VideoCore® IV 3D
Architecture Reference Guide

VideoCoreIV-AG100-R

5300 California Avenue • Irvine, CA 92617 • Phone: 949-926-5000 • Fax: 949-926-5203 September 16, 2013

Revision History

Revision	*Date*	*Change Description*
VideoCoreIV-AG100-R	09/16/13	Initial release

Broadcom Corporation
5300 California Avenue
Irvine, CA 92617

Table of Contents

List of Figures

List of Tables

About This Document

Purpose and Audience

The document details the 3D system in VideoCore® IV and the associated software tasks. The target audience for this document is software and hardware engineers.

Acronyms and Abbreviations

In most cases, acronyms and abbreviations are defined on first use.

For a comprehensive list of acronyms and other terms used in Broadcom documents, go to: http://www.broadcom.com/press/glossary.php.

Document Conventions

The following conventions may be used in this document:

Convention	*Description*
Bold	User input and actions: for example, type **exit**, click **OK,** press **Alt+C**
`Monospace`	Code: `#include <iostream>` HTML: `<td rowspan = 3>` Command line commands and parameters: `wl [-l] <command>`
< >	Placeholders for *required* elements: enter your <username> or `wl <command>`
[ ]	Indicates *optional* command-line parameters: `wl [-l]` Indicates bit and byte ranges (inclusive): [0:3] or [7:0]

Section 1: Introduction

The second generation 3D system in VideoCore® IV is a major step on from the first generation 3D hardware in VideoCore III. The VideoCore IV 3D architecture is scalable, based around multiple specialist floating-point shading processors called QPUs.

To avoid escalating memory bandwidth at higher resolutions and higher performance, the new architecture uses tile-based pixel rendering. This reduces frame-buffer bandwidth by an order of magnitude compared to immediate-mode rendering, and allows 4x multisample antialiasing to be used with little penalty in performance or memory consumption. A fully configured system will support 720p resolution with 4x multisampling at good frame rates with next generation game content.

The main specification points for a fully configured system at 250 MHz are:

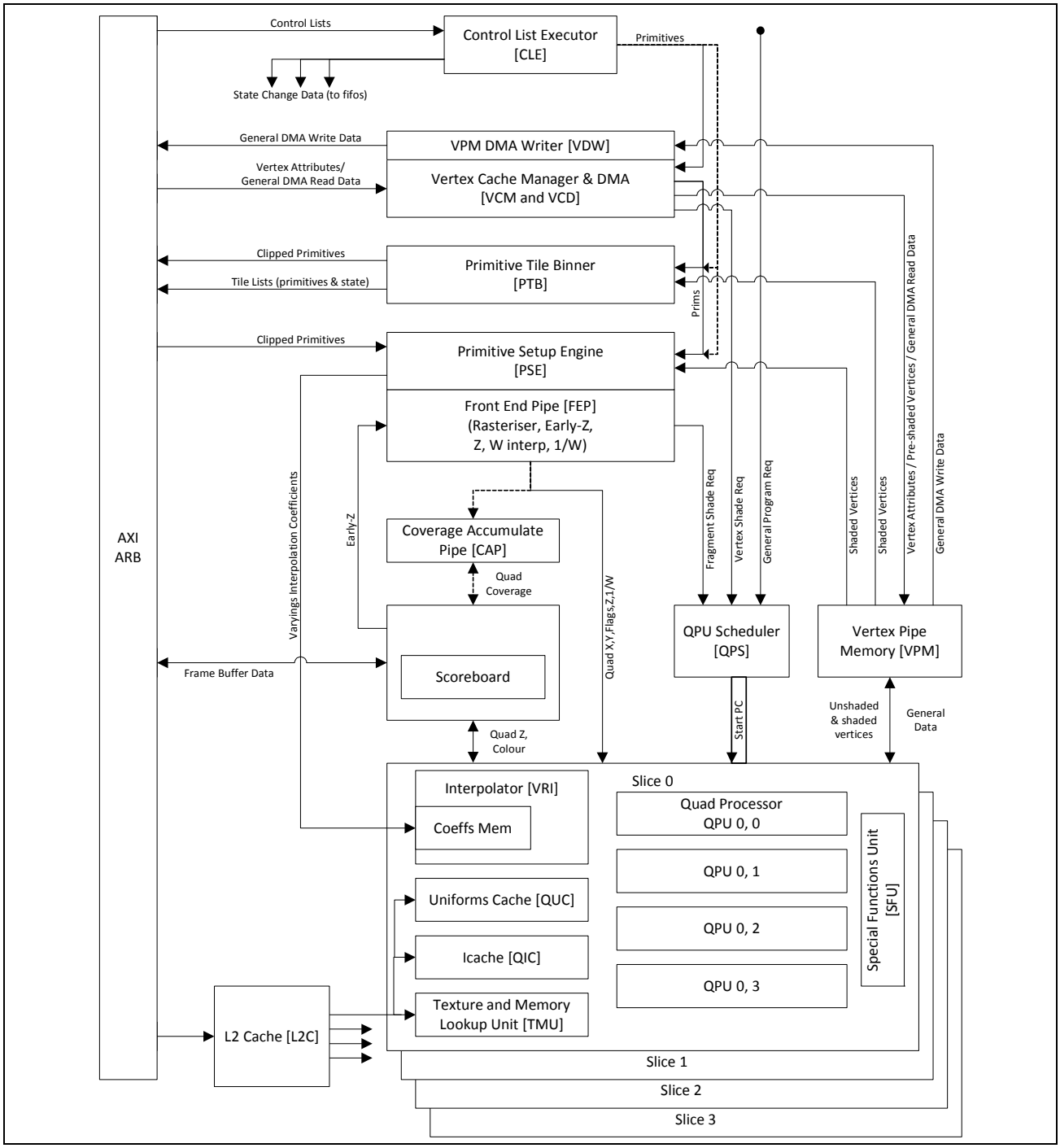
- 25M rendered triangles/s.
- 1G pixels/s with single bilinear texturing, simple shading, 4x multisampling.
- Supports 16x coverage mask antialiasing for 2D rendering at full pixel rate.
- 720p standard resolution with 4x multisampling.
- Supports 16-bit HDR rendering.
- Fully supports OpenGL-ES 1.1/2.0 and OpenVG 1.1.

The VideoCore IV 3D hardware is self-contained and highly automated, requiring little processing bandwidth or real-time intervention from software drivers. This, together with the scalability, makes this 3D architecture suitable for use in a wide variety of SoC systems.

Section 2: Architecture Overview

Figure 1 shows the system block diagram for VideoCore IV 3D.

Figure 1: VideoCore® IV 3D System Block Diagram



Scalability is principally provided by multiple instances of a special purpose floating-point shader processor, termed a Quad Processor (QPU). The QPU is a 16-way SIMD processor. Each processor has two vector floating-point ALUs which carry out multiply and non-multiply operations in parallel with single instruction cycle latency. Internally the QPU is a 4-way SIMD processor multiplexed 4× over four cycles, making it particularly suited to processing streams of quads of pixels.

QPUs are organized into groups of up to four, termed slices, which share certain common resources. Each slice shares an instruction cache, a Special Function Unit (for recip/recipsqrt/log/exp functions), one or two Texture and Memory lookup Units (TMU), and Interpolation units for interpolation of varyings (VRI).

The Vertex Cache Manager and DMA (VCM and VCD) collect together batches of vertex attributes and place these into the Vertex Pipe Memory (VPM). Each batch of vertices is shaded by one of the QPUs and the results are stored back into the VPM.

In the tile binning phase, only the vertex coordinate transform part of the vertex shading is performed. The Primitive Tile Binner (PTB) fetches the transformed vertex coordinates from the VPM, and works out which tiles, if any, each primitive overlaps. As it goes along the PTB builds a list in memory for each tile, which contains all the primitives impacting that tile, plus references to any state changes that apply.

The Primitive Setup Engine (PSE) fetches shaded vertex data from the VPM and calculates setup data for rasterising primitives and the coefficients of the equations for interpolating varyings. The rasteriser setup parameters and the Z and W interpolation coefficients are fed to the Front End Pipeline (FEP), but the varyings interpolation coefficients are stored directly to a memory in every QPU slice, where they are used for just-in-time interpolation.

The Front End Pipeline (FEP) includes the Rasteriser, Z interpolation, Early-Z test, W interpolation and W reciprocal functions, operating at 4 pixels per clock. The early-z test uses a reduced resolution shadow of the tile's Z-buffer, which is maintained by the tile buffer block. Each group of four successive quads of 4 pixels output by the FEP is stored into dedicated registers mapped into the QPU which is scheduled to carry out fragment shading for that group of quads.

The QPUs are scheduled automatically by the hardware via the QPU scheduler (QPS). The VCM/VCD continuously fills input buffers of batches of 16 vertex attributes in the VPM. Whenever there is an input batch ready, the next available QPU is scheduled for vertex shading for that batch. When there is no vertex shading batch ready to process, the next available QPU is scheduled to fragment shade the next batch of four quads that become ready.

There is nominally one Texture and Memory lookup Unit (TMU) per slice, but texturing performance can be scaled by adding TMUs. Due to the use of multiple slices, the same texture will appear in more than one TMU. Each texture unit has a small L1 cache. There is an L2 cache (L2C) shared between all texture units.

The TMU in each slice can be used for general-purpose data lookups from memory as well as filtered texture lookups. Another mechanism for reading and writing main memory is to use the VCD or VDW to DMA data into or out of the VPM from where it can be accessed by the QPUs. The QPUs can also read program constants, as in non-indexed shader uniforms, as a stream of data from main memory via the Uniforms Cache. The Uniforms Cache (and Instruction Cache) for each slice fetch data from main memory via the common L2C.

The PSE stores varying interpolation coefficients simultaneously into memory in every QPU slice. With simultaneous add and multiply, each varying interpolated only costs a single QPU instruction.

A dedicated Coverage Accumulation Pipeline (CAP) is used for OpenVG coverage rendering.

The QPUs and CAP all output pixel data to a shared Tile Buffer (TLB). The standard tile buffer size is 64 × 64 samples, supporting 32 × 32 pixel tiles in 4× multisample mode or 64 × 64 pixel tiles in non-multisample and OpenVG 16× coverage modes. The TLB can also be configured as 64 × 32 samples with 64-bit floating-point color for HDR rendering. For size constrained systems the TLB can be configured as ½ or ¼ sized (64 × 32 or 32 × 32 samples). The TLB normally writes out decimated color data to the main memory frame buffer when rendering of a tile is complete, but can also store and reload the full multisample Z, color, stencil, and alpha-mask tile data to/from memory.

The QPUs perform all color blends in software, but the tile buffer carries out the Z-test and stencil-ops in hardware. The QPUs therefore have read and write interfaces to the tile memory for color data, but only write Z and stencil data. When carrying out OpenVG coverage rendering using the CAP, the tile buffer Z samples are reused for coverage information. The QPUs have an additional interface to read back this coverage value. There is also an 8-bit alpha-mask buffer for OpenVG usage, which the QPUs also have read and write access to.

The 3D pipeline is driven by control lists in memory, which specify sequences of primitives and system state data. There is a Control List Executor (CLE) which interprets the control lists and feeds the pipeline with primitive and state data. The pipeline automation is such that very little interaction is required from the software driver, with most state changes handled efficiently by the hardware without any flushing of the pipeline. In particular, the pixel rendering pass of all tiles can normally be carried out without any driver involvement.

Section 3: Quad Processor

The VideoCore IV quad processor (QPU) is a SIMD machine with the following key features:

- A highly uniform 64-bit instruction encoding
- Four-way physical parallelism
- 16-way virtual parallelism (4-way multiplexed over four successive clock cycles)
- A dual-issue floating-point ALU (one add and multiply per cycle)
- Two large single-ported register files
- Five accumulators
- I/O mapped into the register space
- Support for two hardware threads
- Instruction and register level coupling to the 3D hardware

For all intents and purposes the QPU can be regarded as a 16-way 32-bit SIMD processor with an instruction cycle time of four system clocks. The latency of floating point ALU operations is accommodated within these four clock cycles, giving single cycle operation from the programmer's perspective.

Internally the QPU is a 4-way SIMD processor multiplexed to 16-ways by executing the same instruction for four clock cycles on four different 4-way vectors termed 'quads'. This allows a simple and efficient pipeline design without complex interlocks and forwarding paths, which is well matched to processing a stream of pixel quads. The four clock instruction cycle also allows four QPUs to be clustered together to share a common instruction cache, forming what is termed a processing 'slice'.

The QPU ALU is dual-issue, the design uses a small number of accumulators in conjunction with large single-ported register files to provide the bandwidth needed to perform two binary operations per cycle.

Floating-point reciprocal, reciprocal square root, logarithm, and exponentiation operations are performed by a separate, shared block in each slice.

The QPUs are closely coupled to the 3D hardware specifically for fragment shading, and for this purpose have special signaling instructions and dedicated special-purpose internal registers.

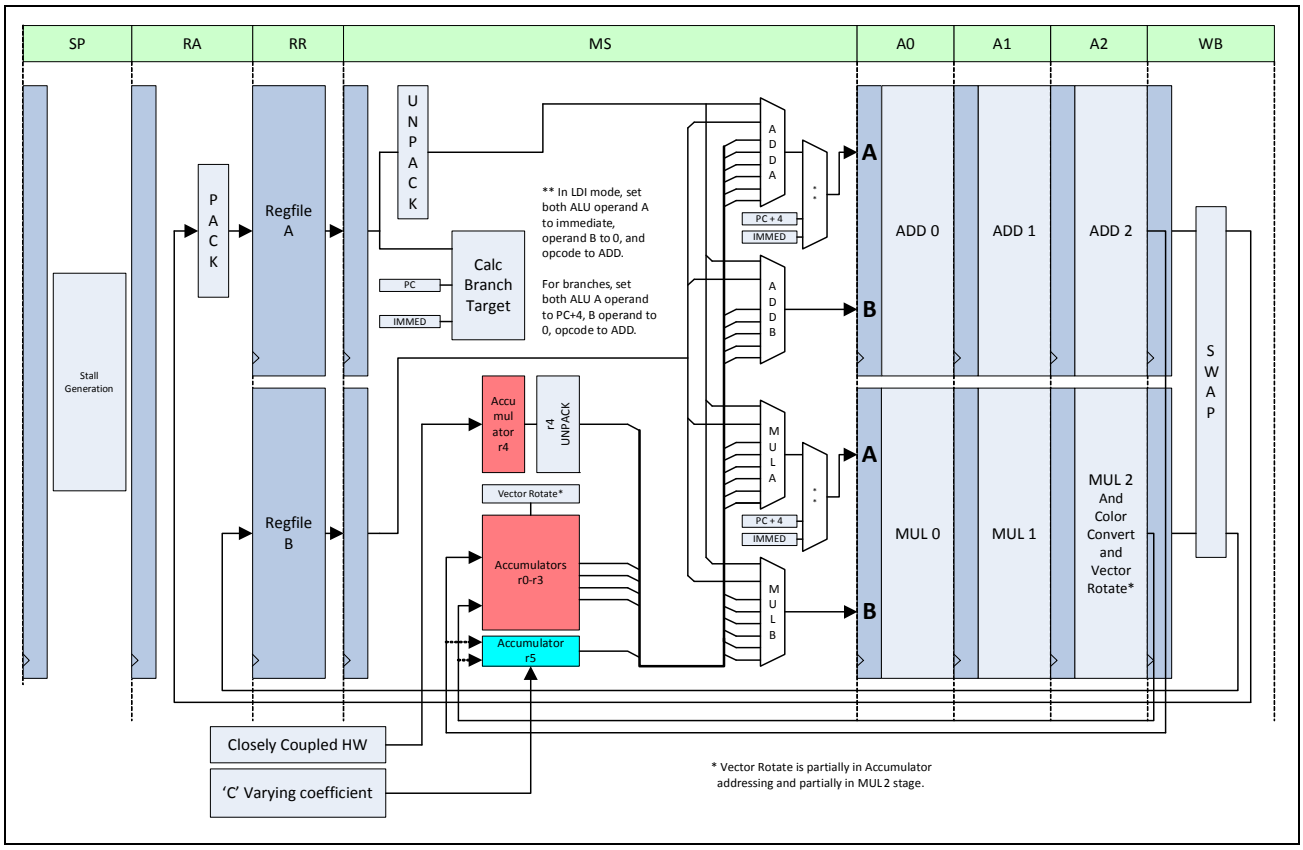
Although they are physically located within, and closely coupled to the 3D system, the QPUs are also capable of providing a general-purpose computation resource for non-3D software, such as video codecs and ISP tasks.

Quad Processor Architecture

Core Pipeline Operation

The front end of each QPU pipeline receives instructions from a shared instruction cache (icache). As one icache unit serves four QPUs in four successive clock cycles the front end pipelines of each of these four QPUs will be at different phases relative to each other. After instruction fetch there is a 're-synchronisation' pipeline stage which brings all of the QPUs into phase with each other. The re-synchronised parts of the QPU pipeline are shown in Figure 2 along with the names assigned to each pipeline stage.

Figure 2: QPU Core Pipeline



Processor Registers

The QPU contains two sets of physical registers consisting of a set of four general-purpose accumulators, two special-purpose accumulators, and two large register-file memories. The register space associated with each of the A and B regfiles can address 64 locations, 32 of these are backed by the physical registers while the other 32 are used to access register-space I/O.

The QPU pipeline is constructed such that register files have an entire pipeline cycle to perform a read operation. As the QPU pipeline length from register file read to write-back is greater than four cycles, one cannot write data to a physical QPU register-file in one instruction and then read that same data for use in the next instruction (no forwarding paths are provided). QPU code is expected to make heavy use of the six accumulator registers, which do not suffer the restriction that data written by an instruction cannot be read in the next instruction.

Accumulator r4 is special in that it is the interface for receiving data from most of the closely coupled hardware units (Texture unit, Tile Buffer, SFU, etc.). These units write data directly into r4. r4 is a read only register from the processors perspective.

Accumulator r5 receives the 'C' coefficient from the varyings interpolation hardware when performing fragment shading. r5 can be written to – however it cannot be used as a general-purpose register as it only contains 32-bits per quad (as the 'C' coefficient is constant across a quad). Reading r5 returns the per-quad 32-bit value replicated across the four elements of that quad. When writing to r5, each of the ls bytes of the four elements of a quad are concatenated to form the 32-bit value.

Finally, the low 4 bits of SIMD element 0 (quad 0, element 0) in r5 can be used to specify one of the possible 16 rotations when performing a horizontal vector rotate of the mul ALU output.

ALUs

The QPU contains two independent (and asymmetric) ALU units, an 'add' unit and a 'mul' unit. The 'add' unit performs add-type operations, integer bit manipulation/shift/logical operations and 8-bit vector add/subtract (with saturation). The multiply unit performs integer and floating point multiply, 8-bit vector adds/subs/min/max and multiply (where the 8-bit vector elements are treated as being in the range [0.0, 1.0]).

The multiply unit also can convert the 32-bit float result to scaled 8-bit data in the range [0.0, 1.0].

Both ALU units can operate on integer or floating point data, and internally always operate on 32-bit data (or vectors of 4 8-bit quantities). The QPU provides hardware to read 16 and 8-bit data from register file A and sign/zero extend (or convert appropriately for 16 bit float) the data before feeding it to the ALUs. There is similar logic to re-convert the 32-bit output from the ALUs to 16/8 bits when writing to the A register file. The pack and unpack blocks in Figure 2 illustrate this hardware.

Accumulators r0-r3 can only operate on 32-bit data and have no pack/unpack functionality.

Both the tile buffer and texture units return packed data into accumulator r4, so this accumulator to unpack 16 bit float or 8-bit data in the range [0.0, 1.0].

The instruction encoding contains two sets of condition fields, one for each of the add and mul ALU pipes. Therefore each ALU can be independently programmed to conditionally write back its data based on the current condition flags.

The two ALUs require a total of up to four input arguments. These can each be selected independently from one of the six accumulators, or one of the register-space A or B read values.

Normally, the add ALU will write back to the A register file space, and the mul ALU to the B register space. There is a single bit in the instruction encoding which will swap the destination for the ALUs so that the add ALU can write to B and the mul to A. Accumulators and register mapped I/O are mapped into both A and B spaces for write-back and hence either ALU can write to any of the five writable accumulators (r5, r0-3) or I/O locations. The behavior is undefined if both ALUs write to the same accumulator or I/O register.

Signaling Bits

QPU instructions have a dedicated 4-bit field (known as the signaling or SIG instruction bits) which provides a way to signal various things to the 3D hardware along with executing a normal instruction, or as a special instruction modifier – for example, to indicate a branch instruction.

A common use of the signaling bits is to signal a read from an external hardware resource into accumulator r4. Data in r4 from the hardware is then available to the instruction following the one which signalled the read.

Load Immediate

QPU Load Immediate instructions are signalled by a signaling field value. The LS 32 bits of the instruction word contain the immediate data, and the MS 32-bits contain the same fields as for ALU instructions, except for the 'UNPACK' field, which is re-used to indicate the type of load immediate. The immediate data supplied by a Load Immediate instruction cannot be used directly as an ALU input argument, and is instead available at the output of the ADD and MUL ALUs where it can be written to a register.

There are three types of load immediate instruction, the first (type0) takes the 32 bit immediate value and replicates it 16-ways across the entire SIMD array. The second two types interpret the immediate data as per-element 2-bit fields, either as signed (type1) or unsigned (type2) integer values.

The QPU does have limited support for supplying immediate values within normal ALU instructions – see "Small Immediates" on page 19.

Small Immediates

Instructions using small immediates are signalled by a special signaling value. Small immediates instructions do not allow a read from the B register file space, instead using the 6-bit B read address field bits to encode the immediate value.

A 5-bit signed immediate, one of several small power of two floating-point values, or one of a possible 16 horizontal vector rotations of the mul ALU output can be specified. Immediate values are replicated across all 16 SIMD elements, are available as if they were normal reads from the 'B' register file.

Branches

QPU branches are conditional based on the status of the ALU flag bits across all 16 elements of the SIMD array. QPU Branch instructions are signalled using a special signaling value. For simplicity the QPUs do not use branch prediction and never cancel the sequentially fetched instructions when a branch is encountered. This means that three 'delay slot' instructions following a branch instruction are always executed. On branch instructions the 'link' address of the current instruction plus four is present in place of the add and mul ALU write-back values if the branch is taken, and may be written to a register-file location to support branch-with-link functionality.

Branch targets are generated by adding the signed immediate filed from the instruction and optionally the program counter and/or a value read from SIMD element 0 in the A register file.

Note: The PC-relative branches are relative to PC+4, not the PC at the branch instruction.

Horizontal Vector Rotation

The 16-way vector output by the mul ALU may be rotated by any of the 16 possible horizontal rotations. This provides the QPUs with most of the image processing flexibility of the VideoCore VPUs, and differentiates the QPU from a conventional 'silo' SIMD processor. The full horizontal vector rotate is only available when both of the mul ALU input arguments are taken from accumulators r0-r3.

Horizontal rotations are specified as part of the instruction word using certain values of the special 'small immediate' encoding (see "Small Immediates" on page 19). The rotation can either be specified directly from the immediate data or taken from accumulator r5, element 0, bits [3:0].

Pack and Unpack

The QPU supports 'unpacking' of 32-bit fields read from the A register file (or in a limited fashion when read from accumulator r4). Repacking of data is also supported on write-back to register file A, or if the data is output from the MUL pipeline, it can also be converted from 32-bit float and written back as 8-bit color to any of the four possible byte locations of the MUL destination (which can be an accumulator or regfile).

The source (packed) data can be 8-bit unsigned integers, 8-bit 'color' values (nominally in the range [0, 1.0]) 16-bit signed integers or 16-bit floats.

Destination (packed) data can be any of the unpackable types, with the addition that saturation can be applied to the 8 and 16 bit immediates if required. Floating point results from the mul ALU can be 'packed' as 8-bit color data in the range [0, 1.0] and written to any destination.

Thread Control

When the QPU is executing a second hardware thread, the upper and lower 16 locations of each physical regfile are swapped by inverting address bit 4. This splits each regfile to provide 16 vectors of local thread storage. Some of the register-space mapped I/O locations for interfacing with the 3D pipeline are also swapped when executing a second thread.

Note: The QPU accumulator registers (and flags) are not preserved across thread switches, so any required program state must be saved in the A and B regfiles.

QPU programs are started by a centralised QPU scheduler, which receives automatic requests from the 3D pipeline to run shader programs. Requests to run general-purpose programs can also be sent to the scheduler by a queue written via the V3DPRQPC and V3DPRQUA system registers. These supply the initial PC address and an optional Uniforms base address for the program.

QPU programs are terminated by a Program End (thread end) signalled instruction. Two delay-slot instructions are executed after the Program End instruction before the QPU becomes idle.

The write cycle of the instruction that signals a program end is used by the hardware to write in the W and Z data for a new thread. The instruction signaling program end must therefore not attempt to write to either of the physical A or B register files. The new W and Z are written to address 14 in regfile A and B respectively and therefore all instructions following and including the instruction signaling the program end (that is, the last three of any thread) must not touch register-file location 14. Internally, the hardware inverts the ls bit of the regfile address space on every new thread, so while the new Z and W arrive at location 14 in the old thread address space, they are in fact located at address 15 in the new thread space.

Once a program has terminated, the QPU is immediately available to the QPU scheduler for a new program, which can be started back-back on the next instruction cycle.

For 3D fragment shader use, each QPU can execute two separate program threads if both the fragment shader programs are marked as threadable. Switching between threads is cooperative using Thread Switch signalled instructions.

When a running thread executes a Thread Switch instruction the thread stops running after a further two delays slot instructions and enters a suspended state. If there is a second suspended thread at this time, that second thread resumes running. When the running thread suspends and there is no second (suspended) thread, the QPU enters an idle state waiting for a new program to be started in the second thread. The final thread switch instruction of a shader must use the Last Thread Switch signal.

When a threadable program terminates with a Program End instruction, any new threadable program that is pending for that processor will be started immediately to follow on back-back from the terminating program. This will happen even if there is a second suspended thread present at the time.

Threads aren't left indefinitely in the suspended state with the QPU idle. It is assumed that a program suspends in order to give TMU lookup operations time to complete. If the QPU detects that a suspended thread's pending TMU requests are complete whilst the QPU is idle, that thread will resume running. The QPU must be idle for this to happen, so a suspended thread will not resume if the other thread finishes one program and starts a new program back-back.

The QPU can only run two threads for programs that are marked as threadable. The QPU scheduler manages the allocation of programs to QPUs automatically, ensuring that both threads have finished on a QPU that is running threadable programs before starting a non-threadable program on that QPU.

The scheduler algorithm automatically allocates threadable and non-threadable programs to QPUs already running the same type of program, attempting to avoid frequent changes of a QPU from running threadable programs to non-threadable programs and back again. For efficient sharing of the QPU processing resources it is usually best to use this automatic allocation.

For specific purposes the V3DQRESVx system registers can be used to exclude individual QPUs from running one or more of the following types of program: fragment shader, vertex shader, coordinate shader, or general-purpose program.

Inter-Processor Synchronisation

There are basic facilities for synchronisation of general-purpose programs that are running on multiple QPUs. There is a single hardware mutex shared between all QPUs, which is gained or released via reads or writes to a register mapped I/O location. There are also sixteen 4-bit counting semaphores, which are accessible by any QPU via a special instruction. Finally, QPUs may individually trigger an interrupt of the host processor.

Register-Mapped Input/Output

Each QPU has access to various data from the 3D hardware via register locations mapped into the regfile A and B address space. The following read and write interfaces are provided:

Varyings

Varyings, which are only of use in pixel shaders, appear in special register locations in the order they are to be consumed by the shader program. Pixel shaders read from a special ‘varyings’ register in the A or B register file space which returns the partially interpolated varying result, and also triggers a write to special accumulator r5 of the ‘C’ varyings coefficient.

Varyings data is accessed from a FIFO directly coupled to the QPU pipeline, which the per-slice varyings hardware will try and keep full.

Uniforms

Uniforms are 32-bit immediate values stored in lists in memory. Nominally, uniforms are stored in the order that they will be read by the QPU program.

A special uniforms base pointer in the QPU regfile space is initially set when a new thread is started, and auto-incremented every time the QPU reads a uniform. The uniforms are accessed by reading from a special register in the A/B regfile space. A uniforms cache and small FIFO in the QPU will keep prefetched uniform values ready for access.

The uniform base pointer can be written (from SIMD element 0) by the processor to reset the stream, there must be at least two nonuniform-accessing instructions following a pointer change before uniforms can be accessed once more.

Note: When ending a thread, the last instructions (including the program end instruction) must not access uniforms as at this time the new uniforms pointer for the next thread is being set up.

Texture and Memory Units

Each QPU has shared access of up to two Texture and Memory Units (TMU) units depending upon the v3d system configuration. Each TMU unit has an associated ‘request’ and ‘receive’ FIFO (per QPU). QPU requests for texels are passed through the request FIFO, and textured pixels are stored in the ‘receive’ FIFO ready for read back by the QPU.

Each of the parameters that can be written to specify a texture lookup have their own register assigned to them (s, t, r, and b). The ‘s’ parameter must be written last and signals to the hardware that all data has been written and it can increment the FIFO (and start accessing the data from memory).

Texture unit reads are signalled using the signaling field, the packed color data arriving in r4 ready for use by the following instruction.

For every texture lookup write that is performed (except for direct memory address lookups), a 32-bit uniform (containing the texture setup information) is also automatically read at the same time and pushed into the texture FIFO. This means that a texture lookup instruction writing to a texture unit must not also be used to fetch a uniform.

When the system is configured with two TMUs per slice, sharing of the texture units is facilitated by automatically swapping the addressing of the two TMUs when accessed by QPU2 and QPU3. This swapping may be disabled for a particular shader to improve local cache coherency, if the shader itself is using both TMUs equally.

Note: The swap must be disabled at least three instructions before the first TMU access in a program.

Special Functions Unit

Each QPU has shared access to a Special Functions Unit (SFU) which can perform several less frequently used ‘exotic’ operations (SQRT, RECIPSQRT, LOG, EXP).

Each exotic operation has its own special register in the A/B regfile space, writing to this register performs the specific function on the write data, and returns the result in r4 ready for use by the 3rd instruction after the write.

Note: When performing an SFU operation, r4 must not be accessed (read or written) in the 2 instructions after the SFU write:

- Instruction N+0 Write to SFU
- Instruction N+1 Don’t touch r4
- Instruction N+2 Don’t touch r4
- Instruction N+3 Instruction can read r4 (SFU result)

Vertex Pipeline Memory

The Vertex Pipeline Memory (VPM) cannot be accessed during pixel shading. To access the VPM, the QPU writes a base address and setup register to 'program' the VPM read/write. For reads, the VPM hardware will try and keep the QPU FIFO full by automatically incrementing its address pointer and writing data when there is space in the FIFO. Writes by the QPU to the VPM end up in the write FIFO, and a write will stall if the FIFO does not have space.

For reads, the user programs the exact number required. The VPM will automatically increment its base address and push data into the QPU read FIFO until it has completed the programmed number of reads after which it will stop supplying data to the read FIFO. The user must make sure that all reads that are programmed are 'consumed' by the shader prior to a program end. The QPU reads will stall whilst the FIFO is empty.

The VDW and VDR, which DMA data out of or into the VPM, are also programmed by writing setup registers which are accessed by the same I/O locations as used for VPM read and write setup. The QPU can then poll for DMA completion or just stall until the DMA is complete by reading other IO locations.

Tile Buffer

The Tile Buffer (TLB) consists of color, Z, stencil, and alphamask buffers. The QPU has direct read and write access to both the color and alphamask buffers, but access to the combined Z and Stencil buffer is more specialised. For normal GL use the QPU can only write a 24-bit Z value to the Z buffer. This causes all Z and Stencil tests and buffer updates to be performed in hardware, and also results in the QPUs multisample mask to be updated. For VG use the QPU can read a coverage level or mask from the contents of the Z and stencil buffer.

Color, alphamask, or coverage buffer reads are signalled using the signaling bits. On the write phase of the signaling instruction, the relevant tile buffer data arrives and is written to accumulator r4. The following instruction can then use the data. The tile buffer writes are simply a matter of writing to the appropriate I/O register.

Inter-Processor Mutex

There is a single mutex shared between all QPUs. To gain access to the mutex the QPU reads the mutex I/O register, and the instruction will stall until access to the mutex is granted. The read data is the default unmapped I/O location read data, which is the SIMD element_number or qpu_number. The mutex is released by a write to the mutex I/O register.

Host Interrupt

Each QPU can individually trigger a host processor interrupt by writing to the host_interrupt I/O location.

Processor Stalls

All accesses to external hardware (TLB, VPM, SFU, TMU, Varyings, Uniforms, Mutex, Semaphore, or Scoreboard wait) can potentially stall the QPU at the 'SP' stage of the pipeline.

For all I/O reads the processor will either have had a full 4-cycles to arbitrate for access to shared hardware (for example, TLB color reads) or is expecting to read or write one if its own FIFOs (Uniforms, Varyings and VPM), and hence can determine in the SP stage whether to stall the front end of the pipeline (cache fetch) if either access to hardware was not granted, or a FIFO is full or empty. The back end of the pipeline is self draining, so stalling does not affect any instructions further up the pipe than the currently stalling one.

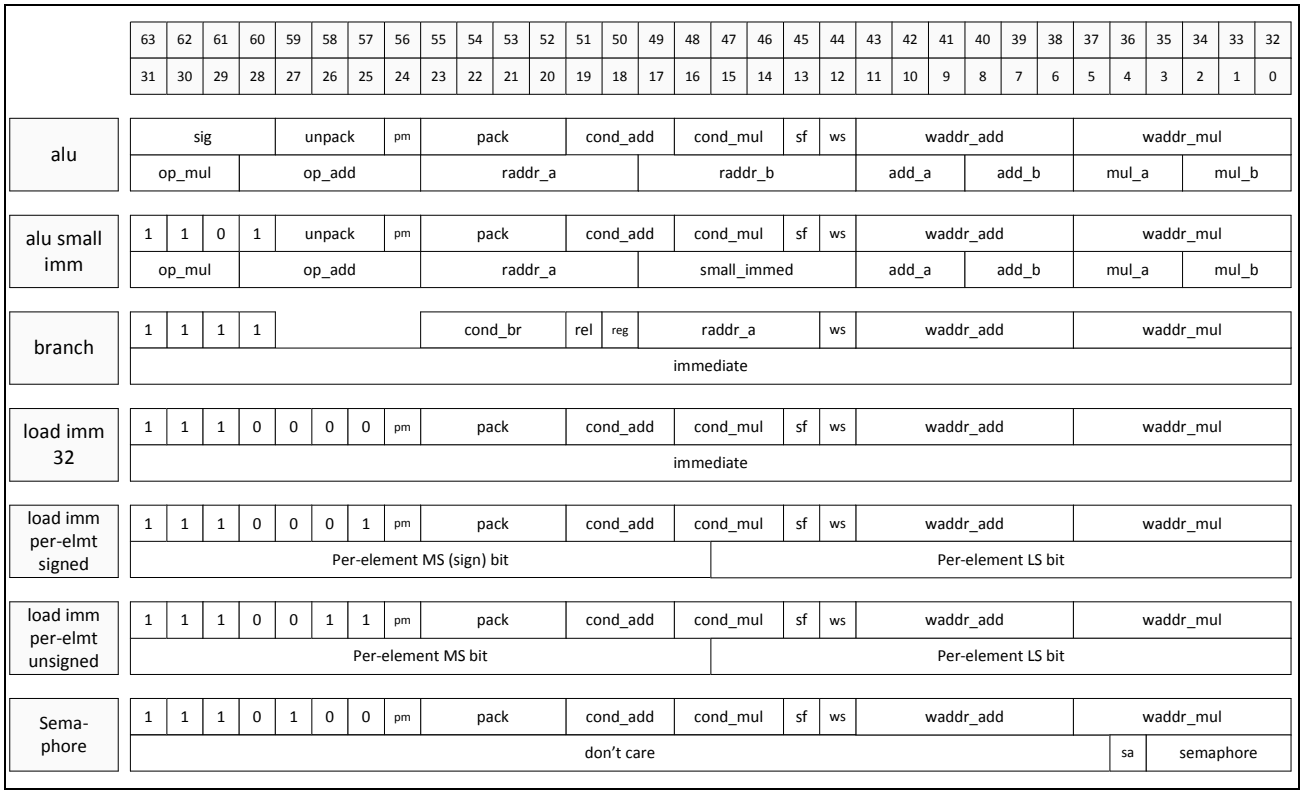
For writes, parts of the back end pipeline are stalled. For example, a write request to the tile buffer isn't acknowledged until three cycles later, so in the meantime three stages of the instruction pipeline are executed. Only those pipeline stages which are executing the current (stalling) instruction are stalled however, so again all instructions already in the pipeline will complete.

Per instruction, only one access to a closely-coupled hardware unit is allowed (either a signalled read, or write) with the exception of varyings, vpm, and uniforms, which can be accessed at any time (as they are aware of pipeline state and will stall if the QPU pipeline stalls).

QPU Instruction Encoding

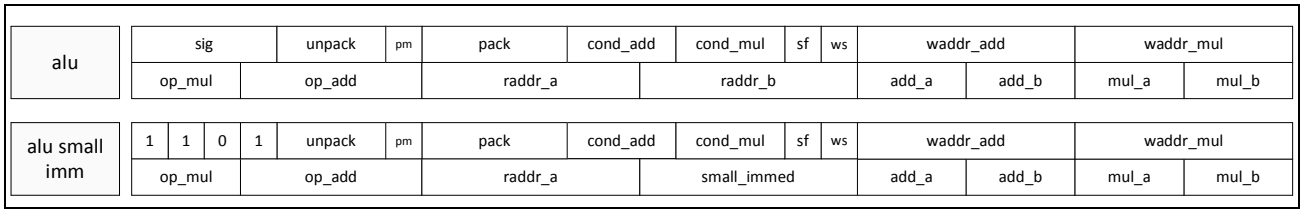
Figure 3 gives the instruction encoding for the four varieties of QPU instruction.

Figure 3: QPU Instruction Encoding



ALU Instructions

Figure 4: ALU Instruction Encoding



An ALU instruction reads up to one value from each register file (or a value from register file A and an immediate), performs an operation in each of the add and multiply units, and writes the results back to the register files or accumulators, optionally updating the flags. The encoding contains the following fields:

Table 1: ALU Instruction Fields

Field	*Bits*	*Description*
sig	4	Signaling bits (see "Signaling Bits" on page 29)
unpack	3	Unpack mode (see "Pack/Unpack Bits" on page 30)
pm	1	Pack/Unpack select (see "Pack/Unpack Bits" on page 30)
pack	4	Pack mode (see "Pack/Unpack Bits" on page 30)
cond_add	3	Add ALU condition code (see "ALU Input Muxes" on page 28)
cond_mul	3	Mul ALU condition code (see "ALU Input Muxes" on page 28)
sf	1	Set flags Flags are updated from the add ALU unless the add ALU performed a NOP (or its condition code was NEVER) in which case flags are updated from the mul ALU
ws	1	Write swap for add and multiply unit outputs If ws = 0, add alu writes to regfile a, mult to regfile b If ws = 1, add alu writes to regfile b, mult to regfile a
waddr_add	6	Write address for add output If ws = 0, regfile writes go to regfile a, else regfile b
waddr_mul	6	Write address for multiply output If ws = 0, regfile writes go to regfile b, else regfile a
op_mul	3	Multiply opcode (see "Op Mul" on page 36)
op_add	5	Add opcode (see "Op Add" on page 35)
raddr_a	6	Read address for register file a
raddr_b	6	Read address for register file b
small_immed	6	Small immediate or specified vector rotation for mul ALU output (see "ALU Input Muxes" on page 28)
add_a	3	Input mux control for A port of add ALU (A add operand) (see "ALU Input Muxes" on page 28)
add_b	3	Input mux control for B port of add ALU (B add operand) (see "ALU Input Muxes" on page 28)
mul_a	3	Input mux control for A port of mul ALU (A mul operand) (see "ALU Input Muxes" on page 28)
mul_b	3	Input mux control for B port of mul ALU (B mul operand) (see "ALU Input Muxes" on page 28)

Each ALU is given its own opcode, and each will execute conditionally based on its own set of condition bits. ALU Instructions and their specific opcode are specified in "QPU Instruction Set" on page 35.

Condition Codes

The QPU keeps a set of N, Z and C flag bits per 16 SIMD element. These flags are updated based on the result of the ADD ALU if the 'sf' bit is set. If the sf bit is set and the ADD ALU executes a NOP or its condition code was NEVER, flags are set based upon the result of the MUL ALU result.

The cond_add and cond_mul fields specify the following conditions:

Table 2: cond_add/cond_mul Conditions

Value	*Condition Code*
0	Never (NB gates ALU – useful for LDI instructions to save ALU power)
1	Always
2	ZS (Z set)
3	ZC (Z clear)
4	NS (N set)
5	NC (N clear)
6	CS (C set)
7	CC (C clear)

ALU Input Muxes

The add_a, add_b, mul_a, and mul_b fields specify the input data for the A and B ports of the ADD and MUL pipelines, respectively.

Table 3: ALU Input Mux Encoding

Value	*Meaning*
0	Accumulator r0
1	Accumulator r1
2	Accumulator r2
3	Accumulator r3
4	Accumulator r4[a]
5	Accumulator r5*
6	Use value from register file A
7	Use value from register file B

a. * The accumulators r4 and r5 have special functions and cannot be used as general-purpose accumulator registers.

Signaling Bits

The 4-bit signaling field signal is connected to the 3d pipeline and is set to indicate one of a number of conditions to the 3d hardware. Values from this field are also used to encode a 'BKPT' instruction, and to encode Branches and Load Immediate instructions.

Table 4: ALU Signaling Bits

Value	*Meaning*
0	Software Breakpoint
1	No Signal
2	Thread Switch (not last)
3	Program End (Thread End)
4	Wait for Scoreboard (stall until this QPU can safely access tile buffer)
5	Scoreboard Unlock
6	Last Thread Switch
7	Coverage load from Tile Buffer to r4
8	Color Load from Tile Buffer to r4
9	Color Load and Program End
10	Load (read) data from TMU0 to r4
11	Load (read) data from TMU1 to r4
12	Alpha-Mask Load from Tile Buffer to r4
13	ALU instruction with raddr_b specifying small immediate or vector rotate
14	Load Immediate Instruction
15	Branch Instruction

The explicit Wait for Scoreboard signal (4) is not required in most fragment shaders, because the QPU will implicitly wait for the scoreboard on the first instruction that accesses the tile buffer.

Small Immediate

The 6 bit small immediate field encodes either an immediate integer/float value used in place of the register file b input, or a vector rotation to apply to the mul ALU output, according to Table 5.

Table 5: Small Immediate Encoding

Value	*Encoding*
0	0
1	1
…	…
14	14
15	15
16	-16

Table 5: Small Immediate Encoding (Cont.)

Value	*Encoding*
17	-15
…	…
30	-2
31	-1
32	0x3f800000 (1.0)
33	0x40000000 (2.0)
…	…
38	0x42800000 (64.0)
39	0x43000000 (128.0)
40	0x3b800000 (1.0/256.0)
41	0x3c000000 (1.0/128.0)
…	…
46	0x3e800000 (1.0/4.0)
47	0x3f000000 (1.0/2.0)
48	Mul output vector rotation is taken from accumulator r5, element 0, bits [3:0]
49	Mul output vector rotated by 1 upwards (so element 0 moves to element 1)
50	Mul output vector rotated by 2 upwards (so element 0 moves to element 2)
…	…
62	Mul output vector rotated by 14 upwards (so element 0 moves to element 14)
63	Mul output vector rotated by 15 upwards (so element 0 moves to element 15)

Pack/Unpack Bits

Normally, the Pack and Unpack fields program the A register file pack/unpack blocks. The A-regfile unpack block will convert packed 8 or 16 bit data to 32 bit values ready for use by the ALUs. Similarly the a-regfile pack block allows the 32-bit ALU result to be packed back into the a-regfile as 8 or 16 bit data.

As well as the a-regfile pack and unpack units, accumulator r4 has a more limited unpack unit which can be used to unpack the color values returned by the tile buffer and texture unit.

Finally, the mul ALU has the ability to convert its floating point result to 8-bit color *c*:

c = sat[round(f * 255)] (sat saturates to [255, 0])

If the pm bit is set, the unpack field programs the r4 unpack unit, and the pack field is used to program the color conversion on the output of the mul unit (that is, enable the conversion and program which byte in the destination regfile/accumulator to write the result to).

Regfile-a unpack operations (pm bit = 0):

Table 6: Regfile-a Unpack Encoding

UNPACK	*Operation*	*Notes*
0	32→32	No unpack (NOP)
1	16a→32	Float16float32 if any ALU consuming data executes float instruction, else signed int16→signed int32
2	16b→32	Float16float32 if any ALU consuming data executes float instruction, else signed int16→signed int32
3	8d→32	Replicate ms byte (alpha) across word: result = {8d, 8d, 8d, 8d}
4	8a→32	8-bit color value (in range [0, 1.0]) to 32 bit float if any ALU consuming data executes float instruction, else unsigned int8 → int32
5	8b→32	8-bit color value (in range [0, 1.0]) to 32 bit float if any ALU consuming data executes float instruction, else unsigned int8 → int32
6	8c→32	8-bit color value (in range [0, 1.0]) to 32 bit float if any ALU consuming data executes float instruction, else unsigned int8 → int32
7	8d→32	8-bit color value (in range [0, 1.0]) to 32 bit float if any ALU consuming data executes float instruction, else unsigned int8 → int32

Regfile-a pack operations (pm bit = 0):

Table 7: Regfile-a Pack Encoding

PACK	*Operation*	*Notes*
0	32→32	No pack (NOP)
1	32→16a	Convert to 16 bit float if input was float result, else convert to int16 (no saturation, just take ls 16 bits)
2	32→16b	Convert to 16 bit float if input was float result, else convert to int16 (no saturation, just take ls 16 bits)
3	32→8888	Convert to 8-bit unsigned int (no saturation, just take ls 8 bits) and replicate across all bytes of 32-bit word
4	32→8a	Convert to 8-bit unsigned int (no saturation, just take ls 8 bits)
5	32→8b	Convert to 8-bit unsigned int (no saturation, just take ls 8 bits)
6	32→8c	Convert to 8-bit unsigned int (no saturation, just take ls 8 bits)
7	32→8d	Convert to 8-bit unsigned int (no saturation, just take ls 8 bits)
8	32→32	Saturate (signed) 32-bit number (given overflow/carry flags)
9	32→16a	Convert to 16 bit float if input was float result, else convert to signed 16 bit integer (with saturation)
10	32→16b	Convert to 16 bit float if input was float result, else convert to signed 16 bit integer (with saturation)
11	32→8888	Convert to 8-bit unsigned int (with saturation) and replicate across all bytes of 32-bit word
12	32→8a	Convert to 8-bit unsigned int (with saturation)
13	32→8b	Convert to 8-bit unsigned int (with saturation)

Table 7: Regfile-a Pack Encoding (Cont.)

PACK	Operation	Notes
14	32→8c	Convert to 8-bit unsigned int (with saturation)
15	32→8d	Convert to 8-bit unsigned int (with saturation)

R4 unpack operations (pm bit = 1):

Table 8: R4 Pack Encoding

UNPACK	Operation	Notes
0	32→32	No unpack (NOP)
1	16a→32	Float16float32
2	16b→32	Float16float32
3	8d→32	Replicate ms byte (alpha) across word: result = {8d, 8d, 8d, 8d}
4	8a→32	8-bit color value (in range [0, 1.0]) to 32 bit float
5	8b→32	8-bit color value (in range [0, 1.0]) to 32 bit float
6	8c→32	8-bit color value (in range [0, 1.0]) to 32 bit float
7	8d→32	8-bit color value (in range [0, 1.0]) to 32 bit float

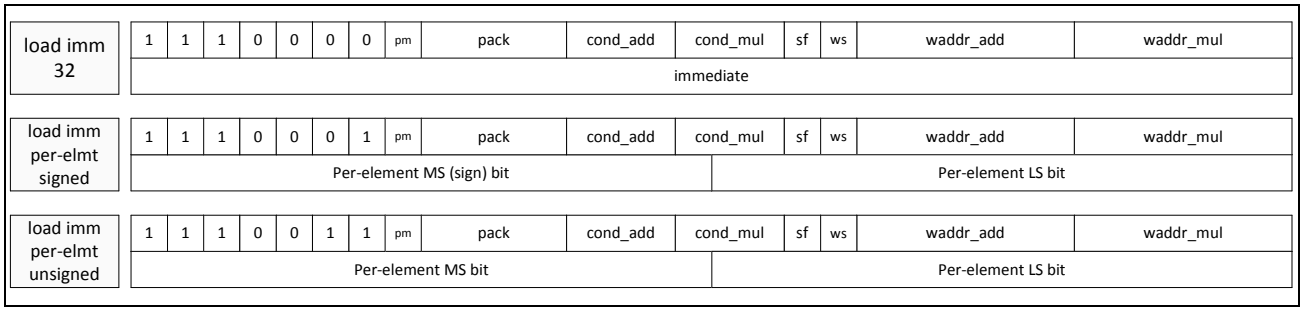
MUL ALU pack operations (pm bit = 1):

Table 9: MUL ALU Pack Encoding

PACK	Operation	Notes
0	32→32	No pack (NOP)
1–2	Reserved	–
3	32→8888	Convert mul float result to 8-bit color in range [0, 1.0] and replicate across all bytes of 32-bit word.
4	32→8a	Convert mul float result to 8-bit color in range [0, 1.0]
5	32→8a	Convert mul float result to 8-bit color in range [0, 1.0]
6	32→8a	Convert mul float result to 8-bit color in range [0, 1.0]
7	32→8a	Convert mul float result to 8-bit color in range [0, 1.0]
8–15	Reserved	–

Load Immediate Instructions

Figure 5: Load Immediate Instruction Encoding



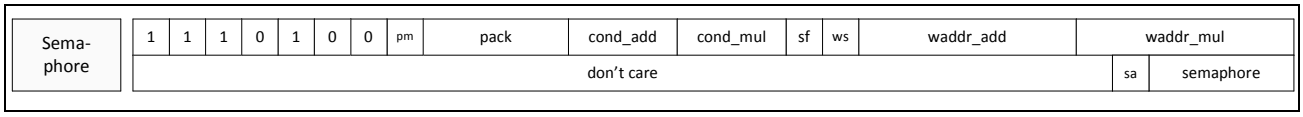
The load immediate instructions can be used to write either a 32-bit immediate across the entire SIMD array, or 16 individual 2-bit (signed or unsigned integer) values per-element.

The encoding contains identical fields to the ALU instructions in the upper 32-bits, while the lower 32 bits contain the immediate value(s) instead of the add and mul opcodes and read/mux fields.

When a load immediate instruction is encountered, the processor feeds the immediate value into the add and mul pipes and sets them to perform a 'mov'. The immediate value turns up at the output of the ALUs as if it were just a normal arithmetic result and hence all of the write fields, conditions and modes (specified in the upper 32-bits of the encoding) work just as they would for a normal ALU instruction.

Semaphore Instruction

Figure 6: Semaphore Instruction Encoding



The dedicated semaphore instruction provides each QPU with access to one of 16 system wide 4-bit counting semaphores. The semaphore accessed is selected by the 4-bit semaphore field. The semaphore is incremented if sa is 0 and decremented if sa is 1. The QPU stalls if it is attempting to decrement a semaphore below 0 or increment it above 15. The QPU may also stall briefly during arbitration access to the semaphore.

The instruction otherwise behaves like a 32-bit load immediate instruction, so the ALU outputs will not generally be useful.

Note: This instruction can stall due to external arbitration, the waddr_add and waddr_mul must not address closely coupled peripherals that can stall, that is, TLB, TMU, or SFU.

Branch Instruction

Figure 7: Branch Instruction Encoding

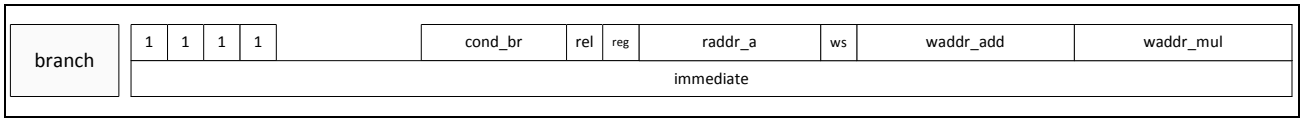


Table 10: Branch Instruction Fields

Field	*Bits*	*Description*
cond_br	4	Branch condition
rel	1	Branch relative. If set, branch target is relative to PC+4 (add PC+4 to target)
reg	1	Add value of raddr_a (value read from SIMD element 0) to branch target.
raddr_a	5	Read address for register file a
waddr_add	6	Write address for ADD ALU (same as ALU instruction)
waddr_mul	6	Write address for MUL ALU (same as ALU instruction)
ws	1	Write swap bit (same as ALU instruction)
immediate	32	Signed 32-bit immediate (always added to branch target – set to 0 if not used)

QPU branches are conditional based on the status of the ALU flag bits across all 16 elements of the SIMD array. If a branch condition is satisfied, a new program counter value is calculated as the sum of the (signed) immediate field, the current PC+4 (if the rel bit is set) and the value read from the a register file SIMD element 0 (if the reg bit is set).

On branch instructions the link address (the current instruction plus four) appears at the output of the add and mul ALUs (in the same way that immediates are passed through these units for load immediate instructions), and therefore may be written to a register-file location to support branch-with-link functionality.

For simplicity, the QPUs do not use branch prediction and never cancel the sequentially fetched instructions when a branch is encountered. This means that three ‘delay slot’ instructions following a branch instruction are always executed.

The list of possible branch conditions is given in Table 11.

Table 11: Branch Conditions

cond_br	*Condition*	*Notes*
0	&{Z[15:0]}	All Z flags set
1	&{~Z[15:0]}	All Z flags clear
2	|{Z[15:0]}	Any Z flags set
3	|{~Z[15:0]}	Any Z flags clear
4	&{N[15:0]}	All N flags set
5	&{~N[15:0]}	All N flags clear
6	|{N[15:0]}	Any N flags set

Table 11: Branch Conditions (Cont.)

cond_br	Condition	Notes
7	|{~N[15:0]}	Any N flags clear
8	&{C[15:0]}	All C flags set
9	&{~C[15:0]}	All C flags clear
10	|{C[15:0]}	Any C flags set
11	|{~C[15:0]}	Any C flags clear
12	Reserved	N/A
13	Reserved	N/A
14	Reserved	N/A
15	AL	Always execute (unconditional)

QPU Instruction Set

Op Add

Table 12: Op Add Instructions

Instruction	opcode	Description
nop	0	No operation
fadd	1	Floating point add
fsub	2	Floating point subtract
fmin	3	Floating point min
fmax	4	Floating point max
fminabs	5	Floating point min of absolute values
fmaxabs	6	Floating point max of absolute values
ftoi	7	Floating point to signed integer
itof	8	Signed integer to floating point
–	9–11	Reserved
add	12	Integer add
sub	13	Integer subtract
shr	14	Integer shift right
asr	15	Integer arithmetic shift right
ror	16	Integer rotate right
shl	17	Integer shift left
min	18	Integer min
max	19	Integer max
and	20	Bitwise AND

Table 12: Op Add Instructions (Cont.)

Instruction	opcode	Description
or	21	Bitwise OR
xor	22	Bitwise Exclusive OR
not	23	Bitwise NOT
clz	24	Count leading zeros
–	25–29	Reserved
v8adds	30	Add with saturation per 8-bit element
v8subs	31	Subtract with saturation per 8-bit element

Op Mul

Table 13: Op Mul Instructions

Instruction	opcode	Description
nop	0	No operation
fmul	1	Floating point multiply
mul24	2	24 bit multiply
V8muld	3	Multiply two vectors of 4 8-bit values in the range [1.0, 0]
V8min	4	Return minimum value per 8-bit element
V8max	5	Return maximum value per 8-bit element
V8adds	6	Add with saturation per 8-bit element
V8subs	7	Subtract with saturation per 8-bit element

Summary of Instruction Restrictions

There are several restrictions on where certain instruction operations may be placed in a program. There are restrictions at the start and end of a program, general restrictions on instruction sequences and restrictions on simultaneous access to closely coupled peripherals in the same instructions. These restrictions are summarized in the following list:

- The last three instructions of any program (Thread End plus the following two delay-slot instructions) must not do varyings read, uniforms read or any kind of VPM, VDR, or VDW read or write.
- The Thread End instruction must not write to either physical regfile A or B.
- The Thread End instruction and the following two delay slot instructions must not write or read address 14 in either regfile A or B.
- The final program instruction (the second delay slot instruction) must not do a TLB Z write.
- A scoreboard wait must not occur in the first two instructions of a fragment shader. This is either the explicit Wait for Scoreboard signal or an implicit wait with the first tile-buffer read or write instruction.
- If TMU_NOSWAP is written, the write must be three instructions before the first TMU write instruction. For example, if TMU_NOSWAP is written in the first shader instruction, the first TMU write cannot occur before the 4^{th} shader instruction.
- An instruction must not read from a location in physical regfile A or B that was written to by the previous instruction.
- After an SFU lookup instruction, accumulator r4 must not be read in the following two instructions. Any other instruction that results in r4 being written (that is, TMU read, TLB read, SFU lookup) cannot occur in the two instructions following an SFU lookup.
- An instruction that does a vector rotate by r5 must not immediately follow an instruction that writes to r5.
- An instruction that does a vector rotate must not immediately follow an instruction that writes to the accumulator that is being rotated.
- After an instruction that does a TLB Z write, the multisample mask must not be read as an instruction input argument in the following two instruction. The TLB Z write instruction can, however, be followed immediately by a TLB color write.
- A single instruction can only perform a maximum of one of the following closely coupled peripheral accesses in a single instruction: TMU write, TMU read, TLB write, TLB read, TLB combined color read and write, SFU write, Mutex read or Semaphore access.

QPU Register Address Map

Table 14 provides the complete QPU register address map. Note that read and write typically have differing functions, and some infrequently used registers are only mapped to the A or B register address space.

Table 14: QPU Register Address Map

Addr	*A rd*	*B rd*	*A wr*	*B wr*
0–31	regfile A	regfile B	regfile A	regfile B
32	UNIFORM_READ	UNIFORM_READ	ACC0	ACC0

Table 14: QPU Register Address Map (Cont.)

Addr	*A rd*	*B rd*	*A wr*	*B wr*
33			ACC1	ACC1
34			ACC2	ACC2
35	VARYING_READ	VARYING_READ	ACC3	ACC3
36			TMU_NOSWAP	TMU_NOSWAP
37			ACC5 (Replicate pixel 0 per quad)	ACC5 (Replicate SIMD element 0)
38	ELEMENT_NUMBER	QPU_NUMBER	HOST_INT	HOST_INT
39	NOP (no read)	NOP (no read)	NOP (no write)	NOP (no write)
40			UNIFORMS_ADDRESS	UNIFORMS_ADDRESS
41	X_PIXEL_COORD	Y_PIXEL_COORD	QUAD_X	QUAD_Y
42	MS_FLAGS	REV_FLAG	MS_FLAGS	REV_FLAG
43			TLB_STENCIL_SETUP	TLB_STENCIL_SETUP
44			TLB_Z	TLB_Z
45			TLB_COLOUR_MS	TLB_COLOUR_MS
46			TLB_COLOUR_ALL	TLB_COLOUR_ALL
47			TLB_ALPHA_MASK	TLB_ALPHA_MASK
48	VPM_READ	VPM_READ	VPM_WRITE	VPM_WRITE
49	VPM_LD_BUSY	VPM_ST_BUSY	VPMVCD_RD_SETUP	VPMVCD_WR_SETUP
50	VPM_LD_WAIT	VPM_ST_WAIT	VPM_LD_ADDR	VPM_ST_ADDR
51	MUTEX_ACQUIRE	MUTEX_ACQUIRE	MUTEX_RELEASE	MUTEX_RELEASE
52			SFU_RECIP	SFU_RECIP
53			SFU_RECIPSQRT	SFU_RECIPSQRT
54			SFU_EXP	SFU_EXP
55			SFU_LOG	SFU_LOG
56			TMU0_S (RETIRING)	TMU0_S (Retiring)
57			TMU0_T	TMU0_T
58			TMU0_R	TMU0_R
59			TMU0_B	TMU0_B
60			TMU1_S (RETIRING)	TMU1_S (Retiring)
61			TMU1_T	TMU1_T
62			TMU1_R	TMU1_R
63			TMU1_B	TMU1_B

Section 4: Texture and Memory Lookup Unit

Each QPU has shared access to one or two (depending upon the block configuration) per-slice Texture and Memory Lookup Unit (TMUs). TMU fetches are initiated by writing setup and sample-index data to QPU I/O register locations, and filtered texture data is fetched into QPU register r4 with a signaling instruction.

Each TMU is 'virtualized' and therefore can theoretically sample from an unlimited number of textures in any given shader. The shader program passes the TMU all the setup data it needs (texture base address, format wrap modes etc.) at the same time as the sampling information using uniforms.

Each TMU performs programmable filtering and automatic level-of-detail (LOD) determination when mipmap textures are used. The texture units support all required OpenGL-ES 1.x and 2.0 texture modes and formats, plus the OpenGL-ES 2.0 Ericsson Texture Compression (ETC1) compressed texture format. 64-bit 'HDR' textures are also supported, where each color channel is represented by a 16-bit float value (in 1.5.10 format). Full bilinear and trilinear filtering are supported for HDR textures just as for normal textures.

The units also support blended 'generic' 8-bit integer and 16-bit float textures, plus point sampled 16-bit integer textures. Furthermore, the texture units support a 1-bpp black and white image format, 4bpp and 1bpp alpha formats as well as CLAMP TO BORDER wrap/clamp mode. The unit also has a mode whereby the (s,t) parameters are mapped to a programmable window on the texture image (a 'child image'), rather than using the entire texture image. These extra features are available to make the texture unit more suitable for use by OpenVG implementations.

The memory organization of most texture types is T-format or LT-format, with LT-format automatically selected for smaller size textures. The texture units also support raster format 32-bit YUYV and RGBA textures, allowing video and image data to be directly used as a texture without needing conversion to T-format. Finally, the TMUs can also be used for general 32-bit data lookup using a direct address.

The TMUs require mipmaps to be stored before the level 0 texture in memory, with addresses automatically determined assuming power of 2 sized images.The TMU unit supports mipmapped non-power-of-two textures, but the mipmap levels greater than 0 must be padded to fit within a power of 2 container image.

QPU Interface

Each TMU has associated with it a 'request' (TFREQ) and 'receive' (TFRCV) FIFO per QPU. QPUs post requests to the request FIFO, which the TMU processes. Textured pixels are returned to the receive FIFO and from there the QPU can fetch the results. Note that for all texture types except RGBA64 the QPU only needs to make one read request as pixel data is returned as 32-bit packed RGBA8888. For RGBA64 texture data, two read requests are required, the first to load the packed 1616 RG data, the second to load the packed BA data.

The TFREQ input FIFO holds two full lots of s, t, r, b data, plus associated setup data, per QPU, that is, there are eight data slots. For each texture request, slots are only consumed for the components of s, t, r, and b actually written. Thus the FIFO can hold four requests of just (s, t) data, or eight requests of just s data (for direct addressed data lookups).

Note that there is one FIFO per QPU, and the FIFO has no concept of threads - that is, multi-threaded shaders must be careful to use only 1/2 the FIFO depth before reading back. Multi-threaded programs must also therefore always thread switch on texture fetch as the other thread may have data waiting in the FIFO.

Since the maximum number of texture requests in the input (TFREQ) FIFO is four lots of (s, t) data, the output (TFRCV) FIFO is sized to holds four lots of max-size color data per QPU. For non-float color, reads are packed RGBA8888 data (one read per pixel). For 16-bit float color, two reads are necessary per pixel, with reads packed as RG1616 then BA1616. So per QPU there are eight color slots in the TFRCV FIFO.

In systems configured with two TMUs per slice, the TMU selected by the I/O register address is automatically swapped for accesses by QPU2 and QPU3. Thus the TMUs are adequately shared if all shaders address only TMU0. The automatic swapping may be disabled per shader/thread by writing 1 to the TMU_SWAP I/O register, if the shader itself distributes texture lookups evenly between the two TMUs. This may help texture cache coherency in the two TMUs.

Texture Data Storage

Texture data is stored in system memory and the TMU automatically fetches data via its internal cache and the 3D system level-2 cache (L2C). The texture's base pointer gives the start address of the LOD0 image data. The mipmaps are stored in reverse order before this in memory. This is so that the LOD0 image does not need to be padded to a power of two size for non-power-of two images, saving significant memory – the remaining smaller mipmaps do need to be padded to a power of two.

The hardware automatically determines type of image (T-format or LT-format) by looking at LOD dimensions. The hardware assumes a level is in T-format unless either the width or height for the level is less than one T-format tile. In this case use the hardware assumes the level is stored in LT-format. T-format and LT-format are defined in "Texture Memory Formats" on page 105.

For cube mapped images, the setup parameters provide a stride to get to the nth face data. Border color data (for the clamp to border wrap mode) isn't set in the configuration data but is encoded at the very end of the texture image data, and is treated just like an extra mipmap level.

Texture and Memory Lookup Unit Setup

For each write of texture unit sampling vector data (t, b, etc.) a uniform is automatically read and is passed to the texture unit through the request FIFO. These config parameters are used to set up all state for the texture being accessed. The 's' parameter must be written last as it triggers a texture FIFO to accept the data and start processing it. All other parameters (t, r, b) can be written in any order. Params that are not written will be treated as zero. Table 15, Table 16, and Table 17 give details of the config parameters.

Uniforms associated with the first 2 TMU data writes set up common things (as we always have to write at least two parameters). The rest of the config setup data can be used to either set up cube map stride, or child image parameters (or not used if only two parameters are required). Note that for 2D child image setup it is necessary to write all four sampling registers, even though the 'r' and (possibly) 'b' parameters are not used (and should be set to zero).

Border color is set by writing the ‘r’ parameter - QPU SIMD element 0 is used to set this. If ‘r’ is not written, the border color will be 0. If cube mapping is enabled, then CLAMP TO BORDER mode should not be used (in the cube map case it doesn’t make sense).

Note: If you write child image setup data then the hardware assumes child image mode.

General-memory lookups are performed by writing to just the ‘s’ parameter, using the absolute memory address. In this case no uniform is read. General-memory lookups always return a 32-bit value, and the bottom two bits of the address are ignored.

Table 15: Texture Config Parameter 0

Config Parameter		***Parameter 0***	
Bits	***Name***	***Description***	***Default***
31:12	BASE	Texture Base Pointer (in multiples of 4Kbytes).	-
11:10	CSWIZ	Cache Swizzle	-
9	CMMODE	Cube Map Mode	-
8	FLIPY	Flip Texture Y Axis	-
7:4	TYPE	Texture Data Type	-
3:0	MIPLVLS	Number of Mipmap Levels minus 1	-

Table 16: Texture Config Parameter 1

Config Parameter		***Parameter 1***	
Bits	***Name***	***Description***	***Default***
31	TYPE4	Texture Data Type Extended (bit 4 of texture type)	-
30:20	HEIGHT	Image Height (0 = 2048)	-
19	ETCFLIP	Flip ETC Y (per block)	-
18:8	WIDTH	Image Width (0 = 2048)	-
7	MAGFILT	Magnification Filter	-
6:4	MINFILT	Minification Filter	-
3:2	WRAP_T	T Wrap Mode (0, 1, 2, 3 = repeat, clamp, mirror, border)	-
1:0	WRAP_S	S Wrap Mode (0, 1, 2, 3 = repeat, clamp, mirror, border)	-

Table 17: Texture Config Parameters 2 and 3

Config Parameter		*Parameter 2 and 3*	
Bits	***Name***	***Description***	***Default***
31:30	PTYPE	Determines meaning of rest of parameter: 0 = Not Used (for example, for 2D textures + bias) 1 = Cube Map Stride 2 = Child Image Dimensions 3 = Child Image Offsets	-
PTYPE = 1 (Cube Map Stride)			
29:12	CMST	Cube Map Stride (in multiples of 4 Kbytes)	0
11:1	–	Reserved	0
0	BSLOD	Disable automatic LOD, use bias only	0
PTYPE = 2 (Child Image Dimensions)			
29:23	–	Reserved	0
22:12	CHEIGHT	Cube Map Stride (in multiples of 4Kbytes)	0
11	–	Reserved	0
10:0	CWIDTH	Disable automatic LOD, use bias only	0
PTYPE = 3 (Child Image Offsets)			
29:23	–	Reserved	0
22:12	CYOFF	Child Image Y Offset	0
0	–	Reserved	0
10:0	CXOFF	Child Image X Offset	0

Texture Data Types

The TMU can read image data with various pixel formats. Table 18 details the supported texture image formats.

Table 18: Texture Data Types

Num	***TYPE***	***Bpp***	***Description***
0	RGBA8888	32	8-bit per channel red, green, blue, alpha
1	RGBX8888	32	8-bit per channel RGA, alpha set to 1.0
2	RGBA4444	16	4-bit per channel red, green, blue, alpha
3	RGBA5551	16	5-bit per channel red, green, blue, 1-bit alpha
4	RGB565	16	Alpha channel set to 1.0
5	LUMINANCE	8	8-bit luminance (alpha channel set to 1.0)
6	ALPHA	8	8-bit alpha (RGA channels set to 0)
7	LUMALPHA	16	8-bit luminance, 8-bit alpha
8	ETC1	4	Ericsson Texture Compression format

Table 18: Texture Data Types (Cont.)

Num	TYPE	Bpp	Description
9	S16F	16	16-bit float sample (blending supported)
10	S8	8	8-bit integer sample (blending supported)
11	S16	16	16-bit integer sample (point sampling only)
12	BW1	1	1-bit black and white
13	A4	4	4-bit alpha
14	A1	1	1-bit alpha
15	RGBA64	64	16-bit float per RGBA channel
16	RGBA32R	32	Raster format 8-bit per channel red, green, blue, alpha
17	YUYV422R	32	Raster format 8-bit per channel Y, U, Y, V

Texture Filter Types

The MINFILT and MAGFILT fields in the setup parameters select the minification and magnification filter type. The minification filter determines what happens if a texture is minified (one screen pixel maps to more than one texture pixel). The magnification filter determines what happens if a texture is magnified (several adjacent screen pixels map to the same texture pixel). Table 19 gives details.

Table 19: Texture Filter Types

Num	Filter	Description
MAGFILT (Magnification Filters):		
0	LINEAR	Sample 2x2 pixels and blend. (bilinear)
1	NEAREST	Sample nearest pixel (point sample)
MINFILT (Minification Filters):		
0	LINEAR	Bilinear sample from LOD 0 only
1	NEAREST	Sample nearest pixel in LOD 0 only
2	NEAR_MIP_NEAR	Sample nearest pixel from nearest LOD level
3	NEAR_MIP_LIN	Sample nearest pixel from nearest 2 LOD levels and blend
4	LIN_MIP_NEAR	Bilinear sample from nearest LOD level
5	LIN_MIP_LIN	Blend Bilinear samples from 2 nearest LOD levels (trilinear)

Texture Modes

The Texture Units support two different modes of operation. For each mode, the [s,t,r] parameters passed to the unit are interpreted differently.

Normal 2D Texture Mode

In normal 2D texture mode, [s, t] are floating-point texture coordinates which when clamped appropriately to [0, 1] according to the selected wrap mode, provide an (x, y) sample point within the two-dimensional texture. The r parameter is not used for normal 2D texturing. The bias parameter is added to the automatically calculated level of detail before mipmap selection and is only used if the shader program writes to the ‘b’ setup register.

Cube Map Mode

In cube mapping mode, per pixel we find a direction vector (rx, ry, rz) emanating from the centre of a cube surrounding our 3D object. Cube mapping uses this direction vector to first find which face of the cube the vector points through (major axis of direction, corresponding to one of the six cube map textures) and then calculates appropriate texture coordinates from the vector from which to sample the texture pixels. Section 3.8.6 of the OpenGL 2.0 specification goes into more detail.

Pixel shader code is expected to find the absolute value major axis of direction |ma| (which isthe largest of |rx| |ry| and |rz|). The shader then divides each of rx, ry, rz by this value to get rx0 ry0 and rz0. One of these values will become ±unity (indicating the major axis of direction).

[s, t, r] must contain values rx0 ry0 and rz0 respectively when the texture unit is in cube map mode.

Interface Registers

The following QPU register addresses are used for TMU access:

Table 20: QPU Register Addresses for TMU Access

Addr	*A rd*	*B rd*	*A wr*	*B wr*
36			TMU_NOSWAP	TMU_NOSWAP
56			TMU0_S (Retiring)	TMU0_S (Retiring)
57			TMU0_T	TMU0_T
58			TMU0_R	TMU0_R
59			TMU0_B	TMU0_B
60			TMU1_S (Retiring)	TMU1_S (Retiring)
61			TMU1_T	TMU1_T
62			TMU1_R	TMU1_R
63			TMU1_B	TMU1_B

The TMU interface registers are defined in the following tables:

Table 21: 2D Texture Lookup Coordinates

Register Name(s)		***TMU_NOSWAP***	
Synopsis		Texture unit swap disable	
Bits	***Name***	***Description***	***Default***
31:1	–	Unused	0
0	SWAPDISA	TMU Swap Disable Writing 1 disables the automatic swapping of the addressed TMU for this thread if this is QPU2 or QPU3 (if the system is configured with two TMUs per slice).	0

Table 22: 2D Texture Lookup Coordinates

Register Name(s)		***TMU_S/T***	
Synopsis		2D Texture Lookup Coordinates	
Bits	***Name***	***Description***	***Default***
31:0	S/T	Texture Coordinates These are scaled floating-point numbers such that S/T of [0, 1] maps to the full width/height of the texture.	0

Table 23: Cube Map Texture Lookup Coordinates

Register Name(s)		***TMU_S/T/R***	
Synopsis		Cube Map Texture Lookup Coordinates	
Bits	***Name***	***Description***	***Default***
31:0	Rx/Ry/Rz	Cube-map direction vector. [S,T,R] gives the floating-point normalized cube-map direction vector [Rx, Ry, Rz]. This is scaled such that the largest of |Rx|, |Ry|, |Rz| is 1.	0

Table 24: Texture Lookup LOD Bias

Register Name(s)		***TMU_B***		
Synopsis		Texture Lookup LOD Bias		
Bits	***Name***	***Description***	***Default***	***Bits***
31:0	BIAS	LOD Bias. Floating point bias added to the automatically calculated mipmap LOD.	0	

Section 5: Tile Buffer

All of the QPUs share access to a single tile buffer (TB) to update the Color, Stencil, and Z data with automatic arbitration. A scoreboard system is used to prevent multiple QPUs from accessing the same pixel samples at the same time and also to ensure that overlapping pixel samples are rendered in the correct order.

The QPUs perform color blending in software, so there are facilities to both read and write the tile's color buffer. For efficiency, however, the Z test and stencil operations are carried out in hardware by the tile buffer system, so there is only a write interface for Z data.

In Coverage accumulation mode using the CAP, the tile's Z and stencil memories are reused to store pixel coverage information. There is an additional read interface to the tile buffer for the QPUs to read back this coverage value.

The tile buffer is configurable with either 32-bit color depth, in standard RGBA8888 format, or with 64-bit color depth for HDR rendering, using 16-bit floating-point RGBA colors. The tile buffer can also be configured with or without 4x multisampling. With 32-bit color depth the tile size is 64x64 pixels in non-multisample mode and 32x32 pixels in 4x multisample mode. With 64-bit color depth the tile height is halved to 32 or 16 pixels in non-multisample or multisample mode respectively.

The configuration of color depth and multisample mode remain fixed for a whole frame, and fragment shaders must be appropriate for the selected configuration.

In normal operation, when the rendering of a tile is complete only the tile color buffer is written out to the frame-buffer in main memory. The multisamples are resolved to a single pixel color value per pixel and the color depth is converted to final format with optional dithering. Linear, T-format and LT-format memory formats are supported for the frame-buffer written to memory.

Once each tile has been written out the Tile Buffer (including early-z buffer) is automatically cleared to pre-configured clear values for color, z and stencil, in preparation for rendering the next tile. These clear values remain fixed for the whole frame, and any further clearing during the tile rendering time must be accomplished by filling an appropriate primitive.

For special purposes, the full unresolved tile buffer may be written out to memory and also reloaded from memory. The memory address to write a tile to or read a tile from is specified individually, and the data is written and read in a format internal to the 3D system. Writing and reading of the color, Z or stencil components of the tile buffer may be masked individually.

The tile buffer configuration, and the reading and writing of the tile buffer to and from memory are all controlled from the control list.

QPU Interface

Scoreboard

To ensure that fragment shaders access the tile-buffer in the correct rendering order, fragment shader programs should signal Wait for Scoreboard in an instruction prior to accessing the tile buffer. This will stall the processor as necessary until all preceding accesses to the same pixel samples from other shaders have completed. The program must signal Unlock Scoreboard after all tile buffer accesses are complete to release following shaders. Note that the Lock Scoreboard signal should not be used outside of fragment shaders.

Color Read and Write

To perform color blending, the QPU must read and write every color multisample individually. The tile color is read one multisample at a time, in a fixed order, by an instruction issued with a Color Load signal. The single multisample color value is loaded into the r4 accumulator ready for use in the following instruction. The color is then written back to the tile buffer one multisample at a time, in the same order, by writing to the TLB_COLOUR_MS I/O register.

When the tile buffer is configured for 32-bit color, a single read or write is required per multisample in RGBA8888 format. With 64-bit color two reads or writes are required per multisample, the first containing Red and Green with the second containing Blue and Alpha, in the ls and ms 16-bits respectively. In non-multisample mode, only a single ‘multisample’ should be read or written.

For opaque colors, the QPU can save instructions by writing the same color value to all the multisamples in a pixel in one go, using the TLB_COLOUR_ALL I/O register. In 32-bit color mode just one write to this register is required; in 64-bit color mode two writes are required, in the order RG then BA. In non-multisample mode, writes to TLB_COLOUR_MS have the same effect as writes to TLB_COLOUR_ALL.

For all color reading and writing, the pixel locations are taken from the internal QPU registers X_COORD and Y_COORD, which are set up automatically per thread when a fragment shader is started. These coordinates may be read by the program, but not written, via register-mapped I/O. Similarly, the multisamples written are masked individually according to the QPU internal MS_FLAGS register. The MS_FLAGS register is also setup when the fragment shader is started, but is subsequently modified as a result of the Z and Stencil tests.

Z and Stencil

To perform the Z and Stencil tests, the program simply writes the Z value to the TLB_Z I/O register. This initiates both the stencil-test and the Z-test, resulting in reads and updates of the tile’s stencil buffer, z buffer and early-z buffer. The pixel coordinates and multisample mask are again taken from the QPU’s internal X_COORD, Y_COORD and MS_FLAGS registers, and the MS_FLAGS register is updated as a result of the test.

The stencil and z test both need additional setup data to configure their operations, which all comes from further internal QPU registers. For the Z test mode there is a Z_Test_Mode internal QPU register that is automatically set up when the fragment shader is started. This setup data originates from the control list, specifying the Z-test function (one of 8 modes) plus enable bits for updating the Z and early-Z buffers.

By default the stencil test is disabled when a fragment shader is started, and the shader program must itself set up an internal Stencil_Mode register if the stencil test is to be used. This Stencil_Mode register, which contains separate modes for front and back facing polygons, is written by making one or more writes to the TLB_STENCIL_SETUP I/O register. When the stencil test is enabled the QPU's internal REV_FLAG register is also used to select between the front and back stencil configuration on a per pixel basis.

Coverage Read

The coverage value produced by the CAP in the tile's combined Z and Stencil buffer is read in a similar manner to the color read. An instruction is issued with a Coverage Load signal and the 8-bit converted coverage level is loaded into the r4 accumulator ready for use in the next instruction.

When the coverage buffer is read, the existing contents may either be preserved or cleared. This is selected by an internal Coverage_Read_Mode QPU register, which is set up automatically when the fragment shader is started, according to state data settings from the control list.

Reading of the coverage level is only applicable to non-multisample modes, and will have undefined behavior if attempted in multisample mode.

Tile Buffer Access Restrictions

There are restrictions on performing simultaneous accesses to the tile buffer from the same QPU in the same program instruction, in order to simplify the arbitration of accesses from multiple QPUs. A program must not make more than one tile buffer access per instruction, with the exception that a multisample read and multisample write can occur in the same instruction. If other multiple accesses are attempted in the same instruction the behavior is undefined.

The Z and Stencil tests modify the MS_FLAGS for subsequent instructions. It is therefore necessary to perform the Z write before writing to the color buffer. It is safe to immediately follow an instruction that writes to the TLB_Z with an instruction that writes to the color buffer.

QPU Registers for Tile Buffer Access

The following QPU signaling codes are used for Tile Buffer access:

Table 25: Tile Buffer Access Signaling Codes

Value	*Meaning*
4	Wait for Scoreboard (stall until this QPU can safely access tile buffer)
5	Scoreboard Unlock
7	Coverage load from Tile Buffer to r4
8	Color Load from Tile Buffer to r4
9	Color Load and Program End

The following QPU register addresses are used for Tile Buffer access:

Table 26: QPU Register Addresses for Tile Buffer Accesses

Addr	*A rd*	*B rd*	*A wr*	*B wr*
41	X_PIXEL_COORD	Y_PIXEL_COORD	QUAD_X	QUAD_Y
42	MS_FLAGS	REV_FLAG	MS_FLAGS	REV_FLAG
43			TLB_STENCIL_SETUP	TLB_STENCIL_SETUP
44			TLB_Z	TLB_Z
45			TLB_COLOUR_MS	TLB_COLOUR_MS
46			TLB_COLOUR_ALL	TLB_COLOUR_ALL

TLB_STENCIL_SETUP is used for setting up the front and back stencil functions in the internal Stencil_Mode register. The stencil front and back functions are both disabled by default when a fragment shader is started, so no writes are required if stencilling is not used. Up to three separate writes are required to fully set up the Stencil Mode, but the most common settings can be achieved with a single write. The type of data is identified by the MSBs of the 32-bits written, and is one of the following:

Table 27: Front or Back Stencil Configuration

Register Name(s)		*TLB_STENCIL_SETUP*
Sub-function		Front or Back Stencil Configuration
Bits	*Name*	*Description*
31:30	WSEL	1 = Write data in bits 29:0 to front stencil configuration 2 = Write data in bits 29:0 to back stencil configuration 3 = Write data in bits 29:0 to front and back stencil configurations
29:28	SFWMC	Stencil write mask code: 0 => write mask = 0x1 1 => write mask = 0x3 2 => write mask = 0xf 3 => write mask = 0xff

Table 27: Front or Back Stencil Configuration (Cont.)

Register Name(s)		***TLB_STENCIL_SETUP***
27:25	SFZFOP	Z-test fail op: (as SFSOP)
24:22	SFZPOP	Z-test pass op: (as SFSOP)
21:19	SFSFOP	Stencil-test fail op: (0-7 = zero, keep, replace, incsat, decsat, invert, inc, dec)
18:16	SFUNC	Stencil-test function (0-7 = never, lt, eq, le, gt, ne, ge, always)
15:8	SFVALUE	Stencil reference value
7:0	SFMASK	Stencil function mask

Table 28: Front and Back Stencil Write Masks Configuration

Register Name(s)		***TLB_STENCIL_SETUP***
Sub-function		Front and Back Stencil Write Masks
Bits	***Name***	***Description***
31:30	ID	0 => Write of Front and Back Stencil Write Masks (in full)
29:16	–	Unused
15:8	SFBWM	Back stencil (full) write mask
7:0	SFFWM	Front stencil (full) write mask

Section 6: FEP-to-QPU Interface

Initial Data

Fragment shaders are started automatically each time the FEP accumulates a vector of up to four quads (16 pixels) to shade together. The quad input data from the FEP is automatically written into per-thread QPU registers when the fragment shader is started. The following data is written to these QPU registers, in addition to the normal PC address, uniforms base address, and uniforms size:

Table 29: FEP Quad Input Data

Regfile Address	*Register name*	*Description*	*Bits*	*Program Readable?*
41-A	X_PIXEL_COORD	X screen coordinate	11	Y
41-B	Y_PIXEL_COORD	Y screen coordinate	11	Y
42-A	MS_MASK	Multisample mask	4	Y
42-B	REV_FLAG	Set for back facing primitive	1	Y
15-A	W	Floating point W	32	Y
15-B	Z	Fixed-point Z	24	Y
–	Z_Test_Mode	Z-test mode, plus z and early-z write enables. From control list state.	5	N
–	Stencil_Mode	Just stencil front and back enables set to 0 at start. Remaining 72 bits set by program if used.	2+	N
–	Coverage_Read_Mode	From control list state.	1	N
–	Varyings setup	Number of varyings, x0/y0 offset and location of coefficients in coefficient memory.	–	N

Varyings Interpolation

The varyings are interpolated using an equation of the form (A*(x-x0)+B*(y-y0))*W+C. The partial varying result V_P=(A*(x-x0)+B*(y-y0)) is calculated in hardware, with V_P*W+C calculated in QPU code. The V_P results are pre-calculated and placed in a small per-thread FIFO, together with the C coefficient. The V_P results are read via the VARYING_READ I/O register. In the same instruction that the FIFO is read the C coefficient is automatically loaded into the r5 accumulator ready for use in the following instruction.

The QPU can pipeline the calculations to achieve a throughput of one varying per cycle, and the shared interpolation hardware plus FIFO is sufficient to sustain this rate for two QPUs in parallel within the slice. Reads of the VARYING_READ register will stall while empty, unless all of the varyings have been read for that thread in which case the read returns undefined data.

The interpolation of varyings is started as soon as possible after a fragment shader is allocated to a QPU, so that the first interpolated results are usually ready by the first instruction of the program. When a shader is started back-to-back with the preceding program, the interpolation can start as early as the Program End instruction of the previous program. For this reason, a fragment shader must finish reading varyings before issuing the Program End instruction. All of the set up varyings must be read before the shader completes.

The FIFO used for the partially interpolated varying results is also used for VPM read and write accesses and VCD control from QPU programs. For this reason a fragment shader program cannot access the VPM or VCD. However, programs that access the VPM or VCD do not need to complete such accesses before the Program End instruction. In this case the outstanding VPM or VCD functions will complete before any varyings interpolation can start on a subsequent fragment shader.

The following QPU registers are used to read varyings.

Table 30: QPU Register Addresses for Reading Varyings

Addr	*A rd*	*B rd*	*A wr*	*B wr*
35	VARYING_READ	VARYING_READ	–	–

Section 7: VPM and VCD

The VPM is a memory buffer intended for loading vectors of unshaded vertex attributes into the QPUs for vertex shading and storing vectors of shaded vertices back out again. For these purposes the VCD automatically loads unshaded vertex attributes into the VPM from main memory using DMA, and the shaded vertices are read directly from the VPM by the PSE or PTB.

Although the VCD and VPM are specifically organized to support reading and writing of vectors of vertices, the VPM and VCD plus VDW facilities are sufficient for general-purpose use. The VCD can load blocks of scattered vectors or 2D byte arrays in memory to the VPM with both horizontal and vertical orientation. The separate VDW block does the reverse, storing vertical or horizontal VPM data out to 2D arrays of data in memory. The QPU can then read and write blocks of 8, 16 and 32-bit vectors from/to the VPM, also with horizontal or vertical orientation.

For vertex shading purposes a portion of the VPM is automatically allocated to the QPU for the batch of vertices to be processed, and the addresses of read and write accesses are automatically mapped to the correct locations in the VPM. General-purposes accesses are mapped to the start of the VPM memory and a portion of the VPM must be reserved for general-purpose use by writing the V3DVPMRSV register.

The minimum VPM size is 8Kbytes, which is the amount required for normal pipelined 3D operation with concurrent vertex and coordinate shading and worst case vertex data size. With this size of memory it is not sensible to divide the VPM between 3D shading functions and general-purpose processing at the same time. Fully configured systems may have up to 16Kbytes of VPM for higher vertex shading performance. With this size of VPM it is practical for some of the memory to be reserved for general-purpose processing whilst 3D is operating so long as at least 8Kbytes is left for 3D use.

Note that the VPM cannot be accessed in Fragment shaders, because the FIFOs in the interface hardware are shared with the Varyings interpolation system.

QPU Reading and Writing of VPM

From the QPU perspective the window into the locally allocated portion of the VPM is a 2D array of 32-bit words, 16 words wide with a maximum height of 64 words. The array is read and written as horizontal or vertical 16-way vectors of 32, 16 or 8-bit data, with natural alignment. Thus horizontal 32-bit vectors start in column 0 and vertical 32-bit vectors must start on a row multiple of 16.

To access the VPM as 16-bit or 8-bit vectors, each 32-bit vector is simply split into 2x 16-bit or 4x 8-bit sub-vectors. There are two alternative split modes supported for sub-vectors: ‘laned’, where each 32-bit word is split into two 16-bit lanes or four 8-bit lanes; or ‘packed’, where the 16-bit or 8-bit sub-vector is taken from the whole of eight or four successive 32-bit words. Some examples are shown in the Figure 8 and Figure 9.

Figure 8: VPM Horizontal Access Mode Examples

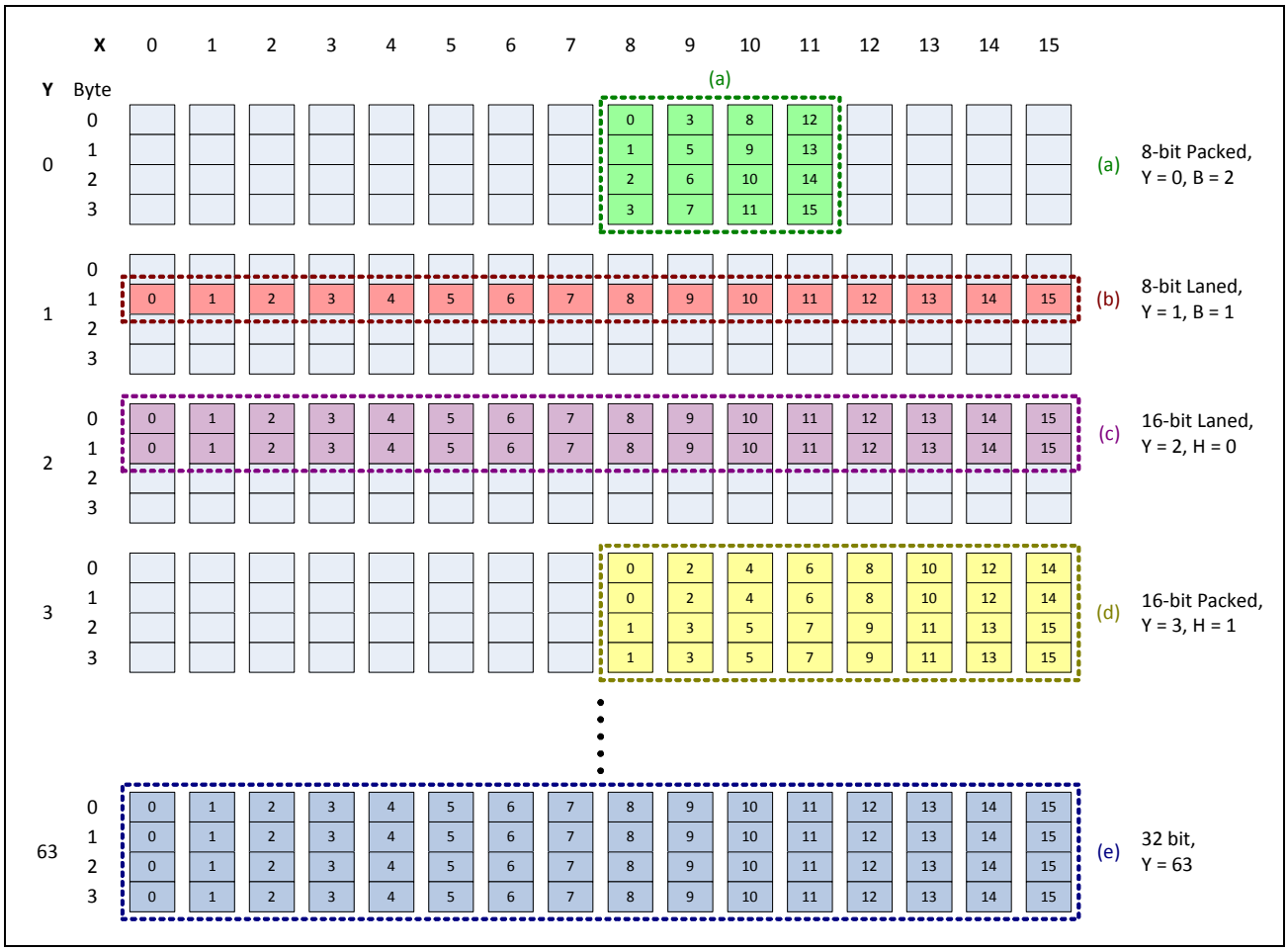
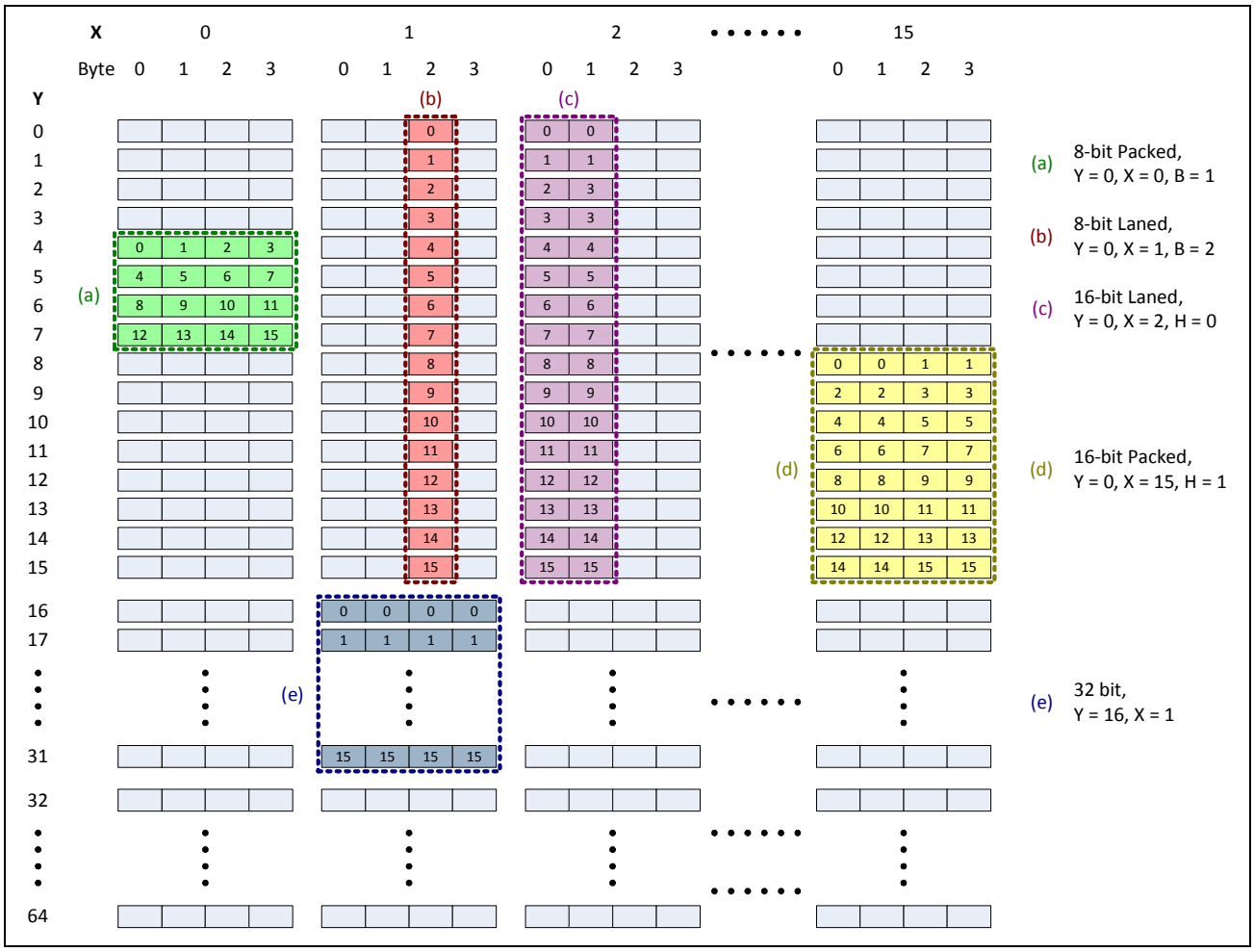


Figure 9: VPM Vertical Access Mode Examples



The QPU reads or writes programmable blocks of data from/to the VPM one vector at a time, by repeatedly reading or writing the VPM_READ or VPM_WRITE register. The setup of type, orientation, location, stride and number of vectors in the block is programmed by a single write to the VPMVCD_RD_SETUP or VPMVCD_WR_SETUP register.

The 32-bits of setup data are provided by element 0 of the vector written to the setup register. Note that a common register location is used for all VPM and VCD setup data, with the type of setup data identified by the MSBs of the 32-bit value written.

When writing vector data to the VPM, the contents of the write setup register are used for an indefinite number of subsequent writes. The write address will wrap when incremented beyond a Y of 63, and writes to addresses outside of the window of allocated VPM space will be masked. Up to two writes are queued in a FIFO, and writes will stall the QPU when the FIFO is full.

Reads of the VPM are setup to read a specific number of vectors, and exactly this number of vectors should be read by the QPU program. After the read setup register is written, read data is available to read after a minimum latency of three QPU instructions. Reads made after this time will stall the QPU until data arrives, but reads made too early or extra reads made beyond the number setup will return immediately with undefined data.

To accommodate the latency from read setup to read data being ready, two read setups can queued at a time. The QPU program must take care to have no more than two read blocks queued at a time, as writes to the setup register will be ignored if the queue is full.

As with writes, the read address will wrap beyond a Y of 63, but the data will always be returned even if the address falls outside of the window of allocated VPM space. When a QPU program finishes, all outstanding VPM reads for that QPU are cancelled. There is no mechanism within QPU program to cancel reads, however.

QPU Control of VCD and VDW

For Vertex and Coordinate shading, the VCD is programmed automatically by the 3D pipeline to fetch vertex attribute data into the VPM in a pre-defined format. For general-purpose use a QPU may itself program the VCD to DMA load data into the VPM or program the VDW to DMA store data from the VPM. The DMA facilities are quite flexible, allowing the loading and storing of 2D structures in memory into vertical or horizontal structures within the VPM. The VCD load and VDW store facilities are not symmetrical.

DMA load and store operations are set up by writes to either the VPMVCD_RD_SETUP register or the VPMVCD_WR_SETUP register respectively. After the DMA is set up, the actual DMA load or store operation is initiated by writing the memory address to the VCD_LD_ADDR or VCD_ST_ADDR register. The address is taken from vector element 0 (multiple block loads/stores using a vector of addresses are not supported currently, but may be in future revisions).

To determine when the DMA operation is complete the QPU can either poll the VCD_LD_BUSY or VCD_ST_BUSY register for a zero value, or simply read from the VCD_LD_WAIT or VCD_ST_WAIT register, which stalls until the respective DMA is complete. A new DMA load or store operation cannot be started until the previous one is complete, but load and store DMA can run concurrently.

There are separate setup register formats for DMA loads and stores. For each of these is a basic DMA setup register and an extra stride setup register. For DMA loads, the extra stride value is only used when selected in the basic DMA load setup register.

Separate base addresses for VCD loads and stores may be set in addition to the VPM read and write base addresses. These are all set via writes to the VPMVCD_WR_SETUP register.

QPU Registers for VPM and VCD Functions

The following QPU register addresses are used for VPM read/write and VCD/VDW load/store operations:

Table 31: QPU Register Addresses for VPM Read/write and VCD/VDW Load/store

Addr	*A rd*	*B rd*	*A wr*	*B wr*
48	VPM_READ	VPM_READ	VPM_WRITE	VPM_WRITE
49	VPM_LD_BUSY	VPM_ST_BUSY	VPMVCD_RD_SETUP	VPMVCD_WR_SETUP
50	VPM_LD_WAIT	VPM_ST_WAIT	VPM_LD_ADDR	VPM_ST_ADDR

VPMVCD_RD_SETUP and VPMVCD_WR_SETUP are multi-purpose registers for setting up VPM block reads and writes or VCD/VDW DMA load and store operations. The value written to this register is always taken from the 32-bit element 0 of the written vector, and the particular type of setup data written is identified from the MSBs of the value written. The following formats of setup data are defined:

Table 32: VPM Generic Block Write Setup Format

Register Name(s)		*VPMVCD_WR_SETUP*
Sub-function		VPM generic block write setup
Bits	***Name***	***Description***
31:30	ID	= 0. Selects VPM generic block write setup register.
29:18	–	Unused
17:12	STRIDE	Stride. This is added to ADDR after every vector written. 0 => 64.
11	HORIZ	0,1 = Vertical, Horizontal
10	LANED	0,1 = Packed, Laned. Ignored for 32-bit width
9:8	SIZE	0,1,2,3 = 8-bit, 16-bit, 32-bit, reserved
7:0	ADDR	Location of the first vector accessed. The LS 1 or 2 bits select the Half-word or Byte sub-vector for 16 or 8-bit width. The LS 4 bits of the 32-bit vector address are Y address if horizontal or X address if vertical. Thus: Horizontal 8-bit: ADDR[7:0] = {Y[5:0], B[1:0]} Horizontal 16-bit: ADDR[6:0] = {Y[5:0], H[0]} Horizontal 32-bit: ADDR[5:0] = Y[5:0] Vertical 8-bit: ADDR[7:0] = {Y[5:4], X[3:0], B[1:0]} Vertical 16-bit: ADDR[6:0] = {Y[5:4], X[3:0], H[0]} Vertical 32-bit: ADDR[5:0] = {Y[5:4], X[3:0]}

Table 33: VPM Generic Block Read Setup Format

Register Name(s)		***VPMVCD_RD_SETUP***
Sub-function		VPM generic block read setup
Bits	***Name***	***Description***
31:30	ID	= 0. Selects VPM generic block read setup.
29:24	–	Unused
23:20	NUM	Number of vectors to read (0 => 16).
19:18	–	Unused
17:12	STRIDE	Stride. This is added to ADDR after every vector read. 0 => 64.
11	HORIZ	0,1 = Vertical, Horizontal
10	LANED	0,1 = Packed, Laned. Ignored for 32-bit width
9:8	SIZE	0,1,2,3 = 8-bit, 16-bit, 32-bit, reserved
7:0	ADDR	Location of the first vector accessed. The LS 1 or 2 bits select the Half-word or Byte sub-vector for 16 or 8-bit width. The LS 4 bits of the 32-bit vector address are Y address if horizontal or X address if vertical. Thus: Horizontal 8-bit: ADDR[7:0] = {Y[5:0], B[1:0]} Horizontal 16-bit: ADDR[6:0] = {Y[5:0], H[0]} Horizontal 32-bit: ADDR[5:0] = Y[5:0] Vertical 8-bit: ADDR[7:0] = {Y[5:4], X[3:0], B[1:0]} Vertical 16-bit: ADDR[6:0] = {Y[5:4], X[3:0], H[0]} Vertical 32-bit: ADDR[5:0] = {Y[5:4], X[3:0]}

Table 34: VCD DMA Store (VDW) Basic Setup Format

Register Name(s)		***VPMVCD_WR_SETUP***
Sub-function		VPM DMA Store (VDW) basic setup
Bits	***Name***	***Description***
31:30	ID	= 2. Selects VDW DMA basic setup
29:23	UNITS	Number of Rows of 2D block in memory (0 => 128)
22:16	DEPTH	Row Length of 2D block in memory (0 => 128)
15	LANED	Write as 0
14	HORIZ	0,1 = Vertical, Horizontal
13:3	VPMBASE	X,Y address of first 32-bit word in VPM to load to/store from. ADDRA[10:0] = {Y[6:0], X[3:0]}
2:0	MODEW	Mode, combining width with start Byte/Half-word offset for 8 and 16-bit widths. 0: width = 32-bit 1: Unused. 2-3: width = 16-bit, Half-word offset (packed only) = MODEW[0] 4-7: width = 8-bit, Byte offset (packed only) = MODEW[1:0]

Table 35: VCD DMA Write (VDW) Stride Setup Format

Register Name(s)		***VPMVCD_WR_SETUP***
Sub-function		VPM DMA Store stride setup
Bits	***Name***	***Description***
31:30	ID	= 3. Selects VDW DMA stride setup
29:17	–	Unused
16	BLOCKMODE	0 = 'row-row' pitch in VPM is 1-row /1-column for horizontal/vertical mode. 1 = rows are packed consecutively in VPM (into rows or columns)
12:0	STRIDE	Distance between last byte of a row and start of next row in memory, in bytes.

Table 36: VCD DMA Load (VDR) Basic Setup Format

Register Name(s)		***VPMVCD_RD_SETUP***
Sub-function		VPM DMA Load (VDR) basic setup
Bits	***Name***	***Description***
31	ID	= 1. Selects VDR DMA basic setup (in addition, bits[30:28] != 1)
30:28	MODEW	Mode, combining width with start Byte/Half-word sel for 8 and 16-bit widths. 0: width = 32-bit 1: selects VPM DMA extended memory stride setup format, defined separately. 2-3: width = 16-bit, Half-word sel (packed only) = MODEW[0] 4-7: width = 8-bit, Byte sel (packed only) = MODEW[1:0]
27:24	MPITCH	Row-to-row pitch of 2D block in memory. If MPITCH is 0, selects MPITCHB from the extended pitch setup register. Otherwise, pitch = 8*2^MPITCH bytes.
23:20	ROWLEN	Row length of 2D block in memory. In units of width (8, 16 or 32 bits). (0 => 16)
19:16	NROWS	Number of rows in 2D block in memory. (0 => 16)
15:12	VPITCH	Row-to-row pitch of 2D block when loaded into VPM memory. (0 => 16). Added to the Y address and Byte/Half-word sel after each row is loaded, for both horizontal and vertical modes. For 8-bit width, VPITCH is added to {Y[1:0], B[1:0]}. For 16-bit width, VPITCH is added to {Y[2:0], H[0]}. For 32-bit width, VPITCH is added to Y[3:0].
11	VERT	0,1 = Horizontal, Vertical
10:0	ADDRXY	X,Y address of first 32-bit word in VPM to load to /store from. ADDRA[10:0] = {Y[5:0], X[3:0]}

Table 37: VCD DMA Load (VDR) Extended Memory Stride Setup Format

Register Name(s)		*VPMVCD_RD_SETUP*
Sub-function		VPM DMA Load (VDR) extended memory stride setup
31:28	ID	= 9. Selects VDR DMA extended memory stride setup
27:13	–	Unused
12:0	MPITCHB	Row-to-row pitch of 2D block in memory, in bytes. Only used if MPITCH in VPM DMA Load basic setup is 0.

VPM Vertex Data Formats

The vertex attribute data is loaded into VPM memory by the VCD in a fixed format, and the shaded vertex data read by the PTB and PSE is read assuming a fixed format. These formats are described in the following sections. Note that variations on the shaded vertex formats may be employed for vertices in memory.

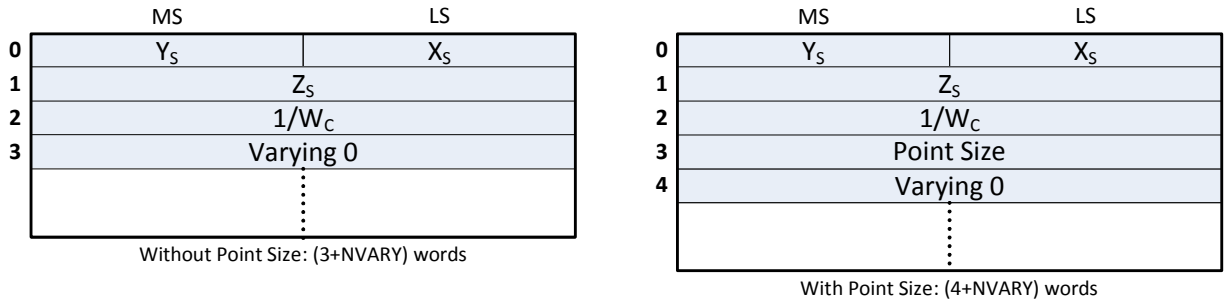
Vertex Attribute Format in VPM from VCD

The vertex attribute data for each of 16 vertices is loaded into a single column for each vertex, such that the QPU can read a horizontal vector of vertices for each individual attribute. The attributes for each vertex are packed into the column according to the setup data supplied to the VCD from the appropriate Shader State Record specified by the control list.

Shaded Vertex Format in VPM for PSE

The PSE expects the shaded data for each vertex to be stored in a single column in the VPM, as 32-bit words according to the following format.

Figure 10: Shaded Vertex Format for PSE

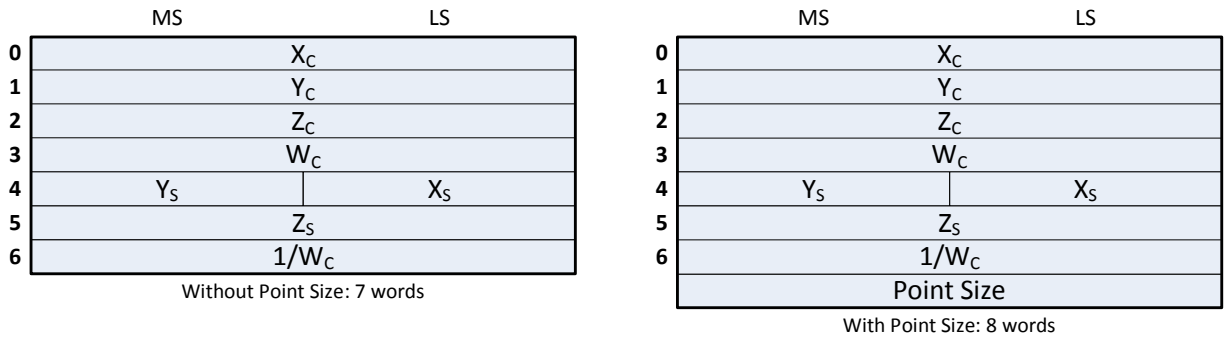


- Z_S, $1/W_C$, Point Size, and Varyings are 32-bit float.
- X_S and Y_S are signed 12.4 fixed point, in pixels, relative to the viewport centre.

Shaded Coordinates Format in VPM for PTB

The PTB expects the shaded data for each coordinate-only vertex to be stored in a single column in the VPM, as 32-bit words according to the following format.

Figure 11: Shaded Coordinates Format for PTB



- X_C, Y_C, Z_C, W_C, Z_S, $1/W_C$, and Point Size are 32-bit float.
- X_S and Y_S are signed 12.4 fixed point, in pixels, relative to the viewport centre.

Section 8: System Control

The 3D system control is highly automated in hardware, being driven by control lists in memory. A hardwired Control List Executer (CLE) reads the control lists and controls and feeds the 3D pipeline. The control lists define virtually all the operations of the 3D system, to the extent that hardly any interaction is required from the host processor after the CLE has started executing a control list. All that the host processor usually needs to do is to write the control list, wait for the rendering of frames to complete and reclaim the memory used.

System Operation

All rendering by the 3D system is in tiles, requiring separate binning and rendering passes to render a frame. In normal operation the host processor creates a control list in memory defining all the operations and supplying all the data for rendering for a complete frame.

Two control lists are required, one for the binning pass and one for the rendering pass. The binning list sets up the tile binning mode configuration, supplying binning memory for the PTB to work with, and then specifies all the state data, shaders, and primitive lists to complete the frame.

During the binning pass the PTB automatically writes out a new control list for rendering each tile during the rendering pass. The binning list must finish with a 'Flush' command to cause the PTB to finalize all these tile lists. All that the host processor then needs to do for the rendering pass list is to set up the tile rendering mode configuration and link together all the tile lists created by the PTB as sub-lists to the main list. The only control items that the host processor needs to add per tile list is a 'tile coords' item before and a 'store tile' item after each tile list, finishing with a 'store tile plus end of frame' item after the last tile list. The host processor has no need to wait for the binning pass to finish before completing the rendering mode list - indeed the complete binning and rendering list can be written before the binning pass has even started.

The CLE has separate threads for executing tile binning and tile rendering lists, allowing the tile rendering for one frame to overlap the tile binning for the following frame. Thread 0 is dedicated to tile binning and thread 1 is dedicated to tile rendering. The two threads can be synchronised via counting semaphores, so that tile rendering can follow on automatically after tile binning. The semaphores may also be used to stop the tile binning getting more than one frame ahead of the tile rendering.

The control list is initially supplied to the CLE for thread *n* via list start and list end addresses in the V3DCT*n*CA and V3DCT*n*EA registers, respectively. The CLE advances V3DCTnCA until it exactly matches V3DCT*n*EA. The host can extend the list at any time by updating V3DCT*n*EA, which will cause processing to resume if the control thread has already reached the end of the list.

The host processor can keep track of progress in complex control lists that contain branches and sub-lists by placing marker entries in the list. The marker count in the V3DCT*n*CS register is incremented when a marker is encountered. The host processor can detect the completion of complete frames by monitoring the V3DBFC and V3DRFC registers, which are incremented each time a frame is completed in binning and rendering mode, respectively.

A number of system events can be programmed to generate host interrupts, to avoid the need for the host processor to poll the 3D system status. Typically the only events of interest are 'end of frame' for tile rendering mode and 'out of tile list memory' or 'overspill memory consumed' in tile binning mode.

System Pipelines and Modes

The 3D system operates conceptually separate pipelines for tile-binning and tile-rendering. Each pipeline operates quite autonomously, driven by the data flow of primitives. State change transitions are passed down the pipeline with the primitives, so there is very rarely any need to drain out the pipeline to change system parameters.

Each pipeline can operate in one of three major modes:

- GL mode, in which vertex shading is employed
- NV mode, without vertex shading, using pre-shaded vertices stored in memory
- VG mode, where vertices are supplied directly from the input primitive list as XY coordinates only.

In GL mode, the pipeline up to the PTB/PSE consists of the following steps:

1. Determine a batch of vertices to shade in the VCM.
2. Find space in the VPM to store the batch of vertex input attributes and shaded vertices.
3. Fetch vertex attributes to the VPM using the VCD.
4. Shade the vertices using a vertex/coordinate shader in a QPU
5. PTB/PSE reads shaded vertex data from the VPM.

This pipeline is driven by the availability of space in the VPM to fetch more vertices, which is governed by the time to fetch vertex attributes, the rate of vertex shading and/or the rate at which the PTB/PSE consumes data.

The NV mode pipeline is similar to GL mode, but Step 3 loads pre-shaded vertex data from memory to the VPM and Step 4 is omitted. As there is no vertex shading, the pipeline is governed by the vertex fetch rate and the PTB/PSE consumption.

In VG mode, Step 1 to Step 4 are completely bypassed and vertices are fed directly to the PTB or PSE. In this mode, the pipeline is purely driven by the speed of the PTB/PSE.

The GL/NV/VG mode is selected by a 'GL/NV/VG Shader State' entries in the control list. The mode may be changed during a frame, with the changeover handled by the hardware such that VG mode primitive data doesn't overtake data in the vertex fetch/vertex shading pipeline.

Up to the PTB and PSE stage, the hardware pipeline resources of the VCD and VPM are shared by the tile-binning and tile-rendering pipelines by multiplexing at the vertex batch level. A fair arbitration scheme is used when both pipelines are demanding data, in particular to avoid the faster pipeline hogging all of the VPM buffering resource. The dynamics of the system may be balanced to some extent by reserving or restricting the use of QPUs for running vertex and coordinate shaders using the V3DQRSV*x* registers.

The rendering mode pipeline after the FEP has two further modes of operation, namely to use the Coverage Accumulation Pipe (CAP) or use fragment shaders. This mode is selected according to the 'Coverage Pipe Select' bit in a 'Configuration Bits' state entry in the control list, and may be changed at will during the frame. The hardware handles the mode changeover, via the tile-buffer scoreboard system, to ensure that the correct pixel rendering order is maintain.

In tile rendering mode, the change from one tile to the next is handled in a pipelined fashion in the hardware, to avoid draining the pipeline. The control list is responsible for ensuring that the state is correctly restored at the start of the rendering of each tile list whilst in tile rendering mode. Specific care is required in this, as the PTB only adds state change data to a list when the state is relevant to primitives actually entered in the list.

The expected way to correctly restore the tile start state is to set the state at the end of tile binning list to reflect the state required at the start of all tile lists. If this state is different to that at the end of the previous frame, the state can be initialized by including the appropriate prefix state items at the start of the binning list before the (obligatory) 'start binning' command. The tile binning list should then finish with a 'Flush All State' command to force the PTB to append any pending state changes to each tile list. Finally, the tile rendering mode list should start with the same prefix of state changes that were included a the start of the binning list, to ensure that the initial state for the first tile is correct.

Section 9: Control Lists

A control list is a sequence of variable length control data records. Each control record starts with single byte id code followed by an implicit number of bytes of data. Control records may include a mix of immediate data and pointers to further data in memory. Each control record specifies either a primitive index list, a piece of state data, a system control action or a branch or link to another control list.

Input primitive lists are specified indirectly by address, with all OpenGL-ES indexed and array primitive list types supported. A special VG primitive list format is also supported, consisting of immediate XY coordinate data, either as an indirect list or as an in-line escape-terminated list. There is also a compressed format for primitives embedded in-line within tile lists, which is the format produced by the PTB. This format has escapes to terminate the list, and branches to allow lists to be efficiently chained from several memory blocks.

State data records in tile lists either directly contain state data, or contain the address of a state data structure stored elsewhere in memory. Top level state data records provide configuration data for the whole frame in tile binning and tile rendering modes. The major dynamic system state data is contained within combined vertex and fragment shader state records, which are specified indirectly. The simplified VG mode shader state, which only specifies fragment shaders, can be specified in-line. The remaining minor dynamic state data records in control lists are hardware register settings. These may be changed independently of the shaders and can be specified individually.

The tables in the following sections describe all supported control records and indirect data structures.

Control Record IDs and Data Summary

Certain control list items are only allowed in binning or rendering mode lists – these are identified by a (B) or (R) suffix. There are restrictions on the use of certain items in lists, noting that some of these rules are applied automatically by the binner for the rendering mode tile lists.

- Binning mode lists start with a Tile Binning Mode Configuration item (112).
- Binning mode lists must have a Start Tile Binning item (6) after any prefix state data before the binning list proper starts.
- Rendering mode lists start with a Tile Rendering Mode Configuration item (113), optionally preceded by a Clear colors item (114) if these are to be changed.
- In rendering mode, each tile list must have one Tile Coordinates item (115) at the start and one Store Tile Buffer item (24, 25, 26, or 28) at the end.
- In rendering mode, if a Load Tile Buffer item (27 or 29) is used, it must occur before the Tile Coordinates item (115) of the tile list of interest.
- A Primitive List Format item (56) must be followed by a Shader State item (64, 65, 66, or 67) to be recognized.

Table 38: Control Record IDs and Data Summary

Control Codes			
	Synopsis (B = binning only, R = rendering only)		
Code	***Bits***	***Offset***	***Field Description***
0	Halt		
1	NOP		
2–3	Reserved		
4	Flush (Add Return-from-sub-list to tile lists and then flush tile lists to memory) (B)		
5	Flush All State (Same as flush, but preceded by the forced writing of the current state to the tile lists) (B)		
6	Start Tile Binning (advances state counter so that initial state items actually go into tile lists) (B)		
7	Increment Semaphore (after tile lists are flushed or last tile written)		
8	Wait on Semaphore (to wait for frame complete in other thread)		
9–15	Reserved		
16	Branch		
	32		Absolute branch address
17	Branch to Sub-list (maximum of 2 levels of nesting)		
	32		Absolute branch address
18	Return from sub-list (ignored if nothing on the return stack)		
19–23	Reserved		
24	Store Multi-sample Resolved Tile Color Buffer (R)		
25	Store Multi-sample Resolved Tile Color Buffer and signal end of frame (R)		
26	Store Full Resolution Tile Buffer (R)		
	28	4	Memory address of Tile. In multiples of 16 bytes.
	1	3	Last Tile of Frame
	1	2	Disable Clear on Write
	1	1	Disable Z/Stencil Buffer write
	1	0	Disable Color Buffer write
27	Re-load Full Resolution Tile Buffer (R)		
	28	4	Memory address of Tile. In multiples of 16 bytes.
	2	2	Unused
	1	1	Disable Z/Stencil Buffer read
	1	0	Disable Color Buffer read

Table 38: Control Record IDs and Data Summary (Cont.)

Control Codes			
	Synopsis (B = binning only, R = rendering only)		
Code	***Bits***	***Offset***	***Field Description***
28	Store Tile Buffer General (R)		
	28	20	Memory base address of frame/tile dump buffer. In multiples of 16 bytes.
	1	19	Last Tile of Frame
	1	18	Disable VG-Mask buffer dump (applies full dump mode only)
	1	17	Disable Z/Stencil buffer dump (applies full dump mode only)
	1	16	Disable Color buffer dump (applies full dump mode only)
	1	15	Disable VG-Mask buffer clear on store/dump
	1	14	Disable Z/Stencil buffer clear on store/dump
	1	13	Disable Color buffer clear on store/dump
	1	12	Disable double-buffer swap in double buffer mode
	2	10	Unused
	2	8	Pixel Color format (0,1,2 = rgba8888, bgr565 dithered, bgr565 no dither). Applies to non-HDR Color store only.
	2	6	Mode: (0,1,2 = Sample 0, Decimate x4, Decimate x16). Applies to non-HDR Color store only. Decimate x16 only available in ms mode.
	2	4	Format (0,1,2 = raster format, T-format, LT-format)
	1	3	Unused
	3	0	Buffer to Store (0,1,2,3,4,5 = None, Color, Z/stencil, Z-only, VG-Mask, Full Dump)
29	Load Tile Buffer General (R)		
	28	20	Memory base address of frame/tile dump buffer. In multiples of 16 bytes.
	1	19	*Unused*
	1	18	Disable VG-Mask buffer load (applies full reload mode only)
	1	17	Disable Z/Stencil buffer load (applies full load mode only)
	1	16	Disable Color buffer load (applies full reload mode only)
	6	10	Unused
	2	8	Pixel Color format (0,1,2 = rgba8888, bgr565 dithered, bgr565 no dither). Applies to non-HDR Color load only.
	2	6	*Unused*
	2	4	Format (0,1,2 = raster format, T-format, LT-format)
	1	3	*Unused*
	3	0	Buffer to Load (0,1,2,3,4,5 = None, Color, Z/stencil, N/A, VG-Mask, Full Reload)
30–31	Reserved		

Table 38: Control Record IDs and Data Summary (Cont.)

Control Codes			
	Synopsis (B = binning only, R = rendering only)		
Code	***Bits***	***Offset***	***Field Description***
Primitive Lists			
32	Indexed Primitive List (OpenGL)		
	32	72	Maximum Index (primitives using a greater index will cause error)
	32	40	Address of Indices List
	32	8	Length (number of Indices)
	4	4	Index type
			0,1 = 8-bit, 16-bit
	4	0	Primitive mode
			0,1,2,3,4,5,6 = points, lines, line_loop, line_strip, triangles, triangle_strip, triangle_fan
33	Vertex Array Primitives (OpenGL)		
	32	40	Index of First Vertex
	32	8	Length (number of Vertices)
	8	0	Primitive mode
			0,1,2,3,4,5,6 = points, lines, line_loop, line_strip, triangles, triangle_strip, triangle_fan
34–40	Reserved		
41	VG Coordinate Array Primitives (only for use in VG shader mode)		
	32	40	Address of Coordinate Array (32-bit x,y screen coordinates)
	32	8	Length (number of primitives)
	4	4	Continuation List (for triangle fans only)
	4	0	Primitive Type
			1,3,4,5,6 = RHTs, RHT_strip, triangles, triangle_strip, triangle_fan
42	VG Inline Primitives (only for use in VG shader mode)		
	>=32	8	Escape terminated uncompressed 32-bit x,y coordinate list
	4	4	Continuation List (for triangle fans only)
	4	0	Primitive Type
			1,3,4,5,6 = RHTs, RHT_strip, triangles, triangle_strip, triangle_fan
43–47	Reserved		
48	Compressed Primitive List (R)		
	>=8		Escape terminated list
49	Clipped Primitive with Compressed Primitive List (R)		
	>=8		Escape terminated list
	29	3	Address of Single Clipped Primitive Data (multiple of 8 bytes)
	3	0	1 flag per vertex of next primitive, to indicate if this vertex is to be clipped
50–55	Reserved		

Table 38: Control Record IDs and Data Summary (Cont.)

Control Codes			
	Synopsis (B = binning only, R = rendering only)		
Code	***Bits***	***Offset***	***Field Description***
56	Primitive List Format (R)		
	4	4	Data Type 1,3 = 16-bit index, 32-bit x/y
	4	0	Primitive Type 0,1,2,3 = Points, Lines, Triangles, RHT
57–63	Reserved		
State Data			
64	GL Shader State		
	28	0	Memory Address of Shader Record (in multiples of 16 bytes)
	1	3	Extended shader record with 26-bit attribute memory stride)
	3	0	Number of attribute arrays (0 => all 8 arrays). Ignored for extended shader record.
65	NV Shader State (no vertex shading)		
	32	0	Memory Address of Shader Record (16-byte aligned)
66	VG Shader State		
	32	0	Memory Address of Shader Record (16-byte aligned)
67	VG Inline Shader Record		
	32	32	Fragment Shader Uniforms Address (4-byte aligned)
	29	3	Fragment Shader Code Address (8-byte multiple)
	3	0	Dual or Single threaded fragment shader 0,1 = Dual threaded, Single threaded.
68–95	Reserved		
96	Configuration Bits		
	6	18	*Unused*
	1	17	Early Z updates enable
	1	16	Early Z enable
	1	15	Z updates enable
	3	12	Depth-Test Function (0-7 = never, lt, eq, le, gt, ne, ge, always)
	1	11	Coverage Read Mode (0,1 = Clear on read, Leave on read)
	2	9	Coverage Update Mode (0-3 = nonzero, odd, or, zero)
	1	8	Coverage Pipe Select
	2	6	Rasteriser Oversample Mode (0,1,2,3 = none, 4x, 16x, Reserved)
	1	5	Coverage Read Type (0 = 4*8-bit level, 1 = 16-bit mask)
	1	4	Antialiased Points and Lines (not actually supported)
	1	3	Enable Depth Offset
	1	2	Clockwise Primitives
	1	1	Enable Reverse Facing Primitive
	1	0	Enable Forward Facing Primitive

Table 38: Control Record IDs and Data Summary (Cont.)

Control Codes			
	Synopsis (B = binning only, R = rendering only)		
Code	***Bits***	***Offset***	***Field Description***
97	Flat Shade Flags		
	32	0	Flat-shading Flags (32x1-bit)
98	Points size		
	32	0	Point Size (float32)
99	Line Width		
	32	0	Line Width (float32)
100	RHT X boundary		
	16	0	RHT primitive X boundary (sint16)
101	Depth Offset		
	16	16	Depth Offset Units (float1-8-7)
	16	0	Depth Offset Factor (float1-8-7)
102	Clip Window		
	16	48	Clip Window Height in pixels (uint16)
	16	32	Clip Window Width in pixels (uint16)
	16	16	Clip Window Bottom pixel coordinate (uint16)
	16	0	Clip Window Left pixel coordinate (uint16)
103	Viewport Offset		
	16	16	Viewport Centre Y-coordinate (sint16)
	16	0	Viewport Centre X-coordinate (sint16)
104	Z min and max clipping planes		
	32	32	Maximum Z_W (float32)
	32	0	Minimum Z_W (float32)
105	Clipper XY Scaling (B)		
	32	32	Viewport Half-Height in $1/16^{th}$ of pixel (float32)
	32	0	Viewport Half-Width in $1/16^{th}$ of pixel (float32)
105	Clipper Z Scale and Offset (B)		
	32	32	Viewport Z Offset (Zc to Zs) (float32)
	32	0	Viewport Z Scale (Zc to Zs) (float32)
107–111	Reserved		

Table 38: Control Record IDs and Data Summary (Cont.)

Control Codes			
	Synopsis (B = binning only, R = rendering only)		
Code	***Bits***	***Offset***	***Field Description***
112	Tile Binning Mode Configuration (B)		
	1	119	Double-buffer in non-ms mode
	2	117	Tile Allocation Block Size (32,64,128,256 bytes)
	2	115	Tile Allocation Initial Block Size (32,64,128,256 bytes)
	1	114	Auto-initialise Tile State Data Array
	1	113	Tile Buffer 64-bit Color Depth
	1	112	Multisample Mode (4x)
	8	104	Height (in tiles)
	8	96	Width (in tiles)
	32	64	Tile State Data Array Base Address (16-byte aligned, size of 48 bytes * num tiles)
	32	32	Tile Allocation Memory Size (bytes)
	32	0	Tile Allocation Memory Address
113	Tile Rendering Mode Configuration (R)		
	3	77	Unused
	1	76	Double-buffer in non-ms mode
	1	75	Early-Z/Early-Cov disable
	1	74	Early-Z Update Direction (0=lt/le, 1=gt/ge)
	1	73	Select Coverage Mode
	1	72	Enable VG Mask buffer
	2	70	Memory Format (0=Linear, 1 = T-format, 2 = LT-format)
	2	68	Decimate mode (0=1x, 1=4x, 2=16x)
	2	66	Non-HDR Frame Buffer Color format (0,1,2=bgr565 dithered, rgba8888, bgr565)
	1	65	Tile Buffer 64-bit Color Depth (HDR mode)
	1	64	Multisample Mode (4x)
	16	48	Height (pixels) (uint16)
	16	32	Width (pixels) (uint16)
	32	0	Memory Address
114	Clear Colors (R)		
	8	96	Clear Stencil (uint8)
	8	88	Clear VG Mask (uint8)
	24	64	Clear Z_S (uint24)
	64	0	Clear Color (2xrgba8888 or rgba16161616)
115	Tile Coordinates (R)		
	8	8	Tile Row Number (int8)
	8	0	Tile Column Number (int8)
116–255	Reserved		

Primitive List Formats

Indexed Primitive List (id = 32) and Vertex Array Primitives (id = 33) records reference primitive lists supplied by the standard OpenGL-ES API functions glDrawElements and glDrawArrays. The other formats are internal, as follows.

VG Coordinate Array Primitives (ID=41)

VG Coordinate Array Primitives are only intended for use in VG shader mode. The vertex screen coordinates (Xs, Ys) are directly supplied as 32-bit input list values (Xs in bits[15:0], Ys in bits[31:16]). Vertex caching and Vertex shading are completely bypassed, and Ws and Zs take fixed values of 1.0 and 0.5 respectively.

VG Coordinate Array Primitive lists can be terminated prematurely and continued in the same manner as VG Inline primitives, as described in the following section.

VG Inline Primitives (ID=42)

VG Inline Primitives are the same as VG Coordinate Array Primitives, but with list of primitives following inline in the control list. The inline list is terminated with the special code value of 0xbfff0000 in place of certain vertices. Triangles, triangle fans and triangle strip lists are all terminated by the 0xbfff0000 code in the 3rd vertex; RHTs and RHT strips must have the termination code in the second vertex. As a special case, the first primitive of a triangle fan can also be terminated by an 0xbfff0000 code in the second vertex, but all 3 vertices must still be present.

Triangle fans may be continued with a continuation list. The continuation list must start off with a triangle of 3 vertices, but the first 1 or 2 vertices are ignored and replaced by the first 1 or 2 vertices of the last triangle in the previous list, which must also have been an XY coordinate triangle fan list. The second vertex of the continuation list is only used in the special case where the previous fan terminated on the second vertex of the first triangle.

The inline list can also be padded with dummy primitives by using the code value of 0xbfff0001 in place of the 3^{rd} vertex of a triangle or 2^{nd} vertex of an RHT. This facility is included to allow fixed length inline lists with variable numbers of primitives.

Compressed Primitive List (ID=48)

Compressed primitive lists are produced by the PTB, and always follow inline in the control list, and code each primitive in a variable number of bytes. Compressed lists are terminated with an escape code, and can also contain embedded branch records. The embedded branches allow the list to be composed of multiple chained memory blocks without wasting memory escaping from and restarting the compressed list.

The format for compressed lists is set by the Primitive List Format control record (id=56). There are 6 format variants of indexed list, namely for Triangles, Lines/RHTs and Points, encoding either 16-bit or 24-bit indices. There are two further formats which encode 16+16-bit (x,y) coordinates for Triangles and RHTs only. The 8 compressed list format variants are defined by the following coding tables.

Table 39: Compressed Triangles List Indices

Compressed Triangles List, 16-bit [24-bit] indices	
Bits	***Content***
Coding 0 (1 byte, 1 relative index)	
1:0	0 => New tri indices (0,1) in common with prev tri indices (2,1) 1 => New tri indices (0,1) in common with prev tri indices (0,2) 2 => New tri indices (0,1) in common with prev tri indices (1,0)
7:2	2's complement difference between new tri index (2) and prev tri index (2). Range (-31, +31). 32 is reserved for special codes.
Coding 1 (2 bytes, 3 relative indices)	
1:0	=3
3:2	=0, 1 or 2
7:4	2's complement difference between new tri index (0) and prev tri index (0) (range -8, +7)
11:8	2's complement difference between new tri index (1) and prev tri index (1) (range -8, +7)
15:12	2's complement difference between new tri index (2) and prev tri index (2) (range -8, +7)
Coding 2 (4 {5} bytes, 1 absolute, 2 relative indices)	
3:0	=15
9:4	2's complement difference between new tri index (1) and new tri index (0) (range -32, +31)
15:10	2's complement difference between new tri index (2) and new tri index (0) (range -32, +31)
31:16 {39:16}	Absolute new tri index (0)
Coding 3 (7 {10} bytes, 3 absolute indices)	
7:0	=129
23:8 [31:8]	Absolute new tri index (0)
39:24 [55:32]	Absolute new tri index (1)
55:40 [79:56]	Absolute new tri index (2)
Relative Branch (3 bytes)	
7:0	=130
23:8	2's complement relative branch (lsb = 32 bytes, range ±1 Mbyte)
Escape Code (1 byte)	
7:0	=128

Table 40: Compressed Lines or RHTs List Indices

Compressed Lines or RHTs List, 16-bit [24-bit] indices	
Bits	***Content***
Coding 0 (1 byte, 1 relative index)	
1:0	0 => New prim index (0) in common with prev prim index (1) 1 => New prim index (0) in common with prev prim index (0)
7:2	2's complement difference between new prim index (1) and prev prim index (1). Range (-31, +31). 32 is reserved for special codes.
Coding 1 (2 bytes, 2 relative indices)	
1:0	=3
3:2	=0, 1 or 2
7:4	2's complement difference between new tri index (0) and prev tri index (0) (range -8, +7)
11:8	2's complement difference between new tri index (1) and prev tri index (1) (range -8, +7)
15:12	Not used
Coding 2 (3 [4] bytes, 1 absolute, 1 relative index)	
1:0	=2
7:2	2's complement difference between new prim index (1) and new prim index (0) Range (-31, +31). 32 is reserved for special codes.
23:8 [31:8]	Absolute new prim index (0)
Coding 3 (5 [7] bytes, 2 absolute indices)	
7:0	=129
23:8 [31:8]	Absolute new prim index (0)
39:24 [55:32]	Absolute new prim index (1)
Relative Branch (3 bytes)	
7:0	=130
23:8	2's complement relative branch (LSB = 32 bytes, range ±1 Mbyte)
Escape Code (1 byte)	
7:0	=128

Table 41: Compressed Points List Indices

Compressed Points List, 16-bit [24-bit] indices	
Bits	***Content***
Coding 0 (1 byte, relative index)	
1:0	=0 or 1
7:2	2's complement difference between new point index and prev point index. Range (-31, +31). 32 is reserved for special codes.
Coding 1 (2 bytes, relative index)	
1:0	=3
15:2	2's complement difference between new point index and prev point index (range -8192, +8191)
Coding 2 (2 bytes, relative index, run length) – NOT IMPLEMENTED	
1:0	=2
7:2	2's complement difference between first point index and prev point index. Range (-31, +31). 32 is reserved for special codes.
15:8	Number of additional points with consecutive indices
Coding 3 (3 [4] bytes, absolute index)	
7:0	=129
23:8 [31:8]	Absolute new point index
Relative Branch (3 bytes)	
7:0	=130
23:8	2's complement relative branch (lsb = 32 bytes, range ±1 Mbyte)
Escape Code (1 byte)	
7:0	=128

Table 42: Compressed Triangles List Coordinates

Compressed Triangles List, 16+16-bit (x,y) Coordinates	
Bits	***Content***
Coding 0 (2 bytes, 1 relative coordinate)	
1:0	0 => New tri indices (0,1) in common with prev tri indices (2,1) 1 => New tri indices (0,1) in common with prev tri indices (0,2) 2 => New tri indices (0,1) in common with prev tri indices (1,0)
7:2, 8	2's complement difference between new tri x (2) and prev tri x (2). Range (-62, +63). 64,65 are reserved for special codes.
15:9	2's complement difference between new tri y (2) and prev tri y (2). Range (-64, +63).

Table 42: Compressed Triangles List Coordinates (Cont.)

Compressed Triangles List, 16+16-bit (x,y) Coordinates	
Bits	***Content***
Coding 1 (3 bytes, 1 relative coordinate)	
1:0	=3
3:2	0 => New tri indices (0,1) in common with prev tri indices (2,1) 1 => New tri indices (0,1) in common with prev tri indices (0,2) 2 => New tri indices (0,1) in common with prev tri indices (1,0)
13:4	2's complement difference between new tri x (2) and prev tri x (2). Range (-512, +511).
23:14	2's complement difference between new tri y (2) and prev tri y (2). Range (-512, +511).
Coding 2 (8 bytes, 1 absolute, 2 relative coordinates)	
3:0	=15
10:4	2's complement difference between new tri x (1) and new tri x (0). Range (-64, +63).
17:11	2's complement difference between new tri y (1) and new tri y (0). Range (-64, +63).
24:18	2's complement difference between new tri x (2) and new tri x (0). Range (-64, +63).
31:25	2's complement difference between new tri y (2) and new tri y (0). Range (-64, +63).
47:32	Absolute new tri x (0)
63:48	Absolute new tri y (0)
Coding 3 (13 bytes, 3 absolute coordinates)	
7:0	=129
39:8	Absolute new tri coordinate (y,x) (0)
71:40	Absolute new tri coordinate (y,x) (1)
103:72	Absolute new tri coordinate (y,x) (2)
Relative Branch (3 bytes)	
7:0	=130
23:8	2's complement relative branch (lsb = 32bytes, range ±1Mbyte)
Escape Code (1 byte)	
7:0	=128

Table 43: Compressed RHTs List Coordinates

Compressed RHTs List, 16+16-bit (x,y) Coordinates	
Bits	***Content***
Coding 0 (2 bytes, 1 relative coordinate)	
1:0	0 => New RHT index (0) in common with prev RHT index (1) 1 => New RHT index (0) in common with prev RHT index (0)
7:2, 8	2's complement difference between new RHT x (1) and prev RHT x (1). Range (-62, +63). 64,65 are reserved for special codes.
15:9	2's complement difference between new RHT y (1) and prev RHT y (1). Range (-64, +63).
Coding 1 (3 bytes, 1 relative coordinate)	
1:0	=3
3:2	0,2 => New RHT index (0) in common with prev RHT index (1) 1 => New RHT index (0) in common with prev RHT index (0)
13:4	2's complement difference between new RHT x (1) and prev RHT x (1). Range (-512, +511).
23:14	2's complement difference between new RHT y (1) and prev RHT y (1). Range (-512, +511).
Coding 2 (6 bytes, 1 absolute, 1 relative coordinates)	
1:0	=2
7:2, 8	2's complement difference between new RHT x (1) and new RHT x (0). Range (-62, +63). 64,65 are reserved for special codes.
15:9	2's complement difference between new RHT y (1) and new RHT y (0). Range (-64, +63).
31:16	Absolute new RHT x (0)
47:32	Absolute new RHT y (0)
Coding 3 (9 bytes, 2 absolute coordinates)	
7:0	=129
39:8	Absolute new RHT coordinate (y,x) (0)
71:40	Absolute new RHT coordinate (y,x) (1)
Relative Branch (3 bytes)	
7:0	=130
23:8	2's complement relative branch (lsb = 32 bytes, range ±1 Mbyte)
Escape Code (1 byte)	
7:0	=128

Clipped Primitive (with Compressed Primitive List) (ID=49)

This indirect record specifies clipped vertex parameters for a single 3D clipped primitive that is generated during tile binning. The record is variable length, only including data for those vertices that need clipping. This only applies to the first primitive in the following compressed primitive list.

Table 44: Clipped Primitive Record

Clipped Primitive	
Bytes	***Content***
0–1 + n*32	X_S coordinate (for n^{th} clipped vertex)
2–3 + n*32	Y_S coordinate
4–7 + n*32	Z_S
8–11 + n*32	$1/W_C$
12–15 + n*32	Varying interpolation vertex 0 coefficient
16–19 + n*32	Varying interpolation vertex 1 coefficient
20–23 + n*32	Varying interpolation vertex 2 coefficient (ignored for lines and RHTs)
24–31 + n*32	Unused

Shader State Record Formats

These specify all data associated with a shader. For normal 3D (GL) mode this includes all the vertex, coordinate and fragment shader programs, uniforms data, vertex attribute array formats and address and shaded vertex data format. There is a different set of data for OpenVG mode, which is used to select any non-vertex shading mode.

For non-vertex shading (NV) mode, where pre-shaded vertex is provided in memory, a simpler set of data is used, without vertex and coordinate shaders, with a single shaded vertex data address and stride in place of multiple vertex attribute arrays.

For VG mode, only a fragment shader with uniforms is supplied. This mode may only use VG Coordinate Array primitive lists (Id = 41) as input.

The GL shader state record is variable length, according to the ls 4 bits of the GL Shader State list item (ID = 64).

Table 45: GL Shader State Record

GL Shader State Record (ID=64)	
Bytes	***Content***
0–1	Flag bits: 2: Enable Clipping 1: Point Size included in shaded vertex data 0: Fragment Shader is single threaded

Table 45: GL Shader State Record (Cont.)

GL Shader State Record (ID=64)	
Bytes	***Content***
2	Fragment Shader Number of Uniforms (not used currently)
3	Fragment Shader Number of Varyings
4–7	Fragment Shader Code Address
8–11	Fragment Shader Uniforms Address
12–13	Vertex Shader Number of Uniforms (not used currently)
14	Vertex Shader Attribute Array select bits (8 bits for 8 attribute arrays)
15	Vertex Shader Total Attributes Size
16–19	Vertex Shader Code Address
20–23	Vertex Shader Uniforms Address
24–25	Coordinate Shader Number of Uniforms (not used currently)
26	Coordinate Shader Attribute Array select bits (8 bits for 8 attribute arrays)
27	Coordinate Shader Total Attributes Size
28–31	Coordinate Shader Code Address
32–35	Coordinate Shader Uniforms Address
36–39 + n*8	Attribute Array [n] Base Memory Address (n = 0-7)
40 + n*8	Attribute Array [n] Number of Bytes-1
41 + n*8	Attribute Array [n] Memory Stride
42 + n*8	Attribute Array [n] Vertex Shader VPM Offset (from Base Address)
43 + n*8	Attribute Array [n] Coordinate Shader VPM Offset (from Base Address)
100–103 + n*4	Extended Attribute Array [n] Memory Stride (optional)

Table 46: NV Shader State Record

NV Shader State Record (no vertex shading) (ID=65)	
Bytes	***Content***
0	Flag bits: 3: Clip Coordinates header included in shaded vertex data 2: Enable Clipping 1: Point Size included in shaded vertex data 0: Fragment Shader is single threaded
1	Shaded Vertex Data Stride
2	Fragment Shader Number of Uniforms (not used currently)
3	Fragment Shader Number of Varyings
4–7	Fragment Shader Code Address
8–11	Fragment Shader Uniforms Address
12–15	Shaded Vertex Data Address (128-bit aligned if including clip coordinate header)

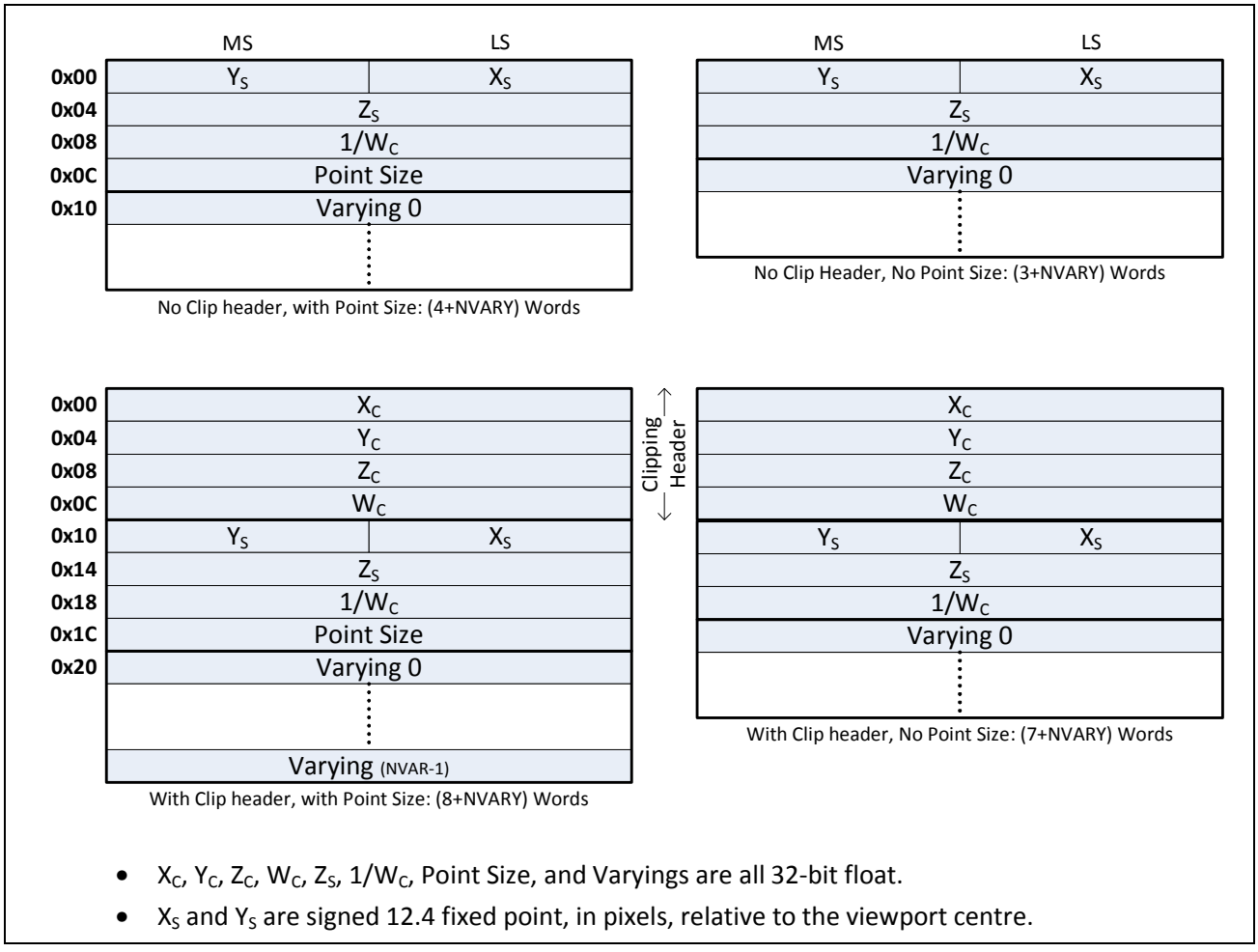
Table 47: VG Shader State Record

VG Shader State Record (ID=66)	
Bytes	***Content***
0	Flag bits: 0: Fragment Shader is single threaded
1	Not used
2	Fragment Shader Number of Uniforms (not used currently)
3	Fragment Shader Number of Varyings
4–7	Fragment Shader Code Address
8–11	Fragment Shader Uniforms Address

Shaded Vertex Format in Memory

In the non-vertex shading (NV) mode, which is set by the NV Shader State Record (ID=65), the memory data format for each (pre-shaded) vertex is one of the following four forms.

Figure 12: Shaded Vertex Memory Formats



- X_C, Y_C, Z_C, W_C, Z_S, $1/W_C$, Point Size, and Varyings are all 32-bit float.
- X_S and Y_S are signed 12.4 fixed point, in pixels, relative to the viewport centre.

Section 10: V3D Registers

There are relatively few memory mapped registers as most of the 3D programmability is provided by the control lists. Since the 'register state' for the various 3D components changes dynamically as primitives flow through the system, there is little utility in being able to read-back such state.

V3D Register Address Map

Table 48: V3D Registers

Address Offset	*Register Name*	*Description*	*Size*
0x00000	V3D_IDENT0	V3D Identification 0 (V3D block identity)	32
0x00004	V3D_IDENT1	V3D Identification 1 (V3D Configuration A)	32
0x00008	V3D_IDENT2	V3D Identification 1 (V3D Configuration B)	32
0x00010	V3D_SCRATCH	Scratch Register	32
0x00020	V3D_L2CACTL	L2 Cache Control	32
0x00024	V3D_SLCACTL	Slices Cache Control	32
0x00030	V3D_INTCTL	Interrupt Control	32
0x00034	V3D_INTENA	Interrupt Enables	32
0x00038	V3D_INTDIS	Interrupt Disables	32
0x00100	V3D_CT0CS	Control List Executor Thread 0 Control and Status.	32
0x00104	V3D_CT1CS	Control List Executor Thread 1 Control and Status.	32
0x00108	V3D_CT0EA	Control List Executor Thread 0 End Address.	32
0x0010c	V3D_CT1EA	Control List Executor Thread 1 End Address.	32
0x00110	V3D_CT0CA	Control List Executor Thread 0 Current Address.	32
0x00114	V3D_CT1CA	Control List Executor Thread 1 Current Address.	32
0x00118	V3D_CT00RA0	Control List Executor Thread 0 Return Address.	32
0x0011c	V3D_CT01RA0	Control List Executor Thread 1 Return Address.	32
0x00120	V3D_CT0LC	Control List Executor Thread 0 List Counter	32
0x00124	V3D_CT1LC	Control List Executor Thread 1 List Counter	32
0x00128	V3D_CT0PC	Control List Executor Thread 0 Primitive List Counter	32
0x0012c	V3D_CT1PC	Control List Executor Thread 1 Primitive List Counter	32
0x00130	V3D_PCS	V3D Pipeline Control and Status	32
0x00134	V3D_BFC	Binning Mode Flush Count	32
0x00138	V3D_RFC	Rendering Mode Frame Count	32
0x00300	V3D_BPCA	Current Address of Binning Memory Pool	32
0x00304	V3D_BPCS	Remaining Size of Binning Memory Pool	32

Table 48: V3D Registers (Cont.)

Address Offset	*Register Name*	*Description*	*Size*
0x00308	V3D_BPOA	Address of Overspill Binning Memory Block	32
0x0030c	V3D_BPOS	Size of Overspill Binning Memory Block	32
0x00310	V3D_BXCF	Binner Debug	32
0x00410	V3D_SQRSV0	Reserve QPUs 0-7	32
0x00414	V3D_SQRSV1	Reserve QPUs 8-15	32
0x00418	V3D_SQCNTL	QPU Scheduler Control	32
0x00430	V3D_SRQPC	QPU User Program Request Program Address	32
0x00434	V3D_SRQUA	QPU User Program Request Uniforms Address	32
0x00438	V3D_SRQUL	QPU User Program Request Uniforms Length	32
0x0043c	V3D_SRQCS	QPU User Program Request Control and Status	32
0x00500	V3D_VPACNTL	VPM Allocator Control	32
0x00504	V3D_VPMBASE	VPM base (user) memory reservation	32
0x00670	V3D_PCTRC	Performance Counter Clear	32
0x00674	V3D_PCTRE	Performance Counter Enables	32
0x00680	V3D_PCTR0	Performance Counter Count 0	32
0x00684	V3D_PCTRS0	Performance Counter Mapping 0	32
0x00688	V3D_PCTR1	Performance Counter Count 1	32
0x0068c	V3D_PCTRS1	Performance Counter Mapping 1	32
0x00690	V3D_PCTR2	Performance Counter Count 2	32
0x00694	V3D_PCTRS2	Performance Counter Mapping 2	32
0x00698	V3D_PCTR3	Performance Counter Count 3	32
0x0069c	V3D_PCTRS3	Performance Counter Mapping 3	32
0x006a0	V3D_PCTR4	Performance Counter Count 4	32
0x006a4	V3D_PCTRS4	Performance Counter Mapping 4	32
0x006a8	V3D_PCTR5	Performance Counter Count 5	32
0x006ac	V3D_PCTRS5	Performance Counter Mapping 5	32
0x006b0	V3D_PCTR6	Performance Counter Count 6	32
0x006b4	V3D_PCTRS6	Performance Counter Mapping 6	32
0x006b8	V3D_PCTR7	Performance Counter Count 7	32
0x006bc	V3D_PCTRS7	Performance Counter Mapping 7	32
0x006c0	V3D_PCTR8	Performance Counter Count 8	32
0x006c4	V3D_PCTRS8	Performance Counter Mapping 8	32
0x006c8	V3D_PCTR9	Performance Counter Count 9	32
0x006cc	V3D_PCTRS9	Performance Counter Mapping 9	32
0x006d0	V3D_PCTR10	Performance Counter Count 10	32
0x006d4	V3D_PCTRS10	Performance Counter Mapping 10	32
0x006d8	V3D_PCTR11	Performance Counter Count 11	32

Table 48: V3D Registers (Cont.)

Address Offset	*Register Name*	*Description*	*Size*
0x006dc	V3D_PCTRS11	Performance Counter Mapping 11	32
0x006e0	V3D_PCTR12	Performance Counter Count 12	32
0x006e4	V3D_PCTRS12	Performance Counter Mapping 12	32
0x006e8	V3D_PCTR13	Performance Counter Count 13	32
0x006ec	V3D_PCTRS13	Performance Counter Mapping 13	32
0x006f0	V3D_PCTR14	Performance Counter Count 14	32
0x006f4	V3D_PCTRS14	Performance Counter Mapping 14	32
0x006f8	V3D_PCTR15	Performance Counter Count 15	32
0x006fc	V3D_PCTRS15	Performance Counter Mapping 15	32
0x00f00	V3D_DBGE	PSE Error Signals	32
0x00f04	V3D_FDBGO	FEP Overrun Error Signals	32
0x00f08	V3D_FDBGB	FEP Interface Ready and Stall Signals, FEP Busy Signals	32
0x00f0c	V3D_FDBGR	FEP Internal Ready Signals	32
0x00f10	V3D_FDBGS	FEP Internal Stall Input Signals	32
0x00f20	V3D_ERRSTAT	Miscellaneous Error Signals (VPM, VDW, VCD, VCM, L2C)	32

V3D Register Definitions

Control List Executor Registers (Per Thread)

Thread 0 is for tile binning and thread 1 is for tile rendering.

Table 49: V3D_CTnCS Register Description

Synopsis	***Control List Executor Thread n Control and Status.***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:16	–	Reserved – write zeros, read as don't care		
15	CTRSTA	Reset bit Writing 1 stops the control thread and resets all bits in this register	W	0
14:12	CTSEMA	Counting Semaphore Current state of the counting semaphore for this thread	R	0
11:10	–	Reserved – write zeros, read as don't care		
9:8	CTRTSD	Return Stack Depth Number of levels of list nesting	R	0
7:6	–	Reserved – write zeros, read as don't care		
5	CTRUN	Control Thread Run 0 = Stopped 1 = Running Writing 1 stops the thread and sets the state to 'Stopped at halt'	R/W	0
4	CTSUBS	Control Thread Sub-mode If RUN = 0: 0 = Stopped at end 1 = Stopped at halt (halt command or user halt) If RUN = 1: 0 = Running normally 1 = Stalled, waiting for other thread to complete in this mode. Writing 1 clears any 'Stopped at halt' condition, and causes the thread to start running if CTLCA != CTLEA	R/W	0
3	CTERR	Control Thread Error Set when stopped with an error, Cleared on restarting	R	0
2:1	–	Reserved – write zeros, read as don't care		
0	CTMODE	Control Thread Mode (binning mode thread only) 1 = Binning Mode, Prefixing (state counter = 0) 0 = Binning Mode, Tile lists started (state counter > 0)	R	0

Table 50: V3D_CTnEA Register Description

Synopsis	***Control List Executor Thread n End Address***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:0	CTLEA	Control List End Address Set to the byte address after the last record in the control list. Writing to this register causes the thread to start running if stopped, but only if CTSUBS is 'Stopped at end'.	R/W	0

Table 51: V3D_CTnCA Register Description

Synopsis	***Control List Executor Thread n Current Address***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:0	CTLCA	Control List Current Address Points to the address of the current record in the control list, or the first record to be processed when stopped. This register can only be written when CTRUN is Stopped. Writing a new address to this register sets CTSUBS to 'Stopped at end'.	R/W	0

Table 52: V3D_CTnRA0 Register Description

Synopsis	***Control List Executor Thread n Return Address 0***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:0	CTLRA	Control List Return Address 0 Address on return address stack. (N.B. We only support a one-deep return address stack)	R	0

Table 53: V3D_CTnLC Register Description

Synopsis	***Control List Executor Thread List Counter***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:16	CTLLCM	Major List Counter Count of Flush commands encountered Reset by writing 1 to bit[16]	R/W	0
15:0	CTLSLCS	Sub-list Counter Count of Return commands encountered Reset by writing 1 to bit[0]	R/W	0

Table 54: V3D_CTnPC Register Description

Synopsis	*Control List Executor Thread Primitive List Counter*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:0	CTLPC	Primitive List Counter Count of primitives remaining whilst processing a primitive list	R	0

V3D Pipeline Registers

Table 55: V3D_PCS Register Description

Synopsis	*V3D Pipeline Control and Status*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:9	–	Reserved – write zeros, read as don't care		
8	BMOOM	Binning Mode Out Of Memory Set when PTB runs out of binning memory while binning. Cleared by writing to V3DBPOS or by BMRST.	R	0
7:4	–	Reserved – write zeros, read as don't care		
3	RMBUSY	Rendering Mode Busy Set while any rendering operations are actually in progress. Clear when rendering operations are stalled or rendering pipeline is empty.	R	0
2	RMACTIVE	Rendering Mode Active Set while rendering pipeline is in use. Clear when rendering pipeline is completely empty (not in use).	R	0
1	BMBUSY	Binning Mode Busy Set while any binning operations are actually in progress. Clear when binning operations are stalled or binning pipeline is empty.	R	0
0	BMACTIVE	Binning Mode Active Set while binning pipeline is in use. Clear when binning pipeline is completely empty (not in use).	R	0

Table 56: V3D_BFC Register Description

Synopsis	***Binning Mode Flush Count***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:8	–	Reserved – write zeros, read as don't care		
7:0	BMFCT	Flush Count The count is incremented in binning mode once the PTB has flushed all tile lists to memory and the PTB has finished with the tile state data array. Writing 1 clears this count.	R/W	0

Table 57: V3D_RFC Register Description

Synopsis	***Rendering Mode Frame Count***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:8	–	Reserved – write zeros, read as don't care		
7:0	RMFCT	Frame Count The count is incremented in rendering mode when the last Tile Store operation of the frame is complete, that is, the tile has been fully written out to memory. Writing 1 clears this count.	R/W	0

Table 58: V3D_BPCA Register Description

Synopsis	***Current Address of Binning Memory Pool***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:0	BMPCA	Current Pool Address The address of the current allocation pointer in the binning memory pool.	R	0

Table 59: V3D_BPCS Register Description

Synopsis	***Remaining Size of Binning Memory Pool***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:0	BMPRS	Size of Pool Remaining The number of bytes remaining in the binning memory pool.	R	0

Table 60: V3D_BPOA Register Description

Synopsis	***Address of Overspill Binning Memory Block***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:0	BMPOA	Address of Overspill Memory Block for Binning The address of additional memory that the PTB can use for binning once the initial pool runs out. This may be set up prior to the PTB actually running out.	R/W	0

Table 61: V3D_BPOS Register Description

Synopsis	***Size of Overspill Binning Memory Block***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:0	BMPOS	Size of Overspill Memory Block for Binning The number of bytes of additional memory that the PTB can use for binning once the initial pool runs out. If this count is zero when the PTB runs out of binning memory, the PTB will halt, waiting for a non-zero value to be written to this register. Once the PTB has taken this overspill memory this register is set to 0. The overspill memory may be set up prior to the PTB actually running out, in which case the PTB can take the new memory and carry on.	R/W	0

QPU Scheduler Registers

Table 62: V3D_SQRSV0 Register Description

Synopsis	***Reserve QPUs 0–7***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:28	QPURSV7	Reservation settings for QPU 7	R/W	0
27:24	QPURSV6	Reservation settings for QPU 6	R/W	0
23:20	QPURSV5	Reservation settings for QPU 5	R/W	0
19:16	QPURSV4	Reservation settings for QPU 4	R/W	0
15:12	QPURSV3	Reservation settings for QPU 3	R/W	0
11:8	QPURSV2	Reservation settings for QPU 2	R/W	0
7:4	QPURSV1	Reservation settings for QPU 1	R/W	0
3:0	QPURSV0	Reservation settings for QPU 0 B[0] set: Do not use for User Programs B[1] set: Do not use for Fragment Shaders B[2] set: Do not use for Vertex Shaders B[3] set: Do not use for Coordinate Shaders	R/W	0

Table 63: V3D_SQRSV1 Register Description

Synopsis	***Reserve QPUs 8–15***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:28	QPURSV15	Reservation settings for QPU 15	R/W	0
27:24	QPURSV14	Reservation settings for QPU 14	R/W	0
23:20	QPURSV13	Reservation settings for QPU 13	R/W	0
19:16	QPURSV12	Reservation settings for QPU 12	R/W	0
15:12	QPURSV11	Reservation settings for QPU 11	R/W	0
11:8	QPURSV10	Reservation settings for QPU 10	R/W	0
7:4	QPURSV9	Reservation settings for QPU 9	R/W	0
3:0	QPURSV8	Reservation settings for QPU 8	R/W	0

Table 64: V3D_SQCNTL Register Description

Synopsis	***QPU Scheduler Control***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:4	–	Reserved – write zeros, read as don't care		
3:2	CSRBL	Coordinate Shader Scheduling Bypass Limit Same function as VSRBL for coordinate shaders.	R/W	0
1:0	VSRBL	Vertex Shader Scheduling Bypass Limit Sets a limit on how many times a threaded (fragment) shader can be allocated to QPUs (with just a single thread free), whilst there is a vertex shader waiting to be scheduled. The limit count is 2^VSRBL, allowing limit counts of 1,2,4 or 8.	R/W	0

\

Table 65: V3D_SRQPC Register Description

Synopsis	***QPU User Program Request Program Address***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:0	QPURQPC	Program Address Writing this register queues a request to run a program starting at the given address, with a uniforms queue starting at the address given by V3DRQUA This is a write-only port to the user request FIFO, and can't be read back. The FIFO is 16 requests deep. If the FIFO is full the request is ignored, the QPURQERR bit in the V3DRQCS register is set and further request are ignored until the error bit is cleared.	W	0

Table 66: V3D_SRQUA Register Description

Synopsis	*QPU User Program Request Uniforms Address*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:0	QPURQUA	Uniforms Address Contains the address of the uniforms stream for the next user program to be queued via a write to V3DRQPC. An address of zero disables the uniforms stream.	R/W	0

Table 67: V3D_SRQUL Register Description

Synopsis	*QPU User Program Request Uniforms Length*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:12	–	Reserved – write zeros, read as don't care		
11:0	QPURQUL	Uniforms Length Contains the maximum length of the uniforms stream for the next user program to be queued via a write to V3DRQPC. A length of zero disables the uniforms stream, and a length > 1023 implies an unlimited uniforms stream.	R/W	0

Table 68: V3D_SRQCS Register Description

Synopsis	*QPU User Program Request Control and Status*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:24	–	Reserved – write zeros, read as don't care		
23:16	QPURQCC	Count of user programs completed Contains the total number of user programs that have run and completed, modulo 256. Reset by writing 1 to bit[16].	R/W	0
15:8	QPURQCM	Count of user program requests made Contains the total number of user program requests made, modulo 256. This is incremented even if the queue is currently full or QPURQERR is set. Reset by writing 1 to bit[8].	R/W	0
7	QPURQERR	Queue Error Set when a request has been made when the queue is full. Reset by writing 1	R/W	0
6	–	Reserved – write zeros, read as don't care		
5:0	QPURQL	Queue Length Contains the number of program requests currently queued. Writing 1 to bit[0] clears the queue.	R/W	0

VPM Registers

Table 69: V3D_VPMBASE Register Description

Synopsis	***VPM base (user) memory reservation***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:5	–	Reserved – write zeros, read as don't care		
4:0	VPMURSV	VPM memory reserved for user programs Contains the amount of VPM memory reserved for all user programs, in multiples of 256 bytes (4x 16-way 32-bit vectors). Can only be written when the V3D system is idle before any coordinate, vertex or 'NV' shading has commenced.	R/W	0

Table 70: V3D_VPACNTL Register Description

Synopsis	***VPM Allocator Control***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:14	–	Reserved – write zeros, read as don't care		
13	VPATOEN	Enable VPM allocation timeout Enables VPM memory allocation timeout using VPARATO and VPABATO. This stops one of binning or rendering mode hogging all the VPM with small allocations whilst the other is waiting for a large allocation.	R/W	0
12	VPALIMEN	Enable VPM allocation limits Enables VPM memory allocation limiting using VPARALIM and VPABALIM.	R/W	0
11:9	VPABATO	Binning VPM allocation timeout Sets a timeout for raising the priority of Binning mode allocation requests. Same function as VPARATO but for binning mode allocation instead.	R/W	0
8:6	VPARATO	Rendering VPM allocation timeout Sets a timeout for raising the priority of Rendering mode allocation requests. Timeout is ((VPARATO < 5) ? VPARATO*2+2: VPARATO*8-24) allocation cycles. If VPATOEN is enabled and timeout number of *binning* allocations succeed while a rendering allocation request is waiting, subsequent binning requests are forced to wait until the rendering allocation succeeds.	R/W	0
5:3	VPABALIM	Binning VPM allocation limit Limits the amount of VPM memory allocated to binning mode. Same function as VPARALIM but for binning mode memory allocation.	R/W	0

Table 70: V3D_VPACNTL Register Description (Cont.)

Synopsis	*VPM Allocator Control*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
2:0	VPARALIM	Rendering VPM allocation limit Limits the amount of VPM memory allocated to rendering mode. If VPALIMEN is enabled, the maximum VPM memory that will be allocated for render mode allocations = (VPARALIM+1)*6*256 bytes.	R/W	0

Cache Control Registers

Table 71: V3D_L2CACTL Register Description

Synopsis	*L2 Cache Control*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:3	–	Reserved – write zeros, read as don't care		
2	L2CCLR	L2 Cache Clear Write '1' to clear the L2 Cache	W	0
1	L2CDIS	L2 Cache Disable Write '1' to disable the L2 Cache	W	0
0	L2CENA	L2 Cache Enable Reads state of cache enable bit. Write '1' to enable the L2 Cache (write of '0' has no effect)	R/W	0

Table 72: V3D_SLCACTL Register Description

Synopsis	*Slices Cache Control*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:28	–	Reserved – write zeros, read as don't care		
27:24	T1CCS0_to_T1CCS3	TMU1 Cache Clear Bits (1 bit per slice in system config) Write '1' to clear TMU1 cache.	W	0
23:20	–	Reserved – write zeros, read as don't care		
19:16	T0CCS0_to_T0CCS3	TMU0 Cache Clear Bits (1 bit per slice in system config) Write '1' to clear TMU0 cache.	W	0
15:12	–	Reserved – write zeros, read as don't care		
11:8	UCCS0_to_UCCS3	Uniforms Cache Clear Bits (1 bit per slice in system config) Write '1' to clear uniforms cache.	W	0

Table 72: V3D_SLCACTL Register Description (Cont.)

Synopsis	*Slices Cache Control*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
7:4	–	Reserved – write zeros, read as don't care		
3:0	ICCS0_to_ICCS3	Instruction Cache Clear Bits (1 bit per slice in system config) Write '1' to clear instruction cache.	W	0

QPU Interrupt Control

These are the interrupts generated by QPU instructions, intended for use for host coordination of general-purpose QPU programs. The interrupt bits are all latched and must be cleared via the V3DDBQITC register.

Table 73: V3D_DBQITC Register Description

Synopsis	*QPU Interrupt Control*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:16	–	Reserved – write zeros, read as don't care		
15:0	IC_QPU0_to_IC_QPU15	QPU Interrupt Control bits (1 bit per QPU in system config) Reads 1 when interrupt is latched. Write 1 to clear interrupt.	R/W	0

Table 74: V3D_DBQITE Register Description

Synopsis	*QPU Interrupt Enables*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:16	–	Reserved – write zeros, read as don't care		
15:0	IE_QPU0_to_IE_QPU15	QPU Interrupt Enable bits (1 bit per QPU in system config) Set bit to allow QPU to generate an interrupt.	R/W	0

Pipeline Interrupt Control

These are the interrupts from the V3D pipeline hardware. The interrupt bits are all latched and must be cleared via the V3DINTCTL register. INT_OUTOMEM is essentially level triggered, so the underlying condition must be cleared before the interrupt bit can be cleared.

Table 75: V3D_INTCTL Register Description

Synopsis	***Interrupt Control***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:4	–	Reserved – write zeros, read as don't care		
3	INT_SPILLUSE	Binner Used Overspill Memory interrupt status Set when the binner starts using the (valid) overspill memory buffer. Write 1 to clear.	R/W	0
2	INT_OUTOMEM	Binner Out of Memory interrupt status Set while the binner needs more memory to complete. Write 1 to clear.	R/W	0
1	INT_FLDONE	Binning Mode Flush Done interrupt status Set when binning is complete with all tile lists flushed to memory. Write 1 to clear.	R/W	0
0	INT_FRDONE	Render Mode Frame Done interrupt status Set when all tiles of the frame have been written to memory. Write 1 to clear.	R/W	0

Table 76: V3D_INTENA Register Description

Synopsis	***Interrupt Enables***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:4	–	Reserved – write zeros, read as don't care		
3	EI_SPILLUSE	Binner Used Overspill Memory interrupt enable Set when the INT_SPILLUSE interrupt is set. Write 1 to enable the interrupt.	R/W	0
2	EI_OUTOMEM	Binner Out of Memory interrupt enable Set when the INT_OUTOMEM interrupt is set. Write 1 to enable the interrupt.	R/W	0
1	EI_FLDONE	Binning Mode Flush Done interrupt enable Set when the INT_FLDONE interrupt is set. Write 1 to enable the interrupt.	R/W	0
0	EI_FRDONE	Render Mode Frame Done interrupt enable Set when the INT_FRDONE interrupt is set. Write 1 to enable the interrupt.	R/W	0

Table 77: V3D_INTDIS Register Description

Synopsis	*Interrupt Disables*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:4	–	Reserved – write zeros, read as don't care		
3	DI_SPILLUSE	Binner Used Overspill Memory interrupt disable Set when the INT_SPILLUSE interrupt is set. Write 1 to disable the interrupt.	R/W	0
2	DI_OUTOMEM	Binner Out of Memory interrupt disable Set when the INT_OUTOMEM interrupt is set. Write 1 to disable the interrupt.	R/W	0
1	DI_FLDONE	Binning Mode Flush Done interrupt disable Set when the INT_FLDONE interrupt is set. Write 1 to disable the interrupt.	R/W	0
0	DI_FRDONE	Render Mode Frame Done interrupt disable Set when the INT_FRDONE interrupt is set. Write 1 to disable the interrupt.	R/W	0

V3D Miscellaneous Registers

Table 78: V3D_SCRATCH Register Description

Synopsis	*Scratch Register*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:0	SCRATCH	Scratch Register Read/Write registers for general purposes	R/W	0

V3D Identity Registers

These are the read-only identity registers, giving the version, revision and configuration parameters for the particular configuration of the V3D system. Reset values marked (*) vary with configuration – the values shown are for the reference configuration.

Table 79: V3D_IDENT0 Register Description

Synopsis	*V3D Identification 0 (V3D block identity)*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:24	TVER	V3D Technology Version Reads technology version = 2	R	2
23:0	IDSTR	V3D Id String Reads as "V3D"	R	"V3D"

Table 80: V3D_IDENT1 Register Description

Synopsis	*V3D Identification 1 (V3D Configuration A)*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:28	VPMSZ	VPM Memory Size (multiples of 1K, 0 => 16K)	R	12*
27:24	HDRT	HDR Support (0 = not supported, 1 = supported)	R	1*
23:16	NSEM	Number of Semaphores	R	16*
15:12	TUPS	Number of TMUs per Slice	R	2*
11:8	QUPS	Number of QPUs per Slice	R	4*
7:4	NSLC	Number of Slices	R	3*
3:0	REV	V3D Revision	R	1*

Table 81: V3D_IDENT2 Register Description

Synopsis	*V3D Identification 1 (V3D Configuration B)*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:12	–	Reserved – write zeros, read as don't care		
11:8	TLBDB	Tile Buffer Double-buffer Mode Support 0 = not supported, 1 = supported	R	1*
7:4	TLBSZ	Tile Buffer Size 0=1/4, 1=1/2, 2=full size (32x32msm)	R	2*
3:0	VRISZ	VRI Memory Size 0=half size 1=full size	R	1*

Performance Counters

These are used for monitoring the performance of the V3D system. There are 16 counters available, each of which can be mapped to count one of up to 30 count sources. The Ids of the available count sources are given in Table 82.

Table 82: Sources for Performance Counters

Performance Counter Count Source IDs	
Count ID	***Count Description***
0	FEP Valid primitives that result in no rendered pixels, for all rendered tiles
1	FEP Valid primitives for all rendered tiles. (primitives may be counted in more than one tile)
2	FEP Early-Z/Near/Far clipped quads

Table 82: Sources for Performance Counters (Cont.)

Performance Counter Count Source IDs	
Count ID	***Count Description***
3	FEP Valid quads
4	TLB Quads with no pixels passing the stencil test
5	TLB Quads with no pixels passing the Z and stencil tests
6	TLB Quads with any pixels passing the Z and stencil tests
7	TLB Quads with all pixels having zero coverage
8	TLB Quads with any pixels having non-zero coverage
9	TLB Quads with valid pixels written to color buffer
10	PTB Primitives discarded by being outside the viewport
11	PTB Primitives that need clipping
12	PSE Primitives that are discarded because they are reversed
13	QPU Total idle clock cycles for all QPUs
14	QPU Total clock cycles for all QPUs doing vertex/coordinate shading
15	QPU Total clock cycles for all QPUs doing fragment shading
16	QPU Total clock cycles for all QPUs executing valid instructions
17	QPU Total clock cycles for all QPUs stalled waiting for TMUs
18	QPU Total clock cycles for all QPUs stalled waiting for Scoreboard
19	QPU Total clock cycles for all QPUs stalled waiting for Varyings
20	QPU Total instruction cache hits for all slices
21	QPU Total instruction cache misses for all slices
22	QPU Total uniforms cache hits for all slices
23	QPU Total uniforms cache misses for all slices
24	TMU Total texture quads processed
25	TMU Total texture cache misses (number of fetches from memory/L2cache)
26	VPM Total clock cycles VDW is stalled waiting for VPM access
27	VPM Total clock cycles VCD is stalled waiting for VPM access
28	L2C Total Level 2 cache hits
29	L2C Total Level 2 cache misses

Table 83: V3D_PCTRC Register Description

Synopsis	***Performance Counter Clear***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:16	–	Reserved – write zeros, read as don’t care		

Table 83: V3D_PCTRC Register Description (Cont.)

Synopsis	***Performance Counter Clear***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
15:0	CTCLR0-CTCLR15	Performance Counter Clear Bits (1 bit per Performance Counter in system config) Write '1' to clear the performance counter	W	0

Table 84: V3D_PCTRE Register Description

Synopsis	***Performance Counter Enables***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:16	–	Reserved – write zeros, read as don't care		
15:0	CTEN0-CTEN15	Performance Counter Enable Bits 1 = performance counter enabled to count, 0 = counter disabled	R/W	0

Table 85: V3D_PCTRn Register Description

Synopsis	***Performance Counts (n = 0-15)***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:0	PCTR	Performance Count Count value.	R/W	0

Table 86: V3D_PCTRSn Register Description

Synopsis	***Performance Counter Mapping (n = 0-15)***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:5	–	Reserved – write zeros, read as don't care		
4:0	PCTRS	Performance Counter Device Id The 'device' that the counter is setup to count	R/W	0

Error and Diagnostic Registers

These registers contain internal hardware error bits and provide other diagnostic information for use in interpreting errors.

Table 87: V3D_ERRSTAT Register Description

Synopsis	***Miscellaneous Error Signals (VPM, VDW, VCD, VCM, L2C)***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:16	–	Reserved – write zeros, read as don't care		
15	L2CARE	L2C AXI Receive Fifo Overrun error	R	0
14	VCMBE	VCM error (binner)	R	0
13	VCMRE	VCM error (renderer)	R	0
12	VCDI	VCD Idle	R	0
11	VCDE	VCD error - FIFO pointers out of sync	R	0
10	VDWE	VDW error - address overflows	R	0
9	VPMEAS	VPM error - allocated size error	R	0
8	VPMEFNA	VPM error - free non-allocated	R	0
7	VPMEWNA	VPM error - write non-allocated	R	0
6	VPMERNA	VPM error - read non-allocated	R	0
5	VPMERR	VPM error - read range	R	0
4	VPMEWR	VPM error - write range	R	0
3	VPAERRGL	VPM Allocator error - renderer request greater than limit	R	0
2	VPAEBRGL	VPM Allocator error - binner request greater than limit	R	0
1	VPAERGS	VPM Allocator error - request too big	R	0
0	VPAEABB	VPM Allocator error - allocating base while busy	R	0

Table 88: V3D_DBGE Register Description

Synopsis	***PSE Error Signals***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:21	–	Reserved – write zeros, read as don't care		
20	IPD2_FPDUSED	error_ipd2_fpdused	R	0
19	IPD2_VALID	error_ipd2_valid	R	0
18	MULIP2	error_mulip2	R	0
17	MULIP1	error_mulip1	R	0
16	MULIP0	error_mulip0	R	0
15:3	–	Reserved – write zeros, read as don't care		
2	VR1_B	Error b reading VPM	R	0
1	VR1_A	error a reading VPM	R	0

Table 88: V3D_DBGE Register Description (Cont.)

Synopsis	*PSE Error Signals*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
0	–	Reserved – write zeros, read as don't care		

Table 89: V3D_FDBGO Register Description

Synopsis	*FEP Overrun Error Signals*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:18	–	Reserved – write zeros, read as don't care		
17	EZREQ_FIFO_ORUN		R	0
16	–	Reserved – write zeros, read as don't care		
15	EZVAL_FIFO_ORUN		R	0
14	DEPTHO_ORUN		R	0
13	DEPTHO_FIFO_ORUN		R	0
12	REFXY_FIFO_ORUN		R	0
11	ZCOEFF_FIFO_FULL	Not an error	R	0
10	XYRELW_FIFO_ORUN		R	0
9:8	–	Reserved – write zeros, read as don't care		
7	XYRELO_FIFO_ORUN		R	0
6	FIXZ_ORUN		R	0
5	XYFO_FIFO_ORUN		R	0
4	QBSZ_FIFO_ORUN		R	0
3	QBFR_FIFO_ORUN		R	0
2	XYRELZ_FIFO_FULL	Not an error	R	0
1	WCOEFF_FIFO_FULL	Not an error	R	0
0	–	Reserved – write zeros, read as don't care		

Table 90: V3D_FDBGB Register Description

Synopsis	*FEP Interface Ready and Stall Signals, plus FEP Busy Signals*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:29	–	Reserved – write zeros, read as don't care		
28	XYFO_FIFO_OP_READY		R	0
27	QXYF_FIFO_OP_READY		R	0
26	RAST_BUSY		R	0
25	EZ_XY_READY		R	0
24	–	Reserved – write zeros, read as don't care		

Table 90: V3D_FDBGB Register Description (Cont.)

Synopsis	*FEP Interface Ready and Stall Signals, plus FEP Busy Signals*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
23	EZ_DATA_READY		R	0
22:8	–	Reserved – write zeros, read as don't care		
7	ZRWPE_READY		R	0
6	ZRWPE_STALL		R	0
5:3	EDGES_CTRLID		R	0
2	EDGES_ISCTRL		R	0
1	EDGES_READY		R	0
0	EDGES_STALL		R	0

Table 91: V3D_FDBGR Register Description

Synopsis	*FEP Internal Ready Signals*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31	–	Reserved – write zeros, read as don't care		
30	FIXZ_READY		R	0
29	–	Reserved – write zeros, read as don't care		
28	RECIPW_READY		R	0
27	INTERPRW_READY		R	0
26:25	–	Reserved – write zeros, read as don't care		
24	INTERPZ_READY		R	0
23	XYRELZ_FIFO_LAST		R	0
22	XYRELZ_FIFO_READY		R	0
21	XYNRM_LAST		R	0
20	XYNRM_READY		R	0
19	EZLIM_READY		R	0
18	DEPTHO_READY		R	0
17	RAST_LAST		R	0
16	RAST_READY		R	0
15	–	Reserved – write zeros, read as don't care		
14	XYFO_FIFO_READY		R	0
13	ZO_FIFO_READY		R	0
12	–	Reserved – write zeros, read as don't care		
11	XYRELO_FIFO_READY		R	0
10:8	–	Reserved – write zeros, read as don't care		
7	WCOEFF_FIFO_READY		R	0

Table 91: V3D_FDBGR Register Description (Cont.)

Synopsis	*FEP Internal Ready Signals*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
6	XYRELW_FIFO_READY		R	0
5	ZCOEFF_FIFO_READY		R	0
4	REFXY_FIFO_READY		R	0
3	DEPTHO_FIFO_READY		R	0
2	EZVAL_FIFO_READY		R	0
1	EZREQ_FIFO_READY		R	0
0	QXYF_FIFO_READY		R	0

Table 92: V3D_FDBGS Register Description

Synopsis	*FEP Internal Stall Input Signals*			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:29	–	Reserved – write zeros, read as don't care		
28	ZO_FIFO_IP_STALL		R	0
27:26	–	Reserved – write zeros, read as don't care		
25	RECIPW_IP_STALL		R	0
24:23	–	Reserved – write zeros, read as don't care		
22	INTERPW_IP_STALL		R	0
21:19	–	Reserved – write zeros, read as don't care		
18	XYRELZ_FIFO_IP_STALL		R	0
17	INTERPZ_IP_STALL		R	0
16	DEPTHO_FIFO_IP_STALL		R	0
15	EZLIM_IP_STALL		R	0
14	XYNRM_IP_STALL		R	0
13	EZREQ_FIFO_OP_VALID		R	0
12	QXYF_FIFO_OP_VALID		R	0
11	QXYF_FIFO_OP_LAST		R	0
10	QXYF_FIFO_OP1_DUMMY		R	0
9	QXYF_FIFO_OP1_LAST		R	0
8	QXYF_FIFO_OP1_VALID		R	0
7	EZTEST_ANYQVALID		R	0
6	EZTEST_ANYQF		R	0
5	EZTEST_QREADY		R	0
4	EZTEST_VLF_OKNOVALID		R	0
3	EZTEST_STALL		R	0

Table 92: V3D_FDBGS Register Description (Cont.)

Synopsis	***FEP Internal Stall Input Signals***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
2	EZTEST_IP_VLFSTALL		R	0
1	EZTEST_IP_PRSTALL		R	0
0	EZTEST_IP_QSTALL		R	0

Table 93: V3D_BXCF Register Description

Synopsis	***Binner Debug***			
Bit(s)	***Field Name***	***Description***	***Type***	***Reset***
31:2	–	Reserved – write zeros, read as don’t care		
1	CLIPDISA	Disable Clipping	R/W	0
0	FWDDISA	Disable Forwarding in State Cache	R/W	0

Section 11: Texture Memory Formats

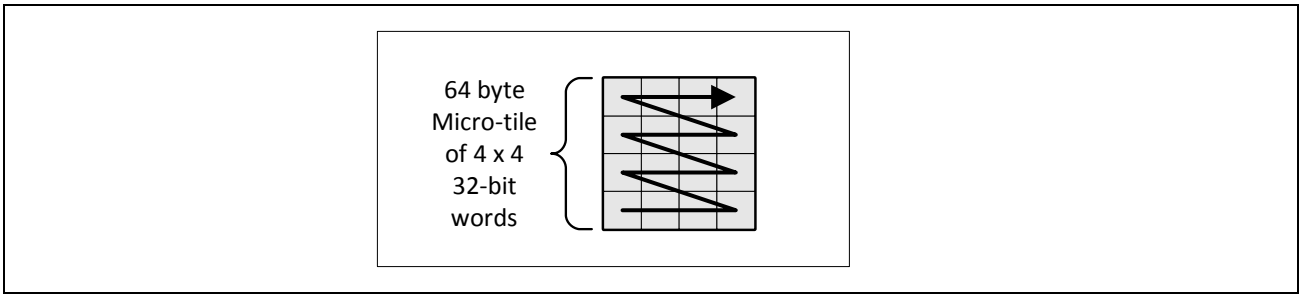
The TMUs require most types of textures to be arranged in memory in T-format or LT-format. The TLB also supports storing of any tile buffer to a frame buffer in T-format or LT-format. The following section describes the memory arrangement of T-format and LT-format images.

All images are assumed to have a bottom-left origin, that is, the first data in memory will be the bottom-left pixel.

Micro-tiles

T-format and LT-format images are composed of a sequence of micro-tiles. A micro-tile is a rectangular image block, with a fixed size of 512-bits (64 bytes). The internal organization of a micro-tile depends on the pixel format of an image. Images of 64bpp, 32bpp, 16bpp, 8bpp, 4bpp or 1bpp are organized as 2 × 4, 4 × 4, 8 × 4, 8 × 8, 16 × 8 or 32 × 16 blocks of pixels respectively, in simple raster order.

Figure 13: Typical Micro-tile Organization



Compressed ETC1 textures are organized as a 2 × 4 block of 64-bit blocks, each of which is in itself a 4 × 4 image block.

Texture Format (T-format)

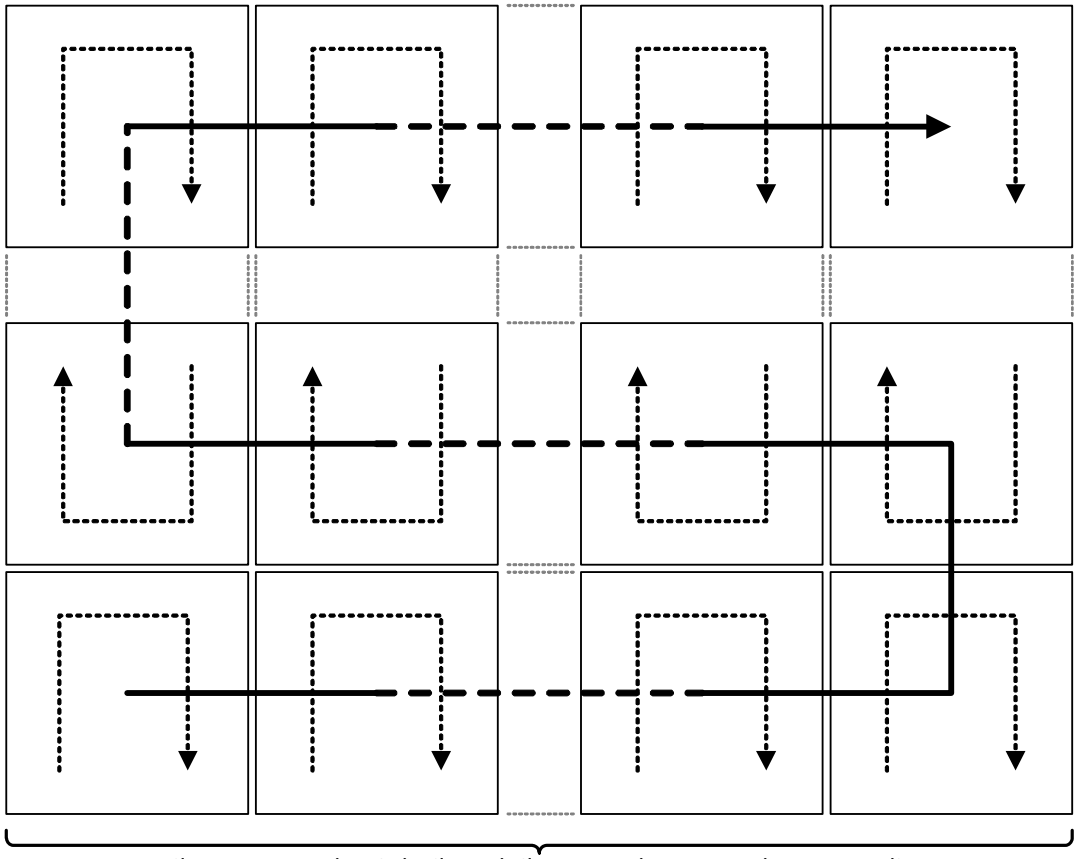
T-format is based around 4 Kbyte tiles of 2D image data, formed from 1 Kbyte sub-tiles of data. As an example, for 32bpp pixel mode, 1 Kbyte equates to a block of 16 × 16 image pixels. A 4 Kbyte tile therefore contains 32 × 32 pixels.

Figure 14 shows how sub-tiles are ordered to form 4K. The arrows in each case indicate the address order of the sub-tiles for odd lines of image tiles (larger dotted arrow), and even lines (smaller solid arrow).

The entire image is then formed from these sub-tiles. The 4K tiles are ordered left to right for odd rows, and right to left for even rows. For odd and even rows, the 1K tile order also changes as shown.

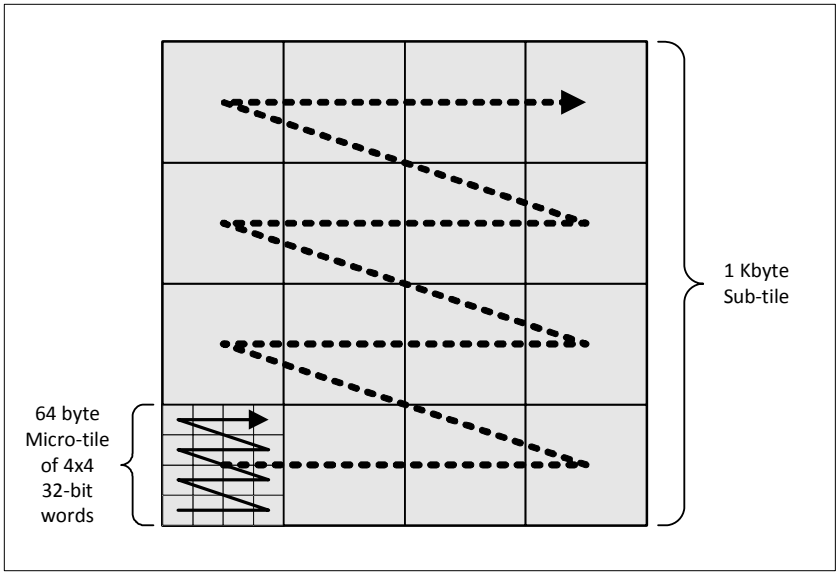
Note that when storing image data in T-format, the data must be padded to be a multiple of 4k tiles in both width and height.

Figure 14: T-Format 4K Tile and 1K Sub-tile Memory Order



Each sub-tile is a 1Kbyte block of pixel data, which is in turn arranged as a sequence of micro-tiles. Each micro-tile is mapped to consecutive addresses. The micro-tiles themselves are arranged in simple raster scan order within the sub-tile.

Figure 15: T-Format Micro-tile Address Order in a 1K Sub-tile



Linear-tile Format (LT-Format)

Linear-tile format is typically used for small textures that are smaller than a full T-format 4K tile, to avoid wasting memory in padding the image out to be a multiple of tiles in size. This format is also micro-tile based but simply stores micro-tiles in a standard raster order.

Appendix A: Errata List

Table 94: Errata List Explanation

Number	*Issue*	*Affects*	*Description*
HW-2116	PTB state counters do not wrap around safely	BCM2835, BCM21553	Rendering errors occur if PTB state counters overflow. Workaround is revert to software detect of near state counter wraparound (by counting primitive draw lists) and use a 'manual' flush_all_state. This should have negligible performance impact for any sensible content.
HW-2253	User shaders cannot use all of the VPM	BCM2835, BCM21553	There are only enough bits in the V3DVPMBASE register and VPM setup registers to address the first 64 rows of the VPM from user shaders. No workaround exists. In later core versions V3DVPMBASE is expanded to 6 bits, and setting bit[29] in the setup registers expands the address field by two bits, shifting other fields up accordingly.
HW-2619	TMU 16-bit blend pass through fails for negative powers of 2	BCM2835	TMU erroneously discards the sign bit of sample components which are precise negative powers of two. Workaround is to preprocess texture data to slightly adjust these values.
HW-2645	16-bit trilinear blend does not work for INF + negative values	BCM2835	A large positive value is clamped with a zeroed mantissa. Interpolation with a previous negative sample falsely sets the sign bit and the result ends up negative. Workaround is to preprocess texture data to slightly adjust these values.
HW-2726	PTB does not handle zero-size points	BCM2835, BCM21553	Software workarounds are feasible, with minimal performance penalty: • If the point size comes from the GL point size, discard points lists if the point size is < 0.125. • If the point size is per vertex from the coordinate/vertex shader, modify only the coordinate shader to clamp the point size to be ≥ 0.125.
HW-2753	Texture child images fail with width or height of 2048	BCM2835, BCM21553	No workaround exists.
HW-2796	Cannot sbwait in first two instructions	BCM2835, BCM21553	Under certain circumstance pixels are drawn to the previous tile if an `sbwait` signal is used on the first or second instruction in a fragment shader. Workaround is to add nops as necessary; this will add one cycle to the shortest of shaders.

Table 94: Errata List Explanation (Cont.)

Number	*Issue*	*Affects*	*Description*
HW-2806	Z test cannot be too late	BCM2835, BCM21553	Multisample flags get written back to wrong quad when the Z test result write collides with the arrival of new thread data. Workaround is to allow extra instructions between the Z write and the instruction which contains the `thread` signal.
HW-2885	Lockup on write to coverage pipe after primitive with non-zero varying count	BCM2835, BCM21553	When the output of the FEP is directed to the coverage pipe, and the VRI module had previously been used with a non-zero number of varyings, a lockup may occur. Workaround is to send at least one quad to a fragment shader with zero varyings, so that in the last slice used the VRI internal number of varyings gets set to zero.
HW-2898	Address generation error for NPOT raster textures	BCM2835, BCM21553	For raster textures the address offset calculation assumes power-of-2 widths when issuing memory lookups. This is not true for mipmap level 0. Workaround is to pad NPOT raster textures appropriately.
HW-2905	Early Z error on full tile load in multisample mode	BCM2835, BCM21553	Workaround is to disable early Z in this case.
HW-2924	Stencil config changes applied per batch rather than per quad	BCM2835, BCM21553	The QPU only applies forward/reverse stencil config changes to the TLB per batch of four quads, rather than per quad. Workaround is to modify the fragment shader to use separate Z writes for forward and reverse facing quads, using manipulation of the ms_mask. If the last valid quad is reverse facing, do the reverse facing quads first and the forwards facing quads last (and vice versa).

Appendix B: Base Addresses

Table 95 lists the address where the V3D registers are located in each product.

Table 95: Base Addresses for V3D Registers

Device	*Address*
BCM2835	0x7ec00000
BCM21553	0x08950000
BCM21654	0x3c00b000

Connecting everything®

BROADCOM CORPORATION
5300 California Avenue
Irvine, CA 92617

Phone: 949-926-5000
Fax: 949-926-5203
E-mail: info@broadcom.com
Web: www.broadcom.com

VideoCoreIV-AG100-R September 16, 2013