bifrost: Fix 0xe
Alyssa Rosenzweig authored 4 years ago

c8a581a2

Bifrost.adoc 78.55 KiB

Introduction

This page aims to describes, in a hopefully understandable way, what I've discovered so far about the instruction set for ARM's Bifrost architecture. In addition to the technical details, I'll try and give a higher-level view of what's going on; that is, *why* I think ARM did something, and the design tradeoffs involved, rather than just the instruction encoding and what it does. Some of this can be ascertained from patents, articles, etc., but some of it will necessarily be guesswork.

Public information from ARM

ARM has publicly explained various parts of the ISA in various places. Obviously, any public information from ARM is incredibly useful in the reverse-engineering process. The two main sources of information are articles/presentations and patents. Articles and presentations are aimed at a general audience, and so they may not go into the detail required to actually understand the ISA, but they are good at giving a broad overview of ARM's intentions in designing the microarchitecture. Patents tend to be more specific, since ARM needs to describe the technical details more precisely for their patent to be accepted, but they're also full of legalese that can be confusing and tedious at times.

Reading patents

Note: the usual IANAL disclaimer applies here. The information here is only meant to be used for reverse engineering, I can't vouch for its accuracy, don't use it for anything legally significant without consulting a lawyer first, etc. You've been warned!

Usually, for our purpose, it's better to skip the claims section and focus on the "embodiments" section. The claims section is important legally, since it governs when ARM can sue someone for patent infringement (in some sense, it's the "meat" of the patent), but we aren't concerned about that. The embodiments section, on the other hand, is supposed to provide the patent examiner and a potential judge/jury context about the invention it's disclosing by describing how ARM is actually using it in its own products. An "embodiment" here is just a way that the invention could actually be used in an actual, shipping product. Technically, how the patent is actually used in practice is irrelevant from a legal point of view, so you'll see language like "in one possible embodiment, ...", but it's usually obvious which "possible embodiment" is the one that ARM is actually using — the patent will go into detail on only one of the possible embodiments, or explicitly talk about a "preferred embodiment" which is usually the one they're actually using.

List o' Links

- Anandtech article, gives a general overview
- UK Patent Application GB2540970A, on the Bifrost clause architecture
- US Patent Application US 2016-0364209 A1, describing a way to implement more efficient argument reduction for logarithms using a lookup table. It seems that the Itanium math libraries used the exact same technique, described in a paper here, specifically in the "novel reduction" section. Yes, this means the patent application is probably bogus. No, I'm not a lawyer.
- WO2017125700A1. This seems to describe encoding two different commutative operations with the same opcode, differentiating between them by whether the larger source comes first (what about when the sources are the same?). This is used for encoding the abs modifier for fp16 instructions.
- US20140354644A1. This describes the scoreboard dependency method described in "Dependency tracking."
- US20170286107A1. Describes how divergence/reconvergence is handled, before describing a modification to handle single-thread locking.

Overview

Bifrost is a clause-oriented architecture. Instructions are grouped into *clauses* that consist of instructions that can be executed back-to-back without any scheduling decisions. A clause consists of a *clause header* with information required for scheduling followed by a series of instructions and constants (immediates) referenced by the instructions. Instead of choosing instructions to run, the scheduler chooses clauses to run. As we'll see, instruction decoding is also per-clause instead of per-instruction. Instructions and immediates are packed into a clause and unpacked at run-time. That way, instructions can be packed to save space, and the unpacking cost is amortized over the entire clause.

Internally, each instruction consists of a register read/write stage, an FMA pipeline stage, and an ADD pipeline stage. The register stage is decoupled in the ISA from the functional units, which simplifies the decode hardware. The register stage bits describe which registers to read/write with various ports, while the FMA and ADD stages have only 3 bits for each source which describe which port to read for each source. In addition, the result of both stages from the previous cycle appears as another "port", allowing instructions to bypass the register file to save power and

the ADD unit. For example, an unfused multiply-add is done by doing a multiply in the FMA unit, using the 0 port for the sum, and then an addition in the ADD unit with the same port (here representing the result of the multiply) in the ADD unit.
Note that in addition to the execution unit, which we've been describing, there are a few other units:

- Varying interpolation unit
- Attribute fetching unit
- Load/store unit
- Texture unit

The execution unit interacts with these fixed-function blocks through special, variable-latency instructions in the FMA and ADD units. They bypass the usual, fixed-latency mechanism for reading/writing registers, and as such instructions in the same clause can't assume that registers have been read/written after the instruction is done. Instead, any dependent instructions must be put into a separate clause with the appropriate dependencies in the clause header set.

Clauses

Conceptually, each clause consists of a clause header, followed by one or more 78-bit instruction words and then zero or more 60-bit constants. (Constants are actually 64 bits, but they're loaded the same port as uniform registers and share the same field in the instruction word, which includes 7 bits to choose which uniform register to load, some of which would be unused for constants, so ARM decided to be clever and stick the bottom 4 bits in each instruction where the constant is loaded, so the actual constants in the instruction stream are only 60 bits). But the instruction fetching hardware only works in 128-bit quadwords, so each clause has to be a multiple of 128 bits. To make the representation of the clauses as compact as possible which still making the decoding circuitry relatively simple, the instructions are packed so that two 128-bit quadwords can store 3 78-bit instructions, or 3 128-bit quadwords can store 4 instructions and a 60-bit constant. There were some bits left over, which seem to have been used to obviate the need to keep track of state between each word, simplifying the decoder and making it possible to decode the quadwords in parallel. Thus, the quadwords can be (almost) arbitrarily reordered while still retaining the meaning of the clause. (It's unknown whether this works in practice, but theoretically it could be done.) Each format fully describes which instruction(s) in the decoded clause the bits in the quadword represent, and whether one of those instructions is the last instruction.

Quadword formats

The bottom 8 bits of each 128-bit quadword are a "tag" that's read by the decoding circuitry. They describe how to interpret the rest of the word, as well as possibly containing some bits from some of the instructions to decode if there were more bits to spare. Occasionally, some of these formats will have space dedicated to holding the first constant in the clause. Each possible tag value, along with the algorithm for determining which tag value/format to use for each quadword, are described below:

Format #1

127-83 (45 bits)	8-82 (75 bits)	3-7 (5 bits)	0-2 (3 bits)
Clause Header	`I0` (0-74)	Tag	`I0` (75-77)

Tag	Description
`00101xxx`	Instructions follow
`01001xxx`	Final quadword
`00001xxx`	Constants follow

Format #2.1

125-127 (3 bits)	83-124 (42 bits)	8-82 (75 bits)	0-7 (8 bits)
`I1` (75-77)	*Unused*	`I1` (0-74)	Tag

Tag	Description
`01000011`	End of clause

`00000011`	Constants follow

Format #2.2

83-127 (45 bits)	8-82 (75 bits)	3-7 (5 bits)	0-2 (3 bits)
`I2` (0-44)	`I1` (0-74)	Tag	`I1` (75-77)

Tag	Description
`00100xxx`	The only tag

Format #3.1

125-127 (3 bits)	113-124 (12 bits)	83-112 (30 bits)	68-82 (15 bits)	8-67 (60 bits)	0-7 (8 bits)
`I2` (75-77)	*Unused*	`I2` (45-74)	*Unused*	Constant (0-59)	Tag

Tag	Description
`00000100`	Constants follow
`01000100`	End of clause

Format #3.2

113-127 (15 bits)	83-112 (30 bits)	8-82 (75 bits)	6-7 (2 bits)	3-5 (3 bits)	0-2 (3 bits)
Constant (0-14)	`I2` (45-74)	`I3` (0-74)	Tag	`I3` (75-77)	`I2` (75-77)

Tag	Description
`10xxxxxx`	The only tag

Format #3.3

125-127 (3 bits)	122-124 (3 bits)	113-121 (9 bits)	83-112 (30 bits)	8-82 (75 bits)	0-7 (8 bits)
`I2` (75-77)	`I3` (75-77)	*Unused*	`I2` (45-74)	`I3` (0-74)	Tag

Tag	Description
`01000101`	End of clause
`00000101`	Constants follow
`00000001`	Instructions follow

Format #4.1

83-127 (45 bits)	8-82 (75 bits)	3-7 (5 bits)	0-2 (3 bits)
Constant (15-59)	`I4` (0-74)	Tag	`I4` (75-77)

`01010xxx`	Final quadword
`00010xxx`	Constants follow

Format #4.2

83-127 (45 bits)	8-82 (75 bits)	3-7 (5 bits)	0-2 (3 bits)
`I5` (0-44)	`I4` (0-74)	Tag	`I4` (75-77)

Tag	Description
`01100xxx`	The only tag

Format #5.1

125-127 (3 bits)	113-124 (12 bits)	83-112 (30 bits)	68-82 (15 bits)	8-67 (60 bits)	0-7 (8 bits)
`I5` (75-77)	*Unused*	`I5` (45-74)	*Unused*	Constant (0-59)	Tag

Tag	Description
`01000110`	Final quadword
`00000110`	Constants follow

Format #5.2

125-127 (3 bits)	122-124 (3 bits)	113-121 (9 bits)	83-112 (30 bits)	8-82 (75 bits)	0-7 (8 bits)
`I5` (75-77)	`I6` (75-77)	*Unused*	`I5` (45-74)	`I6` (0-74)	Tag

Tag	Description
`01000111`	Final quadword
`00000111`	Constants follow

Format #5.3

113-127 (15 bits)	83-112 (30 bits)	8-82 (75 bits)	6-7 (2 bits)	3-5 (3 bits)	0-2 (3 bits)
Constant (0-14)	`I5` (45-74)	`I6` (0-74)	Tag	`I6` (75-77)	`I5` (75-77)

Tag	Description
`11xxxxxx`	The only tag

Format #6

83-127 (45 bits)	8-82 (75 bits)	3-7 (5 bits)	0-2 (3 bits)
Constant (15-59)	`I7` (0-74)	Tag	`I7` (75-77)

`01011xxx`	Final quadword
`00011xxx`	Constants follow

Constant format

68-127 (60 bits)	8-67 (60 bits)	4-7 (4 bits)	0-3 (4 bits)
Constant (0-59)	Constant (0-59)	Tag	`pos`

Tag	Description
`0011xxxx`	Constants follow
`0111xxxx`	End of clause

The `pos` bits describe where in the instruction stream the constants are. In particular, it encodes the total number of instructions in the clause and the number of constants before the current two. Since the packing algorithm below can only produce some of these combinations, not every possible pair of instructions and constants is representable. The table below lists the ones that are currently known.

`pos`	Instructions	Constants
0	1	0
1	2	0
2	4	0
3	3	1
4	5	1
5	4	2
6	7	0
7	6	1
8	5	3
9	8	1
a	7	2
b	6	3
c	8	3
d	7	4
e	6	5

- 7-instruction clauses may load at most 6 constants.
- 8-instruction clauses may load at most 5 constants.

However, there is only one remaining value for `pos` that haven't been observed (f), so there isn't much space left for more constants.

There is a weird restriction on the contents of the constants: when interpreted as an unsigned integer, the most-significant 4 bits of the first constant (bits 64-67 of the quadword) must be less than or equal to the most-significant 4 bits of the second constant (bits 124-127 of the quadword). If this doesn't hold, the constants need to be swapped. Experiments have shown that violating this leads to some of the bits being replaced with random values.

Algorithm for packing clauses

This section describes how the blob compiler seems to use these formats to pack as many instructions and constants into as few words as possible. There may be other equivalent ways to do it, but given how complicated all the different formats are, and that the hardware decoder and encoding algorithm were most likely developed in tandem, it's probably best to stick with what the blob does.

One characteristic of this algorithm that must be kept in mind is that whenever a format in this algorithm requires a constant, we must use a dummy constant if there are no constants left to pack. For example, say that we have a clause with 5 instructions and no constants. The fourth quadword is supposed to contain only instruction 4, which is the final instruction, but there is no format for that. Instead, we add a constant split across the third and fourth quadwords, since there is a format with a final instruction 4 and part of a constant. In general, we need to do this whenever a constant could be inserted "for free", i.e. whenever there are at least three instructions. From a design point of view, this reduces the number of possible formats, which reduces the complexity of the decoder and means less bits are needed to describe the format.

The algorithm is described below:

- Begin by packing instructions into quadwords. When there are no more instructions left to pack, skip to the next step:
  - For the first quadword:
    - Pack the first instruction using format #1
  - For the second quadword:
    - If there's only the second instruction left:
      - Pack the second instruction in this quadword using format #2.1
    - Otherwise:
      - Pack the second and part of the third instruction in this quadword using format #2.2
  - For the third quadword:
    - If there's only the third instruction left:
      - Pack the rest of the third instruction and the first constant in this quadword using format #3.1
    - Otherwise, if there's only the third, fourth, and fifth instruction left to pack:
      - Pack the rest of the third instruction, all of the fourth instruction, and part of the first constant into this quadword using format #3.2
    - Otherwise:
      - Pack the rest of the third instruction and all of the fourth instruction into this quadword using format #3.3
  - For the fourth quadword:
    - If there's only the fifth instruction and the first constant left to pack:
      - Pack all of the fifth instruction and the rest of the first constant into this quadword using format #4.1
    - Otherwise:
      - Pack all of the fifth instruction and part of the sixth instruction into this quadword using format #4.2
  - For the fifth quadword:
    - If there's only the sixth instruction left to pack:
      - Pack the rest of the sixth instruction and all of the first constant into this quadword using format #5.1
    - Otherwise, if there's only the sixth and seventh instruction left to pack:
      - Pack the rest of the sixth instruction and all of the seventh instruction into this quadword using format #5.2

        - Pack the rest of the sixth instruction, all of the seventh instruction, and part of the first constant into this quadword using format #5.3
    - For the sixth quadword:
        - Pack the eigth instruction and the rest of the first constant into this quadword using format #6
- Repeat these steps as long as there are constants still left to pack:
    - Add another quadword
    - Pack the next two constants into that quadword using the constant format

Clause Header

The clause header mainly contains information about "variable-latency" instructions like SSBO loads/stores/atomics, texture sampling, etc. that use separate functional units. There can be at most one variable-latency instruction per clause. It also indicates when execution should stop, and has some information about branching. The format of the header is as follows:

Field	Bits
Unknown	11
Back to back	1
Not end of shader	1
Unknown	2
Elide writes	1
Branch conditional	1
Data Register Write Barrier	1
Data Register	6
Scoreboard dependencies	8
Scoreboard entry	3
Instruction type	4
Unknown	1
Next clause instruction type	4
Unknown	1

The "Back to Back" field is set to true if the execution mask of the next clause is the same as the mask of the current clause and there is no branch instruction in this clause.

The "Elide Writes" field is set to true for fragment shaders, and is intended to implement section 7.1.5 of the GLSL ES spec: "Stores to image and buffer variables performed by helper invocations have no effect on the underlying image or buffer memory.". Helper invocations are threads (invocations) corresponding to pixels in a quad that aren’t actually part of the triangle, but are included to make derivatives work correctly. They’re usually turned on, but they need to be masked off for GLSL-level stores.

The "Branch Conditional" bit is always set if the "Back to Back" field is set to one. Otherwise, it’s either set to one to indicate that this clause is either a conditional branch or a fallthrough branch, or zero to indicate that this clause is unconditionally executed.

The "Data Register Write Barrier" is set when the next clause writes to the data register of some previous clause.

Bifrost is a SIMT architecture, which means that the hardware executes 4 (or later 8) threads at the same time. Divergence and reconvergence is mostly handled automatically by the hardware, but it needs some software help via the "Back to Back" bit.

In addition to the scalar IP, which holds the currently executing clause, and the exec mask which says which threads are currently active, there's the vector IP. For active threads the vector IP is conceptually the same as the scalar IP, however for inactive threads it holds **the address at which the thread re-converges**. Furthermore, bifrost maintains the invariant that **the scalar IP is always the minumum IP of all threads**. That is, the non-active threads all have a greater IP than the active threads. Thus, conceptually after each clause is finished the hardware does the following:

1. Figure out the new IP for each active thread, and update the vector IP.
2. Calculate the minimum IP, set that as the scalar IP, and update the exec mask to contain only threads with the minimum IP.

However, most of the time this isn't necessary. Usually control flows directly from clause to clause, without the exec mask changing. In this case we don't actually have to update the vector IP or exec mask at all. We can just update the scalar IP, only resolving the vector IP when there is non-trivial control flow. This is what the "Back-to-Back" bit is for. It is set when the scalar IP is always normally incremented and the exec mask is unchanged after the clause finishes.

Note that "Back-to-Back" can't always be set, even if the clause doesn't include a branch. For example, imagine this pseudo-assembly for a simple if construct:

```
clause_0: nbb {
...
BRANCH cond, #clause_3
}

clause_1: {
...
}

clause_2: nbb {
...
}

clause_3: {
...
}
```

In addition to `clause_0` which ends in a branch, `clause_2` is also not back-to-back because threads which took the branch may re-converge at `clause_3` following it. These are "fallthrough branches", and must have the "branch conditional" bit set in addition to normal conditional branches.

Register field

A lot of variable-latency instructions have to interact with the register file in ways that would be awkward to express in the usual manner, i.e. with the per-instruction register field. For example, the STORE instruction has to read up to 4 32-bit registers, which the usual pathways for reading a register can't handle — they're designed for reading up to three 32-bit or 64-bit registers each cycle, and it also needs to load a 64-bit address from registers. The LOAD instruction can't write to the register until the operation has finished, possibly well after the instruction executes. For cases like these, there's a "register" field in the clause header that lets the variable-latency instruction read/write one, or a sequence of, registers, using a completely different mechanism. Since there can only be one variable-latency instruction per clause, this field isn't ambiguous about which instruction it applies to. When the variable-latency instruction is supposed to read or write more than one register (e.g. `LOAD.v4i32`), they are read or written in sequence starting with the register specified. There are no restrictions on which register to specifiy, except that the reads/writes cannot go out of bounds of the register file (so a `LOAD.v4i32` with a data register of `R63` would result in a fault, since it would try to write `R63` through `R66`). The blob compiler will only use aligned register pairs and quads, but this doesn't seem to be necessary.

Dependency tracking

No instructions depend upon a high-latency instruction in the same clause and so all the intra-clause scheduling can be done by the compiler. On the other hand, instructions in one clause might depend on a variable-latency instruction in another clause, and the latency obviously can't be known beforehand, so some kind of inter-clause dependency tracking mechanism must exist. Bifrost uses a six-entry scoreboard, with the scoreboard entries and dependencies manually described by the compiler. Each clause has a scoreboard entry, which corresponds to a given bit in the scoreboard. When the clause is dispatched, the bit is set, and when the variable-latency instruction in the clause completes, the bit is cleared. Any clauses afterwards can set that clause's scoreboard entry in its "scoreboard dependencies" bitfield to halt execution until the variable-latency instruction completes.

As a concrete example, consider this program:

```
layout(std430, binding = 0) buffer myBuffer {
    int inVal1, inVal2, outVal;
}
```

```
}
```

It might get translated into something like this, in assembly-like pseudocode (assuming for a second that the loads don’t get combined):

```
{
LOAD.i32 R0, ptr + 0x0
}
{
LOAD.i32 R1, ptr + 0x4
}
{
ADD.i32 R0, R0, R1
STORE.i32 R0, ptr + 0x8
}
```

The third clause must depend on the first two, although the first two are independent and can be executed in any order. The dependency bits to express this would be:

Clause	Scoreboard entry	Scoreboard dependencies
1	0	00000000
2	1	00000000
3	2	00000011

Since the first two clauses have no dependencies, they will be started in-order, one immediately after the other. They will queue up two requests to the load/store unit, with scoreboard tags of 0 and 1, and set bits 0 and 1 of the scoreboard. The first load will clear bit 0 of the scoreboard (based on the tag that was sent with the load) when it is finished, and the second load will clear bit 1. The third clause has bits 0 and 1 set in the dependencies, so it will will wait for bits 0 and 1 to clear before executing. Therefore, it won’t run until both of the loads have been completed.

The final wrinkle in all of this is that the scoreboard dependencies encoded in the clause are actually the dependencies before the *next* clause is ready to execute. So in the above example, the actual encoding for the clauses would look like:

Clause	Scoreboard entry	Scoreboard dependencies
1	0	00000000
2	1	00000011
3	2	00000000

The first clause in a program implicitly has no dependencies. This scheme makes it possible to determine whether the next clause can be run before actually fetching it, presumably simplifying the hardware scheduler a little.

In addition to the normal 6 scoreboard entries available for clauses to wait on other clauses, there are two more entries reserved for tile operations. Bit 6 is cleared when depth and stencil values have been written for earlier fragments, so that the depth and stencil tests can safely proceed. The ATEST instruction (see patent) must wait on this bit. Bit 7 is cleared when blending has been completed for earlier fragments and the results written to the tile buffer, so that blending is possible. The BLEND instruction must wait on this bit. The blob makes also BLEND wait on bit 6, but I don’t think that’s necessary since it also waits on ATEST which waits on bit 6. These scoreboard entries provide similar functionality to the branch-on-no-dependency instruction on Midgard.

There is one last field in the header for write-after-read dependencies involving the data register. When an instruction like the STORE instruction executes, it may not read the data register right away. There might be too many requests queued for the load/store unit at once, so there isn’t space for our STORE data. If another instruction clobbers the data to be stored, it will be corrupted when we actually go to read it. If we need to write to the data register for some other purpose, the "Data Register Write Barrier" bit must be set on the clause before the clause which actually does the write. This bit will stall execution until all the outstanding data register reads actually happen, so the contents aren’t needed anymore. It’s less heavy-handed than waiting for the STORE to actually complete, reducing the time spent stalling.

Instruction type

The "instruction type" and "next clause instruction type" fields tell whether the clause has a variable-latency instruction, and if it does, which kind. Unsurprisingly, the "next clause instruction type" field applies to the next clause to be executed. If the clause doesn’t have any variable-latency instructions, then the whole scoreboarding mechanism is skipped — the clause is always executed immediately and it never sets or clears any

more on the meaning of the 64-bit instruction type, see the 64-Bit Clauses part of the register field documentation.

Value	Instruction type
0	no variable-latency instruction
5	SSBO store
6	SSBO load
15	64-bit (implicitly no variable-latency instruction)

TODO: fill out this table

Instructions

Now that we know how instructions and constants are packed into clauses, let’s take a look at the instructions themselves. Each instruction is executed 3 stages, the register read/write stage, the FMA stage, and the ADD stage, each of which corresponds to a part of the 78 bit instruction word. We’ll describe what they do, and the format of their part of the instruction word, in the next sections. We’ll go by the order they’re executed, as well as the order of the bits in the instruction word.

Register read/write (35 bits)

As the name suggests, this stage reads from and writes to the register file. The current instruction reads from the register file at the same time that the previous instruction writes to the register file. Thus, the field contains both reads from the current instruction and writes from the previous instruction. Presumably, the scheduler makes this happen by interleaving the execution of multiple clauses from different quads. It only executes one instruction from a given quad every 3 cycles, so that the register write phase of one instruction happens at the same time as the register read phase of the next. Of course, it’s possible that the FMA and ADD stages take more than 1 cycle, and more threads are interleaved as a consequence; this is a microarchitectural decision that’s not visible to us. The result is that a write to a register that’s immediately read by the next instruction won’t work, but that’s never necessary anyways thanks to the passthrough sources detailed later.

The register file has four ports, two read ports, a read/write port, and a write port. Thus, up to 3 registers can be read during an instruction, or 2 if the previous instruction writes 2 registers. These ports are represented directly in the instruction word, with a field for telling each port the register address to use. There are three outputs, corresponding to the three read ports, and two inputs, corresponding to the FMA and ADD results from the previous stage. The ports are controlled through what the ARM patent calls the "register access descriptor," which is a 4-bit entry that says what each of the ports should do. Finally, there is the uniform/const port, which is responsible for loading uniform registers and constants embedded in the clause. Note that the uniforms and constants share the same port, which means that only one uniform or one constant (but not both) can be loaded for an instruction. This port supplies 64 bits of data, though, so two 32-bit parts of the same 64-bit value can be accessed in the same instruction.

The format of the register part of the instruction word is as follows:

Field	Bits
Uniform/const	8
Port 3 (write)	6
Port 2 (read/write FMA)	6
Port 0 (read)	5
Port 1 (read)	6
Control	4

Control is what ARM calls the "register access descriptor." To save bits, if the Control field is 0, then Port 1 is disabled, and the field for Port 1 instead contains the "real" Control field in the upper 4 bits. Bit 1 is set to 1 if Port 0 is disabled, and bit 0 is reused as the high bit of Port 0, allowing you to still access all 64 registers. If the Control field isn’t 0, then both Port 0 and Port 1 are always enabled. In this way, the Control field only needs to describe how Port 2 and Port 3 are configured, except for the magic 0 value, reducing the number of bits required.

Port 0 is greater than Port 1, it subtracts 63 from both numbers to get the real register. This lets software encode every possible combination of registers loaded in Port 0 and Port 1, possibly requiring it to swap Port 0 and Port 1.

Additionally, if the register control field writes to Port 3 but doesn't read or write from Port 2, the compiler appers to copy the value in Port 3 over to Port 2. The reason for this is unknown.

Before we get to the actual format of the Control field, though, we need to describe one more subtlety. Each instruction's register field contains the writes for the previous instruction, but what about the writes of the last instruction in the clause? Clauses should be entirely self-contained, so we can't look at the first instruction in the next clause. The answer turns out to be that the first instruction in the clause contains the writes for the last instruction. There are a few extra values for the control field, marked "first instruction," which are only used for the first instruction of a clause. The reads are processed normally, but the writes are delayed until the very end of the clause, after the last instruction. The list of values for the control field is below:

Value	Meaning
1	Write FMA with Port 3
3	Write FMA with Port 3, read with Port 2
4	Read with Port 2
5	Write ADD with Port 3
6	Write ADD with Port 3, read with Port 2
8	Nothing, first instruction
9	Write FMA with Port 3, first instruction
11	Nothing
12	Read with Port 2, first instruction
13	Write ADD with Port 3, first instruction
15	Write ADD with Port 3, write FMA with Port 2

64-bit clauses

When the 64-bit clause type is enabled, instructions operate on aligned 2-register pairs, where the `Rn` contains the low 32 bits for each thread in a quad and `R(n+1)` specifies the high 32. One of the patents mentions an alternate mode where `Rn` contains the first two full 64-bit values for the quad and `R(n+1)` contains the second two, but so far this hasn't been observed. When the 64-bit clause type is enabled, the register field has a totally different encoding. Ports 0-3 are similar, but now they load/store 64 bits each, in addition to the uniform/const port (which remains the same). The format of the register field when this mode is enabled is:

Field	Bits
Uniform/const	8
Port 2	5
Port 3	5
Port 0	4
Port 1	5

Control	5
Unknown (=7)	3

Note that each port has one less bit, since the register must be 2-aligned. For example, setting Port 0 to 1 would mean loading the low 32 bits in `R2` and the high 32 bits in `R3`.

The encoding of what each port does is now significantly more complicated. The previous scheme, where port 1 may contain the actual control field, is significantly expanded: the actual control field may now be spread across the Control, Port 1, and Port 3 fields!

First, we'll describe how ports 0 and 1 are encoded. The same scheme from before is used to save a bit from Port 0, where the fields are compared and both are subtracted from 31 (instead of 63) when `Port 0 > Port 1`. In addition, when `Port 0 == Port 1`, then only port 0 is actually active (port 1 is disabled). However, there are two cases we haven't described how to handle: when loading a single register pair which is at least `R32` (since this is too big for Port 0) and when loading no registers. Both are handled through a few special control values. There is a special value for Control to indicate that Port 1 is to hold a control value, and then the two cases are distinguished by the control value in Port 1. Based on the control value in Port 1, either Port 0 is disabled or 16 is implicitly added to its value, so that 0 loads `R32-33`.

If there is a control value in Port 3, then it controls what port 2 does. Either Port 2 is disabled, or it is used to either load a register, store the FMA result, or store the ADD result. The only remaining cases are when both ports are used: either port 2 is used to load a register while port 3 stores the FMA or ADD result, or both ports are used to store the FMA/ADD results (port 2 stores FMA, port 3 stores ADD). These cases are indicated using the main Control field, in addition to a special case to indicate that there is a control value in Port 3. These cases are also specified in the Port 1 control field if it exists, although it's redundant.

Algother, there are 6 possible cases indicated by the main Control field:

- Both port 0 and port 1 are used, and both port 2 and port 3 are used to store the FMA and ADD results.
- Both port 0 and port 1 are used, port 2 is used to load, and port 3 is used to store the ADD result.
- Same as above, but port 3 is used to store the FMA result.
- Both port 0 and port 1 are used, but port 2 usage is indicated by the control in port 3.
- Port 0 usage is indicated by the control in port 1, and both port 2 and port 3 are used to store the FMA and ADD results.
- Port 0 usage is indicated by the control in port 1, and port 2 usage is indicated by the control in port 3.

And 4 possible cases for port 1 as a control field:

- Add 16 to port 0, and both FMA and ADD are stored (no control in port 3)
- Port 0 is disabled, and both FMA and ADD are stored.
- Add 16 to port 0, and there is a control in port 3.
- Port 0 is disabled, and there is a control in port 3.

The actual encoding of the control field is shown below. The table shows all possible control values, even though each of them can only appear in one location. If a box is left blank, that means it's not specified by that control value (it is specified by the control value in another field). When multiple control values in multiple fields specify what the same field does, they must agree, or the encoding is invalid. The "allowed in" field specifies where this control field is allowed to be. In Port 0, "Read +16" means to add 16 to the value to get the actual register to be read.

Value	Allowed in	Port 0	Port 1	Port 2	Port 3
0	Port 1	Read +16	Control	Write FMA	Write ADD
2	Port 3			Write ADD	Control
3	Port 1	Unused	Control	Write FMA	Write ADD
6	Port 3			Write FMA	Control
7	Port 3			Unused	Control
8	Main	Read	Read	Write FMA	Write ADD

10	Port 3			Read	Control
12	Port 1	Read +16	Control		Control
15	Port 1	Unused	Control		Control
17	Main	Read	Read	Read	Write ADD
26	Main	Read	Read	Read	Write FMA
27	Main	Read	Read		Control
29	Main		Control	Write FMA	Write ADD
31	Main		Control		Control

To decode the register field, first look up the entry corresponding to the value of the Control field (it had better be "allowed in Main"). If there are any fields in that entry that say "Control", look up their values in the same table (they had better have the appropriate "allowed in") and fill in the blanks with what they say. At the end, every field should have a consistent assignment.

Now, we describe the encoding algorithm. First, we assign registers to ports in almost the same manner as before. If there are at least two registers to read, assign the first two to ports 0 and 1, with the smaller in port 0. If there are three registers to read, the third goes in port 2. If both FMA and ADD will write a register, they go in port 2 and port 3 respectively, as before. However, if only one stage writes a register, then port 2 is now preferred over port 3 — only if port 2 is used for reading should port 3 hold the register to be written.

The rest of the encoding algorithm is described below.

- If both port 0 and port 1 are used:
  - If port 0 reads `R32` or above, then subtract both from 15 similar to before, and store them.
  - If port 3 is unused, set Control = 27 and set port 3 as below.
  - If port 2 is used for reading:
    - If port 3 writes ADD, set Main = 17.
    - If port 3 writes FMA, set Main = 26.
  - Otherwise, both port 2 and port 3 must be used for writing, so set Main = 8.
- Otherwise:
  - If port 0 loads `R30` or below, set port 0 and port 1 to the same value, and then follow the instructions above as if both ports 0 and 1 were used.
  - If both FMA and ADD are written, set Main = 29. Otherwise set Main = 31.
  - If port 0 is used, set the value of the Port 0 field to the actual value minus 16, and:
    - If both FMA and ADD are written, set Port 1 = 0.
    - Otherwise, port 3 is unused, so set Port 1 = 12 and set port 3 as below.
  - Otherwise:
    - If both FMA and ADD are written, set Port 1 = 3.
    - Otherwise, port 3 is unused, so set Port 1 = 15 and set port 3 as below.
- If port 3 is unused:
  - If port 2 is unused, set Port 3 = 7.
  - If port 2 writes ADD, set Port 3 = 2.
  - If port 2 writes FMA, set Port 3 = 6.

Unlike the other ports, the uniform/const port always loads 64 bits at a time. If an FMA or ADD instruction only needs 32 bits of data, the high 32 bits or low 32 bits are selected later in the source field, described below.

The uniform/const bits describe what the uniform/const port should load. If the high bit is set, then the low 7 bits describe which pair of 32-bit uniform registers to load. For example, 10000001 would load from uniform registers 2 and 3. If the high bit isn't set, then the next-highest 3 bits indicate what 64-bit immediate to load, while the low 4 bits contain the low 4 bits of the constant. The mapping from from bits to constants is a little strange:

Field value	Constant loaded
4	0
5	1
6	2
7	3
2	4
3	5
0	special (see below)
1	unknown (also special?)

The uniform/const port also supports loading a few "special" 64-bit constants that aren't inline immediates, or loaded through the normal uniform mechanism. These usually have to do with fixed-function hardware like blending, alpha testing, etc. The known ones are listed below:

Field value	Special constant
00	Always zero.
05	Alpha-test data (used with ATEST)
06	gl_FragCoord sample position pointer
08-0f	Blend descriptors 0-7 (used with BLEND to indicate which output to blend with)

The gl_FragCoord pointer is a pointer to an array, indexed by gl_SampleID in R61 (see the varying interpolation section), of 16-bit vec2's that when loaded with a normal LOAD instruction, gives the sample (xy) position used for calculating gl_FragCoord.

Source fields

When the FMA and ADD stages want to use the result of the register stage, they do so through a 3-bit source field in the instruction word. There are as many source fields are there are sources for each operation. The following table shows the meaning of this field:

Field value	Meaning
0	Port 0
1	Port 1
2	Port 2

3	FMA: always 0
	ADD: result of FMA unit from same instruction
4	Low 32 bits of uniform/const
5	High 32 bits of uniform/const
6	Result of FMA from previous instruction (FMA passthrough)
7	Result of ADD from previous instruction (ADD passthrough)

FMA (23 bits)

Both the FMA and ADD units have various instruction formats. The high bits are always an opcode, of varying length. They must have the property that no opcode from one format is a prefix for another opcode in a different format. This guarantees that no instruction is ambiguous. Since there's no format tag, it would seem that decoding which format each instruction has is complicated, although it's possible that some trick is used to speed it up. In the disassmbler, we just try matching each opcode with the actual one, masking off irrelevant bits. I'm only going to list the categories here, and not the actual opcodes; I'll leave the former to the disassembler source code, simply because there are a lot of them and it's tedious to type them all up, and error-prone too. The disassembler is at least easy to test, so the chances of making a mistake are lower.

One Source (FMAOneSrc)

Field	Bits
Src0	3
Opcode	20

Two Source (FMATwoSrc)

Field	Bits
Src0	3
Src1	3
Opcode	17

Floating-point Comparisons (FMAFcmp)

Field	Bits
Src0	3
Src1	3
Src1 absolute value	1
unknown	1
Src1 negate	1
unknown	3

Src0 absolute value	1
Comparison op	3
Opcode	7

Where the comparison ops are given by:

Value	Meaning
0	Ordered Equal
1	Ordered Greater Than
2	Ordered Greater Than or Equal
3	Unordered Not-Equal
4	Ordered Less Than
5	Ordered Less Than or Equal

Two Source with Floating-point Modifiers (FMATwoSrcFmod)

Field	Bits
Src0	3
Src1	3
Src1 absolute value	1
Src0 negate	1
Src1 negate	1
unknown	3
Src0 absolute value	1
unknown	2
Outmod	2
Opcode	6

The output modifier (Outmod) is given by:

Value	Meaning
0	Nothing
1	max(output, 0)

2	clamp(output, -1, 1)
3	saturate - clamp(output, 0, 1)

Three Source (FMAThreeSrc)

Field	Bits
Src0	3
Src1	3
Src2	3
Opcode	14

Three Source with Floating Point Modifiers (FMAThreeSrcFmod)

Field	Bits
Src0	3
Src1	3
Src2	3
unknown	3
Src0 absolute value	1
unknown	2
Outmod	2
Src0 negate	1
Src1 negate	1
Src1 absolute value	1
Src2 absolute value	1
Opcode	2

Four Source (FMAFourSrc)

Field	Bits
Src0	3
Src1	3

Src2	3
Src3	3
Opcode	11

ADD (20 bits)

The instruction formats for ADD are similar, except it can only have up to 2 sources instead of FMA's four sources.

One Source (ADDOneSrc)

Field	Bits
Src0	3
Opcode	17

Two Source (ADDTwoSrc)

Field	Bits
Src0	3
Src1	3
Opcode	14

Two Source with Floating-point Modifiers (ADDTwoSrcFmod)

Field	Bits
Src0	3
Src1	3
Src1 absolute value	1
Src0 negate	1
Src1 negate	1
unknown	2
Outmod	2
unknown	2
Src0 absolute value	1
Opcode	4

Field	Bits
Src0	3
Src1	3
Comparison op	3
unknown	2
Src0 absolute value	1
Src1 absolute value	1
Src0 negate	1
Opcode	6

Special instructions

This section describes some of the instructions that interact with fixed-function units like the texture sampler, varying interpolation unit, etc. They oftentimes use a special format, since they have to pass extra information to the fixed-function block.

Texture instructions

A single instruction encoding is used for every texture operation. What operation to do, and what texture/sampler to use, is encoded in a 32-bit control word which is loaded from the uniform/immediate port. The intention is that it's an immediate, basically used to extend the instruction. Trying to encode everything in the instruction wouldn't go too well, simply because there are too many possible combinations.

The texture instruction is encoded similar to a normal two-source instruction, with the bottom 6 bits devoted to sources, except that bit 6 is also used to encode whether the control word comes from the low 32 bits or high 32 bits of the port. The instruction can read up to 6 registers, with two coming from normal sources and four coming from the special per-clause register (hence they must be adjacent, starting on a multiple of 4). The texture unit then writes the result to the same per-clause register. Hence, those 4 registers are (sometimes) read and then later written.

The format of the control word is as follows:

Field	Bits
Sampler Index/Indirects	4
Texture Index	7
Separate Sampler and Texture	1
Filter	1
unknown	2
Texel Offset	1
Shadow	1
Array	1
Texture Type	2

Compute LOD	1
No LOD Bias	1
Calculate Gradients	1
unknown	1
Result Type	4
unknown	4

Unlike other architectures, where samplers and textures are just handles passed around, Bifrost seems to use an older binding-table-based approach. There is are two tables, one for samplers and one for textures, containing all the state for each, and the texture instruction chooses an index into each table. If the "Separate Sampler and Texture" bit is set, then the "Texture Index" supplies the index into the texture table, and "Sampler Index/Indirect" provides the index into the sampler table. If the bit is unset, then by default, both indices come from from the "Texture Index" field. However, the "Sampler Index/Indirects" field then provides a bitmask of indirect sources. If the low bit is set, the texture index is "indirect," i.e. it is obtained by a register source. Similarly, if the second-lowest bit is set, the sampler index is indirect. The next two bits are unknown, and always observed as 11. Note that indirect indices are only possible if "Separate Sampler and Texture" is unset — if it is set, then both indices are always taken directly from their respective fields in the control word.

The "filter" bit is unset for `texelFetch` and `textureGather` operations, where the normal bilinear filtering pipeline is bypassed. The "texel offset" is used for `textureOffset`, and specifies an extra register source for the offset. "Shadow" is used for shadow textures, and implies an extra source for the depth reference. "Array" is similarly used for array textures, and implies another source for the array index. Texture Type indicates the type of texture, as in the following table:

Field Value	Type
0	Cube
1	Buffer
2	2D
3	3D

The next three bits select between `texture`, `texture` with an LOD bias passed, `textureLod`, and `textureGrad`. All of these provide different ways of changing the computed Level of Detail before sampling the texture. The usage is as follows:

GLSL operation	Compute LOD	No LOD bias	Calculate Gradients
plain `texture`	1	1	1
`texture` with LOD bias	1	0	1
`textureLod`	0	0	1
`textureGrad`	1	1	0

The actual gradients for `textureGrad` would take up too many registers, so they get stored in a separate earlier texture instruction with some unknown fields changed.

Finally, the result type is interpreted based on the following table:

4	32-bit floating point
14	32-bit signed integer
15	32-bit unsigned integer

TODO: compact texture instructions, dual texture instructions

Varying Interpolation

The varying interpolation unit is responsible for, well, interpolating varyings. This gets more complicated with support for per-sample shading and centroid interpolation, which is where most of the complexity comes from. In per-sample shading, the fragment shader is run for each sample, compared to traditional multisampling where the fragment shader is run once per pixel, and samples have different colors only if they are covered by different triangles (or with alpha-to-coverage). In that case, the coordinates used for sampling have to come from the current sample. If the fragment shader is run per-pixel as usual, then the coordinates are normally taken from the center of the pixel. But this doesn't work so well for multisampling, where the sample might not actually lie on the triangle if the triangle doesn't go through the center of the pixel. So, there's an alternate "centroid" sampling method that chooses the coordinates based on the covered samples. Finally, the shader can specify an offset for the coordinates through `interpolateAtOffset` in GLSL. After all this, the result is the coordinates of the point to be sampled. These coordinates are converted to barycentric coordinates, and then the usual barycentric interpolation is applied to the varyings at all three corners of the triangle to get the final result.

On Bifrost, this is all handled through a single instruction. Varyings are always vec4-aligned, i.e. the address is always expressed in 128-bit units, even though there are instructions for interpolating single floats, vec2's, vec3's and vec4's. The instruction always has at least one source, which is always R61. Before the shader starts, R61 is filled out as follows: the low 16 bits contain the sample mask (`gl_SampleMaskIn`), the next 4 bits contain the sample ID if doing per-sample shading (`gl_SampleID`), and the rest of the bits are unknown (always 0?). The sample mask is needed for doing centroid sampling, while the sample ID is needed for doing per-sample shading.

The format of the instruction as follows:

Field	Bits
Src0	3
Address	5
Components	2
Interpolation Type	2
Reuse previous coordinates	1
Flat shading	1
Opcode (=10100)	5

The address field requires some explanation. Only addresses under 20 can be encoded directly. This table shows exactly how the field is decoded.

Bit pattern	Meaning
`0xxxx`	Interpolate at address `0xxxx` .
`100xx`	Interpolate at address `100xx` .
`10110`	gl_FragCoord.w
`10111`	gl_FragCoord.z

`11xxx`	The address is indirect, given by an additional source indicated by `xxx`.

To get the number of components, add one to the components field.

The interpolation field has the following meaning:

Value	Meaning
0	Force per-fragment sampling (when per-sample shading is enabled).
1	Centroid sampling.
2	Normal — per-sample shading if enabled through GL state, otherwise pixel center.
3	Use explicit coordinates loaded through a previous instruction.

If the coordinates computed would be the same as the previous interpolation instruction, the "reuse previous coordinates" bit can be set. Finally, the "flat shading" bit enables flat shading, where the varying is chosen from one of the triangle corners based on the GL provoking vertex rules.

Special functions

Rather than having a dedicated unit for more complicated floating-point operations, Bifrost reuses the fixed-latency general-purpose FMA/ADD pipeline, meaning it uses regular fixed-latency instructions. Since most of these operations have a latency much longer than most FMA and ADD instructions, they have to be broken up into a number of simpler steps. This section explains each of the algorithms and special instructions used to implement each complex builtin function in GLSL.

Reciprocal (G71)

The reciprocal implementation uses an initial table lookup followed by a single step of Newton's method. Given the input `a`, first, we do an argument reduction step, computing a reduced `a1` in the range [0.5, 1.0) and an initial approximation `x1` of `1/a1`. Computing `a1` is done with the `FRCP_FREXPM` instruction which is the same as the mantissa part of `frexp` in C. Except for special cases, it just sets the exponent to -1. `x1` is looked up in a table based on the top 17 bits of the mantissa of `a` (the mantissa of `a` is always the same as the mantissa of `a1`) using the `FRCP_TABLE` instruction. Next, we compute `x2 = x1 * (2 - a1 * x1)` and do the final exponent adjustment to compensate for the argument reduction. `x2` is actually computed as `x1 + x1 * (1 - a1 * x1)`, using two fused multiply-adds, to avoid rounding errors. The computation of `1 - a1 * x1` is particularly sensitive, since `a1 * x1` is very close to one by design. The second fused multiply-add and exponent adjustment are done in the same instruction, using a special `FMA_MSCALE` instruction which takes a fourth argument that is added to exponent of the result. There is an `FRCP_FREXPE` instruction which computes the exponent that `frexp` would have, and then negates it, to get the right bias for the final exponent adjustment.

There is one final complication, which is that if `a` is infinity, then `a1` will be infinity and `x1` will be zero, and multiplying them will give NaN according to IEEE arithmetic, which will propagate to the final result instead of returning the appropriately-signed zero as expected. Also, internal details of the sign of `+/-NaN * +/-NaN` mean that the returned NaN will have the wrong sign when computing 1/NaN. To fix these issues with corner cases, ARM uses a special `rcp_mode` modifier which only exists on the aforementioned `FMA_MSCALE` instruction for the first FMA. The exponent bias is set to 0, so it is otherwise identical to a normal `FMA` instruction.

All together, `x=1/a` is computed as follows:

```
a1 = FRCP_FREXPM a
x1 = FRCP_TABLE a
t1 = FMA_MSCALE.rcp_mode a1, -x1, 1.0, 0
e' = FRCP_FREXPE a
x = FMA_MSCALE t1, x1, x1, e'
```

Reciprocal (later versions)

On every Bifrost core released after G71, there is a new `FRCP_FAST` instruction that does all of the above in one instruction. Since there are two dependent fused multiply-adds, this can't feasibly be done in one cycle. Instead, even though `RCP_FAST` is encoded as an ADD instruction, it actually uses both FMA and ADD units, and the corresponding FMA instruction before it must be a nop.

Similarly to reciprocal, we use a lookup table followed by one iteration of Newton's method. The computation of `a1` as well as the required fixup is computed by the `FSQRT_FREXPM` (shared with square root) and `FRSQ_FREXPE` instructions, which are analogous to `FRCP_FREXPM` and `FRCP_FREXPE`, except that because we're taking the square root, only multiplying by four produces a power-of-two change in the output, so the input is only scaled to [0.25, 1) instead of [0.5, 1.0), and the exponent is divided by two when computing the fixup in `FRSQ_FREXPE`. The Newton's method fixup step is mathematically given by `x2 = .5 * x1 * (3 - a1 * x1 * x1)`. This is rearranged to give `x2 = x1 + x1 * 0.5 * (1 - a1 * x1 * x1)`, which is computed by squaring `x1` and then computing two fused multiply-adds. The multiplication by 0.5 is accomplished by passing -1 to the scale parameter of `FMA_MSCALE` for the first fused multiply-add. Both fused multiply-adds use the `rcp_mode` bit (not sure why the second one does?).

```
a1 = FSQRT_FREXPM a
x1 = FRSQ_TABLE a
t1 = FMA x1, x1, -0
t2 = FMA_MSCALE.rcp_mode a1, -t1, 1.0, -1
e' = FRSQ_FREXPE a
x = FMA_MSCALE.rcp_mode t2, x1, x1, e'
```

Reciprocal square root (later versions)

Similar to `FRCP_FAST`, there is now a `FRSQ_FAST` instruction added on models after G71 which has the same restriction (it also takes 2 cycles).

Square root (G71)

For the argument reduction, square root uses the same `FSQRT_FREXPM` instruction, but since the fixup exponent is the opposite sign of reciprocal square root, there is a `FSQRT_FREXPE` instruction that does the right thing for square roots. The initial estimate `x1` is computed by using `FRSQ_TABLE` to make an initial estimate `r1` of the reciprocal square root, and then computing `x1 = a1 * r1`. There is a subtlety here, since if `a` is either 0 or infinite, then `r1` will be the opposite, and we always want the result to be equal to `a` in that case. This is accomplished using a special `sqrt_mode` modifier on `FMA_MSCALE`, similar to `rcp_mode`.

Next, there is a Newton-Raphson step as before. The normal Newton-Raphson step for square root is `x2 = .5 * (x1 + a / x1)`. First, we rearrange this to `x2 = x1 + .5 * (a / x1 - x1)`, and then pull out a factor of `1/x1` from the error term to get `x2 = x1 + .5 / x1 * (a - x1 * x1)`. The only thing that can't be immediately computed is `1 / x1`, but `r1` is already a good enough approximation of that. So, the formula used is `x2 = x1 + r1 * (a - x1 * x1) * .5`. This doesn't correctly return -0 given -0, however, so the final formula is `x2 = x1 - abs(r1) * (x1 * x1 - a) * .5`. The `* .5` is folded into the first FMA, and the final exponent correction is folded into the second FMA, as usual. The second FMA must use `rcp_mode` to prevent another 0 times infinity problem, although this time, since `x1` is being added, it's ok to return 0 always.

```
a1 = FSQRT_FREXPM input
r1 = FRSQ_TABLE input
x1 = FMA_MSCALE.sqrt_mode a1, r1, -0, 0
e = FSQRT_FREXPE input
t2 = FMA_MSCALE x1, x1, -a1, -1
output = FMA_MSCALE.rcp_mode abs(r1), -t2, x1, e
```

Square root (later)

There is no `FSQRT_FAST` instruction. Instead, we compute the reciprocal square root using the `FRSQ_FAST` instruction, and then multiply by the input to get the answer. We similarly have to be careful about 0, so we use the same `sqrt_mode` modifier. Apparently ARM also decomposes the number before handing it to `FRSQ_FAST` and applies the exponent fixup, although I'm not sure why this is necessary. The result is only 1 less instruction than before.

```
a1 = FSQRT_FREXPM input
e = FSQRT_FREXPE input
r1 = FRSQ_FAST a1
output = FMA_MSCALE.sqrt_mode a1, r1, -0, e
```

Natural & Base-2 Log (G71)

As described in the patent, the argument reduction step for log depends on the fact that `log(a * b) = log(a) + log(b)`. First, we reduce the input `a` to the range [0.75, 1.5) and calculate the required fixup exponent using the `FLOG_FREPXE` instruction. This fixup exponent is converted to a floating-point number, and is then added to the result later (after multiplying by ln(2) for base-e log), using the fact that `log(m * 2^e) = e + log(m)`. We lookup a very coarse-grained reciprocal estimate `r1` to the reduced input `a1` in a table using the `FRCP_APPROX` instruction, and we also lookup `-log(r1)` using another table that stores for each input `x`, `-log(FRCP_APPROX(x))`. Both these instructions take `a`, and do the reduction to [0.75, 1.5) implicitly. Since `log(a1) = log(a1 * r1) - log(r1)`, all we need to do to compute `log(a1)` is compute `log(a1 * r1)`. Since `a1 * r1` is close to 1 by design, we can do this with a polynomial approximation of `log(y + 1)` with only a few terms. The approximation chosen for base *e* is `log(y + 1) = y * (1.0 + y * (a + y * b))` where the precise constants used are `b = 0x3eab3200 (0.33436)` and `a =`

`(1.29249 *  2^{-26})`. `d` holds the error due to approximating `1/ln(2)` as a single-precision floating point number, so that `c + d`, when evaluated at infinite precision, is a much better approximation to `1/ln(2)` than `c` by itself.
base 2 logarithm:

```
a1 = LOG_FREXPM a
r1 = FRCP_APPROX a
xt = FLOG2_TABLE a
e = FLOG_FREXPE a
e = I32_TO_F32 e
x1 = ADD.f32 T1, e, xt
y = FMA.f32 a1, r1, -1.0
y' = FMA.f32 y, 0x3fb8aa3b /* c */, -0
t1 = FMA.f32 y, 0x3eab3200 /* b */, 0xbf0003f0 /* a */
t2 = FMA.f32 y', t1, 0x32a57060 /* d */
x2 = FMA.f32 T0, y, t2, y'
output = ADD.f32 x1, x2
```

base e logarithm:

```
a1 = LOG_FREXPM a
r1 = FRCP_APPROX a
xt = FLOGE_TABLE a
e = FLOG_FREXPE a
e = I32_TO_F32 e
x1 = FMA.f32 T0, e, 0x3f317218 /* ln(2) */, xt
y = FMA.f32 a1, r1, -1.0
t1 = FMA.f32 y, 0x3eab3200 /* b */, 0xbf0003f0 /* a */
t2 = FMA.f32 y, t1, 1.0
output = FMA.f32 {R0, T0}, y, t2, x1
```

Natural & Base-2 Logarithm (later)

Exponent (G71)

To compute the exponent, we want to reduce the input a into a multiple a_1 of $\log_2(b)$, where b here is the base, plus a remainder a_2, since $b^a = 2^{a_1} b^{a_2}$, and as usual, the final multiplication by 2^{a_1} can be done easily using a `*_MSCALE` instruction. In fact, to reduce the range required for the approximation to b^{a_2}, we can go even farther and decompose the input into a multiple of $2^{-4}\log_2(b)$ instead. 2^{a_1} can be computed using a small lookup table for the 4 fractional bits, plus the usual post-correction for the integral bits. Finally, we compute b^{a_2} using a polynomial approximation.

In order to do the decomposition, we use a well-known floating-point trick. Letting m be the number of mantissa bits, we multiply a by $\log_2(b)^{-1}$ and then add $f = 1.5 \times 2^{m-4}$. If we assume that $-2^{m-5} \leq a \times \log_2(b)^{-1} < 2^{m-5}$, then the result will be between 2^{m-4} and 2^{m-3}. In particular, the floating-point exponent will be $m-4$, and the mantissa will be $2^{m-3} + a_1$. We can recover a_1 as an integer by reinterpreting the floating-point number as an integer and subtracting $1.5 \times 2^{m-4}$ reinterpreted as an integer, since this will remove the exponent bits and remove the bias of 2^{m-3} from the mantissa bits. In order to extract the remainder a_2, we undo the addition and scaling, which will produce $2^{-4} \times \log_2(b) \times a_1$, and then subtract the result from a. The result of this addition is clamped from -1 to 1, presumably to prevent too-large offsets.

There are two special cases to worry about: when $a < -2^{m-5}$, and when $a > 2^{m-5}$. In the first case, as long as the result of adding f is positive, then the exponent will be smaller, so our a_1 will be a large negative number, and after adding it to the exponent, we will get 0 as desired. However, if the result is negative, we will get a large positive number due to the sign bit, which is not what we want. So we need to clamp the result so that it does not go below 0 (thankfully, there is the `.clamp_0_inf` modifier which does this without any additional instructions). If $a > 2^{m-5}$, then the exponent will be larger than expected, so we'll get a large positive number for a_1 which will give us infinity as desired.

The blob actually clamps the result to 2^{m+64}, using the `ADD_MSCALE` instruction to scale by 2^{-64} after adding f combined with the `.clamp_0_1` modifier to then clamp the result between 0 and 1. It also adjusts the subsequent steps to account for the extra scaling. However, tests indicate that this is unnecessary.

The polynomial approximation used for base 2 is $(a_2 \times ((a_2 \times a + b) \times a_2 + c)) \times 2^{a_{1f}} + 2^{a_{1f}}$, where the constants are `a = 0x3d635635 (0.05502)`, `b = 0x3e75fffa (0.240234)`, and `c = 0x3f317218 (0.693147)`. Note how the final multiplication by $2^{a_{1f}}$, the result of the table lookup on the fractional part of a_1, has been folded into the expression. The polynomial approximation for base *e* is $(a_2 \times a_2 \times (a_2 \times a + b) + a_2) \times 2^{a_{1f}} + 2^{a_{1f}}$, where `a = 0x3e2aaacd (0.166667)` and `b = 0x3f00010e (0.500016)`.

The final fused multiply-add is actually an `FMA_MSCALE` instruction, with the exponent bias from the integer part of a_1 added in. In addition, the result is clamped from going below 0, presumably to prevent any small errors from making the result go below 0 for very negative inputs. Finally, since all the clamping done earlier flushes NaN's to zero, but we want the output to be NaN if the input is NaN, we take the maximum of the original input and the output. Normally this wouldn't do anything, since $e^x > x$ and $2^x > x$, but we use a special `.nan_wins` modifier which makes sure that the output is NaN if either input is NaN, making sure that the NaN is propagated correctly.

```
t1 = ADD_MSCALE.f32.clamp_0_1 a, 0x49400000 /* 786432.000000 */, -0x40
t2 = ADD_MSCALE.f32 t1, 0xa9400000, 0x40
a2 = ADD.f32.clamp_m1_1 a, -t2
a1t = EXP_TABLE t1
t3 = SUB.i32 t1, 0x29400000 /* 0.000000 */
a1i = ARSHIFT t3, 4
p1 = FMA.f32 a2, 0x3d635635 /* 0.055502 */, 0x3e75fffa /* 0.240234 */
p2 = FMA.f32 p1, a2, 0x3f317218 /* 0.693147 */
p3 = FMA.f32 a2, p2, -0
x = FMA_MSCALE.clamp_0_inf p3, a1t, a1t, a1i
x' = MAX.f32.nan_wins x, a
```

However, the following version was tested to return the same result for every possible input, and is a little simpler:

```
t1 = ADD.f32.clamp_0_inf a, 0x49400000 /* 786432.000000 */
t2 = ADD.f32 t1, 0xc9400000
a2 = ADD.f32.clamp_m1_1 a, -t2
a1t = EXP_TABLE t1
t3 = SUB.i32 t1, 0x49400000 /* 0.000000 */
a1i = ARSHIFT t3, 4
p1 = FMA.f32 a2, 0x3d635635 /* 0.055502 */, 0x3e75fffa /* 0.240234 */
p2 = FMA.f32 p1, a2, 0x3f317218 /* 0.693147 */
p3 = FMA.f32 a2, p2, -0
x = FMA_MSCALE.clamp_0_inf p3, a1t, a1t, a1i
x' = MAX.f32.nan_wins x, a
```

Base *e* exponent:

```
t1 = FMA.f32.clamp_0_1 a, 0x31b8aa3b, 0x3b400000
t2 = ADD.f32 t1, 0xbb400000
a2 = FMA_MSCALE.clamp_m1_1 t2, 0xcd317218 /* -186065280.000000 */, a, 0
a1t = EXP_TABLE t1
a1i = ARSHIFT t3, 4
t3 = SUB.i32 t1, 0x3b400000
p1 = FMA.f32 a2, 0x3e2aaacd /* 0.166667 */, 0x3f00010e /* 0.500016 */
p2 = FMA.f32 a2, p2, -0
p3 = FMA.f32 a2, p2, a2
x = FMA_MSCALE.clamp_0_inf p3, a1t, a1t, a1i
x' = MAX.f32.nan_wins x, a
```

Arc-tangent (G71 and later)

All the inverse trignonometric functions are computed using the two-argument arc-tangent function (sometimes called `atan2`, although it is just an overload of `atan` in GLSL) which determines the angle of a given input 2D vector. In particular, $\arctan(y) = \arctan(y, 1.0)$, $\arcsin(y) = \arctan(y, \sqrt{1-y^2})$ and $\arccos(x) = \arctan(\sqrt{1-x^2}, x)$.

The first thing to note is that we can restrict the problem to the upper-left quadrant using symmetry. We calculate $\arctan(|y|, |x|)$ first. Taking the branch cut along the negative x-axis into account, one can check that in order to calculate $\arctan(y, x)$, it is sufficient to subtract π if *x* is negative, and then change the sign of the result to equal the sign of *y* (effectively negating the result if it has differing sign from *y*). The blob is careful to use integer instructions to determine the sign of *x* and *y*, so that the appropriate sign is returned even if one is zero, and $\arctan(-1, +0) = \pi$ while $\arctan(-1, -0) = -\pi$.

In order to actually calculate the arc-tangent, we use a novel reduction step in addition to the usual polynomial approximation. Before dividing *y* by *x*, we apply a linear transform $R = \begin{bmatrix} 1 & a \\ -a & 1 \end{bmatrix}$ to the original vector $\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$. This matrix has orthogonal columns and positive determinant $1 + a^2$, so it is a rotation (it also scales the input, but since the final division will remove any scale we add, this is irrelevant). In fact, since it sends the basis vector $\hat{x}$ to $\begin{bmatrix} 1 \\ -a \end{bmatrix}$, it is a rotation by an angle of $-\arctan(a)$. So, we have that $\arctan(\mathbf{x}) = \arctan(R\mathbf{x}) + \arctan(a)$, where we are free to choose *a*. If $a = y/x$, then $\arctan(R\mathbf{x}) = 0$ since the y-component of $R\mathbf{x}$ will be 0. We choose *a* to be a coarse-grained approximation of y/x, so that we can lookup $\arctan(a)$ in a table yet the range of $\arctan(R\mathbf{x})$ is small enough that it can be approximated using a polynomial with relatively few terms after dividing the transformed *y* by *x*.

There are a few special cases to deal with. When the coordinates are too large or too small, the calculation of $R\mathbf{x}$ may overflow or underflow, in which case a scale has to be applied to *x* and *y* by adding to the exponent. Also, we have to handle input infinities correctly. All of these special cases are handled using the `ATAN_ASSIST`, `ATAN_LDEXP.X` and `ATAN_LDEXP.Y` instructions. `ATAN_ASSIST` returns the appropriate scale as a 16-bit signed integer in the low 16 bits of the result, and *a* as a half-precision floating point number in the high 16 bits. The result of `ATAN_ASSIST` is

function, except that if passed 0x8000 (-2^{16}) as a scale, they will turn infinity into 1.0. `ATAN_ASSIST` passes this special scale if either *x* or *y* are infinity, so that the infinite coordinate becomes 1.0, while any non-infinite coordinate becomes 0.0 due to underflow. This produces the expected result and avoids infinities permeating the rest of the code. Then, $R\mathbf{x}$ is computed using two fused multiply-adds, using a modifiers to expand the high 16 bits of the `ATAN_ASSIST` instruction to 32 bits while also taking the absolute value of *x* and *y*. Finally, *y* is divided by *x*, the polynomial approximation to $\arctan(R\mathbf{x})$ is computed, $\arctan(a)$ is computed using `ATAN_TABLE` and is added, and the corrections for different quadrants are applied.
atan2:

```
a = ATAN_ASSIST y, x
y' = ATAN_LDEXP.Y.f32 y, a
x' = ATAN_LDEXP.X.f32 x, a
x'' = FMA.f32 {R7, T0}, abs(y'), R4.y, abs(x')
xm = FRCP_FREXPM x''
y'' = FMA.f32 {R4, T0}, -R4.y, abs(x'), abs(y')
xi1 = FRCP_TABLE x''
err = FMA_MSCALE.rcp_mode T0, xm, -xi1, 1.0, 0
xi2 = FMA.f32 err, x'', x''
xe = FRCP_FREXPE x''
y_over_x = FMA_MSCALE y'', xi2, -0, xe
y_over_x_sq = FMA.f32 y_over_x, y_over_x, -0
atan_a = ATAN_TABLE y, x
p1 = FMA.f32 y_over_x_sq, y_over_x_sq, 0x3e4b2a00 /* 0.198402 */, 0xbeaaaaab /* -0.333333 */
p2 = FMA.f32 y_over_x_sq, p1, 1.0
corr = CSEL.UGE.i32 x, 0x80000000, -pi, 0
atan_a = ADD.f32 atan_a, corr
atan_all = FMA_MSCALE p2, y_over_x, atan_a, 0
result = MUX y, atan_all, 0x80000000 # copysign(y, atan_all)
----
```