

Valhall Instruction Set Reference

Portions of this work derive from ISA.xml, which is SPDX-MIT:

Copyright ©2021 Collabora Ltd.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice (including the next paragraph) shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

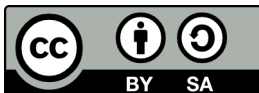
Arm Limited has registered trademarks and uses trademarks. For a list of trademarks of Arm, please see their Trademark list page at <https://www.arm.com/company/policies/trademarks/arm-trademark-list>. Arm® is a registered trademark of Arm. Mali™ is a trademark of Arm.

This work was produced by Collabora, Ltd. independent of Arm with no affiliation.

The work is provided “as is”, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. in no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

Copyright ©2021 Collabora, Ltd.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License (SPDX-CC-BY-SA-4.0). To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.



Change log

2021-07-20: Initial publication.

2021-07-21: Document restrictions on uniforms/constants. Correct various instructions.

2021-07-27: Document restriction on staging registers. Correct various instructions.

2021-08-02: Document execution units for each instruction and update performance characteristics.

2021-08-04: Document IEEE 754 rules for clamps. Document MUX.i32 modes.

2021-12-12: Document texture resource descriptor.

2022-01-27: Document resource access instructions in more depth.

Intro

Valhall is the instruction set used in the latest generation of Arm Mali GPUs, including:

| Name | Codename | Major | Minor |
|------------|----------|-------|-------|
| Mali G77 | Trym | 9 | 0 |
| Mali G57 | Natt-A | 9 | 1 |
| Mali G78 | Borr | 9 | 2 |
| Mali G57 | Natt-B | 9 | 3 |
| Mali G68 | Ottr | 9 | 4 |
| Mali G78AE | Borr-AE | 9 | 5 |

Overview

Valhall is a *linearization* of Bifrost, its predecessor instruction set. Recall Bifrost has instructions paired in tuples, tuples grouped in clauses, and machine state specified in the header of each clause. In Bifrost, it is the compiler’s responsibility to group instructions as part of scheduling. This simplifies the hardware, at the expense of a much more complicated compiler and worse utilization of hardware resources on real workloads.

Valhall replaces Bifrost’s static scheduling with a dynamic hardware scheduler, making it a superscalar architecture. At the instruction level, it is a simple evolution of Bifrost.

Valhall retains Bifrost’s SIMD-within-a-word semantics. The hardware natively supports 8-bit, 16-bit, and 32-bit operations. While 32-bit operations are scalar, 16-bit operations are vectorized in pairs, and 8-bit operations are vectorized in quads. In theory, this permits 16-bit operations to have doubled throughput compared to 32-bit operations.

Valhall improves on Bifrost’s 16-bit support by properly supporting write masks on most 16-bit destinations, so programs using 16-bit operations get register pressure halved even when vectorization is impossible. Additionally, write masks on 32-bit floating point operations have the semantic of converting the result to 16-bit.

Dependencies

Valhall executes new instructions while prior high-latency instructions are still in flight. This allows the architecture to hide the latency of memory accesses, by issuing multiple memory accesses together or executing unrelated arithmetic while the results are pending. When the results of a memory access are required, the compiler must explicitly wait on the previous instruction, inserting a dependency.

Recall on Bifrost, every clause is assigned one of eight “dependency slots”, and every clause can wait on any subset of these slots. Rather than specifying in every instruction, Bifrost specifies this information in the clause header, as a compromise between precision and code size.

Valhall has no clauses (and hence no clause headers), so it must specify this information in the instructions themselves. This becomes practical by making two simplifications to Bifrost:

First, general instructions only need to specify the dependencies. Arithmetic instructions do not need a dependency slot assigned. The few instructions that *do* need dependency slots can specify their slot in another part of the instruction.

Second, Valhall uses only 4 slots instead of 8, saving 4-bits for every instruction. This simplification is practical, as the performance gain from adding extra slots levels off quickly.

Obscurely, instructions that store to memory also require dependency slots assigned. Waiting on these slots allows subgroup memory barriers to be implemented inexpensively.

Branching

Valhall has a single general-purpose branch instruction, branching to a relative offset (in instructions) if its source is nonzero. Higher level control flow is created by chained together branches with comparison instructions. This simplification represents a departure from Bifrost, which specified conditions within the many branch instructions.

Valhall is a warp-based architecture, grouping 16 threads into warps. Divergence of threads within a warp carries a performance penalty. Divergence is handled in hardware, but the compiler must insert some hints to ensure divergence is handled correctly. Namely, the `.reconverge` action is required on any instruction whose successor may be executed with a different execution mask than it. That includes all divergent branches, as well as the last instruction of blocks with divergent fallthrough.

Indirect access to attributes and texture handles must not be divergent. If divergent access is required, the compiler must lower to an if-chain predicated on lane ID:

```
value = 0;
if (lane == 0)
    value = load()
else if (lane == 1)
    value = load()
...
else if (lane == MAX_LANE)
    value = load()
```

As Valhall warps are 16 threads, this requires about 100 instructions! Indirect access should be avoided on Valhall, unless the compiler can prove the index is not divergent.

Texture instructions

Texture instructions are unique in their register requirements; depending on the specific texturing operation, up to a dozen registers can be used on some GPUs. As Valhall is limited in instruction size, there is no room to encode each source register separately. Instead, the *base* register is specified in the instruction, and sources are found in the subsequent registers in the following order:

1. X coordinate (floating-point)
2. Y coordinate (floating-point)
3. Z coordinate (floating-point)
4. Shadow comparison value (floating-point)
5. Texel offset (as a packed 8-bit vector, X in the bottom byte, Y in the second byte)
6. Explicit level-of-detail (as a 16-bit fixed-point, encoding 8:8)
7. Level-of-detail bias (as a 16-bit fixed-point, encoding 8:8)

Shading languages require the use of helper threads, which contribute to screen-space derivative calculations (including the level-of-detail selection in texture instructions), but do not correspond to rasterized pixels. Usually, helper threads do not need to execute texture instructions once the level-of-detail has been selected. Skipping texturing on helper threads can save memory bandwidth. Valhall skips texturing for helper threads if the `.skip` bit is set. `.skip` should be set if the results of texturing do not contribute to future texture, derivative, or subgroup operations as determined by data flow analysis. Once helper invocations are no longer required, the `.td` action should be used to terminate the execution of the unused threads.

Texture projection is not supported by Valhall. Lower to `FRCP.f32` and `FMA.f32` and then do the non-projection texture operation.

Texture resource descriptor

Due to the fixed encoding, there is not space in the texture instruction to encode both texture and sampler indices as either immediates or registers. Instead, this information is encoded in a texture resource descriptor, packed into a single 64-bit word as follows, and passed as the first argument of texture access instructions.

Table 2: Texture Resource Descriptor

| Bits | Value |
|-------|---------------|
| 0-10 | Sampler index |
| 11-15 | Sampler table |
| 16-26 | Texture index |
| 27-31 | Texture table |
| 32-35 | Offset X |
| 36-39 | Offset Y |

The sampler and texture descriptors are respectively found at the given index of the respective table passed in the GPU data structures.

The offsets allow fixing a small immediate nonnegative texel offset, useful for nonnegative arguments to `textureOffset`. Negative offsets must be handled with an explicit offset passed via a staging register; this is slightly less efficient.

Typically, this descriptor is packed at compile-time and passed as a fast-access uniform.

Cube maps

Before texturing from a cube map, the coordinates must be transformed using the Valhall instructions `CUBE_SSEL`, `CUBE_TSEL`, `CUBEFACE1`, and `CUBEFACE2`. These act as their Bifrost counterparts:

- `CUBEFACE1` takes the X, Y, and Z coordinates, and computes $\max \{ |X|, |Y|, |Z| \}$.
- `CUBEFACE2` takes the X, Y, and Z coordinates, and selects the cube face
- `CUBE_SSEL` takes the Z and X coordinates and the cube face, and selects the S coordinate.
- `CUBE_TSEL` takes the Y and Z coordinates and the cube face, and selects the T coordinate.

These operations are used as building blocks for the full cube face transformation. OpenGL specifies that an input vector (x, y, z) is transformed to the vector

$$\begin{bmatrix} \frac{1}{2} \left(\frac{s}{\max \{x, y, z\}} + 1 \right) \\ \frac{1}{2} \left(\frac{t}{\max \{x, y, z\}} + 1 \right) \end{bmatrix}$$

This may be rewritten as

$$\begin{aligned} r &= \frac{1}{2} \cdot (\max \{x, y, z\})^{-1} \\ x' &= s \cdot r + \frac{1}{2} \\ y' &= t \cdot r + \frac{1}{2} \end{aligned}$$

r is computed by `CUBEFACE1.f32`, `FRCP.f32`, and `FMA.f32`. x' , y' are each computed as `FMA.f32`. To workaround numerical issues, it's additionally required to clamp x' , y' to $[0, 1]$; this saturation is free as an output modifier on the `FMA.f32` instructions.

Once the face index and coordinates are computed, they must be packed into Valhall's *Cube Map Descriptor*:

Table 3: Cube Map Descriptor

| Bits | Value |
|-------|--------------|
| 0-28 | S coordinate |
| 29-31 | Face index |
| 32-63 | T coordinate |

Storing only the bottom 29-bits of the 32-bit floating point S coordinate suffices, since the hardware may infer the top 3-bits given the range restrictions of the output of the cube map transform.

The Cube Map Descriptor may be packed efficiently with a bitwise `MUX.i32` instruction, which acts as a per-bit conditional select. By fixing the mask of bits that the S coordinates occupies, the bottom word may be constructed by muxing between the S coordinate and the face index. The upper word requires no packing.

Putting it together, we get the full code sequence for the cube face transform:

```

CUBEFACE1.f32    m, x, y, z
CUBEFACE2.f32    f, x, y, z
FRCP.f32         n, m
FMA.f32          r, n, 0.5, -0.0
CUBE_SSEL.f32    s, z, x, f
CUBE_TSEL.f32    t, y, z, f
FMA.f32.clamp_0_1 x', s, r, 0.5
FMA.f32.clamp_0_1 y', t, r, 0.5
MUX.i32          x', x', f, #0x1FFFFFFF

```

Register file

Like Bifrost, Valhall has 64 registers available, each 32-bits. There is a trade off between register usage and thread count:

| Register usage | Threads |
|----------------|---------|
| ≤ 32 | Full |
| > 32 | Half |

Due to the hardware preloading some state in high registers, the register file is discontinuous with full threads: the 32 available registers are $[R0, R15]$ and $[R48, R63]$.

Access to the register file has a power cost. Previous Mali generations provided “temporary” or “bypass” registers, to pass data within a clause or VLIW bundle without touching the register file. Valhall does not have architecturally-visible temporary registers, as it lacks clauses or VLIW bundles. Instead, Valhall passes register access through a forwarding buffer, offering a dynamic alternative to static temporaries. Although this mechanism is managed in hardware, most instruction sources have a `.discard` hint indicating they WILL NOT be used in any later instructions. The `.discard` hint indicates the hardware MAY evict the referenced register from the forwarding buffer. The register's value after a discard is undefined and should not be read. As short-hand, the discard hint is indicated in Mesa's disassembler by prefixing register sources with a backtick `.

Programmatic blending is implemented via blend shaders, which use the following ABI:

Table 5: ABI for blend shaders

| Register | Value |
|----------|--------------------------|
| R0 | Colour component #0 |
| R1 | Colour component #1 |
| R2 | Colour component #2 |
| R3 | Colour component #3 |
| R52 | Stack pointer, low word |
| R53 | Stack pointer, high word |
| R54 | Return address, low word |

The stack pointer may be calculated by adding the amount of stack used by the caller (the fragment shader) to the base thread local storage pointer, with a sequence like:

```
IADD_IMM.i32.ts r52, tls_ptr, #0x10
MOV.i32.ts r53, tls_ptr.hi
```

The return address may be calculated by adding the current program counter with the length of an instruction (8) times the number of instructions to the next BLEND instruction minus 1. Then when the blend shader jumps back to this address (plus a single instruction), it will blend the next render target.

```
IADD_IMM.i32.id r54, program_counter, #0x8
```

The final render target should set r54 to zero to terminate execution early.

This interface improves on Bifrost's handling of blend shaders by:

- Enabling both the blend shader and the fragment shader to use the stack
- Allowing position-independent blend shaders without passing return offsets in a side channel.

The following registers may be preloaded with hardware state by setting the appropriate flags in the preload descriptor:

Table 6: Preloaded registers in vertex shaders

| Register | Value |
|----------|----------------------|
| R59 | Linear (unfolded) ID |
| R60 | Vertex ID |
| R61 | Instance ID |

Table 7: Preloaded registers in fragment shaders

| Register | Value |
|----------|---|
| R58 | Facingness, bottom bit set for front facing |
| R59 | Fragment coordinates X/Y packed as 16-bit integers |
| R60 | Sample coverage mask |
| R61 | Sample coverage mask input in lower half, sample ID in byte 3 |

Table 8: Preloaded registers in compute shaders

| Register | Value |
|----------|---|
| R55 | Local invocation ID (dimensions #0 and #1, low/high halves) |
| R56 | Local invocation ID (dimension #2, low half) |
| R57 | Work group ID (dimension #0) |
| R58 | Work group ID (dimension #1) |
| R59 | Work group ID (dimension #2) |
| R60 | Global invocation ID (dimension #0) |
| R61 | Global invocation ID (dimension #1) |
| R62 | Global invocation ID (dimension #2) |

Transcendental operations

Transcendental operations on Valhall match Bifrost, using the same building block instructions pieced together by the compiler in the same way. Common operations are explained below. See Mesa’s Bifrost compiler as a reference.

Exponentials

Valhall provides fast computation of base-2 exponents with the `FEXP2.f32` instruction. However, there is a catch: it requires its input in as an 8:24 fixed-point. The floating-point itself is passed as a second argument and used only for numerical compliance in special cases.

Converting a 32-bit floating-point input to 8:24 fixed-point may be done by multiplying by 2^{24} (adding 24 to the exponent via `RSCALE`) and converting to integer. The full code sequence to compute $y = 2^x$ is therefore

```
RSCALE.f32 scaled, x, #24
F32_TO_S32 fixed, scaled
FEXP2.f32 y, fixed, x
```

To compute arbitrary exponents, recall the identity

$$b^x = \left(2^{\log_2(b)}\right)^x = 2^{x \cdot \log_2(b)}$$

For constant base, the \log_2 of the base may be precomputed. Naively computing exponents this way would require an extra multiplication, but we may make two simplifications:

- Valhall has a four-source `FMA_RSCALE.f32` instruction performing a multiply-add in its first three sources and an exponent adjust given by the fourth.
- Multiplying by 2^{24} does not affect the special cases of \exp_2 , so we may choose to pass the scaled argument to `FEXP2.f32` instead of the original one, allowing us to fuse the multiply.

Putting it together gives the code for other bases:

```
FMA_RSCALE.f32 scaled, x, #log2(base), #0.neg, #24
F32_TO_S32 fixed, scaled
FEXP2.f32 y, fixed, scaled
```

Sine and cosine

For \sin , \cos , Valhall contains coarse lookup tables accessible with the `FSIN_TABLE.u6`, `FCOS_TABLE.u6` instructions. These instructions multiply the bottom 6-bits of their input by $\pi/32$ and return the resulting \sin or \cos

value. They are used to calculate sin, cos via a Taylor approximation:

$$f(x + e) = f(x) + ef'(x) + \frac{e^2}{2}f''(x)$$

$$\sin(x + e) = \sin(x) + e \cos(x) - \frac{e^2}{2} \sin(x)$$

$$\cos(x + e) = \cos(x) - e \sin(x) - \frac{e^2}{2} \cos(x)$$

As a numerical trick, we introduce the magic constant `0x49400000`, with the curious property that – when interpreted as a floating-point and added to a floating-point value $x \cdot \frac{2}{\pi}$ – approximately equals $\frac{32}{\pi}(x \bmod 2\pi)$ in the bottom 6-bits. This allows the domain transformation to be done as a single fused floating-point multiply-add with the magic constants.

As one more trick, we use the special `FMA_RSCALE.f32` instruction, which acts like `FMA.f32` in its first three sources but biases the exponent by the value of the fourth source. Recalling that a floating-point is encoded as $m \cdot 2^e$ for mantissa m and exponent e , we may divide by two by biasing the exponent by -1 , as $\frac{1}{2}(m \cdot 2^e) = m \cdot 2^{e-1}$.

Note that we require four magic constants that are not directly encodable. These constants must be lowered to FAU slots. No FAU slot is needed for the first addition, however, as the `FADD_IMM.f32` operation is available.

Putting it together gives a code sequence for sin:

```
FMA.f32 x_u6, x, #TWO_OVER_PI, #SINCOS_BIAS
FADD_IMM.f32 temp, x_u6, #-SINCOS_BIAS
FMA.f32 e, temp, #MPI_OVER_TWO, x
FSIN_TABLE.u6 sinx, x_u6
FCOS_TABLE.u6 cosx, x_u6
FMA_RSCALE.f32 e2_over_2, e, e, -0.0, #-1
FMA.f32 quadratic, e2_over_2.neg, sinx, -0.0
FMA.f32.m1_1 temp2, e, cosx, quadratic
FADD.f32 dst, temp2, sinx
```

and similarly for cos:

```
FMA.f32 x_u6, x, #TWO_OVER_PI, #SINCOS_BIAS
FADD.f32 temp, x_u6, #-SINCOS_BIAS
FMA.f32 e, temp, #MPI_OVER_TWO, x
FSIN_TABLE.u6 sinx, x_u6
FCOS_TABLE.u6 cosx, x_u6
FMA_RSCALE.f32 e2_over_2, e, e, -0.0, #-1
FMA.f32 quadratic, e2_over_2.neg, cosx, -0.0
FMA.f32.m1_1 temp2, e, sinx.neg, quadratic
FADD.f32 dst, temp2, cosx
```

Sources

Regular sources are specified as an 8-bit structure.

| Name | Bits |
|-------|------|
| Value | 0:5 |
| Mode | 6:7 |

Mode is an enumeration with the following values.

| Index | Name |
|-------|-----------------------|
| 0 | Register |
| 1 | Register with discard |
| 2 | Uniform |
| 3 | Immediate |

Value is the index of 32-bit register in a register mode, the index of a 32-bit uniform in uniform mode, or an index into an immediate table in an immediate mode. The discard mode is used to discard the register after reading.

Destinations

Regular destinations are specified as an 8-bit structure.

| Name | Bits |
|-----------------|------|
| Register | 0:5 |
| Write low half | 6 |
| Write high half | 7 |

The write low/high acts as a 16-bit write mask. At least one half **MUST** be written in instruction. Therefore, instructions with no destinations still require a placeholder where the destination otherwise would be; this should be the constant 0xC0.

Staging registers

Message-passing instructions have special register access requirements compared to arithmetic instructions. In particular, they can write multiple subsequent registers, and they may need to reuse a register base as both a read and write due to encoding limitations. These instructions do not use the regular source/destination mechanism, and instead use the following 8-bit staging register structure.

| Name | Bits |
|----------|------|
| Register | 0:5 |
| Read | 6 |
| Write | 7 |

The read/write flags specify the direction of transfer; both may be set simultaneously, but at least one **MUST** be set. In a few cases, both are set even when transferring only in a single logical direction. In a few other cases, only the first 6-bits are stored, and the access flags are implied.

If multiple subsequent staging registers are accessed, the base must be aligned to 2. However, even if 4 registers are accessed, it is not necessary to align to 4, only to 2. This restriction allows the hardware to use a 64-bit data

path without handling unaligned access, which is more efficient. This restriction does not apply if only a single register is accessed.

Instruction metadata

The last 7-bits of every instruction contains metadata for the instruction, playing a similar role to the Bifrost clause header.

| Name | Bits |
|-----------------|------|
| Immediate mode | 0:1 |
| Action | 2:5 |
| Action mode | 6 |
| <i>Reserved</i> | 7 |

The immediate mode is an enumeration controlling how immediates sources are interpreted. Leaving it zero allows the instruction to access the immediate table, but it can be set to access various state of the shader core. See the corresponding enumerations.

If the action mode bit is set, then action is an enumeration controlling machine behaviour around the instruction's execution. If the action mode bit is clear, then action is a bitfield specifying dependencies to wait on before executing the *next* instruction.

| Name | Bits |
|-----------------|------|
| Wait on slot #0 | 0 |
| Wait on slot #1 | 1 |
| Wait on slot #2 | 2 |

Uniform/constant restrictions

Valhall imposes restrictions on access to uniforms and immediates in order to simplify the hardware.

- An instruction may access no more than a single 64-bit uniform slot.
- An instruction may access no more than 64-bits of combined uniforms and constants.
- An instruction may only access uniforms in the default immediate mode.
- An instruction may access no more than a single special immediate (e.g. `lane_id`).

As Valhall is encoded regularly, it is possible to assemble programs exceeding these limits. Such programs are invalid; nevertheless, implementations MAY execute them with UNDEFINED results. Mali G78 will execute such an instruction without faulting, but will select a *single* 64-bit slot corresponding to the maximum uniform index and will overwrite the looked up result, producing incorrect results. These limitations enable hardware optimizations.

These limits are inherited from Bifrost, where both constants and uniforms are explicitly loaded to a “fast access uniform (FAU)” port, capable of loading a single 64-bit value every two instructions. An instruction like `FMA.f32 r2, r2, u4, u7` cannot be encoded, since the uniforms come from different 64-bit slots. In this situation, the Bifrost compiler must reorder the uniforms or insert a move to solve the encoding hazard. Bifrost's FAU-RAM hardware is reused on Valhall, even though the hardware limitations are no longer implicit in the encoding.

For example, to compile `fma(fma(a, #b, #c), #b, #d)`, the constants must be lowered to uniforms. The optimal compile requires duplicating `#b` to two separate uniforms:

```
FMA.f32 r2, r2, u0, u1
FMA.f32 r2, r2, u2, u3
```

For another example, the program `FMA.f32 r0, u4, 0x0, u5` is invalid and will produce incorrect results. The following are legal versions which will function correctly:

```
MOV.i32 r0, u5
FMA.f32 r0, u4, 0x0, r0

MOV.i32 r0, 0x0
FMA.f32 r0, u4, r0, u5
```

In practice, the combined limit for constants and uniforms is rarely hit if pre-shaders are used in tandem with constant folding. However, the limit is visible on shaders that use a four-source conditional select. The optimal compile of `a > #b ? #c : #d` with unique constants require two instructions:

```
IADD_IMM.i32 r1, 0x0, #b
CSEL.f32.gt r0, a, r1, #c, #d
```

Varyings

Unlike older Mali designs, the Valhall hardware allocates memory for varyings automatically. The address of the allocated memory is available as `LEA_ATTR`. It is stored to with standard `STORE` instructions with the appropriate segment (position or varying).

Valhall supports a black hole address. Writes to any address with the top bit set are discarded without faulting. The canonical black hole address is $BH = (1 \ll 63)$. Black hole addresses are useful in the construction for point size writes from a position shader:

```
address = is_writing_points ? LEA_ATTR(r59 + 4) : (1 << 63)
STORE.i16.pos @psiz, address
```

Point sizes should be clamped to hardware bounds $[\frac{1}{2}, 1024]$.

Table of immediates in the Valhall ISA

This immediates are accessible in (almost) any instruction, provided the immediate mode is kept to the default. They optimize for the most common immediate values; any immediate listed here may be used without taking up a uniform slot or a register. Most integer instructions can access separate half-words and individual bytes via swizzles on the source.

| Index | Value | Description |
|-------|------------|--|
| 0 | 0x00000000 | Zero |
| 1 | 0xFFFFFFFF | All ones; integer -1 |
| 2 | 0x7FFFFFFF | Maximum integer; floating-point NaN |
| 3 | 0xFAFCDFE | Integers $(-2, -3, -4, -5)$ |
| 4 | 0x01000000 | 16-bit integer 2^8 |
| 5 | 0x80002000 | Multiples of 16 (0, 32, 0, 128) |
| 6 | 0x70605030 | Multiples of 16 (48, 80, 96, 112) |
| 7 | 0xC0B0A090 | Multiples of 16 (144, 160, 176, 192) |
| 8 | 0x03020100 | Integers (0, 1, 2, 3) |
| 9 | 0x07060504 | Integers (4, 5, 6, 7) |
| 10 | 0x0B0A0908 | Integers (8, 9, 10, 11) |
| 11 | 0x0F0E0D0C | Integers (12, 13, 14, 15) |
| 12 | 0x13121110 | Integers (16, 17, 18, 19) |
| 13 | 0x17161514 | Integers (20, 21, 22, 23) |
| 14 | 0x1B1A1918 | Integers (24, 25, 26, 27) |
| 15 | 0x1F1E1D1C | Integers (28, 29, 30, 31) |
| 16 | 0x3F800000 | Float 1.0 |
| 17 | 0x3DCCCCD | Float 0.1 |
| 18 | 0x3EA2F983 | Float $1/\pi$ |
| 19 | 0x3F317218 | Float $\log(2)$ |
| 20 | 0x40490FDB | Float π |
| 21 | 0x00000000 | Float 0.0 |
| 22 | 0x477FFF00 | Float $65535.0 = 2^{16} - 1$ |
| 23 | 0x5C005BF8 | Half-float $(255.0, 256.0) = (2^8 - 1, 2^8)$ |
| 24 | 0x2E660000 | Half-float $0.1 = 1/10$ |
| 25 | 0x34000000 | Half-float $0.25 = 2^{-2}$ |
| 26 | 0x38000000 | Half-float $0.5 = 2^{-1}$ |
| 27 | 0x3C000000 | Half-float $1.0 = 2^0$ |
| 28 | 0x40000000 | Half-float $2.0 = 2^1$ |
| 29 | 0x44000000 | Half-float $4.0 = 2^2$ |
| 30 | 0x48000000 | Half-float $8.0 = 2^3$ |
| 31 | 0x42480000 | Half-float π |

Enumerations

This section describes each enumeration used in the Valhall ISA. Enumerations are found in the instruction metadata and as modifiers in individual instructions.

Action

Every Valhall instruction can perform an action, like wait on dependency slots. A few special actions are available, specified in the instruction metadata from this enum. The `wait0126` action is required to wait on dependency slot #6 and should be set on the instruction immediately preceding `ATEST`. The `barrier` action may be set on any instruction for subgroup barriers, and should particularly be set with the `BARRIER` instruction for global barriers. The `td` action only applies to fragment shaders and is used to terminate helper invocations, it should be set as early as possible after helper invocations are no longer needed as determined by data flow analysis. The `return` action is used to terminate the shader, although it may be overloaded by the `BLEND` instruction.

The `reconverge` action is required on any instruction immediately preceding a possible change to the mask of active threads in a subgroup. This includes all divergent branches, but it also includes the final instruction at the end of any basic block where the immediate successor (fallthrough) is the target of a divergent branch.

| Index | Value |
|-------|-------------------------|
| 0 | <code>wait0126</code> |
| 1 | <code>barrier</code> |
| 2 | <code>reconverge</code> |
| 3 | <i>Reserved</i> |
| 4 | <i>Reserved</i> |
| 5 | <code>td</code> |
| 6 | <i>Reserved</i> |
| 7 | <code>return</code> |

Immediate mode

Selects how immediates sources are interpreted.

The default value is none.

| Index | Value | Description |
|-------|-----------------|-------------------------|
| 0 | none | No special immediates |
| 1 | ts | Thread storage pointers |
| 2 | <i>Reserved</i> | |
| 3 | id | Thread identification |

Thread storage pointers

Situated between the immediates hard-coded in the hardware and the uniforms defined purely in software, Valhall has a some special “constants” passing through data structures. These are encoded like the table of immediates, as if special constant i were lookup table entry $32 + i$. These special values are selected with the `.ts` modifier.

| Index | Value | Description |
|-------|-------------------------|--|
| 0 | <i>Reserved</i> | |
| 1 | <i>Reserved</i> | |
| 2 | <code>tls_ptr</code> | Thread local storage base pointer (low word) |
| 3 | <code>tls_ptr_hi</code> | Thread local storage base pointer (high word) |
| 4 | <i>Reserved</i> | |
| 5 | <i>Reserved</i> | |
| 6 | <code>wls_ptr</code> | Workgroup local storage base pointer (low word) |
| 7 | <code>wls_ptr_hi</code> | Workgroup local storage base pointer (high word) |

Thread identification

Situated between the immediates hard-coded in the hardware and the uniforms defined purely in software, Valhall has a some special “constants” passing through data structures. These are encoded like the table of immediates, as if special constant i were lookup table entry $32 + i$. These special values are selected with the `.id` modifier.

| Index | Value | Description |
|-------|------------------------|-----------------|
| 0 | <i>Reserved</i> | |
| 1 | <i>Reserved</i> | |
| 2 | <i>lane_id</i> | Lane ID |
| 3 | <i>Reserved</i> | |
| 4 | <i>Reserved</i> | |
| 5 | <i>Reserved</i> | |
| 6 | <i>core_id</i> | Core ID |
| 7 | <i>Reserved</i> | |
| 8 | <i>Reserved</i> | |
| 9 | <i>Reserved</i> | |
| 10 | <i>Reserved</i> | |
| 11 | <i>Reserved</i> | |
| 12 | <i>Reserved</i> | |
| 13 | <i>Reserved</i> | |
| 14 | <i>Reserved</i> | |
| 15 | <i>Reserved</i> | |
| 16 | <i>Reserved</i> | |
| 17 | <i>Reserved</i> | |
| 18 | <i>Reserved</i> | |
| 19 | <i>Reserved</i> | |
| 20 | <i>Reserved</i> | |
| 21 | <i>Reserved</i> | |
| 22 | <i>Reserved</i> | |
| 23 | <i>Reserved</i> | |
| 24 | <i>Reserved</i> | |
| 25 | <i>Reserved</i> | |
| 26 | <i>Reserved</i> | |
| 27 | <i>Reserved</i> | |
| 28 | <i>Reserved</i> | |
| 29 | <i>Reserved</i> | |
| 30 | <i>program_counter</i> | Program counter |
| 31 | <i>Reserved</i> | |

Swizzles (8-bit)

The default value is b0123.

| Index | Value |
|-------|-----------------|
| 0 | b0123 |
| 1 | b3210 |
| 2 | b0101 |
| 3 | b2323 |
| 4 | b0000 |
| 5 | b1111 |
| 6 | b2222 |
| 7 | b3333 |
| 8 | b2301 |
| 9 | b1032 |
| 10 | b0011 |
| 11 | b2233 |
| 12 | <i>Reserved</i> |
| 13 | <i>Reserved</i> |
| 14 | <i>Reserved</i> |
| 15 | <i>Reserved</i> |

Lanes (8-bit)

Used to select the 2 bytes for shifts of 16-bit vectors

| Index | Value |
|-------|-----------------|
| 0 | b02 |
| 1 | <i>Reserved</i> |
| 2 | <i>Reserved</i> |
| 3 | <i>Reserved</i> |
| 4 | b00 |
| 5 | b11 |
| 6 | b22 |
| 7 | b33 |
| 8 | <i>Reserved</i> |
| 9 | <i>Reserved</i> |
| 10 | b01 |
| 11 | b23 |
| 12 | <i>Reserved</i> |
| 13 | <i>Reserved</i> |
| 14 | <i>Reserved</i> |
| 15 | <i>Reserved</i> |

Swizzles (16-bit)

The default value is h01.

| Index | Value |
|-------|-----------------|
| 0 | h00 |
| 1 | h10 |
| 2 | h01 |
| 3 | h11 |
| 4 | b00 |
| 5 | b20 |
| 6 | b02 |
| 7 | b22 |
| 8 | b11 |
| 9 | b31 |
| 10 | b13 |
| 11 | b33 |
| 12 | b01 |
| 13 | b23 |
| 14 | <i>Reserved</i> |
| 15 | <i>Reserved</i> |

Swizzles (32-bit)

The default value is none.

| Index | Value |
|-------|-----------------|
| 0 | none |
| 1 | <i>Reserved</i> |
| 2 | h0 |
| 3 | h1 |
| 4 | b0 |
| 5 | b1 |
| 6 | b2 |
| 7 | b3 |

Swizzles (64-bit)

The default value is none.

| Index | Value |
|-------|-----------------|
| 0 | none |
| 1 | <i>Reserved</i> |
| 2 | h0 |
| 3 | h1 |
| 4 | b0 |
| 5 | b1 |
| 6 | b2 |
| 7 | b3 |
| 8 | w0 |
| 9 | <i>Reserved</i> |
| 10 | <i>Reserved</i> |
| 11 | <i>Reserved</i> |
| 12 | <i>Reserved</i> |
| 13 | <i>Reserved</i> |
| 14 | <i>Reserved</i> |
| 15 | <i>Reserved</i> |

Lane (32-bit)

Used for the lane select of BRANCHZ. To use an 8-bit condition, a separate ICMP is required to cast to 16-bit.

The default value is none.

| Index | Value |
|-------|-----------------|
| 0 | none |
| 1 | h0 |
| 2 | h1 |
| 3 | <i>Reserved</i> |

Load lane (8-bit)

The default value is b0.

| Index | Value | Description |
|-------|-------|----------------------------------|
| 0 | b0 | |
| 1 | b1 | |
| 2 | b2 | |
| 3 | b3 | |
| 4 | h0 | Zero-extend to 16-bit, low-half |
| 5 | h1 | Zero-extend to 16-bit, high-half |
| 6 | w0 | Zero-extend to 32-bit |
| 7 | d0 | Zero-extend to 32-bit |

Load lane (16-bit)

The default value is h0.

| Index | Value | Description |
|-------|-----------------|-----------------------|
| 0 | h0 | Low half |
| 1 | h1 | High half |
| 2 | w0 | Zero-extend to 32-bit |
| 3 | d0 | Zero-extend to 64-bit |
| 4 | <i>Reserved</i> | |
| 5 | <i>Reserved</i> | |
| 6 | <i>Reserved</i> | |
| 7 | <i>Reserved</i> | |

Load lane (32-bit)

The default value is w0.

| Index | Value | Description |
|-------|-----------------|-----------------------|
| 0 | w0 | Zero-extend to 64-bit |
| 1 | d0 | |
| 2 | <i>Reserved</i> | |
| 3 | <i>Reserved</i> | |
| 4 | <i>Reserved</i> | |
| 5 | <i>Reserved</i> | |
| 6 | <i>Reserved</i> | |
| 7 | <i>Reserved</i> | |

Load lane (48-bit)

The default value is identity.

| Index | Value |
|-------|-----------------|
| 0 | <i>Reserved</i> |
| 1 | <i>Reserved</i> |
| 2 | <i>Reserved</i> |
| 3 | <i>Reserved</i> |
| 4 | identity |
| 5 | <i>Reserved</i> |
| 6 | <i>Reserved</i> |
| 7 | <i>Reserved</i> |

Load lane (64-bit)

The default value is identity.

| Index | Value |
|-------|-----------------|
| 0 | <i>Reserved</i> |
| 1 | <i>Reserved</i> |
| 2 | <i>Reserved</i> |
| 3 | <i>Reserved</i> |
| 4 | <i>Reserved</i> |
| 5 | <i>Reserved</i> |
| 6 | <i>Reserved</i> |
| 7 | identity |

Load lane (96-bit)

The default value is identity.

| Index | Value |
|-------|-----------------|
| 0 | <i>Reserved</i> |
| 1 | <i>Reserved</i> |
| 2 | <i>Reserved</i> |
| 3 | <i>Reserved</i> |
| 4 | <i>Reserved</i> |
| 5 | <i>Reserved</i> |
| 6 | identity |
| 7 | <i>Reserved</i> |

Load lane (128-bit)

The default value is identity.

| Index | Value |
|-------|-----------------|
| 0 | <i>Reserved</i> |
| 1 | <i>Reserved</i> |
| 2 | <i>Reserved</i> |
| 3 | <i>Reserved</i> |
| 4 | <i>Reserved</i> |
| 5 | <i>Reserved</i> |
| 6 | <i>Reserved</i> |
| 7 | identity |

Round mode

Corresponds to IEEE 754 rounding modes

The default value is `rte`.

| Index | Value | Description |
|-------|------------------|----------------------------|
| 0 | <code>rte</code> | Round to nearest even |
| 1 | <code>rtp</code> | Round to positive infinity |
| 2 | <code>rtn</code> | Round to negative infinity |
| 3 | <code>rtz</code> | Round to zero |

Result type

Comparison instructions like `FCMP` return a boolean but may encode this boolean in a variety of ways. `i1` gives a OpenGL style `0/1` boolean. `m1` gives a Direct3D style `0/~0` boolean. `f1` gives a floating-point `0.0f / 1.0f` boolean. Switching between these modes is useful to fold a boolean type convert into a comparison. `u1` is used internally to implement 64-bit comparisons.

| Index | Value | Description |
|-------|-----------------|----------------------------|
| 0 | <code>i1</code> | Integer 1 |
| 1 | <code>f1</code> | Float 1 |
| 2 | <code>m1</code> | Minus 1 |
| 3 | <code>u1</code> | Low half of 64-bit compare |

Widen

The default value is none.

| Index | Value |
|-------|-----------------|
| 0 | none |
| 1 | h0 |
| 2 | h1 |
| 3 | <i>Reserved</i> |
| 4 | <i>Reserved</i> |
| 5 | <i>Reserved</i> |
| 6 | <i>Reserved</i> |
| 7 | <i>Reserved</i> |

Clamp

Clamp applied to the destination of a floating-point instruction. Note the clamps may be decomposed as two independent bits for `clamp_0_inf` and `clamp_m1_1`, with `clamp_0_1` arising as the composition of `clamp_0_inf` and `clamp_m1_1` in either order.

Clamps are implemented per the SPIR-V specification:

$$\text{clamp}(x, \ell, h) = \min(\max(x, \ell), h)$$

The min/max functions return the other operand if one operand is NaN, and compare $-0 < +0$. That means the following identities hold for Valhall clamps:

$$\text{clamp}(-0.0, 0.0, 1.0) = +0.0$$

$$\text{clamp}(-\text{NaN}, 0.0, 1.0) = +0.0$$

$$\text{clamp}(\text{NaN}, 0.0, 1.0) = +0.0$$

$$\text{clamp}(-0.0, -1.0, 1.0) = -0.0$$

$$\text{clamp}(\text{NaN}, -1.0, 1.0) = -1.0$$

$$\text{clamp}(-\text{NaN}, -1.0, 1.0) = -1.0$$

$$\max(\text{NaN}, 0.0) = +0.0$$

$$\max(-\text{NaN}, 0.0) = +0.0$$

$$\max(-0.0, 0.0) = +0.0$$

This behaviour is consistent with the FMin/FMax/FClamp and NMin/NMax/NClamp rules prescribed by SPIR-V and governed by IEEE-754. As a consequence, substituting these clamps for equivalent minimum/maximum expressions is legal even with strict floating point rules.

The default value is none.

| Index | Value | Description |
|-------|-------------|--------------------|
| 0 | none | Identity |
| 1 | clamp_0_inf | Clamp positive |
| 2 | clamp_m1_1 | Clamp to $[-1, 1]$ |
| 3 | clamp_0_1 | Clamp to $[0, 1]$ |

Condition

Condition code. Type must be inferred from the instruction. IEEE 754 total ordering only applies to floating point compares. “Not equal” and “greater than or less than” are distinguished by NaN behaviour conforming to the IEEE 754 specification.

| Index | Value | Description |
|-------|-------|---------------------------|
| 0 | eq | Equal |
| 1 | gt | Greater than |
| 2 | ge | Greater than or equal |
| 3 | ne | Not equal |
| 4 | lt | Less than |
| 5 | le | Less than or equal |
| 6 | gtlt | Greater than or less than |
| 7 | total | Totally ordered |

Dimension

Texture dimension.

| Index | Value | Description |
|-------|-------|----------------------------|
| 0 | 1d | 1D or buffer |
| 1 | 2d | 2D or 2D array |
| 2 | 3d | 3D or 3D array |
| 3 | cube | Cube map or cube map array |

LOD mode

Level-of-detail selection mode in a texture instruction.

| Index | Value | Description |
|-------|-----------------|---|
| 0 | zero | Set to zero |
| 1 | computed | Computed based on neighboring fragments |
| 2 | <i>Reserved</i> | |
| 3 | <i>Reserved</i> | |
| 4 | explicit | Explicitly specified in a register |
| 5 | computed_bias | Computed based on neighboring fragments added with bias in a register |
| 6 | grdesc | Derived from a gradient descriptor in registers |
| 7 | <i>Reserved</i> | |

Register format

Format of data loaded to / stored from registers for general memory access.

| Index | Value | Description |
|-------|-----------------|--------------------------|
| 0 | <i>Reserved</i> | |
| 1 | <i>Reserved</i> | |
| 2 | f32 | 32-bit floats |
| 3 | f16 | 16-bit floats |
| 4 | s32 | 32-bit signed integers |
| 5 | s16 | 16-bit signed integers |
| 6 | u32 | 32-bit unsigned integers |
| 7 | u16 | 16-bit unsigned integers |

Vector size

Number of channels loaded/stored for general memory access.

The default value is none.

| Index | Value | Description |
|-------|-------|-------------|
| 0 | none | Scalar |
| 1 | v2 | 2 channels |
| 2 | v3 | 3 channels |
| 3 | v4 | 4 channels |

Slot

Dependency slot set on a message-passing instruction that writes to registers. Before reading the destination, a future instruction must wait on the specified slot. Slot #7 is for BARRIER instructions only.

| Index | Value | Description |
|-------|-----------------|-------------|
| 0 | slot0 | Slot #0 |
| 1 | slot1 | Slot #1 |
| 2 | slot2 | Slot #2 |
| 3 | <i>Reserved</i> | |
| 4 | <i>Reserved</i> | |
| 5 | <i>Reserved</i> | |
| 6 | <i>Reserved</i> | |
| 7 | slot7 | Slot #7 |

Store segment

Memory segment written to by a STORE instruction.

| Index | Value | Description |
|-------|--------|---|
| 0 | global | Global or workgroup local memory |
| 1 | pos | Position output (in a position shader) |
| 2 | vary | Varyings with LEA_VARY computed addresses |
| 3 | t1 | Thread local storage |

Subgroup size

Selects the effective subgroup size from subgroup operations. The hardware warps are sixteen threads on Valhall, but subdividing a warp may be useful for API requirements. In particular, derivatives may be calculated with quads (four threads).

The default value is subgroup16.

| Index | Value | Description |
|-------|------------|-----------------|
| 0 | subgroup2 | Two threads |
| 1 | subgroup4 | Four threads |
| 2 | subgroup8 | Eight threads |
| 3 | subgroup16 | Sixteen threads |

Lane operation

Acts as a modifier on the lane specifier for a CLPER instruction. The accumulate mode is required for efficient subgroup reductions.

The default value is none.

| Index | Value |
|-------|------------|
| 0 | none |
| 1 | xor |
| 2 | accumulate |
| 3 | shift |

Inactive result

Accesses to inactive lanes (due to divergence) in a subgroup is generally undefined in APIs. However, the results of permuting with an inactive lane with `CLPER.i32` are well-defined in Valhall: they return one of the following values, as specified in the `CLPER.i32` instructions. Sometimes certain values enable small optimizations.

The default value is zero.

| Index | Value |
|-------|--------|
| 0 | zero |
| 1 | umax |
| 2 | i1 |
| 3 | v2i1 |
| 4 | smin |
| 5 | smax |
| 6 | v2smin |
| 7 | v2smax |
| 8 | v4smin |
| 9 | v4smax |
| 10 | f1 |
| 11 | v2f1 |
| 12 | infn |
| 13 | inf |
| 14 | v2infn |
| 15 | v2inf |

Mux

Condition to use for a MUX instruction. `neg` checks the sign bit, `int_zero` compares to `0x00000000`, `fp_zero` compares to ± 0.0 as an IEEE 754 float, and `bit` checks each bit separately. The `bit` mode acts like an imaginary `CSEL.v32u1` instruction, and implements `bitselect()` in OpenCL.

The default value is `int_zero`.

| Index | Value | Description |
|-------|-----------------------|---------------------|
| 0 | <code>neg</code> | Negative |
| 1 | <code>int_zero</code> | Integer zero |
| 2 | <code>fp_zero</code> | Floating point zero |
| 3 | <code>bit</code> | Bitwise |

Sample mode

Varying interpolation mode, for choosing the correct sample to interpolate at, allowing the `sample` and `centroid` qualifiers to be implemented, as well as the `interpolateAt*` functions.

| Index | Value | Description |
|-------|-----------------------|-------------|
| 0 | <code>center</code> | Center |
| 1 | <code>centroid</code> | Centroid |
| 2 | <code>sample</code> | Sample |
| 3 | <code>explicit</code> | Explicit |

Update mode

The Valhall GPU maintains hidden state when interpolating varyings, to allow reusing sample location calculations. The update mode of a varying load controls this hidden state.

| Index | Value | Description |
|-------|-----------------|---------------------------------|
| 0 | store | Store interpolation position |
| 1 | retrieve | Retrieve interpolation position |
| 2 | <i>Reserved</i> | |
| 3 | clobber | Clobber saved position |

Instruction reference

The following section each known instruction in the Valhall ISA. It contains the instruction name, syntax, and bit pattern.

Related instructions are grouped together if they share an encoding and semantics. In these cases, multiple syntax specifiers are shown but only one representative bit pattern is given.

NOP

No operation.

Do nothing. Useful at the start of a block for waiting on slots required by the first actual instruction of the block, to reconcile dependencies after a branch. Also useful as the sole instruction of an empty shader.

NOP

Unit CVT.

Opcode 0x0

| Name | Bits |
|-----------------|-------|
| <i>Reserved</i> | 0:39 |
| Placeholder | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

BRANCHZ

Compare to zero and branch.

Branches to a specified relative offset if its source is nonzero (default) or if its source is zero (if `.eq` is set). The offset is 27-bits and sign-extended, giving an effective range of ± 26 -bits. The offset is specified in units of instructions, relative to the *next* instruction. Positive offsets may be interpreted as “number of instructions to skip”. Since Valhall instructions are 8 bytes, this operates as:

$$PC := \begin{cases} PC + 8 \cdot (\text{offset} + 1) & \text{if } \text{src} \stackrel{?}{=} 0 \\ PC + 8 & \text{otherwise} \end{cases}$$

Used with comparison instructions to implement control flow. Tie the source to a nonzero constant to implement a jump. May introduce divergence, so generally requires `.reconverge` flow control.

BRANCHZ{`.eq`} `src{.lane}`, `#offset`

Unit CVT.

Opcode 0x1F

| Name | Bits |
|---------------------------------------|-------|
| Source: Value to compare against zero | 0:7 |
| offset | 8:34 |
| <i>Reserved</i> | 35 |
| eq | 36 |
| lane 0 | 37:38 |
| <i>Reserved</i> | 39 |
| Placeholder | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

DISCARD.f32

Discard fragment.

Evaluates the given condition, and if it passes, discards the current fragment and terminates the thread. The destination should be set to R60. Only valid in a **fragment** shader.

DISCARD.f32{.cond} dest, src0{.abs}{.neg}{.swz}, src1{.abs}{.neg}{.swz}

Unit CVT.

Opcode 0x20

| Name | Bits |
|---|-------|
| Source: Left value to compare | 0:7 |
| Source: Right value to compare | 8:15 |
| <i>Reserved</i> | 16:25 |
| swizzle 1 | 26:27 |
| swizzle 0 | 28:29 |
| <i>Reserved</i> | 30:31 |
| condition | 32:34 |
| <i>Reserved</i> | 35 |
| neg 1 | 36 |
| abs 1 | 37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination: Updated coverage mask (set to R60) | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

BRANCHZI

Compare to zero and branch indirect.

Jump to an indirectly specified address. Used to jump to blend shaders at the end of a fragment shader.

BRANCHZI{.eq} src0, src1

Unit CVT.

Opcode 0x2F

| Name | Bits |
|---------------------------------------|-------|
| Source: Value to compare against zero | 0:7 |
| Source: Branch target | 8:15 |
| <i>Reserved</i> | 16:35 |
| eq | 36 |
| <i>Reserved</i> | 37:39 |
| Placeholder | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

BARRIER

Execution and memory barrier.

General-purpose barrier. Must use slot #7. Must be paired with a `.barrier` action on the instruction.

BARRIER{ .slot }

Opcode 0x45

| Name | Bits |
|-----------------|-------|
| <i>Reserved</i> | 0:29 |
| slot | 30:32 |
| <i>Reserved</i> | 33:39 |
| Placeholder | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

CSEL

Floating-point conditional select.

Evaluates the given condition and outputs either the true source or the false source.

CSEL.f32{.cond} dest, src0, src1, src2, src3

CSEL.v2f16{.cond} dest, src0, src1, src2, src3

Unit CVT.

| Mnemonic | Opcode |
|------------|--------|
| CSEL.f32 | 0x154 |
| CSEL.v2f16 | 0x155 |

| Name | Bits |
|--------------------------------|-------|
| Source: Left value to compare | 0:7 |
| Source: Right value to compare | 8:15 |
| Source: Return value if true | 16:23 |
| Source: Return value if false | 24:31 |
| condition | 32:34 |
| <i>Reserved</i> | 35:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

CSEL

Integer conditional select.

Evaluates the given condition and outputs either the true source or the false source.

Valhall lacks integer minimum/maximum instructions. CSEL instructions with tied operands form the canonical implementations of these instructions. Similarly, the integer sign function is canonically implemented with a pair of CSEL instructions.

CSEL.u32{.cond} dest, src0, src1, src2, src3
CSEL.v2u16{.cond} dest, src0, src1, src2, src3
CSEL.s32{.cond} dest, src0, src1, src2, src3
CSEL.v2s16{.cond} dest, src0, src1, src2, src3

Unit CVT.

| Mnemonic | Opcode |
|------------|--------|
| CSEL.u32 | 0x150 |
| CSEL.v2u16 | 0x151 |
| CSEL.s32 | 0x158 |
| CSEL.v2s16 | 0x159 |

| Name | Bits |
|--------------------------------|-------|
| Source: Left value to compare | 0:7 |
| Source: Right value to compare | 8:15 |
| Source: Return value if true | 16:23 |
| Source: Return value if false | 24:31 |
| condition | 32:34 |
| <i>Reserved</i> | 35:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LD_VAR_SPECIAL

Load special varying.

LD_VAR_SPECIAL{.vector_size}{.register_format}{.slot} @w, src, #index

Unit V.

Opcode 0x56

| Name | Bits |
|------------------------|-------|
| Source | 0:7 |
| <i>Reserved</i> | 8:11 |
| index | 12:15 |
| <i>Reserved</i> | 16:23 |
| register_format | 24:26 |
| <i>Reserved</i> | 27 |
| vector_size | 28:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| <i>Reserved</i> | 36:39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LD_VAR_IMM_F32

Load immediate varying.

Interpolates a given varying

LD_VAR_IMM_F32{.vector_size}{.half}{.smooth}{.slot} @w, src, #index

LD_VAR_IMM_F16{.vector_size}{.half}{.smooth}{.slot} @w, src, #index

Unit V.

| Mnemonic | Opcode |
|----------------|--------|
| LD_VAR_IMM_F32 | 0x5C |
| LD_VAR_IMM_F16 | 0x5D |

| Name | Bits |
|------------------------|-------|
| Source | 0:7 |
| <i>Reserved</i> | 8:15 |
| index | 16:23 |
| half | 24 |
| smooth | 25 |
| <i>Reserved</i> | 26:27 |
| vector_size | 28:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| <i>Reserved</i> | 36:39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LD_ATTR_IMM

Load immediate attribute.

Load vecsize components from the attribute descriptor at entry index of resource table table at index (vertex ID, instance ID), converting to the specified register format.

LD_ATTR_IMM{.vector_size}{.register_format}{.slot} @w, src0, src1, #index, #table

Unit LS.

Opcode 0x66

| Name | Bits |
|------------------------|-------|
| Source: Vertex ID | 0:7 |
| Source: Instance ID | 8:15 |
| table | 16:19 |
| index | 20:23 |
| register_format | 24:26 |
| <i>Reserved</i> | 27 |
| vector_size | 28:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| <i>Reserved</i> | 36:39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LEA_IMAGE_IMM

Load effective address of image texel.

Load the effective address of a texel from the image specified with the given immediate index. Returns three staging register: the low/high 32-bits of the address and the internal conversion descriptor. The format of the internal conversion descriptor is compatible with Bifrost but omits the register format, as this is specified with the ST_CVT instruction on Valhall.

Coordinates are specified as 16-bit integers, packed into 32-bit sources.

LEA_IMAGE_IMM{.vector_size}{.slot} @w, src0, src1, #unk, #index, #unk2

Unit LS.

Opcode 0x67

| Name | Bits |
|---------------------------------|-------|
| Source: X/Y coordinates (16:16) | 0:7 |
| Source: Z/W coordinates (16:16) | 8:15 |
| unk2 | 16:19 |
| index | 20:23 |
| <i>Reserved</i> | 24:27 |
| vector_size | 28:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| unk | 36:39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LD_BUFFER.i8

Global memory load.

Loads a buffer descriptor. If bits 25...31 of the mode descriptor are all-ones, load from the buffer descriptors in the table indexed by the bottom byte of the mode descriptor. If they are all zeroes, load the contents of the buffer in the first table indexed by the bottom byte of the mode descriptor.

LD_BUFFER.i8{.load_lane_8_bit}{.unsigned}{.slot} @w, src0, src1

Unit LS.

Opcode 0x6A

Secondary opcode 0x0

| Name | Bits |
|--|-------|
| Source: Address to load from after adding offset | 0:7 |
| Source: Mode descriptor | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| load_lane_8_bit | 36:38 |
| unsigned | 39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LD_BUFFER.i16

Global memory load.

Loads a buffer descriptor. If bits 25...31 of the mode descriptor are all-ones, load from the buffer descriptors in the table indexed by the bottom byte of the mode descriptor. If they are all zeroes, load the contents of the buffer in the first table indexed by the bottom byte of the mode descriptor.

LD_BUFFER.i16{.load_lane_16_bit}{.unsigned}{.slot} @w, src0, src1

Unit LS.

Opcode 0x6A

Secondary opcode 0x1

| Name | Bits |
|-------------------------|-------|
| Source: Byte offset | 0:7 |
| Source: Mode descriptor | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| load_lane_16_bit | 36:38 |
| unsigned | 39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LD_BUFFER.i24

Global memory load.

Loads a buffer descriptor. If bits 25...31 of the mode descriptor are all-ones, load from the buffer descriptors in the table indexed by the bottom byte of the mode descriptor. If they are all zeroes, load the contents of the buffer in the first table indexed by the bottom byte of the mode descriptor.

LD_BUFFER.i24{.load_lane_24_bit}{.unsigned}{.slot} @w, src0, src1

Unit LS.

Opcode 0x6A

Secondary opcode 0x2

| Name | Bits |
|-------------------------|-------|
| Source: Byte offset | 0:7 |
| Source: Mode descriptor | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| load_lane_24_bit | 36:38 |
| unsigned | 39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LD_BUFFER.i32

Global memory load.

Loads a buffer descriptor. If bits 25...31 of the mode descriptor are all-ones, load from the buffer descriptors in the table indexed by the bottom byte of the mode descriptor. If they are all zeroes, load the contents of the buffer in the first table indexed by the bottom byte of the mode descriptor.

LD_BUFFER.i32{.load_lane_32_bit}{.unsigned}{.slot} @w, src0, src1

Unit LS.

Opcode 0x6A

Secondary opcode 0x3

| Name | Bits |
|-------------------------|-------|
| Source: Byte offset | 0:7 |
| Source: Mode descriptor | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| load_lane_32_bit | 36:38 |
| unsigned | 39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LD_BUFFER.i48

Global memory load.

Loads a buffer descriptor. If bits 25...31 of the mode descriptor are all-ones, load from the buffer descriptors in the table indexed by the bottom byte of the mode descriptor. If they are all zeroes, load the contents of the buffer in the first table indexed by the bottom byte of the mode descriptor.

LD_BUFFER.i48{.load_lane_48_bit}{.unsigned}{.slot} @w, src0, src1

Unit LS.

Opcode 0x6A

Secondary opcode 0x4

| Name | Bits |
|-------------------------|-------|
| Source: Byte offset | 0:7 |
| Source: Mode descriptor | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| load_lane_48_bit | 36:38 |
| unsigned | 39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LD_BUFFER.i64

Global memory load.

Loads a buffer descriptor. If bits 25...31 of the mode descriptor are all-ones, load from the buffer descriptors in the table indexed by the bottom byte of the mode descriptor. If they are all zeroes, load the contents of the buffer in the first table indexed by the bottom byte of the mode descriptor.

LD_BUFFER.i64{.load_lane_64_bit}{.unsigned}{.slot} @w, src0, src1

Unit LS.

Opcode 0x6A

Secondary opcode 0x5

| Name | Bits |
|-------------------------|-------|
| Source: Byte offset | 0:7 |
| Source: Mode descriptor | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| load_lane_64_bit | 36:38 |
| unsigned | 39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LD_BUFFER.i96

Global memory load.

Loads a buffer descriptor. If bits 25...31 of the mode descriptor are all-ones, load from the buffer descriptors in the table indexed by the bottom byte of the mode descriptor. If they are all zeroes, load the contents of the buffer in the first table indexed by the bottom byte of the mode descriptor.

LD_BUFFER.i96{.load_lane_96_bit}{.unsigned}{.slot} @w, src0, src1

Unit LS.

Opcode 0x6A

Secondary opcode 0x6

| Name | Bits |
|-------------------------|-------|
| Source: Byte offset | 0:7 |
| Source: Mode descriptor | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| load_lane_96_bit | 36:38 |
| unsigned | 39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LD_BUFFER.i128

Global memory load.

Loads a buffer descriptor. If bits 25...31 of the mode descriptor are all-ones, load from the buffer descriptors in the table indexed by the bottom byte of the mode descriptor. If they are all zeroes, load the contents of the buffer in the first table indexed by the bottom byte of the mode descriptor.

LD_BUFFER.i128{.load_lane_128_bit}{.unsigned}{.slot} @w, src0, src1

Unit LS.

Opcode 0x6A

Secondary opcode 0x7

| Name | Bits |
|-------------------------|-------|
| Source: Byte offset | 0:7 |
| Source: Mode descriptor | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| load_lane_128_bit | 36:38 |
| unsigned | 39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LD_ATTR

Load indirect attribute.

The index must not diverge within a warp.

LD_ATTR{.vector_size}{.register_format}{.slot} @w, src0, src1, src2

Unit LS.

Opcode 0x76

| Name | Bits |
|------------------------|-------|
| Source: Vertex ID | 0:7 |
| Source: Instance ID | 8:15 |
| Source: Index | 16:23 |
| register_format | 24:26 |
| <i>Reserved</i> | 27 |
| vector_size | 28:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| <i>Reserved</i> | 36:39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LEA_VARY

Load varying effective address.

Loads the effective address of the position buffer (in a position shader) or the varying buffer (in a varying shader). That is, the base pointer plus the vertex's linear ID (the first source) times the buffer's per-vertex stride. LEA_VARY should be executed once in a position/varying shader, with the linear ID preloaded as r59. Each position/varying store can then be constructed as STORE with the base address sourced from the 64-bit destination of LEA_VARY and an appropriately computed offset. Varying stores bypass the usual conversion hardware for attributes; this diverges from earlier Mali hardware.

LEA_VARY{.slot} @w, src, #unk

Unit LS.

Opcode 0x5E

| Name | Bits |
|------------------------|-------|
| Source: Linear ID | 0:7 |
| unk | 8:11 |
| <i>Reserved</i> | 12:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| <i>Reserved</i> | 36:39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LOAD.i8

Global memory load.

Loads from main memory

LOAD.i8{.load_lane_8_bit}{.unsigned}{.slot} @w, src, #offset

Unit LS.

Opcode 0x60

Secondary opcode 0x0

| Name | Bits |
|--|-------|
| Source: Address to load from after adding offset | 0:7 |
| offset | 8:23 |
| <i>Reserved</i> | 24:26 |
| Secondary opcode | 27:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| load_lane_8_bit | 36:38 |
| unsigned | 39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LOAD.i16

Global memory load.

Loads from main memory

LOAD.i16{.load_lane_16_bit}{.unsigned}{.slot} @w, src, #offset

Unit LS.

Opcode 0x60

Secondary opcode 0x1

| Name | Bits |
|--|-------|
| Source: Address to load from after adding offset | 0:7 |
| offset | 8:23 |
| <i>Reserved</i> | 24:26 |
| Secondary opcode | 27:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| load_lane_16_bit | 36:38 |
| unsigned | 39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LOAD.i24

Global memory load.

Loads from main memory

LOAD.i24{.load_lane_24_bit}{.unsigned}{.slot} @w, src, #offset

Unit LS.

Opcode 0x60

Secondary opcode 0x2

| Name | Bits |
|--|-------|
| Source: Address to load from after adding offset | 0:7 |
| offset | 8:23 |
| <i>Reserved</i> | 24:26 |
| Secondary opcode | 27:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| load_lane_24_bit | 36:38 |
| unsigned | 39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LOAD.i32

Global memory load.

Loads from main memory

LOAD.i32{*.load_lane_32_bit*}{*.unsigned*}{*.slot*} @w, src, #offset

Unit LS.

Opcode 0x60

Secondary opcode 0x3

| Name | Bits |
|--|-------|
| Source: Address to load from after adding offset | 0:7 |
| offset | 8:23 |
| <i>Reserved</i> | 24:26 |
| Secondary opcode | 27:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| load_lane_32_bit | 36:38 |
| unsigned | 39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LOAD.i48

Global memory load.

Loads from main memory

LOAD.i48{.load_lane_48_bit}{.unsigned}{.slot} @w, src, #offset

Unit LS.

Opcode 0x60

Secondary opcode 0x4

| Name | Bits |
|--|-------|
| Source: Address to load from after adding offset | 0:7 |
| offset | 8:23 |
| <i>Reserved</i> | 24:26 |
| Secondary opcode | 27:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| load_lane_48_bit | 36:38 |
| unsigned | 39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LOAD.i64

Global memory load.

Loads from main memory

LOAD.i64{.load_lane_64_bit}{.unsigned}{.slot} @w, src, #offset

Unit LS.

Opcode 0x60

Secondary opcode 0x5

| Name | Bits |
|--|-------|
| Source: Address to load from after adding offset | 0:7 |
| offset | 8:23 |
| <i>Reserved</i> | 24:26 |
| Secondary opcode | 27:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| load_lane_64_bit | 36:38 |
| unsigned | 39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LOAD.i96

Global memory load.

Loads from main memory

LOAD.i96{.load_lane_96_bit}{.unsigned}{.slot} @w, src, #offset

Unit LS.

Opcode 0x60

Secondary opcode 0x6

| Name | Bits |
|--|-------|
| Source: Address to load from after adding offset | 0:7 |
| offset | 8:23 |
| <i>Reserved</i> | 24:26 |
| Secondary opcode | 27:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| load_lane_96_bit | 36:38 |
| unsigned | 39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LOAD.i128

Global memory load.

Loads from main memory

LOAD.i128{.load_lane_128_bit}{.unsigned}{.slot} @w, src, #offset

Unit LS.

Opcode 0x60

Secondary opcode 0x7

| Name | Bits |
|--|-------|
| Source: Address to load from after adding offset | 0:7 |
| offset | 8:23 |
| <i>Reserved</i> | 24:26 |
| Secondary opcode | 27:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| load_lane_128_bit | 36:38 |
| unsigned | 39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

STORE

Global memory store.

Stores to main memory

```
STORE.i8{.store_segment}{.slot} @r, src, #offset  
STORE.i16{.store_segment}{.slot} @r, src, #offset  
STORE.i24{.store_segment}{.slot} @r, src, #offset  
STORE.i32{.store_segment}{.slot} @r, src, #offset  
STORE.i48{.store_segment}{.slot} @r, src, #offset  
STORE.i64{.store_segment}{.slot} @r, src, #offset  
STORE.i96{.store_segment}{.slot} @r, src, #offset  
STORE.i128{.store_segment}{.slot} @r, src, #offset
```

Unit LS.

Primary opcode 0x61

| Mnemonic | Secondary opcode |
|------------|------------------|
| STORE.i8 | 0x0 |
| STORE.i16 | 0x1 |
| STORE.i24 | 0x2 |
| STORE.i32 | 0x3 |
| STORE.i48 | 0x4 |
| STORE.i64 | 0x5 |
| STORE.i96 | 0x6 |
| STORE.i128 | 0x7 |

| Name | Bits |
|---|-------|
| Source: Address to store to after adding offset | 0:7 |
| offset | 8:23 |
| store_segment | 24:25 |
| <i>Reserved</i> | 26 |
| Secondary opcode | 27:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| <i>Reserved</i> | 36:39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

ST_CVT

Store with conversion.

Store to memory with data conversion. The address to store to is given in the first source, which must be a 64-bit register (a pair of 32-bit registers). For backwards compatibility with Bifrost, there is a second source which should be the high 32-bits of the register. However, on Valhall the first source is 64-bit so the second source is unused. The third source is the conversion descriptor used for the store.

Used with LEA_IMAGE_IMM to implement image stores.

ST_CVT{*.vector_size*}{*.register_format*}{*.slot*} @r, src0, src1, src2, #unk

Unit LS.

Opcode 0x71

| Name | Bits |
|---|-------|
| Source: 64-bit address to store to (low) | 0:7 |
| Source: 64-bit address to store to (high) | 8:15 |
| Source: Internal conversion descriptor | 16:23 |
| register_format | 24:26 |
| <i>Reserved</i> | 27 |
| vector_size | 28:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| unk | 36:39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LD_TILE

Load from tilebuffer.

Loads a given render target, specified in the pixel indices descriptor, at a given location and sample, and convert to the format specified in the internal conversion descriptor. Used to implement EXT_framebuffer_fetch and internally in blend shaders.

LD_TILE{.slot} @w, src0, src1, src2

Opcode 0x78

| Name | Bits |
|----------------------------------|-------|
| Source: Pixel indices descriptor | 0:7 |
| Source: Coverage mask | 8:15 |
| Source: Conversion descriptor | 16:23 |
| <i>Reserved</i> | 24:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| <i>Reserved</i> | 36:39 |
| Staging | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

BLEND

Blend render target.

Blends a given render target. This loads the API-specified blend state for the render target from the first source. Blend descriptors are available as special immediates. It then reads the colour to be blended from the first staging register, with the specified vector size and register format as desired. The resulting coverage mask is stored to the second set of staging registers.

In the fixed-function path, BLEND sends the colour to the blender to be written to the tilebuffer. Then, if the instruction's flow control specifies termination, the fragment program is ended. If it does not specify termination, BLEND acts as a relative branch, branching with the offset specified as target. This allows the subsequent instructions to be skipped when fixed-function blending is used. Note this implicit branch can never introduce divergence, so `.reconverge` is not required.

In the blend shader path, BLEND ignores the specified flow control and does not branch to the specified offset. Instead, execution continues normally with the next instruction. The compiler should insert code for calling a blend shader after the BLEND instruction unless it is known that a blend shader will never be required.

The indirection is required to support both fixed-function and blend shaders efficiently and without shader variants.

BLEND{*.slot*}{*.vector_size*}{*.register_format*} @r, @w, src, #target

Opcode 0x7F

| Name | Bits |
|--------------------------|-------|
| Source: Blend descriptor | 0:7 |
| target | 8:15 |
| Staging 1 | 16:17 |
| <i>Reserved</i> | 18:23 |
| register_format | 24:26 |
| <i>Reserved</i> | 27 |
| vector_size | 28:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| <i>Reserved</i> | 36:39 |
| Staging 0 | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

ATEST

Alpha test.

Does alpha-to-coverage testing, updating the sample coverage mask. ATEST does not do an implicit discard. It should be executed before the first ZS_EMIT or BLEND instruction.

ATEST@w, src0, src1{.swz}, src2

Opcode 0x7D

| Name | Bits |
|---------------------------------------|-------|
| Source: Input coverage mask | 0:7 |
| Source: Alpha value (render target 0) | 8:15 |
| Source 2 | 16:23 |
| <i>Reserved</i> | 24:25 |
| swizzle 1 | 26:27 |
| <i>Reserved</i> | 28:32 |
| staging_register_count | 33:35 |
| <i>Reserved</i> | 36:39 |
| Staging: Updated coverage mask | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

ZS_EMIT

Depth/stencil write.

Programatically writes out depth, stencil, or both, depending on which modifiers are set. Used to implement `gl_FragDepth` and `gl_FragStencil`.

ZS_EMIT{.z}{.stencil} dest, src0, src1, src2

Opcode 0x7E

| Name | Bits |
|------------------------------------|-------|
| Source: Depth value | 0:7 |
| Source: Stencil value | 8:15 |
| Source: Input coverage mask | 16:23 |
| stencil | 24 |
| z | 25 |
| <i>Reserved</i> | 26:39 |
| Destination: Updated coverage mask | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

CONVERT

Data conversions.

Performs the given data conversion. Note that floating-point rounding is handled via the same hardware and therefore shares an encoding. Round mode is specified where it makes sense.

```
S16_TO_S32{.round} dest, src{.widen}  
S16_TO_F32{.round} dest, src{.widen}  
V2S16_TO_V2F16{.round} dest, src{.widen}  
S32_TO_F32{.round} dest, src{.widen}  
U16_TO_U32{.round} dest, src{.widen}  
U16_TO_F32{.round} dest, src{.widen}  
V2U16_TO_V2F16{.round} dest, src{.widen}  
U32_TO_F32{.round} dest, src{.widen}
```

Unit CVT.

Primary opcode 0x90

| Mnemonic | Secondary opcode |
|----------------|------------------|
| S16_TO_S32 | 0x4 |
| S16_TO_F32 | 0x5 |
| V2S16_TO_V2F16 | 0x7 |
| S32_TO_F32 | 0x9 |
| U16_TO_U32 | 0x14 |
| U16_TO_F32 | 0x15 |
| V2U16_TO_V2F16 | 0x17 |
| U32_TO_F32 | 0x19 |

| Name | Bits |
|--------------------------|-------|
| Source: Value to convert | 0:7 |
| <i>Reserved</i> | 8:15 |
| Secondary opcode | 16:20 |
| <i>Reserved</i> | 21:29 |
| round_mode | 30:31 |
| <i>Reserved</i> | 32:35 |
| widen 0 | 36:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

CONVERT

Float-to-int data conversions.

Performs the given data conversion.

F32_TO_S32{.round} dest, src{.abs}{.neg}

F32_TO_U32{.round} dest, src{.abs}{.neg}

Unit CVT.

Primary opcode 0x90

| Mnemonic | Secondary opcode |
|------------|------------------|
| F32_TO_S32 | 0xC |
| F32_TO_U32 | 0x1C |

| Name | Bits |
|--------------------------|-------|
| Source: Value to convert | 0:7 |
| <i>Reserved</i> | 8:15 |
| Secondary opcode | 16:20 |
| <i>Reserved</i> | 21:29 |
| round_mode | 30:31 |
| <i>Reserved</i> | 32:37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

CONVERT

Float-to-int data conversions.

Performs the given data conversion.

V2F16_TO_V2S16{.round} dest, src{.abs}{.neg}{.swz}

V2F16_TO_V2U16{.round} dest, src{.abs}{.neg}{.swz}

F16_TO_S32{.round} dest, src{.abs}{.neg}{.swz}

F16_TO_U32{.round} dest, src{.abs}{.neg}{.swz}

Unit CVT.

Primary opcode 0x90

| Mnemonic | Secondary opcode |
|----------------|------------------|
| V2F16_TO_V2S16 | 0xE |
| V2F16_TO_V2U16 | 0x1E |
| F16_TO_S32 | 0xA |
| F16_TO_U32 | 0x1A |

| Name | Bits |
|--------------------------|-------|
| Source: Value to convert | 0:7 |
| <i>Reserved</i> | 8:15 |
| Secondary opcode | 16:20 |
| <i>Reserved</i> | 21:27 |
| swizzle 0 | 28:29 |
| round_mode | 30:31 |
| <i>Reserved</i> | 32:37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

F16_TO_F32

16-bit float to 32-bit float conversion.

Converts up with the specified round mode.

F16_TO_F32{ .round } dest, src{ .abs }{ .neg }{ .lane }

Unit CVT.

Opcode 0x90

Secondary opcode 0xB

| Name | Bits |
|--------------------------|-------|
| Source: Value to convert | 0:7 |
| <i>Reserved</i> | 8:15 |
| Secondary opcode | 16:20 |
| <i>Reserved</i> | 21:27 |
| lane 0 | 28 |
| <i>Reserved</i> | 29 |
| round_mode | 30:31 |
| <i>Reserved</i> | 32:37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

CONVERT

8-bit data conversions.

Performs the given data conversion.

S8_TO_S32dest, src{.lane}
S8_TO_F32dest, src{.lane}
S8_TO_S16dest, src{.lane}
S8_TO_F16dest, src{.lane}
U8_TO_U32dest, src{.lane}
U8_TO_F32dest, src{.lane}
U8_TO_U16dest, src{.lane}
U8_TO_F16dest, src{.lane}

Unit CVT.

Primary opcode 0x90

| Mnemonic | Secondary opcode |
|-----------|------------------|
| S8_TO_S32 | 0x0 |
| S8_TO_F32 | 0x1 |
| S8_TO_S16 | 0x2 |
| S8_TO_F16 | 0x3 |
| U8_TO_U32 | 0x10 |
| U8_TO_F32 | 0x11 |
| U8_TO_U16 | 0x12 |
| U8_TO_F16 | 0x13 |

| Name | Bits |
|--------------------------|-------|
| Source: Value to convert | 0:7 |
| <i>Reserved</i> | 8:15 |
| Secondary opcode | 16:20 |
| <i>Reserved</i> | 21:27 |
| lane 0 | 28:29 |
| <i>Reserved</i> | 30:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

FROUND

Floating-point rounding.

Performs the given rounding, using the convert unit.

FROUND.f32{.round} dest, src{.abs}{.neg}{.swz}

FROUND.v2f16{.round} dest, src{.abs}{.neg}{.swz}

Unit CVT.

Primary opcode 0x90

| Mnemonic | Secondary opcode |
|--------------|------------------|
| FROUND.f32 | 0xD |
| FROUND.v2f16 | 0xF |

| Name | Bits |
|--------------------------|-------|
| Source: Value to convert | 0:7 |
| <i>Reserved</i> | 8:15 |
| Secondary opcode | 16:20 |
| <i>Reserved</i> | 21:27 |
| swizzle 0 | 28:29 |
| round_mode | 30:31 |
| <i>Reserved</i> | 32:37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

MOV.i32

Register move.

Canonical register-to-register move.

MOV.i32dest, src

Unit CVT.

Opcode 0x91

Secondary opcode 0x0

| Name | Bits |
|------------------|-------|
| Source | 0:7 |
| <i>Reserved</i> | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

CLZ.u32

Count leading zeroes.

Used as a primitive for various bitwise operations.

CLZ.u32dest, src

Unit CVT.

Opcode 0x91

Secondary opcode 0x4

| Name | Bits |
|------------------|-------|
| Source | 0:7 |
| <i>Reserved</i> | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

CLZ.v2u16

Count leading zeroes.

Used as a primitive for various bitwise operations.

CLZ.v2u16dest, src

Unit CVT.

Opcode 0x91

Secondary opcode 0x5

| Name | Bits |
|------------------|-------|
| Source | 0:7 |
| <i>Reserved</i> | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

CLZ.v4u8

Count leading zeroes.

Used as a primitive for various bitwise operations.

CLZ.v4u8dest, src

Unit CVT.

Opcode 0x91

Secondary opcode 0x6

| Name | Bits |
|------------------|-------|
| Source | 0:7 |
| <i>Reserved</i> | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

IABS.s32

Absolute value.

64-bit abs may be constructed in 4 instructions (5 clocks) by checking the sign with ICMP.s32.lt.m1 hi, 0 and negating based on the result with IADD.s64 and LSHIFT_XOR.i32 on each half.

IABS.s32dest, src{.widen}

Unit CVT.

Opcode 0x91

Secondary opcode 0x8

| Name | Bits |
|------------------|-------|
| Source | 0:7 |
| <i>Reserved</i> | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:35 |
| widen 0 | 36:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

IABS.v2s16

Absolute value.

IABS.v2s16dest, src{.widen}

Unit CVT.

Opcode 0x91

Secondary opcode 0x9

| Name | Bits |
|------------------|-------|
| Source | 0:7 |
| <i>Reserved</i> | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:35 |
| widen 0 | 36:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

IABS.v4s8

Absolute value.

IABS.v4s8dest, src

Unit CVT.

Opcode 0x91

Secondary opcode 0xA

| Name | Bits |
|------------------|-------|
| Source | 0:7 |
| <i>Reserved</i> | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

POPCOUNT.i32

Population count.

Only available as 32-bit. Smaller bitsizes require explicit conversions. 64-bit popcount may be constructed in 3 clocks by separate 32-bit popcounts of each half and a 32-bit add, which is guaranteed not to overflow.

POPCOUNT.i32 dest, src

Unit SFU.

Opcode 0x91

Secondary opcode 0xC

| Name | Bits |
|------------------|-------|
| Source | 0:7 |
| <i>Reserved</i> | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

BITREV.i32

Bitwise reverse.

Only available as 32-bit. Other bitsizes may be derived with swizzles.

BITREV.i32dest, src

Unit SFU.

Opcode 0x91

Secondary opcode 0xD

| Name | Bits |
|------------------|-------|
| Source | 0:7 |
| <i>Reserved</i> | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

NOT.i32

Bitwise complement.

For fully featured bitwise operation, see the shift opcodes.

NOT.i32dest, src

Unit SFU.

Opcode 0x91

Secondary opcode 0xE

| Name | Bits |
|------------------|-------|
| Source | 0:7 |
| <i>Reserved</i> | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

NOT.i64

Bitwise complement.

For fully featured bitwise operation, see the shift opcodes.

NOT.i64dest, src

Unit SFU.

Opcode 0x191

Secondary opcode 0xE

| Name | Bits |
|------------------|-------|
| Source | 0:7 |
| <i>Reserved</i> | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

WMASK

Warp mask.

Returns the mask of lanes ever active within the warp (subgroup), such that the source is nonzero. The number of work-items in a subgroup is given as the popcount of this value with a nonzero input.

An `all()` subgroup operation may be constructed as `WMASK` of the input compared for equality with `WMASK` of a nonzero value.

An `any()` subgroup operation may be constructed as `WMASK` of the input compared against zero.

WMASK{`.subgroup_size`} `dest`, `src`

Unit SFU.

Opcode 0x95

| Name | Bits |
|----------------------------|-------|
| Source | 0:7 |
| <i>Reserved</i> | 8:35 |
| <code>subgroup_size</code> | 36:37 |
| <i>Reserved</i> | 38:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

FREXP

Fraction/exponent extract.

Breaks up the floating-point input into its fractional (mantissa) and exponent parts. By default, this is compatible with the `frexp()` function in APIs. With the `log/sqrt` modifiers, the floating point format is adjusted to be compatible with Valhall's argument reduction for logarithm and square root computation respectively.

```
FREXPM.f32{.sqrt}{.log} dest, src{.swz}  
FREXPM.v2f16{.sqrt}{.log} dest, src{.swz}  
FREXPE.f32{.sqrt}{.log} dest, src{.swz}  
FREXPE.v2f16{.sqrt}{.log} dest, src{.swz}
```

Unit CVT.

Primary opcode 0x99

| Mnemonic | Secondary opcode |
|--------------|------------------|
| FREXPM.f32 | 0x0 |
| FREXPM.v2f16 | 0x1 |
| FREXPE.f32 | 0x2 |
| FREXPE.v2f16 | 0x3 |

| Name | Bits |
|------------------|-------|
| Source | 0:7 |
| <i>Reserved</i> | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:23 |
| <i>sqrt</i> | 24 |
| <i>log</i> | 25 |
| <i>Reserved</i> | 26:27 |
| <i>swizzle 0</i> | 28:29 |
| <i>Reserved</i> | 30:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

SFU

Special function unit.

Performs a given special function. The floating-point reciprocal (FRCP) and reciprocal square root (FRSQ) instructions may be freely used as-is. The logarithm instruction (FLOGD.f32) requires an argument reduction. See the transcendentals section for more information.

FRCP.f32dest, src{.abs}{.neg}{.swz}
FRCP.f16dest, src{.abs}{.neg}{.swz}
FRSQ.f32dest, src{.abs}{.neg}{.swz}
FRSQ.f16dest, src{.abs}{.neg}{.swz}
FLOGD.f32dest, src{.abs}{.neg}{.swz}

Unit SFU.

Primary opcode 0x9C

| Mnemonic | Secondary opcode |
|-----------|------------------|
| FRCP.f32 | 0x0 |
| FRCP.f16 | 0x1 |
| FRSQ.f32 | 0x2 |
| FRSQ.f16 | 0x3 |
| FLOGD.f32 | 0x8 |

| Name | Bits |
|------------------|-------|
| Source | 0:7 |
| <i>Reserved</i> | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:27 |
| swizzle 0 | 28:29 |
| <i>Reserved</i> | 30:37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

SFU

Special function unit.

Performs a given special function. The trigonometric tables (FSIN_TABLE.u6 and FCOS_TABLE.u6) are crude, requiring both an argument reduction and postprocessing.

FSIN_TABLE.u6dest, src

FCOS_TABLE.u6dest, src

Unit SFU.

Primary opcode 0x9C

| Mnemonic | Secondary opcode |
|---------------|------------------|
| FSIN_TABLE.u6 | 0x4 |
| FCOS_TABLE.u6 | 0x5 |

| Name | Bits |
|------------------|-------|
| Source | 0:7 |
| <i>Reserved</i> | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

FADD

Floating-point add.

$$A + B$$

FADD.f32{.clamp} dest, src0{.abs}{.neg}{.swz}, src1{.abs}{.neg}{.swz}

FADD.v2f16{.clamp} dest, src0{.abs}{.neg}{.swz}, src1{.abs}{.neg}{.swz}

Unit FMA.

| Mnemonic | Opcode |
|------------|--------|
| FADD.f32 | 0xA4 |
| FADD.v2f16 | 0xA5 |

| Name | Bits |
|------------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:25 |
| swizzle 1 | 26:27 |
| swizzle 0 | 28:29 |
| <i>Reserved</i> | 30:31 |
| clamp | 32:33 |
| <i>Reserved</i> | 34:35 |
| neg 1 | 36 |
| abs 1 | 37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

FMIN

Floating-point minimum.

$\min\{A, B\}$

FMIN.f32{.clamp} dest, src0{.abs}{.neg}{.swz}, src1{.abs}{.neg}{.swz}

FMIN.v2f16{.clamp} dest, src0{.abs}{.neg}{.swz}, src1{.abs}{.neg}{.swz}

Unit CVT.

| Mnemonic | Opcode |
|------------|--------|
| FMIN.f32 | 0xA4 |
| FMIN.v2f16 | 0xA5 |

| Name | Bits |
|------------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:25 |
| swizzle 1 | 26:27 |
| swizzle 0 | 28:29 |
| <i>Reserved</i> | 30:31 |
| clamp | 32:33 |
| <i>Reserved</i> | 34:35 |
| neg 1 | 36 |
| abs 1 | 37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

FMAX

Floating-point maximum.

$\max\{A, B\}$

FMAX.f32{.clamp} dest, src0{.abs}{.neg}{.swz}, src1{.abs}{.neg}{.swz}

FMAX.v2f16{.clamp} dest, src0{.abs}{.neg}{.swz}, src1{.abs}{.neg}{.swz}

Unit CVT.

| Mnemonic | Opcode |
|------------|--------|
| FMAX.f32 | 0xA4 |
| FMAX.v2f16 | 0xA5 |

| Name | Bits |
|------------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:25 |
| swizzle 1 | 26:27 |
| swizzle 0 | 28:29 |
| <i>Reserved</i> | 30:31 |
| clamp | 32:33 |
| <i>Reserved</i> | 34:35 |
| neg 1 | 36 |
| abs 1 | 37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

V2F32_TO_V2F16

Vectorized floating-point conversion.

Given a pair of 32-bit floats, output a pair of 16-bit floats packed into a 32-bit destination.

V2F32_TO_V2F16dest, src0, src1

Unit CVT.

| Mnemonic | Opcode |
|----------------|--------|
| V2F32_TO_V2F16 | 0xA5 |

| Name | Bits |
|------------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

FRSCALE

Floating-point rescaling.

Computes $A \cdot 2^B$ by adding B to the exponent of A. Used to calculate various special functions, particularly base-2 exponents. Special case handling differs from an actual floating-point multiply, so this should not be used outside fixed instruction sequences.

FRSCALE.f32{.clamp} dest, src0{.abs}{.neg}{.swz}, src1{.abs}{.neg}{.swz}

FRSCALE.v2f16{.clamp} dest, src0{.abs}{.neg}{.swz}, src1{.abs}{.neg}{.swz}

Unit FMA.

| Mnemonic | Opcode |
|---------------|--------|
| FRSCALE.f32 | 0xA4 |
| FRSCALE.v2f16 | 0xA5 |

| Name | Bits |
|------------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:25 |
| swizzle 1 | 26:27 |
| swizzle 0 | 28:29 |
| <i>Reserved</i> | 30:31 |
| clamp | 32:33 |
| <i>Reserved</i> | 34:35 |
| neg 1 | 36 |
| abs 1 | 37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

FEXP.f32

Floating-point exponent.

Calculates the base-2 exponent of an argument specified as a 8:24 fixed-point. The original argument is passed as well for correct handling of special cases.

FEXP.f32{.clamp} dest, src0, src1{.abs}{.neg}

Unit SFU.

Opcode 0xA4

Secondary opcode 0x8

| Name | Bits |
|-----------------------------------|-------|
| Source: Input as 8:24 fixed-point | 0:7 |
| Source: Input as 32-bit float | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:31 |
| clamp | 32:33 |
| <i>Reserved</i> | 34:35 |
| neg 1 | 36 |
| abs 1 | 37 |
| <i>Reserved</i> | 38:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

FADD_LSCALE.f32

Floating-point add with logarithm scale.

Performs a floating-point addition specialized for logarithm computation.

FADD_LSCALE.f32{.clamp} dest, src0{.abs}{.neg}, src1{.abs}{.neg}

Unit FMA.

Opcode 0xA4

Secondary opcode 0x9

| Name | Bits |
|------------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:31 |
| clamp | 32:33 |
| <i>Reserved</i> | 34:35 |
| neg 1 | 36 |
| abs 1 | 37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

IADD

Integer addition.

$A + B$ with optional saturation.

As Valhall lacks swizzle instructions, `IADD.v2i16` with zero is the canonical lowering for swizzles.

```
IADD.u32{.saturate} dest, src0{.widen}, src1{.widen}  
IADD.v2u16{.saturate} dest, src0{.widen}, src1{.widen}  
IADD.v4u8{.saturate} dest, src0{.widen}, src1{.widen}  
IADD.s32{.saturate} dest, src0{.widen}, src1{.widen}  
IADD.v2s16{.saturate} dest, src0{.widen}, src1{.widen}  
IADD.v4s8{.saturate} dest, src0{.widen}, src1{.widen}  
IADD.u64{.saturate} dest, src0{.widen}, src1{.widen}  
IADD.s64{.saturate} dest, src0{.widen}, src1{.widen}
```

Unit CVT.

| Mnemonic | Opcode |
|------------|--------|
| IADD.u32 | 0xA0 |
| IADD.v2u16 | 0xA1 |
| IADD.v4u8 | 0xA2 |
| IADD.s32 | 0xA8 |
| IADD.v2s16 | 0xA9 |
| IADD.v4s8 | 0x1A2 |
| IADD.u64 | 0x1A3 |
| IADD.s64 | 0x1AB |

| Name | Bits |
|------------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:25 |
| widen 1 | 26:29 |
| saturate | 30 |
| <i>Reserved</i> | 31:35 |
| widen 0 | 36:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

MKVEC.v2i16

Make 16-bit vector.

Calculates $A|(B \ll 16)$. Used to implement (ushort2)(A, B)

MKVEC.v2i16dest, src0{.widen}, src1{.widen}

Unit CVT.

Opcode 0xA1

Secondary opcode 0x5

| Name | Bits |
|------------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:25 |
| widen 1 | 26:29 |
| <i>Reserved</i> | 30:35 |
| widen 0 | 36:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

ISUB

Integer subtract.

$A - B$ with optional saturation

ISUB.u32{.saturate} dest, src0{.widen}, src1{.widen}
ISUB.v2u16{.saturate} dest, src0{.widen}, src1{.widen}
ISUB.v4u8{.saturate} dest, src0{.widen}, src1{.widen}
ISUB.s32{.saturate} dest, src0{.widen}, src1{.widen}
ISUB.v2s16{.saturate} dest, src0{.widen}, src1{.widen}
ISUB.v4s8{.saturate} dest, src0{.widen}, src1{.widen}
ISUB.u64{.saturate} dest, src0{.widen}, src1{.widen}
ISUB.s64{.saturate} dest, src0{.widen}, src1{.widen}

Unit CVT.

| Mnemonic | Opcode |
|------------|--------|
| ISUB.u32 | 0xA0 |
| ISUB.v2u16 | 0xA1 |
| ISUB.v4u8 | 0xA2 |
| ISUB.s32 | 0xA8 |
| ISUB.v2s16 | 0xA9 |
| ISUB.v4s8 | 0x1A2 |
| ISUB.u64 | 0x1A3 |
| ISUB.s64 | 0x1AB |

| Name | Bits |
|------------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:25 |
| widen 1 | 26:29 |
| saturate | 30 |
| <i>Reserved</i> | 31:35 |
| widen 0 | 36:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

SHADDX

Shift, extend, and 64-bit add.

Sign or zero extend B to 64-bits, left-shift by `shift`, and add the 64-bit value A. These instructions accelerate address arithmetic, but may be used in full generality for 64-bit integer arithmetic.

SHADDX.u64`dest, src0, src1{.widen}, #shift`

SHADDX.s64`dest, src0, src1{.widen}, #shift`

Unit CVT.

| Mnemonic | Opcode |
|------------|--------|
| SHADDX.u64 | 0x1A3 |
| SHADDX.s64 | 0x1AB |

| Name | Bits |
|------------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Secondary opcode | 16:19 |
| shift | 20:22 |
| <i>Reserved</i> | 23:25 |
| widen 1 | 26:29 |
| <i>Reserved</i> | 30:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

IMUL

Integer multiply.

$A \cdot B$ with optional saturation. Note the multipliers can only handle up to 32-bit by 32-bit multiplies. The 64-bit “multiply” acts like IMUL.u32 but additionally writes the high half of the product to the high half of the 64-bit destination. Along with IADD.u32 and IADD.u64, this allows the construction of a 64-bit multiply in 5 instructions (6 clocks).

```
IMUL.i32{.saturate} dest, src0{.widen}, src1{.widen}  
IMUL.v2i16{.saturate} dest, src0{.widen}, src1{.widen}  
IMUL.v4i8{.saturate} dest, src0{.widen}, src1{.widen}  
IMUL.s32{.saturate} dest, src0{.widen}, src1{.widen}  
IMUL.v2s16{.saturate} dest, src0{.widen}, src1{.widen}  
IMUL.v4s8{.saturate} dest, src0{.widen}, src1{.widen}  
IMULD.u64{.saturate} dest, src0{.widen}, src1{.widen}
```

Unit SFU.

| Mnemonic | Opcode |
|------------|--------|
| IMUL.i32 | 0xA0 |
| IMUL.v2i16 | 0xA1 |
| IMUL.v4i8 | 0xA2 |
| IMUL.s32 | 0xA8 |
| IMUL.v2s16 | 0xA9 |
| IMUL.v4s8 | 0x1A2 |
| IMULD.u64 | 0x1A3 |

| Name | Bits |
|------------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:25 |
| widen 1 | 26:29 |
| saturate | 30 |
| <i>Reserved</i> | 31:35 |
| widen 0 | 36:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

HADD

Integer half-add.

$(A + B) \gg 1$ without intermediate overflow, corresponding to `hadd()` in OpenCL. With the `.rhadd` modifier set, it instead calculates $(A + B + 1) \gg 1$ corresponding to `rhadd()` in OpenCL.

```
HADD.u32{.rhadd} dest, src0{.widen}, src1{.widen}  
HADD.v2u16{.rhadd} dest, src0{.widen}, src1{.widen}  
HADD.v4u8{.rhadd} dest, src0{.widen}, src1{.widen}  
HADD.s32{.rhadd} dest, src0{.widen}, src1{.widen}  
HADD.v2s16{.rhadd} dest, src0{.widen}, src1{.widen}  
HADD.v4s8{.rhadd} dest, src0{.widen}, src1{.widen}
```

Unit CVT.

| Mnemonic | Opcode |
|------------|--------|
| HADD.u32 | 0xA0 |
| HADD.v2u16 | 0xA1 |
| HADD.v4u8 | 0xA2 |
| HADD.s32 | 0xA8 |
| HADD.v2s16 | 0xA9 |
| HADD.v4s8 | 0x1A2 |

| Name | Bits |
|------------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:25 |
| widen 1 | 26:29 |
| rhadd | 30 |
| <i>Reserved</i> | 31:35 |
| widen 0 | 36:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

CLPER

Cross-lane permute.

Selects the value of A in the subgroup lane given by B. This implements subgroup broadcasts. It may be used as a primitive for screen space derivatives in fragment shaders.

```
CLPER.i32{.subgroup_size}{.lane_op}{.inactive_res} dest, src0, src1{.widen}  
CLPER.v2u16{.subgroup_size}{.lane_op}{.inactive_res} dest, src0, src1{.widen}  
CLPER.v4u8{.subgroup_size}{.lane_op}{.inactive_res} dest, src0, src1{.widen}  
CLPER.s32{.subgroup_size}{.lane_op}{.inactive_res} dest, src0, src1{.widen}  
CLPER.v2s16{.subgroup_size}{.lane_op}{.inactive_res} dest, src0, src1{.widen}  
CLPER.v4s8{.subgroup_size}{.lane_op}{.inactive_res} dest, src0, src1{.widen}  
CLPER.u64{.subgroup_size}{.lane_op}{.inactive_res} dest, src0, src1{.widen}  
CLPER.s64{.subgroup_size}{.lane_op}{.inactive_res} dest, src0, src1{.widen}
```

Unit SFU.

| Mnemonic | Opcode |
|-------------|--------|
| CLPER.i32 | 0xA0 |
| CLPER.v2u16 | 0xA1 |
| CLPER.v4u8 | 0xA2 |
| CLPER.s32 | 0xA8 |
| CLPER.v2s16 | 0xA9 |
| CLPER.v4s8 | 0x1A2 |
| CLPER.u64 | 0x1A3 |
| CLPER.s64 | 0x1AB |

| Name | Bits |
|------------------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Secondary opcode | 16:19 |
| <i>Reserved</i> | 20:21 |
| <i>inactive_result</i> | 22:25 |
| <i>widen 1</i> | 26:29 |
| <i>Reserved</i> | 30:31 |
| <i>lane_operation</i> | 32:33 |
| <i>Reserved</i> | 34:35 |
| <i>subgroup_size</i> | 36:37 |
| <i>Reserved</i> | 38:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

FMA

Fused floating-point multiply add.

$$A \cdot B + C$$

FMA.f32{.clamp} dest, src0{.abs}{.neg}{.swz}, src1{.abs}{.neg}{.swz}, src2{.abs}{.neg}{.swz}

FMA.v2f16{.clamp} dest, src0{.abs}{.neg}{.swz}, src1{.abs}{.neg}{.swz}, src2{.abs}{.neg}{.swz}

Unit FMA.

| Mnemonic | Opcode |
|-----------|--------|
| FMA.f32 | 0xB2 |
| FMA.v2f16 | 0xB3 |

| Name | Bits |
|-----------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Source: C | 16:23 |
| swizzle 2 | 24:25 |
| swizzle 1 | 26:27 |
| swizzle 0 | 28:29 |
| <i>Reserved</i> | 30:31 |
| clamp | 32:33 |
| neg 2 | 34 |
| abs 2 | 35 |
| neg 1 | 36 |
| abs 1 | 37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LSHIFT_AND

Left shift and bitwise AND.

Left shifts its first source by a specified amount and bitwise ANDs it with the second source, optionally inverting the second source or the result.

```
LSHIFT_AND.i32{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}  
LSHIFT_AND.v2i16{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}  
LSHIFT_AND.v4i8{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}  
LSHIFT_AND.i64{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}
```

Unit SFU.

| Mnemonic | Opcode |
|------------------|--------|
| LSHIFT_AND.i32 | 0xB4 |
| LSHIFT_AND.v2i16 | 0xB5 |
| LSHIFT_AND.v4i8 | 0xB6 |
| LSHIFT_AND.i64 | 0x1B7 |

| Name | Bits |
|------------------|-------|
| Source: A | 0:7 |
| Source: shift | 8:15 |
| Source: B | 16:23 |
| Secondary opcode | 24:25 |
| widen 1 | 26:29 |
| not_result | 30 |
| <i>Reserved</i> | 31 |
| Secondary opcode | 32 |
| <i>Reserved</i> | 33:34 |
| not 2 | 35 |
| widen 0 | 36:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

RSHIFT_AND

Right shift and bitwise AND.

Right shifts its first source by a specified amount and bitwise ANDs it with the second source, optionally inverting the second source or the result. If signed is set, the hardware performs an arithmetic right shift; otherwise, it performs an unsigned right shift.

RSHIFT_AND.i32{.signed}{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}
RSHIFT_AND.v2i16{.signed}{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}
RSHIFT_AND.v4i8{.signed}{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}
RSHIFT_AND.i64{.signed}{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}

Unit SFU.

| Mnemonic | Opcode |
|------------------|--------|
| RSHIFT_AND.i32 | 0xB4 |
| RSHIFT_AND.v2i16 | 0xB5 |
| RSHIFT_AND.v4i8 | 0xB6 |
| RSHIFT_AND.i64 | 0x1B7 |

| Name | Bits |
|------------------|-------|
| Source: A | 0:7 |
| Source: shift | 8:15 |
| Source: B | 16:23 |
| Secondary opcode | 24:25 |
| widen 1 | 26:29 |
| not_result | 30 |
| <i>Reserved</i> | 31 |
| Secondary opcode | 32 |
| <i>Reserved</i> | 33 |
| signed | 34 |
| not 2 | 35 |
| widen 0 | 36:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LSHIFT_OR

Left shift and bitwise OR.

Left shifts its first source by a specified amount and bitwise ORs it with the second source, optionally inverting the second source or the result.

```
LSHIFT_OR.i32{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}  
LSHIFT_OR.v2i16{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}  
LSHIFT_OR.v4i8{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}  
LSHIFT_OR.i64{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}
```

Unit SFU.

| Mnemonic | Opcode |
|-----------------|--------|
| LSHIFT_OR.i32 | 0xB4 |
| LSHIFT_OR.v2i16 | 0xB5 |
| LSHIFT_OR.v4i8 | 0xB6 |
| LSHIFT_OR.i64 | 0x1B7 |

| Name | Bits |
|------------------|-------|
| Source: A | 0:7 |
| Source: shift | 8:15 |
| Source: B | 16:23 |
| Secondary opcode | 24:25 |
| widen 1 | 26:29 |
| not_result | 30 |
| <i>Reserved</i> | 31 |
| Secondary opcode | 32 |
| <i>Reserved</i> | 33:34 |
| not 2 | 35 |
| widen 0 | 36:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

RSHIFT_OR

Right shift and bitwise OR.

Right shifts its first source by a specified amount and bitwise ORs it with the second source, optionally inverting the second source or the result. If signed is set, the hardware performs an arithmetic right shift; otherwise, it performs an unsigned right shift.

RSHIFT_OR.i32{.signed}{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}
RSHIFT_OR.v2i16{.signed}{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}
RSHIFT_OR.v4i8{.signed}{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}
RSHIFT_OR.i64{.signed}{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}

Unit SFU.

| Mnemonic | Opcode |
|-----------------|--------|
| RSHIFT_OR.i32 | 0xB4 |
| RSHIFT_OR.v2i16 | 0xB5 |
| RSHIFT_OR.v4i8 | 0xB6 |
| RSHIFT_OR.i64 | 0xB7 |

| Name | Bits |
|------------------|-------|
| Source: A | 0:7 |
| Source: shift | 8:15 |
| Source: B | 16:23 |
| Secondary opcode | 24:25 |
| widen 1 | 26:29 |
| not_result | 30 |
| <i>Reserved</i> | 31 |
| Secondary opcode | 32 |
| <i>Reserved</i> | 33 |
| signed | 34 |
| not 2 | 35 |
| widen 0 | 36:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

LSHIFT_XOR

Left shift and bitwise XOR.

Left shifts its first source by a specified amount and bitwise XORs it with the second source, optionally inverting the second source or the result.

```
LSHIFT_XOR.i32{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}  
LSHIFT_XOR.v2i16{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}  
LSHIFT_XOR.v4i8{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}  
LSHIFT_XOR.i64{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}
```

Unit SFU.

| Mnemonic | Opcode |
|------------------|--------|
| LSHIFT_XOR.i32 | 0xB4 |
| LSHIFT_XOR.v2i16 | 0xB5 |
| LSHIFT_XOR.v4i8 | 0xB6 |
| LSHIFT_XOR.i64 | 0x1B7 |

| Name | Bits |
|------------------|-------|
| Source: A | 0:7 |
| Source: shift | 8:15 |
| Source: B | 16:23 |
| Secondary opcode | 24:25 |
| widen 1 | 26:29 |
| not_result | 30 |
| <i>Reserved</i> | 31 |
| Secondary opcode | 32 |
| <i>Reserved</i> | 33:34 |
| not 2 | 35 |
| widen 0 | 36:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

RSHIFT_XOR

Right shift and bitwise XOR.

Right shifts its first source by a specified amount and bitwise XORs it with the second source, optionally inverting the second source or the result. If signed is set, the hardware performs an arithmetic right shift; otherwise, it performs an unsigned right shift.

RSHIFT_XOR.i32{.signed}{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}
RSHIFT_XOR.v2i16{.signed}{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}
RSHIFT_XOR.v4i8{.signed}{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}
RSHIFT_XOR.i64{.signed}{.not_result} dest, src0{.widen}, src1{.lanes}, src2{.not}

Unit SFU.

| Mnemonic | Opcode |
|------------------|--------|
| RSHIFT_XOR.i32 | 0xB4 |
| RSHIFT_XOR.v2i16 | 0xB5 |
| RSHIFT_XOR.v4i8 | 0xB6 |
| RSHIFT_XOR.i64 | 0x1B7 |

| Name | Bits |
|------------------|-------|
| Source: A | 0:7 |
| Source: shift | 8:15 |
| Source: B | 16:23 |
| Secondary opcode | 24:25 |
| widen 1 | 26:29 |
| not_result | 30 |
| <i>Reserved</i> | 31 |
| Secondary opcode | 32 |
| <i>Reserved</i> | 33 |
| signed | 34 |
| not 2 | 35 |
| widen 0 | 36:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

MUX.i32

Mux.

Mux between A and B based on the provided mask. The condition specified as the mux modifier is evaluated on the mask. If true, A is chosen, else B is chosen. The bit modifier acts bitwise, equivalent to `bitselect()` in OpenCL, so `MUX.i32.bit A, B, mask` calculates $(A \ \& \ \text{mask}) \mid (B \ \& \ \sim\text{mask})$.

MUX.i32{.mux} dest, src0, src1, src2

Unit SFU.

Opcode 0xB8

| Name | Bits |
|-----------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Source: Mask | 16:23 |
| <i>Reserved</i> | 24:31 |
| <i>mux</i> | 32:33 |
| <i>Reserved</i> | 34:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

CUBE_SSEL

Cube S-coordinate select.

During a cube map transform, select the S coordinate given a selected face.

CUBE_SSELdest, src0{.abs}{.neg}, src1{.abs}{.neg}, src2

Unit SFU.

Opcode 0xBC

Secondary opcode 0x0

| Name | Bits |
|---|-------|
| Source: Z coordinate as 32-bit floating point | 0:7 |
| Source: X coordinate as 32-bit floating point | 8:15 |
| Source: Cube face index | 16:23 |
| Secondary opcode | 24:27 |
| <i>Reserved</i> | 28:35 |
| neg 1 | 36 |
| abs 1 | 37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

CUBE_TSEL

Cube T-coordinate select.

During a cube map transform, select the T coordinate given a selected face.

CUBE_TSELdest, src0{.abs}{.neg}, src1{.abs}{.neg}, src2

Unit SFU.

Opcode 0xBC

Secondary opcode 0x1

| Name | Bits |
|---|-------|
| Source: Y coordinate as 32-bit floating point | 0:7 |
| Source: Z coordinate as 32-bit floating point | 8:15 |
| Source: Cube face index | 16:23 |
| Secondary opcode | 24:27 |
| <i>Reserved</i> | 28:35 |
| neg 1 | 36 |
| abs 1 | 37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

MKVEC.v4i8

Make 8-bit vector.

Calculates $A|(B \ll 8)|(CD \ll 16)$ for 8-bit A and B and 16-bit CD.

To implement (uchar4) (A, B, C, D) in full generality, use the sequence MKVEC.v4i8 CD, C, D, #0; MKVEC.v4i8 out, A, B, CD

MKVEC.v4i8 also allows zero extending arbitrary 8-bit lanes. For example, to extend r0.b3 to r1, use MKVEC.v4i8 r1, r0.b3, 0x0.b0, 0x0.

MKVEC.v4i8dest, src0{.lane}, src1{.lane}, src2

Unit CVT.

Opcode 0xBD

| Name | Bits |
|-----------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Source: CD | 16:23 |
| <i>Reserved</i> | 24:35 |
| lane 1 | 36:37 |
| lane 0 | 38:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

CUBEFACE1

Cube map transform step 1.

Select the maximum absolute value of its arguments.

CUBEFACE1dest, src0{.abs}{.neg}, src1{.abs}{.neg}, src2{.abs}{.neg}

Unit SFU.

Opcode 0xC0

| Name | Bits |
|---|-------|
| Source: X coordinate as 32-bit floating point | 0:7 |
| Source: Y coordinate as 32-bit floating point | 8:15 |
| Source: Z coordinate as 32-bit floating point | 16:23 |
| <i>Reserved</i> | 24:33 |
| neg 2 | 34 |
| abs 2 | 35 |
| neg 1 | 36 |
| abs 1 | 37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

CUBEFACE2

Cube map transform step 2.

Select the cube face index corresponding to the arguments.

CUBEFACE2dest, src0{.abs}{.neg}, src1{.abs}{.neg}, src2{.abs}{.neg}

Unit SFU.

Opcode 0xC1

| Name | Bits |
|---|-------|
| Source: X coordinate as 32-bit floating point | 0:7 |
| Source: Y coordinate as 32-bit floating point | 8:15 |
| Source: Z coordinate as 32-bit floating point | 16:23 |
| <i>Reserved</i> | 24:33 |
| neg 2 | 34 |
| abs 2 | 35 |
| neg 1 | 36 |
| abs 1 | 37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

IDP

8-bit dot product.

8-bit integer dot product between 4 channel vectors, intended for machine learning. Available in both unsigned and signed variants, controlling sign-extension/zero-extension behaviour to the final 32-bit destination. Saturation is available. Corresponds to the `cl_arm_integer_dot_product_*` family of OpenCL extensions. Not for actual use, just for completeness. Instead, use your platform's neural accelerator.

For $A, B \in \{0, \dots, 255\}^4$ and $\text{Accumulator} \in \mathbb{Z}$, calculates $(A \cdot B) + \text{Accumulator}$ and optionally saturates.

IDP.v4s8{.saturate} dest, src0, src1, src2

IDP.v4u8{.saturate} dest, src0, src1, src2

Unit FMA.

Primary opcode 0xC2

| Mnemonic | Secondary opcode |
|----------|------------------|
| IDP.v4s8 | 0x0 |
| IDP.v4u8 | 0x1 |

| Name | Bits |
|---------------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Source: Accumulator | 16:23 |
| Secondary opcode | 24:27 |
| <i>Reserved</i> | 28:29 |
| saturate | 30 |
| <i>Reserved</i> | 31:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

ICMP

Unsigned integer compare.

Evaluates the given condition, do a logical and/or with the condition in the result source, and return in the given result type (integer one, integer minus one, or floating-point one). The third source is useful for chaining together conditions without intermediate bitwise arithmetic; when this is not desired, tie it to zero and use the OR combine mode (do not set the .and modifier).

The sequence modifier .seq is used to construct 64-bit compares in 2 ICMP.u32 instructions, in conjunction with the u1 result type on the low half, the m1 result type on the high half, and the result of the low half comparison passed as the third source. For comparisons other than 64-bit, do not set the .seq modifier and do not use the u1 result type.

ICMP.u32{.cond}{.result_type}{.and}{.seq} dest, src0{.widen}, src1{.widen}, src2
ICMP.v2u16{.cond}{.result_type}{.and}{.seq} dest, src0{.widen}, src1{.widen}, src2
ICMP.v4u8{.cond}{.result_type}{.and}{.seq} dest, src0{.widen}, src1{.widen}, src2

Unit CVT.

| Mnemonic | Opcode |
|------------|--------|
| ICMP.u32 | 0xF0 |
| ICMP.v2u16 | 0xF1 |
| ICMP.v4u8 | 0xF2 |

| Name | Bits |
|-----------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Source: C | 16:23 |
| and | 24 |
| seq | 25 |
| widen 1 | 26:29 |
| result_type | 30:31 |
| condition | 32:34 |
| <i>Reserved</i> | 35 |
| widen 0 | 36:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

FCMP

Floating-point compare.

Evaluates the given condition, do a logical and/or with the condition in the result source, and return in the given result type (integer one, integer minus one, or floating-point one). The third source is useful for chaining together conditions without intermediate bitwise arithmetic; when this is not desired, tie it to zero and use the OR combine mode (do not set the .and modifier).

FCMP.f32{.cond}{.result_type}{.and} dest, src0{.abs}{.neg}{.swz}, src1{.abs}{.neg}{.swz}, src2

FCMP.v2f16{.cond}{.result_type}{.and} dest, src0{.abs}{.neg}{.swz}, src1{.abs}{.neg}{.swz}, src2

Unit CVT.

| Mnemonic | Opcode |
|------------|--------|
| FCMP.f32 | 0xF4 |
| FCMP.v2f16 | 0xF5 |

| Name | Bits |
|-----------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Source: C | 16:23 |
| and | 24 |
| <i>Reserved</i> | 25 |
| swizzle 1 | 26:27 |
| swizzle 0 | 28:29 |
| result_type | 30:31 |
| condition | 32:34 |
| <i>Reserved</i> | 35 |
| neg 1 | 36 |
| abs 1 | 37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

ICMP

Signed integer compare.

Evaluates the given condition, do a logical and/or with the condition in the result source, and return in the given result type (integer one, integer minus one, or floating-point one). The third source is useful for chaining together conditions without intermediate bitwise arithmetic; when this is not desired, tie it to zero and use the OR combine mode (do not set the .and modifier).

The sequence modifier .seq is used to construct signed 64-bit compares in 1 ICMP.u32 and 1 ICMP.s32 instruction, in conjunction with the u1 result type on the low half, the m1 result type on the high half, and the result of the low half comparison passed as the third source. For comparisons other than 64-bit, do not set the .seq modifier and do not use the u1 result type.

ICMP.s32{.cond}{.result_type}{.and}{.seq} dest, src0{.widen}, src1{.widen}, src2
ICMP.v2s16{.cond}{.result_type}{.and}{.seq} dest, src0{.widen}, src1{.widen}, src2
ICMP.v4s8{.cond}{.result_type}{.and}{.seq} dest, src0{.widen}, src1{.widen}, src2

Unit CVT.

| Mnemonic | Opcode |
|------------|--------|
| ICMP.s32 | 0xF8 |
| ICMP.v2s16 | 0xF9 |
| ICMP.v4s8 | 0xFA |

| Name | Bits |
|-----------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Source: C | 16:23 |
| and | 24 |
| seq | 25 |
| widen 1 | 26:29 |
| result_type | 30:31 |
| condition | 32:34 |
| <i>Reserved</i> | 35 |
| widen 0 | 36:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

IADD_IMM.i32

Integer addition with immediate.

Adds an arbitrary 32-bit immediate embedded within the instruction stream. If no modifiers are required, this is preferred to IADD.i32 with a constant accessed as a uniform. However, if the constant is available inline, IADD.f32 is preferred.

IADD_IMM.i32 with the source tied to zero is the canonical immediate move.

IADD_IMM.i32dest, src, #constant

Unit CVT.

Opcode 0x110

| Name | Bits |
|-------------|-------|
| Source: A | 0:7 |
| constant | 8:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

IADD_IMM.v2i16

Integer addition with immediate.

Adds an arbitrary pair of 16-bit immediates embedded within the instruction stream. If no modifiers are required, this is preferred to `IADD.v2i16` with a constant accessed as a uniform. However, if the constant is available inline, `IADD.v2i16` is preferred. Adding only a single 16-bit constant requires replication of the constant.

IADD_IMM.v2i16dest, src, #constant

Unit CVT.

Opcode 0x111

| Name | Bits |
|-------------|-------|
| Source: A | 0:7 |
| constant | 8:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

IADD_IMM.v4i8

Integer addition with immediate.

Adds an arbitrary quad of 8-bit immediates embedded within the instruction stream. If no modifiers are required, this is preferred to `IADD.v4i8` with a constant accessed as a uniform. However, if the constant is available inline, `IADD.v4i8` is preferred. Adding only a single 8-bit constant requires replication of the constant.

IADD_IMM.v4i8dest, src, #constant

Unit CVT.

Opcode 0x112

| Name | Bits |
|-------------|-------|
| Source: A | 0:7 |
| constant | 8:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

FADD_IMM.f32

Floating-point addition with immediate.

Adds an arbitrary 32-bit immediate embedded within the instruction stream. If no modifiers are required, this is preferred to FADD.f32 with a constant accessed as a uniform. However, if the constant is available inline, FADD.f32 is preferred.

FADD_IMM.f32dest, src, #constant

Unit FMA.

Opcode 0x114

| Name | Bits |
|-------------|-------|
| Source: A | 0:7 |
| constant | 8:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

FADD_IMM.v2f16

Floating-point addition with immediate.

Adds an arbitrary pair of 16-bit immediates embedded within the instruction stream. If no modifiers are required, this is preferred to FADD.v2f16 with a constant accessed as a uniform. However, if the constant is available inline, FADD.v2f16 is preferred. Adding only a single 16-bit constant requires replication of the constant.

FADD_IMM.v2f16dest, src, #constant

Unit FMA.

Opcode 0x115

| Name | Bits |
|-------------|-------|
| Source: A | 0:7 |
| constant | 8:39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

TEX_FETCH

Texel fetch.

Unfiltered textured instruction.

TEX_FETCH{.explicit_offset}{.dim}{.skip}{.slot} @r, @w, src

Unit T.

Opcode 0x125

| Name | Bits |
|----------------------------|-------|
| Source: Image to read from | 0:7 |
| <i>Reserved</i> | 8:10 |
| explicit_offset | 11 |
| <i>Reserved</i> | 12:15 |
| Staging 1 | 16:23 |
| <i>Reserved</i> | 24:27 |
| dimension | 28:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| <i>Reserved</i> | 36:38 |
| skip | 39 |
| Staging 0 | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

TEX

Texture load.

Ordinary texturing instruction using a sampler.

TEX{.explicit_offset}{.shadow}{.lod}{.dim}{.skip}{.slot} @r, @w, src

Unit T.

Opcode 0x128

| Name | Bits |
|----------------------------|-------|
| Source: Image to read from | 0:7 |
| <i>Reserved</i> | 8:10 |
| explicit_offset | 11 |
| shadow | 12 |
| lod_mode | 13:15 |
| Staging 1 | 16:23 |
| <i>Reserved</i> | 24:27 |
| dimension | 28:29 |
| slot | 30:32 |
| staging_register_count | 33:35 |
| <i>Reserved</i> | 36:38 |
| skip | 39 |
| Staging 0 | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

FMA_RSCALE.f32

Fused floating-point multiply add with exponent bias.

First calculates $A \cdot B + C$ and then biases the exponent by D. Used in special transcendental function sequences. It should not be used for general code as its special case handling differs from two back-to-back FMA.f32 operations. Equivalent to FMA.f32 back-to-back with RSCALE.f32

FMA_RSCALE.f32{.clamp} dest, src0{.abs}{.neg}, src1{.abs}{.neg}, src2{.abs}{.neg}, src3

Unit FMA.

Opcode 0x160

| Name | Bits |
|-------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Source: C | 16:23 |
| Source: D | 24:31 |
| clamp | 32:33 |
| neg 2 | 34 |
| abs 2 | 35 |
| neg 1 | 36 |
| abs 1 | 37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

FMA_RSCALE_N.f32

Fused floating-point multiply add with exponent bias and zero override.

First calculates $A \cdot B + C$ and then biases the exponent by D. If $A = 0$ or $B = 0$, the multiply $A \cdot B$ is treated as zero even if an ordinary multiply would return NaN. Used in special transcendental function sequences. It should not be used for general code as its special case handling differs from two back-to-back FMA.f32 operations. Equivalent to FMA.f32 back-to-back with RSCALE.f32

FMA_RSCALE_N.f32{.clamp} dest, src0{.abs}{.neg}, src1{.abs}{.neg}, src2{.abs}{.neg}, src3

Unit FMA.

Opcode 0x161

| Name | Bits |
|-------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Source: C | 16:23 |
| Source: D | 24:31 |
| clamp | 32:33 |
| neg 2 | 34 |
| abs 2 | 35 |
| neg 1 | 36 |
| abs 1 | 37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

FMA_RSCALE_LEFT.f32

Fused floating-point multiply add with exponent bias and asymmetric zero handling.

First calculates $A \cdot B + C$ and then biases the exponent by D. If $A = 0$ or $B = 0$, the multiply is treated as A even if an ordinary multiply would return NaN. Used in special transcendental function sequences. It should not be used for general code as its special case handling differs from two back-to-back FMA.f32 operations. Equivalent to FMA.f32 back-to-back with RSCALE.f32

FMA_RSCALE_LEFT.f32{.clamp} dest, src0{.abs}{.neg}, src1{.abs}{.neg}, src2{.abs}{.neg}, src3

Unit FMA.

Opcode 0x162

| Name | Bits |
|-------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Source: C | 16:23 |
| Source: D | 24:31 |
| clamp | 32:33 |
| neg 2 | 34 |
| abs 2 | 35 |
| neg 1 | 36 |
| abs 1 | 37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

FMA_RSCALE_SCALE16.f32

Fused floating-point multiply add with 16-bit exponent bias.

First calculates $A \cdot B + C$ and then biases the exponent by D, interpreted as a 16-bit value. Used in special transcendental function sequences. It should not be used for general code as its special case handling differs from two back-to-back FMA.f32 operations. Equivalent to FMA.f32 back-to-back with RSCALE.f32

FMA_RSCALE_SCALE16.f32{.clamp} dest, src0{.abs}{.neg}, src1{.abs}{.neg}, src2{.abs}{.neg}, src3

Unit FMA.

Opcode 0x163

| Name | Bits |
|-------------|-------|
| Source: A | 0:7 |
| Source: B | 8:15 |
| Source: C | 16:23 |
| Source: D | 24:31 |
| clamp | 32:33 |
| neg 2 | 34 |
| abs 2 | 35 |
| neg 1 | 36 |
| abs 1 | 37 |
| neg 0 | 38 |
| abs 0 | 39 |
| Destination | 40:47 |
| Opcode | 48:56 |
| Metadata | 57:63 |

Appendix A - Mali Gxx reference

Mali Gxx covers both Bifrost (major 6, 7) and Valhall (major 9). Architecture major 8 did not ship in any products. It was likely an intermediate step in between Bifrost and Valhall, perhaps implementing the Bifrost instruction set with an early version of Valhall's data structures.

| Name | Codename | Major | Minor |
|------------|----------|-------|-------|
| Mali G71 | Mimir | 6 | 0 |
| Mali G72 | Heimdall | 6 | 1 |
| Mali G51 | Sigurd | 7 | 0 |
| Mali G76 | Norr | 7 | 1 |
| Mali G52 | Gondul | 7 | 2 |
| Mali G31 | Dvalin | 7 | 3 |
| Mali G77 | Trym | 9 | 0 |
| Mali G57 | Natt-A | 9 | 1 |
| Mali G78 | Borr | 9 | 2 |
| Mali G57 | Natt-B | 9 | 3 |
| Mali G68 | Ottr | 9 | 4 |
| Mali G78AE | Borr-AE | 9 | 5 |

Appendix B - Performance characteristics

Mali G78 attains the following normalized peak performance, as reported by the Mali Offline Compiler:

- 64 FMA instructions per cycle
- 64 CVT instructions per cycle
- 16 SFU instructions per cycle
- 8 x 32-bit varying channels interpolated per cycle
- 4 texture instructions per cycle
- 1 load/store operation per cycle

64-bit instructions operate at half-speed compared to 32-bit instructions. For example, at most 32 SHADDX.u64 instructions may be executed per normalized cycle, since SHADDX.u64 runs on the CVT unit.

Depending on the cache hit rate, instructions accessing memory can be significantly slower than reported here. Depending on the exact texture instruction issued and the corresponding sampler state, texture instructions can be additionally slower.