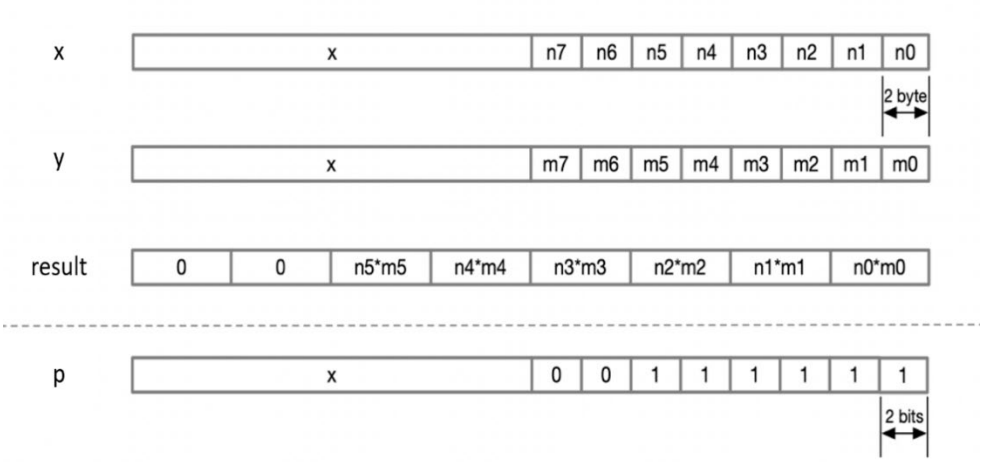

Figure 3-3 __vmul(v16int16_t x, v16int16_t y, v16bool_t p) diagram

__vmule

Table 3-12 __vmule built-in functions

Function	Description

`v16int16_t __vmule(`
`v32int8_t x,`
`v32int8_t y,`
`v32bool_t p);`
	Predicated by the predication variable `p`, multiplies every active element of the low half of tensor variable `x` with the lowest elements of `y`. Inactive elements in the return tensor variable are set to zero.
	When multiplying a 32-bit element by a 32-bit element or a 32-bit element by a 16-bit element, the result of each element pair (higher 32 bits and lower 32 bits, actually) will be put into the adjacent element positions.

`v8int32_t __vmule(`
`v16int16_t x,`
`v16int16_t y,`
`v16bool_t p);`	

`v8int32_t __vmule(` `v16int16_t x,`	

Function	Description
v32int8_t y, v16bool_t p);	
v8int32_t __vmule(v8int32_t x, v8int32_t y, v8bool_t p);	
v8int32_t __vmule(v8int32_t x, v16int16_t y, v8bool_t p);	
v16uint16_t __vmule(v32uint8_t x, v32uint8_t y, v32bool_t p);	
v8uint32_t __vmule(v16uint16_t x, v16uint16_t y, v16bool_t p);	
v8uint32_t __vmule(v16uint16_t x, v32uint8_t y, v16bool_t p);	
v8uint32_t __vmule(v8uint32_t x, v8uint32_t y, v8bool_t p);	
v8uint32_t __vmule(v8uint32_t x, v16uint16_t y, v8bool_t p);	
v16int16_t __vmule(v32int8_t x, v32uint8_t y, v32bool_t p);	
v8int32_t __vmule(v16int16_t x, v16uint16_t y, v16bool_t p);	
v8int32_t __vmule(v16int16_t x, v32uint8_t y, v16bool_t p);	
v8int32_t __vmule(v8int32_t x,	

Function	Description
v8uint32_t y, v8bool_t p);	
v8int32_t __vmule(v8int32_t x, v16uint16_t y, v8bool_t p); (*)	

The following figure shows a simple example in which the return element size is word and input x is of the half-word element size, and y is of the byte element size.

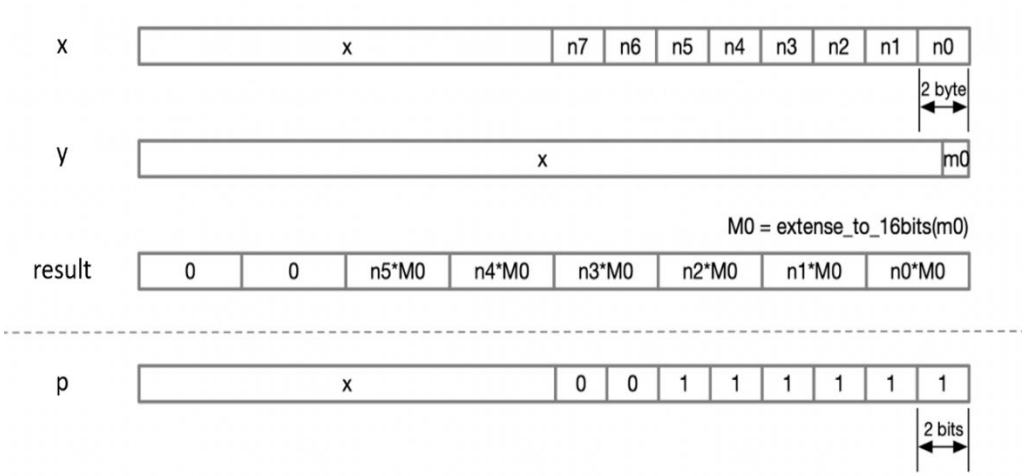

Figure 3-4 __vmule (v16int16_t x, v32int8_t y, v16bool_t p) diagram

__vmulh

Table 3-13 __vmulh built-in functions

Function	Description
v16int16_t __vmulh(v32int8_t x, v32int8_t y, v32bool_t p);	Predicated by the predication variable p, multiplies every active element of the high half of tensor variable x with the corresponding elements of y. Inactive elements in the return tensor variable are set to zero. When multiplying a 32-bit element by a 32-bit element, the result of each element pair (higher 32 bits and lower 32 bits, actually) will be put into the adjacent element positions. These functions are consistent with the corresponding functions of __vmul, except for selecting the elements of the high half of x and y.
v8int32_t __vmulh(v16int16_t x, v16int16_t y, v16bool_t p);	
v16uint16_t __vmulh(v32uint8_t x, v32uint8_t y, v32bool_t p);	
v8uint32_t __vmulh(v16uint16_t x, v16uint16_t y, v16bool_t p);	
v16int16_t __vmulh(	

Function	Description
v32int8_t x, v32uint8_t y, v32bool_t p);	
v8int32_t __vmulh(v16int16_t x, v16uint16_t y, v16bool_t p);	
v8int32_t __vmulh(v8int32_t x, v8int32_t y, v8bool_t p);	
v8uint32_t __vmulh(v8uint32_t x, v8uint32_t y, v8bool_t p);	
v8int32_t __vmulh(v8int32_t x, v8uint32_t y, v8bool_t p);	

The following figure shows a simple example in which the return element size is word and the input is of the half-word element size.

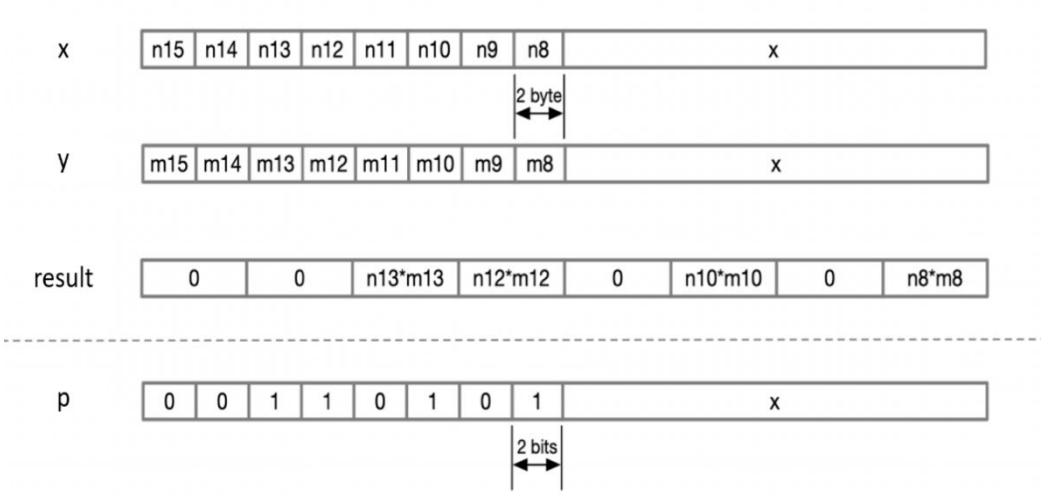

Figure 3-5 __vmulh (v16int16_t x, v16int16_t y, v16bool_t p) diagram

__vmulhe

Table 3-14 __vmulhe built-in functions

Function	Description
v16int16_t __vmulhe(v32int8_t x, v32int8_t y, v32bool_t p);	Predicated by the predication variable p, multiplies every active element of the high half of tensor variable x with the lowest elements of y. Inactive elements in the return tensor variable are set to zero.

Function	Description

`v8int32_t __vmulhe(`
`v16int16_t x,`
`v16int16_t y,`
`v16bool_t p);`
	When multiplying a 32-bit element by a 32-bit element or a 32-bit element by a 16-bit element, the result of each element pair (higher 32 bits and lower 32 bits, actually) will be put into the adjacent element positions.
	These functions are consistent with the corresponding functions of `__vmule`, except for selecting the elements of the high half of `x` and `y`.

`v8int32_t __vmulhe(`
`v16int16_t x,`
`v32int8_t y,`
`v16bool_t p);`	

`v8int32_t __vmulhe(`
`v8int32_t x,`
`v8int32_t y,`
`v8bool_t p);`	

`v8int32_t __vmulhe(`
`v8int32_t x,`
`v16int16_t y,`
`v8bool_t p);`	

`v16uint16_t __vmulhe(`
`v32uint8_t x,`
`v32uint8_t y,`
`v32bool_t p);`	

`v8uint32_t __vmulhe(`
`v16uint16_t x,`
`v16uint16_t y,`
`v16bool_t p);`	

`v8uint32_t __vmulhe(`
`v16uint16_t x,`
`v32uint8_t y,`
`v16bool_t p);`	

The following figure shows a simple example in which the return element size is word and input x is of the half-word element size, y is of the byte element size.

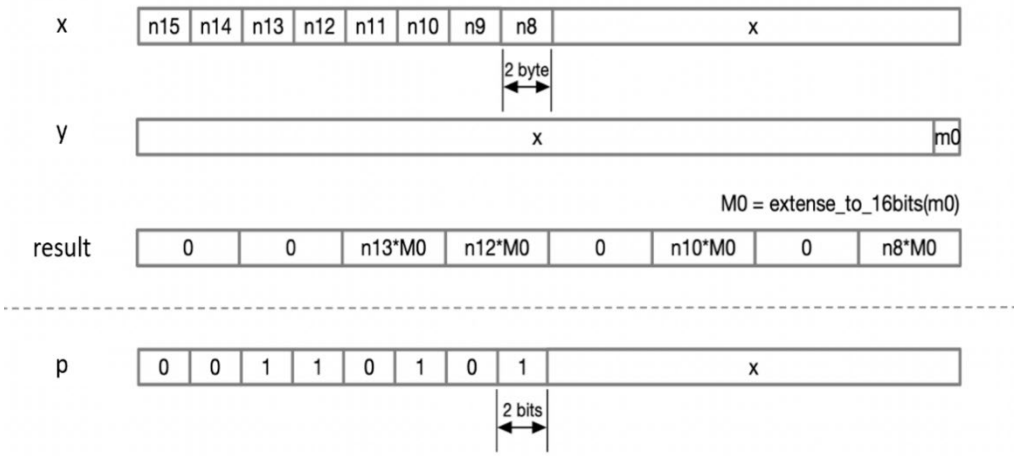

Figure 3-6 __vmulhe(v16int16_t x, v32int8_t y, v16bool_t p) diagram

__vdot

Table 3-15 __vdot built-in functions

Function	Description

`v16int16_t __vdot(`
`v32int8_t x,`
`v32int8_t y,`
`v32bool_t p);`
	Predicated by the predication variable p, multiplies every two adjacent active elements of tensor variable x with the corresponding elements of y, and adds the multiply results as the corresponding elements of the return variable. Inactive elements in the return tensor variable are set to zero.

`v8int32_t __vdot(`
`v16int16_t x,`
`v16int16_t y,`
`v16bool_t p);`	

`v16uint16_t __vdot(`
`v32uint8_t x,`
`v32uint8_t y,`
`v32bool_t p);`	

`v8uint32_t __vdot(`
`v16uint16_t x,`
`v16uint16_t y,`
`v16bool_t p);`	

`v16int16_t __vdot(`
`v32int8_t x,`
`v32uint8_t y,`
`v32bool_t p);`	

`v8int32_t __vdot(`
`v16int16_t x,`
`v16uint16_t y,`
`v16bool_t p);`	

The following figure shows a simple example in which the return element size is word and the input is of the half-word element size.

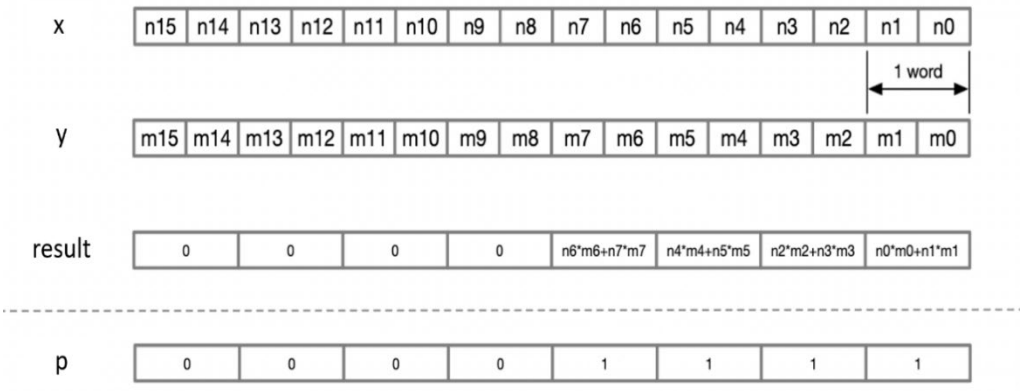

Figure 3-7 __vdot (v16int16_t x, v16int16_t y, v16bool_t p) diagram

__vdote

Table 3-16 __vdote built-in functions

Function	Description

`v16int16_t __vdote(`
`v32int8_t x,`
`v32int8_t y,`
`v32bool_t p);`
	Predicated by the predication variable p, multiplies every two adjacent active elements of tensor variable x with the lowest two elements of y, and adds the multiply results as the corresponding elements of return variable. Inactive elements in the return tensor variable are set to zero.

`v8int32_t __vdote(`
`v16int16_t x,`
`v16int16_t y,`
`v16bool_t p);`	

`v16uint16_t __vdote(`
`v32uint8_t x,`
`v32uint8_t y,`
`v32bool_t p);`	

`v8uint32_t __vdote(`
`v16uint16_t x,`
`v16uint16_t y,`
`v16bool_t p);`	

`v8uint32_t __vdote(`
`v16uint16_t x,`
`v32uint8_t y,`
`v16bool_t p);`	

`v16int16_t __vdote(`
`v32int8_t x,`
`v32uint8_t y,`
`v32bool_t p);`	

`v8int32_t __vdote(`
`v16int16_t x,`
`v16uint16_t y,`
`v16bool_t p);`	

`v8int32_t __vdote(`
`v16int16_t x,`
`v32uint8_t y,`
`v16bool_t p);`	

The following figure shows a simple example in which the return element size is word and input x is of the half-word element size, y is of the byte element size.

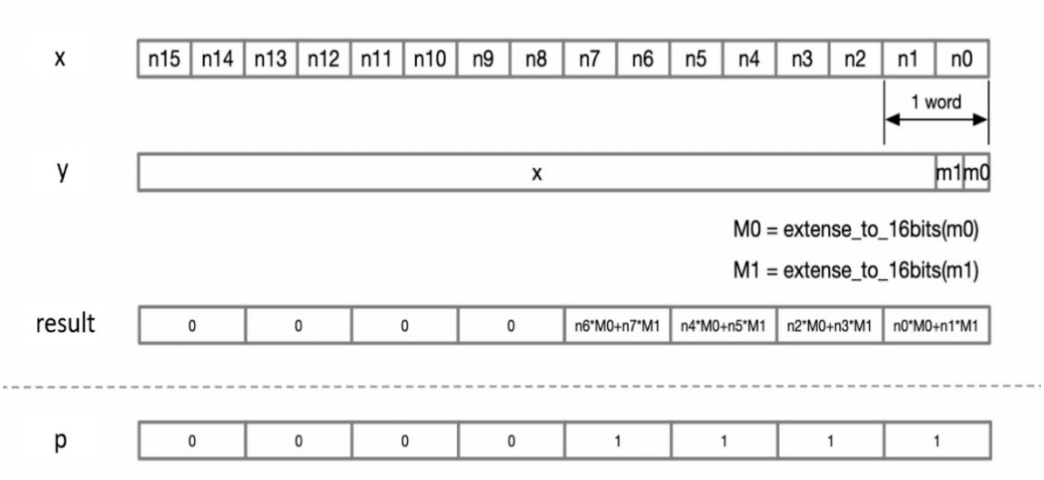

Figure 3-8 __vmulhe (v16int16_t x, v32int8_t y, v16bool_t p) diagram

__vqdot

Table 3-17 __vqdot built-in functions

Function	Description

`v8int32_t __vqdot(`
`v32int8_t x,`
`v32int8_t y,`
`v8bool_t p);`
	Predicated by the predication variable p, multiplies every four adjacent active elements of tensor variable x with the corresponding elements of y, and adds the multiply results as the corresponding elements of the return variable. Inactive elements in the return tensor variable are set to zero.

`v8uint32_t __vqdot(`
`v32uint8_t x,`
`v32uint8_t y,`
`v8bool_t p);`	

`v8int32_t __vqdot(`
`v32int8_t x,`
`v32uint8_t y,`
`v8bool_t p);`	

__vqdot

Table 3-18 __vqdot built-in functions

Function	Description

`v8int32_t __vqdot(`
`v32int8_t x,`
`v32int8_t y,`
`v8bool_t p);`
	Predicated by the predication variable p, multiplies every four adjacent active elements of tensor variable x with the lowest four elements of y, and adds the multiply results as the corresponding elements of the return variable. Inactive elements in the return tensor variable are set to zero.

`v8uint32_t __vqdot(`
`v32uint8_t x,`
`v32uint8_t y,`
`v8bool_t p);`	

`v8int32_t __vqdot(` `v32int8_t x,`	

Function	Description
v32uint8_t y, v8bool_t p);	

__vdpaz

Table 3-19 __vdpaz built-in functions

Function	Description
v16int16_t __vdpaz(v16int16_t acc, v32int8_t x, v32int8_t y, v16bool_t p);	Predicated by the predication variable p, multiplies every two adjacent active elements of tensor variable x with the corresponding elements of y, adds the multiply results, and then performs an accumulate add operation with corresponding active elements of tensor variable acc and return acc. Inactive elements in the return tensor variable are set to zero.
v8int32_t __vdpaz(v8int32_t acc, v16int16_t x, v16int16_t y, v8bool_t p);	
v16uint16_t __vdpaz(v16uint16_t acc, v32uint8_t x, v32uint8_t y, v16bool_t p);	
v8uint32_t __vdpaz(v8uint32_t acc, v16uint16_t x, v16uint16_t y, v8bool_t p);	
V16int16_t __vdpaz(v16int16_t acc, v32int8_t x, v32uint8_t y, v16bool_t p);	
v8int32_t __vdpaz(v8int32_t acc, v16int16_t x, v16uint16_t y, v8bool_t p);	

The following figure shows a simple example in which the return element size is word and the input is of the half-word element size.

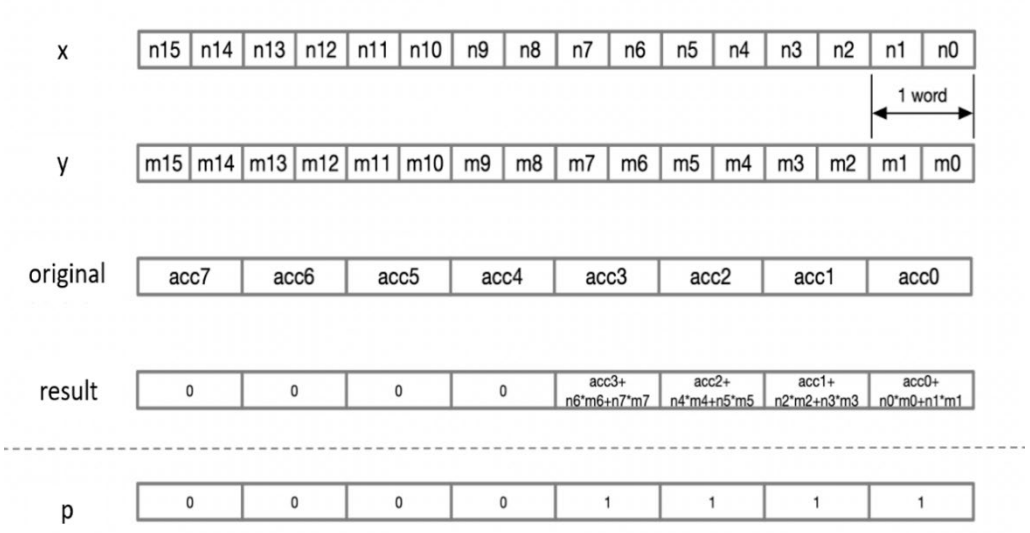

Figure 3-9 __vdot (v8int32_t acc, v16int16_t x, v16int16_t y, v8bool_t p) diagram

__vdpaez

Table 3-20 __vdpaez built-in functions

Function	Description

`v16int16_t __vdpaez(`
`v16int16_t acc,`
`v32int8_t x,`
`v32int8_t y,`
`v16bool_t p);`
	Predicated by the predication variable `p`, multiplies every two adjacent elements of tensor variable `x` with the lowest two elements of `y`, adds the multiply results, and then performs an accumulate add operation with corresponding active elements of tensor variable `acc` and return `acc`. Inactive elements in return tensor variable acc are reset to zero.

`v8int32_t __vdpaez(`
`v8int32_t acc,`
`v16int16_t x,`
`v16int16_t y,`
`v8bool_t p);`	

`v8int32_t __vdpaez(`
`v8int32_t acc,`
`v16int16_t x,`
`v32int8_t y,`
`v8bool_t p);`	

`v16uint16_t __vdpaez(`
`v16uint16_t acc,`
`v32uint8_t x,`
`v32uint8_t y,`
`v16bool_t p);`	

`v8uint32_t __vdpaez(`
`v8uint32_t acc,`
`v16uint16_t x,`
`v16uint16_t y,`
`v8bool_t p);`	

Function	Description

`v8uint32_t __vdpaez(`
`v8uint32_t acc,`
`v16uint16_t x,`
`v32uint8_t y,`
`v8bool_t p);`	

`v16int16_t __vdpaez(`
`v16int16_t acc,`
`v32int8_t x,`
`v32uint8_t y,`
`v16bool_t p);`	

`v8int32_t __vdpaez(`
`v8int32_t acc,`
`v16int16_t x,`
`v16uint16_t y,`
`v8bool_t p);`	

`v8int32_t __vdpaez(`
`v8int32_t acc,`
`v16int16_t x,`
`v32uint8_t y,`
`v8bool_t p);`	

__vdpam

Table 3-21 __vdpam built-in functions

Function	Description

`v16int16_t __vdpam(`
`v16int16_t acc,`
`v32int8_t x,`
`v32int8_t y,`
`v32bool_t p);`
	Predicated by the predication variable `p`, multiplies every two adjacent active elements of tensor variable `x` with the corresponding elements of `y`, adds the multiply results, and then performs an accumulate add operation with corresponding elements of tensor variable `acc` and return `acc`. The multiply results of inactive elements are set to zero.

`v8int32_t __vdpam(`
`v8int32_t acc,`
`v16int16_t x,`
`v16int16_t y,`
`v16bool_t p);`	

`v16uint16_t __vdpam(`
`v16uint16_t acc,`
`v32uint8_t x,`
`v32uint8_t y,`
`v32bool_t p);`	

`v8uint32_t __vdpam(`
`v8uint32_t acc,`
`v16uint16_t x,`
`v16uint16_t y,`
`v16bool_t p);`	

Function	Description

`V16int16_t __vdpam(`
`v16int16_t acc,`
`v32int8_t x,`
`v32uint8_t y,`
`v32bool_t p);`	

`v8int32_t __vdpam(`
`v8int32_t acc,`
`v16int16_t x,`
`v16uint16_t y,`
`v16bool_t p);`	

The following figure shows a simple example in which the return element size is word and input x is of the half-word element size, y is of the byte element size.

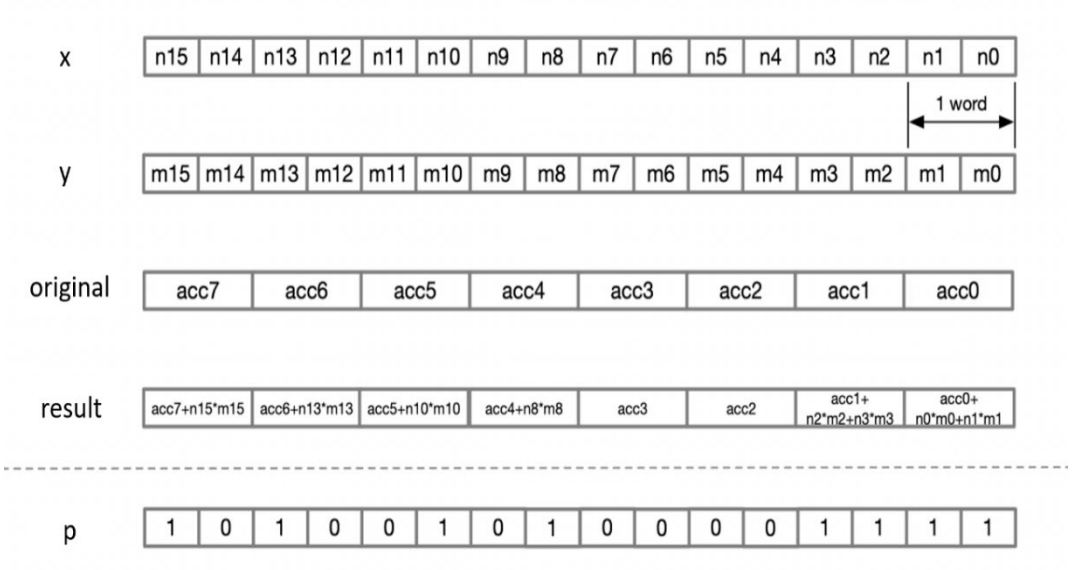

Figure 3-10 __vmulhe (v8int32_t acc, v16int16_t x, v16int16_t y, v16bool_t p) diagram

__vdpaem

Table 3-22 __vdpaem built-in functions

Function	Description

`v16int16_t __vdpaem(`
`v16int16_t acc,`
`v32int8_t x,`
`v32int8_t y,`
`v32bool_t p);`
	Predicated by the predication variable `p`, multiplies every two adjacent active elements of tensor variable `x` with the lowest two elements of `y`, adds the multiply results, and then performs an accumulate add operation with corresponding elements of tensor variable `acc` and return `acc`. The multiply results of inactive elements are set to zero.

`v8int32_t __vdpaem(`
`v8int32_t acc,`
`v16int16_t x,`
`v16int16_t y,`
`v16bool_t p);`	

`v8int32_t __vdpaem(`	

Function	Description
v8int32_t acc, v16int16_t x, v32int8_t y, v16bool_t p);	
v16uint16_t __vdpaem(v16uint16_t acc, v32uint8_t x, v32uint8_t y, v32bool_t p);	
v8uint32_t __vdpaem(v8uint32_t acc, v16uint16_t x, v16uint16_t y, v16bool_t p);	
v8uint32_t __vdpaem(v8uint32_t acc, v16uint16_t x, v32uint8_t y, v16bool_t p);	
V16int16_t __vdpaem(v16int16_t acc, v32int8_t x, v32uint8_t y, v32bool_t p);	
v8int32_t __vdpaem(v8int32_t acc, v16int16_t x, v16uint16_t y, v16bool_t p);	
v8int32_t __vdpaem(v8int32_t acc, v16int16_t x, v32uint8_t y, v16bool_t p);	

__vdpsz

Table 3-23 __vdpsz built-in functions

Function	Description
v16int16_t __vdpsz(v16int16_t acc, v32int8_t x, v32int8_t y, v16bool_t p);	Predicated by the predication variable p, multiplies every two adjacent active elements of tensor variable x with the corresponding elements of y, adds the multiply results, and then performs an accumulate subtract operation with corresponding active elements of tensor variable acc and return acc. Inactive elements in the return tensor variable are set to zero.
v8int32_t __vdpsz(	

Function	Description
v8int32_t acc, v16int16_t x, v16int16_t y, v8bool_t p);	
v16uint16_t __vdpsz(v16uint16_t acc, v32uint8_t x, v32uint8_t y, v16bool_t p);	
v8uint32_t __vdpsz(v8uint32_t acc, v16uint16_t x, v16uint16_t y, v8bool_t p);	
V16int16_t __vdpsz(v16int16_t acc, v32int8_t x, v32uint8_t y, v16bool_t p);	
v8int32_t __vdpsz(v8int32_t acc, v16int16_t x, v16uint16_t y, v8bool_t p);	

__vdpsez

Table 3-24 __vdpsez built-in functions

Function	Description
v16int16_t __vdpsez(v16int16_t acc, v32int8_t x, v32int8_t y, v16bool_t p);	Predicated by the predication variable p, multiplies every two adjacent elements of tensor variable x with the lowest two elements of y, adds the multiply results, and then performs an accumulate subtract operation with corresponding active elements of tensor variable acc and return acc. Inactive elements in return tensor variable acc are reset to zero.
v8int32_t __vdpsez(v8int32_t acc, v16int16_t x, v16int16_t y, v8bool_t p);	
v8int32_t __vdpsez(v8int32_t acc, v16int16_t x, v32int8_t y, v8bool_t p);	
v16uint16_t __vdpsez(	

Function	Description
v16uint16_t acc, v32uint8_t x, v32uint8_t y, v16bool_t p);	
v8uint32_t __vdpsez(v8uint32_t acc, v16uint16_t x, v16uint16_t y, v8bool_t p);	
v8uint32_t __vdpsez(v8uint32_t acc, v16uint16_t x, v32uint8_t y, v8bool_t p);	
v16int16_t __vdpsez(v16int16_t acc, v32int8_t x, v32uint8_t y, v16bool_t p);	
v8int32_t __vdpsez(v8int32_t acc, v16int16_t x, v16uint16_t y, v8bool_t p);	
v8int32_t __vdpsez(v8int32_t acc, v16int16_t x, v32uint8_t y, v8bool_t p);	

__vdpsm

Table 3-25 __vdpsm built-in functions

Function	Description
v16int16_t __vdpsm(v16int16_t acc, v32int8_t x, v32int8_t y, v32bool_t p);	Predicated by the predication variable p, multiplies every two adjacent active elements of tensor variable x with the corresponding elements of y, adds the multiply results, and then performs an accumulate subtract operation with corresponding elements of tensor variable acc and return acc. The multiply results of inactive elements are set to zero.
v8int32_t __vdpsm(v8int32_t acc, v16int16_t x, v16int16_t y, v16bool_t p);	
v16uint16_t __vdpsm(	

Function	Description
v16uint16_t acc, v32uint8_t x, v32uint8_t y, v32bool_t p);	
v8uint32_t __vdpsm(v8uint32_t acc, v16uint16_t x, v16uint16_t y, v16bool_t p);	
V16int16_t __vdpsm(v16int16_t acc, v32int8_t x, v32uint8_t y, v32bool_t p);	
v8int32_t __vdpsm(v8int32_t acc, v16int16_t x, v16uint16_t y, v16bool_t p);	

__vdpsem

Table 3-26 __vdpsem built-in functions

Function	Description
v16int16_t __vdpsem(v16int16_t acc, v32int8_t x, v32int8_t y, v32bool_t p);	Predicated by the predication variable p, multiplies every two adjacent active elements of tensor variable x with the lowest two elements of y, adds the multiply results, and then performs an accumulate subtract operation with corresponding elements of tensor variable acc and return acc. The multiply results of inactive elements are set to zero.
v8int32_t __vdpsem(v8int32_t acc, v16int16_t x, v16int16_t y, v16bool_t p);	
v8int32_t __vdpsem(v8int32_t acc, v16int16_t x, v32int8_t y, v16bool_t p);	
v16uint16_t __vdpsem(v16uint16_t acc, v32uint8_t x, v32uint8_t y, v32bool_t p);	
v8uint32_t __vdpsem(	

Function	Description
v8uint32_t acc, v16uint16_t x, v16uint16_t y, v16bool_t p);	
v8uint32_t __vdpsem(v8uint32_t acc, v16uint16_t x, v32uint8_t y, v16bool_t p);	
V16int16_t __vdpsem(v16int16_t acc, v32int8_t x, v32uint8_t y, v32bool_t p);	
v8int32_t __vdpsem(v8int32_t acc, v16int16_t x, v16uint16_t y, v16bool_t p);	
v8int32_t __vdpsem(v8int32_t acc, v16int16_t x, v32uint8_t y, v16bool_t p);	

__vqdpaz

Table 3-27 __vqdpaz built-in functions

Function	Description
v16int16_t __vqdpaz(v16int16_t acc, v32int8_t x, v32int8_t y, v16bool_t p);	Predicated by the predication variable p, multiplies every four adjacent active elements of tensor variable x with the corresponding elements of y, adds the multiply results, and then performs an accumulate add operation with corresponding active elements of tensor variable acc and return acc. Inactive elements in the return tensor variable are set to zero.
v8int32_t __vqdpaz(v8int32_t acc, v16int16_t x, v16int16_t y, v8bool_t p);	
v16uint16_t __vqdpaz(v16uint16_t acc, v32uint8_t x, v32uint8_t y, v16bool_t p);	
v8uint32_t __vqdpaz(	

Function	Description
v8uint32_t acc, v16uint16_t x, v16uint16_t y, v8bool_t p);	
V16int16_t __vqdpaz(v16int16_t acc, v32int8_t x, v32uint8_t y, v16bool_t p);	
v8int32_t __vqdpaz(v8int32_t acc, v16int16_t x, v16uint16_t y, v8bool_t p);	

__vqdpaez

Table 3-28 __vqdpaez built-in functions

Function	Description
v16int16_t __vqdpaez(v16int16_t acc, v32int8_t x, v32int8_t y, v16bool_t p);	Predicated by the predication variable p, multiplies every four adjacent elements of tensor variable x with the lowest four elements of y, adds the multiply results, and then performs an accumulate add operation with corresponding active elements of tensor variable acc and return acc. Inactive elements in return tensor variable acc are reset to zero.
v8int32_t __vqdpaez(v8int32_t acc, v16int16_t x, v16int16_t y, v8bool_t p);	
v8int32_t __vqdpaez(v8int32_t acc, v16int16_t x, v32int8_t y, v8bool_t p);	
v16uint16_t __vqdpaez(v16uint16_t acc, v32uint8_t x, v32uint8_t y, v16bool_t p);	
v8uint32_t __vqdpaez(v8uint32_t acc, v16uint16_t x, v16uint16_t y, v8bool_t p);	
v8uint32_t __vqdpaez(	

Function	Description
v8uint32_t acc, v16uint16_t x, v32uint8_t y, v8bool_t p);	
v16int16_t __vqdpaez(v16int16_t acc, v32int8_t x, v32uint8_t y, v16bool_t p);	
v8int32_t __vqdpaez(v8int32_t acc, v16int16_t x, v16uint16_t y, v8bool_t p);	
v8int32_t __vqdpaez(v8int32_t acc, v16int16_t x, v32uint8_t y, v8bool_t p);	

__vqdpam

Table 3-29 __vqdpam built-in functions

Function	Description
v16int16_t __vqdpam(v16int16_t acc, v32int8_t x, v32int8_t y, v32bool_t p);	Predicated by the predication variable `p`, multiplies every four adjacent active elements of tensor variable `x` with the corresponding elements of `y`, adds the multiply results, and then performs an accumulate add operation with corresponding elements of tensor variable `acc` and return `acc`. The multiply results of inactive elements are set to zero.
v8int32_t __vqdpam(v8int32_t acc, v16int16_t x, v16int16_t y, v16bool_t p);	
v16uint16_t __vqdpam(v16uint16_t acc, v32uint8_t x, v32uint8_t y, v32bool_t p);	
v8uint32_t __vqdpam(v8uint32_t acc, v16uint16_t x, v16uint16_t y, v16bool_t p);	
V16int16_t __vqdpam(	

Function	Description
v16int16_t acc, v32int8_t x, v32uint8_t y, v32bool_t p);	
v8int32_t __vqdpam(v8int32_t acc, v16int16_t x, v16uint16_t y, v16bool_t p);	

__vqdpaem

Table 3-30 __vqdpaem built-in functions

Function	Description
v16int16_t __vqdpaem(v16int16_t acc, v32int8_t x, v32int8_t y, v32bool_t p);	Predicated by the predication variable p, multiplies every four adjacent active elements of tensor variable x with the lowest four elements of y, adds the multiply results, and then performs an accumulate add operation with corresponding elements of tensor variable acc and return acc. The multiply results of inactive elements are set to zero.
v8int32_t __vqdpaem(v8int32_t acc, v16int16_t x, v16int16_t y, v16bool_t p);	
v8int32_t __vqdpaem(v8int32_t acc, v16int16_t x, v32int8_t y, v16bool_t p);	
v16uint16_t __vqdpaem(v16uint16_t acc, v32uint8_t x, v32uint8_t y, v32bool_t p);	
v8uint32_t __vqdpaem(v8uint32_t acc, v16uint16_t x, v16uint16_t y, v16bool_t p);	
v8uint32_t __vqdpaem(v8uint32_t acc, v16uint16_t x, v32uint8_t y, v16bool_t p);	
V16int16_t __vqdpaem(	

Function	Description
v16int16_t acc, v32int8_t x, v32uint8_t y, v32bool_t p);	
v8int32_t __vqdpaem(v8int32_t acc, v16int16_t x, v16uint16_t y, v16bool_t p);	
v8int32_t __vqdpaem(v8int32_t acc, v16int16_t x, v32uint8_t y, v16bool_t p);	

__vqdpsz

Table 3-31 __vqdpsz built-in functions

Function	Description
v16int16_t __vqdpsz(v16int16_t acc, v32int8_t x, v32int8_t y, v16bool_t p);	Predicated by the predication variable p, multiplies every four adjacent active elements of tensor variable x with the corresponding elements of y, adds the multiply results, and then performs an accumulate subtract operation with corresponding active elements of tensor variable acc and return acc. Inactive elements in the return tensor variable are set to zero.
v8int32_t __vqdpsz(v8int32_t acc, v16int16_t x, v16int16_t y, v8bool_t p);	
v16uint16_t __vqdpsz(v16uint16_t acc, v32uint8_t x, v32uint8_t y, v16bool_t p);	
v8uint32_t __vqdpsz(v8uint32_t acc, v16uint16_t x, v16uint16_t y, v8bool_t p);	
V16int16_t __vqdpsz(v16int16_t acc, v32int8_t x, v32uint8_t y, v16bool_t p);	
v8int32_t __vqdpsz(	

Function	Description
v8int32_t acc, v16int16_t x, v16uint16_t y, v8bool_t p);	

__vqdpsez

Table 3-32 __vqdpsez built-in functions

Function	Description
v16int16_t __vqdpsez(v16int16_t acc, v32int8_t x, v32int8_t y, v16bool_t p);	Predicated by the predication variable p, multiplies every four adjacent elements of tensor variable x with the lowest four elements of y, adds the multiply results, and performs an accumulate subtract operation with corresponding active elements of tensor variable acc and return acc. Inactive elements in return tensor variable acc are reset to zero.
v8int32_t __vqdpsez(v8int32_t acc, v16int16_t x, v16int16_t y, v8bool_t p);	
v8int32_t __vqdpsez(v8int32_t acc, v16int16_t x, v32int8_t y, v8bool_t p);	
v16uint16_t __vqdpsez(v16uint16_t acc, v32uint8_t x, v32uint8_t y, v16bool_t p);	
v8uint32_t __vqdpsez(v8uint32_t acc, v16uint16_t x, v16uint16_t y, v8bool_t p);	
v8uint32_t __vqdpsez(v8uint32_t acc, v16uint16_t x, v32uint8_t y, v8bool_t p);	
v16int16_t __vqdpsez(v16int16_t acc, v32int8_t x, v32uint8_t y, v16bool_t p);	
v8int32_t __vqdpsez(	

Function	Description
v8int32_t acc, v16int16_t x, v16uint16_t y, v8bool_t p);	
v8int32_t __vqdpsez(v8int32_t acc, v16int16_t x, v32uint8_t y, v8bool_t p);	

__vqdpsm

Table 3-33 __vqdpsm built-in functions

Function	Description
v16int16_t __vqdpsm(v16int16_t acc, v32int8_t x, v32int8_t y, v32bool_t p);	Predicated by the predication variable p, multiplies every four adjacent active elements of tensor variable x with the corresponding elements of y, adds the multiply results, and then performs an accumulate subtract operation with corresponding elements of tensor variable acc and return acc. The multiply results of inactive elements are set to zero.
v8int32_t __vqdpsm(v8int32_t acc, v16int16_t x, v16int16_t y, v16bool_t p);	
v16uint16_t __vqdpsm(v16uint16_t acc, v32uint8_t x, v32uint8_t y, v32bool_t p);	
v8uint32_t __vqdpsm(v8uint32_t acc, v16uint16_t x, v16uint16_t y, v16bool_t p);	
V16int16_t __vqdpsm(v16int16_t acc, v32int8_t x, v32uint8_t y, v32bool_t p);	
v8int32_t __vqdpsm(v8int32_t acc, v16int16_t x, v16uint16_t y, v16bool_t p);	

__vqdpsem

Table 3-34 __vqdpsem built-in functions

Function	Description

`v16int16_t __vqdpsem(`
`v16int16_t acc,`
`v32int8_t x,`
`v32int8_t y,`
`v32bool_t p);`
	Predicated by the predication variable `p`, multiplies every four adjacent active elements of tensor variable `x` with the lowest four elements of `y`, adds the multiply results, and then performs an accumulate subtract operation with corresponding elements of tensor variable `acc` and return `acc`. The multiply results of inactive elements are set to zero.

`v8int32_t __vqdpsem(`
`v8int32_t acc,`
`v16int16_t x,`
`v16int16_t y,`
`v16bool_t p);`	

`v8int32_t __vqdpsem(`
`v8int32_t acc,`
`v16int16_t x,`
`v32int8_t y,`
`v16bool_t p);`	

`v16uint16_t __vqdpsem(`
`v16uint16_t acc,`
`v32uint8_t x,`
`v32uint8_t y,`
`v32bool_t p);`	

`v8uint32_t __vqdpsem(`
`v8uint32_t acc,`
`v16uint16_t x,`
`v16uint16_t y,`
`v16bool_t p);`	

`v8uint32_t __vqdpsem(`
`v8uint32_t acc,`
`v16uint16_t x,`
`v32uint8_t y,`
`v16bool_t p);`	

`V16int16_t __vqdpsem(`
`v16int16_t acc,`
`v32int8_t x,`
`v32uint8_t y,`
`v32bool_t p);`	

`v8int32_t __vqdpsem(`
`v8int32_t acc,`
`v16int16_t x,`
`v16uint16_t y,`
`v16bool_t p);`	

`v8int32_t __vqdpsem(` `v8int32_t acc,`	

Function	Description
v16int16_t x, v32uint8_t y, v16bool_t p);	

__vsum

Table 3-35 __vsum built-in functions

Function	Description
v8int32_t __vsum(v8int32_t x v8int32_t y, uimm3, v8bool_t p); v8uint32_t __vsum(v8uint32_t x, v8uint32_t y, uimm3, v8bool_t p);	Predicated by predication variable p, computes the sum of the signed integer in each active element of accumulation variable y, and places the results in the element of accumulation variable x of which the element index is specified by immediate value uimm3. This function returns x as the result. Only the word size of the element is supported.

__vthres

Table 3-36 __vthres built-in functions

Function	Description
gentype __vthres(gentype x, gentype y, pgentype p);	Predicated by the predication variable, uses the element value of tensor variable y as threshold values, and judges each element of tensor variable x with the corresponding threshold value of y. Inactive elements in the result are set to zero. The method of threshold value judgment is that if the judge value is bigger than the threshold value, the judgment result is a threshold value, otherwise, the judgment result is the judge value.

3.2.2 Compare built-in functions

__vceq

Table 3-37 __vceq built-in functions

Function	Description
pgentype __vceq(gentype x, gentype y, pgentype p);	Predicated by predication variable p, compares every active element in the first tensor variable x with corresponding elements in the second tensor variable y, and sets the corresponding elements bit to one in the return predication variable if the corresponding elements between x and y are equal. Inactive elements in the return predication variable are set to zero.
pgentype __vceq(int32_t r, sgentype x, pgentype p);	Predicated by predication variable p, compares every active element in tensor variable x with scalar variable r, and sets the corresponding elements bit to one in the return predication variable

Function	Description

`pgentype __vceq(`
`  uint32_t r,`
`  ugentype x,`
`  pgentype p);`
	if the corresponding element in `x` is equal to `r`. Inactive elements in the return predication variable are set to zero.

__vcneq

Table 3-38 __vcneq built-in functions

Function	Description

`pgentype __vcneq(`
`  gentype x,`
`  gentype y,`
`  pgentype p);`
	Predicated by predication variable `p`, compares every active element in the first tensor variable `x` with corresponding elements in the second tensor variable `y`, and sets the corresponding elements bit to one in the return predication variable if the corresponding elements between `x` and `y` are not equal. Inactive elements in the return predication variable are set to zero.

`pgentype __vcneq(`
`  int32_t r,`
`  sgentype x,`
`  pgentype p);`
	Predicated by predication variable `p`, compares every active element in tensor variable `x` with scalar variable `r`, and sets the corresponding elements bit to one in the return predication variable if the corresponding element in `x` is not equal to `r`. Inactive elements in the return predication variable are set to zero.

`pgentype __vcneq(`
`  uint32_t r,`
`  ugentype x,`
`  pgentype p);`	

__vcge

Table 3-39 __vcge built-in functions

Function	Description

`pgentype __vcge(`
`  gentype x,`
`  gentype y,`
`  pgentype p);`
	Predicated by predication variable `p`, compares every active element in the first tensor variable `x` with corresponding elements in the second tensor variable `y`, and sets the corresponding elements bit to one in the return predication variable if the corresponding elements of `x` is greater than or equal to `y`. Inactive elements in the return predication variable are set to zero.

`pgentype __vcge(`
`  int32_t r,`
`  sgentype x,`
`  pgentype p);`
	Predicated by predication variable `p`, compares every active element in tensor variable `x` with scalar variable `r`, and sets the corresponding elements bit to one in the return predication variable if the corresponding element in `x` is greater than or equal to `r`. Inactive elements in the return predication variable are set to zero.

`pgentype __vcge(`
`  uint32_t r,`
`  ugentype x,`
`  pgentype p);`	

__vcgt

Table 3-40 __vcgt built-in functions

Function	Description

`pgentype __vcgt(`
`  gentype x,`
`  gentype y,`
`  pgentype p);`
	Predicated by predication variable `p`, compares every active element in the first tensor variable `x` with corresponding elements in the second tensor variable `y`, and sets the corresponding elements bit to one in the return predication variable if the corresponding elements of `x` is greater than `y`. Inactive elements in the return predication variable are set to zero.

`pgentype __vcgt(`
`  int32_t r,`
`  sgentype x,`
`  pgentype p);`
	Predicated by predication variable `p`, compares every active element in tensor variable `x` with scalar variable `r`, and sets the corresponding elements bit to one in the return predication variable if the corresponding element in `x` is greater than `r`. Inactive elements in the return predication variable are set to zero.

`pgentype __vcgt(`
`  uint32_t r,`
`  ugentype x,`
`  pgentype p);`	

__vcle

Table 3-41 __vcle built-in functions

Function	Description

`pgentype __vcle(`
`  gentype x,`
`  gentype y,`
`  pgentype p);`
	Predicated by predication variable `p`, compares every active element in the first tensor variable `x` with corresponding elements in the second tensor variable `y`, and sets the corresponding elements bit to one in the return predication variable if the corresponding elements of `x` is less than or equal to `y`. Inactive elements in the return predication variable are set to zero.

`pgentype __vcle(`
`  int32_t r,`
`  gentype x,`
`  pgentype p);`
	Predicated by predication variable `p`, compares every active element in tensor variable `x` with scalar variable `r`, and sets the corresponding elements bit to one in the return predication variable if the corresponding element in `x` is less than or equal to `r`. Inactive elements in the return predication variable are set to zero.

`pgentype __vcle(`
`  uint32_t r,`
`  gentype x,`
`  pgentype p);`	

__vclt

Table 3-42 __vclt built-in functions

Function	Description

`pgentype __vclt(`
`  gentype x,`
`  gentype y,`
`  pgentype p);`
	Predicated by predication variable `p`, compares every active element in the first tensor variable `x` with corresponding elements in the second tensor variable `y`, and sets the corresponding elements bit to one in the return predication variable if the corresponding elements of `x` is less than `y`. Inactive elements in the return predication variable are set to zero.

`pgentype __vclt(`	

Function	Description
int32_t r, gentype x, pgentype p); pgentype __vclt(uint32_t r, gentype x, pgentype p);	Predicated by predication variable p, compares every active element in tensor variable x with scalar variable r, and sets the corresponding elements bit to one in the return predication variable if the corresponding element in x is less than r. Inactive elements in the return predication variable are set to zero.

__vmax

Table 3-43 __vmax built-in functions

Function	Description
sgentype __vmax(sgentype x, imm8); ugentype __vmax(ugentype x, uimm8);	Compares an immediate value (imm8/uimm8) to each element of tensor variable x, and places the larger results in the corresponding elements of the return tensor variable.
gentype __vmax(gentype x, gentype y, pgentype p);	Predicated by the predication variable p, compares every active element in two tensor variables x and y, and places the larger results in the corresponding elements of the return tensor variable. Inactive elements in the destination vector register are set to zero.

__vmin

Table 3-44 __vmin built-in functions

Function	Description
sgentype __vmin(sgentype x, imm8); ugentype __vmin(ugentype x, uimm8);	Compares an immediate value (imm8/uimm8) to each element of tensor variable x, and places the smaller results in the corresponding elements of the return tensor variable.
gentype __vmin(gentype x, gentype y, pgentype p);	Predicated by the predication variable p, compares every active element in two tensor variables x and y, and places the smaller results in the corresponding elements of the return tensor variable. Inactive elements in the destination vector register are set to zero.

__vmaxh

Table 3-45 __vmaxh built-in functions

Function	Description
gentype __vmaxh(gentype x, gentype y);	For every two adjacent elements in the first tensor variable x, compares the elements and places the larger result in the corresponding element of the lower half bits of the return tensor variable. The second tensor variable y does the same thing, but

Function	Description
	places the larger result in the corresponding element of the higher half bits of the return tensor variable.

__vminh

Table 3-46 __vminh built-in functions

Function	Description

`gentype __vminh(`
`  gentype x,`
`  gentype y);`
	For every two adjacent elements in the first tensor variable x, compares the elements and places the smaller result in the corresponding element of the lower half bits of the return tensor variable. The second tensor variable y does the same thing, but places the smaller result in the corresponding element of the higher half bits of the return tensor variable.

__vmaxhi

Table 3-47 __vmaxhi built-in functions

Function	Description

`gentype __vmaxhi(`
`  gentype x,`
`  gentype y);`
	For every two adjacent elements in the first tensor variable x, compares the elements and selects the index of the larger element, and then selects the element from the second tensor variable y with the index and places the selected value in the corresponding element of the lower half bits of the return tensor variable. The higher half bits of the destination tensor register are set to zero. If two adjacent elements are equal, the function selects the index of the lower one.

The following figure shows a simple example in which the element size is word.

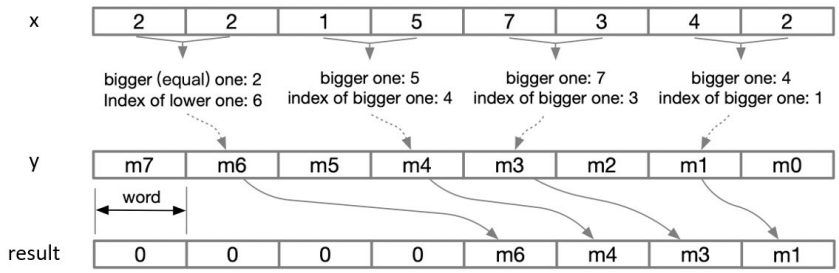

Figure 3-11 __vmaxhi (v8int32_t x, v8int32_t y) diagram

__vminhi

Table 3-48 __vminhi built-in functions

Function	Description

`gentype __vminhi(`
`  gentype x,`
`  gentype y);`
	For every two adjacent elements in the first tensor variable x, compares the elements and selects the index of the larger element, and then selects the element from the second tensor variable y with the index and places the selected value in the corresponding element of the lower half bits of the return tensor variable. The higher half bits of the destination tensor register are set to zero. If two adjacent elements are equal, the function selects the index of the lower one.

3.2.3 Bitwise built-in functions

__vand

Table 3-49 __vadd built-in functions

Function	Description

`gentype __vand(`
`  gentype x,`
`  gentype y,`
`  pgentype p);`
	Predicated by predication variable `p`, performs a bitwise AND on all active elements of tensor variables `x` and `y`. Inactive elements in the return tensor variable are set to zero.

`pgentype __vand(`
`  pgentype px,`
`  pgentype py,`
`  pgentype p);`
	Predicated by predication variable `p`, performs a bitwise AND on all active elements of predication variables `px` and `py`. Inactive elements in the return predication variable are set to zero.

__vor

Table 3-50 __vor built-in functions

Function	Description

`gentype __vor(`
`  gentype x,`
`  gentype y,`
`  pgentype p);`
	Predicated by predication variable `p`, performs a bitwise OR on all active elements of tensor variables `x` and `y`. Inactive elements in the return tensor variable are set to zero.

`pgentype __vor(`
`  pgentype px,`
`  pgentype py,`
`  pgentype p);`
	Predicated by predication variable `p`, performs a bitwise OR on all active elements of predication variables `px` and `py`. Inactive elements in the return predication variable are set to zero.

__vxor

Table 3-51 __vxor built-in functions

Function	Description

`gentype __vxor(`
`  gentype x,`
`  gentype y,`
`  pgentype p);`
	Predicated by predication variable `p`, performs a bitwise exclusive OR on all active elements of tensor variables `x` and `y`. Inactive elements in the return tensor variable are set to zero.

`pgentype __vxor(`
`  pgentype px,`
`  pgentype py,`
`  pgentype p);`
	Predicated by predication variable `p`, performs a bitwise exclusive OR on all active elements of predication variables `px` and `py`. Inactive elements in the return predication variable are set to zero.

__vasr

Table 3-52 __vasr built-in functions

Function	Description
`gentype __vasr(` `  gentype x,`	Predicated by predication variable `p`, performs an arithmetic shift by bit to the right of every active element of tensor variable `x` by the

Function	Description
gentype y, pgentype p);	shift number of corresponding elements of tensor variable y. The shift number y is a vector of unsigned elements in which all bits are significant. Inactive elements in the return tensor variable are set to zero.
gentype __vasr(gentype x, uimmx);	Performs an arithmetic shift by bit to the right of every element of the tensor variable x with the shift number of immediate value uimmx.
gentype __vasr(gentype x, uimmx, pgentype p);	Predicated by predication variable p, performs an arithmetic shift by bit to the right of every active element of tensor variable x with the shift number of immediate value uimmx.

_____ Note_____

The uimmx value depends on gentype. If gentype is v32int8_t/v32uint8_t, uimmx should be uimm3. If gentype is v16int16_t/v16uint16_t, uimmx should be uimm4. If gentype is v8int32_t/v8uint32_t, uimmx should be uimm5.

__vlsr

Table 3-53 __vlsr built-in functions

Function	Description
gentype __vlsr(gentype x, gentype y, pgentype p);	Predicated by predication variable p, performs a logical shift by bit to the right of every active element of tensor variable x by the shift number of corresponding elements of tensor variable y. The shift number y is a vector of unsigned elements in which all bits are significant. Inactive elements in the return tensor variable are set to zero.
gentype __vlsr(gentype x, uimmx);	Performs a logical shift by bit to the right of every element of tensor variable x with the shift number of immediate value uimmx.
gentype __vlsr(gentype x, uimmx, pgentype p);	Predicated by predication variable p, performs a logical shift by bit to the right of every active element of tensor variable x with the shift number of immediate value uimmx.

_____ Note_____

The uimmx value depends on gentype. If gentype is v32int8_t/v32uint8_t, uimmx should be uimm3. If gentype is v16int16_t/v16uint16_t, uimmx should be uimm4. If gentype is v8int32_t/v8uint32_t, uimmx should be uimm5.

__vlsl

Table 3-54 __vlsl built-in functions

Function	Description

`gentype __vlsl(`
`  gentype x,`
`  gentype y,`
`  pgentype p);`
	Predicated by predication variable `p`, performs a logical shift by bit to the left of every active element of tensor variable `x` by the shift number of corresponding elements of tensor variable `y`. The shift number `y` is a vector of unsigned elements in which all bits are significant. Inactive elements in the return tensor variable are set to zero.

`gentype __vlsl(`
`  gentype x,`
`  uimmx);`
	Performs a logical shift by bit to the left of every element of the tensor variable `x` with the shift number of immediate value `uimmx`.

`gentype __vlsl(`
`  gentype x,`
`  uimmx,`
`  pgentype p);`
	Predicated by predication variable `p`, performs a logical shift by bit to the left of every active element of tensor variable `x` with the shift number of immediate value `uimmx`.

_____ Note_____

The uimmx value depends on gentype. If gentype is v32int8_t/v32uint8_t, uimmx should be uimm3. If gentype is v16int16_t/v16uint16_t, uimmx should be uimm4. If gentype is v8int32_t/v8uint32_t, uimmx should be uimm5.

__vror

Table 3-55 __vror built-in functions

Function	Description

`gentype __vror(`
`  gentype x,`
`  gentype y,`
`  pgentype p);`
	Predicated by predication variable `p`, performs a rotate shift by bit to the right of every active element of tensor variable `x` by the shift number of corresponding elements of tensor variable `y`. The shift number `y` is a vector of unsigned elements in which all bits are significant. Inactive elements in the return tensor variable are set to zero.

`gentype __vror(`
`  gentype x,`
`  uimmx);`
	Performs a rotate shift by bit to the right of every element of tensor variable `x` with the shift number of immediate value `uimmx`.

`gentype __vror(`
`  gentype x,`
`  uimmx,`
`  pgentype p);`
	Predicated by predication variable `p`, performs a rotate shift by bit to the right of every active element of tensor variable `x` with the shift number of immediate value `uimmx`.

_____ Note_____

The uimmx value depends on gentype. If gentype is v32int8_t/v32uint8_t, uimmx should be uimm3. If gentype is v16int16_t/v16uint16_t, uimmx should be uimm4. If gentype is v8int32_t/v8uint32_t, uimmx should be uimm5.

__vasl

Table 3-56 __vasl built-in functions

Function	Description

`gentype __vasl(`
`  gentype x,`
`  gentype y,`
`  pgentype p);`
	Predicated by predication variable `p`, performs a saturated arithmetic shift by bit to the left of every active element of tensor variable `x` by the shift number of corresponding elements of tensor variable `y`. The shift number `y` is a vector of unsigned elements in which all bits are significant. Inactive elements in the return tensor variable are set to zero.

`gentype __vasl(`
`  gentype x,`
`  uimmx);`
	Performs a saturated arithmetic shift by bit to the left of every element of tensor variable `x` with the shift number of immediate value `uimmx`.

`gentype __vasl(`
`  gentype x,`
`  uimmx,`
`  pgentype p);`
	Predicated by predication variable `p`, performs a saturated arithmetic shift by bit to the left of every active element of tensor variable `x` with the shift number of immediate value `uimmx`.

Note

The uimmx value depends on gentype. If gentype is v32int8_t/v32uint8_t, uimmx should be uimm3. If gentype is v16int16_t/v16uint16_t, uimmx should be uimm4. If gentype is v8int32_t/v8uint32_t, uimmx should be uimm5.

__vasrr

Table 3-57 __vasrr built-in functions

Function	Description

`gentype __vasrr(`
`  gentype x,`
`  gentype y,`
`  pgentype p);`
	Predicated by predication variable `p`, performs an arithmetic shift by bit to the right with round mode of every active element of tensor variable `x` by the shift number of corresponding elements of tensor variable `y`. The shift number `y` is a vector of unsigned elements in which all bits are significant. Inactive elements in the return tensor variable are set to zero.

`gentype __vasrr(`
`  gentype x,`
`  uimmx);`
	Performs an arithmetic shift by bit to the right with round mode of every element of tensor variable `x` with the shift number of immediate value `uimmx`.

`gentype __vasrr(`
`  gentype x,`
`  uimmx,`
`  pgentype p);`
	Predicated by predication variable `p`, performs an arithmetic shift by bit to the right with round mode of every active element of tensor variable `x` with the shift number of immediate value `uimmx`.

The function supports two types of round modes—round half to nearest even number and round half away from zero.

The mechanism of round half to nearest even number is that, if the fractional part after the shift operation is equal to the half of the maximum fractional number ($\mathbf{2}^{shift}/\mathbf{2}$ in the binary value), the round result is the even number nearest to the value after the shift operation. If the fractional part is not equal to the half value, the result is rounded to the nearest value with the least loss. It means that if the fractional part is larger than the half value, the result will add 1, but if the fractional part is smaller than the half value, the

result will keep the original value. For example, if the shift value is `0b11110110` and shifts right 2 bits, the fractional part will be 0b10 and the value remains `0b11111101`. The half of the maximum fractional number is $\mathbf{2^2/2}$, which is 0b10. The fractional part is equal to the half value, so the shift result will be rounded to the nearest even number, `0b11111110` in this case.

Round half away from zero is also named as round half towards infinity. By this convention, if the fractional part after the shift operation is equal to the half of the maximum fractional number ($\mathbf{2^{shift}/2}$ in the binary value), the round result should add 1 if positive, and subtract 1 if negative. If the fractional part is not equal to the half value, the result is rounded to the nearest value with the least loss.

_____ Note_____

The uimmx value depends on gentype. If gentype is v32int8_t/v32uint8_t, uimmx should be uimm3. If gentype is v16int16_t/v16uint16_t, uimmx should be uimm4. If gentype is v8int32_t/v8uint32_t, uimmx should be uimm5.

__vsetb

Table 3-58 __vsetb built-in functions

Function	Description

`pgentype __vsetb(`
`  pgentype px,`
`  uimm5);`
	Sets one bit in predication variable `px` chosen by immediate value `uimm5`, and keeps the other bits unchanged.

__vinv

Table 3-59 __vinv built-in functions

Function	Description

`gentype __vinv(`
`  gentype x,`
`  pgentype p);`
	Predicated by predication variable `p`, performs a bitwise invert on active elements of tensor variable `x`. Inactive elements in the return tensor variable are set to zero.

`pgentype __vinv(`
`  pgentype px,`
`  pgentype p);`
	Predicated by predication variable `p`, performs a bitwise invert on active elements of predication variable `px`. Inactive elements in the return predication variable are set to zero.

__vclz

Table 3-60 __vclz built-in functions

Function	Description

`gentype __vclz(`
`  gentype x,`
`  pgentype p);`
	Predicated by predication variable `p`, counts leading zero bits in each active element of tensor variable `x`. Inactive elements in the return tensor variable are set to zero.

__vcls

Table 3-61 __vcls built-in functions

Function	Description

`gentype __vcls(`
`  gentype x,`
`  pgentype p);`
	Predicated by predication variable `p`, counts leading sign bits in each active element of tensor variable `x`. Inactive elements in the return tensor variable are set to zero.

__vcnt

Table 3-62 __vcnt built-in functions

Function	Description

`gentype __vcnt(`
`  gentype x,`
`  pgentype p);`
	Predicated by predication variable `p`, counts non-zero bits in each active element of tensor variable `x`. Inactive elements in the return tensor variable are set to zero.

__vbrv

Table 3-63 __vbrv built-in functions

Function	Description

`gentype __vbrv(`
`  gentype x,`
`  pgentype p);`
	Predicated by predication variable `p`, performs bitwise reverse in each active element of tensor variable `x`. Inactive elements in the return tensor variable are set to zero.

__vnsra

Table 3-64 __vnsra built-in functions

Function	Description

`v32int8_t __vnsra_b(`
`  v16int16_t x,`
`  v16int16_t y);`
	Performs an arithmetic narrowing shift right by the value of elements of tensor variable `y`. The value of `y` indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsra_b(`
`  v8int32_t x,`
`  v8int32_t y);`	

`v16int16_t __vnsra_h(`
`  v8int32_t x,`
`  v8int32_t y);`	

`v32int8_t __vnsra_b(`
`  v16int16_t x,`
`  v16int16_t y,`
`  v16bool_t p);`
	Predicated by the predication register, performs an arithmetic narrowing shift right by the value of elements of tensor variable `y`. The value of `y` indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsra_b(`
`  v8int32_t x,`
`  v8int32_t y,`
`  v8bool_t p);`	

Function	Description

`v16int16_t __vnsra_h(`
`v8int32_t x,`
`v8int32_t y,`
`v8bool_t p);`	

`v32int8_t __vnsra_b(`
`v16int16_t x,`
`uimm4);`
	Performs an arithmetic narrowing shift right by the value of immediate uimm4/uimm5. The immediate value indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsra_b(`
`v8int32_t x,`
`uimm5);`	

`v16int16_t __vnsra_h(`
`v8int32_t x,`
`uimm5);`	

`v32int8_t __vnsra_b(`
`v16int16_t x,`
`uimm4,`
`v16bool_t p);`
	Predicated by the predication register, performs an arithmetic narrowing shift right by the value of immediate uimm4/uimm5. The immediate value indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsra_b(`
`v8int32_t x,`
`uimm5,`
`v8bool_t p);`	

`v16int16_t __vnsra_h(`
`v8int32_t x,`
`uimm5,`
`v8bool_t p);`	

__vnsrar

Table 3-65 __vnsrar built-in functions

Function	Description

`v32int8_t __vnsrar_b(`
`v16int16_t x,`
`v16int16_t y);`
	Performs an arithmetic narrowing shift right by the value with rounding of elements of tensor variable `y`. The value of `y` indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsrar_b(`
`v8int32_t x,`
`v8int32_t y);`	

`v16int16_t __vnsrar_h(`
`v8int32_t x,`
`v8int32_t y);`	

`v32int8_t __vnsrar_b(`
`v16int16_t x,`
`v16int16_t y,`
`v16bool_t p);`
	Predicated by the predication register, performs an arithmetic narrowing shift right by the value with rounding of elements of tensor variable `y`. The value of `y` indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

Function	Description

`v32int8_t __vnsrar_b(`
`v8int32_t x,`
`v8int32_t y,`
`v8bool_t p);`	

`v16int16_t __vnsrar_h(`
`v8int32_t x,`
`v8int32_t y,`
`v8bool_t p);`	

`v32int8_t __vnsrar_b(`
`v16int16_t x,`
`uimm4);`
	Performs an arithmetic narrowing shift right by the value with rounding of immediate uimm4/uimm5. The immediate value indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsrar_b(`
`v8int32_t x,`
`uimm5);`	

`v16int16_t __vnsrar_h(`
`v8int32_t x,`
`uimm5);`	

`v32int8_t __vnsrar_b(`
`v16int16_t x,`
`uimm4,`
`v16bool_t p);`
	Predicated by the predication register, performs an arithmetic narrowing shift right by the value with rounding of immediate uimm4/uimm5. The immediate value indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsrar_b(`
`v8int32_t x,`
`uimm5,`
`v8bool_t p);`	

`v16int16_t __vnsrar_h(`
`v8int32_t x,`
`uimm5,`
`v8bool_t p);`	

__vnsras

Table 3-66 __vnsras built-in functions

Function	Description

`v32int8_t __vnsras_b(`
`v16int16_t x,`
`v16int16_t y);`
	Performs an arithmetic narrowing shift right by the value with saturating of elements of tensor variable `y`. The value of `y` indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsras_b(`
`v8int32_t x,`
`v8int32_t y);`	

`v16int16_t __vnsras_h(`
`v8int32_t x,`
`v8int32_t y);`	

Function	Description

`v32uint8_t __vnsras_b(`
`v16uint16_t x,`
`v16uint16_t y);`	

`v32uint8_t __vnsras_b(`
`v8uint32_t x,`
`v8uint32_t y);`	

`v16uint16_t __vnsras_h(`
`v8uint32_t x,`
`v8uint32_t y);`	

`v32int8_t __vnsras_b(`
`v16int16_t x,`
`v16int16_t y,`
`v16bool_t p);`
	Predicated by the predication register, performs an arithmetic narrowing shift right by the value with saturating of elements of tensor variable `y`. The value of `y` indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsras_b(`
`v8int32_t x,`
`v8int32_t y,`
`v8bool_t p);`	

`v16int16_t __vnsras_h(`
`v8int32_t x,`
`v8int32_t y,`
`v8bool_t p);`	

`v32uint8_t __vnsras_b(`
`v16uint16_t x,`
`v16uint16_t y,`
`v16bool_t p);`	

`v32uint8_t __vnsras_b(`
`v8uint32_t x,`
`v8uint32_t y,`
`v8bool_t p);`	

`v16uint16_t __vnsras_h(`
`v8uint32_t x,`
`v8uint32_t y,`
`v8bool_t p);`	

`v32int8_t __vnsras_b(`
`v16int16_t x,`
`uimm4);`
	Performs an arithmetic narrowing shift right by the value with saturating of immediate uimm4/uimm5. The immediate value indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsras_b(`
`v8int32_t x,`
`uimm5);`	

`v16int16_t __vnsras_h(`
`v8int32_t x,`
`uimm5);`	

`v32uint8_t __vnsras_b(` `v16uint16_t x,`	

Function	Description
uimm4);	
v32uint8_t __vnsras_b(v8uint32_t x, uimm5);	
v16uint16_t __vnsras_h(v8uint32_t x, uimm5);	
v32int8_t __vnsras_b(v16int16_t x, uimm4, v16bool_t p);	Predicated by predication register, performs an arithmetic narrowing shift right by the value with saturating of immediate uimm4/uimm5. The immediate value indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.
v32int8_t __vnsras_b(v8int32_t x, uimm5, v8bool_t p);	
v16int16_t __vnsras_h(v8int32_t x, uimm5, v8bool_t p);	
v32uint8_t __vnsras_b(v16uint16_t x, uimm4, v16bool_t p);	
v32uint8_t __vnsras_b(v8uint32_t x, uimm5, v8bool_t p);	
v16uint16_t __vnsras_h(v8uint32_t x, uimm5, v8bool_t p);	

__vnsrasr

Table 3-67 __vnsrasr built-in functions

Function	Description
v32int8_t __vnsrasr_b(v16int16_t x, v16int16_t y);	Performs an arithmetic narrowing shift right by the value with saturating and rounding of elements of tensor variable y. The value of y indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.
v32int8_t __vnsrasr_b(v8int32_t x, v8int32_t y);	
v16int16_t __vnsrasr_h(v8int32_t x,	

Function	Description
`v8int32_t y);`	

`v32uint8_t __vnsrasr_b(`
`v16uint16_t x,`
`v16uint16_t y);`	

`v32uint8_t __vnsrasr_b(`
`v8uint32_t x,`
`v8uint32_t y);`	

`v16uint16_t __vnsrasr_h(`
`v8uint32_t x,`
`v8uint32_t y);`	

`v32int8_t __vnsrasr_b(`
`v16int16_t x,`
`v16int16_t y,`
`v16bool_t p);`
	Predicated by the predication register, performs an arithmetic narrowing shift right by the value with saturating and rounding of elements of tensor variable y. The value of y indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsrasr_b(`
`v8int32_t x,`
`v8int32_t y,`
`v8bool_t p);`	

`v16int16_t __vnsrasr_h(`
`v8int32_t x,`
`v8int32_t y,`
`v8bool_t p);`	

`v32uint8_t __vnsrasr_b(`
`v16uint16_t x,`
`v16uint16_t y,`
`v16bool_t p);`	

`v32uint8_t __vnsrasr_b(`
`v8uint32_t x,`
`v8uint32_t y,`
`v8bool_t p);`	

`v16uint16_t __vnsrasr_h(`
`v8uint32_t x,`
`v8uint32_t y,`
`v8bool_t p);`	

`v32int8_t __vnsrasr_b(`
`v16int16_t x,`
`uimm4);`
	Performs an arithmetic narrowing shift right by the value with saturating and rounding of immediate uimm4/uimm5. The immediate value indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsrasr_b(`
`v8int32_t x,`
`uimm5);`	

`v16int16_t __vnsrasr_h(`
`v8int32_t x,`
`uimm5);`	

Function	Description

`v32uint8_t __vnsrasr_b(`
`v16uint16_t x,`
`uimm4);`	

`v32uint8_t __vnsrasr_b(`
`v8uint32_t x,`
`uimm5);`	

`v16uint16_t __vnsrasr_h(`
`v8uint32_t x,`
`uimm5);`	

`v32int8_t __vnsrasr_b(`
`v16int16_t x,`
`uimm4,`
`v16bool_t p);`
	Predicated by the predication register, performs an arithmetic narrowing shift right by the value with saturating and rounding of immediate uimm4/uimm5. The immediate value indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsrasr_b(`
`v8int32_t x,`
`uimm5,`
`v8bool_t p);`	

`v16int16_t __vnsrasr_h(`
`v8int32_t x,`
`uimm5,`
`v8bool_t p);`	

`v32uint8_t __vnsrasr_b(`
`v16uint16_t x,`
`uimm4,`
`v16bool_t p);`	

`v32uint8_t __vnsrasr_b(`
`v8uint32_t x,`
`uimm5,`
`v8bool_t p);`	

`v16uint16_t __vnsrasr_h(`
`v8uint32_t x,`
`uimm5,`
`v8bool_t p);`	

__vnsrl

Table 3-68 __vnsrl built-in functions

Function	Description

`v32int8_t __vnsrl_b(`
`v16int16_t x,`
`v16int16_t y);`
	Performs a logical narrowing shift right by the value of elements of tensor variable y. The value of y indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsrl_b(`
`v8int32_t x,`
`v8int32_t y);`	

Function	Description

`v16int16_t __vnsrl_h(`
`v8int32_t x,`
`v8int32_t y);`	

`v32int8_t __vnsrl_b(`
`v16int16_t x,`
`v16int16_t y,`
`v16bool_t p);`
	Predicated by the predication register, performs a logical narrowing shift right by the value of elements of tensor variable y. The value of y indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsrl_b(`
`v8int32_t x,`
`v8int32_t y,`
`v8bool_t p);`	

`v16int16_t __vnsrl_h(`
`v8int32_t x,`
`v8int32_t y,`
`v8bool_t p);`	

`v32int8_t __vnsrl_b(`
`v16int16_t x,`
`uimm4);`
	Performs a logical narrowing shift right by the value of immediate uimm4/uimm5. The immediate value indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsrl_b(`
`v8int32_t x,`
`uimm5);`	

`v16int16_t __vnsrl_h(`
`v8int32_t x,`
`uimm5);`	

`v32int8_t __vnsrl_b(`
`v16int16_t x,`
`uimm4,`
`v16bool_t p);`
	Predicated by the predication register, performs a logical narrowing shift right by the value of immediate uimm4/uimm5. The immediate value indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsrl_b(`
`v8int32_t x,`
`uimm5,`
`v8bool_t p);`	

`v16int16_t __vnsrl_h(`
`v8int32_t x,`
`uimm5,`
`v8bool_t p);`	

__vnsrlr

Table 3-69 __vnsrlr built-in functions

Function	Description

`v32int8_t __vnsrlr_b(`
`v16int16_t x,`
`v16int16_t y);`
	Performs a logical narrowing shift right by the value with rounding of elements of tensor variable y. The value of y indicates the shift

Function	Description

`v32int8_t __vnsrlr_b(`
`v8int32_t x,`
`v8int32_t y);`
	size which is an unsigned value in the range of 1 to the number of bits per element.

`v16int16_t __vnsrlr_h(`
`v8int32_t x,`
`v8int32_t y);`	

`v32int8_t __vnsrlr_b(`
`v16int16_t x,`
`v16int16_t y,`
`v16bool_t p);`
	Predicated by predication register, performs a logical narrowing shift right by the value with rounding of elements of tensor variable y. The value of y indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsrlr_b(`
`v8int32_t x,`
`v8int32_t y,`
`v8bool_t p);`	

`v16int16_t __vnsrlr_h(`
`v8int32_t x,`
`v8int32_t y,`
`v8bool_t p);`	

`v32int8_t __vnsrlr_b(`
`v16int16_t x,`
`uimm4);`
	Performs a logical narrowing shift right by the value with rounding of immediate uimm4/uimm5. The immediate value indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsrlr_b(`
`v8int32_t x,`
`uimm5);`	

`v16int16_t __vnsrlr_h(`
`v8int32_t x,`
`uimm5);`	

`v32int8_t __vnsrlr_b(`
`v16int16_t x,`
`uimm4,`
`v16bool_t p);`
	Predicated by predication register, performs a logical narrowing shift right by the value with rounding of immediate uimm4/uimm5. The immediate value indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsrlr_b(`
`v8int32_t x,`
`uimm5,`
`v8bool_t p);`	

`v16int16_t __vnsrlr_h(`
`v8int32_t x,`
`uimm5,`
`v8bool_t p);`	

__vnsrls

Table 3-70 __vnsrls built-in functions

Function	Description

`v32int8_t __vnsrls_b(`
`v16int16_t x,`
`v16int16_t y);`
	Performs a logical narrowing shift right by the value with saturating of elements of tensor variable y. The value of y indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsrls_b(`
`v8int32_t x,`
`v8int32_t y);`	

`v16int16_t __vnsrls_h(`
`v8int32_t x,`
`v8int32_t y);`	

`v32uint8_t __vnsrls_b(`
`v16uint16_t x,`
`v16uint16_t y);`	

`v32uint8_t __vnsrls_b(`
`v8uint32_t x,`
`v8uint32_t y);`	

`v16uint16_t __vnsrls_h(`
`v8uint32_t x,`
`v8uint32_t y);`	

`v32int8_t __vnsrls_b(`
`v16int16_t x,`
`v16int16_t y,`
`v16bool_t p);`
	Predicated by the predication register, performs a logical narrowing shift right by the value with saturating of elements of tensor variable y. The value of y indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsrls_b(`
`v8int32_t x,`
`v8int32_t y,`
`v8bool_t p);`	

`v16int16_t __vnsrls_h(`
`v8int32_t x,`
`v8int32_t y,`
`v8bool_t p);`	

`v32uint8_t __vnsrls_b(`
`v16uint16_t x,`
`v16uint16_t y,`
`v16bool_t p);`	

`v32uint8_t __vnsrls_b(`
`v8uint32_t x,`
`v8uint32_t y,`
`v8bool_t p);`	

`v16uint16_t __vnsrls_h(`
`v8uint32_t x,`
`v8uint32_t y,`	

v8bool_t p);	
v32int8_t __vnsrls_b(v16int16_t x, uimm4);	Performs a logical narrowing shift right by the value with saturating of immediate uimm4/uimm5. The immediate value indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.
v32int8_t __vnsrls_b(v8int32_t x, uimm5);	
v16int16_t __vnsrls_h(v8int32_t x, uimm5);	
v32uint8_t __vnsrls_b(v16uint16_t x, uimm4);	
v32uint8_t __vnsrls_b(v8uint32_t x, uimm5);	
v16uint16_t __vnsrls_h(v8uint32_t x, uimm5);	
v32int8_t __vnsrls_b(v16int16_t x, uimm4, v16bool_t p);	Predicated by the predication register, performs a logical narrowing shift right by the value with saturating of immediate uimm4/uimm5. The immediate value indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.
v32int8_t __vnsrls_b(v8int32_t x, uimm5, v8bool_t p);	
v16int16_t __vnsrls_h(v8int32_t x, uimm5, v8bool_t p);	
v32uint8_t __vnsrls_b(v16uint16_t x, uimm4, v16bool_t p);	
v32uint8_t __vnsrls_b(v8uint32_t x, uimm5, v8bool_t p);	
v16uint16_t __vnsrls_h(v8uint32_t x, uimm5, v8bool_t p);	

__vnsrlsr

Table 3-71 __vnsrlsr built-in functions

Function	Description

`v32int8_t __vnsrlsr_b(`
`v16int16_t x,`
`v16int16_t y);`
	Performs a logical narrowing shift right by the value with saturating and rounding of elements of tensor variable y. The value of y indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsrlsr_b(`
`v8int32_t x,`
`v8int32_t y);`	

`v16int16_t __vnsrlsr_h(`
`v8int32_t x,`
`v8int32_t y);`	

`v32uint8_t __vnsrlsr_b(`
`v16uint16_t x,`
`v16uint16_t y);`	

`v32uint8_t __vnsrlsr_b(`
`v8uint32_t x,`
`v8uint32_t y);`	

`v16uint16_t __vnsrlsr_h(`
`v8uint32_t x,`
`v8uint32_t y);`	

`v32int8_t __vnsrlsr_b(`
`v16int16_t x,`
`v16int16_t y,`
`v16bool_t p);`
	Predicated by the predication register, performs a logical narrowing shift right by the value with saturating and rounding of elements of tensor variable y. The value of y indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.

`v32int8_t __vnsrlsr_b(`
`v8int32_t x,`
`v8int32_t y,`
`v8bool_t p);`	

`v16int16_t __vnsrlsr_h(`
`v8int32_t x,`
`v8int32_t y,`
`v8bool_t p);`	

`v32uint8_t __vnsrlsr_b(`
`v16uint16_t x,`
`v16uint16_t y,`
`v16bool_t p);`	

`v32uint8_t __vnsrlsr_b(`
`v8uint32_t x,`
`v8uint32_t y,`
`v8bool_t p);`	

`v16uint16_t __vnsrlsr_h(`
`v8uint32_t x,`
`v8uint32_t y,`	

Function	Description
v8bool_t p);	
v32int8_t __vnsrlsr_b(v16int16_t x, uimm4);	Performs a logical narrowing shift right by the value with saturating and rounding of immediate uimm4/uimm5. The immediate value indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.
v32int8_t __vnsrlsr_b(v8int32_t x, uimm5);	
v16int16_t __vnsrlsr_h(v8int32_t x, uimm5);	
v32uint8_t __vnsrlsr_b(v16uint16_t x, uimm4);	
v32uint8_t __vnsrlsr_b(v8uint32_t x, uimm5);	
v16uint16_t __vnsrlsr_h(v8uint32_t x, uimm5);	
v32int8_t __vnsrlsr_b(v16int16_t x, uimm4, v16bool_t p);	Predicated by the predication register, performs a logical narrowing shift right by the value with saturating and rounding of immediate uimm4/uimm5. The immediate value indicates the shift size which is an unsigned value in the range of 1 to the number of bits per element.
v32int8_t __vnsrlsr_b(v8int32_t x, uimm5, v8bool_t p);	
v16int16_t __vnsrlsr_h(v8int32_t x, uimm5, v8bool_t p);	
v32uint8_t __vnsrlsr_b(v16uint16_t x, uimm4, v16bool_t p);	
v32uint8_t __vnsrlsr_b(v8uint32_t x, uimm5, v8bool_t p);	
v16uint16_t __vnsrlsr_h(v8uint32_t x, uimm5, v8bool_t p);	

3.2.4 Permutation built-in functions

__vbcast

Table 3-72 __vbcast built-in functions

Function	Description
`gentype __vbcast_b(` `  rgentype x);`	Broadcasts scalar variable `x` into each element of the result tensor.
`gentype __vbcast_h(` `  rgentype x);`	
`gentype __vbcast_w(` `  rgentype x);`	

`gentype __vbcast(`
`  rgentype x,`
`  pgentype p);`
	Predicated by predication variable `p`, broadcasts scalar variable `x` into each element of the result tensor.

The generic type rgentype is used to indicate that the function can take int8_t, uint8_t, int16_t, uint16_t, int32_t, and uint32_t. The type of rgentype and gentype must be the same. For example, if x is int32_t, the return variable should be v8int32_t.

Note

In __vbcast_b, rgentype can only take int8_t/uint8_t. In __vbcast_h, rgentype can only take int16_t/uint16_t. In __vbcast_w, rgentype can only take int32_t/uint32_t.

__vreplic

Table 3-73 __vreplic built-in functions

Function	Description

`gentype __vreplic(`
`  gentype x,`
`  uimmx);`
	Uses immediate value `uimmx` as an index to choose one element from source tensor variable `x` and replicates the element to all elements of the result tensor.

Note

The uimmx value depends on gentype. If gentype is v32int8_t/v32uint8_t, uimmx should be uimm5. If gentype is v16int16_t/v16uint16_t, uimmx should be uimm4. If gentype is v8int32_t/v8uint32_t, uimmx should be uimm3.

The following figure shows an example of the instruction with the word element size.

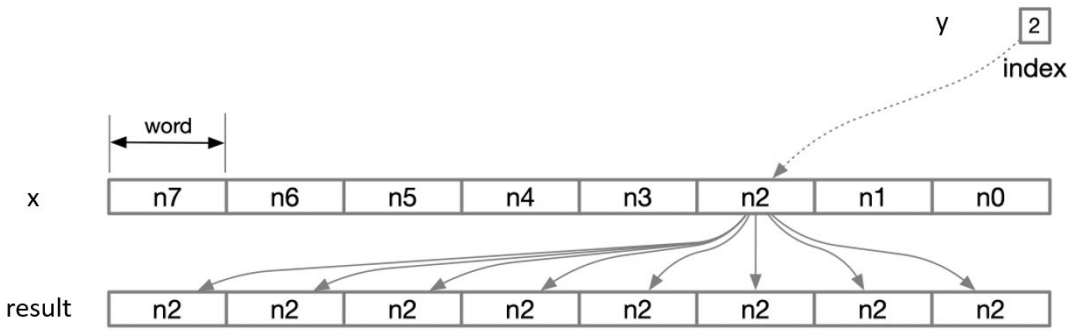

Figure 3-12 __vreplic(v8int32_t, 2) diagram

__vinsert

Table 3-74 __vinsert built-in functions

Function	Description

`v32int8_t __vinsert(`
`  v32int8_t x,`
`  int8_t y,`
`  imm5 z);`
`v32uint8_t __vinsert(`
`  v32uint8_t x,`
`  uint8_t y,`
`  uimm5 z);`
`v16int16_t __vinsert(`
`  v16int16_t x,`
`  int16_t y,`
`  imm4 z);`
`v16uint16_t __vinsert(`
`  v16uint16_t x,`
`  uint16_t y,`
`  uimm4 z);`
`v8int32_t __vinsert(`
`  v8int32_t x,`
`  int32_t y,`
`  imm3 z);`
`v8uint32_t __vinsert(`
`  v8uint32_t x,`
`  uint32_t y,`
`  uimm3 z);`
	Places a copy of the least-significant bits of scalar variable y into the element with a specified index of tensor variable x. The index is indicated by z. Other elements in tensor variable x are unchanged.

__vperm

Table 3-75 __vperm built-in functions

Function	Description

`gentype __vperm(`
`  gentype x,`
`  gentype y,`
	Constructs a table from two tensor variables x and y, reads each element of tensor variable z as the index to select the element from the table, and places the indexed element in the corresponding element of the result tensor. If an index value is greater than or

Function	Description
gentype z);	equal to the number of tensor elements in the table, the function places zero in the corresponding element of the result tensor.

The following figure shows an example of the instruction with the word element size.

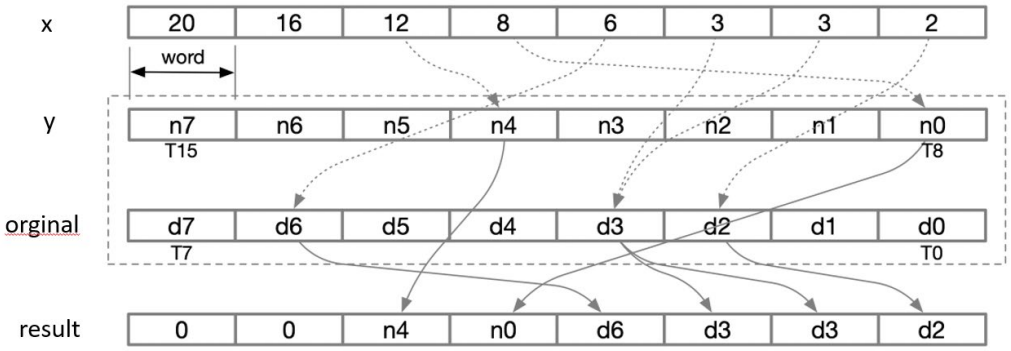

Figure 3-23 __vperm(v8int32_t, v8int32_t, v8int32_t) diagram

__vsel

Table 3-76 __vsel built-in functions

Function	Description
gentype __vsel(gentype x, gentype y, pgentype p);	Predicated by predication variable p, returns the active elements from the first tensor variable x and inactive elements from the second tensor variable y.

The following figure shows an example of the instruction with the word element size.

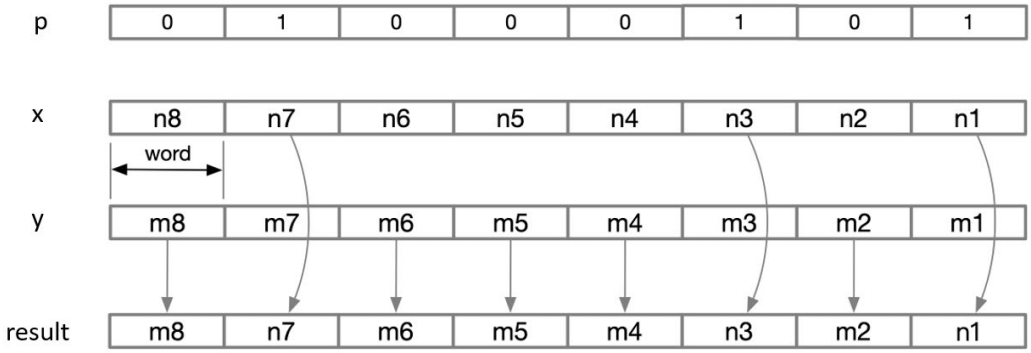

Figure 3-34 __vsel(v8int32_t, v8int32_t, v8bool_t) diagram

__vsxtl

Table 3-77 __vsxtl built-in functions

Function	Description
v8int32_t __vsxtl(v16int16_t x); v16int16_t __vsxtl(	Sign-extends the low half elements of tensor variable x to the double size.

Function	Description
`v32int8_t x);`	

__vsxth

Table 3-78 __vsxth built-in functions

Function	Description

`v8int32_t __vsxth(`
`v16int16_t x);`
`v16int16_t __vsxth(`
`v32int8_t x);`
	Sign-extends the high half elements of tensor variable `x` to the double size.

__vuxtl

Table 3-79 __vuxtl built-in functions

Function	Description

`v8uint32_t __vuxtl(`
`v16uint16_t x);`
`v16uint16_t __vuxtl(`
`v32uint8_t x);`
	Zero-extends the low half elements of tensor variable `x` to the double size.

`v16bool_t __vuxtl(`
`v32bool_t p);`
`v8bool_t __vuxtl(`
`v16bool_t p);`
	Zero-extends the low half elements of tensor predication variable `p` to the double size.

__vuxth

Table 3-80 __vuxth built-in functions

Function	Description

`v8uint32_t __vuxth(`
`v16uint16_t x);`
`v16uint16_t __vuxth(`
`v32uint8_t x);`
	Zero-extends the high half elements of tensor variable `x` to the double size.

__vextl

Table 3-81 __vextl built-in functions

Function	Description

`gentype __vextl(`
`gentype x`
`gentype y);`
	Selects some elements from tensor variables `x` and `y`, joins together and returns. The selected elements are the lower half of the elements.

`pgentype __vextl(`
`pgentype x`
`pgentype y);`
	Selects some elements from predication variables `x` and `y`, joins together and returns. The selected elements are the lower half of the elements.

The following figure shows an example of the instruction with the word element size.

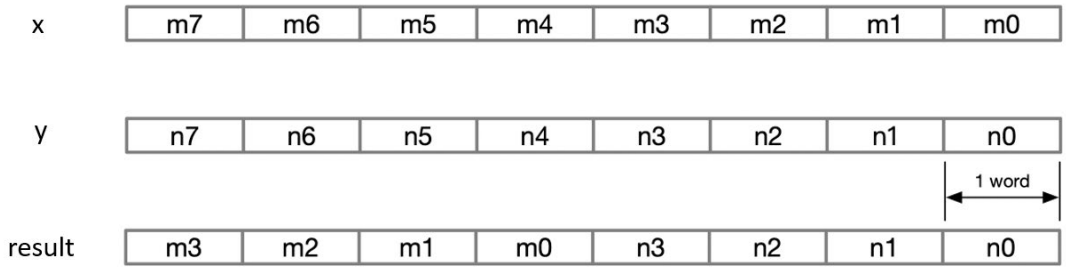

Figure 3-45 __vextl(v8int32_t, v8int32_t) diagram

__vexth

Table 3-82 __vexth built-in functions

Function	Description

`gentype __vexth(`
`  gentype x`
`  gentype y);`
	Selects some elements from tensor variables x and y, joins together and returns. The selected elements are the higher half of the elements.

`pgentype __vexth(`
`  pgentype x`
`  pgentype y);`
	Selects some elements from predication variables x and y, joins together and returns. The selected elements are the higher half of the elements.

The following figure shows an example of the instruction with the word element size.

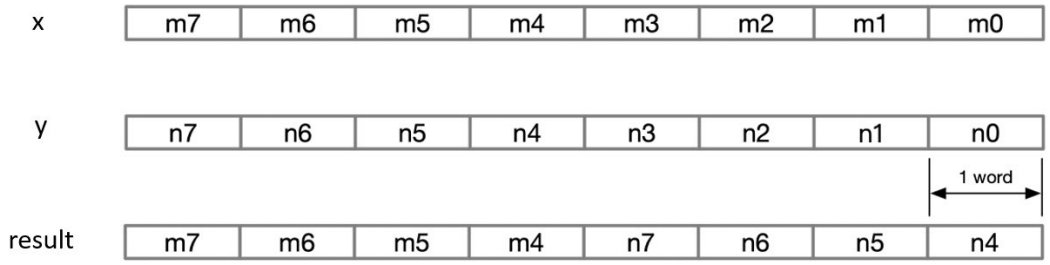

Figure 3-56 __vexth(v8int32_t, v8int32_t) diagram

__vexte

Table 3-83 __vexte built-in functions

Function	Description

`gentype __vexte(`
`  gentype x`
`  gentype y);`
	Selects some elements from tensor variables x and y, joins together and returns. The selected elements are the elements with an even index.

`pgentype __vexte(`
`  pgentype x`
`  pgentype y);`
	Selects some elements from predication variables x and y, joins together and returns. The selected elements are the elements with an even index.

The following figure shows an example of the instruction with the word element size.

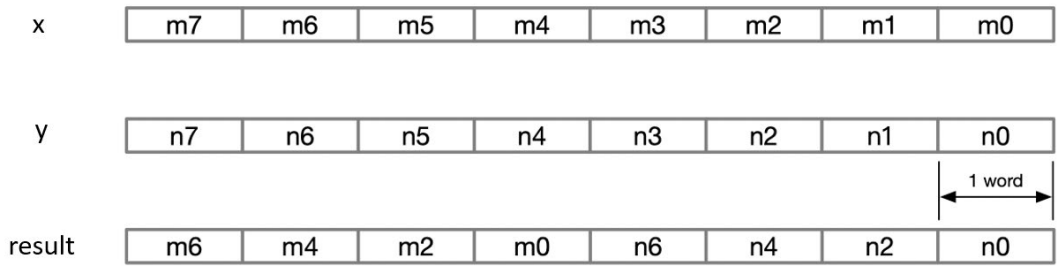

Figure 3-67 __vexte(v8int32_t, v8int32_t) diagram

__vexto

Table 3-84 __vexto built-in functions

Function	Description

`gentype __vexto(`
`  gentype x`
`  gentype y);`
	Selects some elements from tensor variables x and y, joins together and returns. The selected elements are the elements with an odd index.

`pgentype __vexto(`
`  pgentype x`
`  pgentype y);`
	Selects some elements from predication variables x and y, joins together and returns. The selected elements are the elements with an odd index.

The following figure shows an example of the instruction with the word element size.

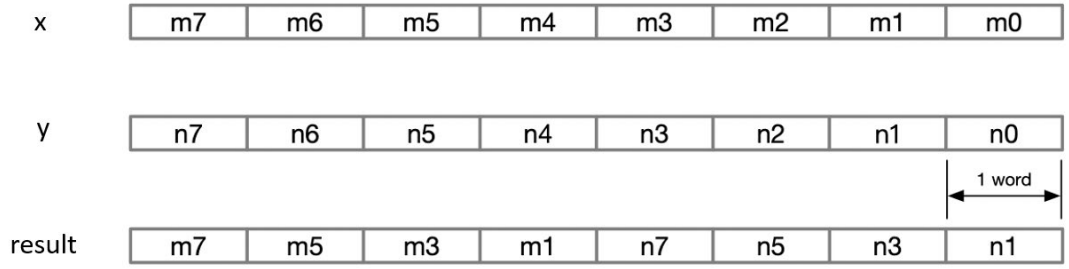

Figure 3-78 __vexto(v8int32_t, v8int32_t) diagram

__vzipl

Table 3-85 __vzipl built-in functions

Function	Description

`gentype __vzipl(`
`  gentype x`
`  gentype y);`
	Selects some elements from tensor variables x and y, rearranges in an interleave alternate sequence, and then returns. The selected elements are the lower half of the elements.

`pgentype __vzipl(`
`  pgentype x`
`  pgentype y);`
	Selects some elements from predication variables x and y, rearranges in an interleave alternate sequence, and then returns. The selected elements are the lower half of the elements.

The following figure shows an example of the instruction with the word element size.

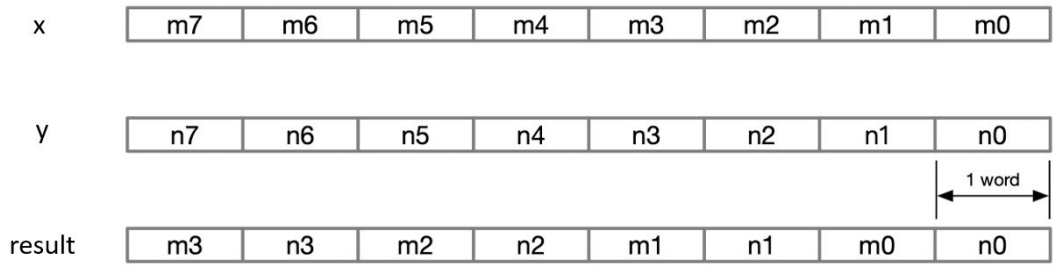

Figure 3-89 __vzipl(v8int32_t, v8int32_t) diagram

__vziph

Table 3-86 __vziph built-in functions

Function	Description

`gentype __vziph(`
`  gentype x`
`  gentype y);`
	Selects some elements from tensor variables x and y, rearranges in an interleave alternate sequence, and then returns. The selected elements are the higher half of the elements.

`pgentype __vziph(`
`  pgentype x`
`  pgentype y);`
	Selects some elements from predication variables x and y, rearranges in an interleave alternate sequence, and then returns. The selected elements are the higher half of the elements.

The following figure shows an example of the instruction with the word element size.

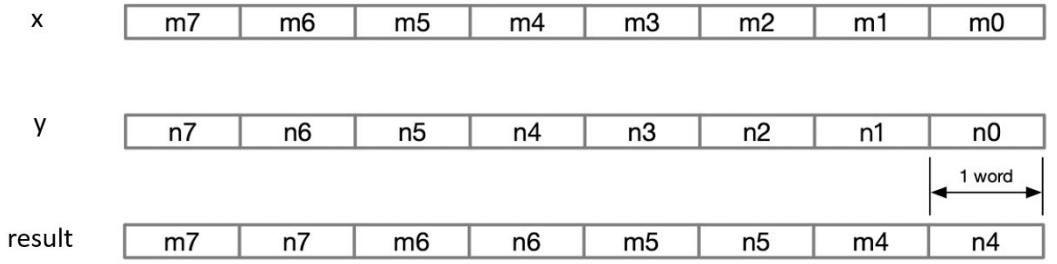

Figure 3-20 __vziph(v8int32_t, v8int32_t) diagram

__vzipe

Table 3-87 __vzipe built-in functions

Function	Description

`gentype __vzipe(`
`  gentype x`
`  gentype y);`
	Selects some elements from tensor variables x and y, rearranges in an interleave alternate sequence, and then returns. The selected elements are the elements with an even index.

`pgentype __vzipe(`
`  pgentype x`
`  pgentype y);`
	Selects some elements from predication variables x and y, rearranges in an interleave alternate sequence, and then returns. The selected elements are the elements with an even index.

The following figure shows an example of the instruction with the word element size.

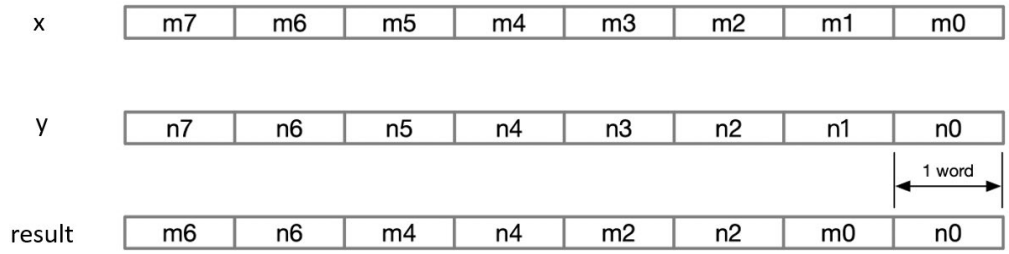

Figure 3-21 __vzipe(v8int32_t, v8int32_t) diagram

__vzipo

Table 3-88 __vzipo built-in functions

Function	Description

`gentype __vzipo(`
`  gentype x`
`  gentype y);`
	Selects some elements from tensor variables x and y, rearranges in an interleave alternate sequence, and then returns. The selected elements are the elements with an odd index.

`pgentype __vzipo(`
`  pgentype x`
`  pgentype y);`
	Selects some elements from predication variables x and y, rearranges in an interleave alternate sequence, and then returns. The selected elements are the elements with an odd index.

The following figure shows an example of the instruction with the word element size.

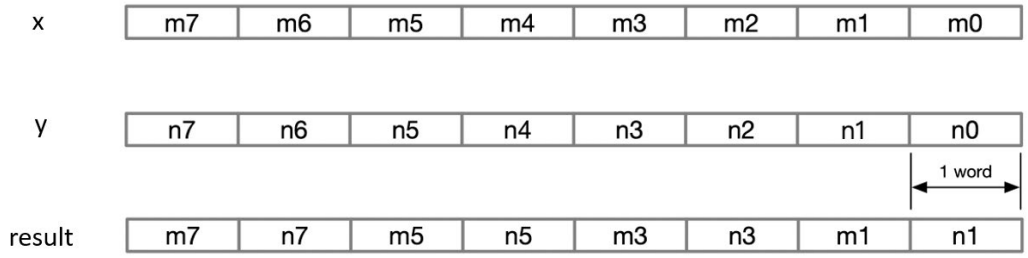

Figure 3-22 __vzipo(v8int32_t, v8int32_t) diagram

__vrevs

Table 3-89 __vrevs built-in functions

Function	Description
`gentype __vrevs(` `  gentype x);`	Reverses the order of all elements in tensor variable x.
`pgentype __vrevs(` `  pgentype x);`	Reverses the order of all elements in predication variable x.

The following figure shows an example of the instruction with the word element size.

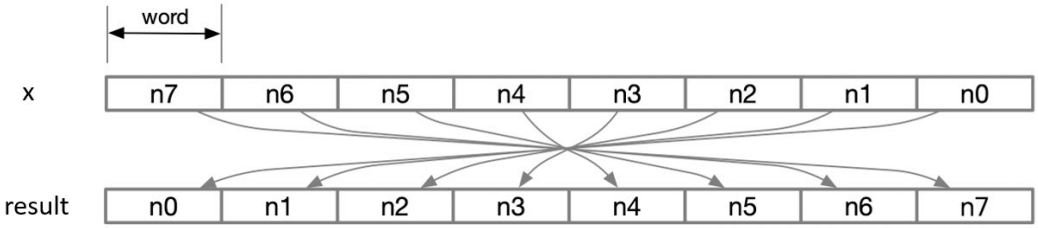

Figure 3-23 __vrevs(v8int32_t) diagram

__vcompt

Table 3-90 __vcompt built-in functions

Function	Description

`gentype __vcompt(`
`  gentype x,`
`  pgentype p);`
	Predicated by predication variable p, reads the active elements from tensor variable x and packs them into the lowest-numbered elements of a tensor. Any remaining elements of the returned tensor are zero.

`pgentype __vcompt(`
`  pgentype x,`
`  pgentype p);`
	Predicated by predication variable p, reads the active elements from predication variable x and packs them into the lowest-numbered elements of a predication. Any remaining elements of the returned predication are zero.

The following figure illustrates the details of this instruction of which the element size is word. The values of p are for example only.

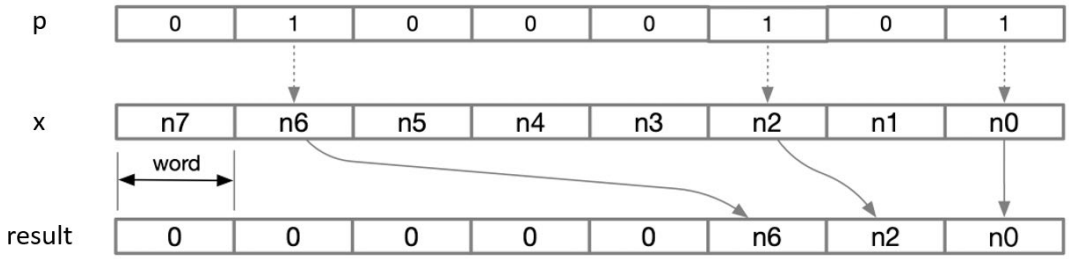

Figure 3-24 __vcompt(v8int32_t, v8bool_t) diagram

__vcompc

Table 3-91 __vcompc built-in functions

Function	Description

`gentype __vcompc(`
`  gentype x,`
`  gentype y,`
`  pgentype p);`
	Predicated by predication variable p, reads the active elements from tensor variable x and packs the low numbered elements of tensor variable y into the higher numbered elements of the returned tensor.

`pgentype __vcompc(`
`  pgentype x,`
`  pgentype y,`
`  pgentype p);`
	Predicated by predication variable p, reads the active elements from predication variable x and packs the low numbered elements of predication variable y into the higher numbered elements of the returned predication.

The following figure illustrates the details of this function of which the element size is word. The values of p are for example only.

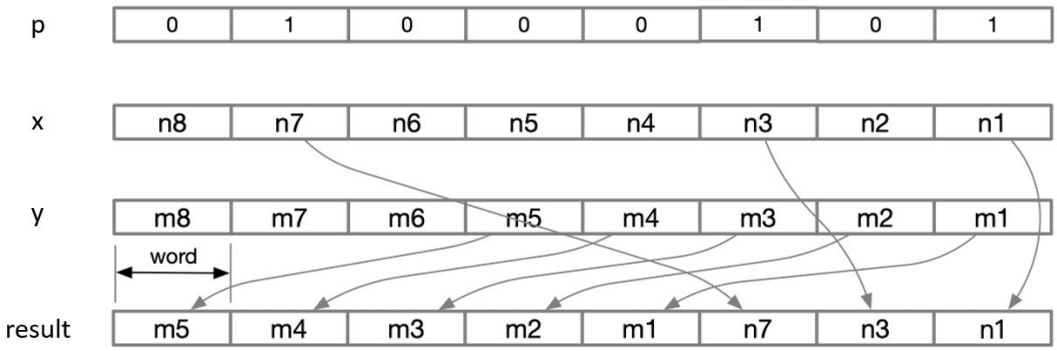

Figure 3-95 __vcompc(v8int32_t, v8int32_t, v8bool_t) diagram

__vshfl

Table 3-92 __vshfl built-in functions

Function	Description

`gentype __vshfl(`
`  gentype x,`
`  uimmx);`
	Performs a rotate shift by element to the right of tensor variable `x` by the shift number of unsigned immediate value `uimmx`.

Note

The uimmx value depends on gentype. If gentype is v32int8_t/v32uint8_t, uimmx should be uimm5. If gentype is v16int16_t/v16uint16_t, uimmx should be uimm4. If gentype is v8int32_t/v8uint32_t, uimmx should be uimm3.

The following figure shows an example of the instruction with the word element size.

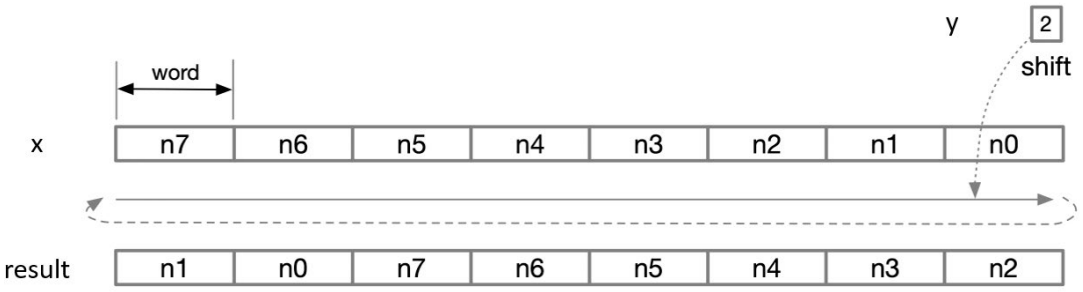

Figure 3-26 __vshfl (v8int32_t, 2) diagram

__vsld

Table 3-93 __vsld built-in functions

Function	Description

`v32int8_t __vsldl(`
`  v32int8_t x,`
`  v32int8_t y,`
`  uimm5);`
`v32uint8_t __vsldl(`
`  v32uint8_t x,`
`  v32int8_t y,`
`  uimm5);`
`v16int16_t __vsldl(`
`  v16int16_t x,`
`  v32int8_t y,`
`  uimm4);`
`v16uint16_t __vsldl(`
`  v16uint16_t x,`
`  v32int8_t y,`
`  uimm4);`
`v8int32_t __vsldl(`
`  v8int32_t x,`
`  v32int8_t y,`
`  uimm3);`
`v8uint32_t __vsldl(`
`  v8uint32_t x,`
`  v32int8_t y,`
`  uimm3);`
	Shifts by element to the left of tensor variable `x` by the shift number of unsigned immediate value `uimm5`/`uimm4`/`uimm3` without the boundary padding. Tensor variable `y` is used to pad the shift remain space.

`v32int8_t __vslddl(`
`  v32int8_t x,`
`  v32int8_t y,`
`  uimm5);`
`v32uint8_t __vslddl(`
`  v32uint8_t x,`
`  v32int8_t y,`
`  uimm5);`
`v16int16_t __vslddl(`
`  v16int16_t x,`
`  v32int8_t y,`
`  uimm4);`
`v16uint16_t __vslddl(`
`  v16uint16_t x,`
`  v32int8_t y,`
`  uimm4);`
`v8int32_t __vslddl(`
`  v8int32_t x,`
`  v32int8_t y,`
`  uimm3);`
	Shifts by element to the left of tensor variable `x` by the shift number of unsigned immediate value `uimm5`/`uimm4`/`uimm3` without boundary padding. Tensor variable `y` is used to pad the shift remain space. The instruction has two lines of shift mode.

Function	Description

`v8uint32_t __vslddl(`
`  v8uint32_t x,`
`  v32int8_t y,`
`  uimm3);`	

`v32int8_t __vsldr(`
`  v32int8_t x,`
`  v32int8_t y,`
`  uimm5);`
`v32uint8_t __vsldr(`
`  v32uint8_t x,`
`  v32int8_t y,`
`  uimm5);`
`v16int16_t __vsldr(`
`  v16int16_t x,`
`  v32int8_t y,`
`  uimm4);`
`v16uint16_t __vsldr(`
`  v16uint16_t x,`
`  v32int8_t y,`
`  uimm4);`
`v8int32_t __vsldr(`
`  v8int32_t x,`
`  v32int8_t y,`
`  uimm3);`
`v8uint32_t __vsldr(`
`  v8uint32_t x,`
`  v32int8_t y,`
`  uimm3);`
	Shifts by element to the right of tensor variable `x` by the shift number of unsigned immediate value `uimm5`/`uimm4`/`uimm3` without boundary padding. Tensor variable `y` is used to pad the shift remain space.

`v32int8_t __vsldfr(`
`  v32int8_t x,`
`  v32int8_t y,`
`  uimm5);`
`v32uint8_t __vsldfr(`
`  v32uint8_t x,`
`  v32int8_t y,`
`  uimm5);`
`v16int16_t __vsldfr(`
`  v16int16_t x,`
`  v32int8_t y,`
`  uimm4);`
`v16uint16_t __vsldfr(`
`  v16uint16_t x,`
`  v32int8_t y,`
`  uimm4);`
`v8int32_t __vsldfr(`
`  v8int32_t x,`
`  v32int8_t y,`
	Shifts by element to the right of tensor variable `x` by the shift number of unsigned immediate value `uimm5`/`uimm4`/`uimm3` with boundary padding. Tensor variable `y` is used to pad the shift remain space.

Function	Description
uimm3); v8uint32_t __vsldfr(v8uint32_t x, v32int8_t y, uimm3);	
v32int8_t __vslddr(v32int8_t x, v32int8_t y, uimm5); v32uint8_t __vslddr(v32uint8_t x, v32int8_t y, uimm5); v16int16_t __vslddr(v16int16_t x, v32int8_t y, uimm4); v16uint16_t __vslddr(v16uint16_t x, v32int8_t y, uimm4); v8int32_t __vslddr(v8int32_t x, v32int8_t y, uimm3); v8uint32_t __vslddr(v8uint32_t x, v32int8_t y, uimm3);	Shifts by element to the right of tensor variable `x` by the shift number of unsigned immediate value `uimm5`/`uimm4`/`uimm3` without boundary padding. Tensor variable `y` is used to pad the shift remain space. The instruction has two lines of shift mode.
v32int8_t __vslddfr(v32int8_t x, v32int8_t y, uimm5); v32uint8_t __vslddfr(v32uint8_t x, v32int8_t y, uimm5); v16int16_t __vslddfr(v16int16_t x, v32int8_t y, uimm4); v16uint16_t __vslddfr(v16uint16_t x, v32int8_t y, uimm4); v8int32_t __vslddfr(v8int32_t x, v32int8_t y, uimm3); v8uint32_t __vslddfr(	Shifts by element to the right of tensor variable `x` by the shift number of unsigned immediate value `uimm5`/`uimm4`/`uimm3` with boundary padding. Tensor variable `y` is used to pad the shift remain space. The instruction has two lines of shift mode.

Function	Description
v8uint32_t x, v32int8_t y, uimm3);	

__vtbl4

Table 3-94 __vtbl4 built-in functions

Function	Description
void __vtbl4_b(v32int8_t x, v32int8_t *y, v32int8_t z);	Constructs a table from variable pointer y with 4 elements length, reads each element of tensor variable z as an index to select the element from the table, and places the indexed element in the corresponding element of tensor variable x. If an index value is greater than or equal to the number of tensor elements in the table, the function places zero in the corresponding element of tensor variable x.
void __vtbl4_h(v16int16_t x, v16int16_t *y, v16int16_t z);	
void __vtbl4_w(v8int32_t x, v8int32_t *y, v8int32_t z);	

3.2.5 Move built-in functions

__vmov

Table 3-95 __vmov built-in functions

Function	Description
pgentype __vmov_b(uint32_t r);	Moves scalar variable r to the return predication variable.
pgentype __vmov_h(uint32_t r);	
pgentype __vmov_w(uint32_t r);	

The generic type rgentype is used to indicate that the function can take int8_t, uint8_t, int16_t, uint16_t, int32_t, and uint32_t. The type of rgentype and pgentype must be the same. For example, if x is int32_t, the return variable should be v8bool_t.

Note

In __vmov_b, rgentype can only take int8_t/uint8_t. In __vmov_h, rgentype can only take int16_t/uint16_t. In __vmov_w, rgentype can only take int32_t/uint32_t.

__vcpy

Table 3-96 __vcpy built-in functions

Function	Description

`int32_t __vcpy(`
`  v8int32_t x,`
`  uimm3);`
	Returns an element value of tensor variable `x` with index `uimm3` as the scalar value.

`uint32_t __vcpy(`
`  v8uint32_t x,`
`  uimm3);`	

`uint32_t __vcpyp(` `  pgentype p);`	Returns predication value `p` as the scalar value.

Note

There is one *Scalar Processing Unit* (SPU) but may be more than one *Tensor Execution Cells* (TECs), you should open one TEC before __vcpy and re-open all TECs after __vcpy.

Taking X1_1204 as an example, you can use __vcpy as follows:

```
__mtctrl0h(0x1, 0x20);  // 0x1, bit[0] is 1, bits[3:1] are 0, which mean that TEC0 is enabled, TEC1/2/3 is disabled.
int32_t value = __vcpy(…);
__mtctrl0h(0xf, 0x20); // 0xf, bits[3:0] are 1, which means all four TECs are enabled.
```

3.2.6 Memory built-in functions

__vload

Table 3-97 __vload built-in functions

Function	Description

`v32int8_t __vload_lsram0_b(`
`  __lsram0 v32int8_t *addr,`
`  v32bool_t p);`
`v32uint8_t __vload_lsram0_b(`
`  __lsram0 v32uint8_t *addr,`
`  v32bool_t p);`
`v16int16_t __vload_lsram0_h(`
`  __lsram0 v16int16_t *addr,`
`  v16bool_t p);`
`v16uint16_t __vload_lsram0_h(`
`  __lsram0 v16uint16_t *addr,`
`  v16bool_t p);`
`v8int32_t __vload_lsram0_w(`
`  __lsram0 v8int32_t *addr,`
`  v8bool_t p);`
`v8uint32_t __vload_lsram0_w(`
`  __lsram0 v8uint32_t *addr,`
`  v8bool_t p);`
	Predicated by predication variable `p`, loads the tensor data from the memory addressed by pointer `addr` and returns the result. Inactive elements in the result are set to zero.
	The pointer only supports getting reference directly from a tensor variable or the element of a tensor array. The following are some examples:
	`__lsram0 v32int8_t in, out;`
	`__lsram0 v32bool_t pg;`
	`out = __vload_lsram0_b(&in, pg);`
	or
	`__lsram0 v32int8_t in[8], out;`
	`__lsram0 v32bool_t pg;`
	`out = __vload_lsram0_b(&in[2], pg);`

Function	Description

`v32int8_t __vload_lsram1_b(`
`  __lsram1 v32int8_t *addr,`
`  v32bool_t p);`
`v32uint8_t __vload_lsram1_b(`
`  __lsram1 v32uint8_t *addr,`
`  v32bool_t p);`
`v16int16_t __vload_lsram1_h(`
`  __lsram1 v16int16_t *addr,`
`  v16bool_t p);`
`v16uint16_t __vload_lsram1_h(`
`  __lsram1 v16uint16_t *addr,`
`  v16bool_t p);`
`v8int32_t __vload_lsram1_w(`
`  __lsram1 v8int32_t *addr,`
`  v8bool_t p);`
`v8uint32_t __vload_lsram1_w(`
`  __lsram1 v8uint32_t *addr,`
`  v8bool_t p);`	

`v32int8_t __vload_gsram0_b(`
`  __gsram0 v32int8_t *addr,`
`  v32bool_t p);`
`v32uint8_t __vload_gsram0_b(`
`  __gsram0 v32uint8_t *addr,`
`  v32bool_t p);`
`v16int16_t __vload_gsram0_h(`
`  __gsram0 v16int16_t *addr,`
`  v16bool_t p);`
`v16uint16_t __vload_gsram0_h(`
`  __gsram0 v16uint16_t *addr,`
`  v16bool_t p);`
`v8int32_t __vload_gsram0_w(`
`  __gsram0 v8int32_t *addr,`
`  v8bool_t p);`
`v8uint32_t __vload_gsram0_w(`
`  __gsram0 v8uint32_t *addr,`
`  v8bool_t p);`	

`v32int8_t __vload_gsram1_b(`
`  __gsram1 v32int8_t *addr,`
`  v32bool_t p);`
`v32uint8_t __vload_gsram1_b(`
`  __gsram1 v32uint8_t *addr,`
`  v32bool_t p);`
`v16int16_t __vload_gsram1_h(`
`  __gsram1 v16int16_t *addr,`
`  v16bool_t p);`
`v16uint16_t __vload_gsram1_h(`
`  __gsram1 v16uint16_t *addr,`	

Function	Description
v16bool_t p); v8int32_t __vload_gsram1_w(__gsram1 v8int32_t *addr, v8bool_t p); v8uint32_t __vload_gsram1_w(__gsram1 v8uint32_t *addr, v8bool_t p);	

__vload_gather

Table 3-98 __vload_gather built-in functions

Function	Description

v32int8_t __vload_gather_b(
__lsram0 v32int8_t *addr,
V16uint16_t offset0,
V16uint16_t offset1);
v32uint8_t __vload_gather_b(
__lsram0 v32uint8_t *addr,
V16uint16_t offset0,
V16uint16_t offset1);
v32int8_t __vload_gather_b(
__lsram1 v32int8_t *addr,
V16uint16_t offset0,
V16uint16_t offset1);
v32uint8_t __vload_gather_b(
__lsram1 v32uint8_t *addr,
V16uint16_t offset0,
V16uint16_t offset1);
	Loads tensor data from different memory addresses calculated by adding the value of scalar variable `addr` to every element value of an offset tensor concatenated with tensor variables `offset0` and `offset1`, then returns the result.
	`offset0` corresponds to the offset of the lower half elements. `offset1` corresponds to the offset of the higher half elements.
	The pointer only supports getting reference directly from a tensor variable or the element of a tensor array. The following are some examples:
	`__lsram0 v32int8_t in, out;`
	`__lsram0 v16uint16_t offset0, offset1;`
	`out = __vload_gather_b(&in, offset0, offset1);`
	or
	`__lsram0 v32int8_t in[8], out;`
	`__lsram0 v16uint16_t offset0, offset1;`
	`out = __vload_gather_b(&in[2], offset0, offset1);`

v16int16_t __vload_gather_h(
__lsram0 v16int16_t *addr,
V16uint16_t offset);
v16uint16_t __vload_gather_h(
__lsram0 v16uint16_t *addr,
V16uint16_t offset);
v16int16_t __vload_gather_h(
__lsram1 v16int16_t *addr,
V16uint16_t offset);
v16uint16_t __vload_gather_h(
__lsram1 v16uint16_t *addr,
V16uint16_t offset);
	Loads tensor data from different memory addresses calculated by adding the value of scalar variable `addr` to every element value of tensor variable `offset`, then returns the result.
	The pointer only supports getting reference directly from a tensor variable or the element of a tensor array. The following are some examples:
	`__lsram0 v16int16_t in, out;`
	`__lsram0 v16uint16_t offset;`
	`out = __vload_gather_h(&in, offset);`
	or
	`__lsram0 v16int16_t in[8], out;`
	`__lsram0 v16uint16_t offset;`
	`out = __vload_gather_h(&in[2], offset);`

v8int32_t __vload_gather_w(__lsram0 v8int32_t *addr, V16uint16_t offset); v8uint32_t __vload_gather_w(__lsram1 v8uint32_t *addr, V16uint16_t offset);	Loads tensor data from different memory addresses calculated by adding the value of scalar variable `addr` to every element value of the lower half of tensor variable `offset`, then returns the result.

`v8int32_t __vload_gather_w(`
`  __lsram0 v8int32_t *addr,`
`  V16uint16_t offset);`
`v8uint32_t __vload_gather_w(`
`  __lsram1 v8uint32_t *addr,`
`  V16uint16_t offset);`
	The pointer only supports getting reference directly from a tensor variable or the element of a tensor array. The following are some examples:
	`__lsram0 v8int32_t in, out;`
	`__lsram0 v16uint16_t offset;`
	`out = __vload_gather_w(&in, offset);`
	or
	`__lsram0 v8int32_t in[8], out;`
	`__lsram0 v16uint16_t offset;`
	`out = __vload_gather_w(&in[2], offset);`

`v32int8_t __vload_gather_mask_b(`
`  __lsram0 v32int8_t *addr,`
`  V16uint16_t offset0,`
`  V16uint16_t offset1,`
`  v32bool_t p);`
`v32uint8_t __vload_gather_mask_b(`
`  __lsram0 v32uint8_t *addr,`
`  V16uint16_t offset0,`
`  V16uint16_t offset1,`
`  v32bool_t p);`
`v32int8_t __vload_gather_mask_b(`
`  __lsram1 v32int8_t *addr,`
`  V16uint16_t offset0,`
`  V16uint16_t offset1,`
`  v32bool_t p);`
`v32uint8_t __vload_gather_mask_b(`
`  __lsram1 v32uint8_t *addr,`
`  V16uint16_t offset0,`
`  V16uint16_t offset1,`
`  v32bool_t p);`
	Predicated by predication variable `p`, loads tensor data from different memory addresses calculated by adding the value of scalar variable `addr` to every element value of an offset tensor concatenated with tensor variables `offset0` and `offset1`, then returns the result. Inactive elements in the result are set to zero.
	`offset0` corresponds to the offset of the lower half elements. `offset1` corresponds to the offset of the higher half elements.
	The pointer only supports getting reference directly from a tensor variable or the element of a tensor array. The following are some examples:
	`__lsram0 v32int8_t in, out;`
	`__lsram0 v16uint16_t offset0, offset1;`
	`__lsram0 v32bool_t pg;`
	`out = __vload_gather_mask_b(&in, offset0, offset1, pg);`
	or
	`__lsram0 v32int8_t in[8], out;`
	`__lsram0 v16uint16_t offset0, offset1;`
	`__lsram0 v32bool_t pg;`
	`out = __vload_gather_mask_b(&in[2], offset0, offset1, pg);`

`v16int16_t __vload_gather_mask_h(`
`  __lsram0 v16int16_t *addr,`
`  V16uint16_t offset,`
`  v16bool_t p);`
`v16uint16_t __vload_gather_mask_h(`
`  __lsram0 v16uint16_t *addr,`
`  V16uint16_t offset,`
`  v16bool_t p);`
`v16int16_t __vload_gather_mask_h(`
`  __lsram1 v16int16_t *addr,`
`  V16uint16_t offset,`
`  v16bool_t p);`
`v16uint16_t __vload_gather_mask_h(`
`  __lsram1 v16uint16_t *addr,`
`  V16uint16_t offset,`
`  v16bool_t p);`
	Predicated by predication variable `p`, loads tensor data from different memory addresses calculated by adding the value of scalar variable `addr` to every element value of tensor variable `offset`, and returns the result. Inactive elements in the result are set to zero.
	The pointer only supports getting reference directly from a tensor variable or the element of a tensor array. The following are some examples:
	`__lsram0 v16int16_t in, out;`
	`__lsram0 v16uint16_t offset;`
	`__lsram0 v16bool_t pg;`
	`out = __vload_gather_mask_h(&in, offset, pg);`
	or
	`__lsram0 v16int16_t in[8], out;`
	`__lsram0 v16uint16_t offset;`
	`__lsram0 v16bool_t pg;`
	`out = __vload_gather_mask_h(&in[2], offset, pg);`

`v8int32_t __vload_gather_mask_w(` `  __lsram0 v8int32_t *addr,`	Predicated by predication variable `p`, loads tensor data from different memory addresses calculated by adding the value of scalar variable `addr` to every element value the lower half of

V16uint16_t offset,
v8bool_t p);
v8uint32_t __vload_gather_mask_w(
__lsram0 v8uint32_t *addr,
V16uint16_t offset,
v8bool_t p);
v8int32_t __vload_gather_mask_w(
__lsram1 v8int32_t *addr,
V16uint16_t offset,
v8bool_t p);
v8uint32_t __vload_gather_mask_w(
__lsram1 v8uint32_t *addr,
V16uint16_t offset,
v8bool_t p);
	tensor variable `offset`, and returns the result. Inactive elements in the result are set to zero.
	The pointer only supports getting reference directly from a tensor variable or the element of a tensor array. The following are some examples:
	`__lsram0 v8int32_t in, out;`
	`__lsram0 v16uint16_t offset;`
	`__lsram0 v8bool_t pg;`
	`out = __vload_gather_mask_w(&in, offset, pg);`
	or
	`__lsram0 v8int32_t in[8], out;`
	`__lsram0 v16uint16_t offset;`
	`__lsram0 v8bool_t pg;`
	`out = __vload_gather_mask_w(&in[2], offset, pg);`

__vstore

Table 3-99 __vstore built-in functions

Function	Description

void __vstore_lsram0_b(
v32int8_t x,
__lsram0 v32int8_t *addr,
v32bool_t p);
void __vstore_lsram0_b(
v32uint8_t x,
__lsram0 v32uint8_t *addr,
v32bool_t p);
void __vstore_lsram0_h(
v16int16_t x,
__lsram0 v16int16_t *addr,
v16bool_t p);
void __vstore_lsram0_h(
v16uint16_t x,
__lsram0 v16uint16_t *addr,
v16bool_t p);
void __vstore_lsram0_w(
v8int32_t x,
__lsram0 v8int32_t *addr,
v8bool_t p);
void __vstore_lsram0_w(
v8uint32_t x,
__lsram0 v8uint32_t *addr,
v8bool_t p);
	Predicated by predication variable `p`, stores tensor data x to the memory addressed by pointer `addr`. Inactive elements in the result memory keep the original value.
	The pointer only supports getting reference directly from a tensor variable or the element of a tensor array. The following are some examples:
	`__lsram0 v32int8_t in, out;`
	`__lsram0 v32bool_t pg;`
	`__vstore_lsram0_b(in, &out, pg);`
	or
	`__lsram0 v32int8_t in, out[8];`
	`__lsram0 v32bool_t pg;`
	`__vstore_lsram0_b(in, &out[2], pg);`

void __vstore_lsram1_b(v32int8_t x, __lsram1 v32int8_t *addr, v32bool_t p); void __vstore_lsram1_b(	

Function	Description
v32uint8_t x, __lsram1 v32uint8_t *addr, v32bool_t p); void __vstore_lsram1_h(v16int16_t x, __lsram1 v16int16_t *addr, v16bool_t p); void __vstore_lsram1_h(v16uint16_t x, __lsram1 v16uint16_t *addr, v16bool_t p); void __vstore_lsram1_w(v8int32_t x, __lsram1 v8int32_t *addr, v8bool_t p); void __vstore_lsram1_w(v8uint32_t x, __lsram1 v8uint32_t *addr, v8bool_t p);	
void __vstore_gsram0_b(v32int8_t x, __gsram0 v32int8_t *addr, v32bool_t p); void __vstore_gsram0_b(v32uint8_t x, __gsram0 v32uint8_t *addr, v32bool_t p); void __vstore_gsram0_h(v16int16_t x, __gsram0 v16int16_t *addr, v16bool_t p); void __vstore_gsram0_h(v16uint16_t x, __gsram0 v16uint16_t *addr, v16bool_t p); void __vstore_gsram0_w(v8int32_t x, __gsram0 v8int32_t *addr, v8bool_t p); void __vstore_gsram0_w(v8uint32_t x, __gsram0 v8uint32_t *addr, v8bool_t p);	
void __vstore_gsram1_b(v32int8_t x, __gsram1 v32int8_t *addr, v32bool_t p);	

Function	Description

`void __vstore_gsram1_b(`
`  v32uint8_t x,`
`  __gsram1 v32uint8_t *addr,`
`  v32bool_t p);`
`void __vstore_gsram1_h(`
`  v16int16_t x,`
`  __gsram1 v16int16_t *addr,`
`  v16bool_t p);`
`void __vstore_gsram1_h(`
`  v16uint16_t x,`
`  __gsram1 v16uint16_t *addr,`
`  v16bool_t p);`
`void __vstore_gsram1_w(`
`  v8int32_t x,`
`  __gsram1 v8int32_t *addr,`
`  v8bool_t p);`
`void __vstore_gsram1_w(`
`  v8uint32_t x,`
`  __gsram1 v8uint32_t *addr,`
`  v8bool_t p);`	

__vstore_scatter

Table 3-100 __vstore_scatter built-in functions

Function	Description

`void __vstore_scatter_b(`
`  v32int8_t x,`
`  __lsram0 v32int8_t *addr,`
`  V16uint16_t offset0,`
`  V16uint16_t offset1);`
`void __vstore_scatter_b(`
`  v32uint8_t x,`
`  __lsram0 v32uint8_t *addr,`
`  V16uint16_t offset0,`
`  V16uint16_t offset1);`
`void __vstore_scatter_b(`
`  v32int8_t x,`
`  __lsram1 v32int8_t *addr,`
`  V16uint16_t offset0,`
`  V16uint16_t offset1);`
`void __vstore_scatter_b(`
`  v32uint8_t x,`
`  __lsram1 v32uint8_t *addr,`
`  V16uint16_t offset0,`
`  V16uint16_t offset1);`
	Stores tensor variable `x` to different memory addresses calculated by adding the value of scalar variable `addr` to every element value of an offset tensor concatenated with tensor variables `offset0` and `offset1`.
	`offset0` corresponds to the offset of the lower half elements. `offset1` corresponds to the offset of the higher half elements.
	The pointer only supports getting reference directly from a tensor variable or the element of a tensor array. The following are some examples:
	`__lsram0 v32int8_t in, out;`
	`__lsram0 v16uint16_t offset0, offset1;`
	`__vstore_scatter_b(in, &out, offset0, offset1);`
	or
	`__lsram0 v32int8_t in, out[8];`
	`__lsram0 v16uint16_t offset0, offset1;`
	`__vstore_scatter_b(in, &out[2], offset0, offset1);`

Function	Description

`void __vstore_scatter_h(`
`  v16int16_t x,`
`  __lsram0 v16int16_t *addr,`
`  V16uint16_t offset);`
`void __vstore_scatter_h(`
`  v16uint16_t x,`
`  __lsram0 v16uint16_t *addr,`
`  V16uint16_t offset);`
`void __vstore_scatter_h(`
`  v16int16_t x,`
`  __lsram1 v16int16_t *addr,`
`  V16uint16_t offset);`
`void __vstore_scatter_h(`
`  v16uint16_t x,`
`  __lsram1 v16uint16_t *addr,`
`  V16uint16_t offset);`
	Stores tensor variable `x` to different memory addresses calculated by adding the value of scalar variable `addr` to every element value of tensor variable `offset`.
	The pointer only supports getting reference directly from a tensor variable or the element of a tensor array. The following are some examples:
	`__lsram0 v16int16_t in, out;`
	`__lsram0 v16uint16_t offset;`
	`__vstore_scatter_h(in, &out, offset);`
	or
	`__lsram0 v16int16_t in, out[8];`
	`__lsram0 v16uint16_t offset;`
	`__vstore_scatter_h(in, &out[2], offset);`

`void __vstore_scatter_w(`
`  v8int32_t x,`
`  __lsram0 v8int32_t *addr,`
`  V16uint16_t offset);`
`void __vstore_scatter_w(`
`  v8uint32_t x,`
`  __lsram0 v8uint32_t *addr,`
`  V16uint16_t offset);`
`void __vstore_scatter_w(`
`  v8int32_t x,`
`  __lsram1 v8int32_t *addr,`
`  V16uint16_t offset);`
`void __vstore_scatter_w(`
`  v8uint32_t x,`
`  __lsram1 v8uint32_t *addr,`
`  V16uint16_t offset);`
	Stores tensor variable `x` to different memory addresses calculated by adding the value of scalar variable `addr` to every element value of the lower half of tensor variable `offset`.
	The pointer only supports getting reference directly from a tensor variable or the element of a tensor array. The following are some examples:
	`__lsram0 v8int32_t in, out;`
	`__lsram0 v16uint16_t offset;`
	`__vstore_scatter_w(in, &out, offset);`
	or
	`__lsram0 v8int32_t in, out[8];`
	`__lsram0 v16uint16_t offset;`
	`__vstore_scatter_w(in, &out[2], offset);`

`void __vstore_scatter_mask_b(`
`  v32int8_t x,`
`  __lsram0 v32int8_t *addr,`
`  V16uint16_t offset0,`
`  V16uint16_t offset1,`
`  v32bool_t p);`
`void __vstore_scatter_mask_b(`
`  v32uint8_t x,`
`  __lsram0 v32uint8_t *addr,`
`  V16uint16_t offset0,`
`  V16uint16_t offset1,`
`  v32bool_t p);`
`void __vstore_scatter_mask_b(`
`  v32int8_t x,`
`  __lsram1 v32int8_t *addr,`
	Predicated by predication variable `p`, stores tensor variable `x` to different memory addresses calculated by adding the value of scalar variable `addr` to every element value of an offset tensor concatenated with tensor variables `offset0` and `offset1`. Inactive elements in the result memory keep the original value.
	`offset0` corresponds to the offset of the lower half elements. `offset1` corresponds to the offset of the higher half elements.
	The pointer only supports getting reference directly from a tensor variable or the element of a tensor array. The following are some examples:
	`__lsram0 v32int8_t in, out;`
	`__lsram0 v16uint16_t offset0, offset1;`
	`__lsram0 v32bool_t pg;`
	`__vstore_scatter_mask_b(in, &out, offset0, offset1, pg);`
	or

Function	Description
V16uint16_t offset0, V16uint16_t offset1, v32bool_t p); void __vstore_scatter_mask_b(v32uint8_t x, __lsram1 v32uint8_t *addr, V16uint16_t offset0, V16uint16_t offset1, v32bool_t p);	__lsram0 v32int8_t in, out[8]; __lsram0 v16uint16_t offset0, offset1; __lsram0 v32bool_t pg; __vstore_scatter_mask_b(in, &out[2], offset0, offset1, pg);

void __vstore_scatter_mask_h(
v16int16_t x,
__lsram0 v16int16_t *addr,
V16uint16_t offset,
v16bool_t p);
void __vstore_scatter_mask_h(
v16uint16_t x,
__lsram0 v16uint16_t *addr,
V16uint16_t offset,
v16bool_t p);
void __vstore_scatter_mask_h(
v16int16_t x,
__lsram1 v16int16_t *addr,
V16uint16_t offset,
v16bool_t p);
void __vstore_scatter_mask_h(
v16uint16_t x,
__lsram1 v16uint16_t *addr,
V16uint16_t offset,
v16bool_t p);
	Predicated by predication variable p, stores tensor variable x to different memory addresses calculated by adding the value of scalar variable addr to every element value of tensor variable offset. Inactive elements in the result memory keep the original value.
	The pointer only supports getting reference directly from a tensor variable or the element of a tensor array. The following are some examples:
	__lsram0 v16int16_t in, out;
	__lsram0 v16uint16_t offset;
	__lsram0 v16bool_t pg;
	__vstore_scatter_mask_h(in, &out, offset, pg);
	or
	__lsram0 v16int16_t in, out[8];
	__lsram0 v16uint16_t offset;
	__lsram0 v16bool_t pg;
	__vstore_scatter_mask_h(in, &out[2], offset, pg);

void __vstore_scatter_mask_w(
v8int32_t x,
__lsram0 v8int32_t *addr,
V16uint16_t offset,
v8bool_t p);
void __vstore_scatter_mask_w(
v8uint32_t x,
__lsram0 v8uint32_t *addr,
V16uint16_t offset,
v8bool_t p);
void __vstore_scatter_mask_w(
v8int32_t x,
__lsram1 v8int32_t *addr,
V16uint16_t offset,
v8bool_t p);
void __vstore_scatter_mask_w(
v8uint32_t x,
__lsram1 v8uint32_t *addr,
	Predicated by predication variable p, stores tensor variable x to different memory addresses calculated by adding the value of scalar variable addr to every element value of the lower half of tensor variable offset. Inactive elements in the result memory keep the original value.
	The pointer only supports getting reference directly from a tensor variable or the element of a tensor array. The following are some examples:
	__lsram0 v8int32_t in, out;
	__lsram0 v16uint16_t offset;
	__lsram0 v8bool_t pg;
	__vstore_scatter_mask_w(in, &out, offset, pg);
	or
	__lsram0 v8int32_t in, out[8];
	__lsram0 v16uint16_t offset;
	__lsram0 v8bool_t pg;
	__vstore_scatter_mask_w(in, &out[2], offset, pg);

Function	Description
V16uint16_t offset, v8bool_t p);	

3.3 DMA built-in functions

This section describes the DMA built-in functions.

3.3.1 DMA built-in function types

Compass C encapsulates DMA operations and supports two types of data transfer—`memcpy` and `stridecpy`. `memcpy` is for continuous data transfer, `stridecpy` is for data transfer with stride.

3.3.2 DMA transfer directions

Compass C supports bi-directional transfer for data and SRAM. SRAM is divided into GSRAM and LSRAM, and SRAM is also divided into two pieces—ping and pong, so there are eight directions in total, LSRAM02DDR, DDR2LSRAM0, LSRAM12DDR, DDR2LSRAM1, GSRAM02DDR, DDR2GSRAM0, GSRAM12DDR, and DDR2LSRAM1. These eight directions are reflected in the names of the DMA functions.

Figure 3-27 DMA transfer directions

3.3.3 DMA modes

The DMA functions of Compass C include three modes—Native, Full, and Single. Because GSRAM is a shared SRAM that can be accessed by multiple TECs simultaneously, the three modes have same behavior for data transfer between DDR and GSRAM. Each TEC has its own LSRAM, so the data distribution between DDR and LSRAM can be different in the three modes.

Native mode

This is the most natural data transfer mode. If there are multiple TECs, the data is stored on the SRAM with the lower TEC number in priority. After calculating the size of the data to be transferred on the LSRAM of each TEC, you can start the DMA and transfer the data to the corresponding LSRAM of each TEC in multiple times. As shown in the following figure, the behavior of `memcpy` and `stridecpy` is the same.

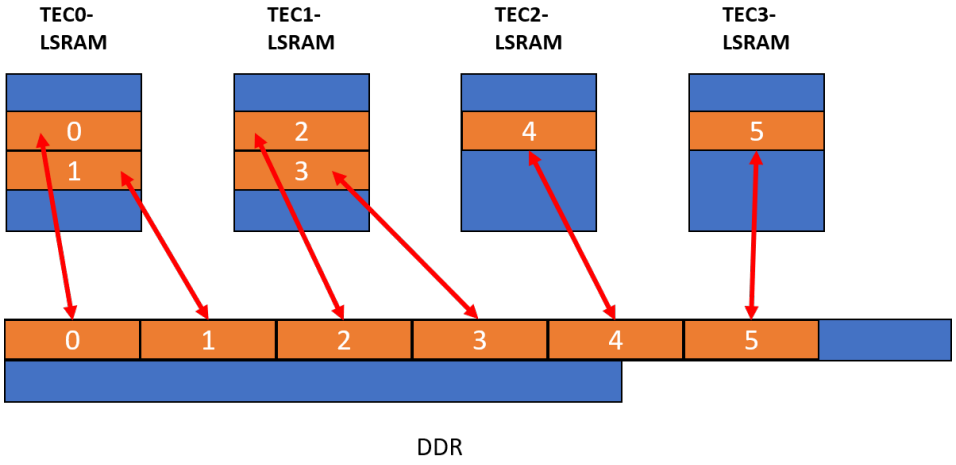

Figure 3-108 Native mode

Full mode

Full mode ensures that all TECs have the same size of data, so the data to be transferred will be equally distributed among all TECs.

In implementation, the full modes of `memcpy` and `stridecpy` are slightly different. `memcpy` calculates the amount of LSRAM data on each TEC before transporting it, while `stridecpy` distributes the data of the width length equally to each TEC with stride as the step size each time, and repeats the process until the data transport is complete.

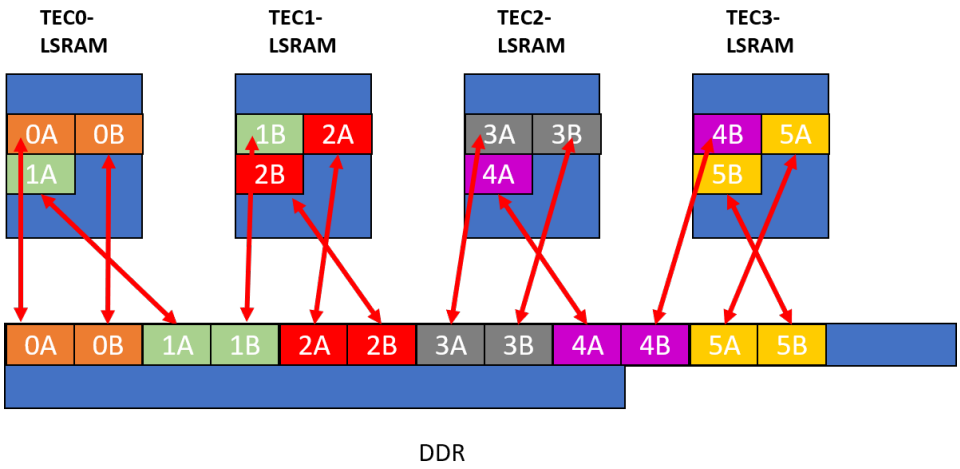

Figure 3-29 Full mode—memcpy

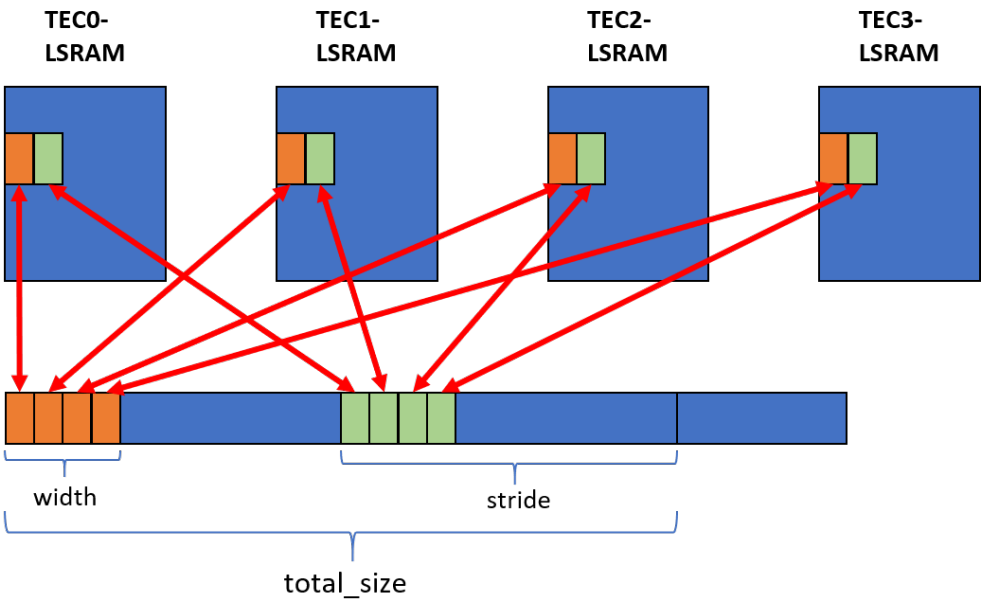

Figure 3-30 Full mode—stridecpy

Single mode

Single mode is the simplest mode. It transports all data to the LSRAM of TEC0.

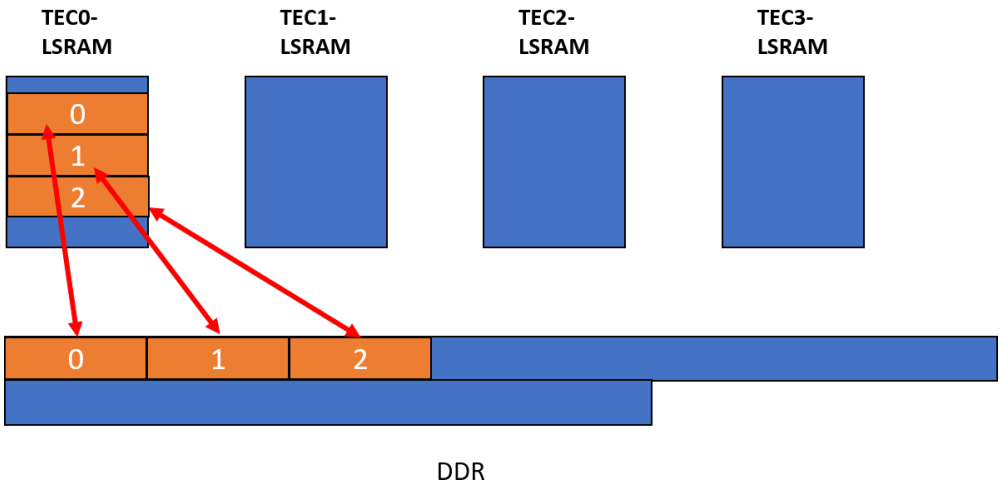

Figure 3-31 Single mode—memcpy

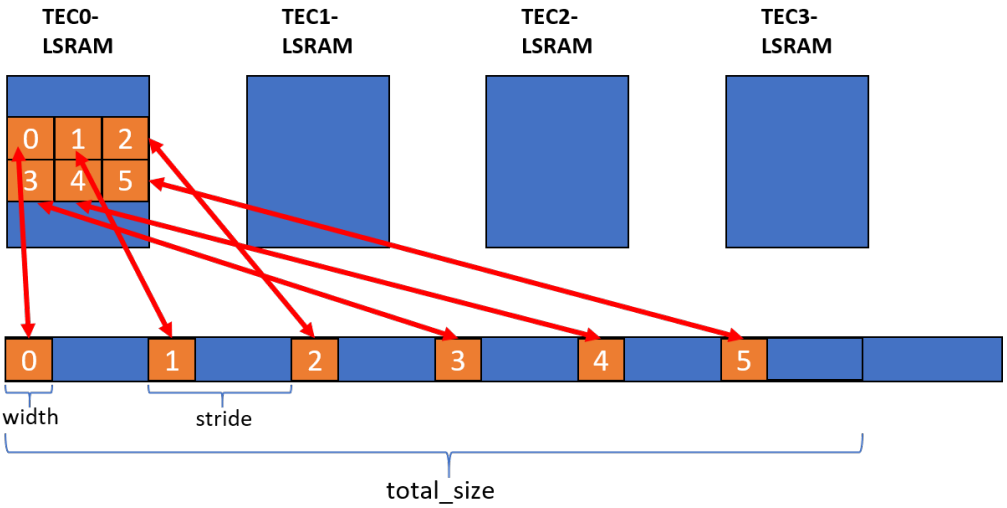

Figure 3-32 Single mode—stridecpy

To easily specify the corresponding mode, Compass C predefines the following enumerations:

```
enum MODE {
    NATIVE,  // Native mode
    FULL,  // Full mode
    SINGLE,  // Single mode
};
```

When using these enumerations, do not make any assumptions about the specific values of these enumerations.

3.3.4 ASID

The NPU applies ASID and introduces the *Address Space Extension* (ASX) to access address space beyond 32 bits. Through ASID, the NPU divides the memory into four regions, and defines the types of data to be stored in each region.

The following table shows the detailed definition of the data types.

Table 3-96 Definition of the ASID

ASID	Description
0	Stores instructions, stacks, and entry function parameters except weight, with the read and write attribute.
1	Stores weight data, with the read-only attribute.
2	Reserved.
3	Reserved.

ASID requires hardware support and support from other build tools and the runtime system. Compass C simplifies ASID related programming. You only need to pass the ASID enumeration type where the data resides when calling the DMA functions, and pass the -DASID compilation parameters to the Compass C compiler `aipucc`.

To easily specify the corresponding ASID, Compass C predefines the following enumerations:

```
enum AS {
```

```
        REGION0,  // ASID 0
        REGION1,  // ASID 1
        REGION2,  // ASID 2
        REGION3,  // ASID 3
    };
```

When using these enumerations, do not make any assumptions about the specific values of these enumerations.

3.3.5 DMA built-in functions list

The DMA built-in functions are divided into two types and eight directions, so there are 16 functions.

The following table lists the DMA built-in functions.

Table 3-97 DMA built-in functions

Function	Description

`int32_t __memcpy_lsram02ddr(`
`uint8_t *ddr,`
`__lsram0 void *sram,`
`uint32_t size,`
`enum MODE m = NATIVE,`
`enum AS as = REGION0);`
`int32_t __memcpy_lsram12ddr(`
`uint8_t *ddr,`
`__lsram1 void *sram,`
`uint32_t size,`
`enum MODE m = NATIVE,`
`enum AS as = REGION0);`
	Copies the data of the `size` byte length from the LSRAM buffer to which `sram` points to the DDR buffer that `ddr` points to. `m` and `as` are enumerated variables, representing the function mode and ASID value respectively. If the function mode and ASID value are not specified, the default values NATIVE and REGION0 are used. For the specific values and definitions, see Sections 3.3.3 and 3.3.4.
	Note: The `size` should be no larger than the size of the buffer.
	If successful, 0 is returned, otherwise a non-zero value is returned.

`int32_t __memcpy_ddr2lsram0(`
`__lsram0 void *sram,`
`uint8_t *ddr,`
`uint32_t size,`
`enum MODE m = NATIVE,`
`enum AS as = REGION0);`
`int32_t __memcpy_ddr2lsram1(`
`__lsram1 void *sram,`
`uint8_t *ddr,`
`uint32_t size,`
`enum MODE m = NATIVE,`
`enum AS as = REGION0);`
	Copies the data of the `size` byte length from the DDR buffer to which `ddr` points to the LSRAM buffer to which `sram` points. `m` and `as` are enumerated variables, representing the function mode and ASID value respectively. If the function mode and ASID value are not specified, the default values NATIVE and REGION0 are used. For the specific values and definitions, see Sections 3.3.3 and 3.3.4.
	Note: The `size` should be no larger than the size of the buffer.
	If successful, 0 is returned, otherwise a non-zero value is returned.

`int32_t __memcpy_gsram02ddr(`
`uint8_t *ddr,`
`__gsram0 void *sram,`
`uint32_t size,`
`enum MODE m = NATIVE,`
`enum AS as = REGION0);`
`int32_t __memcpy_gsram12ddr(`
`uint8_t *ddr,`
	Copies the data of the `size` byte length from the GSRAM buffer to which `SRAM` points to the DDR buffer to which `ddr` points. `m` and `as` are enumerated variables, representing the function mode and ASID value respectively. If the function mode and ASID value are not specified, the default values NATIVE and REGION0 are used. For the specific values and definitions, see Sections 3.3.3 and 3.3.4. GSRAM is a public SRAM that is shared by multiple TECs and only one slice is used at a time, so the data distribution in the three function modes is the same.

Function	Description

`__gsram1 void *sram,`
`uint32_t size,`
`enum MODE m = NATIVE,`
`enum AS as = REGION0);`
	Note: The `size` should be no larger than the size of the buffer.
	If successful, 0 is returned, otherwise a non-zero value is returned.

`int32_t __memcpy_ddr2gsram0(`
`__gsram0 void *sram,`
`uint8_t *ddr,`
`uint32_t size,`
`enum MODE m = NATIVE,`
`enum AS as = REGION0);`
`int32_t __memcpy_ddr2gsram1(`
`__gsram1 void *sram,`
`uint8_t *ddr,`
`uint32_t size,`
`enum MODE m = NATIVE,`
`enum AS as = REGION0);`
	Copies the data of the `size` byte length from the DDR buffer to which `ddr` points to the GSRAM buffer to which `sram` points. `m` and `as` are enumerated variables, representing the function mode and ASID value respectively. If the function mode and ASID value are not specified, the default values NATIVE and REGION0 are used. For the specific values and definitions, see Sections 3.3.3 and 3.3.4. GSRAM is a public SRAM that is shared by multiple TECs and only one slice is used at a time, so the data distribution in the three function modes is the same.
	Note: The `size` should be no larger than the size of the buffer.
	If successful, 0 is returned, otherwise a non-zero value is returned.

`int32_t __stridecpy_lsram02ddr(`
`uint8_t *ddr,`
`__lsram0 void *sram,`
`Uint16_t width,`
`uint32_t stride,`
`uint32_t size,`
`enum MODE m = NATIVE,`
`enum AS as = REGION0);`
`int32_t __stridecpy_lsram12ddr(`
`uint8_t *ddr,`
`__lsram1 void *sram,`
`Uint16_t width,`
`uint32_t stride,`
`uint32_t size,`
`enum MODE m = NATIVE,`
`enum AS as = REGION0);`
	With `stride` as the step and the `width` bytes as the valid data, copies the data from the LSRAM buffer to which `sram` points to the DDR buffer to which `ddr` points in the dispersed approach. The `size` is the total number of bytes occupied by the DDR buffer, which must be an integral multiple of `stride`. `m` and `as` are enumerated variables, representing the function mode and ASID value respectively. If the function mode and ASID value are not specified, the default values NATIVE and REGION0 are used. For the specific values and definitions, see Sections 3.3.3 and 3.3.4. In Full mode, `width` should be an integral multiple of the number of TECs.
	Note: The `size` should be no larger than the size of the DDR buffer. The `stride` should be no larger than `size`. The `width` should be no larger than `stride`. The `size` must be an integral multiple of `stride.`
	If successful, 0 is returned, otherwise a non-zero value is returned.

`int32_t __stridecpy_ddr2lsram0(`
`__lsram0 void *sram,`
`uint8_t *ddr,`
`uint16_t width,`
`uint32_t stride,`
`uint32_t size,`
`enum MODE m = NATIVE,`
`enum AS as = REGION0);`
`int32_t __stridecpy_ddr2lsram1(`
`__lsram1 void *sram,`
`uint8_t *ddr,`
`uint16_t width,`
`Uint32_t stride,`
`uint32_t size,`
`enum MODE m = NATIVE,`
	With `stride` as the step and the `width` bytes as the valid data, copies the data from the DDR buffer to which `ddr` points to the LSRAM buffer to which `sram` points in the centralized approach. The `size` is the total number of bytes occupied by the DDR buffer, which must be an integral multiple of `stride`. `m` and `as` are enumerated variables, representing the function mode and ASID value respectively. If the function mode and ASID value are not specified, the default values NATIVE and REGION0 are used. For the specific values and definitions, see Sections 3.3.3 and 3.3.4. In Full mode, `width` should be an integral multiple of the number of TECs.
	Note: The `size` should be no larger than the size of the DDR buffer. The `stride` should be no larger than `size`. The `width` should be no larger than `stride`. The `size` must be an integral multiple of `stride.`
	If successful, 0 is returned, otherwise a non-zero value is returned.

Function	Description
`enum AS as = REGION0);`	

`int32_t __stridecpy_gsram02ddr(`
`uint8_t *ddr,`
`__gsram0 void *sram,`
`Uint16_t width,`
`uint32_t stride,`
`uint32_t size,`
`enum MODE m = NATIVE,`
`enum AS as = REGION0);`
`int32_t __stridecpy_gsram12ddr(`
`uint8_t *ddr,`
`__gsram1 void *sram,`
`Uint16_t width,`
`uint32_t stride,`
`uint32_t size,`
`enum MODE m = NATIVE,`
`enum AS as = REGION0);`
	With `stride` as the step and the `width` bytes as the valid data, copies the data from the GSRAM buffer to which `sram` points to the DDR buffer to which `ddr` points in the dispersed approach. The `size` is the total number of bytes occupied by the DDR buffer, which must be an integral multiple of `stride`. `m` and `as` are enumerated variables, representing the function mode and ASID value respectively. If the function mode and ASID value are not specified, the default values NATIVE and REGION0 are used. For the specific values and definitions, see Sections 3.3.3 and 3.3.4. GSRAM is a public SRAM that is shared by multiple TECs and only one slice is used at a time, so the data distribution in the three function modes is the same.
	Note: The `size` should be no larger than the size of the DDR buffer. The `stride` should be no larger than `size`. The `width` should be no larger than `stride`. The `size` must be an integral multiple of `stride.`
	If successful, 0 is returned, otherwise a non-zero value is returned.

`int32_t __stridecpy_ddr2gsram0(`
`__gsram0 void *sram,`
`uint8_t *ddr,`
`uint16_t width,`
`uint32_t stride,`
`uint32_t size,`
`enum MODE m = NATIVE,`
`enum AS as = REGION0);`
`int32_t __stridecpy_ddr2gsram1(`
`__gsram1 void *sram,`
`uint8_t *ddr,`
`uint16_t width,`
`uint32_t stride,`
`uint32_t size,`
`enum MODE m = NATIVE,`
`enum AS as = REGION0);`
	With `stride` as the step and the `width` bytes as the valid data, copies the data from the DDR buffer to which `ddr` points to the GSRAM buffer to which `sram` points in the centralized approach. The `size` is the total number of bytes occupied by the DDR buffer, which must be an integral multiple of `stride`. `m` and `as` are enumerated variables, representing the function mode and ASID value respectively. If the function mode and ASID value are not specified, the default values NATIVE and REGION0 are used. For the specific values and definitions, see Sections 3.3.3 and 3.3.4. GSRAM is a public SRAM that is shared by multiple TECs and only one slice is used at a time, so the data distribution in the three function modes is the same.
	Note: The `size` should be no larger than the size of the DDR buffer. The `stride` should be no larger than `size`. The `width` should be no larger than `stride`. The `size` must be an integral multiple of `stride.`
	If successful, 0 is returned, otherwise a non-zero value is returned.

`int32_t __memset_ddr(`
`void *ddr,`
`uint32_t imm_data,`
`uint32_t size,`
`enum AS as = REGION0,`
`enum DU du = BYTE);`
	Fills DDR buffer `ddr` with `imm_data`. The size is the total number of bytes occupied by the DDR buffer.
	`du` is the data unit enum with value BYTE/HALF/WORD.
	• If `du` is BYTE, the lower 8bits of `imm_data` fill to the DDR buffer.
	• If `du` is HALF, the lower 16 bits of `imm_data` fill to the DDR buffer.
	• If `du` is WORD, the total 32-bit `imm_data` fills to the DDR buffer.

The following example shows how to use the DMA functions to copy data from the DDR to LSRAM0 of the 4 TECs, perform calculations, and then copy the results from LSRAM0 of the 4 TECs to the output buffer in the DDR.

```
__lsram0 v8int32_t lsram_input[128];
```

```
__lsram0 v8int32_t lsram_output[4];

__entry void mylayer(uint32_t *input, uint32_t *output) {

    __memcpy_ddr2lsram0(lsram_input, (uint8_t *)input,
sizeof(v8int32_t)*128*4);

    // do computations

    __memcpy_lsram02ddr((uint8_t *)output, lsram_output,
sizeof(v8int32_t)*4*4);

}
```

3.3.6 DMA low-level built-in functions

The high-level DMA built-in functions that are describes in the preceding sections can meet most of the usage scenarios. It is recommended that you use these functions as much as possible when programming. Meanwhile, there are also low-level built-in functions that are closer to the underlying hardware. When the high-level functions cannot meet your requirements, you can use these low-level functions to implement more complex functions.

Table 3-98 DMA low-level built-in functions

Function	Description

`uint32_t __get_lsram0_addr(`
`  uint32_t offset,`
`  uint32_t tec_no);`
`uint32_t __get_lsram1_addr(`
`  uint32_t offset,`
`  uint32_t tec_no);`
	Gets the absolute address of the variable on LSRAM.
	The `offset` is a uint32_t value of the pointer to the variable after the type forced conversion.
	The `tec_no` is the TEC number that starts from 0. It can only be an immediate number here.

`uint32_t __get_gsram0_addr(`
`  uint32_t offset);`
`uint32_t __get_gsram1_addr(`
`  uint32_t offset);`
	Gets the absolute address of the variable on GSRAM.
	The `offset` is a uint32_t value of the pointer to the variable after the type forced conversion.

`int32_t __cfgdma(`
`  uint32_t ddr_addr,`
`  uint32_t sram_addr,`
`  uint32_t ddr_width,`
`  uint32_t sram_width,`
`  uint32_t ddr_stride,`
`  uint32_t sram_stride,`
`  uint32_t ddr_total_size,`
`  uint32_t sram_total_size,`
`  uint32_t channel_no);`
	Configures data size related parameters for DMA channel `channel_no`. Different hardware has different channel numbers. It is recommended that you only use 0–3. It will return 0 if configure successfully.
	• The `ddr_addr` is a uint32_t value of the pointer to the DDR variable after the type forced conversion.
	• The `sram_addr` is the obtained absolute address of the variable on LSRAM or GSRAM when calling the preceding functions (__get_lsram0_addr/__get_lsram1_addr/__get_gsram0_addr/__get_gsram1_addr).
	• The `ddr_width` and `sram_width` are the valid data bytes of the corresponding buffer under the specific step, and should be less than 0x10000.
	• The `ddr_stride` and `sram_stride` are the steps of the corresponding buffer and should be less than 0x10000.
	• The `ddr_total_size` and `sram_total_size` are the total number of bytes occupied by data in the corresponding buffer and should be less than 0x1000000.
	• The `ddr_total_size` must be an integral multiple of `ddr_stride`, and `ddr_total_size` >= `ddr_stride` >= `ddr_width`.
	• The `sram_total_size` must be an integral multiple of `sram_stride`, and `sram_total_size` >= `sram_stride` >= `sram_width`.

Function	Description

`int32_t __cfgdma_v2(`
`uint32_t ddr_addr,`
`uint32_t sram_addr,`
`uint32_t ddr_width,`
`uint32_t sram_width,`
`uint32_t ddr_stride,`
`uint32_t sram_stride,`
`uint32_t ddr_gap,`
`uint32_t sram_gap,`
`uint32_t ddr_trans_num,`
`uint32_t sram_trans_num,`
`uint32_t ddr_trans_size,`
`uint32_t sram_trans_size,`
`uint32_t imm_data,`
`uint32_t channel_no);`
	Configures data size related parameters for DMA channel `channel_no`. Different hardware has different channel numbers. It is recommended that you only use 0–3.
	• The `ddr_addr` is a uint32_t value of the pointer to the DDR variable after the type forced conversion.
	• The `sram_addr` is the obtained absolute address of the variable on LSRAM or GSRAM when calling the preceding functions (__get_lsram0_addr/__get_lsram1_addr/__get_gsram0_addr/__get_gsram1_addr).
	• The `ddr_width` and `sram_width` are the valid data bytes of the corresponding buffer under the specific step and must be less than 0x10000.
	• The `ddr_stride` and `sram_stride` are the steps of the corresponding buffer and must be less than 0x1000000. This is different from `ddr_stride` and `sram_stride` in the __cfgdma function.
	• The `ddr_gap` and `sram_gap` are the data gap of the corresponding buffer, and must be less than 0x1000000.
	• The `ddr_trans_num` and `sram_trans_num` are the data transfer times, and must be less than 0x100.
	• The `ddr_trans_size` and `sram_trans_size` are the total transfer size of bytes and must be less than 0x1000000. They are different from `ddr_total_size` and `sram_total_size` in the __cfgdma function. `ddr_trans_size` should be equal to `sram_trans_size`.
	• The `ddr_trans_size` must be an integral multiple of `ddr_width`, and `sram_trans_size` must be an integral multiple of `sram_width`.
	• The `imm_data` is the immediate value set to the corresponding buffer, which is only used in memset.

`void __startdma(uint32_t cfg_info);`	Configures DMA transfer direction parameters and starts DMA. The configuration information such as transfer direction and channel number is coded in cfg_info. The encoding format of cfg_info is as follows: [31:30]: ASID, 0b00-REGION0, 0b01-REGION1, 0b10-REGION2, 0b11-REGION3. [17:9]: Each bit corresponds to the transfer direction of a channel. `1` Indicates that the data is transferred from SRAM to DDR. `0` Indicates that the data is transferred from DDR to SRAM. [8:0]: Each bit corresponds to a channel. If a bit is set to 1, it indicates that the corresponding channel is enabled and starts the data transfer.
`void __waitdma(void);`	Waits for the completion of DMA transfer.
`void __flush_cache(void);`	Flushes the cache and writes the cached data to DDR. When DMA transfers DDR data or dumps DDR data, hardware does not guarantee the consistency of the data in the data cache and the data in the DDR. At this time, the __flush_cache function needs to be used to refresh the cache to ensure data consistency.

3.4 HWA/AIFF built-in functions

This section describes the HWA/AIFF built-in functions. Currently, only the table look-up family functions are provided. Table look-up using HWA/AIFF is different from using the tensor processor. HWA/AIFF can handle more efficiently because it can process table look-up in batches using hardware.

3.4.1 LUT unit introduction

HWA/AIFF uses LUT which is a table look-up function unit in SDP/ITP to perform the table look-up. The table in LUT has 257 entries, and each entry has 16-bit data. The LUT unit can handle the table look-up of 8-bit or 16-bit data.

The addresses such as input address, output address and table address passed to HWA/AIFF must be 128 bits aligned.

AIFF also supports ASID. There are three places where you can set ASID:

- The parameter ASID. Offline built parameters such as weight, padding, and per channel data will map to this ASID.
- The working space ASID. The data for input or output activation, and elementwise input will map to this ASID.
- The descriptor ASID. The descriptor will map to this ASID.

For more information about LUT, see the relevant *Technical Reference Manuals* (TRMs) delivered with the Arm China Zhouyi products.

3.4.2 LUT built-in functions list

The input and output data type and precision are used to classify different interfaces. The interface is named as __lut_[in type][in precision][out type][out precision]. For example, __lut_i8i8(uint32_t num, int8_t *lut, int8_t *input, int8_t *output, enum AS as_lut, enum AS as_data), which means the data in the table is 8-bit, the input is signed int8, and the output is signed int8. There are eight interfaces.

The input data can be int8_t, uint8_t, int16_t and uint16_t. The output data can be int8_t, uint8_t, int16_t and uint16_t. The table data can be int8_t and int16_t. The precision of the input data, output data and lut data must be the same.

Table 3-99 HWA/AIFF built-in functions

Function	Description

`int32_t __lut_[u/i]8[u/i]8(`
`uint32_t num,`
`int8_t *lut,`
`[u]int8_t *input,`
`[u]int8_t *output`
`enum AS as_lut,`
`enum AS as_data);`
	The input data of which the precision of 8-bit looks up the table which is constructed by lut, and gets the output data of which the precision is also 8-bit.
	• The `num` means the elements number of input data.
	• The `as_lut` is the region in which the lut table is stored.
	• The `as_data` is the region in which the input and output data is stored.

`int32_t __lut_[u/i]16[u/i]16(`
`uint32_t num,`
`int16_t *lut,`
`[u]int16_t *input,`
`[u]int16_t *output`
`enum AS as_lut,`
`enum AS as_data);`
	The input data of which the precision of 16-bit looks up the table which is constructed by lut, and gets the output data of which the precision is also 16-bit.
	• The `num` means the elements number of input data.
	• The `as_lut` is the region in which the lut table is stored.
	• The `as_data` is the region in which the input and output data is stored.

3.5 Miscellaneous built-in functions

In addition to the functions described in the preceding sections, Compass C also provides some additional built-in functions for easy programming.

Table 3-100 Miscellaneous built-in functions

Function	Description
`uint32_t __gettecnum(void);`	Gets the number of TECs.
`void __prefetch(void);`	Prefetches instructions from the current PC, up to 32K.

`void __mtctrl0(`
`  uint32_t value,`
`  uint32_t offset);`
`void __mtctrl1(`
`  uint32_t value,`
`  uint32_t offset);`
`void __mtctrl2(`
`  uint32_t value,`
`  uint32_t offset);`
	Configures the system register value with address offset `offset` on system register group 0/1/2 to `value`, and returns immediately after initiating the operation.

`void __mtctrl0h(`
`  uint32_t value,`
`  uint32_t offset);`
`void __mtctrl1h(`
`  uint32_t value,`
`  uint32_t offset);`
`void __mtctrl2h(`
`  uint32_t value,`
`  uint32_t offset);`
	Configures the system register value with address offset `offset` on system register group 0/1/2 to `value`, and waits in the pipeline until the configuration is complete.

`uint32_t __mfctrl0(`
`  uint32_t value,`
`  uint32_t offset);`
`uint32_t __mfctrl1(`
`  uint32_t value,`
`  uint32_t offset);`
`uint32_t __mfctrl2(`
`  uint32_t value,`
`  uint32_t offset);`
	Reads the system register value with address offset `offset` on system register group 0/1/2 to the variable `value`, and returns immediately after initiating the operation.

`uint32_t __mfctrl0h(`
`  uint32_t value,`
`  uint32_t offset);`
`uint32_t __mfctrl1h(`
`  uint32_t value,`
`  uint32_t offset);`
`uint32_t __mfctrl2h(`
`  uint32_t value,`
`  uint32_t offset);`
	Reads the system register value with address offset `offset` on system register group 0/1/2 to the variable `value`, and waits in the pipeline until the read is complete.

`void __vdivrem_v8int32(` `  __lsram0 v8int32_t *dividend,`	Gets the quotient and the remainder. The dividend, divisor, quotient, and remainder should be pointers that point to the `__lsram0` tensor variables.

Function	Description

`__lsram0 v8int32_t *divisor,`
`__lsram0 v8int32_t *quotient,`
`__lsram0 v8int32_t *remainder);`
`void __vdivrem_v8uint32(`
`__lsram0 v8uint32_t *dividend,`
`__lsram0 v8uint32_t *divisor,`
`__lsram0 v8uint32_t *quotient,`
`__lsram0 v8uint32_t *remainder);`
`void __vdivrem_v16int16(`
`__lsram0 v16int16_t *dividend,`
`__lsram0 v16int16_t *divisor,`
`__lsram0 v16int16_t *quotient,`
`__lsram0 v16int16_t *remainder);`
`void __vdivrem_v16uint16(`
`__lsram0 v16uint16_t *dividend,`
`__lsram0 v16uint16_t *divisor,`
`__lsram0 v16uint16_t *quotient,`
`__lsram0 v16uint16_t *remainder);`
`void __vdivrem_v32int8(`
`__lsram0 v32int8_t *dividend,`
`__lsram0 v32int8_t *divisor,`
`__lsram0 v32int8_t *quotient,`
`__lsram0 v32int8_t *remainder);`
`void __vdivrem_v32uint8(`
`__lsram0 v32uint8_t *dividend,`
`__lsram0 v32uint8_t *divisor,`
`__lsram0 v32uint8_t *quotient,`
`__lsram0 v32uint8_t *remainder);`
	If the element value of divisor is 0, the quotient and the remainder are meaningless.

Chapter 4
Inline assembly

This chapter describes the inline assembly function module of Compass C.

It contains the following sections:

4.1 Syntax

An inline assembly is a feature of the C programming model in the NPU, which allows you to insert assembly instructions in the C language source code. The compiler can distinguish these inline assembly snippet codes and treat them as low-level codes.

The inline assembly can be used in three situations:

- Optimization: You can use inline assembly code to implement the most performance-sensitive parts of the algorithms of your program. Machine code may be more efficient than what the compiler generates.
- Accessing the NPU specific instructions: Some specific instructions in the NPU are not supported in the C programming model. All these instructions can be accessed with inline assembly by C programmers.
- Other special functions that the C programming model does not support, such as special calling conventions, hardware interrupts, and special directives for the linker and assembler.

You should be familiar with the assembly language of the NPU. For more information about the Zhouyi assembly language, see the *Arm China Zhouyi Compass Assembly Programming Guide*.

4.1.1 Basic types

The keyword of the inline assembly is `__asm__`.

The standard inline assembly should be used to identify instruction operands with C variables. It allows you to set the input operands list, output operands list and clobbers list that uses colons (`:`) to delimit the operand lists after the assembler template.

The standard syntax is as follows:

```
__asm__ qualifier (
    assembler template
    : output operands list
    : input operands list
    : Clobbers list
);
```

There are four parts of parameters to construct the main function of inline assembly code. For more information about the parameters, see *4.1.2 Parameters*.

There are two different types of inline assembly code styles that have different token types to indicate the operands placeholder. The following are two simple examples:

- Type 1:

```
int32_t byter1;
int32_t byter2 = 1;
int32_t byter3 = 2;
__asm__(
    "add %[namex], %[namey], %[namez];\n\t"
    : [namex] "=&r"(byter1)
    : [namey] "r"(byter2), [namez] "r"(byter3)
);
```

In this type, `%[namex]`, `%[namey]`, and `%[namez]` in the assembly template string can be treated as the named operands placeholders that are mapped to final variables in the following output or input list.

- Type 2:

```
int32_t byter1;

int32_t byter2 = 1;

int32_t byter3 = 2;

__asm__(

    "add %0, %1, %2;\n\t"

    : "=&r"(byter1))             // byter1 bind with %0

    : "r"(byter2), "r"(byter3)   // byter2 bind with %1 and byter3 bind with
%2

);
```

In this type, `%0`, `%1`, and `%2` in the assembly template string are the position numbers that can be treated as the anonymous operand placeholders. They are linked to final variables in the following output or input list by the listed order.

Arm China recommends that you follow Type 1 to write inline assembly code. The compiler supports both types and you can choose which one that you want to use.

4.1.2 Parameters

Assembly template

This is a literal string that is the template for the assembler code. It is a combination of fixed text and tokens that refer to the input and output parameters. The string can contain any instructions recognized by the assembler, including directives. The compiler does not parse the assembler instructions themselves and does not know what they mean or even whether they are valid assembler inputs. Operands with the specific register or immediate are identified the same as assembly language (without any inline assembler prefix symbol).

With the assembly template, you can use:

- Semicolons (`;`) that are used in assembly code to separate multiple assembler instructions in a single inline assembly string.
- A newline character (`\n`) to break a line.
- A tab character (`\t`) to move to the instruction field for the pretty assembly code format.

You can also write bundled instructions because they comply with the normal syntax of assembly code. The single instruction is also supported.

For more information about the assembly instruction syntax, see the *Arm China Zhouyi Compass Assembly Programming Guide*.

Operand symbolic names can be exchanged by the operand position number (zero-based). This means that `%[namex]` can be exchanged by `%0`. In this case, `[asmSymbolicName]` in the output operands list and input operands list should be ignored.

The following are some special format strings that are supported:

- `%%`: Outputs a single % into the assembler code.
- `%=`: Outputs a number that is unique to each instance of inline assembly code in the entire compilation. This option is useful when creating local labels and referring to them multiple times in a single template that generates multiple assembler instructions.

Output operands list

This list is a comma-separated list of C variables that are modified by the instructions in the assembler template. An empty list is permitted.

If the input operands list is not empty, the colon symbol of the output operands list cannot be omitted. Otherwise, the colon symbol should be omitted when the input operands list is empty.

The following is a simple example for these cases:

```
int32_t byter1 = 0x1;
// code below is for empty output operands list but cannot omit
// the colon symbol.
__asm__(
    "mtctrl %0, #0x10;\n\t"
    :
    : "r"(byter1)
);
// code below is for empty output operands list and can omit
// the colon symbol.
__asm__(
    "nop;\n\t"
);
```

The syntax is as follows:

```
[asmSymbolicName] constraint (CVariableName)
```

Where:

`[asmSymbolicName]` is used to specify a symbolic name for the operand when using named operands placeholders in the assembly template, such as `[namex]`. Otherwise, `[asmSymbolicName]` should be omitted.

`constraint` is a string constant that specifies constraints of the operand. See the following constraints for details.

`CVariableName` specifies a C value to bind with an operand, which is typically a variable name. It must be writable.

A constraint is a string full of letters. The following are the basic letters that are allowed:

- `r`: This register operand is allowed if it is in a scalar register.
- `t`: This register operand is allowed if it is in a tensor register.
- `a`: This register operand is allowed if it is in an accumulator register.
- `p`: This register operand is allowed if it is in a predicate register.

Multiple basic letters can be combined as constraint operands. These constraints are represented as multiple alternatives and the compiler can choose the best one as constraint operands.

Some modifier characters can be used to modify the constraint letters. The following are the modifiers that are allowed:

- `=`: This operand is written by this instruction.
- `+`: This operand is both read and written by the instruction. Others without `=` and `+` can only be read.

- &: This operand is an early-clobbered operand, which is written before the instruction finishes using the input operands. Only written operands can use &.

Input operands list

This list is a comma-separated list of C expressions that are read by the instructions in the assembler template. An empty list is permitted.

If the clobbers list is not empty, the colon symbol of the output operands list and input operands list cannot be omitted. Otherwise, the colon symbol should be omitted when the input operands list is empty.

The following is a simple example for these cases:

```
int32_t byter1;
// code below is for empty input operands list but cannot omit
// the colon symbol.
__asm__(
    "mfctrl r1, #0x10;\n\t"
    "add %0, r1, r1;\n\t"
    : "=&r"(byter1)
    :
    : "r1"
);
// code below is for empty input operands list and can omit
// the colon symbol.
__asm__(
    "mfctrl %0, #0x10;\n\t"
    : "=&r"(byter1)
);
```

The syntax is as follows:

```
[asmSymbolicName] constraint (CExpression)
```

Where:

`[asmSymbolicName]` and `constraint` are the same as the ones in the output operands list.

`CExpression` specifies a C variable or expression that is passed to the inline assembly as an input.

Clobbers list

This list is a comma-separated list of registers or other values that are changed by the assembler template, beyond those listed in the output operands list section. An empty list is permitted.

The colon symbol should be omitted when the clobbers list is empty.

4.1.3 Qualifiers

- `volatile`: The typical use of extended `__asm__` statements is to manipulate input values to produce output values. However, your `__asm__` statements may also produce side effects. If so, you may need to use the `volatile` qualifier to disable certain optimizations.
- `inline`: If you use an `inline` qualifier, the assembler will take the inline assembly code as the possible smallest size.

4.2 Example

4.2.1 A simple example

The inline assembly code inputs two vector variables from DDR to lsram0, adds them by each element, then loads other vector data from lsram1 and multiplies them together under the mask of a predication variable. After then, the third vector data is used to have a max calculation with the multiplied result under the same predication variable. Finally, the maximum result vector data is put to another variable on lsram0. This variable saves the data in the final variable to DDR.

```
#include <aipu.h>


__entry void test(uint32_t ddr_adr_in, uint32_t ddr_adr_out)
{
    __lsram0 v32int8_t u1, u2;
    __lsram0 v16int16_t res, u3;
    int32_t offset = 10;
    __memcpy_ddr2lsram0((__lsram0 void*)&u1, (uint8_t *)(ddr_adr_in),
32);
    __memcpy_ddr2lsram0((__lsram0 void*)&u2, (uint8_t *)(ddr_adr_in +
32), 32);
    __memcpy_ddr2lsram0((__lsram0 void*)&u3, (uint8_t *)(ddr_adr_in +
64), 32);
    __memcpy_ddr2lsram1((__lsram1 void*)&offset, (uint8_t *)(ddr_adr_in +
96), 32);
    uint32_t imm;
    imm = *((uint32_t *)(ddr_adr_in));

    __lsram0 v16bool_t mask;
    mask = __vmov_h(imm);

    __asm__ (
      "add.b t1, %[u1], %[u2] & nop & ld t3, [a0, #10] & nop;\n\t"
      "nop & mul.s.hbb t2, t1, t3, %[p0] & nop & nop;\n\t"
      "max.s.h %[result], t2, %[u3], %[p0] & nop & nop & nop;"
      : [result] "=&t" (res)
      : [u1] "t" (u1), [u2] "t" (u2), [u3] "t" (u3), [p0] "p" (mask)
      : "t1", "t2", "t3"
    );

    __memcpy_lsram02ddr((uint8_t *)ddr_adr_out, (__lsram0 void*)&res,
32);;
```

```
    return;
}
```

4.2.2 Notions

Although there are many advantages in inline assembly, its disadvantages should not be ignored.

- The compiler is more complicated and difficult to analyze and optimize C language code with inline assembly inserted in.
- Inline assembly makes program maintenance more difficult.
- Inline assembly code must be inside a function.

Arm China recommends that you use built-in functions supported by the compiler instead of the inline assembly when possible.

For the inline assembly module, do not use:

- Jump instructions to change the control flow, which will make the compiler difficult to control the code.
- The `goto` qualifier.
- Other constraints which are not mentioned in this document.

Chapter 5
Compilation process and integration

This chapter describes the Compass C compilation process and integration with other tools.

It contains the following sections:

5.1 Compilation process

The process of writing a program for the NPU is the process of implementing the entry function using Compass C. After completing the coding, you need to compile the code and generate the final binary file that can be run on the NPU.

The Compass C compilation process is similar to the traditional compilation process. The compilation process includes three stages—compilation, assembly, and linking.

The following figure shows the compilation process in Compass C.

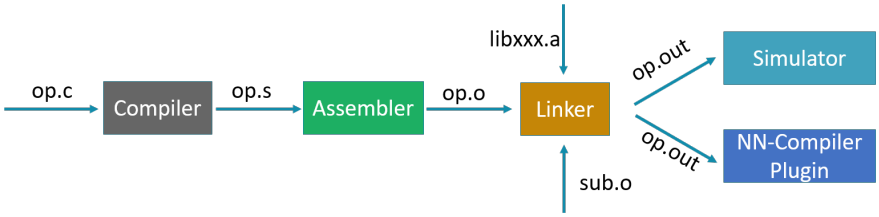

Figure 5-1 Compilation process

In the compilation stage, the `.c` source file is compiled through compiler `aipucc` to generate a `.s` assembly file. In the assembly stage, the `.s` file is turned into a `.o` object file by assembler `aipuas`. In the final stage, linker `aipuld` links one or more `.o` files and the dependent `.a` static library file and generates the final `.out` binary file.

As toolchains for Compass C, the compiler, assembler and linker are complex. There are many options in `aipucc`, `aipuas` and `aipuld`, but most options can be ignored.

Note the options in the following compilation commands:

- To generate the assembly code file for Zhouyi X1_1204 with the `-S` option, use optimization level `O0` (ASID is also supported):

  ```
  $aipucc -O0 -mcpu=X1_1204 -S test.c
  ```

- To generate the object file for Zhouyi Z2_1104 with the `-c` option, use optimization level `O2`:

  ```
  $aipucc -O2 -mcpu=Z2_1104 -DASID -c test.c
  ```

- To generate the executable object file for the build tool or simulator, link with `libcommon.a`:

  ```
  $aipucc -O2 -mcpu=X1_1204 test.c -Lpath_to_libcommon.a -lcommon -o test.out
  ```

- To generate the object file with assembler `aipuas` for the assembly file, run the following:

  ```
  $aipuas -mcpu=X1_1204 test.c -o test.o
  ```

- To generate the executable object file with linker `aipuld` for a single file, link with `libcommon.a`:

  ```
  $aipuld test.o -Lpath_to_libcommon.a -lcommon -o test.out
  ```

- To generate the executable object file with linker `aipuld` for multiple files, run the following:

  ```
  $aipuld test1.o test2.o -o test.out
  ```

5.2 Integration

The compiled `.out` binary files can run on the simulator directly or be used as the gbuilder plugin. If the files are used as the gbuilder plugin, you need to use the build tool to extract the instructions and data from multiple `.out` files, and integrate them to generate a final binary file. This final binary file can work with runtime to run on the NPU. For more information, see the *Arm China Zhouyi Compass Software Technical Overview*.

Chapter 6
Debug

This chapter describes the debug methods of Compass C.

It contains the following sections:

6.1 Printf

To facilitate debugging, Arm China provides a debug built-in function—`printf` to print formatted data to the host terminal after the program ends. The following is the prototype:

```
int printf(const char *format, ...);
```

This function writes the C string pointed by `format` to the NPU buffer, and the host prints the data in the buffer to the screen after the program ends. The buffer size is 1MB, so if the size of data is large than 1MB, the data in buffer will be overwritten from the beginning of the buffer.

If `format` includes format specifiers (subsequences beginning with %), the additional arguments following `format` are formatted and inserted in the resulting string replacing their respective specifiers. Currently, only scalar arguments are supported.

A format specifier follows this prototype:

```
%[flags][width]specifier
```

The following table shows the supported specifiers.

Table 6-1 Specifiers

Specifier	Output	Example
u	Unsigned decimal integer.	7235
d	Signed decimal integer.	392
x	Unsigned hexadecimal integer.	7fa
c	Character.	a
s	String of characters.	sample
p	Pointer address.	0x13c3e
%	A % followed by another % character will write a single % to the stream.	%

The following table shows the optional flags and width that follow these specifications.

Table 6-2 Flags

Flag	Description
#	Used with x specifiers. The value is preceded with `0x` for values different than zero.
0	Left-pads the number with zeroes (0) instead of spaces when padding is specified (see the width sub-specifier).

The following table shows the width description.

Table 6-3 Width

Width	Description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.

The `printf` function does not support printing SRAM variable directly. However, you can transfer the SRAM variable to DDR, and then print it.

The following is a simple example for a 4-TEC chip:

```
#define TECNUM 4

// data in LSRAM0
```

```
__lsram0 v32int8_t lsram0_var[4];
// do something
// buffer for data saving. 512 = sizeof(v32int8_t) * 4 * TECNUM = 32 * 4 * 4;
uint8_t buffer[512];
__memcpy_lsram02ddr(buffer, lsram0_var, 512);
for (uint32_t i = 0; i < 512; i++) {
    printf("%#x", buffer[i]);
}
printf("\n");
```

_____ Note_____

You need to call the corresponding runtime API to display the result of `printf` in the host terminal. You also need to add `EN_PRINTF_BUF=1` in `runtime.cfg` if you use the simulator. For more information, see the *Arm China Zhouyi Compass Software Technical Overview*.

6.2 Debugger

Arm China also provides a debugger named `aipudbg` for debugging NPU applications on the NPU simulator or actual hardware. `aipudbg` allows you to set breakpoints and single-stepping, and to inspect and modify memories and variables.

If you want to use `aipudbg` for debugging, you need to pass the `-g` option to `aipucc` when you compile your applications. In this case, `-O0` is highly recommended.

For more information, see the *Arm China Zhouyi Compass Debugger User Guide*.

6.3 Assertion

Arm China provides a debug macro—`assert` to add diagnostics in your programs. The following is the macro:

```
void assert(int expression);
```

The definition of the assert macro depends on another macro, NDEBUG, which is not defined by the compiler.

If NDEBUG is defined as a macro name at the point in source code or passed as a compilation option, then the assert macro does nothing.

If NDEBUG is not defined, then the assert macro checks whether the expression (which must have scalar type) compares equal to zero. If it does, the assert micro outputs diagnostic information to the printf buffer and the exit program. The diagnostic information is required to include the text of expression and the values of `__FILE__` and `__LINE__`.