Portable ISA Specification

version 0.1

August 13, 2026

Contents

Start

Introduction

This document specifies portable instruction set architecture (PISA) for Intel GPUs, providing a programming model and instruction set for general-purpose parallel programming.

Goals

The goals of PISA include:

- Provide a stable, portable ISA for compilers targeting Intel GPUs
- Provide “close-to-the-metal” performance comparable to native machine ISA code
- Facilitate advanced programming support such as hand-written libraries and inline assembly

Document Overview

This document is organized as follows:

- Architecture Model
  - Virtual Machine — OpenCL 3.0–based execution and memory model.
  - Address Spaces and Types — address spaces and data types.
  - Floating Point — floating-point data types and semantics.
  - Special Registers — access to execution context.
- Programming Interface
  - Assembly Syntax — format and program structure of PISA assembly.
  - Variables — variable declarations, linkage, and initializers.
  - Operands and Addressing — operand types and usage patterns.
  - Calling Convention — declaring and invoking kernels and functions.
  - Memory Access — memory access restrictions and semantics.
  - Compatibility — versioning and backward-compatibility guarantees.
- Instruction Reference
  - Instruction Set — description of core instructions.
- Appendix
  - Index of PISA directives
  - Legal Notices and Disclaimers

Architecture Model

Virtual Machine

The PISA virtual machine is an execution environment that dictates how PISA kernels are structured, launched, and run, largely following the OpenCL 3.0 execution model.

Execution Model

A PISA kernel is a sequence of instructions that executes on the PISA virtual machine. When the host program submits a kernel for execution, the PISA virtual machine creates an N-Dimensional Range (NDRange) of `work-items`, and an instance of the kernel is created for each element in the NDRange. In addition to the total number of work-items (**global size**), the host program may also specify how many work-items to group together into `work-groups` (**local size**). Work-items in the same work-group can communicate through objects in the shared memory, and they can synchronize through the barrier instructions. During execution, each work-group is further divided by the virtual machine into one or more `sub-groups`, and work-items within a sub-group can communicate and synchronize through shuffle, reduce, and barrier instructions. A value or variable is considered **sub-group-uniform** if it is guaranteed to have the same value across all active work-items within a sub-group.

Note

Special registers can be used to query the sub-group, local, and global sizes, as well as work-item IDs.

Order of Execution

The PISA virtual machine executes work-groups concurrently but not necessarily in parallel, and their execution order is unspecified. Work-items in the same work-group execute as members of the same work-group instance and may synchronize and communicate through work-group-level mechanisms, but the virtual machine does not require any particular issue order among them. Sub-groups within the same work-group maintain independent forward progress with respect to each other. In other words, unless one sub-group depends on another sub-group, for example by waiting at a work-group barrier, its execution must not be blocked by that other sub-group.

Memory Model

PISA memory objects may be stored in one of the supported address spaces.

- global memory (visible to all work-items)
- constant memory (visible to all work-items)
- shared local memory (visible to work-items in the same work-group)
- private memory (visible to this work-item only)
- parameter memory (kernel arguments, visible to all work-items)

Note

All address spaces, except for register and parameter, follow the OpenCL definitions.

Memory Consistency

The PISA memory consistency model follows the C++ memory model and extends it with explicit memory scopes. Memory accesses may be non-atomic (`weak`) or atomic with a memory order. `weak` accesses provide no ordering or atomicity guarantees. `relaxed` atomic accesses impose no ordering constraints; stronger orders establish happens-before relationships across work-items.

Memory Order

Atomic memory instructions carry a memory order that establishes happens-before relationships across work-items. Memory order can also be combined with a scope on a fence instruction to order relaxed memory operations.

Note

`weak` is not a memory order. It marks a non-atomic access with no atomicity or ordering guarantees. Use `relaxed` or a stronger order for atomic accesses.

Memory Order Specifiers

PISA Memory Order	C++ equivalent
relaxed	memory_order_relaxed
acquire	memory_order_acquire
release	memory_order_release
acq_rel	memory_order_acq_rel
seq_cst	memory_order_seq_cst

Important

Memory order may only be specified on access to global or shared memory objects.

Memory Scope

Memory scope specifies the set of work-items in which a memory instruction can directly synchronize. A memory scope must be specified when the memory order is stronger than `relaxed` (i.e., `acquire`, `release`, `acq_rel`, or `seq_cst`). It can also be combined with a fence instruction to establish ordering of relaxed memory accesses across the work-items in the specified scope.

Memory Scope Specifiers

PISA Memory Scope	Description
`subgroup`	All work-items in the same sub-group
`workgroup`	All work-items in the same work-group
`gpu`	All work-items in the global NDRange
`system`	All work-items in the global NDRange and the host program

Important

Memory scope may only be specified on access to global or shared memory objects.

Address Spaces and Data Types

Address Spaces

PISA variables reside in one of the address spaces below. Each address space has distinct access speed, visibility, and lifetime characteristics.

Address Spaces

Name	Directive	Description
Register	`.reg`, `.pred`	Private for a work-item
Private	`.private`	Private memory for a work-item
Shared	`.shared`	Shared memory for work-items in a work-group
Global	`.global`	Global memory, accessible by all work-items
Constant	`.const`	Constant memory, initialized and read-only memory
Parameter	`.param`	Exclusively used for kernel argument passing

PISA specifies the size of each memory address space for addressing purposes.

Memory Address Spaces

Address Space	Address Size	Null Pointer Value
Private (`.private`)	32-bit	0xFFFFFFFF (-1)
Shared (`.shared`)	32-bit	0xFFFFFFFF (-1)
Global (`.global`)	64-bit	0
Constant (`.const`)	64-bit	0

Note

An address of a variable in a memory address space can be obtained using the addrof instruction.

Register Space

Register space provides private fast storage location for a work-item. Variables in this space are specified using `.reg` or `.pred` directives, and can be used directly in most instructions. Memory instructions allow transfer of data between register and other spaces.

Private Space

Private space supports memory objects that are private to each work-item. Variables in this space are specified using `.private` directive, and can be accessed using private memory load and store instructions.

Shared Space

Shared space supports memory objects that are shared among work-items within a work-group. Variables in this space are specified using `.shared` directive, and can be accessed using shared memory load, store, and atomic instructions. Shared variables have the same lifetime as a work-group; variables are uninitialized at creation, and their contents disappear when the work-group finishes execution.

All work-items in the same work-group may read and write to any variable in the shared space; they can also synchronize their accesses to the shared space through synchronization instructions, such as fence, which guarantees memory ordering by ensuring that all shared accesses prior to the instruction are observable by all other work-items within the same work-group.

Shared memory can also be dynamically allocated by the host. In this case, the host program provides a pointer to the allocated shared object as a kernel argument:

Global Space

Global space supports memory objects that are shared among work-items in different work-groups. Variables in this space are specified using `.global` directive, and can be accessed using global load, store, and atomic instructions.

Constant Space

Constant space supports **read-only** memory objects that can be accessed by every work-item. Variables in this space are specified using `.const` directive, and can be accessed using constant load instructions. Constant variables are guaranteed to be initialized before they are first accessed.

> **Warning**
>
> Storing data to a `.const` address space results in undefined behavior.

Parameter Space

Parameter space is used exclusively for kernel arguments. Variables in this space are specified using `.param` directive, and can be accessed using `ld.param` instruction.

Data Types

PISA virtual machine supports scalar, vector, and packed data types.

Scalar Types

Scalar Integer Types

Qualifier	Bitwidth
`.8b`	8
`.16b`	16
`.32b`	32
`.64b`	64
`.128b`	128

Scalar Floating Point Types

Qualifier	Bitwidth
`.hf`	16
`.bf`	16
`.f`	32
`.df`	64

Conversion instructions, such as i2f and f2i, also support signed and unsigned versions of scalar integer types.

Scalar Integer Types (Conversion)

Unsigned Qualifier	Signed Qualifier	Bitwidth
`.u8`	`.s8`	8
`.u16`	`.s16`	16

.u32	.s32	32
.u64	.s64	64

Packed Types

A packed type is a vector of multiple scalar values of the same bitwidth. An instruction using a packed type operand will perform the operation independently on each element of the vector.

Packed Data Types

Qualifier	Bitwidth	Element	Count
.8bx4	32	.8b	4
.16bx2	32	.16b	2
.hfx2	32	.hf	2
.bfx2	32	.bf	2

Important

Packed type operands must be register operands; immediate values are not allowed.

Note

Packed type operands (`.bfx2`, `.hfx2`, `.16bx2`, `.8bx4`) must be declared as vector registers of matching element count and size: `.v2.16b` for 2-element types and `.v4.8b` for 4-element types. The packed type qualifier on the instruction determines how the bits are interpreted; the register declaration determines storage layout.

Note

`.hfx2` and `.bfx2` are **floating-point** packed types: their elements carry floating-point semantics including rounding modes, NaN propagation, and saturation. They are distinct from the integer packed types `.16bx2` and `.8bx4` and must not be used interchangeably.

Vector Types

Vector types are supported in limited form by memory access and data movement instructions. As these instructions do not interpret underlying data, vectors can represent both integer and floating-point values.

A vector type is specified as `.vN.B`, where `N` is the number of elements and `B` is the element bitwidth (e.g., `.v4.32b` for a 4-element vector of 32-bit integer or single-precision floating-point values).

Vector Data Types

Bitwidth	Element Count
.8b	2, 3, 4
.16b	2, 3, 4
.32b	2, 3, 4, 5, 6, 7, 8, 16, 32, 64
.64b	2, 3, 4

Floating Point Support

Data Types

PISA virtual machine supports floating point data types of multiple precisions.

Floating Point Data Types

Name	Bit size	Sign bit	Exp bits	Mantissa bits	IEEE-754 compliance
Single precision (.f)	32	[31:31]	[30:23]	[22:0]	Y - IEEE-754-1985
Double precision (.df)	64	[63:63]	[62:52]	[51:0]	Y - IEEE-754-1985
Half float (.hf)	16	[15:15]	[14:10]	[9:0]	Y - IEEE-754-2008
Bfloat16 (.bf)	16	[15:15]	[14:7]	[6:0]	N

Rounding Mode

Rounding mode determines how the result of a floating-point operation is rounded when it cannot be represented exactly in the destination type. It is specified on a per-instruction basis using the rounding mode qualifier.

Floating Point Rounding Modes

Qualifier	Description
`.re`	Round to nearest even.
`.rd`	Round down, toward minus infinity.
`.ru`	Round up, toward plus infinity.
`.rz`	Round toward zero.
`.rna`	Round to nearest, ties away from zero.

Note

Default rounding mode for all floating-point operations is `.re`.

Flush-To-Zero (FTZ)

By default PISA floating-point arithmetic instructions retain denormal values. A floating-point instruction may opt to flush its denormal sources and result (post rounding) to sign-preserving zero by specifying the `.ftz` qualifier.

Not A Number (NaN)

An IEEE-754 NaN has an exponent field with all bits set and a nonzero fraction field. A signaling NaN (sNaN) has zero as the Most-Significant Bit (MSB) of its fraction field, while a quiet NaN (qNaN) has one as the MSB of its fraction field.

When a qNaN is an input to a floating point instruction, the same qNaN is returned as output. If there are multiple NaN input values, one of them will be selected as the output NaN value.

Destination Saturation (DSat)

Floating-point instructions may support an optional destination saturation via `.dsat` qualifier. When `.dsat` is specified, the result of the instruction is clamped to the specific range or value. Exact behavior of this qualifier is specified in the instruction description.

Special Registers

Special registers are predefined variables within PISA virtual machine. They can be accessed anywhere within a PISA program. They may either be programmed by the driver prior to kernel launch (e.g. `%globaloffset`) or may be updated by hardware to reflect current execution state (e.g. `%activemask`).

Special registers with 3-element vector data types (e.g. `.v3.32b`) provide a way to access per-dimension values. Individual elements can be accessed using swizzle syntax.

- `.x` - specifies the value along x dimension
- `.y` - specifies the value along y dimension
- `.z` - specifies the value along z dimension

Important

Special registers with a vector type are accessed one element at a time via the `.x` / `.y` / `.z` swizzle only. The `.w` and multi-element swizzles (`.xy`, `.zw`, `.xyzw`) are not supported, and they cannot be used as whole-vector operands (e.g. with extract / insert).

Note

Unless specified otherwise, special registers are **read-only** and carry **unsigned** data.

Special Registers

Name	Type	Description
`%activemask`	`.32b`	Indicates whether the corresponding work-item within the sub-group is currently active (1 bit per work-item; 1 = active, 0 = inactive). Implicitly updated by control-flow instructions.
`%enqueuedlocalsize`	`.v3.32b`	Specifies the number of work-items per work-group requested at kernel dispatch.
`%globaloffset`	`.v3.64b`	Specifies the global offset of the work-item.
`%globalsize`	`.v3.64b`	Specifies the number of global work-items.
`%groupcount`	`.v3.32b`	Specifies the work-group count.
`%groupid`	`.v3.32b`	Specifies the work-group ID.
`%laneid`	`.32b`	Specifies the work-item ID within a sub-group.
`%localid`	`.v3.32b`	Specifies the work-item ID within a work-group.

`%localsize`	`.v3.32b`	Specifies the work-group size.
`%printfbuffer`	`.64b`	Holds the address of the printf buffer.
`%subgroupsize`	`.32b`	Specifies the size of a sub-group. Current PISA specification fixes this value at **32**.
`%workdim`	`.32b`	Specifies the number of dimensions in use for the kernel dispatch.

Programming Model

Syntax and Format

Program

A PISA **program** consists of one or more modules, with each module being defined in a separate source file. One or more modules can be combined into a single module as either an executable (with entry kernels) or a library.

A **module** contains a header, followed by one or more kernel, function, and global variable declarations and definitions. Kernels, functions, and global variables may also have attributes specified alongside them. A module may also include debugging, profiling, and other identifying information.

A global variable is a variable that is declared at the module level and is accessible by all kernels and functions within the module. Global variables are identified via `.global` or `.const` directives.

A kernel is a special type of function that serves as an entry point for execution on the device, and can only be invoked from the host program. It is identified via `.kernel` directive and has no return value.

A function is a subroutine that can be invoked by kernels or other functions. It is identified via `.function` directive and may have a return value.

Example PISA Module

```
// global variable declaration
.global .align 4 @A[4] = { .32b 42 };

// function returning 32-bit value
.function .32b @funcName() {
  .reg .32b %r0;

  ld.global.32b %r0, [@A];
  return %r0;
}

// kernel entry point
.kernel @kernelName() {
  .reg .32b %rv;

  call %rv, @funcName();
}
```

Statement

A PISA kernel or function is a collection of statements. A **statement** is a variable declaration, instruction, label, scope, or comment.

A **declaration** introduces a variable and allocates storage for it. A declaration must terminate with a `;`.

A **label** is an identifier that marks a position in the code and is used as a target for goto instruction. Labels must appear alone on a line and terminate with a `:`.

An **instruction** specifies an operation to be performed, such as arithmetic, logic, data movement, etc. Instruction must terminate with a `;`.

A **scope** is a block of statements enclosed in curly braces `{}`. Variables declared within a scope are visible throughout that scope and any nested subscopes.

A **comment** provides explanatory notes for human readers and is ignored by the assembler. Comments begin with a `//` and continue to the end of the line.

Example PISA statements

```
// variable declaration
.reg .32b %r, %a, %b;

label1:
  // instruction
  iadd.32b   %r, %a, %b;
```

Identifiers

Named identifiers are used for functions, kernels, variables, registers, and labels. A named identifier consists of an optional prefix character and the name string.

The name string must match regular expression `[a-zA-Z$._][a-zA-Z$._0-9]*`.

- '@' prefix is used for functions, kernels, and memory variable identifiers.
- '%' prefix is used for register identifiers.
- label identifiers do not have a prefix character.

Variables

PISA variable declaration specifies its address space, bitwidth, and name, along with optional attributes.

Bitwidth

Variables have a defined bit-size (bitwidth) but are otherwise untyped. Interpretation of data contained in a variable is up to the instruction using the variable. An instruction operating on 32-bit variables may treat its source operands as either signed 32-bit integers, or single-precision floating point values, depending on the instruction opcode. Variable declarations specify the bitwidth of the variable using scalar or vector data types.

Register Variables

Named register declarations are used to define one or more register variables with the same bitwidth. Name of the register must start with a `%` and is followed by an identifier.

```
// define a 32-bit register variable named %r0
.reg .32b %r0;
// define a 4x32-bit vector register variable named %v1
.reg .v4.32b %v1;
// define multiple 64-bit register variables named %d1, %d2, and %d3
.reg .64b %d1, %d2, %d3;
```

Multiple register variables can also be declared using a range-based syntax, where a half-open index range `<Start~End>` is appended to a base name to generate each variable name. The range includes the lower bound and excludes the upper bound.

```
// define multiple 16-bit register variables named %h0, %h1, %h2, %h3, %h4
.reg .16b %h<0~5>;
// define vector register variables %v4w4, %v4w5, %v4w6, %v4w7
.reg .v4.32b %v4w<4~8>;
```

> **Note**
>
> Register variables can be declared anywhere inside a function or a kernel.

Predicate Variables

Predicate variables reside in register space and hold single-bit values produced by comparison instructions. They can control branches and select instructions, and serve as carry-in/carry-out bits for integer arithmetic overflow detection.

A predicate declaration follows the same syntax as a register variable declaration with the following modifications:

- `.pred` is used instead of `.reg`.
- bitwidth is not specified, as predicates are single-bit values.

```
// define a predicate variable named %p0
.pred %p0;
// define multiple predicate variables named %p1, %p2, %p3
.pred %p<1~4>;
```

> **Note**
>
> Predicate variables can be declared anywhere inside a function or a kernel.

Memory Variables

Memory variable declarations define named allocations in one of a memory address space. The declaration follows the following format:

- address space - `.global`, `.const`, `.private`, or `.shared`
- name of the memory variable, starting with @ followed by an identifier
- allocated size in bytes, specified within `[]`

A memory variable may be used as a source in addrof instruction or in the address expressions of load, store, and atomic instructions.

The allowed scope depends on the address space: `.private` and `.shared` variables must be declared inside a kernel or function, while `.global` variables must be declared at module scope. `.const` variables can be declared at either module scope or inside a kernel or function.

```
// define a 8 byte global variable named @G
.global @G[8];

.kernel @test() {
   // define a 16 byte shared variable named @S
   .shared @S[16];
}
.function void @func() {
   // define a 1 byte private variable named @P
   .private @P[1];
}
```

Alignment

Memory variables are aligned to byte boundary by default. If greater alignment is needed, it can be specified as part of variable declaration using `.align` attribute.

```
// define a private variable @A of 16 bytes, aligned to 4-byte boundary
.private .align 4 @A[16];
```

> **Warning**
>
> Alignment value must be a power of two. Any other alignment results in a compile-time error.

Global Variables

Memory variables declared in module scope (`.global` or `.const` space) are considered global variables. Global variables are visible in the entire module. A global variable that has no static initializer has an undefined value at kernel entry.

Initializers

Global variables support an initializer syntax to specify the variable's initial value. Initializer value is specified via comma-separated pairs of size and immediate integer values, enclosed within `{ }`. A floating-point immediate must be converted to hexadecimal format to be used in an initializer.

An immediate value can be specified as a decimal or a hexadecimal (prefixed with `0x` or `0X`). Hexadecimal literals are zero-extended, while decimal literals are sign-extended to 64 bits. Resulting value is then truncated to the size of the initializer element.

- size of an immediate must be one of `.8b`, `.16b`, `.32b`, or `.64b`
- nested initializer values are not permitted
- immediate values are stored using little-endian encoding.

```
// 4-byte global variable initialized to 100
.global .align 4 @G1[4] = { .32b 100 };
// 8-byte constant variable initialized to -1
.const .align 4 @G2[8] = { .64b 0xFFFFFFFFFFFFFFFF };
// 4-byte global variable initialized to single-precision -1.0
.global .align 4 @G3[4] = { .32b 0xBF800000 };
```

> **Important**
>
> Number of bytes in initializer must match number of bytes in variable declaration

> **Note**
>
> Initializer is mandatory for `.const` variables

Zero-initializer

The `.zero N` directive can be used to zero-initialize a global variable, where `N` represents the number of zero bytes.

```
// 4-byte global variable initialized to zero
.global @C[4] = { .zero 4 };
// 8-byte global variable initialized to alternating 0,-1 values
.global .align 4 @D[8] = { .16b 0xFFFF, .zero 2, .16b 0xFFFF, .zero 2 };
```

is equivalent to

```
.global @C[4] = { .32b 0 };
.global .align 4 @D[8] = { .16b 0xFFFF, .16b 0, .16b 0xFFFF, .16b 0 };
```

Symbol-initializer

Any symbol in module scope, such as functions and global variables, may be used as an initializer value. In such cases, the value represents the 64-bit address of the symbol in its address space (equivalent to using addrof instruction). A constant offset from the symbol address can also be specified using `@<symbol>+<offset>` syntax.

```
.global .align 4 @D[8] = { .16b 0xFFFF, .16b 0, .16b 0xFFFF, .16b 0 };
.function .32b @add(.32b %a, .32b %b) {
   ...
}

// @FAddr is initialized to the address of 'add' function
.const .align 8 @FAddr = { .64b @add };
// @VAddr is initialized to the address of global variable @D
.const .align 8 @VAddr = { .64b @D };
// @GAddr is initialized to the address of the 3rd element of @D
.const .align 8 @GAddr = { .64b @D+4 };
```

Linkage

By default, global variables have internal linkage, meaning that they are only visible within the module in which they are defined.

An `.export` directive is used to indicate external visibility of a variable across different modules. Exported variables can be used to resolve references to symbols with the same name during linking.

```
// allow access to variable @A from another module
.export .global .align 4 @A[4] = { .zero 4 };
```

An `.import` directive is used to indicate that a variable is not defined within the current module. Imported variables do not support initializers.

```
// import global variable @A from another module
.import .global .align 4 @A[4];
```

Important

A variable must not be declared more than once within a module, regardless of its linkage.

Attributes

Several optional attributes may be specified for global variables.

`.section("NAME")` specifies a name of the section of the object file to place the variable in.

```
    // place variable @HI in a section named ".const.data"
    .const .section(".const.data") @HI[3] = { .8b 0x48, .8b 0x49, .8b 0 };
```

`.host_access("NAME")` specifies the name used to refer to the variable in the Host API.

```
    // export global variable @G as "my_global_var" for host access
    .global .align 4 .host_access("my_global_var") @G[4] = { .32b 0 };
```

Operands

PISA instruction has zero or more operands, whose size and type are defined by the instruction template. No implicit type conversion is performed on the operands, and each operand may be a register variable, an immediate value, or an address.

Register Operands

A register operand is specified using the name of a register or a predicate variable.

```
.reg .32b %dst, %src0, %src1;
.pred %pin, %pout;

iadd.32b %dst, %src0, %src1;
uaddc.ci.co.32b %dst, %pout, %src0, %src1, %pin;
```

Immediate Values

An immediate value can be used as a source operand anywhere a scalar register variable is allowed, unless specified otherwise by the instruction template.

An immediate value used in integer instructions can be specified as a decimal or a hexadecimal literal (prefixed with `0x` or `0X`). Hexadecimal literals are zero-extended, while decimal literals are sign-extended to 64 bits. If the resulting 64-bit value does not mathematically fit within the signed or unsigned range of the operand size, a parser error will be generated.

```
  .reg .32b %dst, %src0;

  // add -1 to %src0 and store the result in %dst
  iadd.32b %dst, %src0, -1;
// add 15 to %src0 and store the result in %dst
  iadd.32b %dst, %src0, 0xF;
```

An immediate value used in floating-point instructions may either be a decimal floating-point literal, or in IEEE hexadecimal format. Decimal floating-point literal is treated as a double-precision value, which is converted to type of an operand using implementation-defined rounding and denorm mode.

Important

Size of hexadecimal floating-point value must match size of operand, as no type conversion is allowed.

```
  .reg .32b %dst, %src0;

  // add single-precision 4.2 to %src0 and store the result in %dst
  fadd.f %dst, %src0, 4.2;
// add single-precision -1.0 to %src0 and store the result in %dst
  fadd.f %dst, %src0, 0xBF800000;
```

Note

There is no provision for a vector of immediate values.

Address Operands

Address operands are used in memory-related instructions and are written in the form `[Value1]` or `[Value1 + Value2]`. Instruction descriptions reference the following shorthand names for common operand forms:

Address operand forms

Name	Meaning	Syntax
`MemRR`	32-bit or 64-bit base address with a register offset	`[reg + reg]`, `[var + reg]`
`MemRI`	32-bit or 64-bit base address with a signed immediate offset	`[reg + imm]`, `[var + imm]`
`MemR`	32-bit or 64-bit base address without an explicit offset	`[reg]`, `[var]`
`MemI`	Immediate address value without an explicit offset	`[imm]`

Here reg denotes a scalar register, var denotes a memory variable, and `imm` denotes an immediate value accepted by the instruction template.

Warning

Not all instructions support all operand forms.

A memory variable represents the byte address of a variable in a memory address space. The address size is interpreted as follows:

- 32-bit value for `.private` and `.shared` variables.
- 64-bit value for `.global` and `.const` variables.
- 32 or 64-bit value for a register variable, depending on its declaration.

Either 32-bit or 64-bit arithmetic is performed based on the address operand values, and the result serves as the memory access address.

```
.reg .32b %reg32;
.const .align 4 @constVar[4] = { .32b 42 };
.shared .align 4 @sharedVar[16];

// load value from global address space specified by %reg64
ld.global.32b %reg32, [%reg64];
// load value from 32bit shared address space specified by @sharedVar[4]
ld.shared.32b %reg32, [@sharedVar + 4];
// load value from 64bit const address space specified by @constVar[0]
ld.const.32b %reg32, [@constVar];
```

Warning

An out-of-bounds memory access results in undefined behavior.

Generic Addressing

Address operands also support a generic addressing mode where the address space is not specified at compile time. In this mode, the address operand is treated as a 64-bit **generic** address, with translation to the concrete address space being accomplished via addrcast instruction.

```
.private @A[20];
.reg .64b %gen64;
.reg .32b %prv32, %reg32;

// convert 32-bit private address into a 64-bit generic address
```

```
 addrcast.generic.private %gen64, @A;
// load 4 bytes from offset 12 of @A
 ld.generic.32b %reg32, [%gen64 + 12];
 // update %gen64 to point to @A[4]
 iadd.64b %gen64, %gen64, 4;
 // convert 64-bit generic address back to 32-bit private address
 addrcast.private.generic %prv32, %gen64;
// load 4 bytes from @A[4] using the converted private address
 ld.private.32b %reg32, [%prv32];
```

Vector Operands

Register operands of vector type are supported by a limited number of memory access and data movement instructions. For all other instructions, each element of the vector must be accessed individually.

Elements of a vector operand can be accessed by specialized instructions (e.g. extract and insert) or via swizzle syntax that allows individual access to the first 4 elements of a vector.

Swizzle Values

Swizzle	Element
.x	0
.y	1
.z	2
.w	3

```
.reg .v2.32b %vec;
.reg .32b %dst, %src;

// load 2 32-bit values
ld.private.v2.32b %vec, [%addr];
// access %vec[0]
mov.32b %dst, %vec.x;
// add 2 elements of %vec
iadd.32b %dst, %dst, %vec.y;
```

mov instruction supports additional multi-element swizzles (`.xy`, `.zw`, and `.xyzw`) to enable data movement between vector and scalar variables.

Calling Convention

PISA provides high-level C-like abstractions for calling and returning from a function.

Kernel

A kernel is an entry point for execution and is defined using the `.kernel` directive. Kernel cannot have a return value, but may have zero or more arguments. All kernel arguments are read-only and have per-kernel scope, i.e. all work-items receive the same value of each kernel argument.

```
.kernel @foo() {
   // kernel body
}
```

Note

Kernel can only be invoked from the host program.

Attributes

Several optional attributes may be specified for the kernels, using `.attribute_name(value)` or `.attribute_name` syntax. They are declared between `.kernel` and its name, and are separated by spaces.

```
.kernel .reqd_work_group_size(256, 2, 1) .vec_type_hint(int2) @test() {
   // kernel body
}
```

`.reqd_work_group_size(X, Y, Z)`

Specifies required work-group size. The value `X`, `Y`, and `Z` specify the size of each dimension.

`.vec_type_hint(TYPE)`

Specifies a hint of the computation width of the decorated kernel. The value `TYPE` may be `int`, `int2`, or any other scalar or built-in vector type defined by the OpenCL specification.

Kernel Parameters

Kernel parameters are decorated using mandatory `.param` directive. Each argument is modeled as a byte array stored in an abstract location pointed to by corresponding parameter. Size of the argument in bytes must be specified in corresponding parameter declaration.

Argument data is loaded into registers using special ld.param instruction.

```
.kernel @foo(.param[4] %arg0, .param[8] %arg1) {
   .reg .32b %larg0, %larg1;

   // load contents of %arg0
   ld.param.32b %larg0, [%arg0];
   // load contents of %arg1[4]
   ld.param.32b %larg1, [%arg1 + 4];
}
```

Important

Kernel parameters can only be referenced by `ld.param` instruction.

It is the responsibility of the PISA producer to define how to access struct members via the `ld.param` instruction. Consider the following **naturally aligned** struct:

structure declaration in host program

```
// | x | - | y | y | z | - | - | - | w | w | w | w |
struct {
   unsigned char x;
   unsigned short y;
   unsigned char z;
   unsigned int w;
}
```

structure access within kernel

```
.kernel @foo(.param[12] %arg) {
   .reg .8b %local_x, %local_z;
   .reg .16b %local_y;
   .reg .32b %local_w;

   // x is byte-aligned at offset 0
   ld.param.8b %local_x, [%arg];
   // y is aligned to 2 bytes at offset 2
   ld.param.16b %local_y, [%arg + 2];
   // z is byte-aligned at offset 4
```

```
    ld.param.8b %local_z, [%arg + 4];
    // w is aligned to 4 bytes at offset 8
    ld.param.32b %local_w, [%arg + 8];
}
```

Parameter Attributes

Several optional attributes may be specified for kernel parameters, using `.attribute_name(value)` or `.attribute_name` syntax. They are declared between `.param` and the parameter name, and are separated by spaces.

```
.kernel @foo(.param[8] .align(4) .addrspace(global) %arg0) {
    // kernel body
}
```

`.addrspace(ADDRSPACE)`

Specifies address space memory object points to. `ADDRSPACE` can be `global`, `shared`, or `const`.

`.align(BYTE)`

Specifies the alignment of the kernel parameter in the kernel argument buffer. The default alignment is 8 bytes; `BYTE` must be a power of two, and specifies the alignment in bytes.

`.ptr_align(BYTE)`

Specifies the alignment of the memory pointed to by the kernel parameter. `BYTE` must be a power of two, and specifies alignment of the pointee in bytes.

Function

Functions are declared/defined using the `.function` directive. Functions may have zero or more arguments, and may return a value to the caller. The return type `void` indicates that the function does not return a value.

```
.function void @foo() {
    // function body
}
```

Note

Functions can only be invoked from kernels or other functions.

Linkage

By default, functions have internal linkage, meaning that they are only visible within the module in which they are defined.

An `.export` directive is used to indicate external visibility of a function across different modules. Exported functions can be invoked by other kernels and/or functions defined in another module.

```
// allow access to function @foo from another module
.export .function void @foo() {
    // function body
}
```

An `.import` directive is used to indicate that a function is defined outside the current module. Imported functions must not have a function body defined in the current module, and are only declared with their signature.

```
.import .function .32b @foo(.reg .32b, .reg .32b);
```

Function Arguments

Function arguments are passed from caller to callee as actual arguments in a call instruction. Each actual argument must be a register variable, and must match the type specified in the function declaration, as no implicit conversion is allowed.

```
.function void @foo(.reg .32b %arg0, .reg .v2.32b %arg1) {
   // function body
}
.function void @caller() {
   .reg .32b %a;
   .reg .v2.32b %b;

   // ... code ...
   call void, @foo(%a, %b);
   // ... code ...
}
```

Note

Predicate variables cannot be used as function arguments.

Function Return

Function may return a single value to a caller using the return instruction. The return value may be an immediate or a register variable matching the return type specified in the function declaration, as no implicit conversion is allowed. If no return value is needed, `void` must be specified as the return type.

```
// function returning no value
.function void @foo() {
   // function body
   return;
}
// function returning a 32-bit value
.function .32b @foo() {
   .reg .32b %rv;

   // ... code ...
   return %rv;
}
```

Note

Predicate variables cannot be used as a function return value.

Important

The `return` instruction must be the last instruction in the body of a non-void function.

Memory Access

Access Alignment

For all memory operations, memory access address must be aligned to the bitwidth of `.type`. In cases where the bitwidth of `.type` is smaller than 32b, the address must be aligned to the accessed size, defined as the product of the bitwidth and the vector element count.

Examples of Memory Access Alignment Requirements

Instruction	Address Alignment
ld.64b	64
ld.v2.64b	64
ld.8b	8
ld.v2.8b	16

Data Size

Maximum data size is determined by memory access type, operand's bitwidth and element count.

Supported data sizes for memory instructions

Number of elements	Bitwidth
single element	.8b, .16b, .32b, .64b
.v2	.8b, .16b, .32b, .64b
.v3	.32b
.v4	.8b, .16b, .32b, .64b
.v8	.32b

Instruction definitions may provide additional restrictions on the supported data sizes.

Cache Control

Memory instructions support optional cache control qualifiers that serve as performance hints to the cache subsystem. These qualifiers are advisory and may not be honored by the system; they have no effect on the memory consistency behavior of the program.

Cache control definitions for atomic instructions

Cache control	Description
.uc	Do not cache the data in any cache level.
.L2wb	Write data to L2 but not the next level cache.
.L3wb	Write data to L3 but not the next level cache.

Cache controls for **non-atomic** instructions are described in their respective instruction definitions.

Compatibility

To achieve the goal of stable and portable PISA, each PISA module starts with a header that defines the version of the PISA specification and target architecture that the module is intended for. This information is used by the PISA consumer to ensure that modules are compatible with each other and with the actual GPU target, and to enforce constraints on the usage of features and instructions.

Directives

Version

The `.version` directive specifies the version of the PISA specification used by the module and must be the first statement in a PISA module. The version is composed of `major` and `minor` values, with the following characteristics:

- changes in **major** version indicate incompatible changes to the specification
- changes in **minor** version indicate backward-compatible additions.

```
.version 0.1;
```

Warning

- It is a compile-time error if an unknown `.version` value is specified.
- It is a link-time error if multiple modules specify different `major` values.

Target

The `.target` directive specifies the target architecture for which the module is intended and must be the second statement in a PISA module. The target architecture value is used to determine the set of supported features and instructions required to execute the module.

```
.target 100;
```

Warning

- It is a compile-time error if an unknown `.target` value is specified.
- It is a link-time error if multiple modules specify incompatible `.target` values.
- It is an error to use features or instructions not supported by the specified `.target` value.

Layer Model

Each PISA target architecture defines a set of supported features and instructions. Newer architectures may add features or instructions and may extend existing ones. A PISA program remains compatible with a newer architecture as long as it uses features and instructions still supported on the newer target.

Each PISA instruction in the instruction set is annotated with the minimum PISA version and target architecture required to use it. Each PISA feature is similarly annotated in its definition. The PISA producer may use this information when generating the .target directive in the module header.

The following naming convention is used for PISA target:

- a number indicating generation (e.g. 100)
    - backward compatible with all previous generations
- an optional letter indicating product line (e.g. ‘c’ for compute)
    - backward compatible with product line on current and previous generations
- an optional letter ‘a’ indicating presence of target-specific features
    - not backward compatible with any other target

PISA Compatibility Matrix

PISA .target	Can execute PISA instructions and features from
100	100
100c	100c, 100
100ca	100ca, 100c, 100

Instruction Set

Each instruction description provides a complete overview of its functionality, syntax, and usage. Each description follows a consistent structure:

- **Syntax**: instruction opcode, formats, qualifiers, operands, constraints, and compatibility info.
- **Semantics**: semantic description, usually expressed in C-like operations or pseudocode.
- **Notes**: operational details, constraints, and design rationale when needed.
- **Examples**: examples showing the instruction in a PISA program.

PISA Instruction Format

```
mnemonic operands;
```

- `mnemonic` consists of opcode and zero or more opcode qualifiers
- `operands` consists of zero or more operands, separated by `,`
- qualifiers, operands, or operand groups enclosed within <> are optional
- operand enclosed within `[]` indicate an address operand
- allowed values for an optional qualifier may indicate its **default** value
- allowed values for an optional qualifier may be omitted if its name and value are identical (e.g. `.ftz`)
- the terminating semicolon (`;`) is omitted for brevity in instruction syntax

Integer Arithmetic

dp4a

Compute the dot product of four pairs of bytes from two 32-bit operands, and add a 32-bit accumulator.

Syntax:

```
dp4a.signedness<.dsat>.type dst, src0, src1, src2

  .signedness = { .ss, .us, .su, .uu }
  .type = { .32b }


  - dst is a 32-bit register
  - src0, src1, src2 can be 32-bit registers or immediates
```

Restrictions

- qualifier `.dsat` must not be specified when qualifier `.signedness` is `.uu`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
src2_signed = .signedness == .us || .signedness == .ss
src1_signed = .signedness == .su || .signedness == .ss

if (src1_signed) {
  src1_0 = sign_extend(src1[7:0], 32)
  src1_1 = sign_extend(src1[15:8], 32)
  src1_2 = sign_extend(src1[23:16], 32)
  src1_3 = sign_extend(src1[31:24], 32)
} else {
  src1_0 = zero_extend(src1[7:0], 32)
  src1_1 = zero_extend(src1[15:8], 32)
  src1_2 = zero_extend(src1[23:16], 32)
  src1_3 = zero_extend(src1[31:24], 32)
}
if (src2_signed) {
  src2_0 = sign_extend(src2[7:0], 32)
  src2_1 = sign_extend(src2[15:8], 32)
  src2_2 = sign_extend(src2[23:16], 32)
  src2_3 = sign_extend(src2[31:24], 32)
} else {
  src2_0 = zero_extend(src2[7:0], 32)
  src2_1 = zero_extend(src2[15:8], 32)
  src2_2 = zero_extend(src2[23:16], 32)
  src2_3 = zero_extend(src2[31:24], 32)
}
dst = src0 + src1_0*src2_0 + src1_1*src2_1 + src1_2*src2_2 + src1_3*src2_3
```

Notes:

Enabling saturation via `.dsat` qualifier clamps the final result to between the min/max values representable by signed 32-bit integer. The addition of the results of the multiplications cannot overflow, but the addition of `src0` can cause overflow and therefore obeys the saturation control.

Examples:

```
.reg .32b %dst, %src0, %src1, %src2;

// src0, src1, and src2 are treated as unsigned
dp4a.uu.32b  %dst, %src0, %src1, %src2;
// src1 is treated as signed, src2 as unsigned
dp4a.su.32b  %dst, %src0, %src1, %src2;
// src0, src1, and src2 are treated as signed with saturation
dp4a.ss.dsat.32b  %dst, %src0, %src1, %src2;
```

iabs

Compute the absolute value of an integer.

Syntax:

```
iabs.type dst, src0

  .type = { .16b, .32b, .64b }


  - dst is a register
  - src0 can be a register or an immediate

  * src0 is interpreted as signed integer.
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                      default
dst = |src0|
```

Examples:

```
.reg .32b %dst, %src0;

// absolute value of a 32-bit element
iabs.32b  %dst, %src0;
```

iadd

Add two integer values.

Syntax:

```
iadd<.dsat>.type dst, src0, src1

  .type = { .16b, .32b, .64b }
```

```
  - dst is a register
  - src0, src1 can be registers or immediates

  * src0, src1 are interpreted as signed integers.
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
iadd<.dsat>.type dst, src0, src1

  .type = { .8bx4, .16bx2 }


  - dst, src0, src1 are packed registers

  * src0, src1 are interpreted as signed integers.
```

Restrictions

- element count and bitwidth of `dst`, `src0` and `src1` must match definitions of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                  .type == .16bx2
dst[15:0] = src0[15:0] + src1[15:0]
dst[31:16] = src0[31:16] + src1[31:16]
```

```
                                    .type == .8bx4
dst[7:0] = src0[7:0] + src1[7:0]
dst[15:8] = src0[15:8] + src1[15:8]
dst[23:16] = src0[23:16] + src1[23:16]
dst[31:24] = src0[31:24] + src1[31:24]

                                       default
dst = src0 + src1
```

Notes:

Enabling the `.dsat` qualifier clamps the final result to the minimum and maximum values representable by a signed integer of the element bitwidth of the specified `.type`.

Examples:

```
.reg .32b %dst, %src0, %src1;

// addition of 32-bit elements
iadd.32b  %dst, %src0, %src1;
// element-wise addition of 4x8-bit packed values
.reg .v4.8b %dst_4x8, %src0_4x8, %src1_4x8;
iadd.8bx4  %dst_4x8, %src0_4x8, %src1_4x8;
// element-wise addition of 2x16-bit packed values
.reg .v2.16b %dst_packed, %src0_packed, %src1_packed;
iadd.16bx2  %dst_packed, %src0_packed, %src1_packed;
```

ineg

Negate an integer value.

Syntax:

```
ineg.type dst, src0

  .type = { .16b, .32b, .64b }


  - dst is a register
  - src0 can be a register or an immediate

  * src0 is interpreted as signed integer.
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
dst = -src0
```

Examples:

```
.reg .32b %dst, %src0;

// negate 32-bit element
ineg.32b  %dst, %src0;
```

isub

Subtract two integer values.

Syntax:

```
isub<.dsat>.type dst, src0, src1

  .type = { .16b, .32b, .64b }


  - dst is a register
  - src0, src1 can be registers or immediates

  * src0, src1 are interpreted as signed integers.
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
isub<.dsat>.type dst, src0, src1

  .type = { .16bx2 }


  - dst, src0, src1 are 2x16-bit vector registers

  * src0, src1 are interpreted as signed integers.
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                  .type == .16bx2
dst[15:0] = src0[15:0] - src1[15:0]
dst[31:16] = src0[31:16] - src1[31:16]

                                     default
dst = src0 - src1
```

Notes:

Enabling the `.dsat` qualifier clamps the final result to the minimum and maximum values representable by a signed integer of the element bitwidth of the specified `.type`.

Examples:

```
.reg .32b %dst, %src0, %src1;

// subtraction of 32-bit elements
isub.32b  %dst, %src0, %src1;
// element-wise subtraction of 2x16-bit packed values
.reg .v2.16b %dst_packed, %src0_packed, %src1_packed;
isub.16bx2  %dst_packed, %src0_packed, %src1_packed;
```

sdiv

Divide two signed integers.

Syntax:

```
sdiv<.output>.type dst, src0, src1

  .output = { .quo, .rem }
  .type = { .16b, .32b, .64b }


  - dst is a register
  - src0, src1 can be registers or immediates

  * src0, src1 are interpreted as signed integers.
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                  .output == .rem
dst = src0 % src1

                                     default
dst = src0 / src1
```

Notes:

Division by zero is undefined behavior.

Examples:

```
.reg .32b %dst, %src0, %src1;

// compute quotent from 32-bit signed integer division
sdiv.quo.32b  %dst, %src0, %src1;
// compute remainder from 32-bit signed integer division
sdiv.rem.32b  %dst, %src0, %src1;
```

smad

Multiply and add signed integers.

Widening signed multiply-add:

```
smad.precision.type dst, src0, src1, src2

  .precision = { .full }
  .type = { .16b, .32b }


  - dst is a register
  - src0, src1, src2 can be registers or immediates

  * src0, src1, src2 are interpreted as signed integers.
```

Restrictions

- bitwidth of `src0` and `src1` must match the bitwidth of `.type`
- bitwidth of `dst` and `src2` must be twice the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Signed multiply-add:

```
smad.type dst, src0, src1, src2

  .type = { .16b, .32b, .64b }


  - dst is a register
  - src0, src1, src2 can be registers or immediates

  * src0, src1, src2 are interpreted as signed integers.
```

Restrictions

- bitwidth of `dst`, `src0`, `src1`, and `src2` must match the bitwidth of `.type`

PISA Notes

- requires PISA target **100** or compatible
- introduced in PISA version **0.1**

packed:

```
smad.type dst, src0, src1, src2

  .type = { .16bx2 }


  - dst, src0, src1, src2 are 2x16-bit vector registers

  * src0, src1, src2 are interpreted as signed integers.
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                .precision == .full
src0_extend = sign_extend(src0, bitwidth(.type)*2)
src1_extend = sign_extend(src1, bitwidth(.type)*2)
dst = src0_extend * src1_extend + src2

                                  .type == .16bx2
dst[15:0] = src0[15:0] * src1[15:0] + src2[15:0]
dst[31:16] = src0[31:16] * src1[31:16] + src2[31:16]

                                     default
dst = src0 * src1 + src2
```

Examples:

```
.reg .32b %dst, %src0, %src1, %src2;
.reg .64b %dst64, %src64;

// signed multiply and add 32-bit elements
smad.32b  %dst, %src0, %src1, %src2;
// signed multiply and add 32-bit elements with full precision
smad.full.32b  %dst64, %src0, %src1, %src64;
// element-wise signed multiply and add of 2x16-bit packed values
.reg .v2.16b %dst_packed, %src0_packed, %src1_packed, %src2_packed;
smad.16bx2  %dst_packed, %src0_packed, %src1_packed, %src2_packed;
```

smax

Find the maximum of two signed integers.

Syntax:

```
smax.type dst, src0, src1

  .type = { .16b, .32b, .64b }


  - dst is a register
  - src0, src1 can be registers or immediates

  * src0, src1 are interpreted as signed integers.
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
smax.type dst, src0, src1

  .type = { .8bx4, .16bx2 }


  - dst, src0, src1 are packed registers

  * src0, src1 are interpreted as signed integers.
```

Restrictions

- element count and bitwidth of `dst`, `src0` and `src1` must match definitions of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                    .type == .16bx2
dst[15:0] = src0[15:0] > src1[15:0] ? src0[15:0] : src1[15:0]
dst[31:16] = src0[31:16] > src1[31:16] ? src0[31:16] : src1[31:16]

                                    .type == .8bx4
dst[7:0] = src0[7:0] > src1[7:0] ? src0[7:0] : src1[7:0]
dst[15:8] = src0[15:8] > src1[15:8] ? src0[15:8] : src1[15:8]
dst[23:16] = src0[23:16] > src1[23:16] ? src0[23:16] : src1[23:16]
dst[31:24] = src0[31:24] > src1[31:24] ? src0[31:24] : src1[31:24]

                                       default
dst = src0 > src1 ? src0 : src1
```

Examples:

```
.reg .32b %dst, %src0, %src1;

// maximum element from 2 signed 32-bit elements
smax.32b  %dst, %src0, %src1;
// element-wise maximum of 2x16-bit packed values
.reg .v2.16b %dst_packed, %src0_packed, %src1_packed;
smax.16bx2  %dst_packed, %src0_packed, %src1_packed;
// element-wise maximum of 4x8-bit packed values
.reg .v4.8b %dst_4x8, %src0_4x8, %src1_4x8;
smax.8bx4  %dst_4x8, %src0_4x8, %src1_4x8;
```

`smin`

Find the minimum of two signed integers.

Syntax:

```
smin.type dst, src0, src1

  .type = { .16b, .32b, .64b }


  - dst is a register
  - src0, src1 can be registers or immediates

  * src0, src1 are interpreted as signed integers.
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
smin.type dst, src0, src1

  .type = { .8bx4, .16bx2 }


  - dst, src0, src1 are packed registers

  * src0, src1 are interpreted as signed integers.
```

Restrictions

- element count and bitwidth of `dst`, `src0` and `src1` must match definitions of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                  .type == .16bx2
dst[15:0] = src0[15:0] < src1[15:0] ? src0[15:0] : src1[15:0]
dst[31:16] = src0[31:16] < src1[31:16] ? src0[31:16] : src1[31:16]

                                  .type == .8bx4
dst[7:0] = src0[7:0] < src1[7:0] ? src0[7:0] : src1[7:0]
dst[15:8] = src0[15:8] < src1[15:8] ? src0[15:8] : src1[15:8]
dst[23:16] = src0[23:16] < src1[23:16] ? src0[23:16] : src1[23:16]
dst[31:24] = src0[31:24] < src1[31:24] ? src0[31:24] : src1[31:24]

                                     default
dst = src0 < src1 ? src0 : src1
```

Examples:

```
.reg .32b %dst, %src0, %src1;

// minimum element from 2 signed 32-bit elements
smin.32b  %dst, %src0, %src1;
// element-wise minimum of 2x16-bit packed values
.reg .v2.16b %dst_packed, %src0_packed, %src1_packed;
smin.16bx2  %dst_packed, %src0_packed, %src1_packed;
// element-wise minimum of 4x8-bit packed values
.reg .v4.8b %dst_4x8, %src0_4x8, %src1_4x8;
smin.8bx4  %dst_4x8, %src0_4x8, %src1_4x8;
```

`smul`

Multiply two signed integers.

Widening signed multiply:

```
smul.precision.type dst, src0, src1

  .precision = { .full }
  .type = { .16b, .32b }


  - dst is a register
  - src0, src1 can be registers or immediates

  * src0, src1 are interpreted as signed integers.
```

Restrictions

- bitwidth of `src0` and `src1` must match the bitwidth of `.type`
- bitwidth of `dst` must be twice the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Signed multiply:

```
smul.type dst, src0, src1

  .type = { .16b, .32b, .64b }


  - dst is a register
```

```
  - src0, src1 can be registers or immediates

  * src0, src1 are interpreted as signed integers.
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
smul.type dst, src0, src1

  .type = { .16bx2 }


  - dst, src0, src1 are 2x16-bit vector registers

  * src0, src1 are interpreted as signed integers.
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                .precision == .full

src0_extend = sign_extend(src0, bitwidth(.type)*2)
src1_extend = sign_extend(src1, bitwidth(.type)*2)
dst = src0_extend * src1_extend

                                  .type == .16bx2

dst[15:0] = src0[15:0] * src1[15:0]
dst[31:16] = src0[31:16] * src1[31:16]

                                     default

dst = src0 * src1
```

Examples:

```
.reg .32b %dst, %src0, %src1;
.reg .64b %dst64;

// signed multiply 32-bit elements
smul.32b  %dst, %src0, %src1;
// signed multiply 32-bit elements with full precision
smul.full.32b  %dst64, %src0, %src1;
// element-wise signed multiply of 2x16-bit packed values
.reg .v2.16b %dst_packed, %src0_packed, %src1_packed;
smul.16bx2  %dst_packed, %src0_packed, %src1_packed;
```

uaddc

Add unsigned integers with carry.

Add with carry-in:

```
uaddc.ci.type dst, src0, src1, carry-in

  .type = { .32b }


  - dst is a 32-bit register
  - src0, src1 can be 32-bit registers or immediates
  - carry-in is a predicate register

  * src0, src1 are interpreted as unsigned integers.
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Add with carry-out:

```
uaddc.co.type dst, carry-out, src0, src1

  .type = { .32b }


  - dst is a 32-bit register
  - carry-out is a predicate register
  - src0, src1 can be 32-bit registers or immediates

  * src0, src1 are interpreted as unsigned integers.
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Add with carry-in and carry-out:

```
uaddc.ci.co.type dst, carry-out, src0, src1, carry-in

  .type = { .32b }


  - dst is a 32-bit register
  - carry-out, carry-in are predicate registers
  - src0, src1 can be 32-bit registers or immediates

  * src0, src1 are interpreted as unsigned integers.
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
if ('.ci' is set)
   dst = src0 + src1 + carry-in
else
   dst = src0 + src1

if ('.co' is set)
   carry-out = 1 if the result overflows uint32, 0 otherwise
```

Examples:

Add two 32-bit unsigned values with carry

```
.reg .32b %dst, %src0, %src1;
.pred %pin, %pout;

// add two 32-bit unsigned values with carry-out
uaddc.co.32b  %dst, %pout, %src0, %src1;
// add two 32-bit unsigned values with carry-in
uaddc.ci.32b  %dst, %src0, %src1, %pin;
```

64-bit unsigned addition emulated with two 32-bit uaddc instructions

```
.reg .64b %dst, %src0, %src1;
.reg .v2.32b %dst_vec, %src0_vec, %src1_vec;
.pred %p0;

// unpack 64-bit operands into 32-bit halves (.x = low, .y = high)
mov.64b  %src0_vec.xy, %src0;
mov.64b  %src1_vec.xy, %src1;
// add low 32-bit halves with carry-out
uaddc.co.32b  %dst_vec.x, %p0, %src0_vec.x, %src1_vec.x;
// add high 32-bit halves with carry-in
uaddc.ci.32b  %dst_vec.y, %src0_vec.y, %src1_vec.y, %p0;
// pack 32-bit halves back into 64-bit result
mov.64b  %dst, %dst_vec.xy;
```

udiv

Divide two unsigned integers.

Syntax:

```
udiv<.output>.type dst, src0, src1

  .output = { .quo, .rem }
  .type = { .16b, .32b, .64b }


  - dst is a register
  - src0, src1 can be registers or immediates

  * src0, src1 are interpreted as unsigned integers.
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                  .output == .rem
dst = src0 % src1

                                     default
dst = src0 / src1
```

Notes:

Division by zero is undefined behavior.

Examples:

```
.reg .32b %dst, %src0, %src1;

// compute quotent from 32-bit unsigned integer division
udiv.quo.32b  %dst, %src0, %src1;
// compute remainder from 32-bit unsigned integer division
udiv.rem.32b  %dst, %src0, %src1;
```

`umad`

Multiply and add unsigned integers.

Widening unsigned multiply-add:

```
umad.precision.type dst, src0, src1, src2

  .precision = { .full }
  .type = { .16b, .32b }


  - dst is a register
  - src0, src1, src2 can be registers or immediates

  * src0, src1, src2 are interpreted as unsigned integers.
```

Restrictions

- bitwidth of `src0` and `src1` must match the bitwidth of `.type`
- bitwidth of `dst` and `src2` must be twice the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Unsigned multiply-add:

```
umad.type dst, src0, src1, src2

  .type = { .16b, .32b, .64b }
```

```
- dst is a register
- src0, src1, src2 can be registers or immediates

* src0, src1, src2 are interpreted as unsigned integers.
```

Restrictions

- bitwidth of `dst`, `src0`, `src1`, and `src2` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                .precision == .full
src0_extend = zero_extend(src0, bitwidth(.type)*2)
src1_extend = zero_extend(src1, bitwidth(.type)*2)
dst = src0_extend * src1_extend + src2

                                     default
dst = src0 * src1 + src2
```

Examples:

```
.reg .32b %dst, %src0, %src1, %src2;
.reg .64b %dst64, %src64;

// unsigned multiply and add 32-bit elements
umad.32b  %dst, %src0, %src1, %src2;
// unsigned multiply and add 32-bit elements with full precision
umad.full.32b  %dst64, %src0, %src1, %src64;
```

umax

Find the maximum of two unsigned integers.

Syntax:

```
umax.type dst, src0, src1

  .type = { .16b, .32b, .64b }
```

```
  - dst is a register
  - src0, src1 can be registers or immediates

  * src0, src1 are interpreted as unsigned integers.
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
umax.type dst, src0, src1

  .type = { .16bx2 }


  - dst, src0, src1 are 2x16-bit vector registers

  * src0, src1 are interpreted as unsigned integers.
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                    .type == .16bx2
dst[15:0] = src0[15:0] > src1[15:0] ? src0[15:0] : src1[15:0]
dst[31:16] = src0[31:16] > src1[31:16] ? src0[31:16] : src1[31:16]

                                       default
dst = src0 > src1 ? src0 : src1
```

Examples:

```
.reg .32b %dst, %src0, %src1;

// maximum element from 2 unsigned 32-bit elements
umax.32b  %dst, %src0, %src1;
// element-wise maximum of 2x16-bit packed values
.reg .v2.16b %dst_packed, %src0_packed, %src1_packed;
umax.16bx2  %dst_packed, %src0_packed, %src1_packed;
```

umin

Find the minimum of two unsigned integers.

Syntax:

```
umin.type dst, src0, src1

  .type = { .16b, .32b, .64b }


  - dst is a register
  - src0, src1 can be registers or immediates

  * src0, src1 are interpreted as unsigned integers.
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
umin.type dst, src0, src1

  .type = { .16bx2 }


  - dst, src0, src1 are 2x16-bit vector registers

  * src0, src1 are interpreted as unsigned integers.
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                    .type == .16bx2
dst[15:0] = src0[15:0] < src1[15:0] ? src0[15:0] : src1[15:0]
dst[31:16] = src0[31:16] < src1[31:16] ? src0[31:16] : src1[31:16]

                                       default
dst = src0 < src1 ? src0 : src1
```

Examples:

```
.reg .32b %dst, %src0, %src1;

// minimum element from 2 unsigned 32-bit elements
umin.32b  %dst, %src0, %src1;
// element-wise minimum of 2x16-bit packed values
.reg .v2.16b %dst_packed, %src0_packed, %src1_packed;
umin.16bx2  %dst_packed, %src0_packed, %src1_packed;
```

`umul`

Multiply two unsigned integers.

Widening unsigned multiply:

```
umul.precision.type dst, src0, src1

  .precision = { .full }
  .type = { .16b, .32b }


  - dst is a register
  - src0, src1 can be registers or immediates

  * src0, src1 are interpreted as unsigned integers.
```

Restrictions

- bitwidth of `src0` and `src1` must match the bitwidth of `.type`
- bitwidth of `dst` must be twice the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Unsigned multiply:

```
umul.type dst, src0, src1

  .type = { .16b, .32b, .64b }


  - dst is a register
  - src0, src1 can be registers or immediates

  * src0, src1 are interpreted as unsigned integers.
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                .precision == .full
src0_extend = zero_extend(src0, bitwidth(.type)*2)
src1_extend = zero_extend(src1, bitwidth(.type)*2)
dst = src0_extend * src1_extend

                                      default

dst = src0 * src1
```

Examples:

```
.reg .32b %dst, %src0, %src1;
.reg .64b %dst64;

// unsigned multiply 32-bit elements
umul.32b  %dst, %src0, %src1;
```

```
// unsigned multiply 32-bit elements with full precision
umul.full.32b  %dst64, %src0, %src1;
```

Floating Point

`fabs`

Compute the absolute value of a floating-point number.

Syntax:

```
fabs<.ftz>.type dst, src0

  .type = { .bf, .hf, .f, .df }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
fabs<.ftz>.type dst, src0

  .type = { .bfx2, .hfx2 }


  - dst, src0 are 2x16-bit vector registers
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                              .type == .bfx2 || .type == .hfx2
dst[15:0] = |src0[15:0]|
dst[31:16] = |src0[31:16]|

                                          default
dst = |src0|
```

Notes:

If the source is NaN, then a quiet NaN is returned with its sign bit cleared.

Examples:

```
.reg .32b %dst, %src0;

// absolute value of a single-precision element
fabs.f  %dst, %src0;
// absolute value of a single-precision element with flush-to-zero
fabs.ftz.f  %dst, %src0;
// element-wise absolute value of a 2 x half-float packed value
.reg .v2.16b %dst_packed, %src0_packed;
fabs.hfx2  %dst_packed, %src0_packed;
```

`fadd`

Add two floating-point numbers.

Syntax:

```
fadd<.rndmode><.ftz><.dsat>.type dst, src0, src1

  .rndmode = { .re, .rd, .ru, .rz, .rna }
  .type = { .bf, .hf, .f, .df }


  - dst is a register
  - src0, src1 can be registers or immediates
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
fadd<.rndmode><.ftz><.dsat>.type dst, src0, src1

  .rndmode = { .re, .rd, .ru, .rz, .rna }
  .type = { .bfx2, .hfx2 }


  - dst, src0, src1 are 2x16-bit vector registers
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                          .type == .bfx2 || .type == .hfx2
dst[15:0] = src0[15:0] + src1[15:0]
dst[31:16] = src0[31:16] + src1[31:16]

                                      default
dst = src0 + src1
```

Notes:

Enabling saturation via `.dsat` qualifier clamps the final result to [0.0, 1.0]. NaN results are flushed to +0.0.

Examples:

```
.reg .32b %dst, %src0, %src1;

// addition of single-precision elements
fadd.f  %dst, %src0, %src1;
// addition of single-precision elements with rounding
fadd.ru.f  %dst, %src0, %src1;
// element-wise addition of 2 x half-precision packed values
.reg .v2.16b %dst_packed, %src0_packed, %src1_packed;
fadd.hfx2  %dst_packed, %src0_packed, %src1_packed;
```

`fcos`

Compute the cosine of a floating-point number.

Syntax:

```
fcos<.ftz>.type dst, src0

  .type = { .bf, .hf, .f }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
fcos<.ftz>.type dst, src0

  .type = { .bfx2, .hfx2 }


  - dst, src0 are 2x16-bit vector registers
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                              .type == .bfx2 || .type == .hfx2
dst[15:0] = cos(src0[15:0])
dst[31:16] = cos(src0[31:16])

                                          default

dst = cos(src0)
```

Notes:

The maximum relative error for `fcos` is 2 ULP.

Examples:

```
.reg .32b %dst, %src0;

// cosine value of a single-precision element
fcos.f  %dst, %src0;
// cosine value of a single-precision element with flush-to-zero
fcos.ftz.f  %dst, %src0;
// element-wise cosine of a 2 x half-float packed value
.reg .v2.16b %dst_packed, %src0_packed;
fcos.hfx2  %dst_packed, %src0_packed;
```

`fdiv`

Divide two floating-point numbers.

Floating-point division:

```
fdiv<.rndmode><.ftz>.type dst, src0, src1

  .rndmode = { .re, .rd, .ru, .rz, .rna }
  .type = { .f, .df }


  - dst is a register
  - src0, src1 can be registers or immediates
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Fast floating-point division:

```
fdiv<.ftz>.fast.type dst, src0, src1

  .type = { .bf, .hf, .f, .df }


  - dst is a register
  - src0, src1 can be registers or immediates
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
dst = src0 / src1
```

Notes:

The maximum relative error for `fdiv` with `.fast` qualifier is:

- 0.5 ULP for `.bf`, `.hf`
- 2.5 ULP for `.f`
- 2.0 ULP for `.df`

Instructions without `.fast` qualifier produce IEEE compliant result.

Examples:

```
.reg .32b %dst, %src0, %src1;

// division of single-precision elements
fdiv.f  %dst, %src0, %src1;
// division of single-precision elements with flush-to-zero
```

```
fdiv.ftz.f  %dst, %src0, %src1;
// fast division of single-precision elements
fdiv.fast.f  %dst, %src0, %src1;
```

`fexp2`

Compute the base-2 exponential of a floating-point number.

Syntax:

```
fexp2<.ftz>.type dst, src0

  .type = { .bf, .hf, .f }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
fexp2<.ftz>.type dst, src0

  .type = { .bfx2, .hfx2 }


  - dst, src0 are 2x16-bit vector registers
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                              .type == .bfx2 || .type == .hfx2
dst[15:0] = 2^(src0[15:0])
dst[31:16] = 2^(src0[31:16])

                                          default

dst = 2^(src0)
```

Notes:

The maximum relative error for `fexp2` is 2 ULP.

Examples:

```
.reg .32b %dst, %src0;

// base-2 exponential value of a single-precision element
fexp2.f  %dst, %src0;
// base-2 exponential value of a single-precision element with flush-to-zero
fexp2.ftz.f  %dst, %src0;
// element-wise base-2 exponential of a 2 x half-float packed value
.reg .v2.16b %dst_packed, %src0_packed;
fexp2.hfx2  %dst_packed, %src0_packed;
```

`flog2`

Compute the base-2 logarithm of a floating-point number.

Syntax:

```
flog2<.ftz>.type dst, src0

  .type = { .bf, .hf, .f }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- requires PISA target **100** or compatible
- introduced in PISA version **0.1**

packed:

```
flog2<.ftz>.type dst, src0

  .type = { .bfx2, .hfx2 }


  - dst, src0 are 2x16-bit vector registers
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                              .type == .bfx2 || .type == .hfx2
dst[15:0] = log2(src0[15:0])
dst[31:16] = log2(src0[31:16])

                                        default

dst = log2(src0)
```

Notes:

The maximum relative error for `flog2` is:

- 2^-23 for `src0` value in the range of [0.5..1.5)
- 1 ULP for all other values

Examples:

```
.reg .32b %dst, %src0;

// base-2 log value of a single-precision element
flog2.f  %dst, %src0;
// base-2 log value of a single-precision element with flush-to-zero
flog2.ftz.f  %dst, %src0;
// element-wise base-2 log of a 2 x half-float packed value
.reg .v2.16b %dst_packed, %src0_packed;
flog2.hfx2  %dst_packed, %src0_packed;
```

`fmad`

Compute the fused multiply-add of floating-point numbers.

Syntax:

```
fmad<.rndmode><.ftz><.dsat>.type dst, src0, src1, src2

  .rndmode = { .re, .rd, .ru, .rz, .rna }
  .type = { .bf, .hf, .f, .df }


  - dst is a register
  - src0, src1, src2 can be registers or immediates
```

Restrictions

- bitwidth of `dst`, `src0`, `src1`, and `src2` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
fmad<.rndmode><.ftz><.dsat>.type dst, src0, src1, src2

  .rndmode = { .re, .rd, .ru, .rz, .rna }
  .type = { .bfx2, .hfx2 }


  - dst, src0, src1, src2 are 2x16-bit vector registers
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                              .type == .bfx2 || .type == .hfx2
dst[15:0] = src0[15:0] * src1[15:0] + src2[15:0]
dst[31:16] = src0[31:16] * src1[31:16] + src2[31:16]

                                        default

dst = src0 * src1 + src2
```

Notes:

Enabling saturation via `.dsat` qualifier clamps the final result to [0.0, 1.0]. NaN results are flushed to +0.0.

Examples:

```
.reg .32b %dst, %src0, %src1, %src2;

// fused multiply-add of single-precision elements
fmad.f  %dst, %src0, %src1, %src2;
// fused multiply-add of single-precision elements with rounding
fmad.ru.f  %dst, %src0, %src1, %src2;
// element-wise fused multiply-add of 2 x half-precision packed values
.reg .v2.16b %dst_packed, %src0_packed, %src1_packed, %src2_packed;
fmad.hfx2  %dst_packed, %src0_packed, %src1_packed, %src2_packed;
```

`fmax`

Find the maximum of two floating-point numbers.

Syntax:

```
fmax<.ftz><.dsat><.nanp>.type dst, src0, src1

  .type = { .bf, .hf, .f, .df }


  - dst is a register
  - src0, src1 can be registers or immediates
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
fmax<.ftz><.dsat><.nanp>.type dst, src0, src1

  .type = { .bfx2, .hfx2 }


  - dst, src0, src1 are 2x16-bit vector registers
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                              .type == .bfx2 || .type == .hfx2
dst[15:0] = fmax(src0[15:0], src1[15:0])
dst[31:16] = fmax(src0[31:16], src1[31:16])

                                        default
dst = fmax(src0, src1)
```

Notes:

By default, NaN inputs follow the IEEE 754 `maximumNumber` semantics: if one of the sources is NaN, the other source is returned, and if both sources are NaN a quiet NaN is returned.

When `.nanp` is specified, NaN inputs follow the IEEE 754 `maximum` semantics: if either source is a NaN of any type, the result is a quiet NaN.

Enabling saturation via `.dsat` qualifier clamps the final result to [0.0, 1.0]. NaN results are flushed to +0.0.

Examples:

```
.reg .32b %dst, %src0, %src1;

// maximum value of a single-precision element
fmax.f  %dst, %src0, %src1;
// maximum value of a single-precision element with flush-to-zero
fmax.ftz.f  %dst, %src0, %src1;
// maximum value of a single-precision element with NaN propagation
fmax.nanp.f  %dst, %src0, %src1;
// element-wise maximum of 2 x half-float packed values
.reg .v2.16b %dst_packed, %src0_packed, %src1_packed;
fmax.hfx2  %dst_packed, %src0_packed, %src1_packed;
```

`fmin`

Find the minimum of two floating-point numbers.

Syntax:

```
fmin<.ftz><.dsat><.nanp>.type dst, src0, src1

  .type = { .bf, .hf, .f, .df }
```

```
  - dst is a register
  - src0, src1 can be registers or immediates
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
fmin<.ftz><.dsat><.nanp>.type dst, src0, src1

  .type = { .bfx2, .hfx2 }


  - dst, src0, src1 are 2x16-bit vector registers
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                              .type == .bfx2 || .type == .hfx2
dst[15:0] = fmin(src0[15:0], src1[15:0])
dst[31:16] = fmin(src0[31:16], src1[31:16])

                                          default
dst = fmin(src0, src1)
```

Notes:

By default, NaN inputs follow the IEEE 754 `minimumNumber` semantics: if one of the sources is NaN, the other source is returned, and if both sources are NaN a quiet NaN is returned.

When `.nanp` is specified, NaN inputs follow the IEEE 754 `minimum` semantics: if either source is a NaN of any type, the result is a quiet NaN.

Enabling saturation via `.dsat` qualifier clamps the final result to [0.0, 1.0]. NaN results are flushed to +0.0.

Examples:

```
.reg .32b %dst, %src0, %src1;

// minimum value of a single-precision element
fmin.f  %dst, %src0, %src1;
// minimum value of a single-precision element with flush-to-zero
fmin.ftz.f  %dst, %src0, %src1;
// minimum value of a single-precision element with NaN propagation
fmin.nanp.f  %dst, %src0, %src1;
// element-wise minimum of 2 x half-float packed values
.reg .v2.16b %dst_packed, %src0_packed, %src1_packed;
fmin.hfx2  %dst_packed, %src0_packed, %src1_packed;
```

`fmul`

Multiply two floating-point numbers.

Widening floating-point multiply:

```
fmul.precision<.rndmode><.ftz><.dsat>.type dst, src0, src1

  .precision = { .full }
  .rndmode = { .re, .rd, .ru, .rz, .rna }
  .type = { .bf, .hf }


  - dst is a register
  - src0, src1 can be registers or immediates
```

Restrictions

- bitwidth of `src0` and `src1` must match the bitwidth of `.type`
- bitwidth of `dst` must be twice the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Floating-point multiply:

```
fmul<.rndmode><.ftz><.dsat>.type dst, src0, src1

  .rndmode = { .re, .rd, .ru, .rz, .rna }
  .type = { .bf, .hf, .f, .df }


  - dst is a register
  - src0, src1 can be registers or immediates
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
fmul<.rndmode><.ftz><.dsat>.type dst, src0, src1

  .rndmode = { .re, .rd, .ru, .rz, .rna }
  .type = { .bfx2, .hfx2 }


  - dst, src0, src1 are 2x16-bit vector registers
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                  .precision == .full
src0_extend = extend(src0, bitwidth(.type)*2)
src1_extend = extend(src1, bitwidth(.type)*2)
dst = src0_extend * src1_extend

                              .type == .bfx2 || .type == .hfx2
```

```
dst[15:0] = src0[15:0] * src1[15:0]
dst[31:16] = src0[31:16] * src1[31:16]

                                        default

dst = src0 * src1
```

Notes:

Enabling saturation via `.dsat` qualifier clamps the final result to [0.0, 1.0]. NaN results are flushed to +0.0.

Examples:

```
.reg .16b %srch0, %srch1;
.reg .32b %dst, %src0, %src1;

// multiplication of single-precision elements
fmul.f  %dst, %src0, %src1;
// multiplication of single-precision elements with rounding
fmul.ru.f  %dst, %src0, %src1;
// element-wise multiplication of 2 x half-precision packed values
.reg .v2.16b %dst_packed, %src0_packed, %src1_packed;
fmul.hfx2  %dst_packed, %src0_packed, %src1_packed;
// multiplication of half-precision elements with full precision
fmul.full.hf  %dst, %srch0, %srch1;
```

`fneg`

Negate a floating-point number.

Syntax:

```
fneg<.ftz>.type dst, src0

  .type = { .bf, .hf, .f, .df }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**

- requires PISA target **100** or compatible

packed:

```
fneg<.ftz>.type dst, src0

  .type = { .bfx2, .hfx2 }


  - dst, src0 are 2x16-bit vector registers
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                            .type == .bfx2 || .type == .hfx2
dst[15:0] = -src0[15:0]
dst[31:16] = -src0[31:16]

                                          default

dst = -src0
```

Examples:

```
.reg .32b %dst, %src0;

// negate value of a single-precision element
fneg.f  %dst, %src0;
// negate value of a single-precision element with flush-to-zero
fneg.ftz.f  %dst, %src0;
// element-wise negate of a 2 x half-float packed value
.reg .v2.16b %dst_packed, %src0_packed;
fneg.hfx2  %dst_packed, %src0_packed;
```

`frc`

Compute the fractional part of a floating-point number.

Syntax:

```
frc<.ftz>.type dst, src0

  .type = { .f }


  - dst is a 32-bit register
  - src0 can be a 32-bit register or an immediate
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
dst = src0 - round(src0, .rd)
```

Notes:

The `.rd` rounding mode is always applied for fraction.

Examples:

```
.reg .32b %dst, %src0;

// fraction value of a single-precision element
frc.f  %dst, %src0;
// fraction value of a single-precision element with flush-to-zero
frc.ftz.f  %dst, %src0;
```

`frcp`

Compute the reciprocal of a floating-point number.

Syntax:

```
frcp<.ftz>.type dst, src0

  .type = { .bf, .hf, .f }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
frcp<.ftz>.type dst, src0

  .type = { .bfx2, .hfx2 }


  - dst, src0 are 2x16-bit vector registers
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                              .type == .bfx2 || .type == .hfx2
dst[15:0] = 1/src0[15:0]
dst[31:16] = 1/src0[31:16]

                                          default

dst = 1/src0
```

Notes:

The maximum relative error for `frcp` is 1 ULP.

Examples:

```
.reg .32b %dst, %src0;

// reciprocal value of a single-precision element
```

```
frcp.f  %dst, %src0;
// reciprocal value of a single-precision element with flush-to-zero
frcp.ftz.f  %dst, %src0;
// element-wise reciprocal of a 2 x half-float packed value
.reg .v2.16b %dst_packed, %src0_packed;
frcp.hfx2  %dst_packed, %src0_packed;
```

`frsqrt`

Compute the reciprocal square root of a floating-point number.

Syntax:

```
frsqrt<.ftz>.fast.type dst, src0

  .type = { .bf, .hf, .f, .df }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- requires PISA target **100** or compatible
- introduced in PISA version **0.1**

packed:

```
frsqrt<.ftz>.fast.type dst, src0

  .type = { .bfx2, .hfx2 }


  - dst, src0 are 2x16-bit vector registers
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                              .type == .bfx2 || .type == .hfx2
dst[15:0] = 1/sqrt(src0[15:0])
dst[31:16] = 1/sqrt(src0[31:16])

                                          default

dst = 1/sqrt(src0)
```

Notes:

The maximum relative error for `frsqrt` is 1 ULP.

Examples:

```
.reg .32b %dst, %src0;

// reciprocal square root value of a single-precision element
frsqrt.fast.f  %dst, %src0;
// reciprocal square root value of a single-precision element with flush-to-zero
frsqrt.ftz.fast.f  %dst, %src0;
// element-wise reciprocal square root of a 2 x half-float packed value
.reg .v2.16b %dst_packed, %src0_packed;
frsqrt.fast.hfx2  %dst_packed, %src0_packed;
```

`fsin`

Compute the sine of a floating-point number.

Syntax:

```
fsin<.ftz>.type dst, src0

  .type = { .bf, .hf, .f }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- requires PISA target **100** or compatible
- introduced in PISA version **0.1**

packed:

```
fsin<.ftz>.type dst, src0

  .type = { .bfx2, .hfx2 }


  - dst, src0 are 2x16-bit vector registers
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                              .type == .bfx2 || .type == .hfx2
dst[15:0] = sin(src0[15:0])
dst[31:16] = sin(src0[31:16])

                                        default

dst = sin(src0)
```

Notes:

The maximum relative error for `fsin` is 2 ULP.

Examples:

```
.reg .32b %dst, %src0;

// sine value of a single-precision element
fsin.f  %dst, %src0;
// sine value of a single-precision element with flush-to-zero
fsin.ftz.f  %dst, %src0;
// element-wise sine of a 2 x half-float packed value
.reg .v2.16b %dst_packed, %src0_packed;
fsin.hfx2  %dst_packed, %src0_packed;
```

`fsqrt`

Compute the square root of a floating-point number.

Floating-point square root:

```
fsqrt<.rndmode><.ftz>.type dst, src0

  .rndmode = { .re, .rd, .ru, .rz, .rna }
  .type = { .f, .df }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Fast floating-point square root:

```
fsqrt<.ftz>.fast.type dst, src0

  .type = { .bf, .hf, .f, .df }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
fsqrt<.ftz>.fast.type dst, src0
```

```
.type = { .bfx2, .hfx2 }

- dst, src0 are 2x16-bit vector registers
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                              .type == .bfx2 || .type == .hfx2
dst[15:0] = sqrt(src0[15:0])
dst[31:16] = sqrt(src0[31:16])

                                          default

dst = sqrt(src0)
```

Notes:

Instructions with `.fast` qualifier result in the maximum relative error of:

- 2 ULP for `.df`
- 1 ULP for all other `.type` values

Instructions without `.fast` qualifier produce IEEE compliant result.

Examples:

```
.reg .32b %dst, %src0;

// square root value of a single-precision element
fsqrt.f  %dst, %src0;
// square root value of a single-precision element with flush-to-zero
fsqrt.ftz.f  %dst, %src0;
// element-wise square root of a 2 x half-float packed value
.reg .v2.16b %dst_packed, %src0_packed;
fsqrt.fast.hfx2  %dst_packed, %src0_packed;
```

`fsub`

Subtract two floating-point numbers.

Syntax:

```
fsub<.rndmode><.ftz><.dsat>.type dst, src0, src1

  .rndmode = { .re, .rd, .ru, .rz, .rna }
  .type = { .bf, .hf, .f, .df }


  - dst is a register
  - src0, src1 can be registers or immediates
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
fsub<.rndmode><.ftz><.dsat>.type dst, src0, src1

  .rndmode = { .re, .rd, .ru, .rz, .rna }
  .type = { .bfx2, .hfx2 }


  - dst, src0, src1 are 2x16-bit vector registers
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                              .type == .bfx2 || .type == .hfx2
dst[15:0] = src0[15:0] - src1[15:0]
dst[31:16] = src0[31:16] - src1[31:16]

                                        default

dst = src0 - src1
```

Notes:

Enabling saturation via `.dsat` qualifier clamps the final result to [0.0, 1.0]. NaN results are flushed to +0.0.

Examples:

```
.reg .32b %dst, %src0, %src1;

// subtraction of single-precision elements
fsub.f  %dst, %src0, %src1;
// subtraction of single-precision elements with rounding
fsub.ru.f  %dst, %src0, %src1;
// element-wise subtraction of 2 x half-precision packed values
.reg .v2.16b %dst_packed, %src0_packed, %src1_packed;
fsub.hfx2  %dst_packed, %src0_packed, %src1_packed;
```

`ftanh`

Compute the hyperbolic tangent of a floating-point number.

Syntax:

```
ftanh<.ftz>.type dst, src0

  .type = { .bf, .hf, .f }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
ftanh<.ftz>.type dst, src0

  .type = { .bfx2, .hfx2 }


  - dst, src0 are 2x16-bit vector registers
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                              .type == .bfx2 || .type == .hfx2
dst[15:0] = tanh(src0[15:0])
dst[31:16] = tanh(src0[31:16])

                                          default

dst = tanh(src0)
```

Notes:

The maximum relative error for `ftanh` is 2 ULP.

Examples:

```
.reg .32b %dst, %src0;

// hyperbolic tangent value of a single-precision element
ftanh.f  %dst, %src0;
// hyperbolic tangent value of a single-precision element with flush-to-zero
ftanh.ftz.f  %dst, %src0;
// element-wise hyperbolic tangent of a 2 x half-float packed value
.reg .v2.16b %dst_packed, %src0_packed;
ftanh.hfx2  %dst_packed, %src0_packed;
```

Logic and Shift

and

Compute bitwise AND operation.

Syntax:

```
and.type dst, src0, src1

  .type = { .16b, .32b, .64b }


  - dst is a register
  - src0, src1 can be registers or immediates
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
dst = src0 & src1
```

Examples:

```
.reg .32b %dst, %src0, %src1;

// and of 32-bit elements
and.32b  %dst, %src0, %src1;
```

`asr`

Perform an arithmetic right shift.

Syntax:

```
asr.type dst, src0, src1

  .type = { .16b, .32b, .64b }


  - dst is a register
  - src0 can be a register or an immediate
  - src1 can be a 32-bit register or an immediate

  * src0 is interpreted as signed integer.
  * src1 is interpreted as unsigned integer.
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
asr.type dst, src0, src1

  .type = { .16bx2 }


  - dst, src0 are 2x16-bit vector registers
  - src1 can be a 32-bit register or an immediate

  * src0 is interpreted as signed integer.
  * src1 is interpreted as unsigned integer.
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                  .type == .16bx2
shiftl = unsigned(src1[5:0])
shifth = unsigned(src1[21:16])
dst[31:16] = src0[31:16] >> shifth
dst[15:0]  = src0[15:0]  >> shiftl

                                     default

shift = unsigned(src1[5:0])
dst = src0 >> shift
```

Examples:

```
.reg .32b %dst, %src0, %src1;

// arithmetic right shift of 32-bit element
asr.32b  %dst, %src0, %src1;
.reg .v2.16b %dst_packed, %src0_packed;
// element-wise arithmetic right shift of 2x16-bit packed value
asr.16bx2  %dst_packed, %src0_packed, %src1;
```

bfi

Replace bits in `src0` with bits from `src1`.

Syntax:

```
bfi.type dst, src0, src1, src2, src3

  .type = { .32b }


  - dst is a 32-bit register
  - src0, src1, src2, src3 can be 32-bit registers or immediates
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
offset  = src3[4:0]  // starting bit position
width   = umin(src2[5:0], bitwidth(.type))          // requested number of bits
width   = umin(width, bitwidth(.type) - offset)     // clamp: ignore bits beyond the MSB

// build a mask of `width` ones starting at `offset`, i.e. 1s in [offset, offset+width)
mask_w  = width < bitwidth(.type) ? (1u << width) - 1u : ~0u;  // `width` low bits set
mask    = mask_w << offset;                                    // shift to position
dst     = (src0 & ~mask) | ((src1 << offset) & mask);
```

Examples:

```
.reg .32b %dst, %src0, %src1, %width, %offset;

// extract %width bits from %src1 and insert them at %offset into %src0
bfi.32b  %dst, %src0, %src1, %width, %offset;
```

bfn

Perform an arbitrary bitwise operation.

Syntax:

```
bfn.operation.type dst, src0, src1, src2

  .operation = {see Bfn Operations table}
  .type = { .32b }
```

```
- dst is a 32-bit register
- src0, src1, src2 can be 32-bit registers or immediates
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
dst = bfn[.operation](src0, src1, src2)
```

Notes:

The `.operation` encoding is derived by replacing each source with a corresponding constant and evaluating the desired boolean expression:

- `src0` = `0xaa`
- `src1` = `0xcc`
- `src2` = `0xf0`

For example:

- `src0 | ~src1 & src2` → `0xaa | ~0xcc & 0xf0` = `0xba`
- `(src0 ^ src2) & (src1 ^ src2)` → `(0xaa ^ 0xf0) & (0xcc ^ 0xf0)` = `0x18`

Bfn Operations

.operation	Logic Function
0x00 (invalid)	
0x01	~src0 & ~src1 & ~src2
0x02	src0 & ~src1 & ~src2
0x03	~src1 & ~src2
0x04	~src0 & src1 & ~src2
0x05	~src0 & ~src2
0x06	(src0 ^ src1) & ~src2
0x07	(~src0 | ~src1) & ~src2
0x08	src0 & src1 & ~src2
0x09	(src0 ^ ~src1) & ~src2
0x0a	src0 & ~src2
0x0b	(src0 | ~src1) & ~src2
0x0c	src1 & ~src2
0x0d	(~src0 | src1) & ~src2
0x0e	(src0 | src1) & ~src2

0x0f	~src2
0x10	~src0 & ~src1 & src2
0x11	~src0 & ~src1
0x12	(src0 ^ src2) & ~src1
0x13	(~src0 | ~src2) & ~src1
0x14	~src0 & (src1 ^ src2)
0x15	~src0 & (~src1 | ~src2)
0x16	src0 ^ (src0 & src1 | src1 ^ src2)
0x17	~src0 & ~src1 | (~src0 | ~src1) & ~src2
0x18	(src0 ^ src2) & (src1 ^ src2)
0x19	src0 ^ (src0 & src2 | ~src1)
0x1a	src0 ^ (src0 | ~src1) & src2
0x1b	src0 & ~src2 | ~src0 & ~src1
0x1c	(src0 & src2 | src1) ^ src2
0x1d	~src0 & ~src1 | src1 & ~src2
0x1e	(src0 | src1) ^ src2
0x1f	~src0 & ~src1 | ~src2
0x20	src0 & ~src1 & src2
0x21	(src0 ^ ~src2) & ~src1
0x22	src0 & ~src1
0x23	(src0 | ~src2) & ~src1
0x24	(src0 ^ src1) & (src1 ^ src2)
0x25	src0 ^ (src0 & src1 | ~src2)
0x26	src0 ^ (src0 | ~src2) & src1
0x27	src0 & ~src1 | ~src0 & ~src2
0x28	src0 & (src1 ^ src2)
0x29	src0 ^ (src0 | ~src1) & (src1 ^ ~src2)
0x2a	src0 & (~src1 | ~src2)
0x2b	src0 & ~src1 | (src0 | ~src1) & ~src2
0x2c	(~src0 & src2 | src1) ^ src2
0x2d	(~src0 | src1) ^ src2
0x2e	src0 & ~src1 | src1 & ~src2
0x2f	src0 & ~src1 | ~src2
0x30	~src1 & src2
0x31	(~src0 | src2) & ~src1
0x32	(src0 | src2) & ~src1
0x33	~src1
0x34	(src0 & src1 | src2) ^ src1
0x35	~src0 & ~src2 | ~src1 & src2

0x36	(src0 | src2) ^ src1
0x37	~src0 & ~src2 | ~src1
0x38	(~src0 & src1 | src2) ^ src1
0x39	(~src0 | src2) ^ src1
0x3a	src0 & ~src2 | ~src1 & src2
0x3b	src0 & ~src2 | ~src1
0x3c	src1 ^ src2
0x3d	~src0 & ~src1 | src1 ^ src2
0x3e	src0 & ~src1 | src1 ^ src2
0x3f	~src1 | ~src2
0x40	~src0 & src1 & src2
0x41	~src0 & (src1 ^ ~src2)
0x42	(src0 ^ src1) & (src1 ^ ~src2)
0x43	(src0 & src1 | ~src2) ^ src1
0x44	~src0 & src1
0x45	~src0 & (src1 | ~src2)
0x46	src0 ^ (src0 & src2 | src1)
0x47	~src0 & src1 | ~src1 & ~src2
0x48	(src0 ^ src2) & src1
0x49	src0 ^ (src0 & src2 | src1 ^ ~src2)
0x4a	src0 ^ (src0 | src1) & src2
0x4b	(src0 | ~src1) ^ src2
0x4c	(~src0 | ~src2) & src1
0x4d	~src0 & src1 | (~src0 | src1) & ~src2
0x4e	src0 & ~src2 | ~src0 & src1
0x4f	~src0 & src1 | ~src2
0x50	~src0 & src2
0x51	~src0 & (~src1 | src2)
0x52	src0 ^ (src0 & src1 | src2)
0x53	~src0 & src2 | ~src1 & ~src2
0x54	~src0 & (src1 | src2)
0x55	~src0
0x56	src0 ^ (src1 | src2)
0x57	~src0 | ~src1 & ~src2
0x58	src0 ^ (src0 & ~src1 | src2)
0x59	src0 ^ (~src1 | src2)
0x5a	src0 ^ src2
0x5b	src0 ^ src2 | ~src0 & ~src1
0x5c	~src0 & src2 | src1 & ~src2

0x5d	~src0 | src1 & ~src2
0x5e	src0 ^ src2 | src1 & ~src2
0x5f	~src0 | ~src2
0x60	(src0 ^ src1) & src2
0x61	src0 ^ (src0 & src1 | src1 ^ ~src2)
0x62	src0 ^ (src0 | src2) & src1
0x63	(src0 | ~src2) ^ src1
0x64	src0 ^ (src0 & ~src2 | src1)
0x65	src0 ^ (src1 | ~src2)
0x66	src0 ^ src1
0x67	src0 ^ src1 | ~src0 & ~src2
0x68	src0 ^ (src0 | src1) & (src1 ^ ~src2)
0x69	src0 ^ src1 ^ ~src2
0x6a	src0 ^ src1 & src2
0x6b	src0 ^ src1 & src2 | ~src1 & ~src2
0x6c	(~src0 | ~src2) ^ ~src1
0x6d	src0 ^ (src1 | ~src2) | src1 & ~src2
0x6e	src0 ^ src1 | src0 & ~src2
0x6f	src0 ^ src1 | ~src2
0x70	(~src0 | ~src1) & src2
0x71	~src0 & src2 | (~src0 | src2) & ~src1
0x72	src0 & ~src1 | ~src0 & src2
0x73	~src0 & src2 | ~src1
0x74	~src0 & src1 | ~src1 & src2
0x75	~src0 | ~src1 & src2
0x76	src0 ^ src1 | ~src0 & src2
0x77	~src0 | ~src1
0x78	(~src0 | ~src1) ^ ~src2
0x79	src0 ^ (~src1 | src2) | ~src1 & src2
0x7a	src0 ^ src2 | src0 & ~src1
0x7b	src0 ^ src2 | ~src1
0x7c	~src0 & src1 | src1 ^ src2
0x7d	~src0 | src1 ^ src2
0x7e	src0 ^ src1 | src1 ^ src2
0x7f	~src0 | ~src1 | ~src2
0x80	src0 & src1 & src2
0x81	(src0 ^ ~src1) & (src1 ^ ~src2)
0x82	src0 & (src1 ^ ~src2)
0x83	(~src0 & src1 | ~src2) ^ src1

0x84	(src0 ^ ~src2) & src1
0x85	src0 ^ (src0 & ~src1 | ~src2)
0x86	src0 ^ (src0 | src1) & (src1 ^ src2)
0x87	(~src0 | ~src1) ^ src2
0x88	src0 & src1
0x89	src0 ^ (src0 | ~src2) & ~src1
0x8a	src0 & (src1 | ~src2)
0x8b	src0 & src1 | ~src1 & ~src2
0x8c	(src0 | ~src2) & src1
0x8d	src0 & src1 | ~src0 & ~src2
0x8e	src0 & src1 | (src0 | src1) & ~src2
0x8f	src0 & src1 | ~src2
0x90	(src0 ^ ~src1) & src2
0x91	src0 ^ (src0 & ~src2 | ~src1)
0x92	src0 ^ (src0 | src2) & (src1 ^ src2)
0x93	(~src0 | ~src2) ^ src1
0x94	src0 ^ (src0 & ~src1 | src1 ^ src2)
0x95	src0 ^ (~src1 | ~src2)
0x96	src0 ^ src1 ^ src2
0x97	src0 ^ (~src1 | ~src2) | ~src1 & ~src2
0x98	src0 ^ (src0 | src2) & ~src1
0x99	src0 ^ ~src1
0x9a	src0 ^ ~src1 & src2
0x9b	src0 ^ ~src1 | src0 & ~src2
0x9c	(src0 | ~src2) ^ ~src1
0x9d	src0 ^ ~src1 | src1 & ~src2
0x9e	src0 ^ ~src1 & src2 | src1 & ~src2
0x9f	src0 ^ ~src1 | ~src2
0xa0	src0 & src2
0xa1	src0 ^ (src0 | ~src1) & ~src2
0xa2	src0 & (~src1 | src2)
0xa3	src0 & src2 | ~src1 & ~src2
0xa4	src0 ^ (src0 | src1) & ~src2
0xa5	src0 ^ ~src2
0xa6	src0 ^ src1 & ~src2
0xa7	src0 ^ ~src2 | src0 & ~src1
0xa8	src0 & (src1 | src2)
0xa9	src0 ^ ~src1 & ~src2
0xaa	src0

0xab	src0 | ~src1 & ~src2
0xac	src0 & src2 | src1 & ~src2
0xad	src0 ^ ~src2 | src0 & src1
0xae	src0 | src1 & ~src2
0xaf	src0 | ~src2
0xb0	(src0 | ~src1) & src2
0xb1	src0 & src2 | ~src0 & ~src1
0xb2	src0 & src2 | (src0 | src2) & ~src1
0xb3	src0 & src2 | ~src1
0xb4	(src0 | ~src1) ^ ~src2
0xb5	src0 ^ ~src2 | ~src0 & ~src1
0xb6	src0 ^ src1 & ~src2 | ~src1 & src2
0xb7	src0 ^ ~src2 | ~src1
0xb8	src0 & src1 | ~src1 & src2
0xb9	src0 ^ ~src1 | src0 & src2
0xba	src0 | ~src1 & src2
0xbb	src0 | ~src1
0xbc	src0 & src1 | src1 ^ src2
0xbd	src0 ^ ~src1 | src1 ^ src2
0xbe	src0 | src1 ^ src2
0xbf	src0 | ~src1 | ~src2
0xc0	src1 & src2
0xc1	(src0 & ~src1 | src2) ^ ~src1
0xc2	(~src0 & ~src1 | src2) ^ ~src1
0xc3	src1 ^ ~src2
0xc4	(~src0 | src2) & src1
0xc5	~src0 & ~src2 | src1 & src2
0xc6	(~src0 | src2) ^ ~src1
0xc7	~src0 & src1 | src1 ^ ~src2
0xc8	(src0 | src2) & src1
0xc9	(src0 | src2) ^ ~src1
0xca	src0 & ~src2 | src1 & src2
0xcb	src0 & src1 | src1 ^ ~src2
0xcc	src1
0xcd	~src0 & ~src2 | src1
0xce	src0 & ~src2 | src1
0xcf	src1 | ~src2
0xd0	(~src0 | src1) & src2
0xd1	~src0 & ~src1 | src1 & src2

0xd2	(~src0 | src1) ^ ~src2
0xd3	~src0 & src2 | src1 ^ ~src2
0xd4	~src0 & src1 | (~src0 | src1) & src2
0xd5	~src0 | src1 & src2
0xd6	src0 ^ (src1 | src2) | src1 & src2
0xd7	~src0 | src1 ^ ~src2
0xd8	src0 & src1 | ~src0 & src2
0xd9	src0 ^ ~src1 | src1 & src2
0xda	src0 ^ src2 | src0 & src1
0xdb	src0 ^ src2 | src1 ^ ~src2
0xdc	~src0 & src2 | src1
0xdd	~src0 | src1
0xde	src0 ^ src2 | src1
0xdf	~src0 | src1 | ~src2
0xe0	(src0 | src1) & src2
0xe1	(src0 | src1) ^ ~src2
0xe2	src0 & ~src1 | src1 & src2
0xe3	src0 & src2 | src1 ^ ~src2
0xe4	src0 & src2 | ~src0 & src1
0xe5	src0 ^ ~src2 | src1 & src2
0xe6	src0 ^ src1 | src0 & src2
0xe7	src0 ^ src1 | src1 ^ ~src2
0xe8	src0 & src1 | (src0 | src1) & src2
0xe9	src0 ^ ~src1 & ~src2 | src1 & src2
0xea	src0 | src1 & src2
0xeb	src0 | src1 ^ ~src2
0xec	src0 & src2 | src1
0xed	src0 ^ ~src2 | src1
0xee	src0 | src1
0xef	src0 | src1 | ~src2
0xf0	src2
0xf1	~src0 & ~src1 | src2
0xf2	src0 & ~src1 | src2
0xf3	~src1 | src2
0xf4	~src0 & src1 | src2
0xf5	~src0 | src2
0xf6	src0 ^ src1 | src2
0xf7	~src0 | ~src1 | src2
0xf8	src0 & src1 | src2

0xf9	src0 ^ ~src1 | src2
0xfa	src0 | src2
0xfb	src0 | ~src1 | src2
0xfc	src1 | src2
0xfd	~src0 | src1 | src2
0xfe	src0 | src1 | src2
0xff (invalid)	

Examples:

```
.reg .32b %dst, %src0, %src1, %src2;

// calculate src0 | ~src1 & src2
bfn.0xba.32b  %dst, %src0, %src1, %src2;
// calculate src0 ^ ~src1 & ~src2 | src1 & src2
bfn.0xe9.32b  %dst, %src0, %src1, %src2;
```

bfrev

Reverse bits in a source value.

Syntax:

```
bfrev.type dst, src0

  .type = { .32b }


  - dst is a 32-bit register
  - src0 can be a 32-bit register or an immediate
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
width = bitwidth(.type)
for (i = 0; i < width; ++i) {
   dst[i] = src0[width - i - 1]
}
```

Examples:

```
.reg .32b %dst, %src0;

// reverse bits in a 32-bit element
bfrev.32b  %dst, %src0;
```

cbit

Count the number of set bits.

Syntax:

```
cbit.type dst, src0

  .type = { .16b, .32b }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
dst = 0
for (i = 0; i < bitwidth(.type); ++i) {
  if (src0 & (1 << i))
    dst++
}
```

Examples:

```
.reg .32b %dst, %src0;

// count bits in a 32-bit element
cbit.32b  %dst, %src0;
```

fbh

Find the first bit set from the MSB side.

Syntax:

```
fbh.type dst, src0

  .type = { .16b, .32b }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
if (src0 == 0) {
  dst = ~src0
} else {
  for (dst = 0; dst < bitwidth(.type); ++dst) {
    if (src0 & (1 << (bitwidth(.type) - 1 - dst)))
      break
  }
}
```

Examples:

```
.reg .32b %dst, %src0;

// find first bit set (MSB) in a 32-bit element
fbh.32b  %dst, %src0;
```

fbl

Find the first bit set from the LSB side.

Syntax:

```
fbl.type dst, src0

  .type = { .16b, .32b }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
if (src0 == 0) {
  dst = ~src0
} else {
  for (dst = 0; dst < bitwidth(.type); ++dst) {
    if (src0 & (1 << dst))
      break
  }
}
```

Examples:

```
.reg .32b %dst, %src0;

// find first bit set (LSB) in a 32-bit element
fbl.32b  %dst, %src0;
```

not

Compute bitwise NOT operation.

Syntax:

```
not.type dst, src0

  .type = { .16b, .32b, .64b }
```

```
  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
dst = ~src0
```

Examples:

```
.reg .32b %dst, %src0;

// not of 32-bit element
not.32b  %dst, %src0;
```

or

Compute bitwise OR operation.

Syntax:

```
or.type dst, src0, src1

  .type = { .16b, .32b, .64b }


  - dst is a register
  - src0, src1 can be registers or immediates
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
dst = src0 | src1
```

Examples:

```
.reg .32b %dst, %src0, %src1;

// or of 32-bit elements
or.32b  %dst, %src0, %src1;
```

sbfe

Extract and sign-extend bits.

Syntax:

```
sbfe.type dst, src0, src1, src2

  .type = { .32b }


  - dst is a 32-bit register
  - src0, src1, src2 can be 32-bit registers or immediates
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
offset  = src2[4:0]  // starting bit position
width   = umin(src1[5:0], bitwidth(.type)) // number of bits to extract
lastpos = umin(offset + width, bitwidth(.type)) // ignore bits beyond the MSB
width   = lastpos - offset // effective width

if (width == 0) {
  dst = 0
} else {
```

```
  dst = signed(src0 << (bitwidth(.type) - lastpos)) >> (bitwidth(.type) - width)
}
```

Notes:

Any bits specified in (offset + width) beyond 32 bits are ignored.

Examples:

```
.reg .32b %dst, %src0, %width, %offset;

// extract %width bits from %src0 at %offset
sbfe.32b  %dst, %src0, %width, %offset;
```

shr

Perform a logical right shift.

Syntax:

```
shr.type dst, src0, src1

  .type = { .16b, .32b, .64b }


  - dst is a register
  - src0 can be a register or an immediate
  - src1 can be a 32-bit register or an immediate

  * src0, src1 are interpreted as unsigned integers.
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
shr.type dst, src0, src1
```

```
  .type = { .16bx2 }


  - dst, src0 are 2x16-bit vector registers
  - src1 can be a 32-bit register or an immediate

  * src0, src1 are interpreted as unsigned integers.
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                  .type == .16bx2
shiftl = unsigned(src1[5:0])
shifth = unsigned(src1[21:16])
dst[31:16] = src0[31:16] >> shifth
dst[15:0]  = src0[15:0]  >> shiftl

                                     default

shift = unsigned(src1[5:0])
dst = src0 >> shift
```

Examples:

```
.reg .32b %dst, %src0, %src1;

// logical right shift of 32-bit element
shr.32b  %dst, %src0, %src1;
.reg .v2.16b %dst_packed, %src0_packed;
shr.16bx2  %dst_packed, %src0_packed, %src1;
```

shl

Perform a logical left shift.

Syntax:

```
shl.type dst, src0, src1

  .type = { .16b, .32b, .64b }


  - dst is a register
  - src0 can be a register or an immediate
  - src1 can be a 32-bit register or an immediate
```

```
  * src0, src1 are interpreted as unsigned integers.
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
shl.type dst, src0, src1

  .type = { .16bx2 }


  - dst, src0 are 2x16-bit vector registers
  - src1 can be a 32-bit register or an immediate

  * src0, src1 are interpreted as unsigned integers.
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                  .type == .16bx2
shiftl = unsigned(src1[5:0])
shifth = unsigned(src1[21:16])
dst[31:16] = src0[31:16] << shifth
dst[15:0]  = src0[15:0]  << shiftl

                                     default

shift = unsigned(src1[5:0])
dst = src0 << shift
```

Examples:

```
.reg .32b %dst, %src0, %src1;

// logical left shift of 32-bit element
shl.32b  %dst, %src0, %src1;
.reg .v2.16b %dst_packed, %src0_packed;
shl.16bx2  %dst_packed, %src0_packed, %src1;
```

shf

Perform a funnel shift operation.

Syntax:

```
shf.dir.type dst, src0, src1, src2

  .dir = { .l, .r }
  .type = { .32b }


  - dst is a 32-bit register
  - src0, src1, src2 can be 32-bit registers or immediates

  * src0, src1, src2 are interpreted as unsigned integers.
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
int64_t num = (src0 << 32) | src1
shift = unsigned(src2[4:0])
if (.dir == .l)
  dst = (num << shift) >> 32
else
  dst = num >> shift
```

Notes:

Funnel shift can be used to implement rotate by duplicating same variable in both `src0` and `src1`.

Examples:

```
.reg .32b %dst, %src0, %src1, %src2;

// funnel left shift
shf.l.32b  %dst, %src0, %src1, %src2;
// funnel right shift
shf.r.32b  %dst, %src0, %src1, %src2;
```

ubfe

Extract and zero-extend bits.

Syntax:

```
ubfe.type dst, src0, src1, src2

  .type = { .32b }


  - dst is a 32-bit register
  - src0, src1, src2 can be 32-bit registers or immediates
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
offset  = src2[4:0]  // starting bit position
width   = umin(src1[5:0], bitwidth(.type)) // number of bits to extract
lastpos = umin(offset + width, bitwidth(.type)) // ignore bits beyond the MSB
width   = lastpos - offset // effective width

if (width == 0) {
  dst = 0
} else {
  dst = unsigned(src0 << (bitwidth(.type) - lastpos)) >> (bitwidth(.type) - width)
}
```

Notes:

Any bits specified in (offset + width) beyond 32 bits are ignored.

Examples:

```
.reg .32b %dst, %src0, %width, %offset;

// extract %width bits from %src0 at %offset
ubfe.32b  %dst, %src0, %width, %offset;
```

xor

Compute bitwise XOR operation.

Syntax:

```
xor.type dst, src0, src1

  .type = { .16b, .32b, .64b }


  - dst is a register
  - src0, src1 can be registers or immediates
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
dst = src0 ^ src1
```

Examples:

```
.reg .32b %dst, %src0, %src1;

// xor of 32-bit elements
xor.32b  %dst, %src0, %src1;
```

Data Movement and Conversion

`addrcast`

Convert between specific and generic address spaces.

Convert from specific address space to generic address space:

```
addrcast.to.from dst, src0

  .to = { .generic }
  .from = { .private, .shared, .global, .const }


  - dst is a 64-bit register
  - src0 can be a register or memory variable
```

Restrictions

- bitwidth of `src0` must match the address size of `.from`
- address space of `src0` must match value of `.from`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Convert from generic address space to specific address space:

```
addrcast.to.from dst, src0

  .to = { .global, .private, .shared }
  .from = { .generic }


  - dst is a register
  - src0 is a 64-bit register
```

Restrictions

- bitwidth of `dst` must match the address size of `.to`

PISA Notes

- introduced in PISA version **0.1**

- requires PISA target **100** or compatible

Notes:

The representation of generic address values is implementation-defined. However, casting an address from a specific address space to the generic address space and then casting it back to the original address space is guaranteed to preserve the value. Pointer arithmetic on the generic address behaves the same as on the original specific address, as long as the result stays in bounds.

Note

Casting a generic address in the `.const` address space must use `addrcast.global.generic`.

Warning

Behavior is undefined if the address space of `src0` does not match the `.from` qualifier.

```
.reg .32b %paddr;
.reg .64b %gaddr;
.private .align 4 @A[8];

// convert generic address to private
addrcast.private.generic  %paddr, %gaddr;
// convert private address to generic
addrcast.generic.private  %gaddr, %paddr;
// convert private variable address to generic
addrcast.generic.private  %gaddr, @A;
```

`addrof`

Compute the non-generic address of a source variable.

Address of a memory variable:

```
addrof.type dst, src0

  .type = { .32b, .64b }


  - dst is a register
  - src0 is a memory variable
```

Restrictions

- bitwidth of `dst` must match the bitwidth of `.type`
- address size of storage space of `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Address of a function:

```
addrof.type dst, src0

  .type = { .64b }


  - dst is a 64-bit register
  - src0 is a function name
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
dst = &src0
```

Examples:

```
.import .function void @foo();
.reg .32b %dst;
.reg .64b %dst64;
.private @A[10];
.global @B[4];

// get 32-bit address of private variable
addrof.32b  %dst, @A;
// get 64-bit address of global variable
addrof.64b  %dst64, @B;
// get 64-bit address of function
addrof.64b  %dst64, @foo;
```

`extract.dynamic`

Extract an element from a vector register using a dynamic index.

Syntax:

```
extract.dynamic.vec.type dst, src0, index

  .vec = { .v2, .v3, .v4, .v5, .v6, .v7, .v8, .v16, .v32, .v64 }
  .type = { .32b }


  - dst, index are 32-bit registers
  - src0 is a 32-bit element vector register
```

Restrictions

- number of elements of `src0` must match value of `.vec`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
dst = src0[index]
```

Notes:

Warning

Behavior is undefined if the `index` is out of bounds for `src0`.

Examples:

```
.reg .v4.32b %src0;
.reg .32b %dst, %index;

// extract single element from %src0 at %index
extract.dynamic.v4.32b  %dst, %src0, %index;
```

`extract`

Extract elements from a vector register.

Extract scalar from vector register:

```
extract.index.type dst, src0

  .index = { .0, ..., .63 }
  .type = { .32b }


  - dst is a 32-bit register
  - src0 is a 32-bit element vector register
```

Restrictions

- value of `.index` must be less than the number of elements in `src0`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Extract subvector from vector register:

```
extract.index.vec.type dst, src0

  .index = { .0, ..., .63 }
  .vec = { .v2, .v3, .v4, .v5, .v6, .v7, .v8, .v16, .v32, .v64 }
  .type = { .32b }


  - dst, src0 are 32-bit element vector registers
```

Restrictions

- number of elements of `dst` must match value of `.vec`
- qualifier `.index` must be aligned to the number of elements in `dst`
- sum of value of `.index` and number of elements in `dst` must not exceed the number of elements in `src0`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
for (i = 0; i < element_count(dst); i++) {
  dst[i] = src0[index + i]
}
```

Notes:

It is illegal to specify swizzle for `dst` or `src0`.

Examples:

```
.reg .v4.32b %src0;
.reg .32b %dst;
.reg .v2.32b %dstVec;

// extract single element from %src0 at index 1
extract.1.32b  %dst, %src0;
// extract 2 elements from %src0 at index 2
extract.2.v2.32b  %dstVec, %src0;
```

`f2i`

Convert a floating-point value to an integer.

Syntax:

```
f2i.to.from<.rndmode> dst, src0

  .to = { .u8, .s8, .u16, .s16, .u32, .s32, .u64, .s64 }
  .from = { .bf, .hf, .f, .df }
  .rndmode = { .re, .rd, .ru, .rz, .rna }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` must match the bitwidth of `.to`
- bitwidth of `src0` must match the bitwidth of `.from`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
rnd = round(src0, .rndmode)
dst = type_convert(rnd, .to)
```

Notes:

If the resulting value (post-rounding) exceeds the range that can be represented by the integer type, the conversion result is clamped to the largest/smallest representable value. The conversion rules are summarized below, where **Imax** is the largest representable value by the integer type while **Imin** is the smallest representable value by the same type.

Floating-point to integer conversion (Signed)

src0	dst	Floating-point Exception
qNaN	0	Invalid Operation
sNaN	0	Invalid Operation
+inf	Imax	Invalid Operation
f > Imax	Imax	Invalid Operation
Imin <= f <= Imax	in-range, representable signed integer i	Inexact if rounding changes f
f < Imin	Imin	Invalid Operation
-inf	Imin	Invalid Operation

Floating-point to integer conversion (Unsigned)

src0	dst	Floating-point Exception
qNaN	0	Invalid Operation
sNaN	0	Invalid Operation
+inf	Imax	Invalid Operation
f > Imax	Imax	Invalid Operation
0 <= f <= Imax	in-range, representable unsigned integer i	Inexact if rounding changes f
-0	0	None
-1 < f < 0	0	Inexact if rounded result is zero Invalid Operation if result is -1
-fmax <= f <= -1	0	Invalid Operation
-inf	0	Invalid Operation

Examples:

```
.reg .32b %dst, %src0;

// convert single-precision to signed 32bit integer
f2i.s32.f  %dst, %src0;
// convert single-precision to unsigned 32-bit integer with round-up rounding
f2i.u32.f.ru  %dst, %src0;
```

`fext`

Extend a floating-point value to higher precision.

Syntax:

```
fext.to.from dst, src0

  .to = { .f, .df }
  .from = { .bf, .hf, .f }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` must match the bitwidth of `.to`
- bitwidth of `src0` must match the bitwidth of `.from`
- bitwidth of `.to` must be greater than the bitwidth of `.from`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                    default
dst = type_convert(src0, .to)
```

Notes:

Rules for extending special floating-point values

Source Floating Point Value (hf, bf, f)	Destination Floating Point Result (f, df)
-inf	-inf
-finite	-finite
-denorm	-finite
-0	-0
+0	+0
+denorm	+finite
+finite	+finite
+inf	+inf
NaN	NaN

Examples:

```
.reg .32b %dst;
.reg .16b %src0;

// convert half-precision float to single-precision float
fext.f.hf  %dst, %src0;
```

`frnd`

Round a floating-point number to the nearest integer.

Syntax:

```
frnd.rndmode<.ftz>.type dst, src0

  .rndmode = { .re, .rd, .ru, .rz, .rna }
  .type = { .bf, .hf, .f, .df }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
dst = round(src0, .rndmode)
```

Examples:

```
.reg .32b %dst, %src0;

// round-to-even of single-precision element
frnd.re.f  %dst, %src0;
```

ftrunc

Truncate a floating-point value to lower precision.

Syntax:

```
ftrunc.to.from<.rndmode><.ftz><.dsat> dst, src0

  .to = { .bf, .hf, .f }
  .from = { .f, .df }
  .rndmode = { .re, .rd, .ru, .rz, .rna }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` must match the bitwidth of `.to`
- bitwidth of `src0` must match the bitwidth of `.from`
- bitwidth of `.to` must be less than the bitwidth of `.from`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                      default
dst = truncate(src0, .to)
```

Notes:

Enabling saturation via `.dsat` qualifier clamps the final result to between the min/max float values representable by `.to` type. When `.dsat` is not specified:

- if source > maximum value of destination type, destination is set to +infinity
- if source < minimum value of destination type, destination is set to -infinity

Rules for truncating special floating-point values

Source Floating Point Value (f, df)	Destination Floating Point Result (hf, bf, f)
-inf	-inf
-finite	-finite/-denorm/-0
-denorm	-0
-0	-0
+0	+0
+denorm	+0
+finite	+finite/+denorm/+0
+inf	+inf
NaN	NaN

Examples:

```
.reg .32b %src0;
.reg .16b %dst;

// convert single-precision float to half-precision float
ftrunc.hf.f  %dst, %src0;
// convert single-precision float to half-precision float with rounding
ftrunc.hf.f.ru  %dst, %src0;
```

`ftrunc2`

Truncate and pack two floating-point values into a packed vector.

Syntax:

```
ftrunc2.to<.rndmode><.ftz><.dsat> dst, src0, src1

  .to = { .bfx2, .hfx2 }
  .rndmode = { .re, .rd, .ru, .rz, .rna }


  - dst is a 2x16-bit vector register
  - src0, src1 are 32-bit registers
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                        .to == .hfx2
ftrunc.hf.f dst[15:0], src0
ftrunc.hf.f dst[31:16], src1

                                        .to == .bfx2
ftrunc.bf.f dst[15:0], src0
ftrunc.bf.f dst[31:16], src1
```

Notes:

Enabling saturation via `.dsat` qualifier clamps the final result to between the min/max float values representable by `.to` type. When `.dsat` is not specified:

- if source > maximum value of destination type, destination is set to +infinity
- if source < minimum value of destination type, destination is set to -infinity

Examples:

```
.reg .32b %dst, %src0, %src1;

// truncate and pack 2 single-precision elements (dst must be .v2.16b)
.reg .v2.16b %dst_packed;
ftrunc2.hfx2  %dst_packed, %src0, %src1;
```

i2f

Convert an integer to a floating-point value.

Syntax:

```
i2f.to.from<.rndmode><.dsat> dst, src0

  .to = { .bf, .hf, .f, .df }
  .from = { .u8, .s8, .u16, .s16, .u32, .s32, .u64, .s64 }
  .rndmode = { .re, .rd, .ru, .rz, .rna }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` must match the bitwidth of `.to`
- bitwidth of `src0` must match the bitwidth of `.from`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
dst = type_convert(src0, .to)
```

Notes:

Enabling saturation via `.dsat` qualifier clamps the final result to between the min/max float values representable by `.to` type. When `.dsat` is not specified:

- if source > maximum value of destination type, destination is set to +infinity
- if source < minimum value of destination type, destination is set to -infinity

Examples:

```
.reg .16b %dst16, %src16;
.reg .32b %dst32, %src32;
.reg .64b %dst64, %src64;

// convert unsigned short to half-precision floating-point with implicit .re rounding
i2f.hf.u16  %dst16, %src16;
// convert signed short to single-precision floating-point with explicit rounding
i2f.f.s16.rd  %dst32, %src16;
// convert signed int to double-precision floating-point with explicit rounding and saturation
i2f.df.s32.rz.dsat  %dst64, %src32;
```

`insert.dynamic`

Insert an element into a vector using a dynamic index.

Syntax:

```
insert.dynamic.vec.type dst, src0, index

  .vec = { .v2, .v3, .v4, .v5, .v6, .v7, .v8, .v16, .v32, .v64 }
  .type = { .32b }
```

```
- dst is a 32-bit element vector register
- src0, index are 32-bit registers
```

Restrictions

- number of elements of `dst` must match value of `.vec`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
dst[index] = src0
```

Notes:

Warning

Behavior is undefined if the `index` is out of bounds for `dst`.

Examples:

```
.reg .v4.32b %dst;
.reg .32b %src0, %index;

// insert single element at %index
insert.dynamic.v4.32b  %dst, %src0, %index;
```

`insert`

Insert elements into a vector.

Insert scalar into vector register:

```
insert.index.type dst, src0

  .index = { .0, ..., .63 }
  .type = { .32b }
```

```
  - dst is a 32-bit element vector register
  - src0 is a 32-bit register
```

Restrictions

- value of `.index` must be less than the number of elements in `dst`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Insert subvector into vector register:

```
insert.index.vec.type dst, src0

  .index = { .0, ..., .63 }
  .vec = { .v2, .v3, .v4, .v5, .v6, .v7, .v8, .v16, .v32, .v64 }
  .type = { .32b }


  - dst, src0 are 32-bit element vector registers
```

Restrictions

- number of elements of `src0` must match value of `.vec`
- qualifier `.index` must be aligned to the number of elements in `src0`
- sum of value of `.index` and number of elements in `src0` must not exceed the number of elements in `dst`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
for (i = 0; i < element_count(src); i++) {
  dst[index + i] = src[i]
}
```

Notes:

It is illegal to specify swizzle for `dst` or `src0`.

Examples:

```
.reg .v4.32b %dst;
.reg .32b %src0;
.reg .v2.32b %srcVec;

// insert single element from %src0 at index 1
insert.1.32b  %dst, %src0;
// insert 2 elements from %srcVec at index 2
insert.2.v2.32b  %dst, %srcVec;
```

isaddr

Check if a generic address belongs to a specific address space.

Check if generic address belongs to private/shared/global address space:

```
isaddr.addrspace dst, src0

  .addrspace = { .private, .shared, .global }


  - dst is a 32-bit register
  - src0 is a 64-bit register

  * src0 is interpreted as generic address.
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
if get_address_space(src0) == .addrspace
   dst = ~0
else
   dst = 0
```

Examples:

```
.reg .32b %dst, %src0;
.reg .64b %gen;
```

```
// convert private address %src0 to generic address %gen
addrcast.generic.private %gen, %src0;
isaddr.private   %dst, %gen; // %r = ~0
isaddr.shared    %dst, %gen; // %r = 0
```

mov

Move data between registers.

Scalar to scalar copy:

```
mov.type dst, src0

  .type = { .8b, .16b, .32b, .64b, .128b }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Scalar to vector copy:

```
mov.type dst.swizzle, src0

  .type = { .8b, .16b, .32b, .64b, .128b }


  - dst is a vector register
  - src0 is a register
```

Restrictions

- total width of `dst` must match the bitwidth of `.type`
- bitwidth of `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Vector to scalar copy:

```
mov.type dst, src0.swizzle

  .type = { .8b, .16b, .32b, .64b, .128b }


  - dst is a register
  - src0 is a vector register
```

Restrictions

- bitwidth of `dst` must match the bitwidth of `.type`
- total width of `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Vector to vector copy:

```
mov.type dst.swizzle, src0.swizzle

  .type = { .8b, .16b, .32b, .64b, .128b }


  - dst, src0 are vector registers
```

Restrictions

- total width of `dst` must match the bitwidth of `.type`
- total width of `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
// return the start/end bit for reg.swizzle
getSwizzleBits(reg, swizzle) {
   w = bitwidth(reg)
   switch (swizzle) {
     case .x:     return {0, w - 1}
     case .y:     return {w, 2 * w - 1}
     case .z:     return {2 * w, 3 * w - 1}
     case .w:     return {3 * w, 4 * w - 1}
     case .xy:    return {0, 2 * w - 1}
     case .zw:    return {2 * w, 4 * w - 1}
     case .xyzw:  return {0, 4 * w - 1}
   }
}
```

dst is scalar, src0 is scalar

```
dst = src0
```

dst is vector, src0 is scalar

```
[D_S, D_E] = getSwizzleBits(dst, dst_swizzle)
dst[D_S, D_E] = src0
```

dst is scalar, src0 is vector

```
[S_S, S_E] = getSwizzleBits(src, src_swizzle)
dst = src0[S_S, S_E]
```

dst is vector, src0 is vector

```
[S_S, S_E] = getSwizzleBits(src, src_swizzle)
[D_S, D_E] = getSwizzleBits(dst, dst_swizzle)
dst[D_S, D_E] = src0[S_S, S_E]
```

Notes:

Number of bits being copied is specified by `.type`, and must match the bitwidth of a scalar variable and that of a vector variable after swizzle.

Examples:

```
.reg .32b %dst, %src0;
.reg .v2.32b %dstVec, %src0Vec;

// scalar to scalar move
mov.32b  %dst, %src0;
// scalar to vector move
```

```
mov.32b  %dstVec.x, %src0;
// vector to scalar move
mov.32b  %dst, %src0Vec.y;
// vector to vector move
mov.64b  %dstVec.xy, %src0Vec.xy;
```

pf2i

Convert packed floats to packed integers.

Syntax:

```
pf2i.to.from<.rndmode> dst, src0

  .to = { .u16x2, .s16x2 }
  .from = { .bfx2, .hfx2 }
  .rndmode = { .re, .rd, .ru, .rz, .rna }


  - dst, src0 are 2x16-bit vector registers
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
dst[15:0] = type_convert(src0[15:0], element_type(.to))
dst[31:16] = type_convert(src0[31:16], element_type(.to))
```

Notes:

Each element independently converted following the rules specified in f2i instruction.

`src0` holds a packed floating-point value (`.hfx2` or `.bfx2`) and `dst` holds a packed integer value (`.u16x2` or `.s16x2`). Both operands must be declared as `.v2.16b` registers (see packed types).

Examples:

```
.reg .v2.16b %dst_packed, %src_packed;

// conversion using .v2.16b register declarations
pf2i.s16x2.hfx2  %dst_packed, %src_packed;
```

`sext`

Sign-extend an integer to a larger type.

Syntax:

```
sext.to.from dst, src0

  .to = { .16b, .32b, .64b }
  .from = { .8b, .16b, .32b }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` must match the bitwidth of `.to`
- bitwidth of `src0` must match the bitwidth of `.from`
- bitwidth of `.to` must be greater than the bitwidth of `.from`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
dst = sign_extend(src0)
```

Examples:

```
.reg .32b %dst;
.reg .16b %src0;

// sign-extend 16-bit element
sext.32b.16b  %dst, %src0;
```

`trunc`

Truncate an integer to a smaller type.

Syntax:

```
trunc.to.from dst, src0

  .to = { .8b, .16b, .32b }
  .from = { .16b, .32b, .64b }
```

```
  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` must match the bitwidth of `.to`
- bitwidth of `src0` must match the bitwidth of `.from`
- bitwidth of `.to` must be less than the bitwidth of `.from`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
dst = truncate(src0, .to)
```

Examples:

```
.reg .32b %src0;
.reg .16b %dst;

// truncate 32-bit element
trunc.16b.32b  %dst, %src0;
```

`zext`

Zero-extend an integer to a larger type.

Syntax:

```
zext.to.from dst, src0

  .to = { .16b, .32b, .64b }
  .from = { .8b, .16b, .32b }


  - dst is a register
  - src0 can be a register or an immediate
```

Restrictions

- bitwidth of `dst` must match the bitwidth of `.to`
- bitwidth of `src0` must match the bitwidth of `.from`
- bitwidth of `.to` must be greater than the bitwidth of `.from`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
dst = zero_extend(src0)
```

Examples:

```
.reg .32b %dst;
.reg .16b %src0;

// zero-extend 16-bit element
zext.32b.16b  %dst, %src0;
```

Comparison and Selection

scmp

Compare two signed integer values.

predicate destination:

```
scmp.cop.type  pred, src0, src1

  .cop = { .eq, .ne, .gt, .ge, .lt, .le }
  .type = { .16b, .32b, .64b }


  - pred is a predicate register
  - src0, src1 can be registers or immediates

  * src0, src1 are interpreted as signed integers.
```

Restrictions

- bitwidth of `src0` and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

register destination:

```
scmp.cop.type  dst, src0, src1

  .cop = { .eq, .ne, .gt, .ge, .lt, .le }
  .type = { .16b, .32b, .64b }


  - dst is a 32-bit register
  - src0, src1 can be registers or immediates

  * src0, src1 are interpreted as signed integers
```

Restrictions

- bitwidth of `src0` and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

register destination

```
dst[31:0] = (src0 .cop src1) ? ~0 : 0
```

predicate destination

```
pred = (src0 .cop src1) ? 1 : 0
```

Notes:

- `scmp.eq` is functionally equivalent to `ucmp.eq`.

- `scmp.ne` is functionally equivalent to `ucmp.ne`.

Examples:

```
.reg .16b %src0_16b, %src1_16b;
.reg .32b %dst, %src0, %src1, %src2;
.pred %pin, %pout;

// signed comparison of 32-bit elements (predicate result)
scmp.eq.32b  %pout, %src0, %src1;
// signed comparison of 32-bit elements (register result)
scmp.eq.32b  %dst, %src0, %src1;
// signed comparison of 16-bit elements (register result: %dst contains either 0xFFFFFFFF or 0x00000000)
scmp.ge.16b  %dst, %src0_16b, %src1_16b;
```

`ucmp`

Compare two unsigned integer values.

Unsigned comparison with predicate destination:

```
ucmp.cop.type   pred, src0, src1

  .cop = { .eq, .ne, .gt, .ge, .lt, .le }
  .type = { .16b, .32b, .64b }


  - pred is a predicate register
  - src0, src1 can be registers or immediates

  * src0, src1 are interpreted as unsigned integers.
```

Restrictions

- bitwidth of `src0` and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Unsigned comparison with register destination:

```
ucmp.cop.type   dst, src0, src1

  .cop = { .eq, .ne, .gt, .ge, .lt, .le }
  .type = { .16b, .32b, .64b }
```

```
  - dst is a 32-bit register
  - src0, src1 can be registers or immediates

  * src0, src1 are interpreted as unsigned integers.
```

Restrictions

- bitwidth of `src0` and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
ucmp.cop.type   dst, src0, src1

  .cop = { .eq, .ne, .gt, .ge, .lt, .le }
  .type = { .16bx2 }


  - dst, src0, src1 are 2x16-bit vector registers

  * src0, src1 are interpreted as unsigned integers.
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
                                    .type == .16bx2
dst[15:0] = (src0[15:0] .cop src1[15:0]) ? ~0 : 0
dst[31:16] = (src0[31:16] .cop src1[31:16]) ? ~0 : 0

                                    register destination
dst[31:0] = (src0 .cop src1) ? ~0 : 0

                                    predicate destination
pred = (src0 .cop src1) ? 1 : 0
```

Notes:

- `ucmp.eq` is functionally equivalent to `scmp.eq`.
- `ucmp.ne` is functionally equivalent to `scmp.ne`.

Examples:

```
.reg .16b %src0_16b, %src1_16b;
.reg .32b %dst, %src0, %src1, %src2;
.pred %pin, %pout;

// unsigned comparison of 32-bit elements (predicate result)
ucmp.eq.32b  %pout, %src0, %src1;
// unsigned comparison of 32-bit elements (register result)
ucmp.eq.32b  %dst, %src0, %src1;
// unsigned comparison of 16-bit elements (register result: %dst contains either 0xFFFFFFFF or 0x00000000)
ucmp.ge.16b  %dst, %src0_16b, %src1_16b;
// element-wise unsigned comparison of 2x16-bit packed values (dst is bitmask, src operands are .v2.16b)
.reg .v2.16b %pdst, %psrc0, %psrc1;
ucmp.le.16bx2  %pdst, %psrc0, %psrc1;
```

`fcmp`

Compare two floating-point values.

Floating-point comparison with predicate destination:

```
fcmp.cop<.ftz>.type   pred, src0, src1

  .cop = { .eq, .ne, .gt, .ge, .lt, .le }
  .type = { .bf, .hf, .f, .df }


  - pred is a predicate register
  - src0, src1 can be registers or immediates
```

Restrictions

- bitwidth of `src0` and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Floating-point comparison with register destination:

```
fcmp.cop<.ftz>.type   dst, src0, src1
```

```
  .cop = { .eq, .ne, .gt, .ge, .lt, .le }
  .type = { .bf, .hf, .f, .df }


  - dst is a 32-bit register
  - src0, src1 can be registers or immediates
```

Restrictions

- bitwidth of `src0` and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
fcmp.cop<.ftz>.type   dst, src0, src1

  .cop = { .eq, .ne, .gt, .ge, .lt, .le }
  .type = { .bfx2, .hfx2 }


  - dst, src0, src1 are 2x16-bit vector registers
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

`.type == .bfx2 || .type == .hfx2`

```
dst[15:0] = (src0[15:0] .cop src1[15:0]) ? ~0 : 0
dst[31:16] = (src0[31:16] .cop src1[31:16]) ? ~0 : 0
```

register destination

```
dst[31:0] = (src0 .cop src1) ? ~0 : 0
```

predicate destination

```
pred = (src0 .cop src1) ? 1 : 0
```

Examples:

```
.reg .16b %src0_16b, %src1_16b;
.reg .32b %dst, %src0, %src1, %src2;
.pred %pin, %pout;

// comparison of single-precision elements (predicate result)
fcmp.eq.f  %pout, %src0, %src1;
// comparison of single-precision elements (register result)
fcmp.eq.f  %dst, %src0, %src1;
// comparison of half-precision elements (register result: %dst contains either 0xFFFFFFFF or 0x00000000)
fcmp.ge.hf  %dst, %src0_16b, %src1_16b;
// element-wise comparison of 2 x half-float packed values
// src operands are .v2.16b packed FP; dst is an integer bitmask (0xFFFF or 0x0000 per lane)
.reg .v2.16b %pdst, %psrc0, %psrc1;
fcmp.le.hfx2  %pdst, %psrc0, %psrc1;
```

`sel`

Select one of two sources based on value.

Select based on register value:

```
sel.type   dst, src0, src1, src2

  .type = { .16b, .32b, .64b }


  - dst, src2 are registers
  - src0, src1 can be registers or immediates
```

Restrictions

- bitwidth of `dst`, `src0`, `src1`, and `src2` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Select based on predicate:

```
sel.type   dst, src0, src1, <!>pred

  .type = { .16b, .32b, .64b }


  - dst is a register
  - src0, src1 can be registers or immediates
  - pred is a predicate register
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

register selector

```
dst = (src2 != 0) ? src0 : src1
```

predicate selector

```
if ('!' is set)
   dst = (!pred == 1) ? src0 : src1
else
   dst = (pred == 1) ? src0 : src1
```

Notes:

The register-selector form performs a bitwise integer test on `src2`. For IEEE 754-compliant floating-point zero-checking, use `fcmp.ne` to produce a predicate and select using the predicate-selector form instead.

Examples:

```
.reg .32b %dst, %src0, %src1, %src2;
.pred %pin;

// select one of 32-bit elements (register selector)
sel.32b  %dst, %src0, %src1, %src2;
// select one of 32-bit elements (predicate selector)
sel.32b  %dst, %src0, %src1, %pin;
// select one of 32-bit elements (inverted predicate)
sel.32b  %dst, %src0, %src1, !%pin;
```

Control Flow

`call`

Call a function.

Unconditional function call:

```
call retval, target (<arg1> <, arg2> ... <, argn>)

  - retval can be a register or void
  - target can be a 64-bit register or a function name
```

Restrictions

- type of `retval` must match `target` function return type

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Conditional function call:

```
call.cond <!>pred, retval, target (<arg1> <, arg2> ... <, argn>)

  - pred is a predicate register
  - retval can be a register or void
  - target can be a 64-bit register or a function name
```

Restrictions

- type of `retval` must match `target` function return type

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

unconditional call

```
$caller_PC = $PC_next; // instruction after call
// implementation-defined mechanism to pass arguments to callee
$PC = target
```

conditional call

```
xfer_control = pred
if ('!' is set)
```

```
  xfer_control = !xfer_control
if (xfer_control)
  $caller_PC = $PC_next; // instruction after call
  // implementation-defined mechanism to pass arguments to callee
  $PC = target
else
  nop // fall-through
```

Notes:

<arg1>...<argn> are actual function arguments passed by the caller. Arguments must be register variables, immediate values are not allowed.

When the callee executes the `return` instruction, control flow will be transfered back to the next instruction following the `call` in the caller. If `retval` is not void, the return value specified in the callee's return instruction will also be written to `retval`.

It is undefined behavior if contents of the register do not point to function code.

Examples:

```
.reg .32b %src0, %retval;
.reg .64b %fptr;
.pred %p;

// call a void function with no arguments
call void, @func0();
// call function returning 32-bit value only if %p is true
call.cond %p, %retval, @func1();
// call function with arguments only if %p is false
call.cond !%p, %retval, @func2(%src0);

// call a function through a function pointer
addrof.64b %fptr, @func0;
call void, %fptr();
```

`goto`

Branch to a label.

Unconditional branch:

```
goto label

  - label is a named location within current function
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Conditional branch:

```
goto.cond <!>pred, label

  - pred is a predicate register
  - label is a named location within current function
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

unconditional jump

```
$PC = label;
```

conditional jump

```
xfer_control = pred
if ('!' is set)
  xfer_control = !xfer_control
if (xfer_control)
  $PC = label
else
  nop // fall-through
```

Examples:

```
.pred %p;

// jump to label1 within this function
goto label1;
label1:
// jump to label2 within this function only if %p is true
goto.cond %p, label2;
// jump to label2 within this function only if %p is false
goto.cond !%p, label2;
label2:
```

`return`

Return from a function or end a work-item's execution in a kernel.

Unconditional return:

```
return <retval>

  - retval can be a register or an immediate
```

Restrictions

- type of `retval` must match current function return type

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Conditional return:

```
return.cond <!>pred, retval

  - pred is a predicate register
  - retval can be a register or an immediate
```

Restrictions

- type of `retval` must match current function return type

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

unconditional return

```
$PC = $caller_PC; // instruction after call()
```

conditional return

```
xfer_control = pred
if ('!' is set)
  xfer_control = !xfer_control
if (xfer_control)
  $PC = $caller_PC; // instruction after call()
else
  nop // fall-through
```

Notes:

If `return` is executed in a kernel, it ends this work-item's execution.

`retval` specifies the function return value. It must be omitted for kernels and void functions. In non-void functions, `return` must be the last instruction.

Examples:

```
.pred %p;

// unconditional return (ends work-item in a kernel; returns from .func)
return;
// return only if %p is true
return.cond %p;
// return only if %p is false
return.cond !%p;
```

Sub-group Communication

`fred`

Perform a floating-point reduction operation across participating work-items within a sub-group.

Syntax:

```
fred.op<.nanp>.type dst, src0, src1

  .op = { .min, .max, .absmax }
  .type = { .bf, .hf, .f }


  - dst, src0 are registers
  - src1 can be a 32-bit register or an immediate
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

packed:

```
fred.op<.nanp>.type dst, src0, src1

  .op = { .min, .max, .absmax }
  .type = { .bfx2, .hfx2 }


  - dst, src0 are 2x16-bit vector registers
  - src1 can be a 32-bit register or an immediate
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
allParticipatingLanes = src1 & %activemask
if ((1 << %laneid) & allParticipatingLanes == 0)
  return

reduce(.op, src0, allParticipatingLanes)

reduce(Op, Src, ParticipatingLanes) {
  switch Op {
    case .min:
      dst = fmin<.nanp>(Src of work-items in ParticipatingLanes)
      break
    case .max:
      dst = fmax<.nanp>(Src of work-items in ParticipatingLanes)
      break
    case .absmax:
      dst = fmax<.nanp>(fabs(Src of work-items in ParticipatingLanes))
      break
  }
}
```

Notes:

`src1` is a sub-group-uniform bit mask that selects the participating work-items, with each bit position corresponding to a work-item's %laneid. Non-participating work-items do not contribute to the reduction operation, and do not update their `dst` values.

By default, NaN inputs are suppressed during the reduction: a NaN contributed by a participating work-item is ignored unless every participating work-item contributes a NaN, in which case a quiet NaN is produced.

When `.nanp` is specified, NaN inputs propagate. If any participating work-item contributes a NaN of any type, the reduction result is a quiet NaN.

Examples:

```
.reg .32b %dst, %src0;

// maximum of single-float %src0 across all active work-items
fred.max.f   %dst, %src0, 0xffffffff;
// minimum of single-float %src0 across active work-items with laneid 16-31
fred.min.f   %dst, %src0, 0xffff0000;
// maximum of single-float with NaN propagation
fred.max.nanp.f   %dst, %src0, 0xffffffff;
// element-wise maximum of packed bfloat vector across active work-items with laneid 0-15
.reg .v2.16b %dst_packed, %src0_packed;
fred.max.bfx2  %dst_packed, %src0_packed, 0xffff;
```

ired

Perform an integer reduction operation across participating work-items within a sub-group.

Syntax:

```
ired.op.type dst, src0, src1

  .op = { .sum, .smin, .smax, .umin, .umax, .and, .or, .xor, .absmax }
  .type = { .16b, .32b }


  - dst, src0 are registers
  - src1 can be a 32-bit register or an immediate
```

Restrictions

- bitwidth of `dst` and `src0` must match the bitwidth of `.type`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
allParticipatingLanes = src1 & %activemask
if ((1 << %laneid) & allParticipatingLanes == 0)
  return

reduce(.op, src0, allParticipatingLanes)

reduce(Op, Src, ParticipatingLanes) {
  switch Op {
    case .sum:
      dst = sum(Src of work-items in ParticipatingLanes)
```

```
      break
    case .smin:
      dst = smin(Src of work-items in ParticipatingLanes)
      break
    case .smax:
      dst = smax(Src of work-items in ParticipatingLanes)
      break
    case .umin:
      dst = umin(Src of work-items in ParticipatingLanes)
      break
    case .umax:
      dst = umax(Src of work-items in ParticipatingLanes)
      break
    case .and:
      dst = and(Src of work-items in ParticipatingLanes)
      break
    case .or:
      dst = or(Src of work-items in ParticipatingLanes)
      break
    case .xor:
      dst = xor(Src of work-items in ParticipatingLanes)
      break
    case .absmax:
      dst = umax(iabs(Src of work-items in ParticipatingLanes))
      break
  }
}
```

Notes:

`src1` is a sub-group-uniform bit mask that selects the participating work-items, with each bit position corresponding to a work-item's %laneid. Non-participating work-items do not contribute to the reduction operation, and do not update their `dst` values.

Important

For the `.absmax` operation, the most negative value representable in the specified `.type` is interpreted as the largest positive value; for example, when `.type == .32b`, `0x80000000` is treated as the maximum value.

Examples:

```
.reg .32b %dst, %src0;

// sum of %src0 across all active work-items
ired.sum.32b   %dst, %src0, 0xffffffff;
// signed max of %src0 across active work-items with laneid 0-15
ired.smax.32b  %dst, %src0, 0xffff;
```

redfirstidx

Get the index of the first active work-item within a sub-group.

Syntax:

```
redfirstidx.type dst, src0

  .type = { .32b }


  - dst is a 32-bit register
  - src0 can be a 32-bit register or an immediate
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
participatingLanes = src0 & %activemask

dst = 0
while ((participatingLanes & 1) == 0) {
  dst++
  participatingLanes >>= 1
}
```

Notes:

`src0` is a sub-group-uniform bit mask that selects the participating work-items, with each bit position corresponding to a work-item's %laneid.

Examples:

```
.reg .32b %dst;

// index of the first active work-item across active work-items with laneid 0-15
redfirstidx.32b   %dst, 0xffff;
```

shfl

Shuffle data across work-items within a sub-group.

Syntax:

```
shfl.op<.sg>.type dst, src0, src1, src2

  .op = { .up, .dn, .xor, .idx }
  .type = { .32b }


  - dst, src0 are 32-bit registers
  - src1, src2 can be 32-bit registers or immediates
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
participatingLanes = src2 & %activemask
if ((1 << %laneid) & participatingLanes == 0)
  return

laneOffset = src1[4:0]
if (.sg) {
  cval = src1[12:8]
  segmask = src1[20:16]
} else {
  cval = (.op == .up) ? 0x0 : 0x1F
  segmask = 0x0
}

clampLane = (%laneid & segmask) | (cval & ~segmask)
minLane = (%laneid & segmask)

switch .op{
  case .up:
    srcLane = %laneid - laneOffset
    break
  case .dn:
    srcLane = %laneid + laneOffset
    break
  case .xor:
    srcLane = %laneid ^ laneOffset
    break
  case .idx:
    srcLane = minLane | (laneOffset & ~segmask)
    break
}
inBounds = (.op == .up) ? (srcLane >= clampLane) : (srcLane <= clampLane)
if (!inBounds || !participatingLanes[srcLane]) {
  dst = src0[%laneid]
} else {
  dst = src0[srcLane]
}
```

Notes:

`src1` specifies lane offset in bits [4:0], clamp value in bits [12:8], and segmask in bits [20:16]. Each `c` bit in the table below can be 0 or 1, and is used to encode the clamp value.

Segmask and Clamp Value Encoding

segmask	sub-group width	cval
11110	2	1111c
11100	4	111cc
11000	8	11ccc
10000	16	1cccc
00000	32	ccccc

The qualifier `.sg` is an optional qualifier that indicates that this is a sub-group shuffle operation. If not specified, the instruction defaults to a full-group shuffle operation - equivalent to having 1 sub-group inclusive of all work-items - and bits [20:8] of `src1` are ignored.

`src2` is a sub-group-uniform bit mask that selects the participating work-items, with each bit position corresponding to a work-item's %laneid. Non-participating work-items do not contribute to the shuffle operation, and do not update their `dst` values.

Important

Out-of-bounds work-item access copies its own `src0` to `dst`.

Examples:

```
.reg .32b %dst, %src0, %src1;

// shuffle data across all active work-items with idx op
shfl.idx.32b   %dst, %src0, %src1, 0xffffffff;

// shuffle data up across active work-items with laneid 16-31
shfl.up.32b   %dst, %src0, 1, 0xffff;


.reg .32b %value, %sum, %tmp;

// prefix sum across sub-group of size 32:
// sum: N-1
shfl.up.32b   %tmp, %value, 1, 0xffffffff;
// sum: sum(N, N-1)
iadd.32b   %sum, %value, %tmp;
// tmp: N-2, N-3
shfl.up.32b   %tmp, %sum, 2, 0xffffffff;
// sum: sum(N, N-1, N-2, N-3)
iadd.32b   %sum, %sum, %tmp;
// tmp: sum(N-4, N-5, N-6, N-7)
shfl.up.32b   %tmp, %sum, 4, 0xffffffff;
// sum: sum(N, N-1, N-2, ..., N-7)
```

```
iadd.32b   %sum, %sum, %tmp;
// tmp: sum(N-8, N-9, ..., N-15)
shfl.up.32b   %tmp, %sum, 8, 0xffffffff;
// sum: sum(N, N-1, N-2, ..., N-15)
iadd.32b   %sum, %sum, %tmp;
// tmp: sum(N-16, N-17, ..., N-31)
shfl.up.32b   %tmp, %sum, 16, 0xffffffff;
// sum: sum(N, N-1, N-2, ..., N-31)
iadd.32b   %sum, %sum, %tmp;
// sum: sum(N-1, N-2, ..., N-31)
isub.32b   %sum, %sum, %value;

.reg .32b %flag, %expr, %r1, %r2, %r3, %r4, %r5, %result;
.pred %p;

// check if %expr is the same for every work-item in the sub-group
// Shuffle from immediate lower neighbor
shfl.up.32b   %r1, %expr, 1, 0xffffffff;
ucmp.ne.32b   %p, %r1, %expr;
sel.32b  %flag, 0x0, 0x1, %p;
// Now perform reduction
shfl.xor.32b   %r1, %flag, 16, 0xffffffff;
iadd.32b   %r1, %r1, %flag;
shfl.xor.32b   %r2, %r1, 8, 0xffffffff;
iadd.32b   %r2, %r2, %r1;
shfl.xor.32b  %r3, %r2, 4, 0xffffffff;
iadd.32b   %r3, %r3, %r2;
shfl.xor.32b  %r4, %r3, 2, 0xffffffff;
iadd.32b   %r4, %r4, %r3;
shfl.xor.32b  %r5, %r4, 1, 0xffffffff;
iadd.32b   %result, %r5, %r4;

ucmp.eq.32b   %p, %result, 0;
goto.cond   %p, all_equal;
goto.cond   !%p, not_all_equal;
not_all_equal:
all_equal:

// Sub-group shuffle data down by 2 with sub-thread width of 8
// (laneoffset = 5b00010, cval = 5b00000, segmask=5b11000)
// %a |31|30|29|28|27|26|25|24|23|22|21|20|19|18|17|16|15|..
// %r |  |  |31|30|29|28|27|26|  |  |23|22|21|20|19|18|  |..

.reg .32b %r, %a;
shfl.dn.sg.32b %r, %a, 0x00180002, 0xffffffff;

// Sub-group shuffle data down by 2 with sub-thread width of 8
// (laneoffset = 5b00010, cval = 5b11101, segmask=5b11000)
// %a |31|30|29|28|27|26|25|24|23|22|21|20|19|18|17|16|15|..
// %r |  |  |  |  |29|28|27|26|  |  |  |21|20|19|18|  |..

.reg .32b %r, %a;
shfl.dn.sg.32b %r, %a, 0x00181d02, 0xffffffff;
```

Memory and Atomics

`fatom`

Perform a floating-point atomic operation.

Syntax:

```
fatom.addrspace<.memorder><.memscope><.cc>.op.type dst, [addr], src0 <, src1>

  .addrspace = { .global, .generic, .shared }
  .memorder = { .relaxed, .release, .acquire, .acq_rel }
  .memscope = { .workgroup, .gpu, .system }
  .cc = { .uc, .L2wb, .L3wb }
  .op = { .add, .sub, .min, .max, .cas }
  .type = { .bf, .hf, .f, .df }


  - addr is an address operand, in the form MemRR or MemRI
  - dst, src0, src1 are registers
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`
- operand `src1` is defined only when qualifier `.op` is `.cas`
- qualifier `.cc` must not be specified when qualifier `.addrspace` is `.shared`
- qualifier `.memscope` can be specified only when qualifier `.memorder` != `.relaxed`
- qualifier `.type` cannot be `.df` unless qualifier `.addrspace` is `.global` and qualifier `.op` is one of [`.add`, `.sub`]

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
atomic {
  old = *addr
  switch (op) {
    case cas:
      new = (old == src0) ? src1 : old
      break
    case add:
      new = old + src0
      break
    case sub:
      new = old - src0
      break
```

```
    case min:
      new = fmin(old, src0)
      break
    case max:
      new = fmax(old, src0)
      break
  }
  *addr = new
  dst = old
}
```

Notes:

- memory access requirements must be observed.
- `.generic` address must point to either `.global` or `.shared` address space.
- atomic cache controls may be utilized.

Supported combinations for floating-point atomic operations

.addrspace	.op	.type
.global	.min, .max, .cas	.hf, .bf, .f
.global	.add, .sub	.hf, .bf, .f, .df
.shared	.min, .max, .cas, .add, .sub	.hf, .bf, .f
.generic	.min, .max, .cas, .add, .sub	.hf, .bf, .f

Examples:

```
.reg .32b %dst, %src0, %addr;

// atomic addition of single-precision floating point and memory value
fatom.shared.add.f  %dst, [%addr], %src0;
```

`iatom`

Perform an integer atomic operation.

Syntax:

```
iatom.addrspace<.memorder><.memscope><.cc>.op.type dst, [addr] <, src0> <, src1>

  .addrspace = { .global, .shared, .generic }
  .memorder = { .relaxed, .release, .acquire, .acq_rel }
  .memscope = { .workgroup, .gpu, .system }
  .cc = { .uc, .L2wb, .L3wb }
  .op = { .add, .sub, .smin, .smax, .umin, .umax, .and, .or, .xor, .xchg, .cas, .inc,
          .dec, .incwrap, .decwrap }
  .type = { .16b, .32b, .64b, .128b }
```

```
- addr is an address operand, in the form MemRR or MemRI
- dst, src0, src1 are registers
```

Restrictions

- bitwidth of `dst`, `src0`, and `src1` must match the bitwidth of `.type`
- operand `src0` is not defined when qualifier `.op` is one of [`.inc`, `.dec`]
- operand `src1` is defined only when qualifier `.op` is `.cas`
- qualifier `.type` cannot be `.16b` when qualifier `.op` is one of [`.incwrap`, `.decwrap`]
- qualifier `.type` cannot be `.128b` unless qualifier `.op` is one of [`.xchg`, `.cas`]
- qualifier `.type` cannot be `.64b` unless qualifier `.op` is `.cas` or qualifier `.addrspace` is `.global`
- qualifier `.cc` must not be specified when qualifier `.addrspace` is `.shared`
- qualifier `.memscope` can be specified only when qualifier `.memorder` != `.relaxed`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

```
atomic {
  old = *addr
  switch (op) {
    case cas:
      new = (old == src0) ? src1 : old
      break
    case add:
      new = old + src0
      break
    case sub:
      new = old - src0
      break
    case inc:
      new = old + 1
      break
    case dec:
      new = old - 1
      break
    case incwrap:
      new = (old >= src0) ? 0 : (old + 1); // unsigned
      break
    case decwrap:
      new = ((old == 0) || (old > src0)) ? src0 : (old - 1); // unsigned
      break
    case smin:
      new = smin(old, src0)
```

```
      break
    case smax:
      new = smax(old, src0)
      break
    case umin:
      new = umin(old, src0)
      break
    case umax:
      new = umax(old, src0)
      break
    case and:
      new = old & src0
      break
    case or:
      new = old | src0
      break
    case xor:
      new = old ^ src0
      break
    case xchg:
      new = src0
      break
  }
  *addr = new
  dst = old
}
```

Notes:

- memory access requirements must be observed.
- `.generic` address must point to either `.global` or `.shared` address space.
- atomic cache controls may be utilized.

Supported combinations for integer atomic operations

.addrspace	.op	.type
.global	.add, .sub, .smin, .smax, .umin, .umax, .and, .or, .xor, .xchg, .cas, .inc, .dec	.16b, .32b, .64b
.global	.incwrap, .decwrap	.32b, .64b
.shared, .generic	.add, .sub, .smin, .smax, .umin, .umax, .and, .or, .xor, .xchg, .inc, .dec	.16b, .32b
.shared, .generic	.cas	.16b, .32b, .64b
.shared, .generic	.incwrap, .decwrap	.32b
.global, .shared, .generic	.cas, .xchg	.128b

Examples:

```
.reg .32b %dst, %src0, %addr;

// atomic addition of 32-bit integer and memory value
iatom.shared.add.32b  %dst, [%addr], %src0;
```

ld

Load data from memory.

Load data from memory (non-atomic):

```
ld<.addrspace><.memorder><.cc><.vec>.type dst, [addr]

  .addrspace = { .global, .generic, .shared, .private, .const }
  .memorder = { .weak }
  .cc = { .L1uc.L2uc.L3uc, .L1uc.L2uc.L3c, .L1uc.L2c.L3uc, .L1uc.L2c.L3c,
          .L1c.L2uc.L3uc, .L1c.L2uc.L3c, .L1c.L2c.L3uc, .L1c.L2c.L3c,
          .L1s.L2uc.L3uc, .L1s.L2uc.L3c, .L1s.L2c.L3uc, .L1s.L2c.L3c,
          .ri }
  .vec = { .v2, .v3, .v4, .v8 }
  .type = { .8b, .16b, .32b, .64b }


  - addr is an address operand, in the form MemRR or MemRI
  - dst is a register
```

Restrictions

- bitwidth of `dst` must match the bitwidth of `.type`
- number of elements of `dst` must match value of `.vec`
- qualifier `.vec` cannot be `.v3` unless qualifier `.type` is one of [`.32b`, `.64b`]
- qualifier `.vec` cannot be `.v8` unless qualifier `.type` is `.32b`
- qualifier `.cc` must not be specified when qualifier `.addrspace` is `.shared`
- qualifier `.memorder` must not be specified when qualifier `.addrspace` is one of [`.private`, `.const`]

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Load data from memory (atomic):

```
ld<.addrspace>.memorder<.memscope><.cc>.type dst, [addr]
```

```
.addrspace = { .global, .generic, .shared }
.memorder = { .relaxed, .acquire, .seq_cst }
.memscope = { .workgroup, .gpu, .system }
.cc = { .uc, .L2wb, .L3wb }
.type = { .16b, .32b, .64b, .128b }


- addr is an address operand, in the form MemRR or MemRI
- dst is a register
```

Restrictions

- bitwidth of `dst` must match the bitwidth of `.type`
- qualifier `.cc` must not be specified when qualifier `.addrspace` is `.shared`
- qualifier `.memscope` can be specified only when qualifier `.memorder` is one of [`.acquire`, `.seq_cst`]
- qualifier `.type` cannot be `.64b` unless qualifier `.addrspace` is `.global`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

scalar load

```
dst = *addr
```

vector load

```
for (i = 0; i < element_count(.vec); i++)
   dst[i] = *(addr + i * sizeof(.type))
```

Notes:

- memory access requirements must be observed.
- memory data size restrictions must be observed.
- `.generic` address in atomic load must point to either `.global` or `.shared` address space.
- atomic load may utilize atomic cache controls.

Cache control definitions for non-atomic load instruction

Cache control	Description
.L1c	Cache the data in L1.
.L1uc	Do not cache the data in L1.

.L1s	Cache the data in L1, but make it more likely to be invalidated later.
.L2c	Cache the data in L2.
.L2uc	Do not cache the data in L2.
.L3c	Cache the data in L3.
.L3uc	Do not cache the data in L3.
.ri	Read-invalidate (e.g. last-use) on all levels of cache.

Examples:

```
.reg .16b %r, %b;
.shared .16b @S[100];
.reg .64b %a, %g;
.reg .v4.32b %v;

// load a 16-bit element into register %r from shared variable @S
// and cache it in all levels
ld.shared.L1c.L2c.L3c.16b %r, [@S + %b];

// load a 32-bit vector of 4 elements into vector %v from global memory
ld.global.v4.32b %v, [%g + 16];

// load a 64-bit element into register %a from generic address
// considering acquire memory ordering
ld.generic.acquire.64b %a, [%g];
```

`ld.param`

Load data from kernel parameter.

Syntax:

```
ld.param<.vec>.type dst, [vaddr]

  .vec = { .v2, .v3, .v4, .v8 }
  .type = { .8b, .16b, .32b, .64b }


  - dst is a register
```

Restrictions

- bitwidth of `dst` must match the bitwidth of `.type`
- number of elements of `dst` must match value of `.vec`
- qualifier `.vec` cannot be `.v3` unless qualifier `.type` is one of [`.32b`, `.64b`]
- qualifier `.vec` cannot be `.v8` unless qualifier `.type` is `.32b`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Notes:

`ld.param` supports the following address syntax:

- [@var] - references a kernel argument
- [@var + %reg] - references a kernel argument with a signed 32-bit offset specified by a 32-bit register
- [@var + imm] - references a kernel argument with a signed 32-bit offset specified by an immediate

Examples:

```
.kernel @foo(.param[4] %arg0, .param[8] %arg1) {
  .reg .32b %larg0, %larg1;

  // load contents of %arg0
  ld.param.32b %larg0, [%arg0];
  // load contents of %arg1[4]
  ld.param.32b %larg1, [%arg1 + 4];
}
```

st

Store data to memory.

Store data to memory (non-atomic):

```
st<.addrspace><.memorder><.cc><.vec>.type  [addr], src0

  .addrspace = { .global, .generic, .shared, .private }
  .memorder = { .weak }
  .cc = { .L1uc.L2uc.L3uc, .L1uc.L2uc.L3wb, .L1uc.L2wb.L3uc, .L1uc.L2wb.L3wb,
          .L1wt.L2uc.L3uc, .L1wt.L2uc.L3wb, .L1wt.L2wb.L3uc, .L1wt.L2wb.L3wb,
          .L1s.L2uc.L3uc, .L1s.L2uc.L3wb, .L1s.L2wb.L3uc, .L1wb.L2uc.L3uc,
          .L1wb.L2wb.L3uc, .L1wb.L2uc.L3wb }
  .vec = { .v2, .v3, .v4, .v8 }
  .type = { .8b, .16b, .32b, .64b }


  - addr is an address operand, in the form MemRR or MemRI
  - src0 is a register
```

Restrictions

- bitwidth of `src0` must match the bitwidth of `.type`
- number of elements of `src0` must match value of `.vec`
- qualifier `.vec` cannot be `.v3` unless qualifier `.type` is one of [`.32b`, `.64b`]
- qualifier `.vec` cannot be `.v8` unless qualifier `.type` is `.32b`
- qualifier `.cc` must not be specified when qualifier `.addrspace` is `.shared`
- qualifier `.memorder` must not be specified when qualifier `.addrspace` is `.private`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Store data to memory (atomic):

```
st<.addrspace>.memorder<.memscope><.cc>.type  [addr], src0

  .addrspace = { .global, .generic, .shared }
  .memorder = { .relaxed, .release, .seq_cst }
  .memscope = { .workgroup, .gpu, .system }
  .cc = { .uc, .L2wb, .L3wb }
  .type = { .16b, .32b, .64b, .128b }


  - addr is an address operand, in the form MemRR or MemRI
  - src0 is a register
```

Restrictions

- bitwidth of `src0` must match the bitwidth of `.type`
- qualifier `.cc` must not be specified when qualifier `.addrspace` is `.shared`
- qualifier `.memscope` can be specified only when qualifier `.memorder` is one of [`.release`, `.seq_cst`]
- qualifier `.type` cannot be `.64b` unless qualifier `.addrspace` is `.global`

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Semantics:

scalar store

```
*addr = src0
```

vector store

```
for (i = 0; i < element_count(.vec); i++)
   *(addr + i * sizeof(.type)) = src0[i]
```

Notes:

- memory access requirements must be observed.
- memory data size restrictions must be observed.
- `.generic` address in atomic store must point to either `.global` or `.shared` address space.
- atomic store may utilize atomic cache controls.

Cache control definitions for non-atomic store instruction

Cache control	Description
.L1uc	Bypass L1 and write directly to the next level cache.
.L1wb	Write data to L1 but not the next level cache.
.L1wt	Write data to both L1 and the next level cache.
.L1s	Streaming write to L1.
.L2uc	Bypass L2 and write directly to the next level cache.
.L2wb	Write data to L2 but not the next level cache.
.L3uc	Bypass L3 and write directly to the next level cache.
.L3wb	Write data to L3 but not the next level cache.

Examples:

```
.reg .16b %r;
.reg .32b %off;
.private .16b @P[64];
.reg .64b %a, %g;
.reg .v2.32b %v;

// store a 16-bit element from register %r
// into private array @P
st.private.16b [@P + %off], %r;

// store 32-bit vector of 2 elements from register %v
// into global address [%g]
st.global.v2.32b [%g], %v;

// store 64-bit element from register %a into generic address [%g]
st.generic.64b [%g], %a;
```

Synchronization

`barrier.workgroup`

Synchronize work-items within the same work-group.

Syntax:

```
barrier.workgroup
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Notes:

All work-items in a work-group must execute this instruction before any are allowed to continue execution past it. Specifically,

- If the barrier is inside a conditional statement, all work-items must take that branch if any work-item executes the barrier.
- If the barrier is inside a loop, all work-items must execute the barrier on every iteration.

A work-item that has already exited the kernel via a return instruction is not required to reach the barrier.

If the above conditions are not satisfied, the behavior is undefined.

Examples:

```
// barrier across work-items in the work-group
barrier.workgroup;
```

`fence`

Wait for previous memory accesses to become observable in the specified scope.

Fence on shared or generic address space:

```
fence<.addrspace><.memorder>.memscope

  .addrspace = { .shared, .generic }
  .memorder = { .acquire, .release, .acq_rel, .seq_cst }
  .memscope = { .workgroup, .gpu }
```

PISA Notes

- introduced in PISA version **0.1**

- requires PISA target **100** or compatible

Fence on global address space:

```
fence.addrspace<.memorder>.memscope

  .addrspace = { .global }
  .memorder = { .acquire, .release, .acq_rel, .seq_cst }
  .memscope = { .workgroup, .gpu, .system }
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Sub-group fence:

```
fence.memscope

  .memscope = { .subgroup }
```

PISA Notes

- introduced in PISA version **0.1**
- requires PISA target **100** or compatible

Notes:

A fence instruction guarantees that all prior memory accesses (loads, stores, and atomics) to the specified address space by this work-item are observable for the set of work-items in the specified memory scope. For generic address space, the fence operation will apply to both shared and global memory accesses. Together with other synchronization instructions, they can be used to establish the ordering of relaxed memory operations in different work-items.

Examples:

```
// memory fence at work-group scope
fence.workgroup;
// memory fence at gpu scope with acquire memory ordering
fence.acquire.gpu;
```

Appendix

Directives

Module

- Compatibility
  - `.version` - Version
  - `.target` - Target

Variables

- Declaration
  - `.reg` - Register Variables
  - `.pred` - Predicate Variables
  - `.global` - Memory Variables and Global Variables
  - `.const` - Memory Variables and Global Variables
  - `.private` - Memory Variables
  - `.shared` - Memory Variables
- Initialization
  - `.zero`
- Linkage
  - `.export`
  - `.import`
- Attributes
  - `.align` - Alignment
  - `.section`
  - `.host_access`

Kernels

- Declaration
  - `.kernel`
- Attributes
  - `.reqd_work_group_size`
  - `.vec_type_hint`
- Parameters
  - `.param`
- Parameter Attributes
  - `.align`
  - `.addrspace`
  - `.ptr_align`

Functions

- Declaration
  - `.function`
- Linkage
  - `.export`
  - `.import`

Notices

Legal Notices and Disclaimers