Piculet Specification 0.29
2024-02-21 05:38:28Z

github.com/gabecampb

Preamble

Piculet is a bytecode set designed to provide interfaces for many kinds of programming. Each runtime environment (instance of the virtual machine) requires at least one display/window, and an implementation can optionally support more than one display to allow rendering across multiple monitors. Networking, rasterized graphics, ray traced graphics, audio, and almost all things done in modern software, is supported by this bytecode set.

Memory Layout, Addresses, and Regions

System memory, used for laying out the contents of mapped buffers and as storage for some defined implementation-provided information (see ***Hardware Information***), refers to all memory whose address is greater than the highest address of main memory. Main memory, directly before system memory, is the area of memory allowed for direct access (read/write) by instructions, beginning at memory address 0 up to the last byte. The main memory's contents is initially all 0. Other forms of memory (i.e., graphics states, the displays, the graphics memory, and the contents of any object handled by the implementation) cannot really be accessed as they lie in arbitrary locations outside of main and system memory, or in GPU memory; however, mapped buffer data, the hardware information, and any data the hardware information stores the address for is data whose memory addresses are guaranteed to be in a fixed location within system memory and such data may be read (and written, in the case of mapped buffer data). The locations of the contents of mapped buffers are guaranteed to be static for the duration of their mapping.

Regardless of whether the implementation is storing a buffer in RAM (outside of main and system memory and not in GPU memory) or in GPU memory, mapping that buffer will consist of the implementation copying its store to a region of system memory that is then marked available for read/write, and unmapping will consist of copying the (newly updated) region back to the buffer and then marking the region of system memory it occupied during the mapping inaccessible once again. Modifications to the store of mapped buffers will not take effect until the buffer is unmapped. All data provided for a buffer upload should be in little-endian.

A program will, during the implementation's initialization, be loaded into main memory starting at memory address 0 and up to memory address *n*-1, where *n* is the number of bytes in the program. If the program is 0 bytes long or does not fully fit into main memory, the runtime environment cannot start.

If any buffer data is to be stored in RAM, either by choice of the implementation or by requirement of the specification, it must be stored outside of both main and system memory. System memory is only for buffer mappings and some information defined to be provided by the implementation.

There exists one major region in system memory intended for obtaining information regarding the system and hardware. This is the hardware information, which is read-only and allocated once and updated throughout the lifetime of the runtime environment. Along with this, there are a few other read-only memory regions, referenced by addresses in the hardware information (see ***Hardware Information***).

Values in the hardware information and in the areas of system memory the hardware information refers to are stored in little-endian byte ordering, and they are just arrays of values arranged in the specified order, i.e., the lowest address points to the LSB of the first piece of data.

Cycles, Jump Instructions, and Execution

A cycle is defined as an execution of a varying length of time and count of instructions, spanning from the beginning of execution until the execution ends for one of various defined reasons. The cycle of a thread ends after instruction 37, instruction 40, instruction 41, instructions 44-59 (exceptions stated below), instruction 91, at some operations performed by instruction 124 (see ***Networking***), at termination of the thread, or at non-automatic changes to the value of the PC. When multithreading support is present, threads can execute their cycles simultaneously. Thread creation via instruction 37 ends the cycle of the thread and simply reserves the ID of the new thread, which is output to the output register; the new thread will actually be created at the next cycle of the thread that created it and until then the

reserved thread ID is treated as a non-existent/killed thread. Sleeping a thread via instruction 41 ends the cycle of the thread and prevents its execution for a specified amount of time.

If multithreading is not supported by the implementation, it is emulated on a single thread; all threads will execute one-by-one in rounds and in no specific order, and for each round of cycles if all of the threads are asleep at the end of the round the CPU should sleep for the minimum amount of sleep time across all threads. Any other threads that have a longer sleep duration than this minimum will, of course, still not be cycled in following rounds until their sleep duration has fully passed.

If an execution of instruction 37 or 40 is determined to do nothing due to one of the cases stated in the instruction's definition, the cycle is not ended.

If the conditions for a jump instruction are met, they immediately set the PC and then update the LR to the jump instruction's address + 1. If the conditions for a jump instruction are not met, the instruction does nothing and the cycle is not ended.

Threads

Thread 0 is the main thread of execution and begins at the instruction at memory address 0. Threads can parent (create new) threads; the created thread is considered a descendant of the thread that created it and of all threads whose thread creation led to the creation of its parent thread. If thread 0 is killed, so too is the runtime environment.

The first manually created thread has ID of 1, and all created after that are one higher than the last.

Thread Permissions and Memory Restrictions

For security purposes, it's imperative to be able to properly limit the capabilities of created threads. All threads have an instruction memory minimum, and instruction memory maximum. The range of accessible instruction memory addresses is from the minimum to the maximum (inclusive). Note there is no separate memory for instructions; instructions are obtained from the inside of main memory. When a thread attempts to fetch an instruction from outside of its instruction memory range, that thread will be killed. When an instruction will attempt to access (read/write) any addresses in main memory and not all addresses in the accessed range are within some segment(s) (see ***Memory Segmentation***) of the thread's segment table, that instruction will do nothing and the segmentation fault bit in the SR will be set. If an instruction will attempt to access any addresses of system memory, that instruction will do nothing and the segmentation fault bit in the SR will be set, with exceptions for cases where instruction 120 is attempting to read or write within the range of a single mapped buffer, where a load/store instruction (see ***Grouped Instructions***) is attempting to read or write within the range of a single mapped buffer, or where a load instruction is attempting to read from within the range of either the hardware information or one of the regions that hardware information refers to. If an instruction will attempt to access any mapped buffers for which the mapped object's privacy key does not match the privacy key of the accessing thread, that instruction will do nothing and the segmentation fault bit in the SR will be set. Instructions attempting to read or write to addresses that are part of both main memory and system memory will do nothing, and the segmentation fault bit in the SR will be set. If an instruction successfully reads or writes from main memory or system memory the segmentation fault bit in the SR will be unset. For the few exceptions where it is allowed to read or write to system memory, no offsets are applied to the accessed memory addresses within the range of system memory.

Threads also have permissions for whether or not they can capture images of the display, from the cameras, whether they can record audio, whether or not they can perform networking operations, whether or not they can perform file I/O operations, and whether or not they can create new threads. Additionally, they have a highest accessible directory for file I/O operations. Finally, they also have a 64-bit privacy key; any object generated by a thread will obtain an immutable copy of the privacy key when generated and will only ever able to be used by threads with matching privacy keys by reference to the object's ID, either provided in a register or by object IDs in a region of memory that is addressed by a register (an instruction will be skipped if an object referred to by ID in one of these two locations is not accessible by the thread executing the instruction). Any objects are automatically deleted when all threads with matching privacy keys are killed (every time a thread is killed, there will be a check for all objects to make sure that there still exists a thread with a matching privacy key). Threads can be created to use segment table objects even if their privacy key doesn't match the privacy key of the thread, however the creator thread must have a privacy key matching

that of the segment table object when the new thread is created.

For thread 0, the instruction memory range is set to include the entirety of main memory (minimum is 0, maximum is capacity of main memory in bytes - 1), it has all image and audio capture permissions, it has permission to perform networking operations, it has permission to perform file I/O operations, it has permission to create new threads, the highest accessible directory is /, and its privacy key is 0.

Memory Segmentation

All memory accesses to main memory performed by a thread are influenced by the thread's segment table. A segment table describes all segments of memory; each segment entry consists of the segment's beginning virtual address, the segment's beginning physical address, and the segment's length. All accesses to main memory are specified in terms of addresses in virtual address space, and the runtime environment will remap those addresses prior to the actual main memory access according to what the physical addresses are for the virtual addresses in the thread's segment table. This allows for the fragmentation of memory in such a way that accessed memory addresses can seem to be contiguous (in virtual address space), while in physical address space memory regions are actually dispersed in multiple locations (segments) throughout main memory. Segment tables are used only for main memory accesses and the memory addresses for instructions, while all stored within main memory, are not affected by memory segmentation. A thread is assigned a segment table object when created; the segment table object used by a thread can be modified through segment table updates but what segment table object is used by a thread is set at its creation and cannot change.

If a thread's bound segment table object is deleted, all accesses to main memory will result in a segmentation fault, and the same applies for when either the default segment table object (segment table object with ID 0) or one that hasn't had any segment entries added to it by instruction 103 is bound. Thread 0 is a special case where the entirety of main memory is just treated as a single segment with virtual addresses the same as physical addresses, i.e. all addresses for all main memory accesses are already in physical address space and all of main memory is accessible.

Registers

There are 16 64-bit registers. 11 of them are general-purpose; the last 5 serve unique purposes.
The 12th register is the Standard Output (aka the SO) register.
The Stack Pointer (aka the SP) is the 13th register.
The Status Register (aka the SR) is the 14th register.
The Link Register (aka the LR) is the 15th register.
The Program Counter (aka the PC) is the 16th register.

The SO may immediately output a single character to a terminal or elsewhere for each time the value of this register changes, interpreting each new value as an ASCII character code, although this should not be relied on since this feature is not at all required of an implementation and can simply be useful for testing purposes.
The SP is used to provide stack functionality through push/pop instructions.
The SR is program information/status and can be updated/written to by instructions.
The PC is incremented by 1 after each instruction is executed.
The LR is updated when a jump instruction is executed and all of its jump condition(s) are met (if it has any) to be equal to the PC + 1 (or equivalently, the memory address of jump instruction + 1).

There is a maximum of 3 registers accessible at once by any instruction. These are the primary and secondary registers, which are effectively used as the instruction's operands, and the output register which is the register that is used to get results from instructions. These are not actual registers, but simply references to individual registers (from the 16), and these references are updated by instructions dedicated to changing the primary/secondary/output registers.

Each thread has its own registers. Excepting the PC, all registers always have an initial value of 0. The PC for thread 0 is initialized to 0 but the PC for all created threads is set when they are created.

Instructions that update the SR do so before the end of the instruction's execution.

Before an instruction outputs any data to a register (usually to the output register) that register is first cleared with 0s.

The rightmost bit in a register is the least significant bit (LSB) and the leftmost bit in a register is the most significant bit (MSB).

Status Register

The Status Register (SR) looks like the following:
MSB 16 extra bits | 1 bit for whether or not there was a segmentation fault | 1 bit; conditional bit N | 1 bit; conditional bit Z | 1 bit; conditional bit C | 1 bit; conditional bit V | 16 bits for current file stream (0 represents no file stream) | 1 bit for whether or not the bound file stream is open (1 if it is) | 8 bits for current drive number (0 is the main drive) | 1 bit for whether or not there was a buffer map error (1 if there was) | 1 bit that gets set to 1 at the moment the parent thread of this thread is killed | 7 bits for file I/O errors (described in ***File Errors***) | 1 bit for whether or not there was a buffer allocation error (1 if there was) | 8 bits for the current display (described in ***Displays***) **LSB**

In addition to the defined cases where specific sections of this register are updated, note that all data in this register are mutable and can be manually modified at any point just like any other register. The bit specifying whether or not the current file stream is open is updated once at the end of each of the file I/O instructions, instructions 60-66, with the only exception being when the thread executing the instruction lacks file I/O permission.

Value Representations

Signed integers are in two's complement.
Floating-point values are in IEEE 754 format for both single-precision (32-bit) and double-precision (64-bit).

Note that, throughout the specification, when a number of 0 is said to correspond to 1, it is because that number is equal to the value it actually represents subtracted by 1 (typically just so that a value of 0 cannot be provided).

Comparisons

The SR has four conditional bits: N, Z, C, and V.
The N (Negative) bit is set when the right operand in a comparison is greater than the left operand.
The Z (Zero) bit is set when both operands in a comparison are equal.
The C (Carry) bit is set when the left operand an integer or floating-point comparison is greater than or equal to the right operand, or when any operand in a floating-point comparison is NaN.
The V (oVerflow) bit is set whenever the right operand (secondary register) subtracted from the left operand (primary register) in a 32-bit integer comparison would result in a signed integer outside the range of $[-2^{31},2^{31}-1]$, when that subtraction in a 64-bit integer comparison would result in a signed integer outside the range of $[-2^{63},2^{63}-1]$, or when any operand in a floating-point comparison is NaN.

Carry and zero bit is evaluated by assuming both the operands and result are unsigned integers or floating-point numbers. The overflow and negative bits are evaluated by assuming both the operands and result are signed integers or floating-point numbers.

In floating-point comparisons, if any operand is NaN, only the overflow and carry bits are set to 1. Positive infinity is greater than the other operand unless the other operand is equal, and negative infinity is lesser than the other operand unless the other operand is equal.

When a comparison instruction is run, all conditional bits which do not get set to 1 are set to 0.

Data Buffers

Data buffers are buffers that serve any of the multiple purposes which don't have their own (specialized) buffer object types. They can be used for specifying transformation matrices in building acceleration structures, for specifying indirect draw call parameters, and for providing push constant values. With limited possible use cases, data buffers are usually small enough to be stored in both RAM and in GPU memory, and which copy of the buffer the implementation should use can depend on its usage. For example, an implementation that supports indirect drawing could benefit from reading directly from RAM the indirect draw call information so that it can process and reformat the data in a hidden copy of the buffer to be compatible with however the implementation's graphics API expects indirect draw call parameters to be provided. In ray tracing, the implementation can use the GPU copy of the data buffer to provide transform matrix information for acceleration structure builds. Data buffers also serve as a useful separation from main

memory in command buffer submission; when submitting command buffers that update push constants or use indirect drawing, there is less work needed to check what memory can be accessed by the thread since dedicated data buffers stored outside of main memory are free of the minor performance costs of evaluating memory segmentation to check for segmentation faults.

Uniform Buffer Objects

Uniform buffer objects are buffers used in conjunction with descriptors in order to specify to shaders a collection of data for one or more shader uniforms. These may or may not be stored in GPU memory as implementations running on graphics APIs which do not support uniform buffers would need to instead send the correct portions of the UBO through several individual uniform-setting commands. The format/layout of a UBO is specified by that of the uniform block definition(s) it is accessed through. Floating-point numbers are provided as 4 bytes in IEEE 754 format, and integers are provided as 4 bytes in two's complement. The closer a uniform definition is to the first defined uniform's first (or only) element in a uniform block, the closer its data is to the beginning of the UBO. The result of a UBO not being large enough to provide data to the entirety of a shader uniform block it is accessed through is undefined. If a UBO is too large for a uniform block it's accessed through, the data that will be specified for the uniform block spans from the beginning of the UBO up to the last byte that is used by the uniform block, and the rest of the UBO's data is simply inaccessible. A uniform array appearing in a uniform block is laid out such that the first element (at array index 0) is closer to the beginning of the uniform block and each greater index moves further away from it, as a uniform array is really just composed of contiguous uniforms. Uniform buffers have a maximum size of 16 KB.

Storage Buffer Objects

Storage buffer objects are buffer objects used almost exactly like uniform buffer objects except for that they are accessed from shaders via storage descriptors (they follow the same rules regarding their memory layout), as well as that they may only be accessed by compute or ray tracing shaders and may be both read or written to by those shaders. These may or may not be stored in GPU memory as implementations running on graphics APIs without GPU computing support will need to emulate compute shaders on the CPU. SBOs have a maximum size of 128 MB.

Outdated GPU Limitations

Unfortunately, there are many limitations that exist on outdated GPUs that haven't been an issue on modern hardware for a long time, but because there are still so many devices using them, those limitations can't be ignored and can often get in the way when developing software. Often the decision is made to just not support outdated GPUs in software that uses newer features. These limitations will instead be mitigated by the Piculet runtime environment so that software with the newer features can still run, although the process of mitigation might cause some performance impact. If the runtime environment is running without hardware accelerated graphics, the limitations associated with outdated GPUs will not be present. Systems with outdated GPUs will always need to emulate compute shaders on the CPU as compute shaders are a newer feature in graphics APIs. Since compute shaders are run on the CPU on devices that don't support them, none of the measures used to mitigate the outdated GPU limitations will need to be applied when performing compute operations.

There is a bit specifying whether or not the GPU is outdated in ***Hardware Information***, and it is set if the runtime environment is running with hardware accelerated graphics and the GPU doesn't support integer attributes, unsigned integer types in shaders, uniform buffers, getting a texture's dimensions from within a vertex or pixel shader, texel sampling from within a vertex or pixel shader, texture sampling from specific LOD in pixel shaders, or texture sampling within vertex shaders. If the GPU doesn't support integer attributes, all int, uint, ivec, and uvec attributes in shaders will be automatically translated into their floating-point equivalents; when data is input to shaders, all integers in the data will first be typecast to floating-points without affecting the values stored in the vertex buffer (requiring a copy of the vertex buffer to be made). If the GPU doesn't support unsigned integers in shaders, all uint and uvec will be automatically translated into int and ivec types, and it is up to the author of the shader to make sure that there will not be signed integer overflow if outdated GPUs are to be supported by the software.

There is also a strict limitation on outdated GPUs as to how much uniform data (including push constant uniforms) can be defined, and any vertex/pixel shader containing more uniform components than the limit will be invalid on runtime environments with hardware accelerated graphics and an outdated GPU. There can be up to 512 individual uniform components within vertex shaders and 64 individual uniform components within pixel shaders (the sum of the number of array elements * number of components per element across all uniform arrays). Note that all defined uniforms in

shaders are considered arrays, even if there is only 1 element. Each element in a scalar-typed array (float, int, or uint) may take up to 4 uniform components each. Each element in a vector-typed array (vec, ivec, or uvec), regardless of the vector's size, may take up to 4 uniform components each. Each element in a matrix-typed array takes up to 4 times the minimum count of rows/columns of uniform components each.

Descriptors, Sets, and Layouts

A descriptor is a reference to a resource (data) that resides somewhere in memory (though outside of main and system memory), either the data for a texture, an image (a single level of some texture), a uniform buffer, an acceleration structure, or a storage buffer object's store. A descriptor set is a group of such descriptors, which can be of mixed types. Descriptor sets are configured outside of command buffers, where descriptors can be bound to the descriptor set at the descriptor bindings defined by the descriptor set layout provided during the set's creation. Two descriptor set layouts are considered identical if they have the same number of descriptor bindings, all the same descriptor binding point numbers, and each binding point has the same descriptor type and descriptor count between both descriptor set layouts, without any regard to the order in which they were specified in the array of descriptor set layout creation information.

Within a command buffer, a descriptor set can be bound to any of the command buffer's descriptor set bindings, and which ones can actually be accessed by shaders without having undefined results is determined by the bound pipeline's descriptor set layout for that descriptor set binding as well as the number of descriptor sets the pipeline has access to (also note pipeline creation fails if any of its shaders contain the usage of set bindings that are inaccessible to the pipeline). Accessing descriptor sets will have undefined results if, for any of the bound descriptor sets that the bound pipeline has access to, the descriptor set layout for the pipeline's set binding is not identical to the descriptor set's descriptor set layout, if any set bindings that are accessible by the pipeline do not have a valid descriptor set bound to them, or if, for any of the set bindings accessible by the pipeline, the descriptor set layout object bound for the set binding or for its bound descriptor set has been deleted. Accessing a descriptor set will have undefined results if any of its descriptors refer to an invalid or deleted object. Descriptor sets bound to set bindings will persist throughout the execution of a command buffer, and so descriptor set bindings can be reused across multiple pipelines within a command buffer without issue as long as the descriptor set layouts match properly. Descriptor sets can contain multiple sampler descriptors at sampler descriptor bindings, but descriptor sets containing more than 1 sampler descriptor at any binding can only be used with ray tracing pipelines. When a descriptor set is updated, the descriptors are copied into the descriptor set as they are at the time of the update; the descriptor objects used in a descriptor set update can therefore safely be deleted after the descriptor set has been updated, and modifying the descriptor objects (i.e., what they refer to) will have no effect on the descriptor set unless the descriptor set is updated again. Modifying a descriptor set that is being used by a command buffer that is executing at the time of the update will have undefined results.

Data from descriptors can be accessed through shaders through the configured set and binding numbers, and the locations of attribute inputs are independent of descriptor bindings. The maximum number of descriptor set bindings can be found under ***Hardware Information***. Sampler descriptors refer to a TBO that will be sampled from and include the properties of the texture's sampling with that sampler. An image descriptor refers to a single texture level of a TBO and can be used to read or modify textures from within compute and ray tracing shaders. An acceleration structure descriptor refers to a top-level acceleration structure used in ray tracing. A uniform descriptor refers to a UBO. A storage descriptor refers to an SBO.

Descriptor Set Layout Creation

When creating a descriptor set layout, the number of descriptor bindings in the descriptor set layout, the binding point numbers for each descriptor binding, the number of descriptors in each descriptor binding (for sampler descriptor binding points only), and the type of each descriptor binding in the descriptor set layout are all specified by memory provided to the instruction that creates the descriptor set layout object. The contents of that memory is described by this section of the specification.

The memory block for creating descriptor set layouts is as follows, listed from lowest to highest address:
32 bits for number of descriptor bindings (0 corresponds to 1) | contiguous groups of values giving descriptor information for each of the descriptor bindings; each group has 32 bits for descriptor binding point number, followed by 8 bits for the type of the descriptor binding (0=uniform descriptor, 1=storage descriptor, 2=sampler descriptor,

3=image descriptor, 4=acceleration structure descriptor), followed by 16 bits for the number of descriptors in the descriptor binding (only present for sampler descriptor binding points)

If the descriptor set layout contains any repeating descriptor binding point numbers, the number of descriptors at a sampler descriptor binding is 0, or any of the descriptor binding types are invalid, descriptor set layout creation will fail.

Command Buffers

Command buffers (CBOs) are objects into which commands can be recorded and which can then later be submitted to the GPU for execution, allowing you to add commands or reuse the same commands as often as necessary. The order in which command buffers are processed by each queue on the GPU may or may not be from first to last submitted, and queues will execute commands asynchronously with respect to each other. Commands within a command buffer may execute out of order, however the effects of all commands within a command buffer will occur and become visible to all following commands within the command buffer in the correct order. All changes to buffer memory by work in a queue is guaranteed to be visible once all command buffers in the queue have finished. Instruction 91 can be used to wait until all command buffers submitted by a thread have finished. Instructions which are defined to be recorded into a command buffer are added to the bound command buffer when they are encountered, and any directly-provided values that affect the command (such as number of vertices to draw) will be copied over with it. The IDs for the VBO, IBO, pipeline, and descriptor sets bound to each instance of a submitted command buffer are all 0 before any commands in the command buffer are executed. Resetting a command buffer will erase all commands previously recorded into it, reset the ID for the bound FBO, and also reset what type of pipeline can be bound to it.

Queue Submissions

In instructions that submit command buffers to a queue, the list of command buffers is provided by a memory address to a region specifying the IDs of the command buffers to submit and what queue to submit to. Instruction 89 is used to submit to a graphics queue, and instruction 90 is used to submit to a compute queue.

The memory block for command buffer lists is as follows, listed from lowest to highest address:
16 bits for the queue to submit to (note queue 0 is the first queue) | 32 bits for number of command buffers to submit (0 corresponds to 1) | for each command buffer to submit, 64 bits specifying the command buffer ID (submitted in order; i.e., lowest address ID will be the first submitted to the queue)

If submitting to a graphics or compute queue and the queue to submit to is > the number of graphics or compute queues available (see ***Hardware Information***), or if any of the IDs are not of a valid and compatible command buffer (only command buffers using rasterization pipelines can be submitted to graphics queues, and only command buffers using compute and ray tracing pipelines can be submitted to compute queues), the queue submission fails.

Indirect Drawing

Indirect draw call commands (recorded by instruction 93) allow many draw calls to be submitted from a single command using data buffer objects. The data buffer provides all parameters for draw calls, which are all contiguous groups of values read from the data buffer when the command is executed (each group specifying the parameters for a single draw call). When the command is recorded, some configuration for the indirect drawing command is set by memory addressed by the primary register.

The memory block for configuring the indirect drawing command is as follows, listed from lowest to highest address:
8 bits specifying whether or not the draw calls are indexed (0 if not, non-zero otherwise) | 64 bits specifying the ID of a data buffer object to read indirect draw call parameters from | 64 bits for the offset into the draw call parameters buffer (must be a multiple of 4) | 32 bits specifying the number of draw calls (0 corresponds to 1)

The parameters for each draw call within the buffer for indirect draw call parameters is laid out as follows:
32 bits specifying the count of vertices/indices to draw | 32 bits specifying the number of instances to draw (0 corresponds to 1) | 32 bits specifying the first index in VBO (for non-indexed draw calls) or in IBO (for indexed draw calls)

Indirect non-indexed draw calls will use the vertex buffer bound in the command buffer at the time of execution, and indirect indexed draw calls will use the vertex and index buffers bound in the command buffer at the time of execution.

If the data buffer for indirect draw call parameters had previously been deleted or any of the used VBOs or IBOs are invalid when the command is executed, results are undefined. If reading the draw call parameters exceeds the bounds of the data buffer, the results are undefined.

Push Constants

A special type of uniforms are defined to get their data from data buffers via updates in command buffers, as an alternative to using uniform buffer objects. Uniforms whose data is provided in this way are called push constants, and these uniforms are defined within the enclosures of push constant blocks in shaders. Push constant blocks must be defined in exactly the same way across all shaders in a pipeline (disregarding identifiers), and the uniforms defined within them are laid out in exactly the same way in data buffers as uniform block uniforms are laid out in UBOs. Push constant data is shared across all shaders in a pipeline, and only shaders accessing push constants need to define push constant blocks. There can only be one push constant block defined per shader. A pipeline can have up to 128 bytes of push constant data. All push constant data, at the beginning of the duration of a pipeline's binding within command buffers, are undefined until the data is set via a push constants update command (instruction 95). Each pipeline is created with a specified push constant range; this is the number of bytes of push constant data that the pipeline has.

Compute Shaders

Piculet supports compute shaders to allow code to be executed on the GPU, and they allow data output via storage buffers and writing texture data to images (an image is a single texture level of a texture pointed to by an image descriptor). If the support for GPU computing bit in the hardware information flags (see ***Hardware Information***) is 0, compute shaders can still be used and will be run on the CPU. Compute shaders define their local work-group size; the product of some specified dimensions; and this states the number of work-items per work-group, with each work-item being a shader invocation executed on its own thread with its own memory. Instruction 101 is used to dispatch work-groups, where the global work-group size is the product of the specified dimensions. The global work-group size is the number of work-groups dispatched for execution. Work-items within the same work-group can be executed in parallel and allow synchronization by setting a barrier within the shader. Work-groups can execute in parallel as well, but they cannot be synchronized.

Pipeline Objects

Pipeline objects contain rendering state for use by the commands in a command buffer which are not stored by the command buffer itself. There are three types of pipeline objects; rasterization pipelines, which are for rendering using rasterization, ray tracing pipelines, which are for rendering using ray tracing, and compute pipelines, which are used for GPU computing. Executing an instruction to record a command to a command buffer will have no effect unless it is a pipeline bind or a pipeline bind command has already been recorded into the command buffer at the time of recording.

Pipeline Creation

When creating a pipeline, the shaders and all of the immutable rendering states for that pipeline are specified by memory provided to the instruction that creates the pipeline object. The contents of that memory is described by this section of the specification.

The memory block for creating rasterization pipelines is as follows, listed from lowest to highest address:
64 bits for ID of vertex shader
| 64 bits for ID of pixel shader
| 64 bits for ID of a vertex array object
| 8 bits for culled vertex winding (0=no culling, 1=cw, 2=ccw, 3=cw+ccw)
| 8 bits for primitive assembly type (0=triangles, 1=lines, 2=points)
| 8 bits specifying the number of bytes of push constant data (the push constant range; must be a multiple of 4)
| 16 bits for the number of enabled descriptor sets (0-256)
| for each enabled descriptor set, 64 bits for ID of a descriptor set layout (the first is for set 0, the second for set 1, ...)
| 8 bits for depth test pass condition (0=always pass test, 1=never pass test, 2=if depth is less than stored depth, 3=if depth is less than or equal to stored depth, 4=if depth is greater than stored depth, 5=if depth is greater than or equal to stored depth, 6=if depth is equal to stored depth, 7=if depth is not equal to stored depth)
| 8 bits for whether or not to enable writing to depth buffer (0=enabled, 1=disabled)
| 8 bits for CW face stencil function reference value

| 8 bits for CW face stencil test function pass condition (0=always pass test, 1=never pass test, 2=if reference value is less than stored value, 3=if reference value is less than or equal to stored value, 4=if reference value is greater than stored value, 5=if reference value is greater than or equal to stored value, 6=if reference value is equal to stored value, 7=if reference value is not equal to stored value)
| 8 bits for CW face stencil operation if stencil test fails (how to affect stored value; 0=keeps the stored value, 1=sets the stored value to 0, 2=sets the stored value to the reference value, 3=increments stored value by 1 and clamps to 255, 4=decrements stored value by 1 and clamps to 0, 5=increments stored value by 1 and wraps to 0 if greater than 255, 6=decrements stored value by 1 and wraps to 255 if less than 0, 7=bitwise inversion of stored value)
| 8 bits for CW face stencil operation if stencil test passes but depth test fails (how to affect stored value; same values as previous)
| 8 bits for CW face stencil operation if both stencil and depth test fail (how to affect stored value; same values as previous)
| 8 bits for CW face stencil function mask (bitwise ANDed with the reference and stored value prior to the stencil function's evaluation)
| 8 bits for CW face stencil write mask (which bits of stored value can be written to; 1 represents a writable bit)
| 8 bits for CCW face stencil function reference value
| 8 bits for CCW face stencil function test pass condition (same as for CW face)
| 8 bits for CCW face stencil operation if stencil test fails (same as for CW face)
| 8 bits for CCW face stencil operation if stencil test passes but depth test fails (same as for CW face)
| 8 bits for CCW face stencil operation if both stencil and depth test fail (same as for CW face)
| 8 bits for CCW face stencil function mask (same as for CW face)
| 8 bits for CCW face stencil write mask (same as for CW face)
| 8 bits for color write mask (which components can be written to; writable components are represented by bits of 1 within the 4 bits on LSB end, with the rightmost bit being for A, the next for B, the next for G, and the leftmost of the 4 bits being for R; the last 4 bits on the MSB end are ignored)
| 8 bits for the number of enabled color attachments (up to 8); the 3 bits on the LSB end specify a value 0-7 (0 corresponds to 1) and the 5 bits on MSB end are ignored
| 8 bits for color blending operation (0=add, 1=subtract, 2=reverse subtract, 3=minimum, 4=maximum)
| 8 bits for color blending source factor (0=one, 1=zero, 2=src color, 3=dst color, 4=src alpha, 5=dst alpha, 6=1 subtract src color, 7=1 subtract dst color, 8=1 subtract src alpha, 9=1 subtract dst alpha)
| 8 bits for color blending destination factor (same values as color blending source factor)
| 8 bits for alpha blending operation (same values as color blending operation)
| 8 bits for alpha blending source factor (same values as color blending source factor)
| 8 bits for alpha blending destination factor (same values as color blending source factor)

The memory block for creating compute pipelines is as follows, listed from lowest to highest address:
64 bits for ID of a compute shader
| 8 bits specifying the number of bytes of push constant data (the push constant range; must be a multiple of 4)
| 16 bits for the number of enabled descriptor sets (0-256)
| for each enabled descriptor set, 64 bits for ID of a descriptor set layout (the first is for set 0, the second for set 1, ...)

The memory block for creating ray tracing pipelines is as follows, listed from lowest to highest address:
8 bits specifying the number of bytes of push constant data (the push constant range; must be a multiple of 4)
| 16 bits for the number of enabled descriptor sets (0-256)
| for each enabled descriptor set, 64 bits for ID of a descriptor set layout (the first is for set 0, the second for set 1, ...)
| 32 bits for maximum level of recursion (pipeline creation fails if this is greater than the maximum level of recursion in ***Hardware Information***)
| 32 bits for the number of shaders in this pipeline (if 0, pipeline creation fails)
| 64 bits for each of the ID(s) of each of the shaders in this pipeline, in any order (the first element is the shader with shader index 1, the second element is the shader with shader index 2, and so on)

If any of the values in the memory blocks are out of range, or any of the object IDs don't refer to a valid or existing object, pipeline creation will fail.

If the number of bytes for the push constant range is not a multiple of 4 or is greater than 128, pipeline creation will fail.

During pipeline creation a compiled version of a shader's code (from specified shader object buffers) is created for the pipeline; the source shader object can then be safely deleted. If the shader is deemed to be invalid when judged under all the criteria for a shader's validity (see ***Shader Bytecode***), the shader type is not compatible with the type of pipeline, or if the shader object's buffer is currently mapped, pipeline creation will fail.

If a set binding number is used in one of the pipeline's shaders which is greater than the value given here for the number of descriptor sets accessible by the pipeline - 1, pipeline creation will fail. If any set and binding number pairs defined within any of the pipeline's shaders is not defined as a type compatible with the descriptor type for that set and binding number in the pipeline's descriptor set layouts, pipeline creation will fail. If any sampler definitions in one of the pipeline's shaders are not unsized (i.e. are not defined with an element count of 0) and do not have the same number of descriptors as the corresponding descriptor binding point in the pipeline's descriptor set layouts, pipeline creation will fail.

If a push constant block defined in any of the pipeline's shaders has a size not equal to the push constant range, pipeline creation will fail. If there are differently defined push constant blocks across all of the pipeline's shaders (disregarding identifiers; just considering element counts, data types, and the order of uniform definitions), pipeline creation will fail.

If the number of enabled descriptor sets is greater than the maximum number of descriptor set bindings (see ***Hardware Information***), pipeline creation will fail.

If the total number of uniform descriptors summed across all of the pipeline's descriptor set layouts is greater than the maximum number of accessible uniform descriptors (see ***Hardware Information***), pipeline creation will fail.

If the total number of sampler descriptors summed across all of the pipeline's descriptor set layouts is greater than the maximum number of accessible sampler descriptors (see ***Hardware Information***), pipeline creation will fail; for binding points with more than one sampler descriptor, each descriptor at the binding point counts as 1 sampler descriptor.

If the total number of storage descriptors summed across all of the pipeline's descriptor set layouts is greater than the maximum number of accessible storage descriptors (see ***Hardware Information***), pipeline creation will fail.

If the total number of image descriptors summed across all of the pipeline's descriptor set layouts is greater than the maximum number of accessible image descriptors (see ***Hardware Information***), pipeline creation will fail.

If the total number of acceleration structure descriptors summed across all of the pipeline's descriptor set layouts is greater than the maximum number of accessible acceleration structure descriptors (see ***Hardware Information***), pipeline creation will fail.

Creating a compute or rasterization pipeline will fail when any of the pipeline's descriptor set layouts contain an acceleration structure descriptor binding, or if there are any sampler descriptor bindings in the pipeline's descriptor set layouts which contain more than one descriptor. Creating a rasterization pipeline will fail when any of the pipeline's descriptor set layouts contain any image or storage descriptors or when any output attributes defined in the vertex shader aren't also defined as input attributes under the same identifier and type in the pixel shader.

Shader Compilation

An implementation of Piculet will automatically compile shaders when a pipeline using them is created, and compiled versions of shaders should be reused for new pipelines when they've been used in pipeline creation before but their shader object hasn't been changed (if the buffer was mapped and unmapped since the last compilation then it can be assumed to have been modified).

Vertex Array Object Creation

Vertex array objects describe the layout of vertex attributes found in vertex buffers. When creating a vertex array object, the size of each vertex, the number of vertex attributes, the vertex attribute IDs (which correspond to attribute location numbers in shaders), the offsets for first occurrence of each vertex attribute, and the format for each vertex attribute is specified by memory provided to the instruction that creates the vertex array object.

The memory block for creating vertex array objects is as follows, listed from lowest to highest address:
16 bits for number of vertex attributes (0 corresponds to 1) | 64 bits for size of each vertex (i.e., the stride, in bytes, between each attribute; must be a multiple of 4) | contiguous groups of values giving information of each vertex attribute; each group has 16 bits for attribute ID, followed by 64 bits for the vertex attribute's offset (must be a multiple of 4), followed by 8 bits specifying the format of the vertex attribute (one of the values listed under ***Vertex Formats***)

If any of the vertex formats do not exist, there are any repeating attribute IDs, if the size of each vertex is 0 or not a multiple of 4, if any attribute's offset is not a multiple of 4, or if any attribute's offset summed with the size of its data (determined from the attribute's format) is greater than the stride between vertices, vertex array object creation will fail.

Blending

Blending is the step after the execution of a pixel shader outputs a pixel's RGBA values, where the pixel shader's output (source) components are blended with the components of the pixel already at the location in the color attachment that the pixel is going to (the destination). If the destination color buffer is fixed-point, the source and destination components are clamped to range [0,1] before evaluating any of the blending equation, and then the components that result are converted back into fixed-point integers. The source and destination blending factors and the blending operation are rasterization pipeline state.

sc is the vector of RGBA components from the pixel shader.
dc is the vector of RGBA components in the color attachment.
sfac is the source blending factor.
dfac is the destination blending factor.
op is the blending operation.

The blending equation is:
C = sc*sfac op dc*dfac

There are separate blending factors and operations used for computing the RGB components and the alpha component, so the RGB and A parts of the final RGBA can be calculated differently depending on how the rasterization pipeline is configured.

Factors (which can be set as the source and destination factors for either color or alpha blending) include:
one - the factor is 1 (1,1,1,1)
zero - the factor is 0 (0,0,0,0)
src color - the factor is the components from the pixel shader (R_s,G_s,B_s,A_s)
dst color - the factor is the components in the color attachment (R_d,G_d,B_d,A_d)
src alpha - the factor is the alpha component from the pixel shader (A_s,A_s,A_s,A_s)
dst alpha - the factor is the alpha component in the color attachment (A_d,A_d,A_d,A_d)
1 subtract src color - the factor is from 1 subtract the components from the pixel shader (1-R_s,1-G_s,1-B_s,1-A_s)
1 subtract dst color - the factor is from 1 subtract the components in the color attachment (1-R_d,1-G_d,1-B_d,1-A_d)
1 subtract src alpha - the factor is from 1 subtract the alpha component from the pixel shader (1-A_s,1-A_s,1-A_s,1-A_s)
1 subtract dst alpha - the factor is from 1 subtract the alpha component from the pixel shader (1-A_d,1-A_d,1-A_d,1-A_d)

The blending operations that can be performed include:
add - add the two source colors together
subtract - subtract source color 1 from source color 2
reverse subtract - subtract source color 2 from source color 1
minimum - take the minimum of source color 1 and source color 2
maximum - take the maximum of source color 1 and source color 2
Where source color 1 is the product of sc*sfac, and source color 2 is the product of dc*dfac.

Coordinate System

The coordinate system used for audio occlusion geometry (including their vertex positions and position/rotation transformations) and device position and orientation data is a right-handed coordinate system. For a viewer facing forward at the origin, -Z is pointing forward, +Y is pointing up, and +X is pointing to the right. Rotations are

counterclockwise about these axes, so e.g. if a cube was sitting centered at the origin along with the viewer at its center, from the perspective of the forward-facing viewer a +X angle rotation would move the top side backwards, a +Y angle rotation would move the left side backwards, and a +Z angle rotation would move the top side to the left.

Objects

Instruction 72 is used to create various types of objects which all serve different purposes. Objects are referred to by IDs, and instruction 72 will output the ID of the new object. Objects can be bound using instruction 74 (see ***Object Bindings***). To clear an object's binding is to set that binding to the default object, ID 0. The default object is not considered to be a valid/existing object and cannot be used or bound in instructions, unless it is one of the few instructions defined to allow the use of ID 0 (see below).

All object generations by instruction 72 will produce an ID one greater than the last, starting from 1, regardless of by which thread it was generated or what object type the ID was generated for, so object type can be determined from nothing but the ID and a record of all previous object generations.

If, for any instruction, the ID of an object specified (either directly in a register or in a region of memory addressed by a register) was not previously generated or it had previously been deleted, it is invalid/non-existent and the instruction will do nothing, and the same goes for instructions which try to use or modify a bound object for which there is no valid/existing object bound. If, for any instruction, the ID of an object specified was an object generated with a privacy key not matching that of the thread the instruction is being executed on, nothing will happen (see ***Thread Permissions and Memory Restrictions***). Exceptions to these rules are made for specifying ID 0 as segment table object when creating a thread with instruction 37, for binding object ID 0 with instruction 74, for binding ID 0 as FBO with instruction 75, for setting the SR's allocation error bit to 1 if it's instruction 81, for setting the SR's map error bit to 1 if it's instruction 82, for getting the byte count of the default object (ID 0) with instruction 80, for when binding TBO with ID 0 for a framebuffer attachment with instruction 85, for when the index buffer or data buffer object IDs are 0 in geometry descriptions provided when building a BLAS with instruction 87 (see ***Geometry Descriptions***), for when the audio occlusion geometry in a listener's binding point is being set to the bound audio occlusion geometry by instruction 121 and the ID for the bound audio occlusion geometry is 0, and for when the audio data for an audio source is being set to the specified audio data object by instruction 121 and the ID specified is 0.

Each object is given an immutable copy of the creating thread's privacy key at the time of object generation and will automatically be deleted after a thread is killed if no living threads have a matching privacy key. Thread 0 is the only thread that is allowed to update its own privacy key.

Object Bindings

Each thread has the following states related to which objects are currently bound, each being 64-bit object IDs:
Bound uniform descriptor, bound storage descriptor, bound sampler descriptor, bound image descriptor, bound acceleration structure descriptor, bound descriptor set, bound descriptor set layout, bound vertex array object (VAO), bound vertex buffer (VBO), bound index buffer object (IBO), bound texture buffer object (TBO), bound command buffer object (CBO), bound uniform buffer object (UBO), bound storage buffer object (SBO), bound top-level acceleration structure (TLAS), bound bottom-level acceleration structure (BLAS), bound data buffer object, bound shader binding table (SBT), bound shader object, bound pipeline object, bound framebuffer object (FBO), bound audio data object, bound audio source object, bound audio listener object, bound audio occlusion geometry object, bound video data object, and bound segment table object.

All bits for all of the object bindings are initially 0, therefore all object bindings are initially the default object.

Graphics

Instructions 75-101 are mostly related to graphics/GPU programming.
Instructions 77, 79, 87, 94, 95, 96, and 97 are the only command-recording instructions that aren't ignored for command buffers using ray tracing pipelines.
Instructions 77, 79, 94, 95 and 101 are the only command-recording instructions that aren't ignored for command buffers using compute pipelines.
Instructions 87, 96, 97, and 101 are the only command-recording instructions that are ignored for command buffers using rasterization pipelines.

All instructions that will record a command into the bound command buffer are ignored when no command buffer is bound. All command-recording instructions other than the one that record a pipeline binding command are ignored when a bound command buffer currently has no pipeline binding commands recorded to it. When a command-recording instruction isn't ignored because of these rules and no other conditions causing the instruction to do nothing is encountered, the instruction will have no immediate effect other than to record the command into the command buffer.

In normalized device coordinates, top left is (-1,-1), bottom right is (1,1), closest is Z=0, farthest is Z=1. In texture coordinates, (0,0) is the top left and (1,1) is the bottom right. The U coordinate of a texture is the coordinate for horizontal position, and the V coordinate of a texture is the coordinate for vertical position.

The view volume for clipping clip space coordinates is as follows:
$-W \leq X \leq W$
$-W \leq Y \leq W$
$0 \leq Z \leq W$

Depth outputs are always in the range [0,1]. Stencil outputs are not available for modification in pixel shaders but are instead statically set in the rasterization pipeline; rasterization pipelines can define the way stenciling is performed for both CW and CCW faces. Depth is only written to the depth buffer for a pixel if a depth attachment is present, depth writing is enabled, and the pixel passed the depth test.

Vertex array objects store vertex attribute positions, types, and other data related to their layout for use with vertex buffers (though vertex array objects are not linked to any specific vertex buffer objects; they just provide a description of the layout of vertex attributes). If using a rasterization pipeline that references a vertex array object that has been deleted, results are undefined.

In index buffers, each of the indices are 32-bit unsigned integers.

A shader object stores the source code for a shader, in the form of bytecode fully described in ***Shader Bytecode***.
A pipeline stores one or more compiled shaders; the compiled versions of the source code in shader objects as they were when specified at the time the pipeline was created (see ***Shader Compilation***).

The GPU usually has numerous queues to which command buffers can be submitted, and submitted commands will execute in the fashion described in ***Command Buffers***. Timing, multithreading and concurrent read/write of the graphics memory has to be managed manually, so there is an instruction to pause a thread until all command buffers it had previously submitted have finished (instruction 91).

A vertex attribute is a piece of data input per-vertex to a vertex shader which was read from a VBO using the information of attribute layout and stride from the VAO specified at the time of the rasterization pipeline's creation (the first byte of each vertex attribute is located within the bound VBO at the attribute's offset + stride between attributes * the index of the current vertex). The type of each vertex attribute is one of those described in ***Vertex Formats***.

Texture filtering modes are applied when sampling from textures. The minification filter is applied whenever sampling determines that the texture should be minified (more than one texel maps to a single pixel) and the magnification filter is applied whenever sampling determines the texture should be magnified (more than one pixel maps to a single texel). Each sampler stores its own filtering modes as well as a reference to which texture they sample from.

There is one default framebuffer (FBO 0) per display; this is the displayed image. The runtime environment will not perform anti-aliasing itself, so this has to be done manually in shaders if desired. If the default framebuffer is single-buffered, swapping buffers will do nothing and requests to do so will be silently ignored. The default framebuffer consists of a stencil buffer, a depth buffer, and a fixed-point RGBA color buffer at color attachment 0 (if multi-buffered, buffer swapping via instruction 98 will cycle the default FBO's color attachment 0 between front + back buffers).

The FBO bound to a command buffer using rasterization pipelines is the one whose attachments will be written to by the outputs of pixel shaders during rendering.

Texture filtering is not guaranteed to be supported; if it's not supported, sampling will still work but no filtering will be applied.

For an FBO bound to a command buffer using a rasterization pipeline, existing and valid TBOs must be bound to all color attachments enabled or else any command buffer using that FBO will be skipped in queue submissions. The result of submitting a command buffer using a rasterization pipeline and modifying its bound FBO while the command buffer is still being executed is undefined. Command buffers using ray tracing or compute pipelines cannot access FBOs.

Shader instructions with set and binding numbers refer to descriptor-provided data (uniform buffers, storage buffers, samplers, images, and acceleration structures). Location IDs refer to other input data or locations to output data.

Graphics States

There are many states that influence graphics processing, many of which are in pipelines and command buffers.

Each instance of a submitted command buffer stores their own copies of the following states for their use:
64 bits for the bound FBO, 64 bits for bound pipeline (can be either a ray tracing, compute, or rasterization pipeline), 64 bits for bound VBO, 64 bits for bound IBO, and references to the descriptor set objects bound for each descriptor set binding.
Additionally, each command buffer stores all of the commands recorded to them and all of the data provided for them at the time of recording (e.g. clear color values or object Ids). The only binding states which are set outside of command buffers but cannot be modified within them is the bound FBO.

In addition to all of the rendering states specified during pipeline creation (see ***Pipeline Creation***), each pipeline object stores compiled versions of their shaders, their push constant data (always undefined until a push constants update; see ***Push Constants***), references to the descriptor set layout objects provided for each accessible descriptor set binding, a reference to a vertex array object describing vertex attribute layout information (if it's a rasterization pipeline), and any information regarding the shaders that may be useful for the implementation.

Sampler descriptor objects store (in addition to a reference/binding to a texture object) U and V coordinate wrap modes, and minification and magnification filter modes (the values that these can be is specified in instruction 100, and by default these states are all 0).

Ray Tracing

Ray tracing in Piculet involves the use of several types of objects specialized for ray tracing. There are top-level acceleration structures (TLAS), bottom-level acceleration structures (BLAS), shader binding tables, ray generation shaders, any-hit shaders, closest-hit shaders, and miss shaders. Data buffers are also used for providing transformation matrices in BLAS builds. Acceleration structures contain potentially optimized forms of the geometry being ray traced; top-level acceleration structures contain a collection of optionally transformed bottom-level acceleration structures, which themselves contain optionally transformed geometry. When a bottom-level acceleration structure is modified, any top-level acceleration structure that contains it must be rebuilt or updated before use, as the top-level acceleration structure will likely no longer work. Transformations during the building of acceleration structures are done by using data buffer objects, which contain 12 32-bit floating-points describing 4x3 matrices where the layout is the same as the matrix elements description given under ***Shader Bytecode*** (column-major order). Building of top-level or bottom-level acceleration structures is done via a command which is recorded into command buffers. Geometry is submitted to a bottom-level acceleration structure in the form of geometry descriptions stored in main memory (see ***Geometry Descriptions***). Instances of bottom-level acceleration structures (called geometry instances) are submitted to top-level acceleration structures by providing a location in main memory describing each of the geometry instances; including, among other things, the IDs of each bottom-level acceleration structure (see ***Geometry Instance Descriptions***). Ray tracing shaders include the ray generation, any-hit, closest-hit, and miss shaders.

Ray generation shaders are run for each pixel of a region being ray traced for, any-hit shaders are run for all non-opaque intersections with geometry, and closest-hit is run for the closest intersection with geometry (the accepted intersection which was closest to the ray origin at the end of all ray intersection tests), and if the ray did not have any accepted intersections with geometry the miss shader is run instead.

Geometry is either marked opaque or non-opaque as it is submitted to a bottom-level acceleration structure, and as a bottom-level acceleration structure is submitted to a top-level acceleration structure the opaqueness of all geometries within the instance of the bottom-level acceleration structure can be overridden. The opaqueness of geometries can also be overridden by per-ray flags. For more information, see ***Visibility Modifiers***. For geometry to be intersected by a ray and execute the closest-hit shader for that intersection point, the intersection must have been accepted and must be the smallest distance from the ray origin out of all other intersections that were accepted. A non-opaque geometry intersection is accepted if it was not ignored by the any-hit shader, and an opaque geometry intersection is automatically accepted. If an intersection is non-opaque but there is no any-hit shader, an empty any-hit shader is called; the intersection is accepted in this case since an empty any-hit shader doesn't state to ignore ray intersection.

Searching for and testing intersections along a ray is not done in any specific order, and the closest hit (smallest distance from ray origin to an accepted intersection) is tracked until all intersections along the ray have been tested or until the ray is terminated. Then, the closest-hit shader is run for the closest hit, or the miss shader is run if there were no accepted intersections.

For each ray tracing operation a maximum recursion depth is specified (note this is limited by the maximum recursion depth of the ray tracing pipeline, which is limited by the maximum recursion depth of the implementation found in ***Hardware Information***); this is the maximum number of times that a ray can bounce. For example, the primary ray may produce several secondary rays, each considered to be at depth level 1, then each of them may produce several tertiary rays, each considered to be at depth level 2, and so on and so forth. The maximum recursion depth specified by the trace instruction (96) is how many of these depth levels you can have; as an example, if the recursion depth is 0, the primary ray wouldn't be allowed to trace any secondary rays.

If the support for ray tracing bit in the hardware information flags (see ***Hardware Information***) is 0, objects used in ray tracing can still be created but will not have any contents, and command buffers using ray tracing pipelines will simply not be submitted.

Ray Payloads

A ray payload is data accumulated for a ray as it is being traced; the parent shader (the one that called the trace function to start tracing that ray) defines a ray payload at a location ID with the ray attributes enclosed within a ray payload block. The shaders which are called during the trace can then have the option to define the incoming ray payload, at the same location ID given as ray payload to the trace function in the parent shader and redefining all of the ray attributes as defined for the ray payload at that location in the parent, each ray attribute being defined at the same position in the payload block and using the same types, array sizes, and identifiers (otherwise results are undefined) - this is shared storage between the shader that established a new ray and the new ray's shader invocations. Ray payloads can contain up to 1 KB of data. Ray payloads and incoming ray payloads can be both read and written to.

Geometry Descriptions

When submitting geometries to a bottom-level acceleration structure, each geometry is described in contiguous groups of values comprising the descriptions for each submitted geometry. The first geometry description in the array has a geometry index of 0, and the last geometry has a geometry index equal to the number of geometries - 1; geometry indices are used in the calculation of hit group SBT records (see ***Shader Binding Tables***).

Each geometry description is laid out in memory, from lowest to highest address, as follows:
8-bit unsigned integer specifying the geometry's flags (see ***Visibility Modifiers***) | 64-bit unsigned integer specifying a vertex buffer ID | 64-bit unsigned integer specifying a vertex buffer offset (must be a multiple of 4) | 64-bit unsigned integer specifying an optional index buffer ID (if this is 0, submitting this geometry to the BLAS will be done without using an index buffer) | 64-bit unsigned integer specifying an index buffer offset (must be a multiple of 4) | 64-bit unsigned integer for number of vertices/indices in the geometry | 64-bit unsigned integer for stride between each vertex position (must be a multiple of 4) | 8-bit unsigned integer for vertex format (see ***Vertex Formats***; this must be either 1 or 2) | 64-bit unsigned integer specifying an optional data buffer object ID (if this is 0, no transformation will occur for this geometry) | 64-bit unsigned integer specifying an offset into the data buffer object for this geometry's transformation matrix (must be a multiple of 16)

Geometry Instance Descriptions

When submitting geometry instances to a top-level acceleration structure, each geometry instance is described in an array of values comprising the descriptions for each submitted geometry instance.

Each geometry instance description is laid out in memory, from lowest to highest address, as follows:
64-bit unsigned integer specifying BLAS object ID | 32-bit unsigned integer specifying cull mask and custom index (8 bits at the MSB end are the geometry instance's cull mask and the 24 bits at the LSB end specify a custom index for the geometry instance) | 32-bit unsigned integer specifying geometry instance flags and SBT offset (8 bits at the MSB end are the geometry instance's flags and the 24 bits at the LSB end are the SBT record offset for the geometry instance) | 12 32-bit floating-point values specifying a 4x3 transformation matrix for the geometry instance (with column-major ordering, as described in ***Shader Bytecode***)

Geometry instance flags and cull masks are described in ***Visibility Modifiers***.

Shader Binding Tables

Shader binding tables (SBTs) are buffers which store the 64-bit indices of shaders in the current pipeline that can be used during ray tracing operations. Ray traced scenes may involve the use of hundreds of shaders. There are three types of shader binding tables supplied to a ray trace command; the ray generation SBT, the hit group SBT, and the SBT for miss shaders. Ray generation and miss SBTs consist of contiguous 64-bit shader indices, and hit group SBTs consist of contiguous hit groups. Each hit group in hit group SBTs consist of 2 64-bit shader indices; one for the any-hit shader, followed by one for the closest-hit shader. An SBT record refers to a single hit group, miss shader index, or ray generation shader index in an SBT. The shader indices specifying what shaders to use are extracted from the SBTs when ray tracing. When using a shader index that does not exist in the pipeline, using shader index 0 for anything other than an any-hit shader, or when using a shader index that is of a shader that does not match the type of shader required (e.g. shader index of a miss shader is taken from hit group SBT for the closest-hit shader), the results are undefined.

A miss SBT record index is supplied when tracing a ray from within a shader and in the command to begin a ray tracing operation. The address of the miss shader index to use, with the values supplied by the in-shader trace ray function that created the current ray italicized, is calculated as miss SBT buffer address + 8 * *miss SBT record index*. Note the miss SBT buffer address is the address of the SBT but with the miss SBT offset summed with it, as specified to the instruction that started the ray tracing operation (instruction 96).

A hit group SBT record offset and stride is supplied when tracing a ray from within a shader and in the command to begin a ray tracing operation. The address of the hit group to use, with the values supplied by the in-shader trace ray function that created the current ray italicized and the geometry index of the intersected geometry within the geometry instance bolded, is calculated as hit group SBT buffer address + 16 * (instance SBT record offset + **geometry index** * *SBT record stride* + *SBT record offset*). Note the hit group SBT buffer address is the address of the SBT but with the hit group SBT offset summed with it, as specified to the instruction that started the ray tracing operation (instruction 96).

Instance SBT record offset is the record offset for the geometry instance being intersected, and SBT record offset & stride is provided in the shader function that establishes the tracing of a new ray.

Visibility Modifiers

In ray tracing, there are multiple configurations influencing whether or not a ray hits a specific geometry. Each geometry submitted to a BLAS has opaqueness flags, each geometry (BLAS) instance submitted to a TLAS has flags to override the opaqueness of all geometry within the BLAS, and each ray traced has ray flags which can further modify how the ray will intersect with the geometry in a scene. Additionally, an entire geometry instance will be culled (ignored in ray intersections) if the cull mask of the ray and the geometry instance's cull mask are bitwise ANDed together and the result is zero; a cull mask gets specified by the in-shader trace function.

For all of the flag bits listed below, the option is enabled if the bit is 1.

These are the geometry flag bits for submitting a geometry to a BLAS, found in ***Geometry Descriptions***:
MSB 6 extra bits | Geometry is opaque | No duplicate any-hit invocation (limits to one call of any-hit shader per ray-primitive intersection throughout a ray tracing operation) **LSB**

These are the geometry instance flag bits for submitting a geometry (BLAS) instance to a TLAS, found in ***Geometry Instance Descriptions***:
MSB 28 extra bits | Disable front/back face culling (0=face culling enabled, 1=face culling disabled) | Front face of triangle for culling is the face whose vertices appear CCW from the ray origin (determined in object space, therefore this is not affected by the optional geometry instance transformation for the intersected geometry, but the geometry transformation provided during BLAS creation does have an effect; bit is 0 if the front face is CW and 1 if front face is CCW) | All geometry is opaque (overrides geometry flags) | All geometry is non-opaque (overrides geometry flags) **LSB**
Obviously, the last 2 flags are mutually exclusive and having both enabled may have undefined results.

These are the ray flag bits that are provided for each ray:
MSB 25 extra bits | Force all intersected geometry to be considered opaque | Force all intersected geometry to be considered non-opaque | Terminate ray after first hit, ignoring non-opaque intersections that get ignored by an any-hit shader (hit will still be processed through the closest-hit shader, unless combined with skip closest-hit shader flag) | Skip closest-hit shader | Cull back facing triangles | Cull front facing triangles | Cull all primitives considered opaque by the geometry + geometry instance flags | Cull all primitives considered non-opaque by the geometry + geometry instance flags **LSB**

Due to mutual exclusivities, no more than one flag should be enabled between the first 2 flags, and no more than one flag should be enabled between the last 2 flags. The cull back facing and cull front facing flags are also mutually exclusive to each other. Violating any of these mutual exclusivities may have undefined results.

Displays

Piculet may support up to 256 displays. There is always at least one display or else the runtime environment will terminate. Hardware information stores the dimensions of each display, cursor coordinates (as measured, in pixels, relative to the center of display 0), the touch coordinates for each display (including supported and current touch count), and the number of displays. A display's dimensions are mutable, and there can be a different number of supported touch coordinates for each display.

Which display is currently selected is stored in an 8-bit value in the SR, and displays are numbered consecutively starting at 0. If a display is added, its display number will be one higher than the highest display number. If a display is removed, all of the displays will remain in the same order but the ones with a greater display number will shift back by 1 to reoccupy the deleted display's number. If the value in the SR stating the current display is not an existing display, the thread will behave as if display 0 is selected.

There is a unique default FBO for each display (FBO 0), and the selected display value in the SR determines what default FBO an FBO ID of 0 refers to. If a display is removed while its default FBO is still being drawn to, results are undefined. For any command buffers using rasterization pipelines that is submitted to the GPU with the default FBO bound, the default FBO used will be the one that corresponds to whatever display was selected as the current display at the time that the command buffer submission occurred.

Cursor coordinates are relative to the center of display 0, and if the dimensions of display 0 changes, all cursor coordinates will be reset to (0,0).

Hardware Information

Stored in a memory block in system memory is the following information, listed from lowest to highest address:
32 bits: hardware support information | 8 bits: current number of displays (0 corresponds to 1) | 64 bits: memory address to the dimensions of each display (contiguous pairs of values, each pair is a 32-bit unsigned integer specifying width followed by a 32-bit unsigned integer specifying height) | 64 bits: memory address to 16-bit unsigned integers specifying the supported touch count for each display | 64 bits: memory address to 16-bit unsigned integers specifying current touch count for each display | 64 bits: memory address to 32-bit signed integer touch coordinates | 32 bits: current number of cursors (0 if no mouse is connected) | 64 bits: memory address to 32-bit signed integer cursor coordinates | 64 bits: memory address to additional mouse inputs for each cursor | 8 bits: number of connected keyboards | 64 bits: memory address to 72-bit memory block(s) for pressed keys of each connected keyboard | 8 bits: current number of connected controllers | 8 bits: current number of controller buttons | 8 bits: current number of controller axes | 8 bits: current number of controller positions | 8 bits: current number of controller orientations | 64

bits: memory address to controller information | 8 bits: current number of microphones | 8 bits: current number of available cameras | 64 bits: memory address of camera image dimensions | 32 bits: 32-bit unsigned integer of the minimum clock speed (in MHz) for the CPU (0 if unknown) | 32 bits: 32-bit unsigned integer of the maximum clock speed (in MHz) for the CPU (0 if unknown) | 64 bits: memory address to the current clock speed (in MHz); contiguous 32-bit unsigned integers for each of the CPU cores (0 if unknown) | 16 bits: total number of CPU cores (0 if unknown) | 64 bits: main memory capacity, in bytes | 64 bits: GPU memory capacity, in bytes (0 if unknown) | 64 bits: GPU memory available, in bytes (0 if unknown) | 32 bits: 32-bit floating-point battery percent (0-100; 100 if battery information is unavailable or if the battery is full) | 16 bits: number of graphics queues available (0 corresponds to 1) | 16 bits: number of compute queues available (0 corresponds to 1) | 32 bits: maximum number of descriptor binding points per descriptor set (at least 1024) | 32 bits: maximum number of binding points for audio occlusion geometry per audio listener (at least 16) | 32 bits: maximum texture size (for both width and height; at least 64) | 32 bits: max local work-group size X (minimum of 1024) | 32 bits: max local work-group size Y (minimum of 1024) | 32 bits: max local work-group size Z (minimum of 64) | 32 bits: max local work-group size (minimum of 1024) | 32 bits: max global work-group size X (minimum of 65535) | 32 bits: max global work-group size Y (minimum of 65535) | 32 bits: max global work-group size Z (minimum of 65535) | 32 bits: maximum recursion depth for ray tracing | 32 bits: maximum count of geometries in BLAS | 32 bits: maximum count of geometry instances in TLAS | 32 bits: maximum total count of triangles in TLAS | 8 bits: maximum number of descriptor set bindings in a pipeline (0 corresponds to 1; there is a minimum of 4 supported descriptor set bindings in a pipeline so this value's minimum is 3) | 64 bits: maximum number of accessible acceleration structure descriptors in a pipeline (at least 16) | 64 bits: maximum number of accessible sampler descriptors in a pipeline (at least 7) | 64 bits: maximum number of accessible image descriptors in a pipeline (at least 8) | 64 bits: maximum number of accessible uniform descriptors in a pipeline (at least 8) | 64 bits: maximum number of accessible storage descriptors in a pipeline (at least 4) | 64 bits: address to a null-terminated ASCII string listing supported audio file formats | 64 bits: address to a null-terminated ASCII string listing supported video/image file formats | 16 bits: maximum number of audio channels
The hardware support information looks like this (bit is 1 if true):
MSB 22 extra bits | GPU is outdated | texture filtering support | using hardware accelerated graphics | audio support | 3D audio support | audio occlusion support | sound propagation effects support | video support | ray tracing support | GPU computing support **LSB**

The hardware information is stored in system memory and is read-only.
All addresses provided by hardware information refer to read-only regions of system memory where additional information is recorded.

All bits are initialized to their correct values and continuously updated as soon as a change can be detected.
If CPU information is unavailable, the number of CPU cores and address to maximum MHz of cores are both 0.
If touchscreens are not supported or none of the available displays are touchscreens, the address to the count of supported touches for each display, the address to the current touch count for each display, and the address to touch coordinates are all 0.
If no keyboards are connected, the current number of keyboards and address to key presses are both 0.
If no mice are connected, the current number of cursors, address to cursor coordinates, and address to additional mouse inputs are all 0.
If no controllers are connected, the number of connected controllers, number of controller buttons, number of controller axes, number of controller positions, number of controller orientations, and address of controller information are all 0.
If audio is not supported, the address to the string of supported audio file formats and the maximum number of audio channels are both 0.
If video is not supported, the address to the string of supported video/image file formats is 0.
If audio is supported, the string at the address to supported audio file formats must list one or more formats and the maximum number of audio channels is at least 1.
If video is supported, the string at the address to supported video/image file formats must list one or more formats.
If no cameras are available, the number of available cameras and the address to camera image dimensions are both 0.
If ray tracing is not supported, the maximum recursion depth, maximum count of geometries in a BLAS, maximum count of geometry instances in a TLAS, maximum total count of triangles in a TLAS, and the maximum count of TLAS in a descriptor set are all 0.

Mouse and Touch Information

Defined in ***Hardware Information*** are the states related to the positions of the cursors and touchscreen inputs.

The cursor coordinates is an array of signed integers representing contiguous pairs of (x,y) coordinates of the cursor(s) as measured in pixels, which begins at the center of display 0 (at their origin), and increase/decrease according to mouse movement and direction; one pair is allocated for each connected cursor. The system's cursor(s) are constantly centered on display 0's window in order to not fall off the edge, and the virtual machine should hide the system's cursor(s) when within the runtime environment's window. Cursor coordinates will underflow/overflow when going outside of the range of the 32-bit signed integer's minimum/maximum.

The touch coordinates is an array of signed integer representing (x,y) pixel coordinates for each display. For each display, there is a group of coordinate pairs for each supported touch (if a display does not support any touches, the display will not have a group of touch coordinates). These groups of pairs are contiguous in memory and the groups of pairs are arranged consecutively for each display, starting with the group for display 0; e.g. the first group is the touches for display 0, the second group is the touches for display 1, and so on. Touch coordinates are signed and confined to the bounds of the display (with the origin at the top left pixel, x increasing horizontally, and y increasing vertically), and they are (-1,-1) when there is currently no touch.

Also defined in ***Hardware Information*** is an array of additional mouse inputs, 1 group per mouse which consists of an 8-bit value for mouse buttons followed by 2 signed 32-bit integers for scroll coordinates (X, followed by Y), with the index of each group in the array corresponding to the pair at the same index in the cursor coordinate array (e.g., the first byte of the second group is the additional mouse inputs for the mouse whose coordinates are given by the second cursor coordinate pair).

This is how the mouse buttons byte for each of these additional mouse inputs looks (bit is 1 when pressed down, 0 otherwise):
MSB 5 extra bits | left mouse button | right mouse button | middle mouse button **LSB**

X and Y scroll coordinates start at 0 for each mouse. The X coordinate is decremented by 1 for each scroll to the left, and incremented by 1 for each scroll to the right. The Y coordinate is decremented by 1 for each scroll up, and incremented by 1 for each scroll down.

Controller Information

Defined in ***Hardware Information*** are states for the number of controller buttons, number of controller axes, number of controller positions, number of controller orientations, and an address to the controller information. In the controller information is recorded information about connected controllers, such as gamepads or head-mounted displays. The address of the beginning of controller information is static, so implementations should allocate a maximum amount of memory available for controller information and deny connection of controllers when their connection would surpass that maximum limit. Either implementations or Piculet software should provide menus for configuring and mapping the controller inputs to the controller information. Controller information is composed of several contiguous regions of data, and in order from first to last address are the regions for the buttons, controller axes, positions, and orientations. Each region is of varying sizes based on however many of them are connected.

For each button there is a byte which has a value of 0 if not pressed and a value of 1 if it is pressed.

For each axis there are 3 32-bit floating-point values, the first being the value of the axis, the second being the minimum value for that axis (all bits are set to 1 if unknown), and the third being the maximum value for that axis (all bits are set to 1 if unknown).

At device calibration, the axes that a device's position and orientation are measured on are set into place according to the rules described in ***Coordinate System***. The origin of these axes is placed at the center of the device, and these axes will not shift in their position or orientation until the device is recalibrated. For each position there are 3 32-bit floating-point values measuring, in meters, the position of the device relative to its last calibrated origin point. For each device's orientation there are 4 32-bit floating-point values composing a unit quaternion which specifies the rotation of the device around its axes. In calculating rotation around the axes, the distance of the device from the origin of the device's

axes has no effect, although the original orientation of the axes does; the axes essentially join at the center of the device for the purpose of rotation calculations so that distance of the device from its calibrated origin does not affect the recording of its orientation.

Image Capture

Images can be captured of the display or from the device's cameras. Defined in ***Hardware Information*** are states that provide information about the number of cameras available to capture images from, as well as the dimensions of each camera. If the device has front and back facing cameras, the IDs of the front-facing cameras should come before those of the back-facing cameras. The pixel dimensions of each camera's images are specified in contiguous pairs consisting of two 32-bit unsigned integers, the first integer specifying width and the second the height. The first of these pairs is at the 64-bit address of camera image dimensions, and there is one pair for each available camera, with the number of each pair corresponding to the number of the camera. If there are no available cameras, the address to camera image dimensions is 0.

Images can be captured from the cameras or display and read into main memory using instruction 118. For security, threads have the option to restrict access to this instruction, either allowing camera capture, display capture, both, or none at all. When a thread is created, the thread creating it can choose whether or not to pass down the image capture permissions that it has, and no thread can create a thread that has image capture permissions that it itself does not have. Thread 0 has permission to capture images from both the cameras and the display.

Audio

Piculet has support for 3D audio, audio occlusion, and physics-based sound propagation effects, each of which may or may not be supported. There are two objects allowing audio playback which are placed somewhere in 3D space; audio sources and audio listeners. Audio sources get their audio data from audio data objects bound to them. There is also audio occlusion geometry, which can be bound to the bindings provided for each audio listener. Like all other objects, the audio data objects, audio source objects, audio listener objects, and audio occlusion geometry objects are created by instruction 72. The hardware support information, defined in ***Hardware Information***, specifies whether or not audio is supported, whether or not 3D audio is supported, whether or not audio occlusion is supported (requires that 3D audio is supported), and whether or not sound propagation effects are supported (requires that audio occlusion is supported; if supported it means that the acoustic properties of audio occlusion geometry are actually taken into account and affect propagated sounds). If sound propagation effects are not supported, all audio occlusion geometry absorbs all sound colliding with it; no sound is transmitted, and none is reflected. Most configuration of these objects are done through instruction 121; for e.g., it can be used to adjust the volume of audio source objects. Audio files can be uploaded into audio data objects using instruction 122 by specifying a file path to a file of one of the supported audio formats. If a source and listener are at the same position, the listener hears the source's audio exactly as it is when at the source's set volume. If 3D audio is supported, sound should fade out realistically with distance (as if traveling through air) as the source and listener become more distant. The coordinates for audio sources, listeners, and occlusion geometry vertices are all measured in meters. If 3D audio is not supported there is no sound attenuation and the positions of sources and listeners have no effect. While audio occlusion geometry is bound to a listener, audio occlusion is active if it is supported, and if sound propagation effects is supported as well the propagation of sound from all sources bound to that listener is then subject to the effects of all of that listener's bound audio occlusion geometries' acoustic properties.

A listener can be muted, and a source can be muted for a specific listener, using instruction 121. By default, the audio data object binding of a source object is 0 (therefore there is no audio data for the audio source to play; because of this, the source object has a sample count of 0 and they are 0 samples into audio playback), the volume of a source is 100, listeners are muted for all sources, the sample rate for playback of audio sources is equal to the sample rate of the last bound audio data object (if the last bound audio data object had no audio data or none was bound to that audio source yet, the sample rate for playback is 0), listeners are not muted, listeners have no audio occlusion geometries at any binding points, both sources and listeners have positions at (0,0,0), and a source's behavior for the end of audio playback is to stop playback (which leaves the number of samples into the audio playback equal to the sample count of the audio data - 1, instead of looping playback back to the beginning of the audio data). Each audio occlusion geometry binding in a listener object has an associated position (measured in meters) and rotation (a unit quaternion), which are set to (0,0,0) and (0,0,0,1), respectively, for a binding point whenever instruction 121 is executed to bind an audio occlusion geometry object to that binding point.

If a source has the same position as a listener it is bound to or 3D audio isn't supported, the audio channels of the source will output audio as-is to the appropriate channels of the audio output device, unaffected by occlusion and sound propagation effects. If a multichannel audio source doesn't have the same position as a listener while 3D audio is supported, the audio source will emit the sounds from all of its audio channels during playback.

Audio data can be multichannel. In audio with 2 channels, channel 0 is the left channel and channel 1 is the right channel. Channels are numbered from audio channel 0 upward, and the maximum number of channels that can be loaded, created, or modified is specified by maximum number of audio channels in ***Hardware Information***. The maximum number of audio channels is equal to the minimum of the maximum number of supported channels across all supported audio file types. The sample count of audio data and audio files is the number of samples per channel.

Once audio is loaded into an audio data object, the audio data in the data object is independent of the source from which it was loaded (either from microphone recording, video, or an audio file).

When an audio data object is bound to an audio source, a copy of the audio currently in the audio data object is what the audio source will use until the next time an audio data object bind is performed on the audio source. Due to this, an audio source does not require the audio data object bound to it to continue to exist. For an implementation to cut down on the memory usage from needing to make copies of audio data, it may only store one copy of the audio data object for use across all audio sources it is bound to and only make different copies of it every time the audio data object is modified.

The acoustic properties of geometries that allow sound propagation effects include percentages of how much reflected sound is scattered by the geometry (in range [0,1]; 0 is mirror-like reflection while 1 is a rough surface causing diffuse reflection of sound), and percentages (in range [0,1]) of sound transmitted and absorbed for low, medium, and high frequency sounds. The center frequencies for these three frequency bands are 400 Hz, 2.5 KHz, and 15 KHz.

An audio occlusion geometry object is a buffer specifying the vertex positions of triangles which are grouped by their acoustic properties. Each group consists of a 32-bit unsigned integer specifying the number of triangles in the group (0 corresponds to 1), followed by the 3 vertex positions for each of all of the triangles in the group, with 3 contiguous coordinate pairs composed of 3 32-bit floating-points with the X, followed by the Y, followed by the Z coordinate (with every 9 32-bit floating-points describing one triangle). At the end of each group are the acoustic properties for all triangles in that group; 7 32-bit floating-points in range [0,1] (with values > 1 being treated as a value of 1 and values < 0, infinite, or NaN being treated as a value of 0) for how much reflected sound is scattered, followed by how much low frequency sound is transmitted, followed by how much medium frequency sound is transmitted, followed by how much high frequency sound is transmitted, followed by how much low frequency sound is absorbed, followed by how much medium frequency sound is absorbed, and finally followed by how much high frequency sound is absorbed. If the triangle data or the data for its associated acoustic properties is incomplete (the end of the audio occlusion geometry object buffer comes too soon), all geometry for the group that falls off the end of the buffer is completely ignored. Like with audio data objects bound to an audio source, audio occlusion geometry is copied into a listener's geometry binding points so they must be rebound to a listener after being modified for the updates to take effect, and audio occlusion geometry objects can be deleted after bound to a listener's geometry binding points without having any effect on the geometry stored at the listener's geometry binding points.

Audio Recording

Audio can be recorded from the device's microphones. Defined in ***Hardware Information*** is state that provides information about the number of microphones available. When recording audio it is recorded into an audio file of a supported audio format; if that audio file does not exist it is created, and if it does exist it will be overwritten. Instruction 122 is used to start and stop recording from the microphones (each microphone is not recording anything by default, and this instruction acts as a recording toggler) into a specified audio file. For security, threads may have restricted access to this functionality. Each microphone can only record to one file at a time. Microphone numbering begins at 0 for the first microphone and increments by 1 for each connected microphone. Microphones will always be numbered consecutively, so if one is disconnected the number of all microphones greater than the one disconnected will decrease by 1. Only once an audio recording is stopped should the recorded audio file be touched, as it may or may not even exist before the microphone is finished recording to it. When a thread is created, the thread creating it can choose whether or not to pass down the audio recording permissions that it has, and no thread can create a thread that has

audio recording permissions that it itself does not have. Thread 0 has audio recording permission.

Video

Piculet has support for video playback, which is done by loading a portion of a video file into a video data object. Once video is loaded into a video data object, instruction 123 can be used to place a specific frame into memory where it can then be used. For example, it can be displayed by uploading it into a texture and then drawing the texture to the screen. A video's audio can be played by using instruction 122 to load a portion of the video's audio into an audio data object (unless the video contains no audio data). Once video is loaded into a video data object, the data in the data object is independent of the file from which it was loaded. Image files can also be loaded into video data objects; they'll simply have one frame and no audio data.

Supported File Formats

In ***Hardware Information*** there are two addresses to null-terminated ASCII strings, one listing the supported audio file formats, and one listing the supported video and image file formats. These are lists of supported file extensions which are lowercase and comma-separated, containing no whitespace or periods.

Networking

Piculet has support for both TCP and UDP protocols, with both IPv4 and IPv6 addresses. Instruction 124 is used for controlling networking. When socket objects are created they are defined as TCP or UDP, and IPv4 or IPv6, and also have their IP address and port number assigned. The networking operations; send, receive, connect, ping, get string of hostname at IP, get length of string of hostname at IP, get IP of given hostname, and accepting a connection; will all (if they don't result in an immediate error) end the cycle for the thread they are executed in and the thread may only continue execution once the operation is complete. The implementation may or may not populate the region of memory specified as storage for data as it's being received (i.e., it may opt to copy it all over in one chunk after received), though before an operation to send data out the data to send will be copied to a safe area outside of main memory to ensure that the data to send remains the same while it is being sent out. Only one socket can be listening per port and attempting to set more than one listening socket for a port will result in an error. All of the other networking operations not mentioned in the list above (listen, get system IP, etc.) will not end the cycle or interrupt execution. To close a socket, simply delete the socket object.

Key Presses

In the memory block of hardware information, there are 72 bits (9 bytes) representing all of the recognized key presses, with one for each connected keyboard. The 9-byte units for each connected keyboard are contiguous in the key presses array defined as part of ***Hardware Information***. Their bits are continuously updated and set to 1 when the corresponding key is pressed and set to 0 when not pressed. Bits that are for undefined keys will always be 0. This is the complete list of keys, based on the US QWERTY keyboard. Each row represents 1 of 9 bytes, the leftmost column being the MSB and the rightmost column being the LSB, the topmost row being the lowest address byte and the bottommost row being the highest address byte.

undefined	undefined	undefined	undefined	undefined	shift key	tab key	return key
spacebar	caps lock	escape key	ctrl key	backspace	alt key	↑ arrow key	↓ arrow key
← arrow key	→ arrow key	super key	page ↑ key	page ↓ key	home key	end key	insert key
delete key	1 key	2 key	3 key	4 key	5 key	6 key	7 key
8 key	9 key	0 key	~ key	- key	= key	q key	w key
e key	r key	t key	y key	u key	i key	o key	p key
a key	s key	d key	f key	g key	h key	j key	k key
l key	z key	x key	c key	v key	b key	n key	m key
[key	] key	; key	‘ key	\ key	, key	. key	/ key

Strings and File Paths

Strings are in ASCII encoding, null-terminated, and read up to 1000 characters. If a null terminator isn't found before the 1001st character, the string is invalid. String lengths always include the null terminator byte. All file paths are given from the root of the thread's current drive (stated in the SR); if the current drive is non-existent, things will behave as if

drive 0 (the main drive) is selected. File path and filename rules are as follows (throughout this specification, “filename” refers to the name of a file or a directory):
Filenames are case insensitive. If there are two files in a directory with the same name but different casing, neither will be recognized as existing files (they are effectively invisible) and attempting to create a file in that directory will fail and result in a file error with the unknown error error code.
Filenames can be made up of the characters with ASCII codes greater than 0x1F and less than 0x7F, excluding 0x22, 0x2A, 0x2F, 0x3A, 0x3C, 0x3E, 0x3F, 0x5C, and 0x7C. That is, filenames cannot contain any of the following characters: \, /, :, *, ?, “, <, >, |. Filenames cannot be “.” or “..”. Filenames violating these rules are invalid.
/ is used as a separator between file and directory names and so it is impossible to use them in a filename. Unless at the end of the file path, / has to be followed by a valid filename. If / appears at the end of a file path string, it makes it clear that it is referring to a directory, but this is not necessary as directories cannot contain the same filename twice (therefore a directory and file with the same name cannot coexist in a directory). If more than one / appears next to each other, the file path is invalid.
A file path string need not start with the / character, however it will be implied.
Any file paths containing any invalid filenames are considered invalid.
Any file paths with a / immediately following a filename that is not of a directory are considered invalid.
When an instruction outputs a file path to main memory, the file path string starts with a /, and also ends with a / if it’s of a directory.

File Errors

The following are the error codes that can be set at failure of instruction 60 or 61; the file error bits are set to 1 if the error occurred and 0 if they didn’t (these bits are found in the SR; at the top of the list below is the bit closest to the MSB):

Unknown error; an error not recognized by the error codes occurred.
Read/write permission was denied for the given file path.
The filename already exists at the given file path.
The filename is too long.
The file path contains invalid characters.
Too many files are already open.
The file system is read-only.

If a drive becomes unavailable while it is set to the current drive of a thread, that current drive for that thread will be set to 0.
If the current drive changes while a file stream to a file on the previous drive is still open, the file stream will still be usable and open to the file on the previous drive.
Open file streams whose file’s drive becomes unavailable will be closed.

Drive Types

Obviously, the more types of drives an implementation can support, the better. Implementations must only support drive 0 correctly and it must always be accessible (even if it’s just a RAM drive), and other drives are not guaranteed to even be recognized (though they really should be wherever possible). Implementations should also allow access of individual drive partitions as individual drives.
These are all of the drive type values:
0 - unknown
1 - HDD/SSD
2 - Virtual drive
3 - USB drive
4 - RAM drive

Texture Formats

Let each R, G, B, A letter refer to the pixel components, each of the named bit count. Texture formats and their values include:
0 - R 8-bit signed integer
1 - R 8-bit unsigned integer

2 - R 32-bit floating-point
3 - R 8-bit unsigned fixed-point (range [0,1])
4 - RG 8-bit signed integer
5 - RG 8-bit unsigned integer
6 - RG 32-bit floating-point
7 - RG 8-bit unsigned fixed-point (range [0,1])
8 - RGBA 8-bit signed integer
9 - RGBA 8-bit unsigned integer
10 - RGBA 32-bit floating-point
11 - RGBA 8-bit unsigned fixed-point (range [0,1])
12 - 32-bit floating-point depth
13 - 24-bit fixed-point depth + 8-bit stencil

Vertex Formats

Let each X, Y, Z, W letter refer to the vertex components, each of the named bit count. Vertex formats and their values include:
0 - X 32-bit floating-point
1 - XY 32-bit floating-point
2 - XYZ 32-bit floating-point
3 - XYZW 32-bit floating-point
4 - X 32-bit signed integer
5 - XY 32-bit signed integer
6 - XYZ 32-bit signed integer
7 - XYZW 32-bit signed integer
8 - X 32-bit unsigned integer
9 - XY 32-bit unsigned integer
10 - XYZ 32-bit unsigned integer
11 - XYZW 32-bit unsigned integer

Grouped Instructions

Excepting the output register setter, these are instructions that have 1 as their MSB (leftmost bit). These are defined this way to concisely establish groups of different variations of instructions which take different arbitrary values encoded in a part of the instruction. Bolded bits can have any value.

1100 | **0** | **000**
Move instruction; sets bytes at the LSB end of the primary or secondary register. If the first arbitrary bit is 0, this will move the bytes into the primary register, and into the secondary register otherwise. The value of the last 3 arbitrary bits specifies the number of bytes for the move instruction to move (0 corresponds to 1). The bytes that the move instruction will move immediately follow the move instruction, in little-endian byte ordering (so e.g. the byte immediately following the instruction is set to the LSB of the register), and, in addition to the automatic +1 after the instruction executes, the PC will have the number of bytes added to it before the move takes place in order for execution to skip the bytes being moved (this is an automatic and not a manual modification of the PC, therefore this must not trigger the end of the cycle); the bytes to move are part of the instruction and so must be within the bounds of the instruction memory or else the thread will be killed when the move instruction is encountered. Before moving bytes to a register the register is first cleared with 0s.

1101 | **0** | **000**
Byte swapper; swaps/mirrors bytes within a region at the LSB end of the primary or secondary register (without affecting the bit patterns of the individual bytes), useful for swapping the endianness of data. If the first arbitrary bit is 0, this will swap bytes at the LSB end of the primary register, or bytes at the LSB end of the secondary register otherwise. The value of last 3 arbitrary bits is equal to the number of bytes to swap - 1 (a value of 0 will swap 8 bytes, as if the value was actually 7).

11100 | **0 | 00**
Load bytes into a register from the memory address specified by the other register. If the first arbitrary bit is 0, this will

load a number of bytes specified by the next 2 bits (0=1 byte, 1=2 bytes, 2=4 bytes, 3=8 bytes) into the primary register, and into the secondary register otherwise. Bytes are read in little-endian byte ordering (so e.g. lowest address is read to the LSB end of the register), and before setting the bytes the register is first cleared with 0s.

11101 | **0 | 00**
Pop value to either primary or secondary register from the stack. If the first arbitrary bit is 0, the value is popped to the primary register, and to the secondary register otherwise. Reads some number of bytes specified by the next 2 arbitrary bits (0=1 byte, 1=2 bytes, 2=4 bytes, 3=8 bytes) from main memory at the current address (value of SP register), and then adds that number of bytes to SP if the memory read didn't result in a segmentation fault. Bytes are read in little-endian byte ordering (so e.g. lowest address is read to the LSB end of the register), and before setting the bytes the register is first cleared with 0s.

11110 | **0** | **00**
Store bytes from one register to a memory address specified by the other register. If the first arbitrary bit is 0, this will store a number of bytes specified by the next 2 bits (0=1 byte, 1=2 bytes, 2=4 bytes, 3=8 bytes) into the address specified by the primary register, and into the address specified by the secondary register otherwise. Stored bytes are obtained from the LSB end of the register, and bytes are stored in little-endian byte ordering (so e.g. the LSB of the register is stored at the lowest address).

11111 | **0** | **00**
Push value from either primary or secondary register to the stack. If the first arbitrary bit is 0, the value is from primary register, and from the secondary register otherwise. Subtracts some number of bytes specified by the next 2 arbitrary bits (0=1 byte, 1=2 bytes, 2=4 bytes, 3=8 bytes) from SP and then stores that number of bytes in main memory at the result address (new value of SP) from LSB end of the specified register, stored in little-endian byte ordering (so e.g. the LSB of the register is stored at the lowest address).
1010 | **0000**
Set the primary register to refer to the one named by the 4 arbitrary bits (register 0-15).

1011 | **0000**
Set the secondary register to refer to the one named by the 4 arbitrary bits (register 0-15).

0000 | **0000**
Set the output register to refer to the one named by the 4 arbitrary bits (register 0-15).

100 | **00000**
Performs an arithmetic operation. See ***Arithmetic Operations*** for information on the possible values of the arbitrary bits. Unless the operation performed was a comparison, the result of the arithmetic operation will be output to the output register.

Arithmetic Operations

There are 32 different arithmetic operations supported by Piculet including addition, subtraction, division, multiplication, comparison, raise to power, modulo, and conversions between multiple data types. Additionally, instructions 16-30 and 113-116 perform operations that could not fit to be represented by the 5 arbitrary bits of the arithmetic operation instruction, and instruction 117 can be used to compute the result of some common math functions. Comparisons affect the conditional bits as described in ***Conditionals***. During an operation, the bits out of the range of the size of the operands are ignored and appear as 0 in the result (since any time a register is output to its bits are first cleared to 0). Integer divisions are truncated (the fractional part is ignored). Both integer and floating-point modulo (x mod y = x-(x/y)*y) is calculated with truncated division. Any arithmetic operations that might result in the program crashing (e.g., integer division by 0) will instead result in a value of 0. In floating-point arithmetic operations, if the divisor is 0 or either operands are NaN or positive/negative infinity, the result is NaN. Below, all available arithmetic operations are written in binary form.

00000 32-bit integer addition of primary register with secondary register.
00001 32-bit integer subtraction of secondary register from primary register.
00010 32-bit integer multiply of primary register by secondary register.

00011 32-bit signed integer division of primary register by secondary register.
00100 32-bit unsigned integer division of primary register by secondary register.
00101 32-bit signed integer modulo of primary register by secondary register.
00110 64-bit integer addition of primary register with secondary register.
00111 64-bit integer subtraction of secondary register from primary register.
01000 64-bit integer multiply of primary register by secondary register.
01001 64-bit signed integer division of primary register by secondary register.
01010 64-bit unsigned integer division of primary register by secondary register.
01011 64-bit signed integer modulo of primary register by secondary register.
01100 32-bit floating-point addition of primary register with secondary register.
01101 32-bit floating-point subtraction of secondary register from primary register.
01110 32-bit floating-point multiply of primary register by secondary register.
01111 32-bit floating-point division of primary register by secondary register.
10000 32-bit floating-point primary register raised to the power of the secondary register.
10001 64-bit floating-point addition of primary register with secondary register.
10010 64-bit floating-point subtraction of secondary register from primary register.
10011 64-bit floating-point multiply of primary register by secondary register.
10100 64-bit floating-point division of primary register by secondary register.
10101 64-bit floating-point primary register raised to the power of the secondary register.
10110 32-bit integer comparison between primary and secondary registers.
10111 64-bit integer comparison between primary and secondary registers.
11000 32-bit floating-point comparison between primary and secondary registers.
11001 64-bit floating-point comparison between primary and secondary registers.
11010 Convert primary register from 32-bit floating-point to 64-bit floating-point.
11011 Convert primary register from 64-bit floating-point to 32-bit floating-point.
11100 Convert primary register from signed 32-bit integer to 32-bit floating-point.
11101 Convert primary register from signed 64-bit integer to 64-bit floating-point.
11110 Convert primary register from 32-bit floating-point to signed 32-bit integer.
11111 Convert primary register from 64-bit floating-point to signed 64-bit integer.

SIMD Operations

This section is about the different operations which can be performed by the SIMD instructions (125-127), the effects they have, and how to use them. SIMD operations are performed like the normal arithmetic operations and all of the same rules apply, except they are performed simultaneously if the implementation supports it and the results aren't output to a register but are rather written to a location in memory. Conditional bits will not be updated by SIMD operations.

Each thread has two 256-bit vectors which contain operand values for SIMD operations, with one vector for left operands and one vector for right operands. For each operation performed, values from the same indices within each vector are used as left/right operands starting at the beginning of the vectors, and the result is then written to a location in memory as an array of values, at the same index in the array as the operands.

There is an instruction to write values into a vector for each the left operand and the right operand vector, starting at the beginning of the vector; the initial values in the vectors are undefined. Instruction 127 can then be used to perform a SIMD operation, with the primary register specifying the operation to perform and the number of operations to perform, and the secondary register specifying the address in main memory to write the results to (writes a result for each operation performed).

Each vector has a size of 32 bytes, so the maximum number of operations for 32-bit types is 8 (cannot store more than 8 operands in a vector) and the maximum number of operations for 64-bit types is 4 (cannot store more than 4 operands in a vector).

In instruction 127, the primary register has 8 bits on the LSB end to represent what operation to perform as well as how many operations to perform (1-8 for 32-bit types, and 1-4 for 64-bit types), as shown below:
000 | **00000**
The 5 bit sequence is the arithmetic operation to perform (one of those specified in ***Arithmetic Operations***). The 3-bit sequence is the number of operations to perform (0 corresponds to 1). If the arithmetic operation is a type conversion or comparison, the SIMD instruction will do nothing. If the data type for the operation is 64-bit and the number of operations to perform is > 4, the SIMD instruction will do nothing. There are as many result values written to memory as there are operations performed. All other bits in the primary register (everything other than the byte at the LSB end) are ignored.

Regardless of whether or not the implementation supports proper SIMD operation, it will perform and write the results of each of the operations to memory before the SIMD instruction finishes execution.

Instructions

This section lists the Piculet instructions, including what they are and how they act. Each instruction is a byte in width.

0-15 0-F Output register setters (see ***Grouped Instructions***).
16 10 Toggle each bit of the primary register and then increment it by 1.
17 11 Toggle each bit of the secondary register and then increment it by 1.
18 12 Rotate left the primary register by number of bits specified by secondary register.
19 13 Rotate right the primary register by number of bits specified by secondary register.
20 14 Logical left shift the primary register by number of bits specified by secondary register.
21 15 Logical left shift the secondary register by number of bits specified by primary register.
22 16 Logical right shift the primary register by number of bits specified by secondary register.
23 17 Logical right shift the secondary register by number of bits specified by primary register.
24 18 Arithmetic right shift the primary register by number of bits specified by secondary register.
25 19 Arithmetic right shift the secondary register by number of bits specified by primary register.
26 1A Bitwise OR primary register with secondary register and output the result to the output register.
27 1B Bitwise AND primary register with secondary register and output the result to the output register.
28 1C Bitwise XOR primary register with secondary register and output the result to the output register.
29 1D 32-bit unsigned integer modulo of primary register by secondary register and output the result to the output register.
30 1E 64-bit unsigned integer modulo of primary register by secondary register and output the result to the output register.
31 1F Copy contents of primary register to secondary register.
32 20 Copy contents of secondary register to primary register.
33 21 Clear primary register with 0s.
34 22 Clear secondary register with 0s.
35 23 Clear primary register with 1s.
36 24 Clear secondary register with 1s.
37 25 Create a thread descendant of this one which begins its execution at the instruction addressed by the primary register, and store the thread ID in the output register. The secondary register is an address to information about the new thread's properties. The byte at this address specifies the thread's permissions; the rightmost bit is 1 if the thread can capture images from the display, the first bit from the LSB end is 1 if the thread can capture images from the cameras, the second bit from the LSB end is 1 if the thread can record audio from the microphones, the third bit from the LSB end is 1 if the thread can perform networking operations, the fourth bit from the LSB end is 1 if the thread can perform file I/O operations, and the fifth bit from the LSB end is 1 if the thread can create new threads. The next 8 bytes specify the privacy key for the thread, the next 8 bytes specify the ID of the segment table object for the thread (can be 0, but the thread will not be able to access main memory), the next 8 bytes specify the instruction memory minimum, the next 8 bytes specify the instruction memory maximum, and the next 8 bytes specify the address to a file path string which is the highest accessible directory for the thread. Any permission specified to be passed down to the created thread will not actually be passed down to the new thread unless the thread creating it currently has that permission itself. If attempting to create a thread with an instruction memory range that reaches outside of main memory, the instruction memory minimum is greater than the instruction memory maximum, if the highest accessible directory specified is not equal to or below this thread's highest accessible directory at the time of creation, or if the highest

accessible directory is not an existing directory or an invalid file path string, nothing will happen. If this instruction is executed on a thread without thread creation permission, nothing will happen. For more information on how this instruction behaves, see ***Cycles, Jump Instructions, and Execution***.
38 26 Detach a thread descendant of this one whose ID is specified by the primary register, which makes the thread unable to be joined with using instruction 40. If the primary register is 0 or the thread specified is not a descendant, nothing will happen.
39 27 Destroy a thread descendant of this one whose ID is specified by the primary register. If the primary register is 0 or the thread specified is not a descendant, nothing will happen.
40 28 Join a thread descendent of this one whose ID is specified by the primary register. Ends the cycle and blocks all cycles of this thread until the moment termination of the descendant thread occurs. If the primary register is 0, the thread specified is detached, or the thread specified is not a descendant of this thread, nothing will happen.
41 29 Puts this thread to sleep for a number of nanoseconds specified by the primary register. Exact time slept may be slightly longer than specified. For more information on how this instruction behaves, see ***Cycles, Jump Instructions, and Execution***.
42 2A Used to get information about this thread or to modify the restrictions of a descendant thread. If the primary register is 0, this will output the ID of this thread to the output register. If the primary register is 1, this will output to the output register whether or not this thread has permission to capture images of the display (0 if it doesn't, 1 if it does). If the primary register is 2, this will output to the output register whether or not this thread has permission to capture images from the cameras (0 if it doesn't, 1 if it does). If the primary register is 3, this will output to the output register whether or not this thread has permission to record audio from the microphones (0 if it doesn't, 1 if it does). If the primary register is 4, this will output to the output register whether or not this thread has permission to perform networking operations (0 if it doesn't, 1 if it does). If the primary register is 5, this will output to the output register whether or not this thread has permission to perform file I/O operations (0 if it doesn't, 1 if it does). If the primary register is 6, this will output to the output register whether or not this thread has permission to create new threads (0 if it doesn't, 1 if it does). If the primary register is 7 or 8, this will output to the output register this thread's instruction memory range information (7=instruction memory minimum, 8=instruction memory maximum). If the primary register is 9, this will output to the output register the number of bytes for the null-terminated ASCII string of this thread's highest accessible directory file path string. If the primary register is 10, this will write to memory starting at the address specified by the secondary register this thread's null-terminated highest accessible directory file path string. If the primary register is 11, this will output to the output register this thread's privacy key. If the primary register is 12, the secondary register is a memory address to 8 bytes specifying the ID of a descendant thread (can be 0 if this instruction is run on thread 0 and it is updating its own privacy key), followed by a byte specifying what to update for the specified thread (0=image capture of display permission, 1=image capture from cameras permission, 2=audio recording permission, 3=networking permission, 4=file I/O permission, 5=thread creation permission, 6=instruction memory minimum, 7=instruction memory maximum, 8=highest accessible directory, 9=privacy key), followed by one byte if the updating one of the permissions (0 to disable, non-zero to enable), or 8 bytes otherwise (specifying new values just like during thread creation; e.g. if setting the highest accessible directory, this is a 64-bit address to a file path string). If the primary register is 13, the secondary register is the ID of a descendant thread, and this will output 1 to the output register if the specified descendant has been killed and 0 otherwise. If the primary register is 12 and the thread specified is neither a valid descendant thread nor thread 0 updating its own privacy key, this thread is attempting to give a descendant thread a permission that this thread doesn't have itself, the byte of what to update is > 9, the new instruction memory range reaches outside of main memory, the new instruction memory minimum is greater than the instruction memory maximum, or a highest accessible directory file path is specified that is not equal to or below this thread's highest accessible directory file path, nothing will happen. If the primary register is 13 and the thread specified is not a valid descendant thread, nothing will happen. If the primary register is > 13, nothing will happen.
43 2B Outputs 1 to the output register if the conditions for a specified jump instruction would be met, and 0 otherwise. The primary register is used to specify which jump instruction to check for; it is an offset from instruction number 45 (0 checks if instruction 45 would jump, 1 checks if instruction 46 would jump, ...). If the primary register is > 13, nothing will happen.
44 2C Jump to an instruction addressed by primary register.
45 2D Jump to an instruction addressed by primary register if Z is set.
46 2E Jump to an instruction addressed by primary register if Z is not set.
47 2F Jump to an instruction addressed by primary register if C is set.
48 30 Jump to an instruction addressed by primary register if C is not set.

49 31 Jump to an instruction addressed by primary register if N is set.
50 32 Jump to an instruction addressed by primary register if N is not set.
51 33 Jump to an instruction addressed by primary register if V is set.
52 34 Jump to an instruction addressed by primary register if V is not set.
53 35 Jump to an instruction addressed by primary register if C is set and Z is not set.
54 36 Jump to an instruction addressed by primary register if C is not set or Z is set.
55 37 Jump to an instruction addressed by primary register if N == V.
56 38 Jump to an instruction addressed by primary register if N != V.
57 39 Jump to an instruction addressed by primary register if Z is not set and N == V.
58 3A Jump to an instruction addressed by primary register if Z is set or if N != V.
59 3B Return from branch. Sets PC equal to the LR.
60 3C Open a file, move a file, or create one or more directories whose path is specified by the file path string starting at the memory address specified by the primary register. If that file path is an existing directory and the output register is 0, this moves a file specified by the file path string at the memory address specified by the secondary register to the directory specified by the file path string addressed by the primary register. If the file path at the address specified by primary register ends with /, this creates all directories specified in that path that do not already exist if it is not an existing directory and the output register is 0, and if the output register isn't 0 this will output 1 to the output register if the specified directory exists, or 0 otherwise. If it doesn't end with /, it will create and open the specified file only if the output register is 0, it is not an existing file or directory, and the directory the file will be created in already exists; if the output register isn't 0 this will output 1 to the output register if the specified file exists, and 0 otherwise. If opening either a new file or an already existing one, this will output the file stream's ID (always greater than 0, and they are private to each thread, so there can be more than one file stream open under the same IDs across different threads) to the output register, then set it as the current file stream in the SR. Each file stream can be buffered, and so all changes to a file through a specific file stream may not be present on disk until the file stream is closed. If moving a file and the file path string addressed by the secondary register is invalid or is not an existing directory, nothing will happen. On error, nothing happens other than that the file error bits are updated at the end of the instruction (along with whether or not the current file stream is open; see ***Status Register***). If this thread does not have permission to perform file I/O operations, nothing will happen.
61 3D Delete a file or directory whose path is specified by the file path string starting at memory address specified by the primary register. If some error occurs, the file error bits are updated accordingly at the end of the instruction. To successfully delete a directory, the directory must be empty (not contain any files or directories). If this thread does not have permission to perform file I/O operations, nothing will happen.
62 3E Close a file whose stream is named by the primary register. If the primary register is 0, this will close the current file stream. If the current file stream is 0 or not open, nothing will happen. If this thread does not have permission to perform file I/O operations, nothing will happen.
63 3F Set the current file stream using the 16 bits on the LSB end of the primary register.
64 40 Write bytes from the memory address specified by primary register to the current file stream, beginning on the byte specified by the secondary register (0 being the first byte of the file), for the number of bytes specified by the output register (0 corresponds to 1). If the current file stream is 0 or not open, or if the secondary register + the number of bytes to write is greater than the file's size in bytes, nothing will happen. If this thread does not have permission to perform file I/O operations, nothing will happen.
65 41 Read bytes from the current file stream to memory address specified by primary register, beginning on the byte specified by the secondary register (0 being the first byte of the file), for the number of bytes specified by the output register (0 corresponds to 1). If the current file stream is 0 or not open, or if the secondary register + number of bytes to read is greater than the file's size in bytes, nothing will happen. If this thread does not have permission to perform file I/O operations, nothing will happen.
66 42 Set the byte count of the current file to that specified by the primary register, or output the current byte count of the file to the output register if the primary register is 0. If the current file stream is 0 or not open, or if there is not enough storage available on the drive, nothing will happen. New bytes added to end of file are set to 0, and bytes removed to shrink the file size are truncated off of the end. If this thread does not have permission to perform file I/O operations, nothing will happen.
67 43 Output the number of available drives to the output register.
68 44 Output contiguous groups of values for each available drive describing each drive's information (each group output consists of an 8-bit drive type, followed by 64 bits stating the drive's capacity in bytes, followed by 64 bits stating how much free space, in bytes, there is on the drive) to a memory address specified by the primary register and a limit

for number of groups output specified by the secondary register. See ***Drive Types*** for a complete list of values for drive types. If the secondary register is 0, nothing will happen.
69 45 Output a list of the information of all files and directories located in the directory named by the file path string beginning at the memory address specified by the primary register, by writing to main memory the collected file data beginning at memory address specified by secondary register. The first 8 bytes output will be the number of bytes worth of directory strings output, the next 8 bytes output will be the number of bytes worth of filename strings + the number of bytes worth of file sizes output, followed by the null-terminated strings of directory names, all followed by the null-terminated strings of names of files which are each followed (after the null terminator) by 64 bits stating the size, in bytes, of the file. Note how this instruction only gets filenames and not file paths, and as a result no / will be present at the beginning nor the end of listed file or directory names. Filenames are not listed in any specific order. If the file path specified is invalid or the directory does not exist, nothing will happen.
70 46 Output, to the output register, the total number of bytes instruction 69 will output, with the file path string beginning at the address specified by the primary register. If the file path is invalid or the directory does not exist, nothing will happen.
71 47 Set the current drive with the ID specified by the 8 bits on the LSB end of the primary register. If the drive is non-existent, nothing happens. The ID of drives are numbered consecutively from 0 and the drive IDs correspond with the indices into the array of groups output by instruction 68.
72 48 Generate (validate for use) a new object of type specified by the 6 bits at the LSB end of the primary register (0=cbo, 1=vao, 2=vbo, 3=ibo, 4=tbo, 5=fbo, 6=ubo, 7=sbo, 8=top-level acceleration structure, 9=bottom-level acceleration structure, 10=data buffer object, 11=shader binding table, 12=sampler descriptor, 13=image descriptor, 14=uniform descriptor, 15=storage descriptor, 16=acceleration structure descriptor, 17=descriptor set, 18=descriptor set layout, 19=vertex shader, 20=pixel shader, 21=ray generation shader, 22=any-hit shader, 23=closest-hit shader, 24=miss shader, 25=compute shader, 26=rasterization pipeline, 27=ray tracing pipeline, 28=compute pipeline, 29=audio data object, 30=audio source object, 31=audio listener object, 32=audio occlusion geometry object, 33=video data object, 34=socket object, 35=segment table object) and output its ID to the output register. If those 6 bits have a value > 35, nothing will happen and the output register will not be modified. If a VAO is being generated, the secondary register is an address to a region of memory with the vertex array object creation properties described under ***Vertex Array Object Creation***; if the vertex array object creation fails nothing will happen and the output register will not be modified. If a TBO is being generated, the secondary register specifies one of the texture formats (see ***Texture Formats***) for the generated TBO; if the secondary register isn't a valid TBO format, nothing will happen and the output register will not be modified. If a descriptor set is being generated, the secondary register is the ID of a descriptor set layout object describing the layout of the descriptor set. If a descriptor set layout is being generated, the secondary register is an address to a region of memory with the descriptor set layout creation properties described under ***Descriptor Set Layout Creation***; if the descriptor set layout creation fails nothing will happen and the output register will not be modified. If a pipeline is being generated, the secondary register is an address to a region of memory with pipeline creation properties described under ***Pipeline Creation***; if the pipeline creation fails nothing will happen and the output register will not be modified. If a socket object is being generated, the secondary register is an address to a byte specifying whether the socket type is TCP or UDP (0=TCP, non-zero=UDP), followed by a byte specifying whether the socket's IP address is IPv4 or IPv6 (0=IPv4, non-zero=IPv6), followed by two bytes for the unsigned 16-bit integer port number (value of 0 will automatically bind the socket to an unused, available port), followed by the 4 or 16 bytes for the IP address the socket is bound to.
73 49 Delete an object previously created by instruction 72 with ID specified by the primary register. Will free all of its contents. Does nothing if the object does not exist or the object's buffer is mapped.
74 4A Bind an object generated by instruction 72 with ID specified by the primary register. Binding an object with ID 0 will not do anything unless the 6 bits on the LSB end of the secondary register are one of the values supported by the primary register in instruction 72, to specify which object type's binding should be cleared.
75 4B Bind an FBO whose ID is specified by the primary register to the bound command buffer. If the FBO whose ID was specified is not a valid FBO, nothing will happen (an exception is made for ID 0, to allow binding of the default framebuffer).
76 4C Bind a UBO, SBO, TBO, or top-level acceleration structure with ID specified by the primary register to the bound uniform, storage, sampler, image, or acceleration structure descriptor, depending on the type of the specified object. If a TBO is specified and the secondary register is 0, this will bind the TBO to the bound sampler descriptor. If a TBO is specified and the secondary register is non-zero, this will bind the texture to an image descriptor, with the texture level of the image specified by the 32 bits on the LSB end of the secondary register - 1. If binding a TBO or image with a depth or depth + stencil texture format, nothing will happen. If binding an image and the texture level specified does

not exist in the TBO, nothing will happen.
77 4D Bind a pipeline object whose ID is specified by the primary register to the bound command buffer. If the object whose ID was specified is not a valid pipeline, nothing will happen. Once a command buffer has had its first pipeline binding command recorded to it, ever attempting to bind a pipeline of a different type (rasterization/ray tracing/compute) to that command buffer will do nothing unless all pipeline binding commands have been removed by resetting the command buffer. This is recorded to the bound command buffer.
78 4E Updates the descriptor bindings of the bound descriptor set. The primary register specifies the number of descriptor bindings to update (0 corresponds to 1), and the secondary register is an address to memory which contains contiguous groups of values for each of the descriptor bindings being updated. Each group is composed of a 32-bit unsigned integer specifying the descriptor binding point number, followed by a 64-bit unsigned integer specifying an ID of a descriptor to be assigned to the descriptor binding point, followed by a 16-bit unsigned integer specifying the index in the descriptor binding to bind the descriptor to (these are present only for sampler descriptor bindings). If the descriptor set's descriptor set layout object has been deleted, nothing will happen. If any of the descriptors being set to a binding point are of a type that doesn't match the type of that binding point in the descriptor set's descriptor set layout, nothing will happen. If any binding points are specified for update that do not exist in the descriptor set's descriptor set layout, nothing will happen. If there are any repeated pairs of a descriptor binding point number and a descriptor binding point index, nothing will happen. If the index of a binding point specified to bind a sampler descriptor to is greater than the number of descriptors at the binding in the descriptor set's descriptor set layout - 1, nothing will happen. If any of the descriptor IDs are not of a valid descriptor object, nothing will happen.
79 4F Bind a descriptor set to a set binding in the bound command buffer, or a VBO or IBO within the bound command buffer, whose ID is specified by the primary register. If binding a descriptor set, the 8 bits on the LSB end of the secondary register specifies the set number to bind the specified descriptor set to when the command is executed; if the value of these 8 bits is > the value stored in ***Hardware Information*** for the maximum number of descriptor set bindings in a pipeline, nothing will happen. If binding a descriptor set and the descriptor set's descriptor set layout object has been deleted, nothing will happen. If any sampler descriptor binding points in the descriptor set contain more than 1 sampler descriptor and the command buffer is not using ray tracing pipelines, nothing will happen. If the descriptor set contains any acceleration structure descriptors and the command buffer is not using ray tracing pipelines, nothing will happen. If the descriptor set contains any storage or image descriptors and the command buffer is using rasterization pipelines, nothing will happen. If the descriptor set, VBO, or IBO ID specified is not the ID of a valid object, nothing will happen. This is recorded to the bound command buffer.
80 50 Output to the output register the current byte count of a bound buffer with type specified by the primary register (0=vbo, 1=ibo, 2=tbo, 3=ubo, 4=sbo, 5=top-level acceleration structure, 6=bottom-level acceleration structure, 7=data buffer object, 8=shader binding table, 9=shader object). If the primary register is > 9, nothing will happen. A default buffer (i.e., if object ID 0 is bound) has a byte count of 0.
81 51 Map or unmap a bound buffer with type specified by the primary register (0=vbo, 1=ibo, 2=ubo, 3=sbo, 4=data buffer object, 5=shader binding table, 6=shader object, 7=audio occlusion geometry object) to a region of system memory, allowing read/write to that region while that region is mapped. If mapping, outputs to the output register the address of the beginning of the mapped buffer if mapping was successful. Sets the buffer map error bit in the SR to 1 at the end of the instruction if the primary register is > 7 or there was another error, or sets it to 0 on success (regardless of whether the buffer is being mapped or unmapped).
82 52 Allocate memory for the buffer object whose ID is specified by the primary register and with a byte count specified by the secondary register. If the object specified is not a VBO, IBO, UBO, SBO, data buffer object, shader binding table, a shader object, or audio occlusion geometry object, the buffer allocation error bit will be set to 1 in the SR at the end of the instruction and nothing else will happen; other object types are allocated automatically when created or when data is uploaded to them. If the object specified is currently mapped, the secondary register is 0, or if the allocation fails due to memory limitations or some other reason, the buffer allocation error bit will be set to 1 in the SR at the end of the instruction and nothing else will happen. The values of the bytes of memory allocated are undefined. Allocating an object that has already been previously allocated will result in the old memory being discarded and all bytes in the new memory will have undefined values. If allocation is successful, the buffer allocation error bit in the SR will be set to 0 at the end of the instruction.
83 53 Upload a texture to the bound TBO. The primary register specifies an address which points to an 32-bit unsigned integer for texture width (0 corresponds to 1), followed by a 32-bit unsigned integer for texture height (0 corresponds to 1), followed by a 32-bit unsigned integer specifying the texture level to upload the data to. The secondary register specifies an address to the texture data; this assumes the texture format specified for the TBO at the time of its creation and the first texel at the address is at the top left of the texture and the texels upload to the texture from left to right,

from top to bottom. If a texture level is specified without all texture levels before it having previously been uploaded to (e.g., level 1 cannot be uploaded until level 0 has been) or the width or height are greater than the maximum texture size (defined in ***Hardware Information***), nothing will happen.
84 54 Generate mipmaps for the texture object currently bound, beginning at texture level 0 (full-size) and repeating half-scaled texture level generation throughout consecutive texture levels until the texture reaches 1x1. If texture level 0 is not a valid texture image, nothing happens. All texture levels generated for will be overwritten. If the texture object currently bound is of a depth or depth + stencil format, nothing will happen.
85 55 Attach a specified texture level of the currently bound texture to one of the color, depth, or stencil attachments of the currently bound framebuffer, based on the format of the bound texture (if the texture is depth + stencil, this will set both the stencil and depth attachments). If the bound texture object is of a color format (not a depth or depth + stencil texture), the 3 bits on the LSB end of the primary register specifies the color attachment to set (0-7) and the 4 bytes on the LSB end of the secondary register specifies the texture level of the texture to bind to the attachment. If the default texture object (texture object 0) is bound, this will unbind a set texture from the bound framebuffer's framebuffer attachment specified by the 4 bits at the LSB end of the primary register (0-7 is one of the color attachments, 8 is the stencil attachment, and 9 is the depth attachment). If unbinding a texture object from a framebuffer attachment and the primary register is > 9, nothing will happen. If the dimensions of the texture being set is not the same as that of any textures currently set to attachments of the framebuffer, nothing will happen.
86 56 Clear buffer(s) of the framebuffer bound to the command buffer at the time of its execution. The primary register specifies which attachment to clear (0=all enabled color attachments, 1=color attachment 0, 2=color attachment 1, ..., 8=color attachment 7, 9=depth attachment, 10=stencil attachment), and, depending on what type of attachment is being cleared, the secondary register specifies the address of the beginning of 4 32-bit floating-points for RGBA values to clear to (specified in that order; i.e., R is the first element in the array at the address, and A is the fourth element), or specifies in 4 bytes at the LSB of the secondary register a 32-bit floating-point for depth to clear to, or an 8-bit stencil value to clear to in the byte at the secondary register's LSB end. Depth and color clearing values are clamped to range [0,1]. If the primary register is > 10, nothing will happen. This is recorded to the bound command buffer.
87 57 Build or update a bound acceleration structure. If the primary register is 0 or 1, this will build the bound top-level acceleration structure (0 if it cannot be updated later on, or 1 if it can be updated later on). If the primary register is 2, this will update the bound top-level acceleration structure. For either building or updating the top-level acceleration structure, the secondary register is an address to a 32-bit unsigned integer specifying geometry instance count (only present for builds; if an acceleration structure update, there is assumed to be as many geometry instances provided as the acceleration structure was last built with), followed by that number of (contiguous) geometry instance descriptions (see ***Geometry Instance Descriptions***; if any of these geometry instance descriptions are not identical, ignoring the transformation matrix, to the geometry instance description in the last build, results are undefined), all followed by an 8-bit unsigned integer which is only present for builds specifying how to optimize this acceleration structure (0=optimize for build speed, 1=optimize for tracing performance, 2=optimize for compact size; if any BLAS in the given geometry instances for a TLAS build were not built with the same optimization type results are undefined). If the primary register is 3 or 4, this will build the bound bottom-level acceleration structure (3 if it cannot be updated later on, or 4 if it can be updated later on). If the primary register is 5, this will update the bound bottom-level acceleration structure. For either building or updating the bottom-level acceleration structure, the secondary register is an address to a 32-bit unsigned integer specifying geometry count (only present for builds; if an acceleration structure update, there is assumed to be as many geometries provided as the acceleration structure was last built with), followed by that number of (contiguous) geometry descriptions (see ***Geometry Descriptions***; if any of these geometry descriptions are not identical, ignoring data buffer object IDs, to the geometry description in the last build, or if any of the geometry descriptions refer to a valid data buffer object where the same geometry description in the last build didn't refer to a valid data buffer object or vice versa, results are undefined), all followed by an 8-bit integer which is only present for builds specifying how to optimize this acceleration structure (0=optimize for build speed, 1=optimize for tracing performance, 2=optimize for compact size). If the primary register is > 5, or if the instruction will cause the number of geometries in a BLAS, the number of geometry instances in a TLAS, or the number of triangles in all geometries in a BLAS to exceed their specified limit (see ***Hardware Information***), nothing will happen. If the 8-bit value provided for optimization type is > 2, nothing will happen. If attempting to update an acceleration structure that was not previously built or which was not built with the ability to be updated, nothing will happen. If the vertex buffer object, index buffer offset, or vertex position stride in any of the geometry descriptions in a BLAS build is not a multiple of 4, or the transformation matrix offset in any of the geometry descriptions in a BLAS build is not a multiple of 16, nothing will happen. The first 3 columns of transformation matrices given for geometry or geometry instances should form invertible 3x3 matrices, or else the acceleration structure may not build/update properly or at all. This is recorded to the bound command buffer.

88 58 Resets the bound command buffer.
89 59 Submits command buffers to a graphics queue for execution. The primary register is an address to memory listing the command buffers and queue number as described under ***Queue Submissions***; if the queue submission fails, nothing will happen.
90 5A Submits command buffers to a compute queue for execution. The primary register is an address to memory listing the command buffers and queue number as described under ***Queue Submissions***; if the queue submission fails, nothing will happen.
91 5B Ends the cycle and prevents all execution of this thread until the moment all command buffers previously submitted on this thread have finished. On implementations that don't support multithreading, this may start waiting until all previously submitted commands finish after all threads have finished executing a cycle.
92 5C Submit draw call of the vertex buffer bound in the command buffer at the time of execution. The primary register is an address to an 8-bit unsigned integer specifying whether or not the draw call is indexed (0 if non-indexed, non-zero otherwise), followed by a 32-bit unsigned integer specifying the count of vertices/indices to draw, followed by a 32-bit unsigned integer specifying the first VBO or IBO index, followed by a 32-bit unsigned integer specifying the number of instances to draw (0 corresponds to 1). If the draw call is indexed, the draw call will use the index buffer bound in the command buffer at the time of execution to get the indices of vertices in the VBO. This is recorded to the bound command buffer.
93 5D Submit an indirect draw call of the vertex buffer bound in the command buffer at the time of execution. The primary register is an address to memory providing information about the indirect draw calls. See ***Indirect Drawing*** for further information about this command. If the offset into the draw call parameters buffer is not a multiple of 4, nothing will happen. This is recorded to the bound command buffer.
94 5E Updates the store of a specified data buffer. The primary register is an address to a 64-bit unsigned integer specifying the ID of the data buffer object whose buffer will be updated, followed by a 64-bit unsigned integer specifying an offset into the data buffer, followed by a 16-bit unsigned integer specifying the number of bytes to update (0 corresponds to 1), all followed by the new data for the specified region of the data buffer (all read and copied to the command buffer when the command is recorded). If the region to update exceeds the bounds of the data buffer when the command is executed, results are undefined. If the offset or number of bytes being updated is not a multiple of 4, nothing will happen. This is recorded to the bound command buffer.
95 5F Updates push constant data. The primary register is an address to a 64-bit unsigned integer specifying the ID of the data buffer object the push constants will be read from, followed by a 64-bit unsigned integer specifying an offset into the data buffer, followed by an 8-bit unsigned integer specifying the number of bytes of push constant data to update (0 corresponds to 1). If the offset or number of bytes being updated is not a multiple of 4, nothing will happen. If push constants are set outside of the bounds of a pipeline's push constant range, results are undefined. See ***Push Constants*** for more information. This is recorded to the bound command buffer.
96 60 Dispatch a ray tracing operation. The primary register is an address to a 64-bit unsigned integer which is the ID of an SBT for the ray generation shader, followed by a 64-bit unsigned integer which is an offset into the ray generation SBT for the location of the shader index of the ray generation shader to use (must be a multiple of 8), followed by a 64-bit unsigned integer which is the ID of an SBT for the miss shaders, followed by a 64-bit unsigned integer which is an offset into that SBT for the location of the base of the miss SBT for the ray tracing operation (must be a multiple of 8), followed by a 64-bit unsigned integer which is the ID of a hit group SBT, followed by a 64-bit unsigned integer which is an offset into that SBT for the location of the base of the hit group SBT for the ray tracing operation (must be a multiple of 8), followed by a 32-bit unsigned integer which is the width of the ray tracing operation, followed by an 32-bit unsigned integer which is the height of the ray tracing operation, followed by an 32-bit unsigned integer which is the maximum recursion depth of the ray tracing operation (see ***Ray Tracing***). If either of the width or height is 0 or the maximum recursion depth specified is greater than the ray tracing pipeline's maximum recursion depth, nothing will happen. If any of the SBT offsets given are not a multiple of 8, nothing will happen. This is recorded to the bound command buffer.
97 61 Copy acceleration structure with ID specified by primary register to acceleration structure with ID specified by secondary register. If the acceleration structures specified are not of the same type, nothing will happen. This is recorded to the bound command buffer.
98 62 Swap color buffers of the default FBO for the current display. If it is single-buffered, nothing happens.
99 63 Sets the current display by setting current display value in the SR. The 8 bits on the LSB end of the primary register specify the display number. If the display number specified is > the current number of displays - 1, nothing will happen.
100 64 Set the bound sampler descriptor's filtering or texture wrapping modes. The 2 bits on the LSB end of the primary

register states what to update (0=minification filter, 1=magnification filter, 2=U wrapping mode, 3=V wrapping mode), and secondary register is what to update to. Minification filters include 0=nearest (point sampling), 1=linear (weighted average of neighbor texels), 2=nearest mipmap nearest (choose mipmap that most closely matches size of pixel being textured and use nearest sampling), 3=linear mipmap nearest (choose mipmap that most closely matches size of pixel being textured and use linear sampling), 4=nearest mipmap linear (choose two mipmaps that most closely match the size of the pixel being textured and use nearest sampling from both, and calculate weighted average of both samples for final texture value), 5=linear mipmap linear (choose two mipmaps that most closely match the size of the pixel being textured and use linear sampling for both, and calculate weighted average of both samples for final texture value). Magnification filters include 0=nearest (point sampling) and 1=linear (weighted average of neighbor texels). U and V wrapping modes include 0=clamp to edge (if out of bounds, clamp coordinate to nearest edge (coordinate will be 0 or 1), 1=mirrored repeat (if the integer part of the coordinate is even, the actual coordinate calculated is the fractional part, and if it's odd, it can be calculated as 1 - fractional part of coordinate), and 2=repeat (use only fractional part of coordinate). If the secondary register is out of the maximum range depending on what is being set, nothing will happen.

101 65 Dispatch work-groups for compute shader executions. The primary register specifies a 64-bit address of 3 32-bit unsigned integers for the X, Y, and Z, respectively, for the work-group dimensions. If any of the dimensions exceed their global work-group maximum, the product of all 3 dimensions exceeds the maximum global work-group size (see ***Hardware Information***), or any of the dimensions specified are 0, nothing will happen. This is recorded to the bound command buffer.

102 66 Outputs, to the output register, the memory address of the beginning of the hardware information.

103 67 Updates the bound segment table object. If the primary register is 0, this will create a new segment entry in the bound segment table and output the ID of the new segment entry to the output register. If the primary register is 1, this will update a segment entry in the bound segment table, and the secondary register is the 64-bit address to a 64-bit unsigned integer specifying the new starting virtual address, followed by a 64-bit unsigned integer specifying the new starting physical address, followed by a 64-bit unsigned integer specifying the new length for the segment, all followed by a 64-bit unsigned integer specifying the ID of the segment entry to update. If the primary register is 2, this will mark inactive the segment entry whose ID is specified by the secondary register (an inactive segment is ignored when reading a segment table, and resetting the entry to default state is the only way to make it active again). If the primary register is 3, this will output to the output register the starting virtual address of the segment entry whose ID is specified by the secondary register. If the primary register is 4, this will output to the output register the starting physical address of the segment entry whose ID is specified by the secondary register. If the primary register is 5, this will output to the output register the length of the segment entry whose ID is specified by the secondary register. If the primary register is 6, this will output to the output register the number of segment entries in the bound segment table object (including both active and inactive entries). If the primary register is 7, the bound segment table will be reset (clears all segment entries, and the first ID of the next created segment entry will be 0). If the primary register is 8, this will output to the output register 1 if the segment entry whose ID is specified by the secondary register is inactive, and 0 otherwise. If the primary register is 9, this will reset any segment entry whose ID is specified by the secondary register to the default segment state (active, and with length, virtual address, and physical address as 0). Segment entry IDs start at 0 and will increase by 1 for each new segment entry, which is active by default and with a length, virtual address, and physical address of 0 (when a segment table is reset, its entry ID counter is reset to 0). When segment entries have overlapping virtual addresses, the segment that will be used for mapping virtual addresses to physical addresses is undefined. If specifying a segment entry ID and the segment entry doesn't exist in the segment table object or it is marked inactive, nothing will happen unless the specified segment is inactive and the primary register is 8. If updating a segment entry and the specified new length is 0 or either of the new virtual or physical address ranges would exceed main memory, nothing will happen. If no segment table object is bound, the bound the segment table object is the one being used by this thread, or the primary register is > 9, nothing will happen.

104 68 Set 56 0s at the MSB end of the primary register.

105 69 Set 48 0s at the MSB end of the primary register.

106 6A Set 32 0s at the MSB end of the primary register.

107 6B Set 56 bits at the MSB end of the primary register to 1 if the MSB of the 8 bits at the LSB end is 1, and to 0 otherwise.

108 6C Set 48 bits at the MSB end of the primary register to 1 if the MSB of the 16 bits at the LSB end is 1, and to 0 otherwise.

109 6D Set 32 bits at the MSB end of the primary register to 1 if the MSB of the 32 bits at the LSB end is 1, and to 0 otherwise.

110 6E Toggle each bit of the primary register.
111 6F Toggle the MSB of the 32 bits at the LSB end of the primary register.
112 70 Toggle the MSB of the primary register.
113 71 64-bit integer increment by 1 of the primary register (result is stored in primary register).
114 72 64-bit integer decrement by 1 of the primary register (result is stored in primary register).
115 73 32-bit floating-point modulo of primary register by secondary register, and output the result to the output register. If the divisor is 0 or either operands are negative, NaN, or positive/negative infinity, the result is NaN.
116 74 64-bit floating-point modulo of primary register by secondary register, and output the result to the output register. If the divisor is 0 or either operands are negative, NaN, or positive/negative infinity, the result is NaN.
117 75 Calculates the result of a single variable math function and outputs the result to the output register. The primary register specifies the function to perform (0=tan, 1=sin, 2=cos, 3=arctan, 4=arcsin, 5=arccos, 6=tanh, 7=sinh, 8=cosh, 9=arctanh, 10=arcsinh, 11=arccosh, 12=natural logarithm, 13=base 10 logarithm, 14=floor, 15=ceiling, 16=absolute value, 17=integer absolute value) and the secondary register contains the 64-bit floating-point value, or 64-bit signed integer value if the primary register is 17, to perform the function on. The floor function will output the largest whole number less than or equal to the given value, and the ceiling function will output the smallest whole number greater than or equal to the given value, though each output as 64-bit floating-points. For all trigonometric functions, the angles input and output are in radians. If passing invalid/undefined values to these functions they will result in NaN. If the primary register is > 17, nothing will happen.
118 76 Make a copy of the current display or camera image beginning at the memory address specified by the secondary register. The primary register specifies where to capture the image from; either 0 to capture the current display or another value to specify the number of an available camera to capture from (see ***Image Capture***). If the primary register is > the number of available cameras, nothing will happen. At the provided memory address is the first pixel output; pixel array indices increase from left to right, starting at the top row, and each pixel is a 32-bit RGBA value; A being the byte at the lowest address with undefined value. If this thread does not have the necessary permission for the requested image capture, nothing will happen.
119 77 Output current time information to the output register. The primary register specifies what to output (0=the number of nanoseconds in wall-clock time elapsed since the launch of the runtime environment, 1=UTC number of seconds after the minute (integer values in range of [0,59], but up to 60 to account for leap seconds), 2=UTC minutes after the hour, 3=UTC hours since midnight (integer values in range [0,23]), 4=UTC day of month (integer values in range of [1,31], 5=UTC months since January (integer values in range of [0,11]), 6=UTC year, 7=UTC days since Sunday (integer values in range of [0,6]), 8=UTC days since January 1 (integer values in range of [0,365]), 9=whether specified UTC time is in Daylight Saving Time (0 if it is not, 1 if it is), 10=UTC number of milliseconds since midnight). The number of nanoseconds since the launch of the runtime environment must be measured as well as possible, with at least millisecond accuracy, and will reset to 0 on overflow. On failure, any one of these will set all bits of the output register to 1. If the primary register is > 10, nothing will happen.
120 78 Copies a region of memory to some location in memory. The primary register specifies the address of the first byte of the copied region, the secondary register specifies the number of bytes to copy, and the output register specifies the address of the first byte of the region that the copy is made at. The region that is copied to can overlap the original region and is an exact copy of the original. If the secondary register is 0, nothing will happen.
121 79 Get information from, or configure, audio sources and listeners. The primary register specifies what to do (0=set volume of bound audio source, 1=mute bound audio source for bound listener, 2=unmute bound audio source for bound listener, 3=bind a specified audio data object to the bound audio source, 4=get the 64-bit unsigned integer of number of samples of audio data for bound audio source, 5=get the 64-bit unsigned integer of number of samples into audio playback by bound audio source, 6=set the 64-bit unsigned integer of number of samples into audio playback by bound audio source, 7=set behavior for the end of playback for the bound audio source, 8=set X coordinate of bound audio source, 9=set Y coordinate of bound audio source, 10=set Z coordinate of bound audio source, 11=mute bound audio listener, 12=unmute bound audio listener, 13=set X coordinate of bound audio listener, 14=set Y coordinate of bound audio listener, 15=set Z coordinate of bound audio listener, 16=bind the bound audio occlusion geometry object to bound listener object at a specified binding point, 17=clear all audio occlusion geometry bindings of the bound audio listener, 18=set the position for a specified audio occlusion geometry binding point in a specified listener, 19=set the rotation for a specified audio occlusion geometry binding point in a specified listener, 20=set the sample rate of playback for the bound audio source, 21=get the sample rate of playback for the bound audio source), and the secondary register specifies data or an object ID. If the primary register is 0, the 4 bytes on the LSB end of the secondary register specify the 32-bit floating-point value to set the volume of the bound audio source object to; if this value is not within the range [0,100], nothing will happen. If the primary register is 3, the secondary register is the ID of

an audio data object and if the secondary register is 0 it has the effect of unbinding any audio data from the audio source. If getting audio information (the primary register is 4, 5, or 21), the requested information will be output to the output register. If the primary register is 6 and the secondary register specifies a value greater than the number of samples worth of audio data - 1, nothing will happen. If the primary register is 7, the secondary register can be either 0 to specify that the audio source will just play the audio once, or non-zero to specify that the audio source will loop. If setting a coordinate (the primary register is 8, 9, 10, 13, 14, or 15), the 4 bytes on the LSB end of the secondary register specify the 32-bit floating-point coordinate. If the primary register is 16, the secondary register must be a value less than the number of binding points per audio listener (see ***Hardware Information***) or else nothing will happen, and if the bound audio occlusion geometry object is the default object (ID 0) it has the effect of unbinding any audio occlusion geometry object currently at the specified binding point. If the primary register is 18 or 19, the secondary register is an address of a 64-bit unsigned integer specifying the audio listener object ID, followed by a 32-bit unsigned integer specifying the binding point, followed by 3 32-bit floating-point values which specify the X, followed by Y, followed by Z, position (if primary register is 18) or (if primary register is 19) a unit quaternion providing a rotation around X, Y, and Z axes. For cases where the primary register has a value of 18 or 19, if the audio listener object ID specified does not refer to a valid object, or if the binding point number provided is not less than the number of binding points per audio listener, nothing will happen. If the primary register is 19 and the quaternion specified is not a unit quaternion, nothing will happen. If the primary register is 20, the secondary register specifies the new sample rate for audio playback for the bound audio source. If the primary register is > 21, nothing will happen.

122 7A Get or set information related to audio data/files. Note audio samples are always numbered such that the first of them is sample 0, in both audio files and audio data objects. The sample count of audio is the number of samples found in each channel. Samples (in Hz) are provided and obtained from audio data as 32-bit signed integers specifying the samples of PCM audio data; when writing or reading audio data to/from a file, the sample values will be scaled to the correct range as required. If the primary register is 0, this is to load audio data from a file to the bound audio data object, clearing all data previously in the audio data object, and the secondary register is a 64-bit address to a 32-bit unsigned integer of the first sample, followed by a 32-bit unsigned integer of the last sample, followed by a 64-bit address to the ASCII string of the file path; if this file path is invalid or the file specified is not of a supported audio format, samples that don't exist in the file are requested, or if the number of the last sample is > the first sample, nothing will happen. If the primary register is 1, this is to output the sample count of a specified audio file to the output register, and the secondary register specifies a 64-bit address to the ASCII string of the file path; if this file path is invalid or the file specified is not of a supported audio format, nothing will happen. If the primary register is 2, this is to output the sample count of the bound audio data object to the output register. If the primary register is 3, this is to output the channel count of the bound audio data object to the output register. If the primary register is 4, this is to set an audio sample in a specified audio file at a specified channel (0 is left channel, 1 is right channel), and the secondary register specifies an address to a 32-bit unsigned integer for the sample number, followed by a 32-bit signed integer for the audio sample value, followed by a 16-bit unsigned integer specifying the channel, followed by a 64-bit address to the ASCII string of the file path; if this file path is invalid, the file specified is not of a supported audio format, the channel specified does not exist in the audio file, the sample does not exist in the audio file, or the channel integer + 1 is greater than the maximum number of audio channels, nothing will happen. If the primary register is 5, this is to output the frequency of a specified sample in the bound audio data object to the output register, and the secondary register is a 64-bit address to a 32-bit unsigned integer specifying the sample number, followed by a 16-bit unsigned integer specifying the channel; if the sample or channel specified does not exist in the audio data object, or the channel integer + 1 is greater than the maximum number of audio channels, nothing will happen. If the primary register is 6, this is to get the sample rate of the audio in the bound audio data object, which will be output to the output register. If the primary register is 7, this is to output the sample rate of the audio in a specified audio file to the output register, and the secondary register is a 64-bit address to the ASCII string of the file path; if this file path is invalid or the file specified is not of a supported audio format, nothing will happen. If the primary register is 8, this is to output the channel count of a specified audio file to the output register, and the secondary register is a 64-bit address to the ASCII string of the file path; if this file path is invalid or the file specified is not of a supported audio format, nothing will happen. If the primary register is 9, this is to set the sample rate, the sample count, and the channel count of a specified audio file, discarding all of the previous audio data in the file and having all audio channels' samples cleared to 0. The secondary register is a 64-bit address to a 32-bit unsigned integer specifying the new sample rate, followed by a 32-bit unsigned integer specifying the new sample count, followed by a 16-bit unsigned integer specifying the new channel count, followed by a 64-bit address to the ASCII string of the file path; if this file path is invalid, the channel count is greater than the maximum number of audio channels, or if the sample rate or either of the sample or channel count is 0, nothing will happen. If the file provided is not a valid audio file but the filename is of a supported audio format, blank

samples and channels will be created and the file's old data will be discarded; this is useful in the creation of new, properly formatted audio files. If the file provided is not a valid audio file and the filename is not of a supported audio format, nothing will happen. If the primary register is 10, this is to load the audio data in a section of the bound video data object into the bound audio data object, clearing all data previously in the audio data object, and the secondary register is a 64-bit address of a 32-bit unsigned integer of the first frame (note frame numbering starts at frame 0), followed by a 32-bit unsigned integer of the last frame (again, starts at 0); if the last frame is > the first frame, the video data object contains no audio data, or frames are referred to that do not exist in the video, nothing will happen. For sections of the video that have no audio, blank audio will be loaded into the bound audio data object. If the primary register is 11, this is to start/stop recording from one of the microphones into a specified audio file which will be created if it doesn't exist and overwritten if it does, and the secondary register is a 64-bit address to an 8-bit value of the number of the microphone (see ***Audio Recording***), followed by a 64-bit address to the ASCII string of the file path; if this file path is invalid, the file specified to record into does not have the extension of a supported audio format, the microphone specified does not exist, or the thread does not have audio recording permission, nothing will happen (note that the file path is ignored if this instruction is just stopping the audio recording of a microphone). If the primary register is 12, this is to output 1 if a specified microphone is currently recording audio to the output register and 0 if it's not, and the secondary register is the number of a microphone (see ***Audio Recording***); if the specified microphone does not exist, nothing will happen. If the primary register is > 12, or if accessing a file and this thread does not have permission to perform file I/O operations, nothing will happen.

123 7B Get or set information related to video data/files. Note video frames are always numbered such that the first of them is frame 0, in both video files and video data objects. An implementation that supports image file formats considers them as audioless video files that can be accessed through this instruction, and the frame rate and frame count of non-animated image files is always 0 and 1, respectively. If the primary register is 0, this is to load video data from a file to the bound video data object, clearing all data previously in the video data object and setting the output register to 1 if the file had an alpha channel and to 0 otherwise, and the secondary register is an address to a 32-bit unsigned integer of the first frame, followed by a 32-bit unsigned integer of the last frame, followed by a 64-bit address to the ASCII string of the file path; if this file path is invalid or the file specified is not of a supported video/image format, frames that don't exist in the file are requested, or if the number of the last frame is > the first frame, nothing will happen. If the primary register is 1, this is to output the frame count of a specified video file to the output register, and the secondary register is a 64-bit address to the ASCII string of the file path; if this file path is invalid or the file specified is not of a supported video/image format, nothing will happen. If the primary register is 2, this is to output the frame count of the bound video data object to the output register. If the primary register is 3, this is to get the resolution of the video in a specified video file, and the secondary register is a 64-bit address to a 32-bit unsigned integer that the output width will be written to, followed by a 32-bit unsigned integer that the output height will be written to, followed by a 64-bit address to the ASCII string of the file path; if this file path is invalid or the file specified is not of a supported video/image format, nothing will happen. If the primary register is 4, this is to get the resolution of video in the bound video data object, and the secondary register is a 64-bit address to a 32-bit unsigned integer of the output width, followed by a 32-bit unsigned integer of the output height. If the primary register is 5, this is to get the frame rate of the video in the bound video data object, which will be output to the output register. If the primary register is 6, this is to output the frame rate of the video in a specified video file to the output register, and the secondary register is a 64-bit address to the ASCII string of the file path; if this file path is invalid or the file specified is not of a supported video/image format, nothing will happen. If the primary register is 7, this is to load a frame of the bound video data object to memory in an 8-bit per RGBA component format (note that the value of the A component is 0 for video that was loaded from a file that lacked an alpha channel), and the secondary register is a 64-bit address to a 32-bit unsigned integer of the frame number, followed by a 64-bit address for the beginning of the output pixel array (indices increase from left to right, starting at the top row, and each pixel is a 32-bit RGBA value; A being the byte at the lowest address); if the frame does not exist in the video, nothing will happen. If the primary register is 8, this is to overwrite a frame of the video in a specified video file, and the secondary register is a 64-bit address to a 32-bit unsigned integer of the frame number, followed by a 64-bit address to the pixel array (same format and layout as in loading video frames), followed by a 64-bit address to the ASCII string of the file path; if this file path is invalid, the file specified is not of a supported video/image format, or the frame specified does not exist in the video, nothing will happen. If the primary register is 9, this is to set the frame count and frame rate of a specified video file, removing frames from the end of the video if the new frame count is less than the count of frames already in the video or adding blank frames to the end of the video if the frame count is greater than what it was, and the secondary register is a 64-bit address to a 32-bit unsigned integer specifying the new frame rate, followed by a 32-bit unsigned integer specifying the new frame count, followed by 2 32-bit unsigned integers specifying video resolution (width followed by height), followed by a 64-bit

address to the ASCII string of the file path; if this file path is invalid, the resolution does not match that of any frames that already exist in the file, the width is 0, the height is 0, or the frame count is 0, nothing will happen. If the file provided is not a valid video file but the filename is of a supported video/image format, blank frames will be created and the file's old contents will be discarded; this is useful in the creation of new, properly formatted video files. If a file of a non-animated image format is provided and the specified frame rate and frame count are not 0 and 1 respectively, nothing will happen. Otherwise, if a file that is not of a non-animated image format is provided and the frame rate is 0, nothing will happen. If the file provided is not a valid video file and the filename is not of a supported video/image format, nothing will happen. If the primary register is > 9, or if accessing a video file and this thread does not have permission to perform file I/O operations, nothing will happen.

124 7C Networking instruction. See the ***Networking*** section for more information on instruction behavior. If the primary register is 0, the secondary register is an address to a byte specifying whether to get the IPv4 (if byte is 0) or IPv6 (if byte is non-zero) address of the hostname named by the ASCII string beginning after that byte, and following the ASCII string is an error status byte (which is set to 1 if there was an error and the IP could not be obtained and set to 0 otherwise) which is followed by either 4 or 16 bytes of memory that will be set to the obtained 4 or 16 byte IP address. If the primary register is 1 this will get the public IPv4 address of this system; the secondary register is an address to an error status byte (which is set to 1 if there was an error and the IP could not be obtained and set to 0 otherwise), followed by 4 bytes of memory that will be set to the obtained IPv4 address of the system. If the primary register is 2 this will get the public IPv6 address of this system; the secondary register is an address to an error status byte (which is set to 1 if there was an error and the IP could not be obtained and set to 0 otherwise), followed by 16 bytes of memory that will be set to the obtained IPv6 address of the system. If the primary register is 3 this will get the hostname of a given IP; the secondary register is an address to a byte specifying whether the bytes following gives a IPv4 (byte is 0) or IPv6 address (byte is non-zero), followed by an error status byte (set to 1 if the hostname could not be obtained and 0 otherwise), followed by memory that will be set to the obtained ASCII string of the hostname for the given IP (only if the hostname could be obtained). If the primary register is 4, this will get the system's hostname; the secondary register is an address to an error status byte (set to 1 if the hostname could not be obtained and 0 otherwise), followed by memory that will be set to the obtained ASCII string of the system's hostname (only if the hostname could be obtained). If the primary register is 5 this will get the length of a hostname's string given IP; the secondary register is an address to a byte specifying whether the bytes following gives a IPv4 (byte is 0) or IPv6 address (byte is non-zero), followed by an error status byte (set to 1 if the hostname length could not be obtained and 0 otherwise), followed by a 32-bit unsigned integer which is set to the length of the string of the hostname for the given IP (size, in bytes, including the null character; set to 0 if hostname could not be obtained). If the primary register is 6 this will get the length of the system's hostname string; the secondary register is an address to an error status byte (set to 1 if the hostname length could not be obtained and 0 otherwise), followed by a 32-bit unsigned integer which is set to the length of the system's hostname (size, in bytes, including the null character; set to 0 if the hostname could not be obtained). If the primary register is 7 this will measure the ping for an IP; the secondary register is an address to a byte specifying whether the bytes following gives a IPv4 or IPv6 address, following the IP address is a 32-bit unsigned integer of the time for timeout (in milliseconds), followed by an error status byte (set to 0 if the IP was reached, 1 if the IP could not be reached), followed by a 32-bit unsigned integer which is set to the time (in milliseconds) it took to reach the IP address after the operation is complete (left untouched if there was an error). If the primary register is 8 this will send data from a socket; the secondary register is an address to a 64-bit unsigned integer specifying the ID of the socket object to send data from, followed by a 64-bit address of the data to send, followed by a 32-bit unsigned integer which is the length of the data to send (in bytes), followed by (if sending from a UDP socket) a 16-bit unsigned integer which is the destination port, followed by (if sending from a UDP socket) a byte specifying whether the destination IP is IPv4 (byte is 0) or IPv6 (byte is non-zero), followed by (if sending from a UDP socket) 4 or 16 bytes for the destination IP address, followed by an error status byte (set to 1 at the end of the operation if there was an error sending the data or to 0 otherwise). If sending from a non-connected TCP socket, nothing will happen. If the primary register is 9 this will receive data through a socket; the secondary register is an address to a 64-bit unsigned integer specifying the ID of the socket object to receive data from, followed by a 64-bit address which is where to write the received data to, followed by a 32-bit unsigned integer specifying the amount of data (in bytes) to receive (if receiving through a UDP socket the entire packet must be read at once, otherwise the excess bytes will be discarded, the rest of the data in the packet is lost, and the error status byte is set to 1 - if receiving through a TCP socket the excess bytes will be the first ones received next time), followed by (if receiving through a UDP socket) a 64-bit address of memory set to the sender's information (a byte which will be set to 0 if its IP is IPv4 and 1 if its IP is IPv6, followed by 4 or 16 bytes set to the sender's IP address), followed by (if receiving through a TCP socket) a 32-bit unsigned integer which will be set to the number of bytes that were received and written to memory, all followed by an error status byte (set to 1 at the end of the operation if there was an error receiving the

data or to 0 otherwise). If receiving from a non-connected TCP socket, nothing will happen. If the primary register is 10 this will mark a TCP socket as listening; the secondary register is an address to a 64-bit unsigned integer specifying the ID of a TCP socket object to mark as listening on its port (allowing it to start accepting pending connection requests), followed by an error status byte (set to 0 if there was no error and 1 otherwise). If the secondary register is not a TCP socket object, the port already has a socket listening on it, or the TCP socket specified is already connected, listening, or closed, nothing will happen. If the primary register is 11, the secondary register is a 64-bit address to a 64-bit unsigned integer specifying the ID of a listening TCP socket object which a pending connection request will be accepted from, followed by an error status byte (set to 0 if there was no error and 1 otherwise; with no updates to the data following if there was an error), followed by a 64-bit unsigned integer which will be set to the ID of a new TCP socket object which has been automatically generated that is connected to the accepted connection, followed by a 16-bit unsigned integer to which the port of the accepted connection is output, followed by a byte which is set to 0 if the accepted connection is IPv4 or 1 if the accepted connection is IPv6, followed by 4 or 16 bytes which is set to the IP address of the accepted connection. If the secondary register is not a listening TCP socket object, nothing will happen. If the primary register is 12, the secondary register is a 64-bit address to a 64-bit unsigned integer of the ID of the TCP socket to request a connection for (this is the client), followed by a 16-bit unsigned integer specifying the server's port number, followed by a byte specifying whether the server's IP is IPv4 (byte is 0) or IPv6 (byte is non-zero), followed by 4 or 16 bytes for the IP address of the server to request connection to, followed by an error status byte (set to 0 if there was no error and 1 otherwise). If the TCP socket is connected, listening, or closed, nothing will happen. If the primary register is 13, the secondary register is a 64-bit address to a 64-bit unsigned integer of the ID of the TCP socket to close, followed by an error status byte (set to 0 if there was an error and 1 otherwise). If the TCP socket is not connected, nothing will happen. If the primary register is > 13 or this thread does not have networking permissions, nothing will happen.
125 7D Stores value(s) in this thread's left operand SIMD vector (see ***SIMD Operations***). The primary register specifies the number of bytes to store starting at the beginning of the vector, and the secondary register is a memory address to the values to store in the vector. If the primary register is 0 or > 32, nothing will happen
126 7E Stores value(s) in this thread's right operand SIMD vector (see ***SIMD Operations***). The primary register specifies the number of bytes to store starting at the beginning of the vector, and the secondary register is a memory address to the values to store in the vector. If the primary register is 0 or > 32, nothing will happen
127 7F Perform a SIMD operation. The byte at the LSB end of the primary register has value and meaning described under ***SIMD Operations***, and the secondary register is the memory address for the location to output the results. This instruction behaves as described in the ***SIMD Operations*** section.

Shader Bytecode

This is essentially a high-level shading language, but in a bytecode format. The instructions are of varying lengths.

Data type names use prefix u- for unsigned integer and i- for signed integer.

Floating-point numbers in shaders are single-precision (32-bit), and signed/unsigned integers are 32-bit.

A sampler contains the state for texture filtering modes to use and the texture to sample from during texture sampling. If the sampler does not refer to a valid texture, sampling from it will result in undefined values.
In unsigned and signed integer textures, usampler and isampler are used, respectively, to sample the texture as-is.
In floating-point textures, sampler is used to sample the texture as-is.
In fixed-point textures, sampler is used to sample the floating-points in range [0,1] and everything output to one will be in the same range.
Sampling an integer texture with any linear minification or magnification filtering will have undefined results.
Sampling a texture using mipmap filtering when the levels for that texture (from level 0 upward) are not all half the dimensions of the texture level before it (truncated division by 2, minimum of 1, until the texture is 1x1), will have undefined results.
When sampling from a texture that's not using mipmap filtering without manually specifying the level-of-detail, level 0 will be sampled. When sampling from a texture that is using mipmap filtering without manually specifying the level-of-detail, mipmapping is performed for the texture sampling as usual.
Sampling using a type of sampler incompatible with the type of texture being sampled will have undefined results.

An image allows for read and write access to a single texture level of some texture from within compute or ray tracing shaders, and each image defined in a shader is of a specified image format (one of the texture formats). If the texture

format of the texture bound to the image descriptor at the image descriptor binding point is not the same as the image format, attempting to read or write to the image will have undefined results.

Code is read from top to bottom (from the beginning of the shader's binary to the end), and if an identifier is used before the definition of the identifier, the shader is invalid. Any uses of an identifier under an incorrect type where it is not an operand of assignment or arithmetic with a type that can be typecast to the correct one will also cause the shader to be invalid; conditional evaluations also get an exception from these rules and their type rules are specified in the definition of conditionals. If attempting to use a sampler, isampler, or usampler for anything other than getting texture dimensions or performing texture sampling, attempting to use an image for anything other than getting image dimensions or performing an image read/write operation, or attempting to use an acceleration structure for anything other than specifying the acceleration structure for the shader function to trace a ray, nothing will happen. Identifiers are defined for functions, variables, uniforms, attributes, ray attributes, parameters, and local variables.

Output attributes are used in pixel shaders to provide final color values to the color attachments of the bound framebuffer with color attachment IDs equal to their location ID (e.g. the first color attachment, color attachment 0, gets color from output attribute at location 0, and starting from location 0 there are as many consecutive locations used up for color attachments as there are accessible color attachments in the current pipeline). These outputs should be vector types if the color attachment has more than one component per pixel or scalar types (floating-point or signed/unsigned integer) if the color attachment only has one component per pixel. All values output to color components of fixed-point attachments are clamped to range [0,1]. Output attribute vectors/scalars should be floating-point if outputting to a fixed-point or floating-point type color attachment, signed integer if outputting to a signed integer color attachment (each color component is clamped to range [-128,127]), or unsigned integer if outputting to an unsigned integer color attachment (each color component is clamped to range [0,255]), and they should have a component count which is equal to the number of components per pixel of the color attachment or else results are undefined.

All depth outputs are clamped to [0,1].

Sampling a texture from a dataless texture level (through either a sampler or image) will have undefined results.

All descriptor-provided data is initially undefined, but supplied the through the binding of descriptors.

Vertex attributes are attributes loaded to the vertex shader on a per-vertex basis from the vertex buffer object being rendered, from positions described by the vertex attribute layout information in the VAO for the bound rasterization pipeline.

Vector types which lack the u- or i- prefixes are composed of floating-point values.

In arithmetic and assignment, operands or assigned values which are integers (int/uint/ivec*/uvec*) will be converted to a floating-point type (float/vec*) if any of the operands or the data to output the result to is of a floating-point type (float/vector/matrix) before the operation is performed. Before a scalar/vector resulting from an operation is output to the whatever data is specified to receive the result, it is typecast to the type of data to which it is output. Note that samplers, images, acceleration structures, entire vectors, and entire matrices cannot be converted into a scalar type, or vice versa, and any shaders attempting to do so will be invalid.

For all trigonometric functions the angles provided or output are in radians. If the value provided for any of these functions is undefined for the function, the resulting value is undefined. The floor operation will get largest integer number less than or equal to the given value, and the ceiling operation will get smallest integer number greater than or equal to the given value; both of these operations are performed on and result in floating-point numbers.

Each gray-highlighted section of text present in the shader bytecode specification defines a shader instruction, and each region between bars (|) is referred to as a token. If a token is surrounded by square brackets, it means that it may be omitted from the instruction depending on the context (when it can be omitted and when it can't will be made clear in the definition of the instruction). If any instruction is found to have a token outside of a valid range the shader will be invalid; e.g., if an instruction accesses the Z or W component in a vec2. Tokens consisting of more than one byte are in

little-endian byte ordering.

All array indices in shader instructions, represented in a shader instruction's definition by an "index" token, have varying lengths depending on the way that the index is provided. Array indices can be constants, uniform-provided values, use the current instance ID (only allowed in vertex shaders), use the current loop iteration, or, if it is in a ray tracing shader, the array index can be provided by the value of a uint variable; the array index can also be provided by a uint variable in the vertex shader but only when accessing uniform arrays. Constant indices are 16 bits wide, and the 16-bit constant value will be less than 65533 for a constant-provided index. If the constant value is 65533, the index is provided by the instance ID; the 16-bit value following the constant value specifies a multiplier (0 corresponds to 1), and the 32-bit value following that is a 32-bit signed integer specifying an offset; altogether this forms the array index expression instance_ID*multiplier+offset. If the constant value is 65534, the index is uniform-provided; the 16-bit value following the constant value specifies the identifier of a uniform providing the index (which must be a uint and must contain only 1 array element), the 16-bit value following that will specify a multiplier (0 corresponds to 1), and the 32-bit value following that is a 32-bit signed integer specifying an offset; altogether this forms the array index expression uniform*multiplier+offset. If the constant value is 65535, the index is 32 bits wide; the 16-bit value following the constant value is either 65535 (specifies to use the current loop iteration as index) or it specifies the identifier of a uint variable providing the index (which must contain only 1 array element). For indices using the current loop iteration, the first iteration of the loop the instruction is located in will be index 0 and for each iteration of the loop the instruction is located in the index will increase by 1; if an instruction using a loop iteration index is not located within a loop the shader is invalid. If, in any instruction, array access via a loop iteration index is used but the maximum number of iterations the loop can run (given in loop opener) is greater than the number of elements in an accessed array of a defined size, the shader is invalid.

Variable arrays at the end of storage blocks, and sampler arrays in ray tracing shaders, can be unsized. Since the array size is unknown and ray tracing shaders can use fully dynamic (unpredictable) array indices, the shader compiler cannot check if invalid array indices will be accessed, but results are undefined when accessing array indices outside of the bounds of any array.

Any shaders that have more than one occurrence of the same attribute/payload locations, identifiers, or set and binding pairs being defined (across the entire shader for all uniforms, uniform blocks, storage blocks, globally defined variables, functions, attributes, or ray attributes) are invalid. Shaders in which any of the functions reuse identifiers across local variable and parameter definitions (note that parameters are essentially local variables) within the same function scope are invalid, and shaders using identifiers for local variables and parameter definitions when any of the identifiers were defined globally at any point before the function definition (or if they were used as the function's identifier) are invalid as well.

Global scope is said to be at level 0; functions begin on level 1, and each level above that begins/ends with the opener/closer of a conditional branch or loop. There can be 7 levels nested within a function, and the shader is invalid if any loop or conditional openers are present on level 8. Variable definitions for local variables can only be at level 1 of a function.

For the representation of constants, integers are 32-bit, signed integers are represented in two's complement, and floating-point values are in 32-bit IEEE 754 format.

An attribute is data input or output from or to a vertex or pixel shader, such as vertex attributes to the vertex shader or color outputs from the pixel shader, which is not uniform data. Attributes that are output from the vertex shader and input to the pixel shader are linked by identifiers. Ray attributes are different from attributes and are defined in ray tracing shaders in ray payload or incoming ray payload blocks.

The attribute location IDs of vertex shader inputs and pixel shader outputs are not related; e.g., location 0 could be defined for vertex position input and then defined as pixel color output to location 0 (color attachment 0). Vertex shader outputs and pixel shader inputs do not have location IDs, since they are linked by matching identifiers.

In instruction definitions, hexadecimal numbers appearing inside or outside of brackets specify the values of the opcode bytes.

Only definitions for variables, uniforms, attributes, ray attributes, and functions, openers for uniform, push constant, storage, ray payload, and incoming ray payload blocks, the beginning of the main function, and closers for uniform, push constant, storage, ray payload, and incoming ray payload blocks can occur in global scope. Identifiers must only be used after their definition.

In compute and ray tracing shaders, no input/output attributes may be defined.

All variables, ray attributes, and uniforms are considered arrays with an element count of at least 1.

Variables defined within a storage block will read/write whichever data is located at their position within the SBO referred to by the storage descriptor at the storage block's binding point. Uniforms defined within a uniform block will read from whichever data is located at their position within the UBO referred to by the uniform descriptor at the uniform block's binding point, or access the sampler/image/acceleration structure referred to by the sampler/image/acceleration structure descriptor at that point if the uniform block defines a sampler type, image, or acceleration structure. Uniforms defined within a push constant block will be read from the pipeline's push constant data.

Due to the way that GPUs handle writing operations in shaders, writing operations to images or storage buffers performed by a shader invocation are only guaranteed to be visible to that shader invocation, and only if the data is read through the exact same identifier used for the write operation. After a barrier instruction in compute shaders, the effects of all writing operations that took place prior to the barrier will become fully visible across all shader invocations in the work-group. All writing operations performed by a shader invocation will be completed at the end of the shader invocation. In pixel shaders, sampling from a texture that is being written to as part of the FBO will have undefined results.

Any shader that repeats the same pair of set and binding numbers for any uniform/storage block openers is invalid (this includes acceleration structures, images, and samplers; since they are defined as single-uniform uniform blocks). Any shader that repeats the same location ID for any attribute/ray payload/incoming ray payload definitions is invalid.

The first 12 bytes in a compute shader are 3 32-bit unsigned integers specifying the X, Y, and then Z for the local work-group dimensions (the product of them being the local-work group size, which is the number of work-items to run per work-group). If any of these dimensions are 0, exceed their maximum local work-group dimension limit (see ***Hardware Information***), or the product of all 3 dimensions exceeds the maximum local work-group size (see ***Hardware Information***), shader is invalid.

Any shader attempting to modify an input attribute or uniform is invalid.

VARIABLE, ATTRIBUTE, RAY ATTRIBUTE, AND UNIFORM DEFINITIONS

`| 8-bit opcode | 8-bit extra info #1 | 16-bit identifier | [16-bit extra info #2] |`
A variable, attribute, ray attribute, or uniform definition. Their initial values are undefined.
Valid opcodes include:
*! input attribute (00)
*! flat interpolation output attribute (01)
*! smooth interpolation output attribute (02)
*! noperspective interpolation output attribute (03)
^. uniform (04)
^ variable (05)
^: ray attribute (06)

Attribute interpolation modes (smooth, flat, and noperspective) have meaning only on attributes output from vertex to pixel shaders and affect how the values are interpolated before input to the pixel shader. flat is for the absence of interpolation (pixel shader input attribute values will be the values set in the vertex shader invocation for the primitive's first specified vertex), noperspective is for linear interpolation and smooth is for perspective-correct interpolation. If any integer typed output attributes (int/uint/ivec*/uvec*) are defined without the flat interpolation mode in a vertex shader,

the vertex shader is invalid.

The extra info 1 is expected to be:
a type (0=vec2, 1=vec3, 2=vec4, 3=ivec2, 4=ivec3, 5=ivec4, 6=uvec2, 7=uvec3, 8=uvec4, 9=mat2x2, 10=mat2x3, 11=mat2x4, 12=mat3x2, 13=mat3x3, 14=mat3x4, 15=mat4x2, 16=mat4x3, 17=mat4x4, 18=float, 19=signed integer, 20=unsigned integer, 21=sampler, 22=isampler, 23=usampler, 24=image, 25=acceleration structure)

The extra info 2 is expected in those with a * next to them to be:
a number for the attribute location ID (only present for vertex shader input attributes or pixel shader output attributes)

The extra info 2 is expected in those with a ^ next to them to be:
an array element count. If this is an acceleration structure uniform definition, this must be 1 or else the shader is invalid. If this is a sampler uniform definition in a vertex, pixel, or compute shader, this must be 1 or else the shader is invalid. If this is an image uniform definition, this instead specifies the image's format and must specify one of the texture formats (see ***Texture Formats***), or else the shader is invalid. This can only be 0 if in variable definition when it's the last variable defined in a storage block (which defines an unsized variable array), or if it's in sampler uniform definition in ray tracing shaders (which defines an unsized sampler array).

Those with ! next to them must occur only in global scope.
Those with . next to them must occur only in uniform or push constant blocks.
Those with : next to them must occur only in ray payload or incoming ray payload blocks.

Uniform definitions are the only instructions that can occur within a uniform block, and uniform blocks must contain at least one uniform definition, or else the shader is invalid.
Only uniforms defined in uniform blocks can be of sampler, image, or acceleration structure types, and such a uniform can only be defined in uniform blocks that have no uniform definitions other than it. Samplers are provided by sampler descriptor bindings, images are provided by image descriptor bindings, and acceleration structures are provided by acceleration structure descriptor bindings. Shaders which define any image uniforms that are not compute or ray tracing shaders are invalid. Shaders which define any acceleration structure uniforms that are not ray tracing shaders are invalid. If an any-hit shader defines an acceleration structure uniform, the shader is invalid.

`| 8-bit opcode | 8-bit extra info #1 | 32-bit extra info #2 |`
Opens a uniform block. Extra info 1 is the set binding number, and extra info 2 is the descriptor binding point number. opcode is (hex) 07.

`| 8-bit opcode |`
Closes a uniform block. opcode is (hex) 08.
Uniform definitions are the only instructions that can occur within a push constant block, and push constant blocks must contain at least one uniform definition, or else the shader is invalid.

`| 8-bit opcode |`
Opens a push constant block. If there is more than one of these in a shader, the shader is invalid. A shader does not need to have a push constant block. opcode is (hex) 09.

`| 8-bit opcode |`
Closes a push constant block. opcode is (hex) 08.

Variable definitions are the only instructions that can occur within a storage block, and storage blocks must contain at least one variable definition, or else the shader is invalid.

`| 8-bit opcode | 8-bit extra info #1 | 32-bit extra info #2 |`
Opens a storage block. Extra info 1 is the set binding number, and extra info 2 is the descriptor binding point number. Can only be present in compute, ray generation, closest-hit, any-hit, or miss shaders. opcode is (hex) 0A.

| 8-bit opcode |
Closes a storage block. opcode is (hex) 08.

Ray attributes may only be defined within ray payload or incoming ray payload blocks, and within them there must be at least one ray attribute defined, or else the shader is invalid. Ray payloads and incoming ray payloads can only be present in certain ray tracing shaders. See ***Ray Payloads***.

| 8-bit opcode | 16-bit extra info |
Opens a ray payload block. Extra info is the location ID of the ray payload. Can only be present in ray generation, closest-hit, or miss shaders. opcode is (hex) 0B.

| 8-bit opcode |
Closes a ray payload block. opcode is (hex) 08.

| 8-bit opcode | 16-bit extra info |
Opens an incoming ray payload block. Extra info is the location ID of the ray payload that this will access, as provided to the trace function that created this ray. If there is more than one of these in a shader, the shader is invalid. Can only be present in closest-hit, any-hit, or miss shaders. opcode is (hex) 0C.

| 8-bit opcode |
Closes an incoming ray payload block. opcode is (hex) 08.

MATRICES

Matrices are composed of ***width*** *x* ***height*** elements of floating-points. As an example, a mat3x4 will be 3 elements in width and 4 elements in height. They are in column-major format; each column is a vector with lowest index at top and highest index at the bottom. Index 0 is at the top left and the highest index is at the bottom right of the matrix. As an example, below is illustrated a 4x4 matrix and the indices of each matrix element.

0 4 8 12
1 5 9 13
2 6 10 14
3 7 11 15

If a uniform or variable is a matrix type or an array of matrices, each matrix element would be read from buffer memory with index 0 of each matrix being closer (when compared to the other matrix elements in the matrix) to the beginning of the uniform/push constant/storage block's data.

FUNCTION CALLS AND DEFINITIONS

All instructions not present in a function body are in the global scope. All within a function body are within the function's local scope. Function definitions must not occur within a function, or else the shader is invalid.

> Parameter: a value or reference to data a function is defined to take in and/or output
> Argument: the value or reference to data given for a parameter at time of function call

A function definition looks like the following:
| 8-bit opcode #1 | 16-bit identifier | parameter definitions | 8-bit opcode #2 | ... | 8-bit opcode #3 |
The first opcode is function definition (0D).
The second opcode is the function's body opener (0E).
The third opcode is the function's body closer (08).
The identifier is the identifier for the function.
The parameter definitions is the list of parameter definitions, if there are any.
The ... is the function body (instructions to run when the function is called).

A single parameter definition looks like the following:
| 8-bit opcode | 8-bit extra info #1 | 16-bit identifier | 16-bit extra info #2 |
The opcode is an in (0F), out (10), or in+out (11) parameter definition.
The identifier is the parameter's identifier; parameters effectively define variables local to the function.
The extra info #1 is the type of the parmeter (arguments must be of the same type; 0=vec2, 1=vec3, 2=vec4, 3=ivec2, 4=ivec3, 5=ivec4, 6=uvec2, 7=uvec3, 8=uvec4, 9=mat2x2, 10=mat2x3, 11=mat2x4, 12=mat3x2, 13=mat3x3, 14=mat3x4, 15=mat4x2, 16=mat4x3, 17=mat4x4, 18=float, 19=signed integer, 20=unsigned integer).
The extra info #2 is an array element count (arguments provided in function calls must have been defined with the same element count; if 0, the shader is invalid).

Input parameters take in copies of values from the variable the caller passed as argument, output parameters output values to the variable the caller passed as argument (the output is undefined unless the parameter is assigned within the function), and input/output parameters take in copies of values from the variable the caller passed as argument and then output values to the variable the caller passed as argument (the output is undefined unless the parameter is assigned within the function). A function's parameters are simply treated as local variable definitions, and so they can be both read and written to within the function regardless of whether it is defined as an in/out/in+out parameter.

`| 8-bit opcode | 16-bit identifier | arguments |`
Calls a function; the identifier states the function to call, and arguments is the list of arguments (must be variables; this must be as many contiguous 16-bit identifiers as defined parameters, and each must be of the same element count and type as the parameter). Can only be present in the main function. opcode is (hex) 12.

`| 8-bit opcode |`
Returns from any function. Can only be present within a function (including the main function). opcode is (hex) 13.

`| 8-bit opcode |`
Start the main function; everything after this instruction will be considered to be within the main function's body and be treated accordingly (e.g., function definitions may not occur after this, variables defined will be local variables, etc.). A shader must have one (and only one) occurrence of this instruction, and it can only be present in global scope. opcode is (hex) 14.

ARITHMETIC

Vector operations:
`| 8-bit opcode | 16-bit identifier | [index] |`
The identifier is the name of the vector to perform the operation on (variable, attribute, or ray attribute), and the index is always and only present for variables for indexing into variable or ray attribute arrays. If the variable, attribute, or ray attribute isn't of a vector type, the shader is invalid.
The opcode can be one the following, but only when the vector is not an unsigned integer type:
negate all components (15)
absolute value of all components (16)
Or any one of the following to perform the operation on each component, but only when the vector is floating-point:
normalize the vector (17)
floor (18)
ceiling (19)
tan (1A)
sin (1B)
cos (1C)
arctan (1D)
arcsin (1E)
arccos (1F)
tanh (20)
sinh (21)
cosh (22)
arctanh (23)

arcsinh (24)
arccosh (25)
natural logarithm (26)
base 2 logarithm (27)

Matrix operations:
| 8-bit opcode | 16-bit identifier #1 | [index #1] | 16-bit identifier #2 | [index #2] |
The first identifier identifies where the output of the operation will be assigned (variable, attribute, or ray attribute), and the second identifier is the name of the matrix to perform the operation on (variable, attribute, ray attribute, or uniform). The indices are always and only present for variables, ray attributes, and uniforms for indexing into variable, ray attribute, or uniform arrays. If the second identifier isn't of a matrix type, the shader is invalid. The output must be of a matrix type where the dimensions are those of the input's swapped if getting the transpose of a matrix, of type float if getting the determinant of a matrix, and the same matrix type as the input (second identifier) if getting the inverse of a matrix. If the operation is inverse and the matrix dimensions of the input and output are not equal to each other, the shader is invalid. If the operation is inverse and the input matrix does not have an inverse (it is singular), then the output matrix will have undefined values.
The opcode can be one of:
inverse (28)
determinant (29)
transpose (2A)

| 8-bit opcode | 16-bit identifier #1 | [index #1] | [8 bits for vector/matrix element #1] | (output for result)
| 16-bit identifier #2 | [index #2] | [8 bits for vector/matrix element #2] | (left operand)
| 16-bit identifier #3 | [index #3] | [8 bits for vector/matrix element #3] | (right operand)
The first identifier is that of a variable, attribute, or ray attribute to output the result to, the second identifier is that of a variable, attribute, ray attribute, or uniform for use as the left operand, and the third identifier is that of a variable, attribute, ray attribute, or uniform for use as the right operand. Index 1 is always and only present for variables and ray attributes for indexing into variable or ray attribute arrays. Indices 2 and 3 are always and only present for variables, ray attributes, and uniforms for indexing into variable, ray attribute, or uniform arrays. The 8 bits for matrix elements are always and only present for matrix types; if the matrix being used does not have enough elements the shader is invalid. The 8 bits for vector elements are always and only present for vector types; its value specifying the X, Y, Z, or W element to use (0, 1, 2, or 3, respectively). If the 8 bits for vector element is > the number of elements in the vector - 1, the entire vector will be used. All entire vectors must have the same number of elements as the others; the right operand (second identifier) can only be an entire vector if the left operand (third identifier) is as well, and the left operand can only be an entire vector if the output is as well. If the left and right operands are scalars but the output is to an entire vector, the result of the operation will be assigned to all elements of that vector. If the left operand is an entire vector and the right operand is a scalar, the operation is performed on each component of the left operand with the scalar as the right operand and the resulting vector is assigned to the vector being output to. If the left and right operand are both entire vectors, the operation is performed componentwise and the resulting vector is assigned to the vector being output to.
The opcode can be one of:
add (2B)
multiply (2C)
divide (2D)
subtract (2E)
raise to the power of (2F)

| 8-bit opcode | 16-bit identifier #1 | [index #1] | [8 bits for vector/matrix element #1] | (output for result)
| 16-bit identifier #2 | [index #2] | [8 bits for vector/matrix element #2] | (left operand)
| 32 bits for value | (right operand)
The first identifier is that of a variable, attribute, or ray attribute to output the result to, the second identifier is that of a variable, attribute, ray attribute, or uniform for use as the left operand. The 32 bits for value is the constant for the right operand. The constant is given as the same data type as the left operand (or as the same data type as each vector/matrix element if the left operand is a vector or matrix type). Index 1 is always and only present for variables and ray attributes for indexing into variable or ray attribute arrays. Index 2 is always and only present for variables, ray attributes, and uniforms for indexing into variable, ray attribute, or uniform arrays. The 8 bits for matrix elements are

always and only present for matrix types; if the matrix being used does not have enough elements the shader is invalid. The 8 bits for vector element is always and only present for vector types; its value specifying the X, Y, Z, or W element to use (0, 1, 2, or 3, respectively). If the 8 bits for vector element is > the number of elements in the vector - 1, the entire vector will be used. The left operand can only be an entire vector if the output is as well, and both must have the same number of elements; the operation will be performed on each component of the left operand with the constant as the right operand and the result vector will be assigned to the vector being output to. If the left operand is a scalar but the output is to an entire vector, the result of the operation will be assigned to all elements of that vector.
add (30)
multiply (31)
divide (32)
subtract (33)
raise to the power of (34)
The operation can also be performed with swapped operands (left operand after the operator and right operand before the operator, allowing e.g., X=5-X):
add with swapped operands (35)
multiply with swapped operands (36)
divide with swapped operands (37)
subtract with swapped operands (38)
raise to the power of with swapped operands (39)

Scalar operations:
| 8-bit opcode | 16-bit identifier | [index] | [8 bits for vector/matrix element] |
The identifier is that of a variable, attribute, or ray attribute to modify, and the index is always and only present for variables and ray attributes for indexing into variable or ray attribute arrays. The 8 bits for vector element is always and only present for vector types; its value specifying the X, Y, Z, or W element to affect (0, 1, 2, or 3, respectively); if the vector does not have enough elements the shader is invalid. The 8 bits for matrix element is always and only present for matrix types; if the matrix being used does not have enough elements the shader is invalid.
The opcode can be one of the following if it's a signed integer or floating-point:
negate (3A)
absolute value (3B)
Or any one of the following, but only when the scalar is floating-point:
reciprocate (3C)
floor (3D)
ceiling (3E)
tan (3F)
sin (40)
cos (41)
arctan (42)
arcsin (43)
arccos (44)
tanh (45)
sinh (46)
cosh (47)
arctanh (48)
arcsinh (49)
arccosh (4A)
natural logarithm (4B)
base 2 logarithm (4C)

Vector-vector operations:
8-bit opcode	16-bit identifier #1	[index #1]
16-bit identifier #2	[index #2]
16-bit identifier #3	[index #3]
The first identifier is the vector or float to output the result to (variable, attribute, or ray attribute), the second is the vector for the left operand (variable, attribute, ray attribute, or uniform), and the third is the vector for the right

operand (variable, attribute, ray attribute, or uniform). The indices are always and only present for variables, ray attributes, and uniforms for indexing into variable, ray attribute, or uniform arrays. For cross product, the output, left operand, and right operand must all be of type vec3. For dot product, the left and right operand must be floating-point vectors of the same component count and the output must be of type float.
The opcode can be one of:
cross product (4D)
dot product (4E)

8-bit opcode	16-bit identifier #1	[index #1]
16-bit identifier #2	[index #2]
16-bit identifier #3	[index #3]

Matrix-vector multiplication. The first identifier is the vector to output to (variable, attribute, or ray attribute), the second is the matrix for the left operand (variable, attribute, ray attribute, or uniform), and the third is the vector for the right operand (variable, attribute, ray attribute, or uniform). The indices are always and only present for variables, ray attributes, and uniforms for indexing into variable, ray attribute, or uniform arrays. Both vectors must have a component count equal to the matrix's height, and both vectors must be floating-point. opcode is (hex) 4F.

8-bit opcode	16-bit identifier #1	[index #1]
16-bit identifier #2	[index #2]
16-bit identifier #3	[index #3]

Matrix-matrix multiplication. The first identifier is the matrix to output to (variable, attribute, or ray attribute), the second is the matrix for the left operand (variable, attribute, ray attribute, or uniform), and the third is the matrix for the right operand (variable, attribute, ray attribute, or uniform). The indices are always and only present for variables, ray attributes, and uniforms for indexing into variable, ray attribute, or uniform arrays. The width of the left operand must be equal to the height of the right operand, and the output matrix must have height equal to the height of the left operand and width equal to the width of the right operand, or else the shader is invalid. opcode is (hex) 50.

Swizzling of vectors:
| 8-bit opcode | 16-bit identifier | [index] | 8-bit result |
Replaces vector components with copies from the original set of components. The index is always and only present for variables and ray attributes for indexing into variable or ray attribute arrays. The 8-bit result is 2 bits per (up to) 4 vector elements, the 2 bits on the farthest right being the component to set for the new X component (0=X, 1=Y, 2=Z, 3=W), next 2 to the left the new Y, next 2 to the left the new Z, and the 2 at the MSB of the byte the new W; unused bits are ignored (e.g. the bits for new Z or W in a vec2), and if any of the bits used refer to a component not in the original vector (e.g. W in a vec3) the shader is invalid. The identifier must be of a vector type and must be a variable, attribute, or ray attribute, or else the shader is invalid. opcode is (hex) 51.

ASSIGNMENT

Left operand: variable, attribute, or ray attribute, right operand: constant
| 8-bit opcode | 16-bit identifier | [index] | [8 bits for vector/matrix element] | 32 bits for value |
The identifier is that of a variable, attribute, or ray attribute to assign to. The index is always and only present for variables and ray attributes for indexing into variable or ray attribute arrays. The 8 bits for vector element is always and only present for vector types; its value specifying the X, Y, Z, or W element to use (0, 1, 2, or 3, respectively); if the vector does not have enough elements the shader is invalid. The 8 bits for matrix element is always and only present for matrix types; if the matrix does not have enough elements the shader is invalid. The constant is given as the same data type as the variable, attribute, or ray attribute being assigned to (or as the same data type of each vector/matrix element if it's a vector or matrix). opcode is (hex) 52.

Left operand: variable or ray attribute array, right operand: array of constants
| 8-bit opcode | 16-bit identifier | 16-bit starting index | 8 bits for value count | n bits for value(s) |
The first identifier is that of a variable or ray attribute to assign the value(s) to (must not be of a vector or matrix type). The 16-bit starting index is the index into variable or ray attribute arrays for assignment to begin at (the first value will be set at this index, the second value will be assigned at this index+1, and so on). This instruction assigns values beginning at the specified index in the variable or ray attribute array, and a number of values specified by the 8-bit

value count (where 0 corresponds to 1) will be assigned to the variables or ray attributes in the array. The value(s) span 32 bits multiplied by the number of values/elements being assigned, and their type is that of the variable or ray attribute. If this instruction occurs describing assignment of more values than are in the array or if the assignments would exceed the end of the array, the shader is invalid. opcode is (hex) 53.

Left operand: variable, attribute, or ray attribute, right operand: variable, attribute, ray attribute, or uniform
| 8-bit opcode | 16-bit identifier #1 | [index #1] | [8 bits for vector/matrix element #1] |
| 16-bit identifier #2 | [index #2] | [8 bits for vector/matrix element #2] |
The first identifier is that of a variable, attribute, or ray attribute to assign the value to, and the second identifier is that of a variable, attribute, ray attribute, or uniform for the value to assign. Index 1 is always and only present for variables and ray attributes for indexing into variable or ray attribute arrays. Index 2 is always and only present for variables, ray attributes, and uniforms for indexing into variable, ray attribute, or uniform arrays. The 8 bits for vector elements are always and only present for vector types; its value specifying the X, Y, Z, or W element to use (0, 1, 2, or 3, respectively); if the first one is > the number of elements in the left (vector) operand - 1, the entire vector will be assigned, and the second vector element must also be > the number of elements in the right operand - 1 to denote that the entire vector will be used. If the left operand is an entire vector, the right operand must be a vector of the same component count. The 8 bits for matrix elements are always and only present for matrix types; they can either both be > the number of elements in the matrix - 1 to assign the entire matrix (in which case the shader is invalid if the two matrices aren't of the same type), or they can both be the index for a valid matrix element to assign a matrix element to a matrix element, and if neither is the case the shader is invalid. opcode is (hex) 54.

| 8-bit opcode | 16-bit identifier | [index] |
Sets the vertex position output; the identifier is that of a variable, attribute, or uniform to get the clip space position from; this must be of type vec4 or else the shader is invalid. The index is always and only present for variables and uniforms for indexing into variable or uniform arrays. If a vertex shader lacks this instruction, the position output will automatically be (0,0,0,1). Can only be present in vertex shaders. opcode is (hex) 55.

| 8-bit opcode | 16-bit identifier | [index] |
Sets the depth output; the identifier is that of a variable, attribute, or uniform to get the depth from; this must be of type float or else the shader is invalid. The index is always and only present for variables or uniforms for indexing into variable or uniform arrays. If a pixel shader lacks this instruction, the depth output will automatically be the perspective-corrected, interpolated depth. Can only be present in pixel shaders. opcode is (hex) 56.

| 8-bit opcode | 16-bit identifier | [index] |
Gets the current primitive's instance ID. Starts at 0 for the first instance of a geometry in a draw call, and increments by 1 for each instance of the geometry drawn throughout the draw call. The identifier is that of a variable or attribute to assign the instance ID; this must be either of type int or uint, or else the shader is invalid. The index is always and only present for variables for indexing into variable arrays. Can only be present in vertex shaders. opcode is (hex) 57.

IMAGE OPERATIONS

| 8-bit opcode | 16-bit identifier #1 | 16-bit identifier #2 | index #1 | 16-bit identifier #3 | index #2 |
Reads a pixel from an image. The first identifier is that of an image to read the pixel from (a uniform), the second identifier is that of a vec4, ivec4, or uvec4 to output the read the pixel data to (a variable, ray attribute, or uniform), and the third identifier is that of a uvec2 specifying the (x,y) image coordinates of the pixel to read (a variable, ray attribute, or uniform). The indices are for indexing into the variable, ray attribute, or uniform arrays. If the image is defined with a fixed-point or floating-point format, the second identifier must be a vec4 or else the shader is invalid. If the image is defined with a signed integer format, the second identifier must be an ivec4 or else the shader is invalid. If the image is defined with an unsigned integer format, the second identifier must be a uvec4 or else the shader is invalid. The values of unused vector elements will be undefined (e.g., in reading from an RG image, the values output for Z and W are undefined). Note RGBA corresponds to the vector elements XYZW, respectively. Can only be present in compute, ray generation, closest-hit, any-hit, or miss shaders. opcode is (hex) 58.

| 8-bit opcode | 16-bit identifier #1 | 16-bit identifier #2 | index #1 | 16-bit identifier #3 | index #2 |
Writes to a pixel in an image. The first identifier is that of an image to write the pixel data to (a uniform), the second identifier is that of a vec4, ivec4, or uvec4 to get the the pixel data from (a variable, ray attribute, or uniform), and the third identifier is that of a uvec2 specifying the (x,y) image coordinates of the pixel to write to (a variable, ray attribute, or uniform). The indices are for indexing into the variable, ray attribute, or uniform arrays. If the image is defined with a fixed-point or floating-point format, the second identifier must be a vec4 or else the shader is invalid. If the image is defined with a signed integer format, the second identifier must be an ivec4 or else the shader is invalid. If the image is defined with an unsigned integer format, the second identifier must be a uvec4 or else the shader is invalid. When writing components that are outside of the range [0,1] to fixed-point images, the results are undefined. The values of unused vector elements are ignored (e.g., in writing to an RG image, only the X and Y are used). Note RGBA corresponds to the vector elements XYZW, respectively. Can only be present in compute, ray generation, closest-hit, any-hit, or miss shaders. opcode is (hex) 59.

| 8-bit opcode | 16-bit identifier #1 | 16-bit identifier #2 | index |
Gets the dimensions of an image. The first identifier is that of an image to get the dimensions of (a uniform), the second identifier is that of an ivec2 to output the (x,y) dimensions to (a variable or ray attribute), and the index is for indexing into the variable or ray attribute array. Can only be present in compute, ray generation, closest-hit, any-hit, or miss shaders. opcode is (hex) 5A.

TEXTURE OPERATIONS

Texture coordinate sampling, specified LOD:
8-bit opcode	16-bit identifier #1	[index #1]
16-bit identifier #2	index #2
16-bit identifier #3	[index #3]
16-bit identifier #4	[index #4]	[8 bits for vector/matrix element]
Samples the R, G, B, and A of a texel using texture coordinates. The first identifier is that of a vec4, ivec4, or uvec4 to output to (a variable, attribute, or ray attribute), the second identifier is that of a sampler type (a uniform), the third identifier is that of a vec2 specifying texture coordinates (a variable, attribute, ray attribute, or uniform), and the fourth identifier is that of a float specifying the level-of-detail for the sampling (a variable, attribute, ray attribute, or uniform). The indices are always and only present for variables, ray attributes, and uniforms for indexing into variable, ray attribute, or uniform arrays. If in a vertex or pixel shader, the second index must be either a constant or use the current loop iteration. The 8 bits for vector element is always and only present for vector types; its value specifying the X, Y, Z, or W element to use (0, 1, 2, or 3, respectively); if the vector being used does not have enough elements the shader is invalid. The 8 bits for matrix element is always and only present for matrix types; if the matrix being used does not have enough elements the shader is invalid. The type of sampler (sampler, isampler, or usampler) must correspond with the output vector type (vec4, ivec4, or uvec4), or else the shader is invalid. The values of untouched vector elements after texture sampling are undefined (e.g., in sampling an RG texture, only X and Y are set; Z and W are undefined). Note RGBA corresponds to the vector elements XYZW, respectively. Out of bounds texture coordinates will be subject to the wrapping rules of the texture object being sampled. The result vector from sampling from a negative level-of-detail, or one greater than the number of texture levels in the TBO being sampled, is undefined. The result vector is undefined if in a pixel shader and outdated GPU is set in ***Hardware Information***. opcode is (hex) 5B.

Texture coordinate sampling, automatic LOD:
8-bit opcode	16-bit identifier #1	[index #1]
16-bit identifier #2	16-bit identifier #3	[index #2]
16-bit identifier #4	[index #3]	[8 bits for vector/matrix element]
The same as above, except that the float that specifies LOD instead specifies a bias to be summed with the LOD which gets automatically calculated for the sample. Can only be present in pixel shaders, and can only be present on level 1 of the main function. opcode is (hex) 5C.

Texel coordinate sampling, specified LOD:
8-bit opcode	16-bit identifier #1	[index #1]
16-bit identifier #2	index #2
16-bit identifier #3	[index #3]
16-bit identifier #4	[index #4]	[8 bits for vector/matrix element]
Samples the R, G, B, and A of a texel using texel coordinates. The first identifier is that of a vec4, ivec4, or uvec4 to output to (a variable, attribute, or ray attribute), the second identifier is that of a sampler type (a uniform), the third identifier is that of an ivec2 specifying texture coordinates (a variable, attribute, ray attribute, or uniform), and the fourth identifier is that of a uint specifying the level-of-detail for the sampling (a variable, attribute, ray attribute, or uniform). The indices are always and only present for variables, ray attributes, or uniforms for indexing into variable, ray attribute, or uniform arrays. If in a vertex or pixel shader, the second index must be either a constant or use the current loop iteration. The 8 bits for vector element is always and only present for vector types; its value specifying the X, Y, Z, or W element to use (0, 1, 2, or 3, respectively); if the vector being used does not have enough elements the shader is invalid. The 8 bits for matrix element is always and only present for matrix types; if the matrix being used does not have enough elements the shader is invalid. The type of sampler (sampler, isampler, or usampler) must correspond with the output vector type (vec4, ivec4, or uvec4). The values of untouched vector elements after texture sampling are undefined (e.g., in sampling an RG texture, only X and Y are set; Z and W are undefined). Note RGBA corresponds to the vector elements XYZW, respectively. Out of bounds texel coordinates will be subject to the wrapping rules of the texture object being sampled. The result vector from sampling from a negative level-of-detail, or one greater than the number of texture levels in the TBO being sampled, is undefined. The result vector is undefined if in a vertex or pixel shader and outdated GPU is set in ***Hardware Information***. opcode is 5D.

| 8-bit opcode | 16-bit identifier #1 | index #1 | 16-bit identifier #2 | [index #2] | 16-bit identifier #3 | [index #3] |
Gets the dimensions of a specified texture level in a texture. The first identifier is that of a sampler that refers to the texture (a uniform), the second identifier is that of an ivec2 to output the (x,y) dimensions to (a variable, attribute, or ray attribute), and the third identifier is that of an int that specifies the texture level to get the dimensions of (a variable, attribute, ray attribute, or uniform). The indices are always and only present for variables, ray attributes, and uniforms for indexing into variable, ray attribute, or uniform arrays. If getting the dimensions of a non-existent texture level or outdated GPU is set in ***Hardware Information***, results are undefined. opcode is (hex) 5E.

CONDITIONALS

| 8-bit opcode #1 | condition(s) | 8-bit opcode #2 | ... | 8-bit opcode #3 |
A conditional branch. The first opcode is one of if (5F), else-if (60), or else (61), the second opcode is the body opener (0E), and the third is the body closer (08). The condition(s), present only for if and else-if, is a list of one or more conditions, each of the form:
| value #1 | 8-bit relational operator | value #2 | [conjunction] | [additional condition] |.
In order for the code in the conditional branch (denoted by an ellipsis) to be executed, the condition(s) must all pass when evaluated. A relational operator and two values (also called operands) form a condition. The relational operator is greater than (62), less than (63), less than or equal to (64), greater than or equal to (65), the same as (66), or not the same as (67). Value 1 and 2 are a variable, attribute, ray attribute, or uniform, and one of them can optionally be a constant with the same type as the other value. Each value is of the form:
| 8-bit type | [16-bit identifier] | [index] | [8 bits for vector/matrix element] | [32 bits for value] |.
If type is 0, this is a variable, attribute, ray attribute, or uniform, and the identifier must be present, the 32-bit value must be omitted, the index is always and only present for variables, ray attributes, or uniforms for indexing into variable, ray attribute, or uniform arrays, the 8 bits for vector element is always and only present for vectors; its value specifying the X, Y, Z, or W element to use (0, 1, 2, or 3, respectively); if the vector does not have enough elements the shader is invalid, and the 8 bits for matrix element is always and only present for matrix types; if the matrix being used does not have enough elements the shader is invalid. When one operand is an integer but the other is a floating-point, the integer is typecast to floating-point before the condition is checked. If either of the values are of a sampler or acceleration structure type, the shader is invalid. If type is non-zero, everything omissible must be omitted except for the 32-bit value, which is of the same type as the other operand if their type is 0, otherwise both 32-bit values are floating-point.

The conjunction is either or (68) or and (69); conditions are read from left to right and the final decision of whether or not two or more conditions pass can be set to depend on each other by using the and conjunction between

them; e.g., “A or B and C” would say that condition A, or both B and C, has to be true. “A and B or C” would say that A and B have to be true, or C has to be true. “A or B and C or D" would say that A has to be true, or both B and C, or D. The additional condition is another condition; two values with a relational operator in the middle, optionally followed by another conjunction to introduce yet another condition. Each conjunction must be followed by an additional condition.

Between attached conditional branches (between if and else-if, and between else-if and else) there must be no other code. An if branch will begin a new chain of conditionals. An else-if branch must not be without an attached if branch at the top of a chain of conditionals, and an else branch must not be outside of a chain of conditionals. An else branch will end a chain of conditionals. Conditionals must contain no variable definitions.

LOOPS

| 8-bit opcode #1 | 16-bit iteration count | 8-bit opcode #2 | ... | 8-bit opcode #3 |
A loop. The first opcode is that of a loop (6A), the iteration count is the number of times to run the loop (if 0, the shader is invalid), the second opcode is that of a body opener (0E), and the third opcode is that of a body closer (08). The code denoted by the ellipsis is the code for the loop to run. Loops must contain no variable definitions.

| 8-bit opcode |
Break out of the current loop immediately and continue execution from the end of the loop. Will have no effect outside of a loop. opcode is (hex) 6B.

| 8-bit opcode |
Continue to next iteration of current loop immediately. Will have no effect outside of a loop. opcode is (hex) 6C.

RAY TRACING

8-bit opcode	16-identifier #1	16-bit identifier #2	index #1	16-bit identifier #3	index #2
16-bit identifier #4	index #3	32-bit identifier #5	index #4	16-bit identifier #6	index #5
16-bit identifier #7	index #6	16-bit identifier #8	index #7	16-bit identifier #9	index #8
16-bit identifier #10	index #9	16-bit extra info
Trace a ray. The first 16-bit identifier must be that of an acceleration structure uniform; this is the TLAS for the new ray to traverse through. The second identifier is that of a uint variable to get the new ray’s flags from (the upper 25 bits are ignored); see ***Visibility Modifiers***. If mutual exclusivity is ever violated with any of the flags, results are undefined. The third identifier is that of a uint variable to get the ray’s 8-bit cull mask from (the upper 24 bits are ignored). The ray’s 8-bit cull mask will be compared to that of the geometry instances it intersects with; see ***Visibility Modifiers***. The fourth identifier is that of a uint variable to get the hit group SBT record offset from; see ***Shader Binding Tables***. The fifth identifier is that of a uint variable to get the hit group SBT record stride from. The sixth identifier is that of a uint variable to get the miss SBT record index from. The seventh identifier is that of a vec3 variable to get the ray’s origin position from. The eighth identifier is that of a vec3 variable to get the ray’s direction from. The ninth identifier is that of a float variable to get the ray’s minimum length from. The tenth identifier is that of a float variable to get the ray’s maximum length from. The indices are for indexing into variable arrays. The 16-bit extra info is the location ID of the ray payload that will be accessible for read/write when an incoming ray payload is defined at the same location ID in shaders called during the tracing of the new ray, and the incoming ray payload should be defined with a format identical to the ray payload it accesses or else results are undefined; if a ray payload was not defined within this shader at this location ID the shader is invalid. The range where intersections can occur along a ray is within the ray’s minimum and maximum length. Can only be present in ray generation, closest-hit, and miss shaders. opcode is (hex) 6D.

| 8-bit opcode |
Ignores a ray intersection; ends execution of the any-hit shader and continues the trace without modifying the maximum length. Can only be present in any-hit shaders. opcode is (hex) 6E.

| 8-bit opcode |
Terminate a ray; updates the ray’s maximum length and ends execution of the any-hit shader, then invoking the current closest-hit shader. Can only be present in any-hit shaders. opcode is (hex) 6F.

| 8-bit opcode | 8-bit extra info | 16-bit identifier | index |

Get state of the current ray being traced. The identifier is that of a variable to output the state to, and the index is for indexing into a variable array. The extra info specifies the type of information to get, which determines what type of variable should be given. If extra info is 0, the variable is a uvec3 to set to the current ray's coordinates (x,y; ray tracing is done for each pixel within the rectangle whose dimensions are given by the instruction 96 that initiated the ray tracing operation) and the current recursion level (z). If extra info is 1, the variable is a uvec3 to set to the ray trace operation's width (x), height (y), and recursion depth (z). If extra info is 2, the variable is a uint to set to the intersected primitive's index (any-hit and closest-hit shaders only) within the intersected geometry instance. If extra info is 3, the variable is a uint to set to the index of the geometry instance intersected by the ray (any-hit and closest-hit shaders only). If extra info is 4, the variable is a uint to set to the custom index of the geometry instance intersected by the ray (any-hit and closest-hit shaders only). If extra info is 5, the variable is a vec3 to set to the ray's origin in object space (origin without being affected by the intersected geometry instance's transformation matrix; any-hit, closest-hit, and miss shaders only). If extra info is 6, the variable is a vec3 to set to the ray's direction in object space (direction without being affected by the intersected geometry instance's transformation matrix; any-hit, closest-hit, and miss shaders only). If extra info is 7, the variable is a vec3 to set to the ray's origin in world space (origin after being affected by the intersected geometry instance's transformation matrix; any-hit, closest-hit, and miss shaders only). If extra info is 8, the variable is a vec3 to set to the ray's direction in world space (direction after being affected by the intersected geometry instance's transformation matrix; any-hit, closest-hit, and miss shaders only). If extra info is 9, the variable is a mat4x3 to set to the object-to-world matrix (determined by the intersected geometry instance; any-hit and closest-hit shaders only). If extra info is 10, the variable is a mat4x3 to set to the world-to-object matrix (determined by the intersected geometry instance; any-hit and closest-hit shaders only). If extra info is 11, the variable is a float to set to the ray's minimum length; any-hit, closest-hit, and miss shaders only. If extra info is 12, the variable is a float to set to the ray's maximum length; any-hit, closest-hit, and miss shaders only. In the closest-hit shader, the maximum length is the closest distance to an accepted intersection. In the any-hit shader, the maximum length is the distance to the primitive that is currently being intersected. In a miss shader, it is the maximum length of the ray as provided to the trace ray shader function that created this ray. If extra info is 13, the variable is a uint to set to the current ray's *Force all intersected geometry to be considered opaque* flag (1 if set, 0 if not); any-hit, closest-hit, and miss shaders only. If extra info is 14, the variable is a uint to set to the current ray's *Force all intersected geometry to be considered non-opaque* flag (1 if set, 0 if not); any-hit, closest-hit, and miss shaders only. If extra info is 15, the variable is a uint to set to the current ray's *Terminate ray after first hit* flag (1 if set, 0 if not); any-hit, closest-hit, and miss shaders only. If extra info is 16, the variable is a uint to set to the current ray's *Skip closest-hit shader* flag (1 if set, 0 if not); any-hit, closest-hit, and miss shaders only. If extra info is 17, the variable is a uint to set to the current ray's *Cull back facing triangles* flag (1 if set, 0 if not); any-hit, closest-hit, and miss shaders only. If extra info is 18, the variable is a uint to set to the current ray's *Cull front facing triangles* flag (1 if set, 0 if not); any-hit, closest-hit, and miss shaders only. If extra info is 19, the variable is a uint to set to the current ray's *Cull all primitives considered opaque by the geometry and geometry instance flags* flag (1 if set, 0 if not); any-hit, closest-hit, and miss shaders only. If extra info is 20, the variable is a uint to set to the current ray's *Cull all primitives considered non-opaque by the geometry and geometry instance flags* flag (1 if set, 0 if not); any-hit, closest-hit, and miss shaders only. opcode is (hex) 70.

COMPUTE

| 8-bit opcode |

Barrier instruction. No work-items in the work-group will continue execution past this point until all work-items in the work-group have reached it. Can only be present on level 1 of the main function. Can only be present in compute shaders. opcode is (hex) 71.

| 8-bit opcode | 8-bit extra info | 16-bit identifier | index |

Get information about compute shader invocation. The identifier is that of a variable to output the information to, and the index is for indexing into the variable array. The extra info specifies the type of information to get. If extra info is 0, the variable is a uvec3 to set to the dimensions provided for the dispatch work-groups instruction. If extra info is 1, the variable is a uvec3 to set to the XYZ of the work-group the current work-item is within. If extra info is 2, the variable is a uvec3 to set to the XYZ of the current work-item within the current work-group. Note the minimum coordinates for the (x,y,z) of a work-group or work-item is (0,0,0), and the maximum of each coordinate is the corresponding dimension from the global/local work-group dimensions, subtract 1. Can only be present in compute shaders. opcode is (hex) 72.

PIXEL

| 8-bit opcode |

Discards the current pixel. Can only be present in pixel shaders. opcode is (hex) 73.