PSP VFPU instruction set documentation

Introduction

This document describes how the PSP VFPU instruction set operates. We attempted to collect all the knowledge available in the community and put it toghether in a document that can be used as a reference for developers and enthusiasts.

The goal is to describe the behaviour of the hardware unit with as much detail as possible in a way that every statement can be verified. For this reason, every functional detail described in the docs must have a test that validates it. Of course some things are harder to validate (like hardware bugs) so there's some statements that won't have tests for them at this time.

MIPS allegrex CPU

The Allegrex CPU is a MIPS CPU based on the `MIPS II` architecture. This is a 32 bit CPU and architecture that has many similarities with other CPUs of the same architecture. However, if we only focus on the instruction set, the main differences with other CPUs in the `MIPS II` family would be:

- Lack of 64 bit FPU support (only single float support).
- Lack of MMU/TLB (only certain memory protections are available).
- Some extra MIPS32r2 instructions (mostly arithmetic and bit manipulation).
- Other COP0 instructions, some borrowed from MIPS32
- Lack of *Coprocessor 3* and custom *Coprocessor 2* (VFPU)

Most of the extra instructions that are present in the CPU are identical to their MIPS32 counterparts. In some cases though, the encoding is slightly different.

VFPU unit

The PSP VFPU is a coprocessor unit that can perform vector/matrix float and integer operations on a set of 128 bit registers. It features dedicated units to perform the most usual operations that 3D videogames require.

Register set

The CPU features 128 registers, each of them 32 bit wide. Most of the time they are interpreted as IEEE-754 compliant floating point registers, although some instructions will interpret them as integers (or other formats such as 8/16 bit packed integers). The registers can be addressed individually but also in a more powerful way by grouping them as vectors or matrices.

Registers will usually be represented in their matrix layout. The VFPU has 8 matrices, each of them containing 16 elements (4 rows by 4 columns). For each of the 8 total available matrices, the elements are arranged in the following fashion (`X` represents the matrix number, 0 to 7):

Single 32 bit elements

SX00	SX10	SX20	SX30
SX01	SX11	SX21	SX31
SX02	SX12	SX22	SX32
SX03	SX13	SX23	SX33

When the registers are referenced as vectors, they are grouped as rows and columns of a given matrix. This is important since it means that a vector is composed of elements from a single matrix and cannot access elements across multiple matrices. There's 2D, 3D and 4D vectors, usually called pair, trio and quad respectively. Single elements can be viewed as 1D vectors, and most instructions are available in all four possible vector sizes (which makes the instruction set very uniform). Not all access patterns are possible: pair and trio registers have 128 possible addressing modes while quad has only 64. The available patterns are described as follows:

2D vector rows

RX00		RX20	
RX01		RX21	
RX02		RX22	
RX03		RX23	

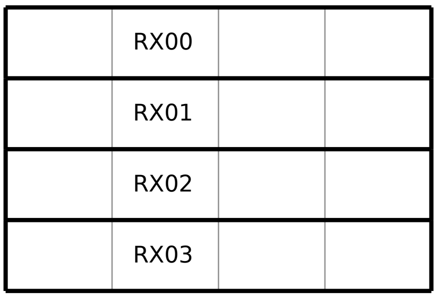
3D vector rows

	RX00		
	RX01		
	RX02		
	RX03		

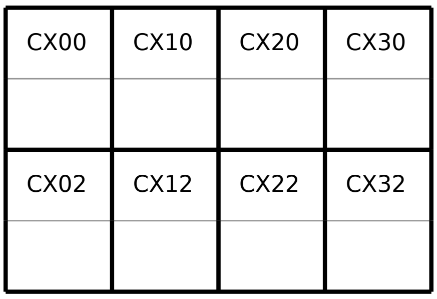
3D vector rows

		RX10	
		RX11	
		RX12	
		RX13	

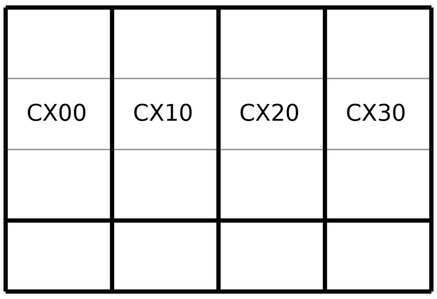
4D vector rows



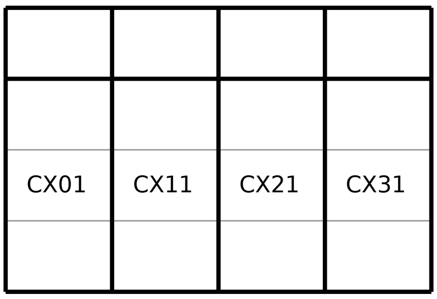
2D vector cols



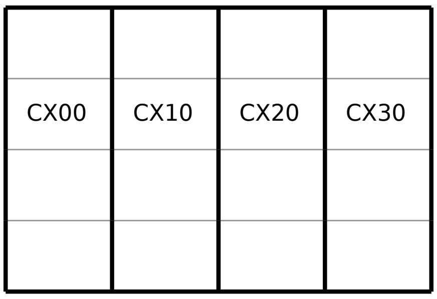
3D vector cols



3D vector cols

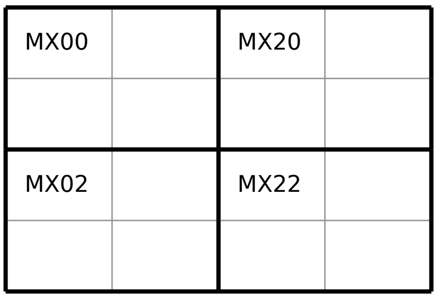


4D vector cols

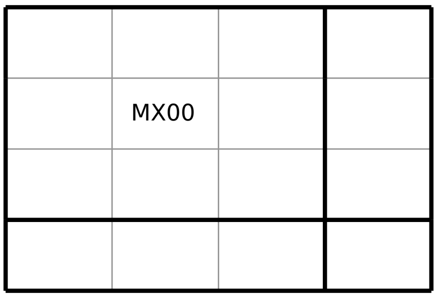


Matrix addressing is similar to vectors: registers can be read vertically or horizontally. That means matrices can be accessed in a row major and column major mode (ie. by accessing them as a set of rows or columns). Similarly there's three possible sizes: 2x2, 3x3 and 4x4, containing 4, 9 and 16 registers respectively. Again not all addressing patterns are available, having 64 possible addressing modes for 2x2 and 3x3 matrices, but only 16 for 4x4 matrices. These are:

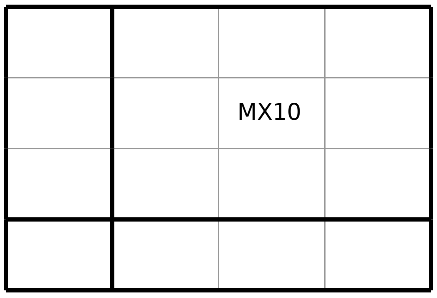
2D matrix



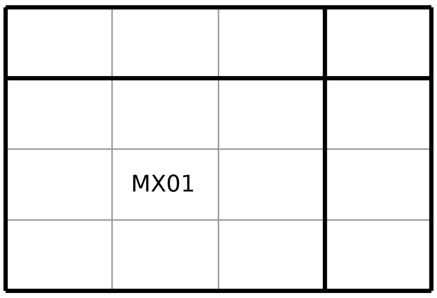
3D matrix



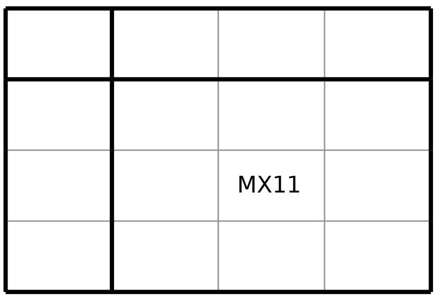
3D matrix



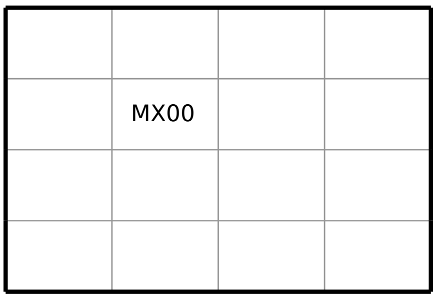
3D matrix



3D matrix



4D matrix



There's also a small set of eight "control" registers that are used for a variety of things, such as prefix state, comparison flag bits, etc. These registers are defined as follow:

- Reg 128 (VFPU_PFXS): holds the rs prefix value.
- Reg 129 (VFPU_PFXT): holds the rt prefix value.
- Reg 130 (VFPU_PFXD): holds the rd prefix value.
- Reg 131 (VFPU_CC): holds the condition code value.
- Reg 135 (VFPU_REV): read only register with VFPU revision information.
- Regs 136 to 147 (VFPU_RCX0 to VFPU_RCX7): Pseudorandom generator state.

Some of these registers are never accessed directly but rather using some VFPU instructions (ie. prefixes, condition code, etc). However these can be read and written in some useful cases, for instance thread context saving and restoration (so that the VFPU state is preserved across thread rescheduling).

Register hazards

Most CPUs have what's called "hazard detection logic", which tracks register reads and writes so that things happen in the right order and results actually make sense. In the VFPU this is also the case, however some operations are quite complex and can be complex to track.

Control registers seem to have some hazards, for instance "mfvc" instruction has a one cycle hazard with any previous vcmp instruction. That means a vnop or some other VFPU instruction should be inserted between a vcmp and mfvc instruction pair to get the right VFPU_CC value.

Some VFPU instructions (mostly dealing with matrices and transformations) require that the input and output registers do not overlap. This has to do with how the hardware performs the operations internally: the VFPU can perform most vector-vector operations in a native way, but matrix operations seem to be decomposed into series of vector-vector operations (ie. a vmmul seems to be a sequence of vtfm operations). Since the results are only partial, the inputs are overwritten before the CPU can even read them, causing incorrect results for the operation.

The affected instructions are divided in two groups, a group that does not allow any sort of overlap, and another group that allows some limited overlap. Instructions vmmul, vtfm2/3/4, vhtfm2/3/4, vqmul and vcrsp do not allow any sort of overlap between input and output registers. These instructions perform operations by repeating a dot product operation multiple times, which results in partial updates of the output register. This partial updates overwrite the input register causing the result to be incorrect.

Instructions that allow partial overlaps are vsin, vcos, vasin, vnsin, vexp2, vrexp2, vlog2, vsqrt, vrsq, vrcp, vnrcp, vdiv, vmscl and vmmov. Single versions (.s) are not affected by this restriction. These instructions are also internally decomposed into a bunch of smaller operations (for instance trigonometric operations are decomposed into a series of single (.s) operations). The registers are allowed to overlap as long as they are compatible in terms of element count and access "direction" (ie. a matrix must be read using the same mode).

Examples

```
vmscl.p M000, M022, S100    # No overlap, always OK
vmscl.p M000, M000, S100    # M000 overlaps with itself, OK
vmscl.p M000, E000, S100    # Invalid overlap, matrix order is different
vmscl.t M000, M011, S100    # Overlapping registers are not identical

vcos.q R000, C000           # Invalid overlap (one element only)
vcos.q R000, R000           # Identical overlap, OK
```

Floating point format

Although the FPU seems IEEE-754 compliant, it has a couple of non-standard features that break this compatibility. Its rounding mode is hardwired to "round to nearest" mode, so that users cannot choose another rounding mode. It also lacks support for denormal numbers (also called subnormals): when an operation produces a subnormal number, it rounds it to zero. If the input of an operation is a denormal number, it will also be treated as zero.

See the `ieee754-fun.c` file for tests.

Instruction execution

The VFPU is a pipelined CPU with an issue width of one. That means that instructions take multiple cycles to execute, since they execute partially during each cycle, and a maximum of one new instruction begins execution each cycle. Instructions that block the pipeline for more than one cycle can be identified by having a throughput different than one. These block the pipeline for a certain number of cycles before a new instruction can enter it.

An instruction usually begins executing whenever its input registers are `ready`, that is, any previous instruction writing those registers have fully completed their execution. For this reason it is important to closely observe the instruction latency, measured in cycles, since an instruction might have to wait for its inputs to become available, reducing efficiency. A common strategy is to interleave non-dependant instructions to `hide` latency and avoid wasting CPU cycles.

The pipeline structure looks more or less as follows:

- Register read
- Input prefix operations
- VFPU operation (arithmetic, logic)
- Output prefix operation
- Register write

Prefix operations allow to perform certain operations on the inputs before the actual instruction operation and some other operations on the output.

Prefix operations

VFPU operations can operate on one or two inputs (`rs` and `rt`) and one output (`rd`). The input values can be pre-processed by using the `VFPU_PFXS` and `VFPU_PFXT` registers (and therefore `vpfxs` and `vpfxt` instructions). The result of the operation being written to `rd` can be post-processed by using the `VFPU_PFXD` register (`vpfxd` instruction).

Valid operations for input registers are:

- Sign change (negation)
- Absolute value
- Swizzle (rearranging elments in a row/col)
- Override element with constant value.

Operations available to the output register post-processing are:

- Value clamping (to ranges 0..1 or -1..1)
- Write masking (disable register write)

There's some restrictions on their usage. The assembler will signal an error should you violate any of the restrictions.

- Constant values can only be 0, 1, 2, 3, 1/2, 1/3, 1/4, 1/6 or any of their negative counterparts
- Swizzle cannot extend beyond the operand size (ie. you cannot use .z with a an instruction that uses single or pair elements).

A few examples to showcase input prefixes:

```
# Sign change prefix
vmul.p R000, R001, R002[-x,-y]     # Multiplies two rows negating one of the inputs
                                   # S000 = S001 * -S002;  S010 = S011 * -S012

vfad.q R000, R001[x,-y,z,-w]       # Funnel-add all elements with some changed signs
                                   # S000 = S001 - S011 + S021 - S031

# Absolute value prefix
vdot.p S000, R001[|x|,|y|], R002   # Dot product with forced absolute value for R001
                                   # S000 = |S001| * S002 + |S011| * S012
# Negative and absolute value prefixes
vdot.p S000, R001[-|x|,-|y|], R002   # Dot product with forced negative values
                                     # S000 = -|S001| * S002 - |S011| * S012

# Swizzle prefix
vdot.q R000, R001, R002[x,y,x,y]   # Multiplies with repeating values
                                   # S000 = S001 * S002;  S010 = S011 * S012
                                   # S020 = S021 * S002;  S030 = S031 * S012

# Constant value prefixes
vdot.t R000, R001, R002[1,2,3]     # Second operand ignored, overrides to (1,2,3)
                                   # S000 = S001 + S011 * 2 + S021 * 3

vdot.t R000, R001, R002[x,-2,-y]   # Mix swizzle and constant elements
                                   # S000 = S001 * S002 - S011 * 2 - S021 * S012
```

Some more examples for output prefixes.

```
vmul.p R000[[-1:1],[-1:1]], R001, R002  # Multiplies with output saturation
                                        # S000 = min(1.0f, max(-1.0f, S001 * S002))
                                        # S010 = min(1.0f, max(-1.0f, S011 * S012))
```

Adding a prefix modifier to an operand will result in `vpfxs/t/d` instructions being emitted before the actual instruction. This syntax exists just to make assembly coding more comfortable to the user. When using the disassembler the prefix instructions will be clearly visible.

```
# The following operand-decorated instruction:
vmul.q R000, R100[x,y,x,y], R200[-x,-y,z,w]

# is actually encoded as a sequence of instructions:
vpfxs [x,y,x,y]
vpfxt [-x,-y,z,w]
vmul.q R000, R100, R200
```

Prefix instructions consume one cycle and have no visible latency (the "decorated" instruction doesn't have to wait any extra cycles). In some cases it might be faster to not use prefixes and use other instructions (vcst, vabs, vneg, vsat0/1 are some similar alternatives), particularly when optimizing for throughput. The advantage of using prefixes is that latency is kept low (since they have no latency and the extra operation is "included" in the instruction pipeline).

Allegrex Instructions

Bit manipulation instructions

The following instructions exist in the Allegrex CPU and share the same MIPS32 encodings:

- *seb*: Sign extend byte (byte to word signed extension)
- *seh*: Sign extend half-word (half-word to word signed extension)
- *ext*: Extract bit field (extract a bit field in a zeroed register)
- *ins*: Insert bit field (insert lower bits into another register)
- *wsbh*: Swap bytes within a half-word

Other instructions that are borrowed from MIPS32 but have a different encoding are:

- *clo*: Count leading ones (uses some unused `SPECIAL` encodings)
- *clz*: Count leading zeros (uses some unused `SPECIAL` encodings)

The bit manipulation Allegrex specific instructions are:

- *wsbw*: Swap bytes in word (uses `BSHFL` encoding adjacent to `wsbh`)
- *bitrev*: Reverse bits in a word (uses unused `BSHFL` encoding)

Arithmetic-Logical instructions

Allegrex features some instructions present in MIPS32 and MIPS32r2 with identical encoding to these:

- *rotr*: Rotate word right by a fixed amount
- *rotrv*: Rotate word right by a variable amount
- *movz*: Conditional register move on zero
- *movn*: Conditional register move on non-zero

Other instructions that have some particular encoding are multiply-accumulate instructions. Some overlap with `MIPS R4010` encodings and some others just use unused encodings. They all use unused `SPECIAL` opcodes:

- *madd*: Signed multiply-accumulate integer
- *maddu*: Unsigned multiply-accumulate integer
- *msub*: Signed multiply-subtract integer
- *msubu*: Unsigned multiply-subtract integer

There's also two novel Allegrex instructions that are used to perform faster compare-and-move operations. These use free `SPECIAL` opcodes as well:

- *min*: Selects smallest (signed) value between two registers.
- *max*: Selects greatest (signed) value between two registers.

bvf

VFPU branch on false

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	1	0	0	0	vfpucc			0	0	offset															

Syntax

bvf imm3, offset

Description

Branch on VFPU CC register being false

Instruction performance

Throughput: 1 cycles/instruction
Latency: 4 cycles

bvfl

VFPU likely branch on false

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	1	0	0	0	vfpucc			1	0	offset															

Syntax

bvfl imm3, offset

Description

Branch on VFPU CC register being false (likely)

Instruction performance

Throughput: 1 cycles/instruction
Latency: 4 cycles

bvt

VFPU branch on true

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	1	0	0	0	vfpucc			0	1	offset															

Syntax

bvt imm3, offset

Description

Branch on VFPU CC register being true

Instruction performance

Throughput: 1 cycles/instruction
Latency: 4 cycles

bvtl

VFPU likely branch on true

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	1	0	0	0	vfpucc			1	1	offset															

Syntax

bvtl imm3, offset

Description

Branch on VFPU CC register being true (likely)

Instruction performance

Throughput: 1 cycles/instruction
Latency: 4 cycles

mtvc

Move GPR to VFPU control register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	1	1	1	gpr					0	0	0	0	0	0	0	0	vfpucc							

Syntax

mtvc rt, imm8

Description

Writes the contents of a CPU general purpose register to the specified VFPU control register

mfvc

Move VFPU control register to GPR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	1	gpr					0	0	0	0	0	0	0	0	vfpucc							

Syntax

mfvc rt, imm8

Description

Writes the contents of the specified VPFU control register into a CPU general purpose register

Hazards

The instruction does not have interlocks, so the result of a vcmp instruction is only available one cycle later. You will need to interleave at least one VFPU instruction between a vcmp and mfvc (ie. a vnop).

vmtvc Move vector register to VFPU control register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	1	0	0	0	1	0	rs							vfpucc							

Syntax

vmtvc imm8, rs

Description

Writes the contents of a VFPU vector general to the specified VFPU control register

vmfvc

Move VFPU control register to vector register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	1	0	0	0	0	vfpucc								0	rd						

Syntax

vmfvc rd, imm8

Description

Writes the contents of the specified VPFU control register into a VFPU vector register

Hazards

The instruction does not have interlocks, so the result of a previous vcmp instruction is only available one cycle later. You will need to interleave at least one VFPU instruction between a vcmp and mfvc (ie. a vnop).

lv.s

Load VFPU element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	0	gpr					rtlo					offset														rthi	

Syntax

lv.s rd, imm14(rt)

Description

Performs a 4 byte memory load to a VFPU register. Address must be 4 byte aligned or a fault is generated.

Allowed prefixes

- rd: Not supported

lv.q

Load VFPU quad element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	gpr					rtlo					offset														0	rthi

Syntax

lv.q rd, imm14(rt)

Description

Performs a 16 byte memory load to a VFPU quad register. Address must be 16 byte aligned or a fault is generated.

Allowed prefixes

- rd: Not supported

lvl.q

Load left VFPU quad element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	gpr					rtlo					offset														0	rthi

Syntax

lvl.q rd, imm14(rt)

Description

Performs a 16 byte left unaligned memory load to a VFPU quad register. Instruction ignores the two LSB (forces them to zero), so the address is assumed aligned to 4 bytes. This instruction is similar to MIPS LWL instruction: loads the most significant elements from the specified address leaving the other elements unchanged. Users can use `ulv.q` pseudoinstruction to generate a sequence of `lvl.q` and `lvr.q` instructions in order to load unaligned data. You can check `psp-tests/manual/memops.c` to see examples on how the instruction behaves.

Bugs

The instruction has an errata on PSP-1000 models that causes FPU register corruption (these are the MIPS CPU FPU registers, not the VFPU registers). The bottom 5 bits of the VFPU destination register determine which FPU register will be corrupted. A workaround is to assume the side effect (ie. mark the register are clobbered).

Allowed prefixes

- rd: Not supported

lvr.q

Load right VFPU quad element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	gpr					rtlo					offset														1	rthi

Syntax

lvr.q rd, imm14(rt)

Description

Performs a 16 byte right unaligned memory load to a VFPU quad register. Instruction ignores the two LSB (forces them to zero), so the address is assumed aligned to 4 bytes. This instruction is similar to MIPS LWR instruction: loads the least significant elements from the specified address leaving the other elements unchanged. Users can use `ulv.q` pseudoinstruction to generate a sequence of `lvl.q` and `lvr.q` instructions in order to load unaligned data. You can check `psp-tests/manual/memops.c` to see examples on how the instruction behaves.

Bugs

The instruction has an errata on PSP-1000 models that causes FPU register corruption (these are the MIPS CPU FPU registers, not the VFPU registers). The bottom 5 bits of the VFPU destination register determine which FPU register will be corrupted. A workaround is to assume the side effect (ie. mark the register are clobbered).

Allowed prefixes

- rd: Not supported

sv.s

Store VFPU element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	gpr					rtlo					offset														rthi	

Syntax

sv.s rs, imm14(rt)

Description

Performs a 4 byte memory store from a VFPU register. Address must be 4 byte aligned or a fault is generated.

Allowed prefixes

- rd: Not supported

sv.q

Store VFPU quad element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	gpr					rtlo					offset														0	rthi

Syntax

sv.q rs, imm14(rt)

Description

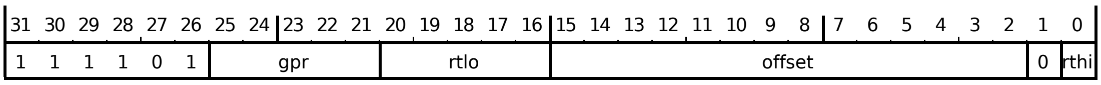
Performs a 16 byte memory store from a VFPU quad register. Address must be 16 byte aligned or a fault is generated.

Allowed prefixes

- rd: Not supported

svl.q

Store left VFPU quad element



Syntax

svl.q rs, imm14(rt)

Description

Performs a 16 byte left unaligned memory store from a VFPU quad register. Instruction ignores the two address LSB (forces them to zero), so the address is assumed aligned to 4 bytes. This instruction is similar to MIPS SWL instruction: stores the most significant part of the elements to the specified address leaving any other elements unchanged. Users can use `usv.q` pseudoinstruction to generate a sequence of `svl.q` and `svr.q` instructions in order to store unaligned data. You can check `psp-tests/manual/memops.c` to see examples on how the instruction behaves.

Allowed prefixes

- rd: Not supported

svr.q

Store right VFPU quad element

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	gpr					rtlo					offset														1	rthi

Syntax

svr.q rs, imm14(rt)

Description

Performs a 16 byte right unaligned memory store from a VFPU quad register. Instruction ignores the two address LSB (forces them to zero), so the address is assumed aligned to 4 bytes. This instruction is similar to MIPS SWR instruction: stores the least significant part of the elements to the specified address leaving any other elements unchanged. Users can use `usv.q` pseudoinstruction to generate a sequence of `svl.q` and `svr.q` instructions in order to store unaligned data. You can check `psp-tests/manual/memops.c` to see examples on how the instruction behaves.

Allowed prefixes

- rd: Not supported

vadd.s

Add elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	0	0	rt							0	rs							0	rd						

Syntax

vadd.s rd, rs, rt

Description

Performs element-wise floating point addition

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] + rt[0]
```

vadd.p

Add elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	0	0	rt							0	rs							1	rd						

Syntax

vadd.p rd, rs, rt

Description

Performs element-wise floating point addition

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] + rt[0]
rd[1] = rs[1] + rt[1]
```

vadd.t

Add elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	0	0				rt				1				rs				0				rd			

Syntax

vadd.t rd, rs, rt

Description

Performs element-wise floating point addition

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] + rt[0]
rd[1] = rs[1] + rt[1]
rd[2] = rs[2] + rt[2]
```

vadd.q

Add elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	0	0	rt							1	rs							1	rd						

Syntax

vadd.q rd, rs, rt

Description

Performs element-wise floating point addition

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] + rt[0]
rd[1] = rs[1] + rt[1]
rd[2] = rs[2] + rt[2]
rd[3] = rs[3] + rt[3]
```

vsub.s

Subtract elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	0	1	rt							0	rs							0	rd						

Syntax

vsub.s rd, rs, rt

Description

Performs element-wise floating point subtraction

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] - rt[0]
```

vsub.p

Subtract elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	0	1	rt							0	rs							1	rd						

Syntax

vsub.p rd, rs, rt

Description

Performs element-wise floating point subtraction

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] - rt[0]
rd[1] = rs[1] - rt[1]
```

vsub.t

Subtract elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	0	1	rt							1	rs							0	rd						

Syntax

vsub.t rd, rs, rt

Description

Performs element-wise floating point subtraction

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] - rt[0]
rd[1] = rs[1] - rt[1]
rd[2] = rs[2] - rt[2]
```

vsub.q

Subtract elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	0	1	rt							1	rs							1	rd						

Syntax

vsub.q rd, rs, rt

Description

Performs element-wise floating point subtraction

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] - rt[0]
rd[1] = rs[1] - rt[1]
rd[2] = rs[2] - rt[2]
rd[3] = rs[3] - rt[3]
```

vmul.s

Multiply elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	0	rt							0	rs							0	rd						

Syntax

vmul.s rd, rs, rt

Description

Performs element-wise floating point multiplication

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] * rt[0]
```

vmul.p

Multiply elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	0	rt							0	rs							1	rd						

Syntax

vmul.p rd, rs, rt

Description

Performs element-wise floating point multiplication

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] * rt[0]
rd[1] = rs[1] * rt[1]
```

vmul.t

Multiply elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	0	rt							1	rs							0	rd						

Syntax

vmul.t rd, rs, rt

Description

Performs element-wise floating point multiplication

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] * rt[0]
rd[1] = rs[1] * rt[1]
rd[2] = rs[2] * rt[2]
```

vmul.q

Multiply elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	0	rt							1	rs							1	rd						

Syntax

vmul.q rd, rs, rt

Description

Performs element-wise floating point multiplication

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] * rt[0]
rd[1] = rs[1] * rt[1]
rd[2] = rs[2] * rt[2]
rd[3] = rs[3] * rt[3]
```

vdiv.s

Divide elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	1	1	1	rt							0	rs							0	rd						

Syntax

vdiv.s rd, rs, rt

Description

Performs element-wise floating point division

Instruction performance

Throughput: 14 cycles/instruction
Latency: 17 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)
- rt: Full support (swizzle, abs(), neg() and constants)

Pseudocode

```
rd[0] = rs[0] / rt[0]
```

vdiv.p

Divide elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	1	1	1	rt							0	rs							1	rd						

Syntax

vdiv.p rd, rs, rt

Description

Performs element-wise floating point division

Instruction performance

Throughput: 28 cycles/instruction
Latency: 31 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rt: Not supported
- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = rs[0] / rt[0]
rd[1] = rs[1] / rt[1]
```

vdiv.t

Divide elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	1	1	1	rt							1	rs							0	rd						

Syntax

vdiv.t rd, rs, rt

Description

Performs element-wise floating point division

Instruction performance

Throughput: 42 cycles/instruction
Latency: 45 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rt: Not supported
- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = rs[0] / rt[0]
rd[1] = rs[1] / rt[1]
rd[2] = rs[2] / rt[2]
```

vdiv.q

Divide elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	1	1	1	rt							1	rs							1	rd						

Syntax

vdiv.q rd, rs, rt

Description

Performs element-wise floating point division

Instruction performance

Throughput: 56 cycles/instruction
Latency: 59 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rt: Not supported
- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = rs[0] / rt[0]
rd[1] = rs[1] / rt[1]
rd[2] = rs[2] / rt[2]
rd[3] = rs[3] / rt[3]
```

vmin.s

Select smallest elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	0	1	0	rt							0	rs							0	rd						

Syntax

vmin.s rd, rs, rt

Description

Performs element-wise floating point min(rs, rt) operation

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = fminf(rs[0], rt[0])
```

vmin.p

Select smallest elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	0	1	0	rt							0	rs							1	rd						

Syntax

vmin.p rd, rs, rt

Description

Performs element-wise floating point min(rs, rt) operation

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = fminf(rs[0], rt[0])
rd[1] = fminf(rs[1], rt[1])
```

vmin.t

Select smallest elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	0	1	0	rt							1	rs							0	rd						

Syntax

vmin.t rd, rs, rt

Description

Performs element-wise floating point min(rs, rt) operation

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = fminf(rs[0], rt[0])
rd[1] = fminf(rs[1], rt[1])
rd[2] = fminf(rs[2], rt[2])
```

vmin.q

Select smallest elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	0	1	0	rt							1	rs							1	rd						

Syntax

vmin.q rd, rs, rt

Description

Performs element-wise floating point min(rs, rt) operation

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = fminf(rs[0], rt[0])
rd[1] = fminf(rs[1], rt[1])
rd[2] = fminf(rs[2], rt[2])
rd[3] = fminf(rs[3], rt[3])
```

vmax.s

Select biggest elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	0	1	1	rt							0	rs							0	rd						

Syntax

vmax.s rd, rs, rt

Description

Performs element-wise floating point max(rs, rt) operation

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = fmaxf(rs[0], rt[0])
```

vmax.p

Select biggest elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	0	1	1	rt							0	rs							1	rd						

Syntax

vmax.p rd, rs, rt

Description

Performs element-wise floating point max(rs, rt) operation

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = fmaxf(rs[0], rt[0])
rd[1] = fmaxf(rs[1], rt[1])
```

vmax.t

Select biggest elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	0	1	1	rt							1	rs							0	rd						

Syntax

vmax.t rd, rs, rt

Description

Performs element-wise floating point max(rs, rt) operation

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = fmaxf(rs[0], rt[0])
rd[1] = fmaxf(rs[1], rt[1])
rd[2] = fmaxf(rs[2], rt[2])
```

vmax.q

Select biggest elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	0	1	1	rt							1	rs							1	rd						

Syntax

vmax.q rd, rs, rt

Description

Performs element-wise floating point max(rs, rt) operation

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = fmaxf(rs[0], rt[0])
rd[1] = fmaxf(rs[1], rt[1])
rd[2] = fmaxf(rs[2], rt[2])
rd[3] = fmaxf(rs[3], rt[3])
```

vscmp.s

Compare and set elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	0	1	rt							0	rs							0	rd						

Syntax

vscmp.s rd, rs, rt

Description

Performs element-wise floating point comparison. The result is -1.0f, 0.0f or 1.0f depending on whether the input vs is less that vt, equal, or greater, respectively.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] < rt[0] ? -1f : rs[0] > rt[0] ? 1.0f : 0.0f
```

vscmp.p — Compare and set elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	0	1	rt							0	rs							1	rd						

Syntax

vscmp.p rd, rs, rt

Description

Performs element-wise floating point comparison. The result is -1.0f, 0.0f or 1.0f depending on whether the input vs is less that vt, equal, or greater, respectively.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] < rt[0] ? -1f : rs[0] > rt[0] ? 1.0f : 0.0f
rd[1] = rs[1] < rt[1] ? -1f : rs[1] > rt[1] ? 1.0f : 0.0f
```

vscmp.t

Compare and set elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	0	1	rt							1	rs							0	rd						

Syntax

vscmp.t rd, rs, rt

Description

Performs element-wise floating point comparison. The result is -1.0f, 0.0f or 1.0f depending on whether the input vs is less that vt, equal, or greater, respectively.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] < rt[0] ? -1f : rs[0] > rt[0] ? 1.0f : 0.0f
rd[1] = rs[1] < rt[1] ? -1f : rs[1] > rt[1] ? 1.0f : 0.0f
rd[2] = rs[2] < rt[2] ? -1f : rs[2] > rt[2] ? 1.0f : 0.0f
```

vscmp.q

Compare and set elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	0	1	rt							1	rs							1	rd						

Syntax

vscmp.q rd, rs, rt

Description

Performs element-wise floating point comparison. The result is -1.0f, 0.0f or 1.0f depending on whether the input vs is less that vt, equal, or greater, respectively.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] < rt[0] ? -1f : rs[0] > rt[0] ? 1.0f : 0.0f
rd[1] = rs[1] < rt[1] ? -1f : rs[1] > rt[1] ? 1.0f : 0.0f
rd[2] = rs[2] < rt[2] ? -1f : rs[2] > rt[2] ? 1.0f : 0.0f
rd[3] = rs[3] < rt[3] ? -1f : rs[3] > rt[3] ? 1.0f : 0.0f
```

vsge.s Compare greater or equal and set elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	1	0	rt							0	rs							0	rd						

Syntax

vsge.s rd, rs, rt

Description

Performs element-wise floating point bigger-or-equal comparison. The result will be 1.0 if vs is bigger or equal to vt, otherwise will be zero.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] >= rt[0] ? 1.0f : 0.0f
```

vsge.p Compare greater or equal and set elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	1	0	rt							0	rs							1	rd						

Syntax

vsge.p rd, rs, rt

Description

Performs element-wise floating point bigger-or-equal comparison. The result will be 1.0 if vs is bigger or equal to vt, otherwise will be zero.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] >= rt[0] ? 1.0f : 0.0f
rd[1] = rs[1] >= rt[1] ? 1.0f : 0.0f
```

vsge.t Compare greater or equal and set elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	1	0	rt							1	rs							0	rd						

Syntax

vsge.t rd, rs, rt

Description

Performs element-wise floating point bigger-or-equal comparison. The result will be 1.0 if vs is bigger or equal to vt, otherwise will be zero.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] >= rt[0] ? 1.0f : 0.0f
rd[1] = rs[1] >= rt[1] ? 1.0f : 0.0f
rd[2] = rs[2] >= rt[2] ? 1.0f : 0.0f
```

vsge.q Compare greater or equal and set elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	1	0	rt							1	rs							1	rd						

Syntax

vsge.q rd, rs, rt

Description

Performs element-wise floating point bigger-or-equal comparison. The result will be 1.0 if vs is bigger or equal to vt, otherwise will be zero.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] >= rt[0] ? 1.0f : 0.0f
rd[1] = rs[1] >= rt[1] ? 1.0f : 0.0f
rd[2] = rs[2] >= rt[2] ? 1.0f : 0.0f
rd[3] = rs[3] >= rt[3] ? 1.0f : 0.0f
```

vslt.s

Compare less-than and set elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	1	1	rt							0	rs							0	rd						

Syntax

vslt.s rd, rs, rt

Description

Performs element-wise floating point less-than comparison. The result will be 1.0 if vs less than vt, otherwise will be zero.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] < rt[0] ? 1.0f : 0.0f
```

vslt.p

Compare less-than and set elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	1	1	rt							0	rs							1	rd						

Syntax

vslt.p rd, rs, rt

Description

Performs element-wise floating point less-than comparison. The result will be 1.0 if vs less than vt, otherwise will be zero.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] < rt[0] ? 1.0f : 0.0f
rd[1] = rs[1] < rt[1] ? 1.0f : 0.0f
```

vslt.t

Compare less-than and set elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	1	1	rt							1	rs							0	rd						

Syntax

vslt.t rd, rs, rt

Description

Performs element-wise floating point less-than comparison. The result will be 1.0 if vs less than vt, otherwise will be zero.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] < rt[0] ? 1.0f : 0.0f
rd[1] = rs[1] < rt[1] ? 1.0f : 0.0f
rd[2] = rs[2] < rt[2] ? 1.0f : 0.0f
```

vslt.q Compare less-than and set elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	1	1	rt							1	rs							1	rd						

Syntax

vslt.q rd, rs, rt

Description

Performs element-wise floating point less-than comparison. The result will be 1.0 if vs less than vt, otherwise will be zero.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] < rt[0] ? 1.0f : 0.0f
rd[1] = rs[1] < rt[1] ? 1.0f : 0.0f
rd[2] = rs[2] < rt[2] ? 1.0f : 0.0f
rd[3] = rs[3] < rt[3] ? 1.0f : 0.0f
```

vcrs.t

Partial vector cross product

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	1	0	1	rt							1	rs							0	rd						

Syntax

vcrs.t rd, rs, rt

Description

Performs a partial cross-product operation

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Full support (masking and saturation)
- rt: Not supported
- rs: Not supported

Pseudocode

```
rd[0] = rs[1] * rt[2]
rd[1] = rs[2] * rt[0]
rd[2] = rs[0] * rt[1]
```

vcrsp.t

Vector cross product

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	rt							1	rs							0	rd						

Syntax

vcrsp.t rd, rs, rt

Description

Performs a full cross-product operation

Instruction performance

Throughput: 3 cycles/instruction
Latency: 9 cycles

Register overlap compatibility

Output register cannot overlap with input registers

Allowed prefixes

- rt: Not supported
- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = rs[1] * rt[2] - rs[2] * rt[1]
rd[1] = rs[2] * rt[0] - rs[0] * rt[2]
rd[2] = rs[0] * rt[1] - rs[1] * rt[0]
```

vqmul.q

Quaternion multiplication

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	rt							1	rs							1	rd						

Syntax

vqmul.q rd, rs, rt

Description

Performs a vector-matrix homogeneous transform (matrix-vector product), with a vector result

Instruction performance

Throughput: 4 cycles/instruction
Latency: 10 cycles

Register overlap compatibility

Output register cannot overlap with input registers

Allowed prefixes

- rt: Not supported
- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = rs[3] * rt[0] - rs[2] * rt[1] + rs[1] * rt[2] + rs[0] * rt[3]
rd[1] = rs[3] * rt[1] + rs[2] * rt[0] + rs[1] * rt[3] - rs[0] * rt[2]
rd[2] = rs[3] * rt[2] + rs[2] * rt[3] - rs[1] * rt[0] + rs[0] * rt[1]
rd[3] = rs[3] * rt[3] - rs[2] * rt[2] - rs[1] * rt[1] - rs[0] * rt[0]
```

vsbn.s

Change exponent scale

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	1	0	rt							0	rs							0	rd						

Syntax

vsbn.s rd, rs, rt

Description

Rescales rs operand to have rt as exponent. This would be equivalent to ldexp(frexp(rs, NULL), rt + 128). If we express the number in its IEEE754 terms, that is, if rs can be expressed as ±m * 2^e, the instruction will replace "e" with the value of rt + 127 mod 256.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = (fpiszero(rs[0]) || fpisnanorinf(rs[0])) ? rs[0] : (rs[0] & 0x807FFFFF) |
(((rt[0] + 127) & 0xFF) << 23)
```

vscl.p

Vector scalar scale

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	0	rt							0	rs							1	rd						

Syntax

vscl.p rd, rs, rt

Description

Scales a vector (element-wise) by an scalar factor

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)
- rt: Not supported

Pseudocode

```
rd[0] = rs[0] * rt[0]
rd[1] = rs[1] * rt[0]
```

vscl.t

Vector scalar scale

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	0	rt							1	rs							0	rd						

Syntax

vscl.t rd, rs, rt

Description

Scales a vector (element-wise) by an scalar factor

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)
- rt: Not supported

Pseudocode

```
rd[0] = rs[0] * rt[0]
rd[1] = rs[1] * rt[0]
rd[2] = rs[2] * rt[0]
```

vscl.q

Vector scalar scale

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	0	rt							1	rs							1	rd						

Syntax

vscl.q rd, rs, rt

Description

Scales a vector (element-wise) by an scalar factor

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)
- rt: Not supported

Pseudocode

```
rd[0] = rs[0] * rt[0]
rd[1] = rs[1] * rt[0]
rd[2] = rs[2] * rt[0]
rd[3] = rs[3] * rt[0]
```

vdot.p

Vector dot product

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	rt							0	rs							1	rd						

Syntax

vdot.p rd, rs, rt

Description

Performs vector floating point dot product

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] * rt[0] + rs[1] * rt[1]
```

vdot.t

Vector dot product

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	rt							1	rs							0	rd						

Syntax

vdot.t rd, rs, rt

Description

Performs vector floating point dot product

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] * rt[0] + rs[1] * rt[1] + rs[2] * rt[2]
```

vdot.q

Vector dot product

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	rt							1	rs							1	rd						

Syntax

vdot.q rd, rs, rt

Description

Performs vector floating point dot product

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] * rt[0] + rs[1] * rt[1] + rs[2] * rt[2] + rs[3] * rt[3]
```

vdet.p

2x2 matrix determinant

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	1	1	0	rt							0	rs							1	rd						

Syntax

vdet.p rd, rs, rt

Description

Performs a 2x2 matrix determinant between two matrix rows

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)
- rt: Not supported

Pseudocode

```
rd[0] = rs[0] * rt[1] - rs[1] * rt[0]
```

vhdp.p

Homogeneous dot product

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	1	0	0	rt							0	rs							1	rd						

Syntax

vhdp.p rd, rs, rt

Description

Performs vector floating point homegeneous dot product

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)
- rs: Not supported

Pseudocode

```
rd[0] = rs[0] * rt[0] + rt[1]
```

vhdp.t

Homogeneous dot product

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	1	0	0	rt							1	rs							0	rd						

Syntax

vhdp.t rd, rs, rt

Description

Performs vector floating point homegeneous dot product

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)
- rs: Not supported

Pseudocode

```
rd[0] = rs[0] * rt[0] + rs[1] * rt[1] + rt[2]
```

vhdp.q

Homogeneous dot product

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	1	0	0	rt							1	rs							1	rd						

Syntax

vhdp.q rd, rs, rt

Description

Performs vector floating point homegeneous dot product

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)
- rs: Not supported

Pseudocode

```
rd[0] = rs[0] * rt[0] + rs[1] * rt[1] + rs[2] * rt[2] + rt[3]
```

vmov.s

Vector copy

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	rs							0	rd						

Syntax

vmov.s rd, rs

Description

Element-wise data copy

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0]
```

vmov.p

Vector copy

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0				rs				1				rd			

Syntax

vmov.p rd, rs

Description

Element-wise data copy

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0]
rd[1] = rs[1]
```

vmov.t

Vector copy

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	rs							0	rd						

Syntax

vmov.t rd, rs

Description

Element-wise data copy

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0]
rd[1] = rs[1]
rd[2] = rs[2]
```

vmov.q

Vector copy

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	rs							1	rd						

Syntax

vmov.q rd, rs

Description

Element-wise data copy

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0]
rd[1] = rs[1]
rd[2] = rs[2]
rd[3] = rs[3]
```

vabs.s

Absolute value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	rs							0	rd						

Syntax

vabs.s rd, rs

Description

Performs element-wise floating point absolute value

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = fabsf(rs[0])
```

vabs.p

Absolute value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	rs							1	rd						

Syntax

vabs.p rd, rs

Description

Performs element-wise floating point absolute value

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = fabsf(rs[0])
rd[1] = fabsf(rs[1])
```

vabs.t

Absolute value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	rs							0	rd						

Syntax

vabs.t rd, rs

Description

Performs element-wise floating point absolute value

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = fabsf(rs[0])
rd[1] = fabsf(rs[1])
rd[2] = fabsf(rs[2])
```

vabs.q

Absolute value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	rs							1	rd						

Syntax

vabs.q rd, rs

Description

Performs element-wise floating point absolute value

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = fabsf(rs[0])
rd[1] = fabsf(rs[1])
rd[2] = fabsf(rs[2])
rd[3] = fabsf(rs[3])
```

vneg.s

Floating point negation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	rs							0	rd						

Syntax

vneg.s rd, rs

Description

Performs element-wise floating point negation

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = -rs[0]
```

vneg.p

Floating point negation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	rs							1	rd						

Syntax

vneg.p rd, rs

Description

Performs element-wise floating point negation

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = -rs[0]
rd[1] = -rs[1]
```

vneg.t

Floating point negation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	rs							0	rd						

Syntax

vneg.t rd, rs

Description

Performs element-wise floating point negation

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = -rs[0]
rd[1] = -rs[1]
rd[2] = -rs[2]
```

vneg.q

Floating point negation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	rs							1	rd						

Syntax

vneg.q rd, rs

Description

Performs element-wise floating point negation

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = -rs[0]
rd[1] = -rs[1]
rd[2] = -rs[2]
rd[3] = -rs[3]
```

vsat0.s

Saturate float to 0..1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	rs							0	rd						

Syntax

vsat0.s rd, rs

Description

Saturates inputs to the [0.0f ... 1.0f] range

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = fminf(fmaxf(rs[0], 0.0f), 1.0f)
```

vsat0.p

Saturate float to 0..1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0				rs				1				rd			

Syntax

vsat0.p rd, rs

Description

Saturates inputs to the [0.0f ... 1.0f] range

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = fminf(fmaxf(rs[0], 0.0f), 1.0f)
rd[1] = fminf(fmaxf(rs[1], 0.0f), 1.0f)
```

vsat0.t

Saturate float to 0..1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1	rs							0	rd						

Syntax

vsat0.t rd, rs

Description

Saturates inputs to the [0.0f ... 1.0f] range

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = fminf(fmaxf(rs[0], 0.0f), 1.0f)
rd[1] = fminf(fmaxf(rs[1], 0.0f), 1.0f)
rd[2] = fminf(fmaxf(rs[2], 0.0f), 1.0f)
```

vsat0.q

Saturate float to 0..1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1	rs							1	rd						

Syntax

vsat0.q rd, rs

Description

Saturates inputs to the [0.0f ... 1.0f] range

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = fminf(fmaxf(rs[0], 0.0f), 1.0f)
rd[1] = fminf(fmaxf(rs[1], 0.0f), 1.0f)
rd[2] = fminf(fmaxf(rs[2], 0.0f), 1.0f)
rd[3] = fminf(fmaxf(rs[3], 0.0f), 1.0f)
```

vsat1.s

Saturate float to -1..1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	rs							0	rd						

Syntax

vsat1.s rd, rs

Description

Saturates inputs to the [-1.0f ... 1.0f] range

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = fminf(fmaxf(rs[0], -1f), 1.0f)
```

vsat1.p

Saturate float to -1..1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	rs							1	rd						

Syntax

vsat1.p rd, rs

Description

Saturates inputs to the [-1.0f ... 1.0f] range

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = fminf(fmaxf(rs[0], -1f), 1.0f)
rd[1] = fminf(fmaxf(rs[1], -1f), 1.0f)
```

vsat1.t

Saturate float to -1..1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	1	0	1	1	rs							0	rd						

Syntax

vsat1.t rd, rs

Description

Saturates inputs to the [-1.0f ... 1.0f] range

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = fminf(fmaxf(rs[0], -1f), 1.0f)
rd[1] = fminf(fmaxf(rs[1], -1f), 1.0f)
rd[2] = fminf(fmaxf(rs[2], -1f), 1.0f)
```

vsat1.q	Saturate float to -1..1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	1	0	1	1	rs							1	rd						

Syntax

vsat1.q rd, rs

Description

Saturates inputs to the [-1.0f ... 1.0f] range

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = fminf(fmaxf(rs[0], -1f), 1.0f)
rd[1] = fminf(fmaxf(rs[1], -1f), 1.0f)
rd[2] = fminf(fmaxf(rs[2], -1f), 1.0f)
rd[3] = fminf(fmaxf(rs[3], -1f), 1.0f)
```

vrcp.s

Reciprocate elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	rs							0	rd						

Syntax

vrcp.s rd, rs

Description

Performs element-wise floating point reciprocal

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3.5 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Relative error is smaller than **6.3e-07**

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = 1.0f / rs[0]
```

vrcp.p

Reciprocate elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	rs							1	rd						

Syntax

vrcp.p rd, rs

Description

Performs element-wise floating point reciprocal

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3.5 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Relative error is smaller than **6.3e-07**

Instruction performance

Throughput: 2 cycles/instruction
Latency: 8 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = 1.0f / rs[0]
rd[1] = 1.0f / rs[1]
```

vrcp.t

Reciprocate elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	1				rs				0				rd			

Syntax

vrcp.t rd, rs

Description

Performs element-wise floating point reciprocal

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3.5 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Relative error is smaller than **6.3e-07**

Instruction performance

Throughput: 3 cycles/instruction
Latency: 9 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = 1.0f / rs[0]
rd[1] = 1.0f / rs[1]
rd[2] = 1.0f / rs[2]
```

vrcp.q

Reciprocate elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14–8	7	6–0
1	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	1	rs	1	rd

Syntax

vrcp.q rd, rs

Description

Performs element-wise floating point reciprocal

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3.5 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Relative error is smaller than **6.3e-07**

Instruction performance

Throughput: 4 cycles/instruction
Latency: 10 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = 1.0f / rs[0]
rd[1] = 1.0f / rs[1]
rd[2] = 1.0f / rs[2]
rd[3] = 1.0f / rs[3]
```

vrsq.s

Reciprocal square root

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	0	0	1	0	rs							0	rd						

Syntax

vrsq.s rd, rs

Description

Performs element-wise floating pointreciprocal square root

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3.5 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Relative error is smaller than **7.3e-07**

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = 1.0f / sqrt(rs[0])
```

vrsq.p

Reciprocal square root

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	0	0	1	0	rs							1	rd						

Syntax

vrsq.p rd, rs

Description

Performs element-wise floating pointreciprocal square root

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3.5 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Relative error is smaller than **7.3e-07**

Instruction performance

Throughput: 2 cycles/instruction
Latency: 8 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = 1.0f / sqrt(rs[0])
rd[1] = 1.0f / sqrt(rs[1])
```

vrsq.t

Reciprocal square root

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	0	0	1	1	rs							0	rd						

Syntax

vrsq.t rd, rs

Description

Performs element-wise floating pointreciprocal square root

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3.5 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Relative error is smaller than **7.3e-07**

Instruction performance

Throughput: 3 cycles/instruction
Latency: 9 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = 1.0f / sqrt(rs[0])
rd[1] = 1.0f / sqrt(rs[1])
rd[2] = 1.0f / sqrt(rs[2])
```

vrsq.q

Reciprocal square root

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	0	0	1	1	rs							1	rd						

Syntax

vrsq.q rd, rs

Description

Performs element-wise floating pointreciprocal square root

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3.5 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Relative error is smaller than **7.3e-07**

Instruction performance

Throughput: 4 cycles/instruction
Latency: 10 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = 1.0f / sqrt(rs[0])
rd[1] = 1.0f / sqrt(rs[1])
rd[2] = 1.0f / sqrt(rs[2])
rd[3] = 1.0f / sqrt(rs[3])
```

vsin.s

Sine function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	rs							0	rd						

Syntax

vsin.s rd, rs

Description

Performs element-wise floating point sin(π/2·rs) operation

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Absolute error is smaller than **4.8e-07**

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = sin(rs[0] * M_PI_2)
```

vsin.p

Sine function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	rs							1	rd						

Syntax

vsin.p rd, rs

Description

Performs element-wise floating point sin(π/2·rs) operation

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Absolute error is smaller than **4.8e-07**

Instruction performance

Throughput: 2 cycles/instruction
Latency: 8 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = sin(rs[0] * M_PI_2)
rd[1] = sin(rs[1] * M_PI_2)
```

vsin.t

Sine function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	0	1	0	1				rs				0				rd			

Syntax

vsin.t rd, rs

Description

Performs element-wise floating point sin(π/2·rs) operation

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Absolute error is smaller than **4.8e-07**

Instruction performance

Throughput: 3 cycles/instruction
Latency: 9 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = sin(rs[0] * M_PI_2)
rd[1] = sin(rs[1] * M_PI_2)
rd[2] = sin(rs[2] * M_PI_2)
```

vsin.q

Sine function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	0	1	0	1	rs							1	rd						

Syntax

vsin.q rd, rs

Description

Performs element-wise floating point sin(π/2·rs) operation

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Absolute error is smaller than **4.8e-07**

Instruction performance

Throughput: 4 cycles/instruction
Latency: 10 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = sin(rs[0] * M_PI_2)
rd[1] = sin(rs[1] * M_PI_2)
rd[2] = sin(rs[2] * M_PI_2)
rd[3] = sin(rs[3] * M_PI_2)
```

vcos.s

Cosine function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	0	1	1	0	rs							0	rd						

Syntax

vcos.s rd, rs

Description

Performs element-wise floating point cos(π/2·rs) operation

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 2.5 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Absolute error is smaller than **4e-07**

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = cos(rs[0] * M_PI_2)
```

vcos.p

Cosine function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	0	1	1	0	rs							1	rd						

Syntax

vcos.p rd, rs

Description

Performs element-wise floating point cos(π/2·rs) operation

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 2.5 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Absolute error is smaller than **4e-07**

Instruction performance

Throughput: 2 cycles/instruction
Latency: 8 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = cos(rs[0] * M_PI_2)
rd[1] = cos(rs[1] * M_PI_2)
```

vcos.t

Cosine function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	0	1	1	1	rs							0	rd						

Syntax

vcos.t rd, rs

Description

Performs element-wise floating point cos(π/2·rs) operation

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 2.5 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Absolute error is smaller than **4e-07**

Instruction performance

Throughput: 3 cycles/instruction
Latency: 9 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = cos(rs[0] * M_PI_2)
rd[1] = cos(rs[1] * M_PI_2)
rd[2] = cos(rs[2] * M_PI_2)
```

vcos.q

Cosine function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	0	1	1	1				rs				1				rd			

Syntax

vcos.q rd, rs

Description

Performs element-wise floating point cos(π/2·rs) operation

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 2.5 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Absolute error is smaller than **4e-07**

Instruction performance

Throughput: 4 cycles/instruction
Latency: 10 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = cos(rs[0] * M_PI_2)
rd[1] = cos(rs[1] * M_PI_2)
rd[2] = cos(rs[2] * M_PI_2)
rd[3] = cos(rs[3] * M_PI_2)
```

vexp2.s

Base-2 exponentiation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	rs							0	rd						

Syntax

vexp2.s rd, rs

Description

Performs element-wise floating point exp2(rs) operation

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details. Inputs larger than 127 result in overflow (cannot represent over 2^127)

Relative error is smaller than **7.2e-07**

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = (rs[0] >= 128) ? INFINITY : (rs[0] <= -127) ? 0.0f : exp2(rs[0])
```

vexp2.p

Base-2 exponentiation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0				rs				1				rd			

Syntax

vexp2.p rd, rs

Description

Performs element-wise floating point exp2(rs) operation

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details. Inputs larger than 127 result in overflow (cannot represent over 2^127)

Relative error is smaller than **7.2e-07**

Instruction performance

Throughput: 2 cycles/instruction
Latency: 8 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = (rs[0] >= 128) ? INFINITY : (rs[0] <= -127) ? 0.0f : exp2(rs[0])
rd[1] = (rs[1] >= 128) ? INFINITY : (rs[1] <= -127) ? 0.0f : exp2(rs[1])
```

vexp2.t

Base-2 exponentiation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	1	0	0	1	rs							0	rd						

Syntax

vexp2.t rd, rs

Description

Performs element-wise floating point exp2(rs) operation

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details. Inputs larger than 127 result in overflow (cannot represent over 2^127)

Relative error is smaller than **7.2e-07**

Instruction performance

Throughput: 3 cycles/instruction
Latency: 9 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = (rs[0] >= 128) ? INFINITY : (rs[0] <= -127) ? 0.0f : exp2(rs[0])
rd[1] = (rs[1] >= 128) ? INFINITY : (rs[1] <= -127) ? 0.0f : exp2(rs[1])
rd[2] = (rs[2] >= 128) ? INFINITY : (rs[2] <= -127) ? 0.0f : exp2(rs[2])
```

vexp2.q

Base-2 exponentiation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	1	0	0	1	rs							1	rd						

Syntax

vexp2.q rd, rs

Description

Performs element-wise floating point exp2(rs) operation

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details. Inputs larger than 127 result in overflow (cannot represent over 2^127)

Relative error is smaller than **7.2e-07**

Instruction performance

Throughput: 4 cycles/instruction
Latency: 10 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = (rs[0] >= 128) ? INFINITY : (rs[0] <= -127) ? 0.0f : exp2(rs[0])
rd[1] = (rs[1] >= 128) ? INFINITY : (rs[1] <= -127) ? 0.0f : exp2(rs[1])
rd[2] = (rs[2] >= 128) ? INFINITY : (rs[2] <= -127) ? 0.0f : exp2(rs[2])
rd[3] = (rs[3] >= 128) ? INFINITY : (rs[3] <= -127) ? 0.0f : exp2(rs[3])
```

vlog2.s

Base-2 logarithm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	1	0	1	0	rs							0	rd						

Syntax

vlog2.s rd, rs

Description

Performs element-wise floating point log2(rs) operation

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. Accuracy varies greatly depending on the input value. Please refer to psp-tests/accuracy for more details.

Absolute error is smaller than **3e-05**

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = log2(rs[0])
```

vlog2.p

Base-2 logarithm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	1	0	1	0	rs							1	rd						

Syntax

vlog2.p rd, rs

Description

Performs element-wise floating point log2(rs) operation

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. Accuracy varies greatly depending on the input value. Please refer to psp-tests/accuracy for more details.

Absolute error is smaller than **3e-05**

Instruction performance

Throughput: 2 cycles/instruction
Latency: 8 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = log2(rs[0])
rd[1] = log2(rs[1])
```

vlog2.t

Base-2 logarithm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	1	0	1	1	rs							0	rd						

Syntax

vlog2.t rd, rs

Description

Performs element-wise floating point log2(rs) operation

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. Accuracy varies greatly depending on the input value. Please refer to psp-tests/accuracy for more details.

Absolute error is smaller than **3e-05**

Instruction performance

Throughput: 3 cycles/instruction
Latency: 9 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = log2(rs[0])
rd[1] = log2(rs[1])
rd[2] = log2(rs[2])
```

vlog2.q

Base-2 logarithm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	1	0	1	1	rs							1	rd						

Syntax

vlog2.q rd, rs

Description

Performs element-wise floating point log2(rs) operation

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. Accuracy varies greatly depending on the input value. Please refer to psp-tests/accuracy for more details.

Absolute error is smaller than **3e-05**

Instruction performance

Throughput: 4 cycles/instruction
Latency: 10 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = log2(rs[0])
rd[1] = log2(rs[1])
rd[2] = log2(rs[2])
rd[3] = log2(rs[3])
```

vlgb.s

LogB calculation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	1	0	1	1	1	0	rs							0	rd						

Syntax

vlgb.s rd, rs

Description

Performs element-wise logB() calculation

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = logbf(rs[0])
```

vsbz.s

Reset exponent scale

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	1	0	1	1	0	0				rs				0				rd			

Syntax

vsbz.s rd, rs

Description

Rescales rs operand to have zero as exponent, so that it is reduced to the [1.0, 2.0) interval. This is essentially equivalent to the vsbn instruction with rt=0.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = (fpiszero(rs[0]) || fpisnan(rs[0])) ? rs[0] : (rs[0] & 0x007FFFFF) |
0x3F800000
```

vwbn.s

Floating point modulus

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	1	imval								0	rs							0	rd						

Syntax

vwbn.s rd, rs, scale

Description

TODO: Document this better. Performs some sort of modulus operation.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = ivwbn(rs[0], imval)
```

Used functions

```
uint32_t ivwbn(uint32_t arg, unsigned imm) {
  uint32_t sbit = arg & 0x80000000;
  uint32_t exp = (arg >> 23) & 0xff;
  uint32_t m = (arg & 0x007FFFFF) | 0x800000;
  if (!exp || exp == 0xff)
    return arg | (imm << 23);

  if (imm > exp) {
    unsigned sh = (imm - exp) & 0xf;
    m >>= sh;
  } else {
    unsigned sh = (exp - imm) & 0xf;
    m <<= sh;
  }
  return sbit | (m & 0x7FFFFF) | (imm << 23);
}
```

vsqrt.s

Square root

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	1	1	0	0	rs							0	rd						

Syntax

vsqrt.s rd, rs

Description

Performs element-wise floating point aproximate square root

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Relative error is smaller than **7.1e-07**

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = sqrt(rs[0])
```

vsqrt.p

Square root

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	1	1	0	0	rs							1	rd						

Syntax

vsqrt.p rd, rs

Description

Performs element-wise floating point aproximate square root

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Relative error is smaller than **7.1e-07**

Instruction performance

Throughput: 2 cycles/instruction
Latency: 8 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = sqrt(rs[0])
rd[1] = sqrt(rs[1])
```

vsqrt.t

Square root

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	1	1	0	1	rs							0	rd						

Syntax

vsqrt.t rd, rs

Description

Performs element-wise floating point aproximate square root

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Relative error is smaller than **7.1e-07**

Instruction performance

Throughput: 3 cycles/instruction
Latency: 9 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = sqrt(rs[0])
rd[1] = sqrt(rs[1])
rd[2] = sqrt(rs[2])
```

vsqrt.q

Square root

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	1	1	0	1	rs							1	rd						

Syntax

vsqrt.q rd, rs

Description

Performs element-wise floating point aproximate square root

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Relative error is smaller than **7.1e-07**

Instruction performance

Throughput: 4 cycles/instruction
Latency: 10 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = sqrt(rs[0])
rd[1] = sqrt(rs[1])
rd[2] = sqrt(rs[2])
rd[3] = sqrt(rs[3])
```

vasin.s

Arc sine function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	1	1	1	0	rs							0	rd						

Syntax

vasin.s rd, rs

Description

Performs element-wise floating point asin(rs)·2/π operation

Accuracy

This function provides an approximate value. The precision seems quite good for arguments between -0.5 and 0.5 (around 2.5e-7), but it becomes very inaccurate outside of this range, as it approaches +/-1. Please refer to psp-tests/accuracy for more details.

Absolute error is smaller than **0.02**

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = asin(rs[0]) / M_PI_2
```

vasin.p

Arc sine function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	1	1	1	0	rs							1	rd						

Syntax

vasin.p rd, rs

Description

Performs element-wise floating point asin(rs)·2/π operation

Accuracy

This function provides an approximate value. The precision seems quite good for arguments between -0.5 and 0.5 (around 2.5e-7), but it becomes very inaccurate outside of this range, as it approaches +/-1. Please refer to psp-tests/accuracy for more details.

Absolute error is smaller than **0.02**

Instruction performance

Throughput: 2 cycles/instruction
Latency: 8 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = asin(rs[0]) / M_PI_2
rd[1] = asin(rs[1]) / M_PI_2
```

vasin.t

Arc sine function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	1	1	1	1				rs				0				rd			

Syntax

vasin.t rd, rs

Description

Performs element-wise floating point asin(rs)·2/π operation

Accuracy

This function provides an approximate value. The precision seems quite good for arguments between -0.5 and 0.5 (around 2.5e-7), but it becomes very inaccurate outside of this range, as it approaches +/-1. Please refer to psp-tests/accuracy for more details.

Absolute error is smaller than **0.02**

Instruction performance

Throughput: 3 cycles/instruction
Latency: 9 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = asin(rs[0]) / M_PI_2
rd[1] = asin(rs[1]) / M_PI_2
rd[2] = asin(rs[2]) / M_PI_2
```

vasin.q

Arc sine function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	1	1	1	1	rs							1	rd						

Syntax

vasin.q rd, rs

Description

Performs element-wise floating point asin(rs)·2/π operation

Accuracy

This function provides an approximate value. The precision seems quite good for arguments between -0.5 and 0.5 (around 2.5e-7), but it becomes very inaccurate outside of this range, as it approaches +/-1. Please refer to psp-tests/accuracy for more details.

Absolute error is smaller than **0.02**

Instruction performance

Throughput: 4 cycles/instruction
Latency: 10 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = asin(rs[0]) / M_PI_2
rd[1] = asin(rs[1]) / M_PI_2
rd[2] = asin(rs[2]) / M_PI_2
rd[3] = asin(rs[3]) / M_PI_2
```

vnrcp.s

Negative reciprocal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0				rs				0				rd			

Syntax

vnrcp.s rd, rs

Description

Performs element-wise floating point negated reciprocal

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3.5 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Relative error is smaller than **6.3e-07**

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rd: Full support (masking and saturation)
- rs: Not supported

Pseudocode

```
rd[0] = -1f / rs[0]
```

vnrcp.p

Negative reciprocal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	rs							1	rd						

Syntax

vnrcp.p rd, rs

Description

Performs element-wise floating point negated reciprocal

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3.5 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Relative error is smaller than **6.3e-07**

Instruction performance

Throughput: 2 cycles/instruction
Latency: 8 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = -1f / rs[0]
rd[1] = -1f / rs[1]
```

vnrcp.t

Negative reciprocal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	1	0	0	0	1	rs							0	rd						

Syntax

vnrcp.t rd, rs

Description

Performs element-wise floating point negated reciprocal

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3.5 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Relative error is smaller than **6.3e-07**

Instruction performance

Throughput: 3 cycles/instruction
Latency: 9 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = -1f / rs[0]
rd[1] = -1f / rs[1]
rd[2] = -1f / rs[2]
```

vnrcp.q

Negative reciprocal

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	1	0	0	0	1	rs							1	rd						

Syntax

vnrcp.q rd, rs

Description

Performs element-wise floating point negated reciprocal

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3.5 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Relative error is smaller than **6.3e-07**

Instruction performance

Throughput: 4 cycles/instruction
Latency: 10 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = -1f / rs[0]
rd[1] = -1f / rs[1]
rd[2] = -1f / rs[2]
rd[3] = -1f / rs[3]
```

vnsin.s

Negative sine function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	1	0	1	0	0	rs							0	rd						

Syntax

vnsin.s rd, rs

Description

Performs element-wise floating point -sin(π/2·rs) operation

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Absolute error is smaller than **4.8e-07**

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rd: Full support (masking and saturation)
- rs: Not supported

Pseudocode

```
rd[0] = -sin(rs[0] * M_PI_2)
```

vnsin.p

Negative sine function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	1	0	1	0	0	rs							1	rd						

Syntax

vnsin.p rd, rs

Description

Performs element-wise floating point -sin(π/2·rs) operation

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Absolute error is smaller than **4.8e-07**

Instruction performance

Throughput: 2 cycles/instruction
Latency: 8 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = -sin(rs[0] * M_PI_2)
rd[1] = -sin(rs[1] * M_PI_2)
```

vnsin.t

Negative sine function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	1	0	1	0	1	rs							0	rd						

Syntax

vnsin.t rd, rs

Description

Performs element-wise floating point -sin(π/2·rs) operation

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Absolute error is smaller than **4.8e-07**

Instruction performance

Throughput: 3 cycles/instruction
Latency: 9 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = -sin(rs[0] * M_PI_2)
rd[1] = -sin(rs[1] * M_PI_2)
rd[2] = -sin(rs[2] * M_PI_2)
```

vnsin.q

Negative sine function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	1	0	1	0	1				rs				1				rd			

Syntax

vnsin.q rd, rs

Description

Performs element-wise floating point -sin(π/2·rs) operation

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details.

Absolute error is smaller than **4.8e-07**

Instruction performance

Throughput: 4 cycles/instruction
Latency: 10 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = -sin(rs[0] * M_PI_2)
rd[1] = -sin(rs[1] * M_PI_2)
rd[2] = -sin(rs[2] * M_PI_2)
rd[3] = -sin(rs[3] * M_PI_2)
```

vrexp2.s

Base-2 negative exponentiation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	1	1	0	0	0	rs							0	rd						

Syntax

vrexp2.s rd, rs

Description

Performs element-wise floating point 1/exp2(rs) operation (equivalent to exp2(-rs))

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details. Inputs larger than 127 result in overflow (cannot represent over 2^127)

Relative error is smaller than **7.2e-07**

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rd: Full support (masking and saturation)
- rs: Not supported

Pseudocode

```
rd[0] = (rs[0] >= 127) ? 0.0f : (rs[0] <= -128) ? INFINITY : exp2(-rs[0])
```

vrexp2.p

Base-2 negative exponentiation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	1	1	0	0	0	rs							1	rd						

Syntax

vrexp2.p rd, rs

Description

Performs element-wise floating point 1/exp2(rs) operation (equivalent to exp2(-rs))

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details. Inputs larger than 127 result in overflow (cannot represent over 2^127)

Relative error is smaller than **7.2e-07**

Instruction performance

Throughput: 2 cycles/instruction
Latency: 8 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = (rs[0] >= 127) ? 0.0f : (rs[0] <= -128) ? INFINITY : exp2(-rs[0])
rd[1] = (rs[1] >= 127) ? 0.0f : (rs[1] <= -128) ? INFINITY : exp2(-rs[1])
```

vrexp2.t

Base-2 negative exponentiation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	1	1	0	0	1	rs							0	rd						

Syntax

vrexp2.t rd, rs

Description

Performs element-wise floating point 1/exp2(rs) operation (equivalent to exp2(-rs))

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details. Inputs larger than 127 result in overflow (cannot represent over 2^127)

Relative error is smaller than **7.2e-07**

Instruction performance

Throughput: 3 cycles/instruction
Latency: 9 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = (rs[0] >= 127) ? 0.0f : (rs[0] <= -128) ? INFINITY : exp2(-rs[0])
rd[1] = (rs[1] >= 127) ? 0.0f : (rs[1] <= -128) ? INFINITY : exp2(-rs[1])
rd[2] = (rs[2] >= 127) ? 0.0f : (rs[2] <= -128) ? INFINITY : exp2(-rs[2])
```

vrexp2.q

Base-2 negative exponentiation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	1	1	0	0	1				rs				1				rd			

Syntax

vrexp2.q rd, rs

Description

Performs element-wise floating point 1/exp2(rs) operation (equivalent to exp2(-rs))

Accuracy

This function provides an approximate value, with lower accuracy to what FP32 IEEE754 numbers can represent. The lowest 3 mantissa bits seem to be innacurate. Please refer to psp-tests/accuracy for more details. Inputs larger than 127 result in overflow (cannot represent over 2^127)

Relative error is smaller than **7.2e-07**

Instruction performance

Throughput: 4 cycles/instruction
Latency: 10 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = (rs[0] >= 127) ? 0.0f : (rs[0] <= -128) ? INFINITY : exp2(-rs[0])
rd[1] = (rs[1] >= 127) ? 0.0f : (rs[1] <= -128) ? INFINITY : exp2(-rs[1])
rd[2] = (rs[2] >= 127) ? 0.0f : (rs[2] <= -128) ? INFINITY : exp2(-rs[2])
rd[3] = (rs[3] >= 127) ? 0.0f : (rs[3] <= -128) ? INFINITY : exp2(-rs[3])
```

vsrt1.q

Element min-sort pass #1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1	rs							1	rd						

Syntax

vsrt1.q rd, rs

Description

Performs a min() sorting step between elements pairs 0-1 and 2-3, shuffling them depending on their values.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Full support (masking and saturation)
- rs: Not supported

Pseudocode

```
rd[0] = fminf(rs[0], rs[1])
rd[1] = fmaxf(rs[0], rs[1])
rd[2] = fminf(rs[2], rs[3])
rd[3] = fmaxf(rs[2], rs[3])
```

vsrt2.q

Element min-sort pass #2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	0	0	0	1	1	rs							1	rd						

Syntax

vsrt2.q rd, rs

Description

Performs a min() sorting step between elements pairs 3-0 and 1-2, shuffling them depending on their values.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Full support (masking and saturation)
- rs: Not supported

Pseudocode

```
rd[0] = fminf(rs[0], rs[3])
rd[1] = fminf(rs[1], rs[2])
rd[2] = fmaxf(rs[1], rs[2])
rd[3] = fmaxf(rs[0], rs[3])
```

vsrt3.q

Element max-sort pass #1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	1	0	0	0	1	rs							1	rd						

Syntax

vsrt3.q rd, rs

Description

Performs a max() sorting step between elements pairs 0-1 and 2-3, shuffling them depending on their values.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

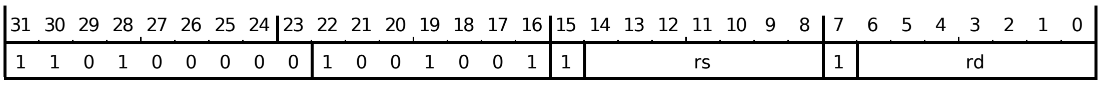
- rd: Full support (masking and saturation)
- rs: Not supported

Pseudocode

```
rd[0] = fmaxf(rs[0], rs[1])
rd[1] = fminf(rs[0], rs[1])
rd[2] = fmaxf(rs[2], rs[3])
rd[3] = fminf(rs[2], rs[3])
```

vsrt4.q

Element max-sort pass #2



Syntax

vsrt4.q rd, rs

Description

Performs a max() sorting step between elements pairs 3-0 and 1-2, shuffling them depending on their values.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Full support (masking and saturation)
- rs: Not supported

Pseudocode

```
rd[0] = fmaxf(rs[0], rs[3])
rd[1] = fmaxf(rs[1], rs[2])
rd[2] = fminf(rs[1], rs[2])
rd[3] = fminf(rs[0], rs[3])
```

vbfy1.p

Butterfly function #1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	rs							1	rd						

Syntax

vbfy1.p rd, rs

Description

Performs a `butterfly` operation between the input elements.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Full support (masking and saturation)
- rs: Not supported

Pseudocode

```
rd[0] = rs[0] + rs[1]
rd[1] = rs[0] - rs[1]
```

vbfy1.q

Butterfly function #1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	rs							1	rd						

Syntax

vbfy1.q rd, rs

Description

Performs a \`butterfly\` operation between the input elements.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Full support (masking and saturation)
- rs: Not supported

Pseudocode

```
rd[0] = rs[0] + rs[1]
rd[1] = rs[0] - rs[1]
rd[2] = rs[2] + rs[3]
rd[3] = rs[2] - rs[3]
```

vbfy2.q

Butterfly function #2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	0	0	1	1	1	rs							1	rd						

Syntax

vbfy2.q rd, rs

Description

Performs a `butterfly` operation between the input elements.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Full support (masking and saturation)
- rs: Not supported

Pseudocode

```
rd[0] = rs[0] + rs[2]
rd[1] = rs[1] + rs[3]
rd[2] = rs[0] - rs[2]
rd[3] = rs[1] - rs[3]
```

vsgn.s

Sign function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	rs							0	rd						

Syntax

vsgn.s rd, rs

Description

Performs element-wise floating point sign(rs) operation. This function returns -1, 0 or 1 depending on whether the input is negative zero or positive respectively.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] < 0 ? -1f : rs[0] > 0 ? 1.0f : 0.0f
```

vsgn.p

Sign function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	rs							1	rd						

Syntax

vsgn.p rd, rs

Description

Performs element-wise floating point sign(rs) operation. This function returns -1, 0 or 1 depending on whether the input is negative zero or positive respectively.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] < 0 ? -1f : rs[0] > 0 ? 1.0f : 0.0f
rd[1] = rs[1] < 0 ? -1f : rs[1] > 0 ? 1.0f : 0.0f
```

vsgn.t

Sign function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	1	0	1	0	1				rs				0				rd			

Syntax

vsgn.t rd, rs

Description

Performs element-wise floating point sign(rs) operation. This function returns -1, 0 or 1 depending on whether the input is negative zero or positive respectively.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] < 0 ? -1f : rs[0] > 0 ? 1.0f : 0.0f
rd[1] = rs[1] < 0 ? -1f : rs[1] > 0 ? 1.0f : 0.0f
rd[2] = rs[2] < 0 ? -1f : rs[2] > 0 ? 1.0f : 0.0f
```

vsgn.q

Sign function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	1	0	1	0	1	rs							1	rd						

Syntax

vsgn.q rd, rs

Description

Performs element-wise floating point sign(rs) operation. This function returns -1, 0 or 1 depending on whether the input is negative zero or positive respectively.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] < 0 ? -1f : rs[0] > 0 ? 1.0f : 0.0f
rd[1] = rs[1] < 0 ? -1f : rs[1] > 0 ? 1.0f : 0.0f
rd[2] = rs[2] < 0 ? -1f : rs[2] > 0 ? 1.0f : 0.0f
rd[3] = rs[3] < 0 ? -1f : rs[3] > 0 ? 1.0f : 0.0f
```

vocp.s

One complement function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	rs							0	rd						

Syntax

vocp.s rd, rs

Description

Performs element-wise one's complement (1.0f - x)

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Full support (masking and saturation)
- rs: Not supported

Pseudocode

```
rd[0] = 1 - rs[0]
```

vocp.p — One complement function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	rs							1	rd						

Syntax

vocp.p rd, rs

Description

Performs element-wise one's complement (1.0f - x)

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Full support (masking and saturation)
- rs: Not supported

Pseudocode

```
rd[0] = 1 - rs[0]
rd[1] = 1 - rs[1]
```

vocp.t

One complement function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	0	1	0	0	1	rs							0	rd						

Syntax

vocp.t rd, rs

Description

Performs element-wise one's complement (1.0f - x)

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Full support (masking and saturation)
- rs: Not supported

Pseudocode

```
rd[0] = 1 - rs[0]
rd[1] = 1 - rs[1]
rd[2] = 1 - rs[2]
```

vocp.q One complement function

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	0	1	0	0	1	rs							1	rd						

Syntax

vocp.q rd, rs

Description

Performs element-wise one's complement (1.0f - x)

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Full support (masking and saturation)
- rs: Not supported

Pseudocode

```
rd[0] = 1 - rs[0]
rd[1] = 1 - rs[1]
rd[2] = 1 - rs[2]
rd[3] = 1 - rs[3]
```

vi2f.s

Integer to float with scaling

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	1	0	0	imval					0	rs							0	rd						

Syntax

vi2f.s rd, rs, scale

Description

Performs element-wise integer to float conversion with optional scaling factor. The integer is divided by 2^scale after the conversion.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = ldexp(rs[0], -imval)
```

vi2f.p

Integer to float with scaling

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	1	0	0	imval					0	rs							1	rd						

Syntax

vi2f.p rd, rs, scale

Description

Performs element-wise integer to float conversion with optional scaling factor. The integer is divided by 2^scale after the conversion.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = ldexp(rs[0], -imval)
rd[1] = ldexp(rs[1], -imval)
```

vi2f.t Integer to float with scaling

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	1	0	0	imval					1	rs							0	rd						

Syntax

vi2f.t rd, rs, scale

Description

Performs element-wise integer to float conversion with optional scaling factor. The integer is divided by 2^scale after the conversion.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = ldexp(rs[0], -imval)
rd[1] = ldexp(rs[1], -imval)
rd[2] = ldexp(rs[2], -imval)
```

vi2f.q

Integer to float with scaling

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	1	0	0	imval					1	rs							1	rd						

Syntax

vi2f.q rd, rs, scale

Description

Performs element-wise integer to float conversion with optional scaling factor. The integer is divided by 2^scale after the conversion.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = ldexp(rs[0], -imval)
rd[1] = ldexp(rs[1], -imval)
rd[2] = ldexp(rs[2], -imval)
rd[3] = ldexp(rs[3], -imval)
```

vf2in.s Float to integer round-to-nearest with scaling

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	0	0	0	imval					0	rs							0	rd						

Syntax

vf2in.s rd, rs, scale

Description

Performs element-wise float to integer conversion with optional scaling factor, rounding to the nearest integer

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = rintf(rs[0] * pow(2.0f, imval))
```

vf2in.p Float to integer round-to-nearest with scaling

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	0	0	0	imval					0	rs							1	rd						

Syntax

vf2in.p rd, rs, scale

Description

Performs element-wise float to integer conversion with optional scaling factor, rounding to the nearest integer

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = rintf(rs[0] * pow(2.0f, imval))
rd[1] = rintf(rs[1] * pow(2.0f, imval))
```

vf2in.t Float to integer round-to-nearest with scaling

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	0	0	0	imval					1	rs							0	rd						

Syntax

vf2in.t rd, rs, scale

Description

Performs element-wise float to integer conversion with optional scaling factor, rounding to the nearest integer

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = rintf(rs[0] * pow(2.0f, imval))
rd[1] = rintf(rs[1] * pow(2.0f, imval))
rd[2] = rintf(rs[2] * pow(2.0f, imval))
```

vf2in.q Float to integer round-to-nearest with scaling

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	0	0	0	imval					1	rs							1	rd						

Syntax

vf2in.q rd, rs, scale

Description

Performs element-wise float to integer conversion with optional scaling factor, rounding to the nearest integer

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = rintf(rs[0] * pow(2.0f, imval))
rd[1] = rintf(rs[1] * pow(2.0f, imval))
rd[2] = rintf(rs[2] * pow(2.0f, imval))
rd[3] = rintf(rs[3] * pow(2.0f, imval))
```

vf2iz.s

Float to integer truncation with scaling

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	0	0	1	imval					0	rs							0	rd						

Syntax

vf2iz.s rd, rs, scale

Description

Performs element-wise float to integer conversion with optional scaling factor, truncating the decimal argument (that is, rounding towards zero)

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = truncf(rs[0] * pow(2.0f, imval))
```

vf2iz.p Float to integer truncation with scaling

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	0	0	1	imval					0	rs							1	rd						

Syntax

vf2iz.p rd, rs, scale

Description

Performs element-wise float to integer conversion with optional scaling factor, truncating the decimal argument (that is, rounding towards zero)

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = truncf(rs[0] * pow(2.0f, imval))
rd[1] = truncf(rs[1] * pow(2.0f, imval))
```

vf2iz.t

Float to integer truncation with scaling

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	0	0	1	imval					1	rs							0	rd						

Syntax

vf2iz.t rd, rs, scale

Description

Performs element-wise float to integer conversion with optional scaling factor, truncating the decimal argument (that is, rounding towards zero)

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = truncf(rs[0] * pow(2.0f, imval))
rd[1] = truncf(rs[1] * pow(2.0f, imval))
rd[2] = truncf(rs[2] * pow(2.0f, imval))
```

vf2iz.q

Float to integer truncation with scaling

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	0	0	1	imval					1	rs							1	rd						

Syntax

vf2iz.q rd, rs, scale

Description

Performs element-wise float to integer conversion with optional scaling factor, truncating the decimal argument (that is, rounding towards zero)

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = truncf(rs[0] * pow(2.0f, imval))
rd[1] = truncf(rs[1] * pow(2.0f, imval))
rd[2] = truncf(rs[2] * pow(2.0f, imval))
rd[3] = truncf(rs[3] * pow(2.0f, imval))
```

vf2iu.s Float to integer round-up with scaling

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	0	1	0	imval					0	rs							0	rd						

Syntax

vf2iu.s rd, rs, scale

Description

Performs element-wise float to integer conversion with optional scaling factor, rounding up (that is, towards the next, equal or greater, integer value)

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = ceilf(rs[0] * pow(2.0f, imval))
```

vf2iu.p

Float to integer round-up with scaling

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	0	1	0	imval					0	rs							1	rd						

Syntax

vf2iu.p rd, rs, scale

Description

Performs element-wise float to integer conversion with optional scaling factor, rounding up (that is, towards the next, equal or greater, integer value)

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = ceilf(rs[0] * pow(2.0f, imval))
rd[1] = ceilf(rs[1] * pow(2.0f, imval))
```

vf2iu.t

Float to integer round-up with scaling

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	0	1	0	imval					1	rs							0	rd						

Syntax

vf2iu.t rd, rs, scale

Description

Performs element-wise float to integer conversion with optional scaling factor, rounding up (that is, towards the next, equal or greater, integer value)

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = ceilf(rs[0] * pow(2.0f, imval))
rd[1] = ceilf(rs[1] * pow(2.0f, imval))
rd[2] = ceilf(rs[2] * pow(2.0f, imval))
```

vf2iu.q

Float to integer round-up with scaling

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	0	1	0	imval					1	rs							1	rd						

Syntax

vf2iu.q rd, rs, scale

Description

Performs element-wise float to integer conversion with optional scaling factor, rounding up (that is, towards the next, equal or greater, integer value)

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = ceilf(rs[0] * pow(2.0f, imval))
rd[1] = ceilf(rs[1] * pow(2.0f, imval))
rd[2] = ceilf(rs[2] * pow(2.0f, imval))
rd[3] = ceilf(rs[3] * pow(2.0f, imval))
```

vf2id.s Float to integer round-down with scaling

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	0	1	1	imval					0	rs							0	rd						

Syntax

vf2id.s rd, rs, scale

Description

Performs element-wise float to integer conversion with optional scaling factor, rounding down (that is, towards the previous, equal or smaller, integer value)

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = floorf(rs[0] * pow(2.0f, imval))
```

vf2id.p

Float to integer round-down with scaling

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	0	1	1	imval					0	rs							1	rd						

Syntax

vf2id.p rd, rs, scale

Description

Performs element-wise float to integer conversion with optional scaling factor, rounding down (that is, towards the previous, equal or smaller, integer value)

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = floorf(rs[0] * pow(2.0f, imval))
rd[1] = floorf(rs[1] * pow(2.0f, imval))
```

vf2id.t

Float to integer round-down with scaling

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	0	1	1	imval					1	rs							0	rd						

Syntax

vf2id.t rd, rs, scale

Description

Performs element-wise float to integer conversion with optional scaling factor, rounding down (that is, towards the previous, equal or smaller, integer value)

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

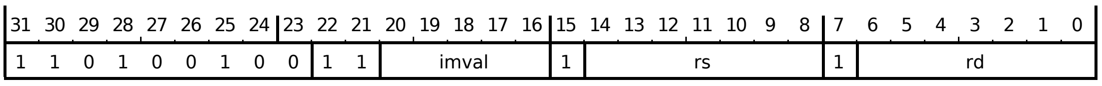
Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = floorf(rs[0] * pow(2.0f, imval))
rd[1] = floorf(rs[1] * pow(2.0f, imval))
rd[2] = floorf(rs[2] * pow(2.0f, imval))
```

vf2id.q Float to integer round-down with scaling



Syntax

vf2id.q rd, rs, scale

Description

Performs element-wise float to integer conversion with optional scaling factor, rounding down (that is, towards the previous, equal or smaller, integer value)

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = floorf(rs[0] * pow(2.0f, imval))
rd[1] = floorf(rs[1] * pow(2.0f, imval))
rd[2] = floorf(rs[2] * pow(2.0f, imval))
rd[3] = floorf(rs[3] * pow(2.0f, imval))
```

vrot.p

Rotation matrix row calculation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	imval					0	rs							1	rd						

Syntax

vrot.p rd, rs, imm5

Description

Calculates a rotation matrix row, given an angle argument

Accuracy

This function provides the same accuracy as its vsin/vcos counterparts.

Absolute error is smaller than **4.8e-07**

Instruction performance

Throughput: 2 cycles/instruction
Latency: 8 cycles

Register overlap compatibility

Output register cannot overlap with input registers

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = ivrot(0, rs[0], imval)
rd[1] = ivrot(1, rs[0], imval)
```

Used functions

```
float ivrot(unsigned elem, float arg, unsigned imm) {
  unsigned cl = imm & 3;
  unsigned sl = (imm >> 2) & 3;
  float s = sin(arg * M_PI_2);
  float c = cos(arg * M_PI_2);
  if (imm & 0x10)
    s = -s;

  // Special case where all elements are sine but one
  if (cl == sl)
    return (elem == cl) ? c : s;

  // Each bit pair indicates the position
  return (elem == cl) ? c :
         (elem == sl) ? s : 0.0f;
}
```

vrot.t

Rotation matrix row calculation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	imval					1	rs							0	rd						

Syntax

vrot.t rd, rs, imm5

Description

Calculates a rotation matrix row, given an angle argument

Accuracy

This function provides the same accuracy as its vsin/vcos counterparts.

Absolute error is smaller than **4.8e-07**

Instruction performance

Throughput: 2 cycles/instruction
Latency: 8 cycles

Register overlap compatibility

Output register cannot overlap with input registers

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = ivrot(0, rs[0], imval)
rd[1] = ivrot(1, rs[0], imval)
rd[2] = ivrot(2, rs[0], imval)
```

Used functions

```
float ivrot(unsigned elem, float arg, unsigned imm) {
  unsigned cl = imm & 3;
  unsigned sl = (imm >> 2) & 3;
  float s = sin(arg * M_PI_2);
  float c = cos(arg * M_PI_2);
  if (imm & 0x10)
    s = -s;

  // Special case where all elements are sine but one
  if (cl == sl)
    return (elem == cl) ? c : s;

  // Each bit pair indicates the position
  return (elem == cl) ? c :
         (elem == sl) ? s : 0.0f;
}
```

vrot.q

Rotation matrix row calculation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	imval					1	rs							1	rd						

Syntax

vrot.q rd, rs, imm5

Description

Calculates a rotation matrix row, given an angle argument

Accuracy

This function provides the same accuracy as its vsin/vcos counterparts.

Absolute error is smaller than **4.8e-07**

Instruction performance

Throughput: 2 cycles/instruction
Latency: 8 cycles

Register overlap compatibility

Output register cannot overlap with input registers

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = ivrot(0, rs[0], imval)
rd[1] = ivrot(1, rs[0], imval)
rd[2] = ivrot(2, rs[0], imval)
rd[3] = ivrot(3, rs[0], imval)
```

Used functions

```
float ivrot(unsigned elem, float arg, unsigned imm) {
  unsigned cl = imm & 3;
  unsigned sl = (imm >> 2) & 3;
  float s = sin(arg * M_PI_2);
  float c = cos(arg * M_PI_2);
  if (imm & 0x10)
    s = -s;

  // Special case where all elements are sine but one
  if (cl == sl)
    return (elem == cl) ? c : s;

  // Each bit pair indicates the position
  return (elem == cl) ? c :
         (elem == sl) ? s : 0.0f;
}
```

vsocp.s

One complement with saturation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	0	1	0	1	0	rs							0	rd						

Syntax

vsocp.s rd, rs

Description

Performs element-wise one's complement (1.0f - x) with saturation to [0.0f ... 1.0f]

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = fminf(fmaxf(1.0f - rs[0], 0.0f), 1.0f)
rd[1] = fminf(fmaxf(rs[0], 0.0f), 1.0f)
```

vsocp.p

One complement with saturation

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	0	1	0	1	0	rs							1	rd						

Syntax

vsocp.p rd, rs

Description

Performs element-wise one's complement (1.0f - x) with saturation to [0.0f ... 1.0f]

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = fminf(fmaxf(1.0f - rs[0], 0.0f), 1.0f)
rd[1] = fminf(fmaxf(rs[0], 0.0f), 1.0f)
rd[2] = fminf(fmaxf(1.0f - rs[1], 0.0f), 1.0f)
rd[3] = fminf(fmaxf(rs[1], 0.0f), 1.0f)
```

vavg.p

Calculate element average

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	0	1	1	1	0	rs							1	rd						

Syntax

vavg.p rd, rs

Description

Calculates the average value of the vector elements

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = (rs[0] + rs[1]) / 2
```

vavg.t

Calculate element average

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	0	1	1	1	1				rs				0				rd			

Syntax

vavg.t rd, rs

Description

Calculates the average value of the vector elements

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = (rs[0] + rs[1] + rs[2]) / 3
```

vavg.q

Calculate element average

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	0	1	1	1	1				rs				1				rd			

Syntax

vavg.q rd, rs

Description

Calculates the average value of the vector elements

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = (rs[0] + rs[1] + rs[2] + rs[3]) / 4
```

vfad.p

Calculate element sum

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	0	1	1	0	0	rs							1	rd						

Syntax

vfad.p rd, rs

Description

Adds all vector elements toghether producing a single result

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] + rs[1]
```

vfad.t

Calculate element sum

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	0	1	1	0	1	rs							0	rd						

Syntax

vfad.t rd, rs

Description

Adds all vector elements toghether producing a single result

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] + rs[1] + rs[2]
```

vfad.q

Calculate element sum

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	0	0	1	1	0	1	rs							1	rd						

Syntax

vfad.q rd, rs

Description

Adds all vector elements toghether producing a single result

Instruction performance

Throughput: 1 cycles/instruction
Latency: 7 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = rs[0] + rs[1] + rs[2] + rs[3]
```

vcmp.s

Compare vector elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	0	0	0	rt							0	rs							0	0	0	0	cond			

Syntax

vcmp.s cond, rs, rt

Description

Performs an element wise comparison specified by the immediate and writes the result to VFPU_CC. Aggregated `and` and `or` operations are also calculated for convenience.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 8 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)

Pseudocode

```
vfpu_cc[0] = comparefn(cond, rs[0], rt[0])
vfpu_cc[4] = vfpu_cc[0]
vfpu_cc[5] = vfpu_cc[0]
```

Used functions

```
unsigned comparefn(unsigned cond, float rs, float rt) {
  switch (cond) {
  case 0: return 0;
  case 1: return rs == rt;
  case 2: return rs < rt;
  case 3: return rs <= rt;
  case 4: return 1;
  case 5: return rs != rt;
  case 6: return rs >= rt;
  case 7: return rs > rt;
  case 8: return rs == 0;
  case 9: return isnan(rs);
  case 10: return isinf(rs);
  case 11: return isinf(rs) || isnan(rs);
  case 12: return rs != 0;
  case 13: return !isnan(rs);
  case 14: return !isinf(rs);
  case 15: return !isnan(rs) && !isinf(rs);
  };
}
```

vcmp.p

Compare vector elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	0	0	0	rt							0	rs							1	0	0	0	cond			

Syntax

vcmp.p cond, rs, rt

Description

Performs an element wise comparison specified by the immediate and writes the result to VFPU_CC. Aggregated `and` and `or` operations are also calculated for convenience.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 8 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)

Pseudocode

```
vfpu_cc[0] = comparefn(cond, rs[0], rt[0])
vfpu_cc[1] = comparefn(cond, rs[1], rt[1])
vfpu_cc[4] = vfpu_cc[0] | vfpu_cc[1]
vfpu_cc[5] = vfpu_cc[0] & vfpu_cc[1]
```

Used functions

```
unsigned comparefn(unsigned cond, float rs, float rt) {
  switch (cond) {
  case 0: return 0;
  case 1: return rs == rt;
  case 2: return rs < rt;
  case 3: return rs <= rt;
  case 4: return 1;
  case 5: return rs != rt;
  case 6: return rs >= rt;
  case 7: return rs > rt;
  case 8: return rs == 0;
  case 9: return isnan(rs);
  case 10: return isinf(rs);
  case 11: return isinf(rs) || isnan(rs);
  case 12: return rs != 0;
  case 13: return !isnan(rs);
  case 14: return !isinf(rs);
  case 15: return !isnan(rs) && !isinf(rs);
  };
}
```

vcmp.t

Compare vector elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	0	0	0	rt							1	rs							0	0	0	0	cond			

Syntax

vcmp.t cond, rs, rt

Description

Performs an element wise comparison specified by the immediate and writes the result to VFPU_CC. Aggregated `and` and `or` operations are also calculated for convenience.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 8 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)

Pseudocode

```
vfpu_cc[0] = comparefn(cond, rs[0], rt[0])
vfpu_cc[1] = comparefn(cond, rs[1], rt[1])
vfpu_cc[2] = comparefn(cond, rs[2], rt[2])
vfpu_cc[4] = vfpu_cc[0] | vfpu_cc[1] | vfpu_cc[2]
vfpu_cc[5] = vfpu_cc[0] & vfpu_cc[1] & vfpu_cc[2]
```

Used functions

```
unsigned comparefn(unsigned cond, float rs, float rt) {
  switch (cond) {
  case 0: return 0;
  case 1: return rs == rt;
  case 2: return rs < rt;
  case 3: return rs <= rt;
  case 4: return 1;
  case 5: return rs != rt;
  case 6: return rs >= rt;
  case 7: return rs > rt;
  case 8: return rs == 0;
  case 9: return isnan(rs);
  case 10: return isinf(rs);
  case 11: return isinf(rs) || isnan(rs);
  case 12: return rs != 0;
  case 13: return !isnan(rs);
  case 14: return !isinf(rs);
  case 15: return !isnan(rs) && !isinf(rs);
  };
}
```

vcmp.q

Compare vector elements

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	0	0	0	rt							1	rs							1	0	0	0	cond			

Syntax

vcmp.q cond, rs, rt

Description

Performs an element wise comparison specified by the immediate and writes the result to VFPU_CC. Aggregated `and` and `or` operations are also calculated for convenience.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 8 cycles

Allowed prefixes

- rt: Full support (swizzle, abs(), neg() and constants)
- rs: Full support (swizzle, abs(), neg() and constants)

Pseudocode

```
vfpu_cc[0] = comparefn(cond, rs[0], rt[0])
vfpu_cc[1] = comparefn(cond, rs[1], rt[1])
vfpu_cc[2] = comparefn(cond, rs[2], rt[2])
vfpu_cc[3] = comparefn(cond, rs[3], rt[3])
vfpu_cc[4] = vfpu_cc[0] | vfpu_cc[1] | vfpu_cc[2] | vfpu_cc[3]
vfpu_cc[5] = vfpu_cc[0] & vfpu_cc[1] & vfpu_cc[2] & vfpu_cc[3]
```

Used functions

```
unsigned comparefn(unsigned cond, float rs, float rt) {
  switch (cond) {
  case 0: return 0;
  case 1: return rs == rt;
  case 2: return rs < rt;
  case 3: return rs <= rt;
  case 4: return 1;
  case 5: return rs != rt;
```

```
    case 6: return rs >= rt;
    case 7: return rs > rt;
    case 8: return rs == 0;
    case 9: return isnan(rs);
    case 10: return isinf(rs);
    case 11: return isinf(rs) || isnan(rs);
    case 12: return rs != 0;
    case 13: return !isnan(rs);
    case 14: return !isinf(rs);
    case 15: return !isnan(rs) && !isinf(rs);
    };
  }
```

vidt.p

Identity matrix row/col initialize

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1				rd			

Syntax

vidt.p rd

Description

Initializes destination register as an identity matrix row (all zeros but one). The behaviour depends on the destination register number.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rd: Full support (masking and saturation)

vidt.q

Identity matrix row/col initialize

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	1	rd						

Syntax

vidt.q rd

Description

Initializes destination register as an identity matrix row (all zeros but one). The behaviour depends on the destination register number.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rd: Full support (masking and saturation)

vzero.s

Clear vector to zero

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0				rd			

Syntax

vzero.s rd

Description

Writes zeros (0.0f) into the destination register

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = 0
```

vzero.p

Clear vector to zero

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	rd						

Syntax

vzero.p rd

Description

Writes zeros (0.0f) into the destination register

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = 0
rd[1] = 0
```

vzero.t

Clear vector to zero

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0				rd			

Syntax

vzero.t rd

Description

Writes zeros (0.0f) into the destination register

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = 0
rd[1] = 0
rd[2] = 0
```

vzero.q

Clear vector to zero

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	1	rd						

Syntax

vzero.q rd

Description

Writes zeros (0.0f) into the destination register

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = 0
rd[1] = 0
rd[2] = 0
rd[3] = 0
```

vone.s

Clear vector to one

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	rd						

Syntax

vone.s rd

Description

Writes ones (1.0f) into the destination register

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = 1.0f
```

vone.p

Clear vector to one

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	1	rd						

Syntax

vone.p rd

Description

Writes ones (1.0f) into the destination register

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = 1.0f
rd[1] = 1.0f
```

vone.t

Clear vector to one

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	rd						

Syntax

vone.t rd

Description

Writes ones (1.0f) into the destination register

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = 1.0f
rd[1] = 1.0f
rd[2] = 1.0f
```

vone.q

Clear vector to one

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	1	rd						

Syntax

vone.q rd

Description

Writes ones (1.0f) into the destination register

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = 1.0f
rd[1] = 1.0f
rd[2] = 1.0f
rd[3] = 1.0f
```

vrnds.s

Random seed

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0				rs				0	0	0	0	0	0	0	0

Syntax

vrnds.s rs

Description

Uses the integer value as a seed for the pseudorandom number generator.

Allowed prefixes

- rs: Not supported

vrndi.s

Random integer

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	rd						

Syntax

vrndi.s rd

Description

Writes pseudorandom 32 bit numbers to the destination elements (full 32bit range)

Instruction performance

Throughput: 3 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Full support (masking and saturation)

vrndi.p

Random integer

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1	rd						

Syntax

vrndi.p rd

Description

Writes pseudorandom 32 bit numbers to the destination elements (full 32bit range)

Instruction performance

Throughput: 6 cycles/instruction
Latency: 8 cycles

Allowed prefixes

- rd: Not supported

vrndi.t

Random integer

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	rd						

Syntax

vrndi.t rd

Description

Writes pseudorandom 32 bit numbers to the destination elements (full 32bit range)

Instruction performance

Throughput: 9 cycles/instruction
Latency: 11 cycles

Allowed prefixes

- rd: Not supported

vrndi.q

Random integer

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	1	rd						

Syntax

vrndi.q rd

Description

Writes pseudorandom 32 bit numbers to the destination elements (full 32bit range)

Instruction performance

Throughput: 12 cycles/instruction
Latency: 14 cycles

Allowed prefixes

- rd: Not supported

vrndf1.s

Random float in [1..2] range

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	rd						

Syntax

vrndf1.s rd

Description

Writes pseudorandom numbers to the destination elements so that each element (x) can assert 1.0f <= x < 2.0f

Instruction performance

Throughput: 3 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Full support (masking and saturation)

vrndf1.p

Random float in [1..2] range

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	1	rd						

Syntax

vrndf1.p rd

Description

Writes pseudorandom numbers to the destination elements so that each element (x) can assert 1.0f <= x < 2.0f

Instruction performance

Throughput: 6 cycles/instruction
Latency: 8 cycles

Allowed prefixes

- rd: Not supported

vrndf1.t

Random float in [1..2] range

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	rd						

Syntax

vrndf1.t rd

Description

Writes pseudorandom numbers to the destination elements so that each element (x) can assert 1.0f <= x < 2.0f

Instruction performance

Throughput: 9 cycles/instruction
Latency: 11 cycles

Allowed prefixes

- rd: Not supported

vrndf1.q

Random float in [1..2] range

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	1	rd						

Syntax

vrndf1.q rd

Description

Writes pseudorandom numbers to the destination elements so that each element (x) can assert 1.0f <= x < 2.0f

Instruction performance

Throughput: 12 cycles/instruction
Latency: 14 cycles

Allowed prefixes

- rd: Not supported

vrndf2.s

Random float in [2..4] range

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	rd						

Syntax

vrndf2.s rd

Description

Writes pseudorandom numbers to the destination elements so that each element (x) can assert 2.0f <= x < 4.0f

Instruction performance

Throughput: 3 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Full support (masking and saturation)

vrndf2.p

Random float in [2..4] range

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	1	rd						

Syntax

vrndf2.p rd

Description

Writes pseudorandom numbers to the destination elements so that each element (x) can assert 2.0f <= x < 4.0f

Instruction performance

Throughput: 6 cycles/instruction
Latency: 8 cycles

Allowed prefixes

- rd: Not supported

vrndf2.t

Random float in [2..4] range

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	rd						

Syntax

vrndf2.t rd

Description

Writes pseudorandom numbers to the destination elements so that each element (x) can assert 2.0f <= x < 4.0f

Instruction performance

Throughput: 9 cycles/instruction
Latency: 11 cycles

Allowed prefixes

- rd: Not supported

vrndf2.q

Random float in [2..4] range

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	0	1	rd						

Syntax

vrndf2.q rd

Description

Writes pseudorandom numbers to the destination elements so that each element (x) can assert 2.0f <= x < 4.0f

Instruction performance

Throughput: 12 cycles/instruction
Latency: 14 cycles

Allowed prefixes

- rd: Not supported

vmmul.p

Matrix by matrix multiplication

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	rt							0	rs							1	rd						

Syntax

vmmul.p rd, rs, rt

Description

Performs a matrix multiplication

Instruction performance

Throughput: 4 cycles/instruction
Latency: 10 cycles

Register overlap compatibility

Output register cannot overlap with input registers

Allowed prefixes

- rt: Not supported
- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = rs[0] * rt[0] + rs[1] * rt[1]
rd[1] = rs[2] * rt[0] + rs[3] * rt[1]
rd[2] = rs[0] * rt[2] + rs[1] * rt[3]
rd[3] = rs[2] * rt[2] + rs[3] * rt[3]
```

vmmul.t

Matrix by matrix multiplication

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	rt							1	rs							0	rd						

Syntax

vmmul.t rd, rs, rt

Description

Performs a matrix multiplication

Instruction performance

Throughput: 9 cycles/instruction
Latency: 15 cycles

Register overlap compatibility

Output register cannot overlap with input registers

Allowed prefixes

- rt: Not supported
- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = rs[0] * rt[0] + rs[1] * rt[1] + rs[2] * rt[2]
rd[1] = rs[3] * rt[0] + rs[4] * rt[1] + rs[5] * rt[2]
rd[2] = rs[6] * rt[0] + rs[7] * rt[1] + rs[8] * rt[2]
rd[3] = rs[0] * rt[3] + rs[1] * rt[4] + rs[2] * rt[5]
rd[4] = rs[3] * rt[3] + rs[4] * rt[4] + rs[5] * rt[5]
rd[5] = rs[6] * rt[3] + rs[7] * rt[4] + rs[8] * rt[5]
rd[6] = rs[0] * rt[6] + rs[1] * rt[7] + rs[2] * rt[8]
rd[7] = rs[3] * rt[6] + rs[4] * rt[7] + rs[5] * rt[8]
rd[8] = rs[6] * rt[6] + rs[7] * rt[7] + rs[8] * rt[8]
```

vmmul.q

Matrix by matrix multiplication

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	rt							1	rs							1	rd						

Syntax

vmmul.q rd, rs, rt

Description

Performs a matrix multiplication

Instruction performance

Throughput: 16 cycles/instruction
Latency: 22 cycles

Register overlap compatibility

Output register cannot overlap with input registers

Allowed prefixes

- rt: Not supported
- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = rs[0] * rt[0] + rs[1] * rt[1] + rs[2] * rt[2] + rs[3] * rt[3]
rd[1] = rs[4] * rt[0] + rs[5] * rt[1] + rs[6] * rt[2] + rs[7] * rt[3]
rd[2] = rs[8] * rt[0] + rs[9] * rt[1] + rs[10] * rt[2] + rs[11] * rt[3]
rd[3] = rs[12] * rt[0] + rs[13] * rt[1] + rs[14] * rt[2] + rs[15] * rt[3]
rd[4] = rs[0] * rt[4] + rs[1] * rt[5] + rs[2] * rt[6] + rs[3] * rt[7]
rd[5] = rs[4] * rt[4] + rs[5] * rt[5] + rs[6] * rt[6] + rs[7] * rt[7]
rd[6] = rs[8] * rt[4] + rs[9] * rt[5] + rs[10] * rt[6] + rs[11] * rt[7]
rd[7] = rs[12] * rt[4] + rs[13] * rt[5] + rs[14] * rt[6] + rs[15] * rt[7]
rd[8] = rs[0] * rt[8] + rs[1] * rt[9] + rs[2] * rt[10] + rs[3] * rt[11]
rd[9] = rs[4] * rt[8] + rs[5] * rt[9] + rs[6] * rt[10] + rs[7] * rt[11]
rd[10] = rs[8] * rt[8] + rs[9] * rt[9] + rs[10] * rt[10] + rs[11] * rt[11]
rd[11] = rs[12] * rt[8] + rs[13] * rt[9] + rs[14] * rt[10] + rs[15] * rt[11]
rd[12] = rs[0] * rt[12] + rs[1] * rt[13] + rs[2] * rt[14] + rs[3] * rt[15]
rd[13] = rs[4] * rt[12] + rs[5] * rt[13] + rs[6] * rt[14] + rs[7] * rt[15]
```

```
rd[14] = rs[8] * rt[12] + rs[9] * rt[13] + rs[10] * rt[14] + rs[11] * rt[15]
rd[15] = rs[12] * rt[12] + rs[13] * rt[13] + rs[14] * rt[14] + rs[15] * rt[15]
```

vmscl.p

Matrix scale by single factor

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	rt							0	rs							1	rd						

Syntax

vmscl.p rd, rs, rt

Description

Performs a matrix scaling by a single factor

Instruction performance

Throughput: 2 cycles/instruction
Latency: 6 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rt: Not supported
- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = rs[0] * rt[0]
rd[1] = rs[1] * rt[0]
rd[2] = rs[2] * rt[0]
rd[3] = rs[3] * rt[0]
```

vmscl.t

Matrix scale by single factor

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	rt							1	rs							0	rd						

Syntax

vmscl.t rd, rs, rt

Description

Performs a matrix scaling by a single factor

Instruction performance

Throughput: 3 cycles/instruction
Latency: 7 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rt: Not supported
- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = rs[0] * rt[0]
rd[1] = rs[1] * rt[0]
rd[2] = rs[2] * rt[0]
rd[3] = rs[3] * rt[0]
rd[4] = rs[4] * rt[0]
rd[5] = rs[5] * rt[0]
rd[6] = rs[6] * rt[0]
rd[7] = rs[7] * rt[0]
rd[8] = rs[8] * rt[0]
```

vmscl.q

Matrix scale by single factor

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	rt							1	rs							1	rd						

Syntax

vmscl.q rd, rs, rt

Description

Performs a matrix scaling by a single factor

Instruction performance

Throughput: 4 cycles/instruction
Latency: 8 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rt: Not supported
- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = rs[0] * rt[0]
rd[1] = rs[1] * rt[0]
rd[2] = rs[2] * rt[0]
rd[3] = rs[3] * rt[0]
rd[4] = rs[4] * rt[0]
rd[5] = rs[5] * rt[0]
rd[6] = rs[6] * rt[0]
rd[7] = rs[7] * rt[0]
rd[8] = rs[8] * rt[0]
rd[9] = rs[9] * rt[0]
rd[10] = rs[10] * rt[0]
rd[11] = rs[11] * rt[0]
rd[12] = rs[12] * rt[0]
rd[13] = rs[13] * rt[0]
```

```
rd[14] = rs[14] * rt[0]
rd[15] = rs[15] * rt[0]
```

vmmov.p

Copy matrix

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	rs							1	rd						

Syntax

vmmov.p rd, rs

Description

Element-wise data copy

Instruction performance

Throughput: 2 cycles/instruction
Latency: 4 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = rs[0]
rd[1] = rs[1]
rd[2] = rs[2]
rd[3] = rs[3]
```

vmmov.t

Copy matrix

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	1	rs							0	rd						

Syntax

vmmov.t rd, rs

Description

Element-wise data copy

Instruction performance

Throughput: 3 cycles/instruction
Latency: 5 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = rs[0]
rd[1] = rs[1]
rd[2] = rs[2]
rd[3] = rs[3]
rd[4] = rs[4]
rd[5] = rs[5]
rd[6] = rs[6]
rd[7] = rs[7]
rd[8] = rs[8]
```

vmmov.q

Copy matrix

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	0	0	0	0	0	1	rs							1	rd						

Syntax

vmmov.q rd, rs

Description

Element-wise data copy

Instruction performance

Throughput: 4 cycles/instruction
Latency: 6 cycles

Register overlap compatibility

Output register can only overlap with input registers if they are identical

Allowed prefixes

- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = rs[0]
rd[1] = rs[1]
rd[2] = rs[2]
rd[3] = rs[3]
rd[4] = rs[4]
rd[5] = rs[5]
rd[6] = rs[6]
rd[7] = rs[7]
rd[8] = rs[8]
rd[9] = rs[9]
rd[10] = rs[10]
rd[11] = rs[11]
rd[12] = rs[12]
rd[13] = rs[13]
```

```
rd[14] = rs[14]
rd[15] = rs[15]
```

vmidt.p

Set matrix to identity

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	rd						

Syntax

vmidt.p rd

Description

Writes the identity matrix into the destination register

Instruction performance

Throughput: 2 cycles/instruction
Latency: 4 cycles

Allowed prefixes

- rd: Not supported

Pseudocode

```
rd[0] = 1f
rd[1] = 0f
rd[2] = 0f
rd[3] = 1f
```

vmidt.t

Set matrix to identity

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0				rd			

Syntax

vmidt.t rd

Description

Writes the identity matrix into the destination register

Instruction performance

Throughput: 3 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Not supported

Pseudocode

```
rd[0] = 1f
rd[1] = 0f
rd[2] = 0f
rd[3] = 0f
rd[4] = 1f
rd[5] = 0f
rd[6] = 0f
rd[7] = 0f
rd[8] = 1f
```

vmidt.q

Set matrix to identity

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	1	rd						

Syntax

vmidt.q rd

Description

Writes the identity matrix into the destination register

Instruction performance

Throughput: 4 cycles/instruction
Latency: 6 cycles

Allowed prefixes

- rd: Not supported

Pseudocode

```
rd[0] = 1f
rd[1] = 0f
rd[2] = 0f
rd[3] = 0f
rd[4] = 0f
rd[5] = 1f
rd[6] = 0f
rd[7] = 0f
rd[8] = 0f
rd[9] = 0f
rd[10] = 1f
rd[11] = 0f
rd[12] = 0f
rd[13] = 0f
rd[14] = 0f
rd[15] = 1f
```

vmzero.p

Clear matrix to zero

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	rd						

Syntax

vmzero.p rd

Description

Writes a zero matrix into the destination register

Instruction performance

Throughput: 2 cycles/instruction
Latency: 4 cycles

Allowed prefixes

- rd: Not supported

Pseudocode

```
rd[0] = 0
rd[1] = 0
rd[2] = 0
rd[3] = 0
```

vmzero.t

Clear matrix to zero

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	rd						

Syntax

vmzero.t rd

Description

Writes a zero matrix into the destination register

Instruction performance

Throughput: 3 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Not supported

Pseudocode

```
rd[0] = 0
rd[1] = 0
rd[2] = 0
rd[3] = 0
rd[4] = 0
rd[5] = 0
rd[6] = 0
rd[7] = 0
rd[8] = 0
```

vmzero.q

Clear matrix to zero

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	1	rd						

Syntax

vmzero.q rd

Description

Writes a zero matrix into the destination register

Instruction performance

Throughput: 4 cycles/instruction
Latency: 6 cycles

Allowed prefixes

- rd: Not supported

Pseudocode

```
rd[0] = 0
rd[1] = 0
rd[2] = 0
rd[3] = 0
rd[4] = 0
rd[5] = 0
rd[6] = 0
rd[7] = 0
rd[8] = 0
rd[9] = 0
rd[10] = 0
rd[11] = 0
rd[12] = 0
rd[13] = 0
rd[14] = 0
rd[15] = 0
```

vmone.p

Clear matrix to one

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	1	rd						

Syntax

vmone.p rd

Description

Overwrites all elements in a matrix with ones (1.0f)

Instruction performance

Throughput: 2 cycles/instruction
Latency: 4 cycles

Allowed prefixes

- rd: Not supported

Pseudocode

```
rd[0] = 1
rd[1] = 1
rd[2] = 1
rd[3] = 1
```

vmone.t

Clear matrix to one

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	rd						

Syntax

vmone.t rd

Description

Overwrites all elements in a matrix with ones (1.0f)

Instruction performance

Throughput: 3 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Not supported

Pseudocode

```
rd[0] = 1
rd[1] = 1
rd[2] = 1
rd[3] = 1
rd[4] = 1
rd[5] = 1
rd[6] = 1
rd[7] = 1
rd[8] = 1
```

vmone.q

Clear matrix to one

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	1	rd						

Syntax

vmone.q rd

Description

Overwrites all elements in a matrix with ones (1.0f)

Instruction performance

Throughput: 4 cycles/instruction
Latency: 6 cycles

Allowed prefixes

- rd: Not supported

Pseudocode

```
rd[0] = 1
rd[1] = 1
rd[2] = 1
rd[3] = 1
rd[4] = 1
rd[5] = 1
rd[6] = 1
rd[7] = 1
rd[8] = 1
rd[9] = 1
rd[10] = 1
rd[11] = 1
rd[12] = 1
rd[13] = 1
rd[14] = 1
rd[15] = 1
```

vtfm2.p

Vector by matrix transform

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	1				rt				0				rs				1				rd			

Syntax

vtfm2.p rd, rs, rt

Description

Performs a vector-matrix transform (matrix-vector product), with a vector result

Instruction performance

Throughput: 2 cycles/instruction
Latency: 8 cycles

Register overlap compatibility

Output register cannot overlap with input registers

Allowed prefixes

- rt: Not supported
- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = rs[0] * rt[0] + rs[1] * rt[1]
rd[1] = rs[2] * rt[0] + rs[3] * rt[1]
```

vtfm3.t

Vector by matrix transform

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	0	rt							1	rs							0	rd						

Syntax

vtfm3.t rd, rs, rt

Description

Performs a vector-matrix transform (matrix-vector product), with a vector result

Instruction performance

Throughput: 3 cycles/instruction
Latency: 9 cycles

Register overlap compatibility

Output register cannot overlap with input registers

Allowed prefixes

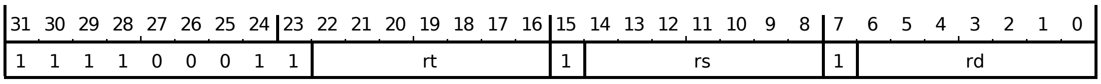
- rt: Not supported
- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = rs[0] * rt[0] + rs[1] * rt[1] + rs[2] * rt[2]
rd[1] = rs[3] * rt[0] + rs[4] * rt[1] + rs[5] * rt[2]
rd[2] = rs[6] * rt[0] + rs[7] * rt[1] + rs[8] * rt[2]
```

vtfm4.q

Vector by matrix transform



Syntax

vtfm4.q rd, rs, rt

Description

Performs a vector-matrix transform (matrix-vector product), with a vector result

Instruction performance

Throughput: 4 cycles/instruction
Latency: 10 cycles

Register overlap compatibility

Output register cannot overlap with input registers

Allowed prefixes

- rt: Not supported
- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = rs[0] * rt[0] + rs[1] * rt[1] + rs[2] * rt[2] + rs[3] * rt[3]
rd[1] = rs[4] * rt[0] + rs[5] * rt[1] + rs[6] * rt[2] + rs[7] * rt[3]
rd[2] = rs[8] * rt[0] + rs[9] * rt[1] + rs[10] * rt[2] + rs[11] * rt[3]
rd[3] = rs[12] * rt[0] + rs[13] * rt[1] + rs[14] * rt[2] + rs[15] * rt[3]
```

vhtfm2.p Vector by matrix homogeneous transform

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	1	rt							0	rs							0	rd						

Syntax

vhtfm2.p rd, rs, rt

Description

Performs a vector-matrix homogeneous transform (matrix-vector product), with a vector result

Bugs

Whenever the used output register rd is 64 or above the output is incorrect. The result is rotated left by one position around the 4-element register (row or column). Check vfpu-bugs.c for more information and examples.

Instruction performance

Throughput: 2 cycles/instruction
Latency: 8 cycles

Register overlap compatibility

Output register cannot overlap with input registers

Allowed prefixes

- rt: Not supported
- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = rs[0] * rt[0] + rs[1]
rd[1] = rs[2] * rt[0] + rs[3]
```

vhtfm3.t Vector by matrix homogeneous transform

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	0	rt							0	rs							1	rd						

Syntax

vhtfm3.t rd, rs, rt

Description

Performs a vector-matrix homogeneous transform (matrix-vector product), with a vector result

Bugs

Whenever the used output register rd is 64 or above the output is incorrect. The result is rotated left by two position around the 4-element register (row or column). Check vfpu-bugs.c for more information and examples.

Instruction performance

Throughput: 3 cycles/instruction
Latency: 9 cycles

Register overlap compatibility

Output register cannot overlap with input registers

Allowed prefixes

- rt: Not supported
- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = rs[0] * rt[0] + rs[1] * rt[1] + rs[2]
rd[1] = rs[3] * rt[0] + rs[4] * rt[1] + rs[5]
rd[2] = rs[6] * rt[0] + rs[7] * rt[1] + rs[8]
```

vhtfm4.q Vector by matrix homogeneous transform

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	1	rt							1	rs							0	rd						

Syntax

vhtfm4.q rd, rs, rt

Description

Performs a vector-matrix homogeneous transform (matrix-vector product), with a vector result

Instruction performance

Throughput: 4 cycles/instruction
Latency: 10 cycles

Register overlap compatibility

Output register cannot overlap with input registers

Allowed prefixes

- rt: Not supported
- rs: Not supported
- rd: Not supported

Pseudocode

```
rd[0] = rs[0] * rt[0] + rs[1] * rt[1] + rs[2] * rt[2] + rs[3]
rd[1] = rs[4] * rt[0] + rs[5] * rt[1] + rs[6] * rt[2] + rs[7]
rd[2] = rs[8] * rt[0] + rs[9] * rt[1] + rs[10] * rt[2] + rs[11]
rd[3] = rs[12] * rt[0] + rs[13] * rt[1] + rs[14] * rt[2] + rs[15]
```

vcmovf.s

Conditional move (false)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	1	0	1	0	1	cc			0	rs							0	rd						

Syntax

vcmovf.s rd, rs, imm3

Description

Performs a register move operation (like vmov) conditional to a VFPU_CC bit being zero. If imm3 has the special value of 6, each vector lane will check its corresponding bit instead. This can be used to conditionally move each of the elements based on, for instance, a vcmp operation. A value of 7 in imm3 is not specified.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Not supported

Pseudocode

```
rd[0] = (cc == 6) ? (vfpu_cc[0] ? rd[0] : rs[0]) : (vfpu_cc[cc] ? rd[0] : rs[0])
```

vcmovf.t

Conditional move (false)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	1	0	1	0	1	cc			1	rs							0	rd						

Syntax

vcmovf.t rd, rs, imm3

Description

Performs a register move operation (like vmov) conditional to a VFPU_CC bit being zero. If imm3 has the special value of 6, each vector lane will check its corresponding bit instead. This can be used to conditionally move each of the elements based on, for instance, a vcmp operation. A value of 7 in imm3 is not specified.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Not supported

Pseudocode

```
rd[0] = (cc == 6) ? (vfpu_cc[0] ? rd[0] : rs[0]) : (vfpu_cc[cc] ? rd[0] : rs[0])
rd[1] = (cc == 6) ? (vfpu_cc[1] ? rd[1] : rs[1]) : (vfpu_cc[cc] ? rd[1] : rs[1])
rd[2] = (cc == 6) ? (vfpu_cc[2] ? rd[2] : rs[2]) : (vfpu_cc[cc] ? rd[2] : rs[2])
```

vcmovf.p

Conditional move (false)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	1	0	1	0	1	cc			0	rs							1	rd						

Syntax

vcmovf.p rd, rs, imm3

Description

Performs a register move operation (like vmov) conditional to a VFPU_CC bit being zero. If imm3 has the special value of 6, each vector lane will check its corresponding bit instead. This can be used to conditionally move each of the elements based on, for instance, a vcmp operation. A value of 7 in imm3 is not specified.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Not supported

Pseudocode

```
rd[0] = (cc == 6) ? (vfpu_cc[0] ? rd[0] : rs[0]) : (vfpu_cc[cc] ? rd[0] : rs[0])
rd[1] = (cc == 6) ? (vfpu_cc[1] ? rd[1] : rs[1]) : (vfpu_cc[cc] ? rd[1] : rs[1])
```

vcmovf.q

Conditional move (false)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	1	0	1	0	1	cc			1	rs							1	rd						

Syntax

vcmovf.q rd, rs, imm3

Description

Performs a register move operation (like vmov) conditional to a VFPU_CC bit being zero. If imm3 has the special value of 6, each vector lane will check its corresponding bit instead. This can be used to conditionally move each of the elements based on, for instance, a vcmp operation. A value of 7 in imm3 is not specified.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Not supported

Pseudocode

```
rd[0] = (cc == 6) ? (vfpu_cc[0] ? rd[0] : rs[0]) : (vfpu_cc[cc] ? rd[0] : rs[0])
rd[1] = (cc == 6) ? (vfpu_cc[1] ? rd[1] : rs[1]) : (vfpu_cc[cc] ? rd[1] : rs[1])
rd[2] = (cc == 6) ? (vfpu_cc[2] ? rd[2] : rs[2]) : (vfpu_cc[cc] ? rd[2] : rs[2])
rd[3] = (cc == 6) ? (vfpu_cc[3] ? rd[3] : rs[3]) : (vfpu_cc[cc] ? rd[3] : rs[3])
```

vcmovt.s

Conditional move (true)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	1	0	1	0	0	cc			0	rs							0	rd						

Syntax

vcmovt.s rd, rs, imm3

Description

Performs a register move operation (like vmov) conditional to a VFPU_CC bit being one. If imm3 has the special value of 6, each vector lane will check its corresponding bit instead. This can be used to conditionally move each of the elements based on, for instance, a vcmp operation. A value of 7 in imm3 is not specified.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Not supported

Pseudocode

```
rd[0] = (cc == 6) ? (vfpu_cc[0] ? rs[0] : rd[0]) : (vfpu_cc[cc] ? rs[0] : rd[0])
```

vcmovt.t

Conditional move (true)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	1	0	1	0	0	cc			1	rs							0	rd						

Syntax

vcmovt.t rd, rs, imm3

Description

Performs a register move operation (like vmov) conditional to a VFPU_CC bit being one. If imm3 has the special value of 6, each vector lane will check its corresponding bit instead. This can be used to conditionally move each of the elements based on, for instance, a vcmp operation. A value of 7 in imm3 is not specified.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Not supported

Pseudocode

```
rd[0] = (cc == 6) ? (vfpu_cc[0] ? rs[0] : rd[0]) : (vfpu_cc[cc] ? rs[0] : rd[0])
rd[1] = (cc == 6) ? (vfpu_cc[1] ? rs[1] : rd[1]) : (vfpu_cc[cc] ? rs[1] : rd[1])
rd[2] = (cc == 6) ? (vfpu_cc[2] ? rs[2] : rd[2]) : (vfpu_cc[cc] ? rs[2] : rd[2])
```

vcmovt.p

Conditional move (true)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	1	0	1	0	0	cc			0	rs							1	rd						

Syntax

vcmovt.p rd, rs, imm3

Description

Performs a register move operation (like vmov) conditional to a VFPU_CC bit being one. If imm3 has the special value of 6, each vector lane will check its corresponding bit instead. This can be used to conditionally move each of the elements based on, for instance, a vcmp operation. A value of 7 in imm3 is not specified.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Not supported

Pseudocode

```
rd[0] = (cc == 6) ? (vfpu_cc[0] ? rs[0] : rd[0]) : (vfpu_cc[cc] ? rs[0] : rd[0])
rd[1] = (cc == 6) ? (vfpu_cc[1] ? rs[1] : rd[1]) : (vfpu_cc[cc] ? rs[1] : rd[1])
```

vcmovt.q

Conditional move (true)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	1	0	1	0	1	0	0	cc			1	rs							1	rd						

Syntax

vcmovt.q rd, rs, imm3

Description

Performs a register move operation (like vmov) conditional to a VFPU_CC bit being one. If imm3 has the special value of 6, each vector lane will check its corresponding bit instead. This can be used to conditionally move each of the elements based on, for instance, a vcmp operation. A value of 7 in imm3 is not specified.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Not supported

Pseudocode

```
rd[0] = (cc == 6) ? (vfpu_cc[0] ? rs[0] : rd[0]) : (vfpu_cc[cc] ? rs[0] : rd[0])
rd[1] = (cc == 6) ? (vfpu_cc[1] ? rs[1] : rd[1]) : (vfpu_cc[cc] ? rs[1] : rd[1])
rd[2] = (cc == 6) ? (vfpu_cc[2] ? rs[2] : rd[2]) : (vfpu_cc[cc] ? rs[2] : rd[2])
rd[3] = (cc == 6) ? (vfpu_cc[3] ? rs[3] : rd[3]) : (vfpu_cc[cc] ? rs[3] : rd[3])
```

vi2uc.q

Pack integer to unsigned char

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	1	1	1	0	0	1	rs							1	rd						

Syntax

vi2uc.q rd, rs

Description

Converts the four integer inputs to char and packs them as a single element word. The conversion process takes the 8 most significant bits of each integer and clamps any negative input values to zero.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = (rs[0] & 0x80000000 ? 0 : ((rs[0] >> 23))) | (rs[1] & 0x80000000 ? 0 : ((rs[1]
>> 23) << 8)) | (rs[2] & 0x80000000 ? 0 : ((rs[2] >> 23) << 16)) | (rs[3] &
0x80000000 ? 0 : ((rs[3] >> 23) << 24))
```

vi2c.q

Pack integer to char

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	1	1	1	0	1	1				rs				1				rd			

Syntax

vi2c.q rd, rs

Description

Converts the four integer inputs to char and packs them as a single element word. The conversion process takes the 8 most significant bits of each integer.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = ((rs[0] >> 24)) | ((rs[1] >> 24) << 8) | ((rs[2] >> 24) << 16) | ((rs[3] >>
24) << 24)
```

vi2us.p

Pack integer to unsigned short

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	1	1	1	1	0	0	rs							1	rd						

Syntax

vi2us.p rd, rs

Description

Converts the integer inputs to short and packs them in pairs in the output register. The conversion process takes the 16 most significant bits of each integer and clamps any negative input values to zero.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = (rs[0] & 0x80000000 ? 0 : ((rs[0] >> 15))) | (rs[1] & 0x80000000 ? 0 : ((rs[1]
>> 15) << 16))
```

vi2us.q

Pack integer to unsigned short

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	1	1	1	1	0	1	rs							1	rd						

Syntax

vi2us.q rd, rs

Description

Converts the integer inputs to short and packs them in pairs in the output register. The conversion process takes the 16 most significant bits of each integer and clamps any negative input values to zero.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = (rs[0] & 0x80000000 ? 0 : ((rs[0] >> 15))) | (rs[1] & 0x80000000 ? 0 : ((rs[1] >> 15) << 16))
rd[1] = (rs[2] & 0x80000000 ? 0 : ((rs[2] >> 15))) | (rs[3] & 0x80000000 ? 0 : ((rs[3] >> 15) << 16))
```

vi2s.p

Pack integer to short

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	0	rs							1	rd						

Syntax

vi2s.p rd, rs

Description

Converts the integer inputs to short and packs them in pairs in the output register. The conversion process takes the 16 most significant bits of each integer.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = ((rs[0] >> 16)) | ((rs[1] >> 16) << 16)
```

vi2s.q

Pack integer to short

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1	rs							1	rd						

Syntax

vi2s.q rd, rs

Description

Converts the integer inputs to short and packs them in pairs in the output register. The conversion process takes the 16 most significant bits of each integer.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = ((rs[0] >> 16)) | ((rs[1] >> 16) << 16)
rd[1] = ((rs[2] >> 16)) | ((rs[3] >> 16) << 16)
```

vf2h.p

Pack float to float16

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	0	rs							1	rd						

Syntax

vf2h.p rd, rs

Description

Converts the float inputs to float16 (half-float) and packs them in pairs in the output register. The conversion process may naturally result in precision loss.

Notes

The conversion discards the most significant mantissa bits. This can affect NaN encoding.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = (ifloat32(rs[0])) | (ifloat32(rs[1]) << 16)
```

Used functions

```
uint16_t ifloat32(uint32_t fp32) {
  uint16_t exponent = (fp32 >> 23) & 0xFF;
  uint32_t mantissa = (fp32 & 0x7FFFFF);
  uint16_t sign = (fp32 >> 16) & 0x8000;

  if (!exponent)
    return sign;   // Denormals rounded to zero
  if (exponent == 255) {
    // Inf/Nan case
    // Note: there's a bug around NaN conversion,
    // sometimes a NaN will be converted to Inf depending on the mantissa
    // (ie. 0x7ffff000 is a NaN but will be converted to +Inf)
    exponent = 31;
    mantissa &= 0x3FF;
  }
  else if (exponent <= 112) {
    // Too small to be represented (or zero or subnormal)
    mantissa = 0;
    exponent = 0;
  }
  else if (exponent >= 143) {
    // Too big to be represented (map to inf)
    mantissa = 0;
    exponent = 31;
  }
  else {
    // Convert with mantissa precision loss
    exponent -= 127 - 15;
    mantissa >>= 13;
  }

  return sign | (exponent << 10) | mantissa;
}
```

vf2h.q

Pack float to float16

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	1	rs							1	rd						

Syntax

vf2h.q rd, rs

Description

Converts the float inputs to float16 (half-float) and packs them in pairs in the output register. The conversion process may naturally result in precision loss.

Notes

The conversion discards the most significant mantissa bits. This can affect NaN encoding.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rs: Full support (swizzle, abs(), neg() and constants)
- rd: Partial support (masking only)

Pseudocode

```
rd[0] = (ifloat32(rs[0])) | (ifloat32(rs[1]) << 16)
rd[1] = (ifloat32(rs[2])) | (ifloat32(rs[3]) << 16)
```

Used functions

```
uint16_t ifloat32(uint32_t fp32) {
  uint16_t exponent = (fp32 >> 23) & 0xFF;
  uint32_t mantissa = (fp32 & 0x7FFFFF);
  uint16_t sign = (fp32 >> 16) & 0x8000;

  if (!exponent)
    return sign;   // Denormals rounded to zero
  if (exponent == 255) {
    // Inf/Nan case
    // Note: there's a bug around NaN conversion,
    // sometimes a NaN will be converted to Inf depending on the mantissa
    // (ie. 0x7ffff000 is a NaN but will be converted to +Inf)
    exponent = 31;
    mantissa &= 0x3FF;
  }
  else if (exponent <= 112) {
    // Too small to be represented (or zero or subnormal)
    mantissa = 0;
    exponent = 0;
  }
  else if (exponent >= 143) {
    // Too big to be represented (map to inf)
    mantissa = 0;
    exponent = 31;
  }
  else {
    // Convert with mantissa precision loss
    exponent -= 127 - 15;
    mantissa >>= 13;
  }

  return sign | (exponent << 10) | mantissa;
}
```

vs2i.s

Unpack short to integer

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	1	1	0	1	1	0	rs							0	rd						

Syntax

vs2i.s rd, rs

Description

Converts the input packed shorts into full 32 bit integers in the output register. The input is placed on the most significant bits of the output integer, while the least significant bits are filled with zeros.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rd: Partial support (masking only)
- rs: Not supported

Pseudocode

```
rd[0] = (rs[0]) << 16
rd[1] = (rs[0] >> 16) << 16
```

vs2i.p

Unpack short to integer

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	1	1	0	1	1	0	rs							1	rd						

Syntax

vs2i.p rd, rs

Description

Converts the input packed shorts into full 32 bit integers in the output register. The input is placed on the most significant bits of the output integer, while the least significant bits are filled with zeros.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rd: Partial support (masking only)
- rs: Not supported

Pseudocode

```
rd[0] = (rs[0]) << 16
rd[1] = (rs[0] >> 16) << 16
rd[2] = (rs[1]) << 16
rd[3] = (rs[1] >> 16) << 16
```

vus2i.s

Unpack short to unsigned integer

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	1	1	0	1	0	0	rs							0	rd						

Syntax

vus2i.s rd, rs

Description

Converts the input packed shorts into full 32 bit integers in the output register. The input is placed on the most significant bits of the output integer, while the least significant bits are filled with zeros.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rd: Partial support (masking only)
- rs: Not supported

Pseudocode

```
rd[0] = ((rs[0]) << 15) & 0x7FFFFFFF
rd[1] = ((rs[0] >> 16) << 15) & 0x7FFFFFFF
```

vus2i.p — Unpack short to unsigned integer

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	1	1	0	1	0	0	rs							1	rd						

Syntax

vus2i.p rd, rs

Description

Converts the input packed shorts into full 32 bit integers in the output register. The input is placed on the most significant bits of the output integer, while the least significant bits are filled with zeros.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rd: Partial support (masking only)
- rs: Not supported

Pseudocode

```
rd[0] = ((rs[0]) << 15) & 0x7FFFFFFF
rd[1] = ((rs[0] >> 16) << 15) & 0x7FFFFFFF
rd[2] = ((rs[1]) << 15) & 0x7FFFFFFF
rd[3] = ((rs[1] >> 16) << 15) & 0x7FFFFFFF
```

vc2i.s

Unpack char to integer

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	1	1	0	0	1	0	rs							0	rd						

Syntax

vc2i.s rd, rs

Description

Converts the input packed chars into full 32 bit integers in the output register. The input is placed on the most significant bits of the output integer, while the least significant bits are filled with zeros.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rd: Partial support (masking only)
- rs: Not supported

Pseudocode

```
rd[0] = (rs[0]) << 24
rd[1] = (rs[0] >> 8) << 24
rd[2] = (rs[0] >> 16) << 24
rd[3] = (rs[0] >> 24) << 24
```

vuc2ifs.s

Unpack char to unsigned integer

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	1	1	0	0	0	0	rs							0	rd						

Syntax

vuc2ifs.s rd, rs

Description

Converts the input packed chars into full 32 bit integers in the output register. The input is placed on the most significant bits of the output integer, while the least significant bits are filled with zeros XXXXXs.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rd: Partial support (masking only)
- rs: Not supported

Pseudocode

```
rd[0] = (((rs[0]) & 0xFF) << 23) | (((rs[0]) & 0xFF) << 15) | (((rs[0]) & 0xFF) << 7)
| (((rs[0]) & 0xFF) >> 1)
rd[1] = (((rs[0] >> 8) & 0xFF) << 23) | (((rs[0] >> 8) & 0xFF) << 15) | (((rs[0] >> 8)
& 0xFF) << 7) | (((rs[0] >> 8) & 0xFF) >> 1)
rd[2] = (((rs[0] >> 16) & 0xFF) << 23) | (((rs[0] >> 16) & 0xFF) << 15) | (((rs[0] >>
16) & 0xFF) << 7) | (((rs[0] >> 16) & 0xFF) >> 1)
rd[3] = (((rs[0] >> 24) & 0xFF) << 23) | (((rs[0] >> 24) & 0xFF) << 15) | (((rs[0] >>
24) & 0xFF) << 7) | (((rs[0] >> 24) & 0xFF) >> 1)
```

vh2f.s

Unpack float16 to float

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	0	rs							0	rd						

Syntax

vh2f.s rd, rs

Description

Converts the input packed float16 into full 32 bit floating point numbers.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Full support (masking and saturation)
- rs: Not supported

Pseudocode

```
rd[0] = ifloat16(rs[0])
rd[1] = ifloat16(rs[0] >> 16)
```

Used functions

```
uint32_t ifloat16(uint16_t fp16) {
  // Format is S.EEEEE.MMMMMMMMMM
  uint32_t exponent = (fp16 >> 10) & 0x1F;
  uint32_t mantissa = (fp16 & 0x3FF);
  uint32_t sign = (fp16 & 0x8000) << 16;

  if (!exponent)
    return sign;   // Denormals rounded to zero

  if (exponent == 31) {   // NaN/Inf
    exponent = 255;
  }
  else {
    mantissa <<= 13;
    exponent += 127 - 15;
  }

  // Direct conversion, no mantissa/exponent conversion
  return sign | (exponent << 23) | mantissa;
}
```

vh2f.p

Unpack float16 to float

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	0	rs							1	rd						

Syntax

vh2f.p rd, rs

Description

Converts the input packed float16 into full 32 bit floating point numbers.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Full support (masking and saturation)
- rs: Not supported

Pseudocode

```
rd[0] = ifloat16(rs[0])
rd[1] = ifloat16(rs[0] >> 16)
rd[2] = ifloat16(rs[1])
rd[3] = ifloat16(rs[1] >> 16)
```

Used functions

```
uint32_t ifloat16(uint16_t fp16) {
  // Format is S.EEEEE.MMMMMMMMMM
  uint32_t exponent = (fp16 >> 10) & 0x1F;
  uint32_t mantissa = (fp16 & 0x3FF);
  uint32_t sign = (fp16 & 0x8000) << 16;

  if (!exponent)
    return sign;   // Denormals rounded to zero

  if (exponent == 31) {   // NaN/Inf
    exponent = 255;
  }
  else {
    mantissa <<= 13;
    exponent += 127 - 15;
  }

  // Direct conversion, no mantissa/exponent conversion
  return sign | (exponent << 23) | mantissa;
}
```

vt4444.q

ABGR4444 color conversion

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	1	1	0	0	1	1	rs							1	rd						

Syntax

vt4444.q rd, rs

Description

Converts four ABGR8888 color points to ABGR4444. The output 16 bit values are packed into a vector register pair.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Not supported

Pseudocode

```
rd[0] = ((rs[0] >> 4) & 0x0000000F) | ((rs[0] >> 8) & 0x000000F0) | ((rs[0] >> 12) &
0x00000F00) | ((rs[0] >> 16) & 0x0000F000) | ((rs[1] << 12) & 0x000F0000) | ((rs[1] <<
8) & 0x00F00000) | ((rs[1] << 4) & 0x0F000000) | ((rs[1]) & 0xF0000000)
rd[1] = ((rs[2] >> 4) & 0x0000000F) | ((rs[2] >> 8) & 0x000000F0) | ((rs[2] >> 12) &
0x00000F00) | ((rs[2] >> 16) & 0x0000F000) | ((rs[3] << 12) & 0x000F0000) | ((rs[3] <<
8) & 0x00F00000) | ((rs[3] << 4) & 0x0F000000) | ((rs[3]) & 0xF0000000)
```

vt5551.q

ABGR1555 color conversion

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	1	1	0	1	0	1	rs							1	rd						

Syntax

vt5551.q rd, rs

Description

Converts four ABGR8888 color points to ABGR1555. The output 16 bit values are packed into a vector register pair.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Not supported

Pseudocode

```
rd[0] = ((rs[0] >> 3) & 0x0000001F) | ((rs[0] >> 6) & 0x000003E0) | ((rs[0] >> 9) &
0x00007C00) | ((rs[0] >> 16) & 0x00008000) | ((rs[1] << 13) & 0x001F0000) | ((rs[1] <<
10) & 0x03E00000) | ((rs[1] << 7) & 0x7C000000) | ((rs[1]) & 0x80000000)
rd[1] = ((rs[2] >> 3) & 0x0000001F) | ((rs[2] >> 6) & 0x000003E0) | ((rs[2] >> 9) &
0x00007C00) | ((rs[2] >> 16) & 0x00008000) | ((rs[3] << 13) & 0x001F0000) | ((rs[3] <<
10) & 0x03E00000) | ((rs[3] << 7) & 0x7C000000) | ((rs[3]) & 0x80000000)
```

vt5650.q

BGR565 color conversion

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	0	1	1	0	1	1	1	rs							1	rd						

Syntax

vt5650.q rd, rs

Description

Converts four ABGR8888 color points to BGR565. The output 16 bit values are packed into a vector register pair.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rs: Partial support (swizzle only)
- rd: Not supported

Pseudocode

```
rd[0] = ((rs[0] >> 3) & 0x0000001F) | ((rs[0] >> 5) & 0x000007E0) | ((rs[0] >> 8) & 0x0000F800) | ((rs[1] << 13) & 0x001F0000) | ((rs[1] << 11) & 0x07E00000) | ((rs[1] << 8) & 0xF8000000)
rd[1] = ((rs[2] >> 3) & 0x0000001F) | ((rs[2] >> 5) & 0x000007E0) | ((rs[2] >> 8) & 0x0000F800) | ((rs[3] << 13) & 0x001F0000) | ((rs[3] << 11) & 0x07E00000) | ((rs[3] << 8) & 0xF8000000)
```

viim.s — Load constant integer value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	1	0	rd							imval															

Syntax

viim.s rd, imm16

Description

Loads a signed 16 bit immediate value (converted to floating point) in a register

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = (float)(int16_t)(imval)
```

vfim.s

Load constant float value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	1	1	rd							imval															

Syntax

vfim.s rd, imm16

Description

Loads a float16 immediate value in a register

Instruction performance

Throughput: 1 cycles/instruction
Latency: 5 cycles

Allowed prefixes

- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = ifloat16(imval)
```

Used functions

```
uint32_t ifloat16(uint16_t fp16) {
  // Format is S.EEEEE.MMMMMMMMMM
  uint32_t exponent = (fp16 >> 10) & 0x1F;
  uint32_t mantissa = (fp16 & 0x3FF);
  uint32_t sign = (fp16 & 0x8000) << 16;

  if (!exponent)
    return sign;   // Denormals rounded to zero

  if (exponent == 31) {   // NaN/Inf
    exponent = 255;
  }
  else {
    mantissa <<= 13;
    exponent += 127 - 15;
  }

  // Direct conversion, no mantissa/exponent conversion
  return sign | (exponent << 23) | mantissa;
}
```

vcst.s

Load special constant

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	1	imval					0	0	0	0	0	0	0	0	0	rd						

Syntax

vcst.s rd, imm5

Description

Loads a predefined indexed floating point constant specified by the immediate field

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = fpcst(imval)
```

Used functions

```
uint32_t fpcst(uint8_t cnum) {
  const uint32_t cntlist[] = {
    0x7f7fffff,   // VFPU_HUGE  [3.40282346e+38] (max exp & mantissa)
    0x3fb504f3,   // SQRT(2)    [1.41421353e+00]
    0x3f3504f3,   // SQRT(1/2)  [7.07106769e-01]
    0x3f906ebb,   // 2/SQRT(PI) [1.12837922e+00]
    0x3f22f983,   // 2/PI       [6.36619746e-01]
    0x3ea2f983,   // 1/PI       [3.18309873e-01]
    0x3f490fdb,   // PI/4       [7.85398185e-01]
    0x3fc90fdb,   // PI/2       [1.57079637e+00]
    0x40490fdb,   // PI         [3.14159274e+00]
    0x402df854,   // e          [2.71828174e+00]
    0x3fb8aa3b,   // LOG2(e)    [1.44269502e+00]
    0x3ede5bd9,   // LOG10(e)   [4.34294492e-01]
    0x3f317218,   // LOGe(2)    [6.93147182e-01]
    0x40135d8e,   // LOGe(10)   [2.30258512e+00]
    0x40c90fdb,   // 2PI        [6.28318548e+00]
    0x3f060a92,   // PI/6       [5.23598790e-01]
    0x3e9a209b,   // LOG10(2)   [3.01030009e-01]
    0x40549a78,   // LOG2(10)   [3.32192802e+00]
    0x3f5db3d7,   // SQRT(3)/2  [8.66025388e-01]
  };
  return cntlist[cnum - 1];
}
```

vcst.p

Load special constant

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	1	imval					0	0	0	0	0	0	0	0	1	rd						

Syntax

vcst.p rd, imm5

Description

Loads a predefined indexed floating point constant specified by the immediate field

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = fpcst(imval)
rd[1] = fpcst(imval)
```

Used functions

```
uint32_t fpcst(uint8_t cnum) {
  const uint32_t cntlist[] = {
    0x7f7fffff,   // VFPU_HUGE  [3.40282346e+38] (max exp & mantissa)
    0x3fb504f3,   // SQRT(2)    [1.41421353e+00]
    0x3f3504f3,   // SQRT(1/2)  [7.07106769e-01]
    0x3f906ebb,   // 2/SQRT(PI) [1.12837922e+00]
    0x3f22f983,   // 2/PI       [6.36619746e-01]
    0x3ea2f983,   // 1/PI       [3.18309873e-01]
    0x3f490fdb,   // PI/4       [7.85398185e-01]
    0x3fc90fdb,   // PI/2       [1.57079637e+00]
    0x40490fdb,   // PI         [3.14159274e+00]
    0x402df854,   // e          [2.71828174e+00]
    0x3fb8aa3b,   // LOG2(e)    [1.44269502e+00]
    0x3ede5bd9,   // LOG10(e)   [4.34294492e-01]
    0x3f317218,   // LOGe(2)    [6.93147182e-01]
    0x40135d8e,   // LOGe(10)   [2.30258512e+00]
    0x40c90fdb,   // 2PI        [6.28318548e+00]
    0x3f060a92,   // PI/6       [5.23598790e-01]
    0x3e9a209b,   // LOG10(2)   [3.01030009e-01]
    0x40549a78,   // LOG2(10)   [3.32192802e+00]
    0x3f5db3d7,   // SQRT(3)/2  [8.66025388e-01]
  };
  return cntlist[cnum - 1];
}
```

vcst.t

Load special constant

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	1	imval					1	0	0	0	0	0	0	0	0	rd						

Syntax

vcst.t rd, imm5

Description

Loads a predefined indexed floating point constant specified by the immediate field

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = fpcst(imval)
rd[1] = fpcst(imval)
rd[2] = fpcst(imval)
```

Used functions

```
uint32_t fpcst(uint8_t cnum) {
  const uint32_t cntlist[] = {
    0x7f7fffff,   // VFPU_HUGE  [3.40282346e+38] (max exp & mantissa)
    0x3fb504f3,   // SQRT(2)    [1.41421353e+00]
    0x3f3504f3,   // SQRT(1/2)  [7.07106769e-01]
    0x3f906ebb,   // 2/SQRT(PI) [1.12837922e+00]
    0x3f22f983,   // 2/PI       [6.36619746e-01]
    0x3ea2f983,   // 1/PI       [3.18309873e-01]
    0x3f490fdb,   // PI/4       [7.85398185e-01]
    0x3fc90fdb,   // PI/2       [1.57079637e+00]
    0x40490fdb,   // PI         [3.14159274e+00]
    0x402df854,   // e          [2.71828174e+00]
    0x3fb8aa3b,   // LOG2(e)    [1.44269502e+00]
    0x3ede5bd9,   // LOG10(e)   [4.34294492e-01]
    0x3f317218,   // LOGe(2)    [6.93147182e-01]
    0x40135d8e,   // LOGe(10)   [2.30258512e+00]
    0x40c90fdb,   // 2PI        [6.28318548e+00]
    0x3f060a92,   // PI/6       [5.23598790e-01]
    0x3e9a209b,   // LOG10(2)   [3.01030009e-01]
    0x40549a78,   // LOG2(10)   [3.32192802e+00]
    0x3f5db3d7,   // SQRT(3)/2  [8.66025388e-01]
  };
  return cntlist[cnum - 1];
}
```

vcst.q

Load special constant

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	1	1	imval					1	0	0	0	0	0	0	0	1	rd						

Syntax

vcst.q rd, imm5

Description

Loads a predefined indexed floating point constant specified by the immediate field

Instruction performance

Throughput: 1 cycles/instruction
Latency: 3 cycles

Allowed prefixes

- rd: Full support (masking and saturation)

Pseudocode

```
rd[0] = fpcst(imval)
rd[1] = fpcst(imval)
rd[2] = fpcst(imval)
rd[3] = fpcst(imval)
```

Used functions

```
uint32_t fpcst(uint8_t cnum) {
  const uint32_t cntlist[] = {
    0x7f7fffff,   // VFPU_HUGE  [3.40282346e+38] (max exp & mantissa)
    0x3fb504f3,   // SQRT(2)    [1.41421353e+00]
    0x3f3504f3,   // SQRT(1/2)  [7.07106769e-01]
    0x3f906ebb,   // 2/SQRT(PI) [1.12837922e+00]
    0x3f22f983,   // 2/PI       [6.36619746e-01]
    0x3ea2f983,   // 1/PI       [3.18309873e-01]
    0x3f490fdb,   // PI/4       [7.85398185e-01]
    0x3fc90fdb,   // PI/2       [1.57079637e+00]
    0x40490fdb,   // PI         [3.14159274e+00]
    0x402df854,   // e          [2.71828174e+00]
    0x3fb8aa3b,   // LOG2(e)    [1.44269502e+00]
    0x3ede5bd9,   // LOG10(e)   [4.34294492e-01]
    0x3f317218,   // LOGe(2)    [6.93147182e-01]
    0x40135d8e,   // LOGe(10)   [2.30258512e+00]
    0x40c90fdb,   // 2PI        [6.28318548e+00]
    0x3f060a92,   // PI/6       [5.23598790e-01]
    0x3e9a209b,   // LOG10(2)   [3.01030009e-01]
    0x40549a78,   // LOG2(10)   [3.32192802e+00]
    0x3f5db3d7,   // SQRT(3)/2  [8.66025388e-01]
  };
  return cntlist[cnum - 1];
}
```

vnop

Nop (no operation)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Syntax

vnop

Description

Does nothing and wastes one VFPU cycle. Used to avoid pipeline hazards. This instruction does consume prefixes.

vflush

Write buffer flush

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0	1

Syntax

vflush

Description

Waits until the write buffer has been flushed

vsync

Pipeline synchronize

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0

Syntax

vsync

Description

Waits until all operations in the VFPU pipeline have completed

vpfxs

Source prefix

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	0	0	imm24																							

Syntax

vpfxs imm24

Description

Sets the prefix operation code in the VFPU_PFXS ($128) register

Notes

Overrides any previous state of the VFPU_PFXS register.

Only the 20 lowest significant bits are set.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 1 cycles

vpfxt

Target prefix

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	0	1	imm24																							

Syntax

vpfxt imm24

Description

Sets the prefix operation code in the VFPU_PFXT ($129) register

Notes

Overrides any previous state of the VFPU_PFXT register.

Only the 20 lowest significant bits are set.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 1 cycles

vpfxd

Destination prefix

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	0	imm24																							

Syntax

vpfxd imm24

Description

Sets the prefix operation code in the VFPU_PFXD ($130) register

Notes

Overrides any previous state of the VFPU_PFXD register.

Only the 12 lowest significant bits are set.

Instruction performance

Throughput: 1 cycles/instruction
Latency: 1 cycles

Known VFPU bugs/errata

The VFPU has some known bugs or errata. Some of the bugs have been fixed in later hardware revisions, but some others have been *kept* for compatibility reasons.

vhtfm output register written incorrectly

Instructions vhtfm2.p and vhtfm3.t feature a bug that affects all PSP models. This bug is triggered whenever the output register `rd` is 64 or higher, which is incidentally the reason why this bug doesn't affect vhtfm4.q.

Registers below 64 are left and top aligned registers, that is, registers in the form of `RX0Y` or `CXY0`. However registers above 64 are in the form `RX1Y`, `RX2Y`, `CXY1` or `CXY2`, that is, they are *shifted* by one or two columns or rows (not at the edge of the matrix). Whenever these registers are used the instruction makes a mistake writing the register and shifts the row or column by an incorrect number of elements, causing an unvoluntary *corruption* in the output value.

This writing error seems to be deterministic. For vhtfm2.p the result is written in one element shifted to the left/up. For vhtfm3.t the shift happens to the right/down, wrapping around the edges of the matrix. You can find the tests and examples in `psp-tests/manual/vfpu-bugs.c`

ulv.q (lvl.q / lvr.q) register corruption

In PSP 1000 devices, the lvl.q and lvr.q instructions (which are usually expanded from ulv.q macros) present a bug (fixed in 2000 and later models). When any of these instructions is executed, the CPU corrupts the FPU register bank (that is, Coprocessor 1, which is unrelated to the VFPU).

The bug causes an unexpected write to an FPU register whenever the instruction is executed. The value being written is apparently whatever value was left in the coprocessor bus (which in some incidental cases causing no corruption if the previous and new values are identical). The bus seems to be used by mfc1/mtc1 and some other COP1 instructions.

The corrupted register is deterministic, its value is derived from the VFPU destination register. The lowest 5 bits of the register indicate the FPU register that will be corrupted (ie. `C000` corrupts `$f0`, `C010` corrupts `$f1` and so on). As a workaround in inline VFPU assembly, it is possible to designate the register as *clobbered*, which will make gcc assume it was corrupted/written, thus using other registers or saving and restoring it.

References

Most of the information comes from various sources that are either incomplete or in a form that is not easy to read as documentation. Some information comes from talks and chats with members of the PSP community or sources thare are now unfortunately dead.

Some sources contain errors, incorrect or imprecise information. No source should be taken without question and every assertion validated.

- yet another PlayStationPortable Documentation
- PPSSPP emulator
- PS2 dev forums