

 AMD	ORIGINATE	EDIT DATE	DOCUMENT-VER. NUM.	PAGE
	[CREATEDATE]	[SAVEDATE \@ "d-MMM-yy" *]	0.32	[PAGE] of [NUMPAGES]
Author: Brian Emberling, Jeremy Furtak, Andrew Pomianowski, Guennadi Riguer, Andrew Brown				
ISSUED TO:		COPY NO.		
SQ Thread Trace 2.3 User's Guide Rev 0.32 – Last Edit: [SAVEDATE \@ "d-MMM-yy" * MERGEFORMAT]				
THIS DOCUMENT CONTAINS INFORMATION THAT COULD BE SUBSTANTIALLY DETRIMENTAL TO THE INTEREST OF AMD THROUGH UNLICENSED USE OR UNAUTHORIZED DISCLOSURE.				
1. Preserve this document's integrity: [SYMBOL 222 \f "Symbol" \s 8 \h] Do not reproduce any portions of it. [SYMBOL 222 \f "Symbol" \s 8 \h] Do not separate any pages from this cover. 2. This document is issued to you alone. Do not transfer it to or share it with another person, even within your organization. 3. Store this document in a locked cabinet accessible only by authorized users. Do not leave it unattended. 4. When you no longer need this document, return it to AMD. Please do not discard it.				
"Copyright 2012, Advanced Micro Devices, Inc. ("AMD"). All rights reserved. This work contains confidential, proprietary to the reader information and trade secrets of AMD. No part of this document may be used, reproduced, or transmitted in any form or by any means without the prior written permission of AMD." AMD, the AMD Arrow Logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. PCIe is a registered trademark of PCI-SIG. HDMI is a trademark of HDMI Licensing, LLC. AMD (NYSE: AMD) is a semiconductor design innovator leading the next era of vivid digital experiences with its ground-breaking AMD Fusion Accelerated Processing Units (APU). AMD's graphics and computing technologies power a variety of devices including PCs, game consoles and the powerful computers that drive the Internet and businesses. For more information, visit [HYPERLINK "http://www.amd.com"].				

 AMD	ORIGINATE [CREATEDATE MM/dd/yyyy hh:mm AM/PM]	EDIT DATE [SAVEDATE \@ "d-MMM-w" * MM/dd/yyyy hh:mm AM/PM]	DOCUMENT-VER. NUM. 0.32	PAGE [PAGE] of [NUMPAGES]
--	---	--	----------------------------	----------------------------------

TABLE OF CONTENTS

[TOC \o "1-6" \t "Figcaptiontop,3, Figcaptionbottom,3"]

 AMD	ORIGINATE [CREATEDATE]	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *]	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	-----------------------------	---	---------------------------	----------------------------------

1 Scope of this document

This document describes, from a *software* perspective, the thread tracing capabilities of Southern Islands-based GPU architectures, configuration of thread tracing settings, and the types of visualization that the thread trace data will enable.

1.1 Thread Trace Versions

Thread trace began with the “Evergreen” family of GPUs and has evolved with each generation. It had a complete overhaul starting with Southern Islands.

- TT 1.0 : Evergreen (“8xx”)
- TT 1.1 : Northern Islands (“9xx”)
- TT 2.0 : Southern Islands (“GFX6”)
- TT 2.1 : Sea Islands (“GFX7”)
- TT 2.2 : Volcanic Islands (“GFX8”)
- TT 2.3 : GFX9

1.1.1 Changes in GFX9

- Increase PC from 40 to 48 bits in INST_PC token
- Add an ERROR bit in INST_PC (bit 15) to indicate that this is a trap being taken due to a memory violation, xnack-error or illegal instruction.
- Add a new INST token: FATAL_HALT indicating that this wave was halted due to a fatal error (ECC error, illegal instruction, xnack-error or memory violation while in the trap handler).
- Fixed the “timing skew” of different tokens from the same wave so that now they all appear at the correct time.
- Remove SQTT_MODE.PRIV bit
- Add SQTT_MODE.TC_PERF_EN which controls whether or not thread-trace traffic to the L2 cache is counted in TCC’s perfcounters or not.
- Add a UTC_ERROR bit in the SQTT_STATUS register indicating that thread-trace has encountered a page-translation (or permission) error and cannot continue. S/W must reset this bit to re-enable TTrace.
- Next Generation Graphics Pipeline (NGG) eliminates hardware LS and ES shader stages.
 - LS is always merged into HS; ES is always merged into GS
- Added INST tokens DIDT_STALL_START and _END to indicate when the shader is stalled by the power manager.
- S_BARRIER: Prior to GFX9, barriers showed up as an IMMED token when the barrier was complete. In GFX9, a barrier shows up as an instruction (OTHER_MSG) when the instruction is issued, and an IMMED token is generated when the barrier is complete.

1.1.2 Known Bugs

TT 2.1

- Export Grant tokens for VS shaders have the wrong type: position grants show up as parameter.
- Various clock gating issues resulting in dropped tokens interrupt problems. Disable clock gating to work around this while using threadtrace.
- Thread trace ‘capture mode all’ does not capture perf counter events after thrad-trace-off event
- “PS1” tokens are never generated, even for waves from “packer 1”

TT 2.2

- If an instruction is issued immediately after a S_SETVSKIP, it might not be logged by thread trace. This is timing dependent, and applies only to INTERNAL (IMMED) instructions.
- “PS1” tokens are never generated, even for waves from “packer 1”. *Fixed in GFX9/TT2.3.*

TT 2.3

AMD1044_00048683

 AMD	ORIGINATE [CREATEDATE]	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *]	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	-----------------------------	---	---------------------------	----------------------------------

- GFX9 thread trace must not be turned off (mode=off) or buffer-reset while thread-trace is active: must wait for thread-trace idle before either of the actions otherwise a hang could occur (or at least writes to memory may go to the wrong address)
- When SQ generates thread trace tokens for misc instructions, SQG writes it to its internal history buffer for output. The purpose of the history buffer is to delay these instructions by a few cycles so they are in order with respect to scalar alu tokens. There is a bug where the history buffer is not drained completely and stale data remains in it. When a new bunch of waves start, this stale data is incorrectly associated with the new wave and incorrectly gets sent to the thread trace buffer. The end result is the thread trace buffer might contain extra incorrect INST tokens with TYPE JUMP/NEXT/OTHER-MSG. This bug was fixed in Vega11 but exists in all variants before Vega11 including all previous TT versions.

2 Architectural Overview

A brief and simplified overview of the architecture of Southern Islands-based GPUs is presented to provide context for the remainder of the document. Details of the discussion and diagram will be limited to those that are relevant to understanding thread tracing on SI.

The Vector Scalar Processor (VSP) consists of 4 multiply-accumulate (MACC) units. Each MACC unit is capable of executing a single precision floating point instruction per clock cycle for a single thread.

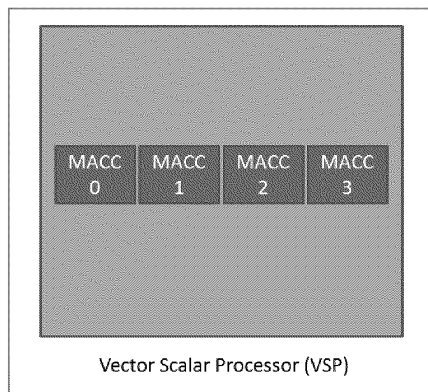


Figure [STYLEREF 1 \s].[SEQ Figure * ARABIC \s 1]: The SI Vector Scalar Processor has 4 multiply-accumulate units

A Single Instruction Multiple Data (SIMD) unit consists of 4 VSP units ([REF _Ref258925367 \h]). Therefore, each SIMD unit is capable of performing 16 single precision floating point operations per clock cycle.

AMD1044_00048684

 AMD	ORIGINATE [CREATEDATE]	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *]	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	-----------------------------	---	---------------------------	----------------------------------

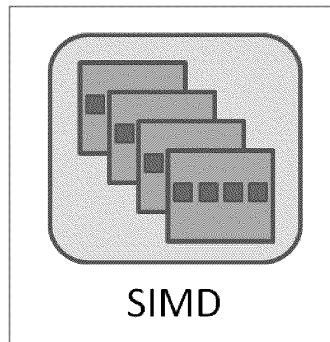


Figure [STYLeref 1 \s].[SEQ Figure * ARABIC \s 1]: A Single Instruction Multiple Data (SIMD) unit consists of 4 Vector Scalar Processors (VSP)

Two SIMD units are combined to form a SIMD Pair (SP) ([REF_Ref258925439 \h]). Two SIMD Pairs are combined with a Shader Sequencer (SQ) and a Local Data Store (LDS) to form a Compute Unit (CU) ([REF_Ref258925498 \h]). Each Compute Unit contains 64 single precision multiply-accumulate units.

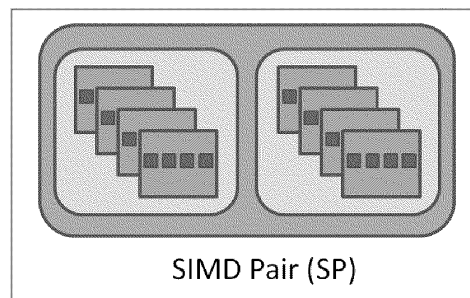


Figure [STYLeref 1 \s].[SEQ Figure * ARABIC \s 1]: A SIMD Pair (SP) consists of two SIMD units

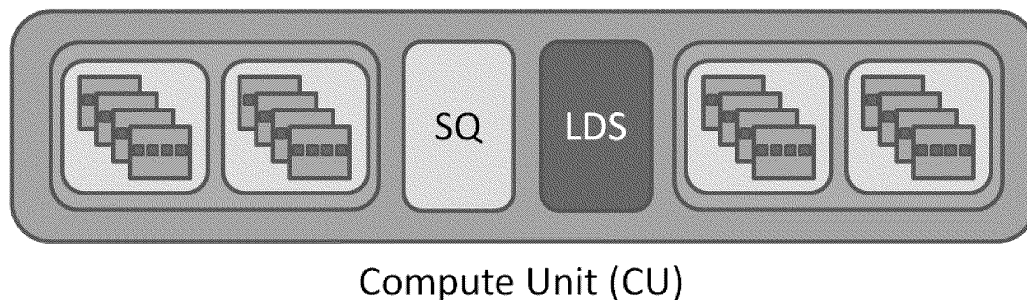


Figure [STYLeref 1 \s].[SEQ Figure * ARABIC \s 1]: The Compute Unit (CU) groups two SIMD Pair (SP) units with a Shader Sequencer (SQ) and a Local Data Store (LDS)

On AMD graphics processors, a collection of threads that executes a common instruction is called a *wavefront*. A wavefront is assigned to a SIMD unit upon creation and remains assigned to that same SIMD unit for its entire lifetime. The Local Data Store (LDS) is high speed memory that can be shared by the threads in a wavefront. Latency can effectively be hidden by assigning multiple wavefronts to a Compute Unit.

AMD1044_00048685

 AMD	ORIGINATE [CREATEDATE	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	---------------------------	---	-------------------------------	----------------------------------

The SQ unit maintains instruction buffers for the wavefronts assigned to a CU, and is responsible for coordinating execution within a wavefront instruction stream as well as across multiple wavefronts. The SQ unit is also responsible for implementing the thread trace functionality in SI hardware.

Each group of 4 SQ units is serviced by a Sequencer Cache (SQC) unit. The SQC unit consists of two caches: an instruction cache, and a constant (read-only) data cache. The SQC caches are backed by the texture L2 cache. (The SQC module is relevant to the performance counters that will be described later in the document. However, for simplicity, the SQC unit will not be shown in subsequent architectural hierarchy diagrams.)

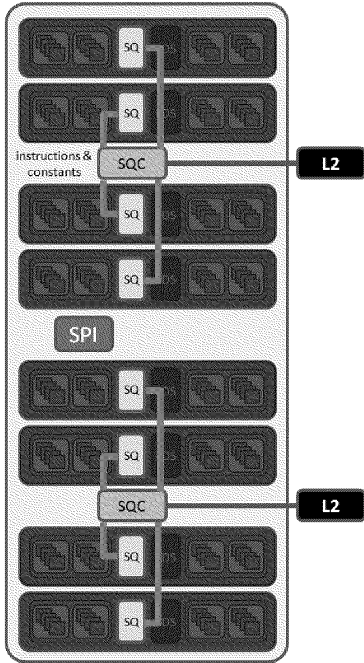


Figure [STYLEREF 1 \s].[SEQ Figure * ARABIC \s 1]: Each group of 4 SQ units is serviced by a Sequencer Cache (SQC) unit

The next levels of the hierarchy are the Shader Array (SH) and the Shader Engine (SE). A Shader Array contains a variable number of Compute Units, and a Shader Engine (SE) contains 1 or 2 Shader Arrays served by a single Shader Processor Input (SPI) unit ([REF _Ref258925566 \h]).

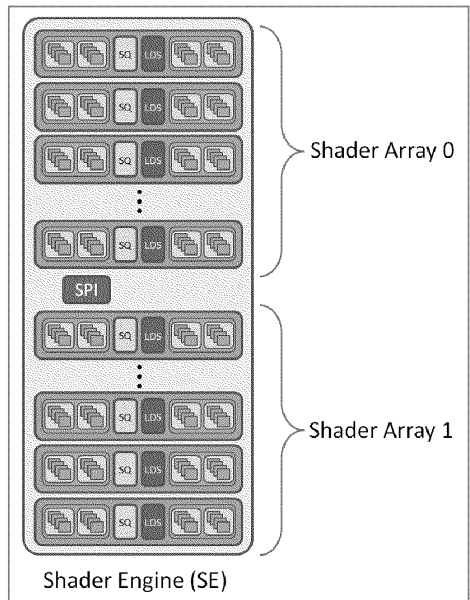


Figure [STYLeref 1 \s].[SEQ Figure * ARABIC \s 1]: A Shader Engine (SE) contains 1 or 2 Shader Arrays (SH) with a single Shader Processor Input (SPI) unit)

Some GPUs combine two (2) Shader Engines ([REF _Ref258925669 \h]) for increased processing capabilities.

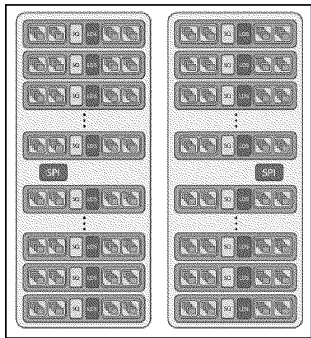


Figure [STYLeref 1 \s].[SEQ Figure * ARABIC \s 1]: The architecture supports multiple identical Shader Engine (SE) units within a single chip

3 Thread tracing

GPUs with thread tracing functionality can export a stream of data to GPU-addressable memory (either local or host system remote) for offline performance analysis. Thread tracing information is exported in the form of *tokens*. Visualization and monitoring of these recorded tokens, in conjunction with tools that incorporate specific knowledge of the GPU architecture, will provide developers with insight into observed performance and enable tuning to the strength of a particular GPU.

AMD1044_00048687

 AMD	ORIGINATE [CREATEDATE	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	---------------------------	---	---------------------------	----------------------------------

The thread trace functionality in Southern Islands and beyond (TT2.0) is an extension of the thread tracing capabilities introduced in the Evergreen (R8xx) family (TT1.0), with modifications to accommodate for changes that have evolved in more recent AMD GPU architectures.

The design of a thread tracing system must balance a number of often conflicting constraints – including chip area, trace detail, hardware performance (for both tracing and non-tracing scenarios), and memory bandwidth to support generated trace logs. The TT2.0 architecture manages these constraints by defining two levels of detail for thread tracing tokens generated at the Compute Unit level of the hierarchy. The higher level of thread tracing detail consists of instruction level tokens, and will be generated by a single *target* Compute Unit. The lower level of detail will be generated by all Compute Units, and we will refer to tokens in this lower level of detail as *global* events.

There are 3 types of thread tracing tokens (see [REF_Ref258925743 \h]):

- *Global* tokens: These tokens describe thread execution and are generated by *all* Compute Units in a Shader Engine (SE).
- *Target* tokens: These tokens describe thread execution and are generated by *a single* Compute Unit within a Shader Engine (SE).
- *Reference* tokens: These tokens provide context information for *global* and *target* tokens generated in the Compute Units. Examples of reference token information are elapsed clock cycles (to allow for the generation of precise timing information), register writes, and stalls induced by the thread tracing system itself.

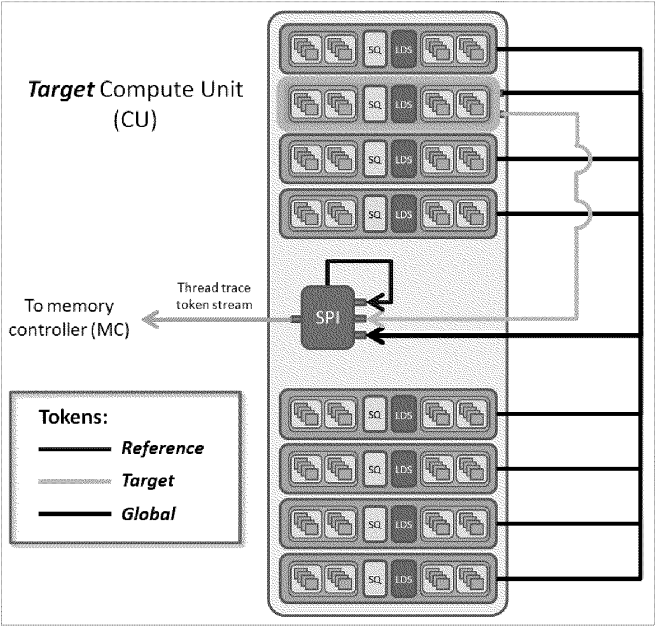


Figure [STYLeref 1 \s].[SEQ Figure * ARABIC \s 1]: Tokens from a single *target* Compute Unit (CU) are combined with *global* tokens from all CUs and *reference* tokens in the thread trace token stream

It is expected that combining *reference* token information and high-level thread trace data from *global* tokens with more detailed instruction-level token information for a single *target* CU will provide adequate insight into hardware execution for effective performance tuning.

3.1 Thread tracing message formats

AMD1044_00048688

 AMD	ORIGINATE [CREATEDATE	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	---------------------------	---	---------------------------	-------------------------------------

In the remainder of this document, the use of the string SQ_THREAD_TRACE will be replaced with {SQT} in an effort to shorten identifier names and simplify formatting. For example, the identifier SQ_THREAD_TRACE_TOKEN_WAVE_PS_ALLOC will be written as {SQT}_TOKEN_WAVE_PS_ALLOC. Header files and register names are expected to contain the full, unabbreviated identifier. (Note that there are some identifiers which use the abbreviation TTRACE instead of the full words THREAD_TRACE. Those identifiers reflect the naming used in the actual register specification, and are not the result of an abbreviation introduced by this document.)

Each thread trace token is written into the user-specified buffer using one of several *message* formats (or message structures), as shown in [REF_Ref258925875 \h]. Note that there is a one-to-one correspondence between message formats and thread tracing tokens, *except* for the case of the {SQT}_MSG_EVENT and {SQT}_MSG_WAVE messages. The {SQT}_MSG_EVENT message format is used to indicate the {SQT}_TOKEN_EVENT, {SQT}_TOKEN_EVENT_CS, and {SQT}_TOKEN_EVENT_GFX1 events. The {SQT}_MSG_WAVE message is used to indicate {SQT}_TOKEN_WAVE_END, {SQT}_TOKEN_WAVE_PS_ALLOC, and {SQT}_TOKEN_WAVE_VS_ALLOC events. The TOKEN_TYPE field of the message structures is used to indicate the token type for all message types.

Message (Token ID)	Description (Token Type)	Field [Bits]	Field Description
{SQT}_MSG_TIME (TT2.0 only) ({SQT}_TOKEN_TIME)	Time has passed (REFERENCE)	TOKEN_TYPE [3:0]	Token type (see [REF_Ref272917213 \h * MERGEFORMAT])
		TIME_DELTA [13:4]	Time increment since last msg
		TIME_RESET [14]	0: Add time to delta counter 1: Time counter needs to be reset relative to next TIMESTAMP token
		PACKET_LOST [15]	0: Normal operation 1: Packet was lost
{SQT}_MSG_TIMESTAMP ({SQT}_TOKEN_TIMESTAMP)	Resynchronize the timestamp (REFERENCE)	TOKEN_TYPE [3:0]	Token type (see [REF_Ref272917213 \h * MERGEFORMAT])
		TIME [63:16]	Timestamp
{SQT}_MSG_REG ({SQT}_TOKEN_REG)	A register write has occurred (REFERENCE)	TOKEN_TYPE [3:0]	Token type (see [REF_Ref272917213 \h * MERGEFORMAT])
		TIME_DELTA [4]	Time increment since last msg
		DISPATCHER [9:5] (TT2.0 only)	Dispatcher ID
		PIPE_ID [6:5] (TT2.1+)	Compute/gfx pipeline ID
		ME_ID [8:7] (TT2.1+)	Micro-engine ID

AMD1044_00048689

 AMD	ORIGINATE [CREATEDATE	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	---------------------------	---	-------------------------------	----------------------------------

		REG_DROPPED_PREV[9] (TT2.1+) Indicates whether a prior sequence of REG tokens may be missing from thread trace. Tokens for register reads of REG_TYPE=OTHER registers are always dropped to avoid causing polling for the core idle condition to fail. Normally, the only other registers that may be dropped are those bound for the SQ or SQC that cannot be stalled because they are not from the CPG. The hardware does not check all the masks to see if a REG token would have been written to the thread trace buffer before recording the dropped-data event, so this bit may represent a sequence of tokens that would not have been written to the trace buffer. In a 2-SH-per-SE configuration, we may also drop a REG token to keep within our instantaneous bandwidth limit of 512-bits per clock. Finally, if TOKEN_MASK.REG_DROP_ON_STALL is set, REG tokens may also be dropped when the thread trace output fifo to memory has exceeded its highwater mark	
		REG_TYPE [12:10]	Register type (see [REF _Ref258931955 \h * MERGEFORMAT])
		REG_PRIV [14]	1 for privileged register access
		REG_OP [15]	Register bus op ([REF _Ref258931978 \h * MERGEFORMAT])
		REG_ADDR [31:16]	Register address in DWORDs
		DATA [63:32]	Register data
{SQTT}_MSG_REG_CS (TT2.1+) ({SQTT}_TOKEN_REG_CS) ({SQTT}_TOKEN_REG_CSPRIV)	({SQTT}_TOKEN_REG_CS) A register write has occurred for a COMPUTE_* register or a compute shader pipeline event has been initiated. (REFERENCE) ({SQTT}_TOKEN_REG_CSPRIV) There has been a compute pipeline private data, state or threadgroup update. (REFERENCE)	TOKEN_TYPE [3:0]	Token type (see [REF _Ref272917213 \h * MERGEFORMAT])
		TIME_DELTA [4]	Time increment since last msg
		PIPE_ID [6:5]	Compute/gfx pipeline ID
		ME_ID [8:7]	Micro-engine ID
		REG_ADDR [15:9]	REG_ADDR represents the LSB's of the offset within the COMPUTE_* group of registers (those seen over the CSDATA bus to SPI). The REG_ADDR field is actually missing the 2 LSB's of the address (since they're all dword-aligned addresses), so it represents register-address-offset [8:2] REG_CS: These offsets go from COMPUTE_DISPATCH_INITIATOR (offset=0) to COMPUTE_USER_DATA15 (offset=0x13c) REG_CSPRIV: The offsets go from CSPRIV_CONNECT to CSPRIV_THREAD_TRACE_EVENT
		REG_DATA [47:16]	Register data
{SQTT}_MSG_WAVE_START ({SQTT}_TOKEN_WAVE_START)	A wavefront has started (GLOBAL)	TOKEN_TYPE [3:0]	Token type (see [REF _Ref272917213 \h * MERGEFORMAT])

AMD1044_00048690

 AMD	ORIGINATE [CREATEDATE	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	---------------------------	---	---------------------------	----------------------------------

	WR_EXEC is (TARGET) (TT2.2+)		MERGEFORMAT])
		TIME_DELTA [4]	Time increment since last msg
		SH_ID [5]	Shader Array ID
		CU_ID [9:6]	Compute Unit ID
		WAVE_ID [13:10]	Wavefront ID
		SIMD_ID [15:14]	SIMD ID
		DISPATCHER [20:16]	Dispatcher description (See Section [REF Ref270411597 \r \h])
		VS_NO_ALLOC_OR_GROUPED [21]	<u>VS waves</u> : indicates that no separate allocation is expected <u>Other wave types</u> : indicates wave is part of a thread group
{SQTT}_MSG_WAVE (({SQTT}_TOKEN_WAVE_PS_ALLOC) (({SQTT}_TOKEN_WAVE_VS_ALLOC) (({SQTT}_TOKEN_WAVE_ALLOC) (({SQTT}_TOKEN_WAVE_END)	- Output space has been allocated for PS color/Z (TOKEN_WAVE_PS_ALLOC) (TARGET) (TT2.0 only) - Output space has been allocated for VS position (TOKEN_WAVE_VS_ALLOC) (TARGET) (TT2.0 only) - Output space has been allocated for either PS color/Z or VS position depending on wave type (TOKEN_WAVE_ALLOC) (TARGET) (TT2.1+) - Wavefront completion (TOKEN_WAVE_END) (GLOBAL)	COUNT [28:22]	Num active threads (0 to 64) (TT2.2+) : if count > 64, indicates this is actually a WR_EXEC, and COUNT = {5'b11000, atc, first_wave_in_tg }
		TG_ID [31:29]	Thread group ID
		TOKEN_TYPE [3:0]	Token type (see [REF _Ref272917213 \h * MERGEFORMAT])
		TIME_DELTA [4]	Time increment since last msg
		SH_ID [5]	Shader Array ID
		CU_ID [9:6]	Compute Unit ID
		WAVE_ID [13:10]	Wavefront ID
		SIMD_ID [15:14]	SIMD ID
{SQTT}_MSG_EVENT (({SQTT}_TOKEN_EVENT) (({SQTT}_TOKEN_EVENT_CS) (({SQTT}_TOKEN_EVENT_GFX1)	An event has reached the top of the shader stage (({SQTT}_TOKEN_EVENT) (REFERENCE) An event has reached the top of a compute shader stage. (({SQTT}_TOKEN_EVENT_CS) (REFERENCE) An event has reached the top of a shader stage for the second GFX pipe. (({SQTT}_TOKEN_EVENT_GFX1) (REFERENCE)	TOKEN_TYPE [3:0]	Token type (see [REF _Ref272917213 \h * MERGEFORMAT])
		TIME_DELTA [4]	Time increment since last msg
		SH_ID [5]	Shader Array ID
		STAGE [8:6]	Graphics rings: Hardware stage this event has reached (PS, VS, GS, ES, HS, LS, CS, PS1[REF PS1_Note \h]) Compute rings: Dispatcher number
		EVENT_TYPE [15:10]	Type of event type (See VGT_EVENT_INITIATOR register description and [REF APPENDIX_A \h * MERGEFORMAT] for details.)
{SQTT}_MSG_INST (({SQTT}_TOKEN_INST)	The shader has executed an instruction (TARGET)	TOKEN_TYPE [3:0]	Token type (see [REF _Ref272917213 \h * MERGEFORMAT])
		TIME_DELTA [4]	Time increment since last msg

AMD1044_00048691

 AMD	ORIGINATE [CREATEDATE	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	---------------------------	---	-------------------------------	----------------------------------

		WAVE_ID [8:5]	Wavefront ID
		SIMD_ID [10:9]	SIMD ID
		SIZE [11] (TT2.0-2.1 only)	Instr size: 0 = 32 bits, 1 = 64 bits
		INST_TYPE [15:12] (TT2.0-2.1 only)	Instruction type ([REF _Ref258932028 \h * MERGEFORMAT])
		INST_TYPE [15:11] (TT2.2+)	Instruction type ([REF _Ref258932028 \h * MERGEFORMAT])
{SQTT}_MSG_INST_PC ({SQTT}_TOKEN_INST_PC)	The shader has explicitly written the program counter (PC) value (TARGET)	TOKEN_TYPE [3:0]	Token type (see [REF _Ref272917213 \h * MERGEFORMAT])
		TIME_DELTA [4]	Time increment since last msg
		WAVE_ID [8:5]	Wavefront ID
		SIMD_ID [10:9]	SIMD ID
		ERROR[15]	Indicates that this PC is a jump to a trap handler due to an error: memviol, illegal instruction, or xnack-error.
		PC [63:16]	Program Counter
{SQTT}_MSG_INST_USERDATA ({SQTT}_TOKEN_INST_USERDATA)	The shader has written some user data into the thread trace buffer (GLOBAL)	TOKEN_TYPE [3:0]	Token type (see [REF _Ref272917213 \h * MERGEFORMAT])
		TIME_DELTA [4]	Time increment since last msg
		SH_ID [5]	Shader Array ID
		CU_ID [9:6]	Compute Unit ID
		WAVE_ID [13:10]	Wavefront ID
		SIMD_ID [15:14]	SIMD ID
{SQTT}_MSG_ISSUE ({SQTT}_TOKEN_ISSUE)	Instruction scheduling information: an instruction was issued and/or stalled due to arbitration priority (TARGET)	DATA [47:16]	User data
		TOKEN_TYPE [3:0]	Token type (see [REF _Ref272917213 \h * MERGEFORMAT])
		TIME_DELTA [4]	Time increment since last msg
		SIMD_ID [6:5]	SIMD ID
		INST0 [9:8]	Instruction state ([REF _Ref258932054 \h * MERGEFORMAT])
		INST1 [11:10]	
		INST2 [13:12]	
		INST3 [15:14]	
		INST4 [17:16]	
		INST5 [19:18]	
		INST6 [21:20]	
		INST7 [23:22]	
		INST8 [25:24]	
		INST9 [27:26]	
{SQTT}_MSG_PERF ({SQTT}_TOKEN_PERF)	Performance counter increment has been received from a Compute Unit (GLOBAL)	TOKEN_TYPE [3:0]	Token type (see [REF _Ref272917213 \h * MERGEFORMAT])
		TIME_DELTA [4]	Time increment since last msg
		SH_ID [5]	Shader Array ID
		CU_ID [9:6]	Compute Unit ID
		CNTR_BANK [11:10]	0: Bank 0 (counters 0 to 3)

AMD1044_00048692

			1: Bank 1 (counters 4 to 7) 2: Bank 2 (counters 8 to 11) 3: Bank 3 (counters 12 to 15)
		CNTR0 [24:12]	Counter increment for counter CNTR_BANK*4
		CNTR1 [37:25]	Counter increment for counter CNTR_BANK*4 + 1
		CNTR2 [50:38]	Counter increment for counter CNTR_BANK*4 + 2
		CNTR3 [63:51]	Counter increment for counter CNTR_BANK*4 + 3
{SQTT}_MSG_MISC (TT2.0 only) ({SQTT}_TOKEN_MISC)	Miscellaneous thread tracing token (REFERENCE)	TOKEN_TYPE [3:0]	Token type (see [REF _Ref272917213 \h * MERGEFORMAT])
		TIME_DELTA [4]	Time increment since last msg
		SH_ID [5]	Shader array ID (if applicable)
		MISC_TOKEN_TYPE [7:6]	Miscellaneous token subtype (see [REF SQ_THREAD_TRACE_MISC_T OKEN_TYPE_Table \h * MERGEFORMAT])
		RESERVED [15:8]	Padding to 16 bits
{SQTT}_MSG_MISC (TT2.1+) ({SQTT}_TOKEN_MISC)	Miscellaneous thread tracing token (REFERENCE)	TOKEN_TYPE [3:0]	Token type (see [REF _Ref272917213 \h * MERGEFORMAT])
		TIME_DELTA [11:4]	Time increment since last msg
		SH_ID [12]	Shader array ID (if applicable)
		MISC_TOKEN_TYPE [15:13]	Miscellaneous token subtype (see [REF _Ref316456710 \h * MERGEFORMAT])

Table [STYLEREF 2 \s],[SEQ Table * ARABIC \s 2]: Thread tracing message formats

*The GPU's Scan Converter (SC) block has two parallel and independent back ends that feed pixel shader wavefronts to a Shader Array. The PS1 shader type exists to distinguish between events from the two SC back ends. PS1 is a pixel shader from the second packer. *In GFX6 chips with 2 Shader arrays, PS was SH0, PS1 was SH1.*

TOKEN_TYPE	Value	Description	Message encoding
{SQTT}_TOKEN_TIME (TT2.0 only)	0	Time has passed	{SQTT}_MSG_TIME (TT2.0 only)
{SQTT}_TOKEN_TIMESTAMP	1	Resync the timestamp	{SQTT}_MSG_TIMESTAMP
{SQTT}_TOKEN_REG	2	A register write has occurred	{SQTT}_MSG_REG
{SQTT}_TOKEN_WAVE_START	3	A wavefront has started	{SQTT}_MSG_WAVE_START
{SQTT}_TOKEN_WAVE_PS_ALLOC (TT2.0 only)	4	Output space has been allocated for color/Z	{SQTT}_MSG_WAVE
{SQTT}_TOKEN_WAVE_VS_ALLOC (TT2.0 only)	5	Output space has been allocated for vertex position	{SQTT}_MSG_WAVE
{SQTT}_TOKEN_WAVE_ALLOC (TT2.1+)	4	Output space has been allocated for vertex position or color/Z	{SQTT}_MSG_WAVE
{SQTT}_TOKEN_WAVE_END	6	Wavefront completion	{SQTT}_MSG_WAVE
{SQTT}_TOKEN_EVENT	7	An event has reached the top of a shader stage. In-order with WAVE_START.	{SQTT}_MSG_EVENT
{SQTT}_TOKEN_EVENT_CS	8	An event has reached the top of a compute shader stage. In-order with WAVE_START	{SQTT}_MSG_EVENT

AMD1044_00048693

 AMD	ORIGINATE [CREATEDATE]	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *]	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	-----------------------------	---	---------------------------	----------------------------------

{SQTT}_TOKEN_EVENT_GFX1	9	An event has reached the top of a shader stage for the second GFX pipe. In-order with WAVE_START.	{SQTT}_MSG_EVENT
{SQTT}_TOKEN_INST	10	The shader has executed an instruction	{SQTT}_MSG_INST
{SQTT}_TOKEN_INST_PC	11	The shader has explicitly written the PC value	{SQTT}_MSG_INST_PC
{SQTT}_TOKEN_INST_USERDATA	12	The shader has written user data into the thread trace buffer	{SQTT}_MSG_INST_USERDATA
{SQTT}_TOKEN_ISSUE	13	Provides information about instruction scheduling	{SQTT}_MSG_ISSUE
{SQTT}_TOKEN_PERF	14	The performance counter delta has been updated	{SQTT}_MSG_PERF
{SQTT}_TOKEN_MISC (TT2.0 only)	15	A miscellaneous event has been sent	{SQTT}_MSG_MISC (TT2.0 only)
{SQTT}_TOKEN_MISC (TT2.1+)	0	A miscellaneous event has been sent	{SQTT}_MSG_MISC (TT2.1+)
{SQTT}_TOKEN_REG_CS (TT2.1+)	15	A compute state update packet has been received by the SPI.	{SQTT}_MSG_REG_CS
{SQTT}_TOKEN_REG_CS_PRIV (TT2.1+)	5	There has been a compute pipeline private data, state or threadgroup update.	{SQTT}_MSG_REG_CS

Table [STYLeref 2 \s],[SEQ Table * ARABIC \s 2]: Thread tracing token identifiers used for the message TOKEN_TYPE fields

Identifier	Value	Description
{SQTT}_REG_TYPE_EVENT	0	Event initiator
{SQTT}_REG_TYPE_DRAW	1	Draw initiator
{SQTT}_REG_TYPE_DISPATCH	2	Dispatch initiator
{SQTT}_REG_TYPE_USERDATA	3	User data written via register-writes. Hardware does not interpret the values written, and simply logs them as reg-writes in the ttrace buffer.
{SQTT}_REG_TYPE_MARKER	4	Thread trace marker event (DEPRECATED)
{SQTT}_REG_TYPE_GFXDEC	5	GFXDEC register (8-state)
{SQTT}_REG_TYPE_SHDEC	6	SHDEC register (many state)
{SQTT}_REG_TYPE_OTHER	7	Other registers This bit must be set to zero to prevent a hang condition. <i>When the RLC decides to shut off clocks, it will send out a few registers writes ("other") which thread-trace tries to write out to memory, but the RLC shuts off the clocks before the memory controller has acknowledged those writes.</i>

Table [STYLeref 2 \s],[SEQ Table * ARABIC \s 2]: Register types for the {SQTT}_MSG_REG message

*See the {SQTT}_USERDATA register in [REF_Ref258932378 \h * MERGEFORMAT]

Identifier	Value	Description
{SQTT}_REG_OP_READ	0	Register read
{SQTT}_REG_OP_WRITE	1	Register write

Table [STYLeref 2 \s],[SEQ Table * ARABIC \s 2]: Register operations for the {SQTT}_MSG_REG message

Instruction Type ID	#	Description
{SQTT}_INST_TYPE_SMEM (TT2.0-2.1 only)	0	Scalar memory fetch has been issued
{SQTT}_INST_TYPE_SALU (TT2.0-2.1 only)	1	Scalar ALU instruction has been issued

AMD1044_00048694

 AMD	ORIGINATE [CREATEDATE	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [
--	---------------------------	---	---------------------------	-----------------------

{SQTT}_INST_TYPE_VMEM_RA (TT2.0 only)	2	Vector memory read instruction has been issued
{SQTT}_INST_TYPE_VMEM_WA (TT2.0 only)	3	Vector memory write instruction has been issued
{SQTT}_INST_TYPE_VMEM_WD (TT2.0 only)	4	Vector memory write data has been sent
{SQTT}_INST_TYPE_VMEM_RD (TT2.1+)	2	Vector memory read instruction has been issued
{SQTT}_INST_TYPE_VMEM_WR (TT2.1+)	3	Vector memory write instruction has been issued
{SQTT}_INST_TYPE_FLAT_WR (TT2.1+)	4	Flat write instruction has been issued.
{SQTT}_INST_TYPE_FLAT_RD (TT2.1+)	14	Flat read instruction has been issued.
{SQTT}_INST_TYPE_VALU (TT2.0-2.1 only)	5	Vector ALU instruction has been issued
{SQTT}_INST_TYPE_LDS	6	Local Data Store (LDS) instruction has been issued
{SQTT}_INST_TYPE_PC	7	A new PC was written (see INST_PC token for new PC value). This token may arrive after WAVE_END.
{SQTT}_INST_TYPE_EXPREQ_GDS	8	GDS instruction has been requested
{SQTT}_INST_TYPE_EXPREQ_GFX	9	Export (pixel/parameter/position/gds) instruction has been requested
{SQTT}_INST_TYPE_EXPGNT_PAR_COL	10	Export (pixel or parameter) has been granted
{SQTT}_INST_TYPE_EXPGNT_POS_GDS	11	Export (position or GDS) has been granted
{SQTT}_INST_TYPE_JUMP	12	Branch jump taken
{SQTT}_INST_TYPE_NEXT	13	Branch jumped skipped
{SQTT}_INST_TYPE_GEOM (TT2.0 only)	14	Geometry instruction has been issued
{SQTT}_INST_TYPE_OTHER_MSG	15	Other type of instruction/message. Includes ENDPGM, INTERRUPT and geometry emit/cuts (the latter is in TT2.1+ only, as in TT2.0 there is a special GEOM instruction type). See [REF_Ref273104370 \h * MERGEFORMAT] for a list of instructions.
{SQTT}_INST_TYPE_SMEM_RD (TT2.2+)	0	Scalar memory data read has been issued
{SQTT}_INST_TYPE_SMEM_WR (TT2.2+)	16	Scalar memory data write has been issued
{SQTT}_INST_TYPE_SALU_32 (TT2.2+)	1	Scalar ALU instruction has been issued (32-bit data)
{SQTT}_INST_TYPE_VALU_32 (TT2.2+)	5	Vector ALU instruction has been issued (32-bit data)
{SQTT}_INST_TYPE_SALU_64 (TT2.2+)	17	Scalar ALU instruction has been issued (64-bit data)
{SQTT}_INST_TYPE_VALU_64 (TT2.2+)	18	Vector ALU instruction has been issued (64-bit data)
{SQTT}_INST_TYPE_SMEM_RD_REPLAY (TT2.2+)	19	"REPLAY" indicates that this instruction is being executed for the 2 nd or more time. The first execution was aborted due to an ATC XNACK. The "_REPLAY" instructions are otherwise identical to their non-replay versions.
{SQTT}_INST_TYPE_SMEM_WR_REPLAY (TT2.2+)	20	
{SQTT}_INST_TYPE_VMEM_RD_REPLAY (TT2.1+)	21	
{SQTT}_INST_TYPE_VMEM_WR_REPLAY (TT2.1+)	22	
{SQTT}_INST_TYPE_FLAT_WR_REPLAY (TT2.1+)	23	
{SQTT}_INST_TYPE_FLAT_RD_REPLAY (TT2.1+)	24	
{SQTT}_INST_TYPE_FATAL_HALT (TT2.3+)	25	FATAL_HALT indicates that the wave has halted due to a fatal error: illegal instruction, xnack-error, xnack while not in allow-replay mode, ECC error.
{SQTT}_INST_TYPE_DIDT_STALL_START	26	DIDT (power) monitor has started/ended stalling the SQ from issuing VALU instructions to reduce power. (this token can occasionally be silently dropped by hardware with no indication).
{SQTT}_INST_TYPE_DIDT_STALL_END (TT2.3+)	27	

Table [STYLEREf 2 \s].[SEQ Table * ARABIC \s 2]: Instruction types for the INST_TYPE field of the {SQTT}_MSG_INST message

Instruction State ID	Value	Description
{SQTT}_ISSUE_NULL	0	No change in instruction state
{SQTT}_ISSUE_STALL	1	Begin stall for non-immediate instruction. If stalling continues for more than 1 cycle, NULL or no token is used to continue the stall rather than repeating the stall token. The stall will eventually be

AMD1044_00048695

 AMD	ORIGINATE [CREATEDATE	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	---------------------------	---	---------------------------	----------------------------------

		ended by an INST or IMMED token.
{SQTT}_ISSUE_INST	2	Issue non-immediate instruction. An INST token will follow.
{SQTT}_ISSUE_IMMED	3	Issue instruction immediately. No INST is expected, either because it is a miscellaneous instruction (nop/wait/set_prio) or because it is skipped using the vector instruction skip (VSKIP)

Table [STYLeref 2 \s],[SEQ Table * ARABIC \s 2]: States for the INST0...INST9 fields of the {SQTT}_MSG_ISSUE message

Miscellaneous Token Type ID	Value	Description
{SQTT}_MISC_TOKEN_SURF_SYNC	0	Surface sync has been received
{SQTT}_MISC_TOKEN_TTRACE_STALL_BEGIN	1	Thread trace has begun stall
{SQTT}_MISC_TOKEN_TTRACE_STALL_END	2	Thread trace stall has ended

Table [STYLeref 2 \s],[SEQ Table * ARABIC \s 2]: Token subtypes for the miscellaneous thread trace token ({SQTT}_TOKEN_MISC) for TT2.0

Miscellaneous Token Type ID	Value	Description
{SQTT}_MISC_TOKEN_TIME	0	A nop just to increment the time counter (replaces the TT2.0 TIME token).
{SQTT}_MISC_TOKEN_TIME_RESET	1	Indicates that a TIMESTAMP token will be sent soon and that the timestamp for this token and subsequent tokens prior to the next TIMESTAMP token should be timestamped relative to that future TIMESTAMP token. Issued when 1024 or more clocks go by with no activity. Always followed by TIMESTAMP.
{SQTT}_MISC_TOKEN_PACKET_LOST	2	Reports that data has been lost since the previous token.
{SQTT}_MISC_TOKEN_SURF_SYNC	3	Surface sync has been received
{SQTT}_MISC_TOKEN_TTRACE_STALL_BEGIN	4	Thread trace has begun stall
{SQTT}_MISC_TOKEN_TTRACE_STALL_END	5	Thread trace stall has ended
{SQTT}_MISC_TOKEN_SAVE_CONTEXT {2.2+}	6	Context save has been initiated
{SQTT}_MISC_TOKEN_SHOOT_DOWN {2.2+}	7	TLB shutdown started

Table [STYLeref 2 \s],[SEQ Table * ARABIC \s 2]: Token subtypes for the miscellaneous thread trace token ({SQTT}_TOKEN_MISC) for TT2.1+

Instructions	
S_NOP	S_SLEEP
S_SENDMSG	S_SETPRIO
S_SENDMSGHALT	S_TRAP
S_BARRIER	S_ICACHE_INV
S_WAITCNT	S_DCACHE_INV
S_ENDPGM	S_INCPERFLVL
S_SETHALT	S_DECPERFLVL
S_SETVSKIP	S_TTRACEDATA

Table [STYLeref 2 \s],[SEQ Table * ARABIC \s 2]: Miscellaneous instructions described by {SQTT}_INST_TYPE_OTHER_MSG and {SQTT}_ISSUE_MASK_STALLED_AND_IMMED

AMD1044_00048696

 AMD	ORIGINATE [CREATEDATE	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	---------------------------	---	---------------------------	----------------------------------

3.1.1 Time data for thread tracing messages

All thread tracing messages (except the {SQTT}_MSG_TIME and {SQTT}_MSG_TIMESTAMP messages) contain a single bit *TIME_DELTA* field, which is used to indicate the elapsed time since the previous message. The *TIME_DELTA* field indicates whether the message token occurred during the same clock as the previous message token (*TIME_DELTA* = 0), or whether the token occurred 4 cycles after the previous token (*TIME_DELTA* = 1). If a token is about to be written, and the time since the last token exceeds 4 cycles, then a special {SQTT}_MSG_TIME message will be inserted into the buffer. Tokens with multiple *TIME_DELTA* bits can encode longer time deltas, such as the TT2.0 {SQTT}_MSG_TIME message and the TT2.1 {SQTT}_MSG_MISC msg. Since each time increment represents 4 clocks, the time field can represent relative elapsed times of up to $(2^{10}-1) * 4 = 4092$ cycles for TT2.0 and $(2^8-1) * 4 = 1020$ cycles for TT2.1+.

For longer periods of idle time, there is a {SQTT}_MSG_TIMESTAMP message. The {SQTT}_MSG_TIMESTAMP message is a 64-bit message with a 48 bit *TIME* field. The {SQTT}_MSG_TIMESTAMP message is added to the current thread trace buffer when thread tracing begins. For thread tracing modes that use multiple buffers, the {SQTT}_MSG_TIMESTAMP message will be placed at the beginning of *each* buffer. (More information regarding the use of multiple buffers is found in Section [REF _Ref259700814 r\h].)

Note: the MSG_TIMESTAMP reports time in units of cycles, not in units of 4-clocks like {SQTT}_MSG_MISC.

3.1.2 Inst and Issue Tokens – Multicycle Instructions

ISSUE tokens are generated when an instruction is selected for execution by the arbiter. INST tokens are generated on the last cycle of execution of a multi-cycle instruction.

VALU instructions use these two tokens to indicate how many cycles it took to issue the instruction. LDS, FLAT and VMEM instructions on the other hand use these two tokens to indicate how many cycles were spent reading data out of VGPRs to initiate the instruction (address and write-data), not the number of cycles to send the command (which is 4 for buffers, 8 for image read/write and 12 for sample).

Some instructions take more than one quad-cycle to issue. These instructions are (but not always):

- VMEM : if more than one address component (or a 64-bit component), MIMG with sampler or 256b T#
- FLAT : if 64-bit address, or a write (address and data)
- VALU : 64-bit operations, transcendental ops.
- LDS : indexed operations if more than 32 bits of address or data per thread

Thread trace reports multicycle instructions on using the ISSUE and INST tokens. The ISSUE token reports the first cycle, and the INST token the last cycle. For example, a texture instruction which takes 16 clocks to issue would be reported as:

```
ISSUE:    INST  NULL  NULL  NULL    -- sometimes may be STALLs instead of NULLs
INST:     -    -    -    -    VMEMRD
```

3.1.3 Exports

“Export” refers to the process of moving data from VGPRs to the output buffer. For Pixel shaders this occurs for color and depth data, and for Vertex shaders for position and parameter data. This path is also used for sending data to GDS.

Unlike other instructions, exports occur in two parts and are reported separately in threadtrace.

When the instruction is issued, this is really a request for the export to occur and not the export itself.

Thread trace logs this as an ISSUE (INST) and as an INST (EXPREQ_GFX or EXPREQ_GDS).

All PS and VS shader exports use the GFX type for their normal exports, and GDS has its own type.

AMD1044_00048697

 AMD	ORIGINATE [CREATEDATE]	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *]	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	-----------------------------	---	---------------------------	-------------------------------------

An export request can be for multiple DWORDs per thread.

Sometime later, the export request is granted and this is what actually reads the data from the VGPRs and transfers it to the output buffer. This is indicated in threadtrace by an INST (EXPGNT_PAR_COL or _POS_GDS) token.

In hardware, there is an EXPGNT for each 4 clocks worth of export activity and they can be in any order, however thread-trace only reports the final EXPGNT for a given EXPREQ.

For Pixel shaders, EXPGNT_PAR_COL means this is a color or depth export, and EXPGNT_POS_GDS means it is a GDS export.

For Vertex shaders, EXPGNT_PAR_COL means this is a Parameter export, and EXPGNT_POS_GDS means it is either a Position export or a GDS operation.

3.1.4 Context Switching

Thread trace attempts to capture as much of the context switching activity as possible. The CP communicates with the SR block (save/restore context) via the GRBM interface, so the register writes to initiate a context switch will be captured as normal.

There are no special tokens associated with context restore – a restored wave looks identical to a new wave and it is not possible to determine either when a wave has completed its context restore, nor when the SPI has started generating new waves vs. restored waves.

The context-save process consists of a few steps that are visible in thread-trace:

1. Register writes to the SR block over GRBM to begin the process.
2. SPI broadcast a SAVE_CONTEXT command once to all waves. SQ_TT_TOKEN_MISC.save_context
3. Each wave responds with either “save-me” (not visible). or “program-end” (visible as normal)
4. SPI then sends “WR_EXEC” (memory address) to each wave which asked to be saved (via “save-me”). This is visible in thread-trace, but appears as a WAVE_START command.
WR_EXEC can be differentiated from WAVE_START from an illegal value in the COUNT field. If the COUNT > 64, then it’s a WR_EXEC and the 7-bit count field holds: {5'b11000, atc, first_wave_in_tg }
5. Waves save their state and then end as normal. (WAVE_END – no difference from normal wave-end).

Modified Tokens for Context Switch:

The “new wave” token can also represent WREXEC and NEW_WAVE_RESTORE. They are differentiate by using illegal values of the “count” field, as:

WrExec = COUNT[6:0] = {5'b11000, atc, first_wave_in_tg }
NewWaveRestore = COUNT[6:0] = {5'b11001, atc, first_wave_in_tg }

3.1.5 Dispatcher Description

The DISPATCHER field in the {SQTT}_MSG_REG and {SQTT}_MSG_WAVE_START messages provides additional information on the source of the events that resulted in the generation of these messages. DISPATCHER is a 5-bit field, with the most significant bit (MSB) indicating whether the REG/WAVE_START originated from graphics or a compute workload. The interpretation of the remaining 4 least significant bits (LSBs) is dependent on the value of the most significant bit. (See [REF_Ref270420030 \h].)

For graphics workloads, a single bit is used to indicate the source graphics ring (on variants with 2 graphics rings), and the remaining 3 bits give the wave type. For compute tasks, the 4 least significant bits indicate the dispatcher ID. For example, for the “Tahiti” and “Verde” Southern Islands variants, the graphics dispatchers have ID numbers between 0 and 7 (inclusive). The graphics compute dispatcher has an ID of 6 and the second PS packer has an ID of 7, while the CS dispatchers corresponding to ring 1 and ring 2 compute have ID values of 0 and 1 (respectively).

```
struct DISPATCHER
{
    union
    {
        struct
        {
```

AMD1044_00048698

 AMD	ORIGINATE [CREATEDATE	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	---------------------------	---	---------------------------	----------------------------------

```

        int wave_type : 3;      // PS, VS, GS, ES, HS, LS, CS, PS1
        int ring_id   : 1;
    } gfx_dispatcher;          // use when type == 1

    struct
    {
        int id : 4;
    } compute_dispatcher;      // use when type == 0
};
int type : 1;                  // 0 - compute , 1 = graphics
};

```

Figure [STYLeref 3 \s].1: Notional C++ description of DISPATCHER field contents

3.1.6 Tracing events

The [REF SQ_TTRACE_MSG_EVENT \h] event in Table 3.1.1 occurs for *all* events. See the register description for VGT_EVENT_INITIATOR and [REF APPENDIX_A \h] for a list of the event types.

In SI, CI and VI, events will only be captured between THREAD_TRACE_START and _STOP, regardless of Capture-mode. In PI and beyond, events will be captured without requiring THREAD_TRACE_START.

3.1.7 Lost Packets

The thread trace hardware has a buffer in memory used to hide the latency of requests to memory. If this buffer fills up, thread trace will attempt to stall the processes that are generating thread trace data. However, if the buffer is found to be full, some tokens will be lost. If this happens, a token is inserted in the thread trace buffer to indicate that tokens have been lost and that the performance analysis tool should proceed with caution because some tokens may be missing. In TT2.0, lost packets are reported via a TIME packet with the PACKET_LOST bit set. In TT2.1, lost packets are noted with a MISC(PACKET_LOST) token.

3.1.8 Tracing performance counters

Performance counter tokens are *global* in the TT2.0 implementation – they can be generated from all Compute Units. Examples of specific occurrences that can be captured by performance counter tokens are instruction issues, stalled cycles, and cache hits. The set of events that can be counted in TT2.0 are described in Appendix B. These values are particularly interesting as they are returned separately for each CU, allowing very fine grain reporting of values such as the number of texture instructions in-flight.

The performance counter *mode* (see [REF _Ref318470989 \h]) controls how the counter values are modified as execution progresses. A subset of the standard performance counting modes is supported due to the distributed nature of the counters in the SQ module implementation.

In TT2.1 performance counter increments *always* saturate. In TT2.0, the 16-bit performance counter values could wrap around if SQ_PERFCOUNTR_CTRL.RATE is set to too big a value.

SQ_PERF_MODE	Value	Description
SQ_PERF_MODE_ACCUM	0	Accumulate increment values over enabled cycles. This is typically how perfcounters are used.
SQ_PERF_MODE_MAX	2	Store the largest increment received over cycles counted
SQ_PERF_MODE_DIRTY	3	Record whether a nonzero increment was ever received
SQ_PERF_MODE_CYCLES_SINCE_FIRST_EVENT	5	Count the cycles since a nonzero increment value
SQ_PERF_MODE_CYCLES_SINCE_LAST_EVENT	6	Count the cycles since a zero increment value

Table [STYLeref 3 \s].[SEQ Table * ARABIC 3]: Performance counter modes that are valid for thread tracing (values for SQ_PERFCOUNTER<#>_CTRL.PERF_MODE)

AMD1044_00048699

 AMD	ORIGINATE [CREATEDATE	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	---------------------------	---	---------------------------	-------------------------------------

3.2 User data

The {SQTT}_TOKEN_INST_USERDATA token is used to describe user-defined data written by a shader program into the thread tracing output buffer. The *user data* token is a global token, and therefore this instruction-level capability is available to all Compute Units. (The shader program *user data* feature should not be confused with the host-side *user data* that can be placed into the thread tracing stream via the {SQTT}_TOKEN_REG token.)

The S_TTRACEDATA instruction is used to emit user data into the thread trace buffer. The S_TTRACEDATA instruction writes 32 bits (from the memory location specified by the M0 register) to the thread trace buffer. See the Southern Islands Shader Programming Guide ("SI_Shader_Programming.docx) for more information on the S_TTRACEDATA instruction and the M0 register.

3.2.1 Conditional user data writes

The *user data* token is a global token, and therefore the *user data* thread trace token can be generated from all Compute Units. Shader programs can perform conditional writes of thread trace user data to reduce memory bandwidth consumption. One approach to accomplish this is to restrict user data writes to the *target* Compute Unit. The STATUS architectural register contains a single bit (TTRACE_CU_EN) that indicates whether or not the CU that is executing the current wavefront has *target* tracing enabled. (See the Southern Islands Shader Programming Guide for more information on the STATUS architectural register and the specific Southern Islands instructions. As discussed below, the {SQTT}_MASK register has a field called CU_SEL that, in conjunction with the SH_SEL field, allows the driver to indicate which Compute Unit is the target Compute Unit.) The S_GETREG_B32 instruction can be used to retrieve the contents of the STATUS architectural register, and the S_TTRACEDATA instruction execution can be conditional upon the value of the STATUS.TTRACE_CU_EN field.

3.3 Configuration and control of thread tracing functionality

The TT.20 functionality has been developed under the assumption that a single centralized software unit is responsible for configuring and managing thread trace functionality. A single set of hardware registers is used, and no attempt is made to prevent over-subscription of clients to thread tracing functionality.

Thread tracing has been designed to be exposed to debugging and profiling tools, and not to end user applications.

The registers for driver-level control of thread tracing are shown in [REF_Ref258932378 \h]. These registers are *single-copy global configuration* registers. That is, there is a single instance of these registers that is valid for the entire GPU. (This should not be confused with the fact that writes to some of these registers can be *broadcast* to different Shader Engines – see Section [REF_Ref259700814 \r \h]). The thread tracing registers are non-privileged – there are no ring or buffer restrictions on thread trace register access. Also, thread trace registers are not a per-graphics context registers (as they may have been in previous chip families).

(In general, *global configuration* registers for thread trace are intended to be written during GPU initialization only, but performance counting and thread trace registers are an exception.)

Register (Access)	Description	Field [Bits]	Field Description
{SQTT}_BASE (read/write)	Trace buffer base address	ADDR [31:0]	GPU address for trace buffer (in 4 KB units), i.e. address bits [43:12]
{SQTT}_BASE2 (read/write)	Trace buffer base address	ADDR [3:0]	GPU upper address for trace buffer (in 4 KB units), i.e. address bits [47:44]
{SQTT}_SIZE (read/write)	Trace buffer size	SIZE [21:0]	Size of trace buffer in GPU memory (in 4 KB units). A value of 0 disables

AMD1044_00048700

 AMD	ORIGINATE [CREATEDATE]	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *]	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	-----------------------------	---	---------------------------	----------------------------------

			tracing.
{SQTT}_MASK (read/write)	Shader engine SIMD selection mask. (Thread sampling will be reset when this register is written.)	CU_SEL [3:0]	Select <i>target</i> CU
		SH_SEL [5]	Select Shader Array (SH) that contains the <i>target</i> CU
		REG_STALL_EN [7] (TT2.1+)	Allow graphics pipe registers to be stalled when thread trace fifo reaches highwater mark.
		SIMD_EN [11:8]	Bitmask to enable or disable <i>target</i> tokens for different SIMDs
		VM_ID_MASK [13:12]	Select which VM IDs to capture (see [REF_Ref316396420 \h])
		SPI_STALL_EN[14] (TT2.1+)	Enable stalling of the SPI wave creation engine when the trace memory write Fifo have reached their highwater mark
		SQ_STALL_EN[15] (TT2.1+)	Enable stalling of the SQ instruction scheduler when the ttrace memory write Fifo have reached their highwater mark
{SQTT}_WPTR (read only)	Write pointer to trace buffer	WPTR [29:0]	Offset of write pointer in thread trace buffer, in units of 32 bytes
		READ_OFFSET [31:30]	Pointer to the first token in last completed buffer (units of 16-bits). Writing RESET_BUFFER in {SQTT}_CTRL will reset this to 0.
{SQTT}_STATUS (read only)	Thread tracing status information	FINISH_PENDING [2:0] (TT2.0 only)	1 bit per ring. Set when THREAD_TRACE_FINISH event has been received. Cleared when RESET_BUFFER field of {SQTT}_CTRL is set or THREAD_TRACE_FINISH event completes pipeline
		FINISH_PENDING [9:0] (TT2.1+)	Set when a THREAD_TRACE_FINISH event has been received. One bit per pipe. Cleared when SQ_THREAD_TRACE_CTRL.RESET_BUFFER is set or THREAD_TRACE_FINISH event completes the pipeline. LSB = ME1, pipe 0. MSB = GFX(ME0), max pipe.
		FINISH_DONE [18:16] (TT2.0 only)	1 bit per ring. Set when a THREAD_TRACE_FINISH event completes and all data has been written to the buffer. Cleared when {SQTT}_CTRL.RESET_BUFFER field is set or a new THREAD_TRACE_FINISH event is received.
		FINISH_DONE [25:16] (TT2.1+)	Set when a THREAD_TRACE_FINISH event has completed and all data has been written to the buffer. One bit per pipe. Cleared when SQ_THREAD_TRACE_CTRL.RESET_BUFFER is set or a new THREAD_TRACE_FINISH event is received. LSB = ME1, pipe 0. MSB

AMD1044_00048701

 AMD	ORIGINATE [CREATEDATE]	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *]	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	-----------------------------	---	---------------------------	----------------------------------

			= GFX(ME0), max pipe.
		UTC_ERR[28]	Set when thread trace receives a page translation error or permission error. This is a fatal thread-trace error. Once set to 1, thread trace is disabled until s/w writes this bit back to zero.
		NEW_BUF [29]	A new buffer has been provided to be used when the current buffer fills
		BUSY [30]	Flag indicating that tokens have been logged and are being written to memory. The BASE and SIZE registers should not be changed while this bit is set.
		FULL [31]	Flag indicating a complete buffer has been written. In wrap mode, this means the pointer has wrapped since the BASE field in {SQTT}_BUFFER was last written. Writing the BASE field of {SQTT}_BUFFER or writing the RESET_BUFFER field of {SQTT}_CTRL will clear this bit.
{SQTT}_CNTR (read only)	Counter for thread trace data generated since last reset	CNTR [31:0]	Saturating counter for the number of 32 byte blocks of data generated by thread trace (but not necessarily written)
{SQTT}_HIWATER	High water mark for stalling chip due to thread trace	HIWATER [2:0]	A value of 0..7 indicating what fraction (1/8, 2/8, ..., 1) of the thread trace fifo must be full before we trigger a stall. Failure to trigger a stall in time may result in lost thread trace packets.
{SQTT}_TOKEN_MASK (read/write)	Thread tracing token mask	TOKEN_MASK [15:0]	Mask indicating which trace token IDs will be included in the trace. Cannot mask out: timestamp, user_data
		REG_MASK [23:16]	Mask indicating which register types should be included in the trace
		REG_DROP_ON_STALL [24] (TT2.1+)	If set, registers are dropped when thread trace fifo reaches the highwater mark. This is provided to help avoid losing packets, at the expense of losing register data.
{SQTT}_TOKEN_MASK2 (TT2.1+) (read/write)	Thread tracing token mask	INST_MASK [15:0]	Mask indicating which trace instructions will be included in the trace. Removing INST_PC is a particularly good idea if you don't need it because tracing it can cause extra stalling.
{SQTT}_MODE (read/write)	Thread tracing mode selection	MASK_PS [2:0]	Subset sampling for PS/VS/GS/ES/HS/LS/CS waves (see [REF_Ref258932583 \h * MERGEFORMAT]) (applies to target tokens only)
		MASK_VS [5:3]	
		MASK_GS [8:6]	
		MASK_ES [11:9]	
		MASK_HS [14:12]	
		MASK_LS [17:15]	
		MASK_CS [20:18]	
		MODE [22:21]	Enable / Disable thread tracing

AMD1044_00048702

 AMD	ORIGINATE [CREATEDATE]	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *]	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	-----------------------------	---	---------------------------	----------------------------------

			0 = OFF : disable tracing 1 = ON : enable tracing
		CAPTURE_MODE [24:23]	Dictates how THREAD_TRACE_START / STOP events affect token collection (see [REF_Ref272919045 \h])
		AUTOFLUSH_EN [25]	Disable/enable autoflush (see Section [REF_Thread_tracing_mode_configuration \r \h])
		PRIV [26] (TT2.0-2.2)	Indicates whether thread trace does privileged memory writes
		TCC_PERF_EN[26] (TT2.3+)	Indicates that thread-trace will contribute (or not) to TCC performance counting.
		ISSUE_MASK [28:27]	Specifies which types of instruction scheduling updates are recorded ([REF_Ref384130861 \h])
		TEST_MODE [29]	0: Normal operation 1: Don't record time delta so that memory dump can be compared
		INTERRUPT_EN [30]	Enable interrupts when wrapping or finishing a buffer, or UTC errors.
		WRAP [31]	Boolean to indicate that memory buffer will wrap around instead of stopping at the end.
{SQTT}_CTRL (write only)	Thread tracing control register	RESET_BUFFER [31]	Setting this bit resets the thread trace write pointer, overwriting the previous trace buffer data. It also clears the FULL field of the {SQTT}_STATUS register.
{SQTT}_PERF_MASK (read/write)	Mask of CUs for performance counting	SH0_MASK [15:0]	Bitfield for enabled CUs in SH0
		SH1_MASK [31:16]	Bitfield for enabled CUs in SH1
SQ_PERFCOUNTER_CTRL (read/write)	Performance counter control register	PS_EN [0]	Include PS/VS/GS/ES/HS/LS/CS threads in performance counters (per-stage counters only)
		VS_EN [1]	
		GS_EN [2]	
		ES_EN [3]	
		HS_EN [4]	
		LS_EN [5]	
		CS_EN [6]	
		CNTR_RATE [12:8]	Period at which counters are collected. (Data is collected every N+1 intervals.)
		DISABLE_FLUSH [13]	If set, hardware will not wait for performance counters to flush when a sample is taken.
SQ_PERFCOUNTER0_SELECT ... SQ_PERFCOUNTER15_SELECT (read/write)	SQ performance counter selection registers	PERF_SEL [8:0]	Performance counter (see SQ performance counter guide for the list of available events)
		SIMD_MASK [27:24]	Mask of which SIMDs will contribute to this counter. (Only affects counters marked "per-SIMD.")
		PERF_MODE [31:28]	Counter mode (see [REF_Ref318470989 \h])

AMD1044_00048703

 AMD	ORIGINATE [CREATEDATE	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	---------------------------	---	-------------------------------	-------------------------------------

{SQTT}_USERDATA_0 ... {SQTT}_USERDATA_4 (write only)	Cause a USERDATA token to be generated	DATA [31:0]	Data to be written as payload
COMPUTE_THREAD_TRACE_ENABLE		THREAD_TRACE_ENABLE[0]	enable thread trace for pipe 1 and pipe 2 compute.
SPI_CONFIG_CNTL		ENABLE_SQG_TOP_EVENTS ENABLE_SQG_BOP_EVENTS	Enables events to be sent to shader. These bits must be enabled if capture-mode is select or select-detail. (TOP = top of pipe, BOP = bottom of pipe).
SQ_UTCL1_CNTL2		PREFETCH_PAGE	Controls the number of memory pages for the UTC (page table cache) to prefetch: 0-15. <i>Typically, set to 15.</i>

Table [STYLEREf 2 \s].[SEQ Table * ARABIC \s 2]: Thread tracing control registers

Identifier	Value	Description
{SQTT}_WAVE_MASK_NONE	0	Don't trace threads of this type
{SQTT}_WAVE_MASK_ALL	1	Trace all threads of this type on target CU when tracing is enabled
<i>Other values are not implemented</i>	<i>2-7</i>	<i>not implemented</i>

Table [STYLEREf 2 \s].[SEQ Table * ARABIC \s 2]: Subset sampling values for the {SQTT}_MODE register

{SQTT}_ISSUE_MASK Identifier	Value	Description
{SQTT}_ISSUE_MASK_ALL	0	Generate {SQTT}_TOKEN_INST tokens for all instructions
{SQTT}_ISSUE_MASK_STALLED	1	Generate {SQTT}_TOKEN_INST tokens for stalled instructions only
{SQTT}_ISSUE_MASK_STALLED_AND_IMMED	2	Generate {SQTT}_TOKEN_INST messages for stalled and other (no op/wait/set prio/etc.) instructions
{SQTT}_ISSUE_MASK_IMMED (TT2.1+)	3	Generate {SQTT}_TOKEN_INST messages only for other (no op/wait/set prio/etc.) instructions

Table [STYLEREf 2 \s].[SEQ Table * ARABIC \s 2]: Valid values for the ISSUE_MASK field of the {SQTT}_MODE register

Mode {SQTT}_CAPTURE_MODE	Value	Description	Global/Reference Data Collection	Target (a.k.a. Detail) Data Collection
ALL	0	Capture all data in thread trace buffer	Immediately when {SQTT}_MODE.MODE = ON	Immediately when {SQTT}_MODE.MODE = ON
SELECT	1	Capture data only between THREAD_TRACE_START and THREAD_TRACE_STOP events	Between THREAD_TRACE_START and THREAD_TRACE_STOP	Between THREAD_TRACE_START and THREAD_TRACE_STOP
SELECT_DETAIL	2	Capture <i>target</i> (a.k.a. <i>detail</i>) data between THREAD_TRACE_START and THREAD_TRACE_STOP events, and global/reference data at <i>all</i> times	Immediately when {SQTT}_MODE.MODE = ON	Between THREAD_TRACE_START and THREAD_TRACE_STOP

Table [STYLEREf 2 \s].[SEQ Table * ARABIC \s 2]: Thread trace capture modes ({SQTT}_CAPTURE_MODE)

AMD1044_00048704

Global Tokens	Reference Tokens	Target Tokens
WAVE_START WAVE_END INST_USERDATA PERF	TIME TIMESTAMP REG* EVENT* MISC	WAVE_*ALLOC INST INST_PC ISSUE WR_EXEC (a form of wave_start)

Table [STYLEREF 2 \s].5: Thread trace Token Types

One expected usage model for thread trace is that a debugger/profiler will halt an application at the point of interest (flushing the GPU), insert a thread trace start event, and resume application processing. Thread tracing data will be captured until the debugger/profiler halts the application, flushing the GPU and stopping thread tracing. The thread tracing events *are not* privileged in this scenario.

3.3.1 Thread tracing output buffer configuration

The *thread trace output buffer* is a region of GPU-writable memory into which thread tracing stream output data will be placed. The base address and size of the thread tracing buffer must be specified in the {SQTT}_BASE.ADDR and {SQTT}_SIZE.SIZE registers. Like TT1.0, TT2.0 offers two output modes that dictate the behavior when the output buffer is full: *linear* and *wrap*.

The operation of *linear* mode in TT2.0 is identical to previous architectures – when the output buffer is filled, no further thread tracing writes will occur. The total number of pages that were generated can be obtained via the {SQTT}_CNTR register. In general, it is difficult to predict the amount of data that will be generated by the thread tracing system. One approach to dealing with this uncertainty is to use *linear* mode and a *two-pass* collection. For the *first* pass, {SQTT}_SIZE.SIZE is set to zero and thread tracing is performed. Although the generated thread trace data will be discarded due to lack of space in the output buffer, the {SQTT}_CNTR register can be read to determine the appropriate buffer size to hold the thread trace output. (Note that this size is only approximate, since discrepancies can be expected between runs with thread tracing enabled.) After allocating an appropriately sized buffer, the *second* pass is performed. The value of {SQTT}_CNTR can be examined after the second pass to ensure that the buffer size was adequate.

The operation of *wrap* mode in TT2.0 is a departure from previous architectures. TT2.0 *wrap* mode allows a driver/application to use a *double buffered* approach to thread trace data collection. The output buffer address in {SQTT}_BASE is not in effect until one of the following events occurs:

- A write to {SQTT}_CTRL sets the RESET_BUFFER bit
- A THREAD_TRACE_FINISH event reaches the bottom of the pipeline
- The thread trace buffer wraps around

(This is true in both *linear* and *wrap* modes.) It is the third event that can be used to perform double buffering of thread trace output in *wrap* mode. Pseudocode for a double buffered approach is shown below.

1. Allocate thread trace buffers *A* and *B*, both with some size N_{Pages}
2. Set {SQTT}_BASE.ADDR to *A*
3. Write the {SQTT}_CTRL.RESET_BUFFER bit. This will cause the address of buffer *A* to become the thread trace output buffer address.
4. Set {SQTT}_BASE.ADDR to *B*
5. When buffer *A* is filled, the address of buffer *B* will become the new base address and an interrupt handler will be called. In the interrupt handler, copy thread trace output data from buffer *A* to a storage location, and set {SQTT}_BASE.ADDR to *A*.
6. When buffer *B* is filled, the address of buffer *A* will become the new base address and an interrupt handler will be called. In the interrupt handler, copy thread trace output data from buffer *B* to a storage location, and set {SQTT}_BASE.ADDR to *B*.

AMD1044_00048705

 AMD	ORIGINATE [CREATEDATE	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	---------------------------	---	---------------------------	----------------------------------

7. Go to 5
8. Enable thread-trace (mode = on)

Note that value of the {SQTT}_WPTR.READ_OFFSET field will be set in wrap mode when the buffer wraps around to the beginning. In addition, an {SQTT_MSG_TIMESTAMP} message will be placed at the beginning of each buffer.

On GPU cores with multiple shader engines, it is necessary to specify separate non-overlapping thread trace output buffers for each shader engine. The GRBM_GFX_INDEX register can be used to control whether register writes are processed by a single Shader Engine or whether they are broadcast to all Shader Engines. The fields of the GRBM_GFX_INDEX register are shown in [REF_Ref258932715 \h]. Per-Shading Engine output buffer configuration should disable broadcast writes by setting the SE_BROADCAST_WRITES bit to 0 and setting the appropriate Shader Engine index in SE_INDEX. Shader Engine broadcast mode should be enabled by setting the SE_BROADCAST_WRITES bit to 1 after the SE-specific values are set.

GRBM_GFX_INDEX Field	Bits	Default	Description
INSTANCE_INDEX	[7:0]	0x0	Specifies which instance of a block will process reads/writes. Only valid for writes if INSTANCE_BROADCAST_WRITES = 0
SE_INDEX	[23:16]	0x0	Specifies which Shader Engine (SE) will process reads/writes. Only valid for writes if SE_BROADCAST_WRITES = 0.
INSTANCE_BROADCAST_WRITES	[30]	0x1	If enabled, all instances of a block will process writes.
SE_BROADCAST_WRITES	[31]	0x1	If enabled, all shader engines will process writes.

Table [STYLEREf 3 \s].[SEQ Table * ARABIC \s 3]: GRBM_GFX_INDEX register fields

Thread trace data is not committed to memory immediately. Writes to the buffer are collected into 256-bit packets before they are sent to the Memory Controller (MC). The BUSY field of the {SQTT}_STATUS register indicates that thread trace data writes are pending.

3.3.2 Thread tracing *control* configuration

Thread tracing is enabled and disabled by the standard graphics pipeline event mechanism. Events are inserted into the pipeline between graphics draw calls or compute dispatch calls via writes to the VGT_EVENT_INITIATOR register. This register is a context (state set) register that is part of the Vertex Geometry Tessellator (VGT) module, and is used by device drivers and other internal GPU modules to trigger events. Commands from any ring as well as both indirect buffers (IB1 and IB2) can write to the VGT_EVENT_INITIATOR register. (In other words, thread trace start and stop events can come from either graphics or compute rings.) VGT_EVENT_INITIATOR is a 32-bit register with a 6-bit field (bits 0 through 5) called EVENT_TYPE. The EVENT_TYPE values relevant to thread tracing are shown in [REF_Ref259537618 \h]. The PERFCOUNTER_START and PERFCOUNTER_STOP events control the generation of {SQTT}_MSG_PERF messages (see [REF_Ref258925875 \h]), whereas the THREAD_TRACE_START and THREAD_TRACE_STOP events control the generation of the rest of the thread trace message types. The THREAD_TRACE_FINISH event causes an explicit flush of the thread trace buffer. The flush is not guaranteed to be complete until one of the following is true:

- the GUI is idle, or
- BOTH a subsequent timestamp has been returned and {SQTT}_STATUS.BUSY reads 0.

If {SQTT}_MODE.INTERRUPT_EN is set, the THREAD_TRACE_FINISH event will generate an interrupt, which the driver can wait for before checking the {SQTT}_STATUS.BUSY field to see if the buffer has been flushed. This bit also enables interrupt generation for UTC-errors (page mapping errors).

For CAPTURE_MODE_SELECT, the thread trace unit remembers which PIPE and ME_ID were used to write the VGT_EVENT_INITIATOR, and only that pipe/me will log thread trace events.

AMD1044_00048706

Identifier	Description
PERFCOUNTER_START	Start enabled performance counters. Inserted by the driver.
PERFCOUNTER_STOP	Stop enabled performance counters. Inserted by the driver.
THREAD_TRACE_START	Enable thread trace in SQ. Inserted by the driver.
THREAD_TRACE_STOP	Disable thread trace in SQ. Inserted by the driver.
THREAD_TRACE_FLUSH	Flush the thread trace buffer to memory.
THREAD_TRACE_FINISH	Flush the thread trace buffer to memory and reset the memory write address to the value last written to {SQTT}_BASE (which may change the destination buffer).

Table [STYLEREFF 3 \s],[SEQ Table * ARABIC \s 3]: VGT_EVENT_INITIATOR.EVENT_TYPE values for thread tracing

The RESET_BUFFER field of the {SQTT}_CTRL register resets the thread trace write pointer (field WPTR in {SQTT}_WPTR) and sets the FULL field in {SQTT}_STATUS to false. Setting RESET_BUFFER does *not* reset the time counter for thread tracing (unlike the behavior in TT1.0).

Note: reset_buffer should not be issued while thread-trace is active (is writing tokens to memory). This could lead to a hang in GFX9 products. Same for turning thread-trace off (mode=off) – must wait for thread-trace idle first.

Global performance counter options are controlled via the SQ_PERFCOUNTER_CTRL register. The PS_EN, VS_EN, GS_EN, ES_EN, HS_EN, LS_EN, and CS_EN fields control performance counter functionality for the different shader types. (Note that these fields apply to only those counters marked “per-stage” in SQ Performance Counter User’s Guide) The SQ_PERFCOUNTER_CTRL.CNTR_RATE field specifies how often counter values are sampled and placed in the thread trace buffer. Decreasing the sampling frequency reduces demands on memory bandwidth and buffer space. However, decreasing the sampling frequency also creates a risk of counter overflow for events that tend to have high per-cycle increments.

The sampling frequency fractions of SQ_PERFCOUNTER_CTRL.CNTR_RATE are relative to the *maximum* performance counter frequency, which is a function of the GPU hardware variant. Under the *maximum* rate, performance counter values can be recorded for a bank of 4 counters every 4 clocks from a *single* Compute Unit (CU) on *each* Shader Array (SH). As an example calculation, consider a GPU variant with 2 Shader Arrays (SH), and 8 Compute Units (CU). If 16 performance counters are active, the *maximum* performance counter sampling period for this variant will be 128 clock cycles.

Note that changes to the SQ_PERFCOUNTER_CTRL register while waves are in-flight may result in data loss.

The {SQTT}_PERF_MASK register allows individual compute units (CUs) in Shader Arrays 0 (SH0) and 1 (SH1) to be enabled or disabled.

Individual performance counter options are controlled with the SQ_PERFCOUNTERx_SEL registers (where “x” is an integer value between 0 and 15 inclusive). Note that SIMD_MASK field is only relevant for performance counters described as “per-SIMD” in SQ Performance Counter User’s Guide. Also, changes to the SQ_PERFCOUNTERx_SEL registers while waves are in flight or before reading back thread trace data may produce incorrect results.

3.3.3 Thread tracing mode configuration

The {SQTT}_MODE register is responsible for global control of which tokens are captured, and from which shader types. The {SQTT}_MODE register may be written at any time, but thread tracing will be reset when this register is written.

AMD1044_00048707

 AMD	ORIGINATE [CREATEDATE	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	---------------------------	---	---------------------------	----------------------------------

“Auto-flushing” (controlled by the {SQTT}_MODE.AUTOFLUSH_EN bit) is a mechanism that periodically flushes thread trace data to memory. The ability to disable auto-flush is only expected to be useful for testing.

The WRAP field in this register controls whether the buffer is filled in *linear* mode or *wrap* mode. In *linear* mode, writes to the thread trace buffer will be disabled once the number of bytes written becomes equal to the size as specified in the {SQTT}_SIZE register. In *wrap* mode, writes to the thread trace buffer will continue at the start of the buffer once the end of the buffer has been reached. See Section [REF_Ref259700814 \r \h] for more information on using *wrap* mode.

The CAPTURE_MODE field of the {SQTT}_MODE register controls how THREAD_TRACE_START and THREAD_TRACE_STOP events affect thread tracing data capture. (See [REF_Ref272919045 \h] for valid values for CAPTURE_MODE.)

3.3.4 Access control

In the GPU architecture, each command stream is associated with a virtual memory ID (also known as the VMID), which controls the page tables used to access memory. The VMID of the last write to the {SQTT}_BASE register is stored internally within the shader engine and it is assumed that this is the VMID being used by the application “under trace.” Only privileged register streams may write the {SQTT}_BASE register, although the CP PM4 ucode could make this register and others available to the user-mode driver. This VMID will be used to prevent the generation and export to memory of thread trace tokens generated by other applications. This mechanism prevents a malicious application from trying to capture another applications thread trace data. Be aware that if the user tries to write the {SQTT}_BASE address register and there is thread trace data outstanding for the previous VMID that held the base, then the base address register write will not be performed. In general, writes to thread trace control registers from VMIDs that do not own thread trace will be ignored while thread trace is busy. Driver software is responsible for ensuring that the thread trace buffer is only handed over to a new application after the old application has completed writing its data to a buffer.

So-called “privileged” driver code may bypass this security mechanism to trace data from multiple applications. Some command streams – generally those generated by the KMD by writing directly to the ring buffer via the bus interface -- are considered by the GPU to be “privileged”. It is noted if the {SQTT}_BASE register is configured by a privileged command stream. Data generated by command streams that do not share VMID “under trace” will not be traced unless {SQTT}_BASE register is written by one of these privileged command stream. The VM_ID_MASK field of the {SQTT}_MASK register can be used by a privileged thread trace driver to determine whether to trace streams for VMIDs other than the one used when configuring the thread trace control registers. [REF_Ref285644256 \h] describes the possible values of this field.

Starting in Volcanic Islands, these registers are no longer privileged.

Identifier	Value	Description
{SQTT}_VM_ID_MASK_SINGLE	0	Capture only data from the VM_ID used to write {SQTT}_BASE
{SQTT}_VM_ID_MASK_ALL	1	Capture all data from all VM_IDs
{SQTT}_VM_ID_MASK_SINGLE_DETAIL	2	Capture all data but only get <i>target</i> (a.k.a. <i>detail</i>) data from VM_ID used to write {SQTT}_BASE

Table [STYLEREF 3 \s].[SEQ Table * ARABIC \s 3]. Thread trace VM_ID mask ({SQTT}_VM_ID_MASK)

4 Performance effects of thread tracing

The main challenge in designing a thread tracing system that will allow the processor state information to be logged without adversely affecting performance is the *amount* of data that is generated in a parallel architecture with high

AMD1044_00048708

 AMD	ORIGINATE [CREATEDATE]	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *]	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	-----------------------------	---	---------------------------	----------------------------------

clock speeds. There are two main constraints on the data bandwidth for thread tracing on the GPU architecture. The first constraint is a 256-bit per clock (per-Shader Engine) limit on the amount of tracing data that can be generated. This constraint exists due to the design of the Shader Sequencer (SQ) units and the Shader Processor Input (SPI) unit. The second constraint is the rate at which the memory controller (MC) can effectively write data, since writes from the thread tracing system will compete with the application being traced for MC memory bandwidth. (Note that there is an additional constraint for shader program *user data* writes, which will be discussed further below.)

The GPU's thread tracing hardware uses the concept of *backpressure* to maintain a coherent thread trace buffer state. When thread tracing is enabled and writes being sent to the thread tracing buffer exceed the available resource constraints, the Shader Sequencer (SQ) will throttle execution so that issued writes have time to complete. Although the backpressure influences the recorded thread trace (by introducing stalls that are not present when an application is run without thread tracing), the integrity of the thread trace buffer contents are maintained and the trace can be still be analyzed.

Execution stalls due to thread trace backpressure can be detected via {SQTT}_TOKEN_MISC tokens in the thread trace buffer. (See [REF_Ref258925875 \h] for details on the associated {SQTT}_MSG_MISC message format.) {SQTT}_MSG_MISC messages will be found in the thread trace buffer, with MISC_TOKEN_TYPE field values of {SQTT}_MISC_TOKEN_TTRACE_STALL_BEGIN and {SQTT}_MISC_TOKEN_TTRACE_STALL_END to mark the beginning and end of the affected time period. It is also possible to use a performance counter to count the number of cycles stalled due to thread trace backpressure (see SQ_PERF_SEL_TTRACE_STALL in SQ Performance Counter User's Guide).

To avoid backpressure influence on observed performance, the amount of data written to the thread trace buffer should be limited. Since the amount of memory bandwidth available to service thread tracing buffer writes is application-dependent, there are no set rules to determine how much thread tracing is "too much." A certain amount of trial and error should be used to determine how much thread tracing data can be written without drastically changing the application running time. To provide some guidance on the impact on bandwidth of the different thread tracing tokens, [REF_Ref258932740 \h] shows the thread tracing messages from [REF_Ref258925875 \h], along with a high level (and approximate) assessment of the relative frequency of the message. Analysis of existing traces on Evergreen hardware is currently being analyzed to provide additional data from existing applications.

Message	Size (bits)	Expected Frequency	Comments
{SQTT}_MSG_TIME (TT2.0 only)	16	Low	Explicit time messages occur only when no other tokens occur
{SQTT}_MSG_TIMESTAMP	64	Low	Timestamp tokens occur only during long idle periods
{SQTT}_MSG_REG	64	Medium	
{SQTT}_MSG_WAVE_START	32	Low to Medium	
{SQTT}_MSG_WAVE	16	Low to Medium	Same frequency as {SQTT}_MSG_WAVE_START
{SQTT}_MSG_EVENT	16	Low	
{SQTT}_MSG_INST	16	High	
{SQTT}_MSG_INST_PC	64	Low	Jump instructions are expected to occur infrequently
{SQTT}_MSG_INST_USERDATA	48	Variable	Frequency of user data writes depends on shader program, but shader engine architecture has additional constraints on user data writes.
{SQTT}_MSG_ISSUE	32	High	Indicates that an instruction was issued and/or stalled due to arbitration priority.
{SQTT}_MSG_PERF	64	Low	Counted events have highly variable frequencies, but MSG_PERF messages are only generated when a local counter overflows
{SQTT}_MSG_MISC	16	Low	For TT2.1, includes explicit time messages that occur only when no other tokens occur
{SQTT}_MSG_REG_CS (TT2.1+)	48	High	

AMD1044_00048709

 AMD	ORIGINATE [CREATEDATE	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	---------------------------	---	---------------------------	----------------------------------

Table [STYLEREf 1 \s],[SEQ Table * ARABIC \s 1]: Thread tracing message sizes and expected message frequency

The {SQTt}_MSG_INST_USERDATA message warrants special consideration, since the shader engine places an additional constraint on shader program instruction *user data* writes: each Shader Array (SA) (see [REF _Ref258925566 \h]) contains a single slot that can accept a 48-bit *user data* message *every other clock*. This slot is shared by *all* wavefronts executing on *all* Compute Units in the Shader Array. The precise frequency of *user data* writes that can be performed without execution stall is dependent on the number of Compute Units in the Shader Array. In general, shader program *user data* writes should be used sparingly to avoid stalls induced by this constraint.

One technique for reducing the frequency of shader program *user data* writes will be to conditionally execute the user data write instruction on a subset of threads. It is expected that the shader program will be able to determine if the CU being used for execution is the *target* CU by examining the hardware status register of the wave executing the program. Limiting *user data* writes to the *target* CU via shader program conditional execution, which is effectively a manual technique to make a *global* token into a *target* token, will be covered in a future revision of this document.

5 Processing thread trace buffer data

5.1 Associating wavefront data with specific dispatches

Correlation between wavefront thread tracing data and the specific draw/dispatch events that generated the wavefront is accomplished by combining data from register write messages and event messages. Writes to the VGT_EVENT_INITIATOR register will result in the placement of {SQTt}_MSG_REG messages in the thread trace buffer. In addition, the beginning of each shader stage will result in the placement of {SQTt}_MSG_EVENT messages in the thread trace buffer. Applications and tools can correlate wavefront data to a specific draw or dispatch command by counting draw/dispatch calls at the API level and matching them one-for-one with the register write messages ([REF SQ_TTRACE_MSG_REG \h]) in the thread trace buffer. Event messages ([REF SQ_TTRACE_MSG_EVENT \h]) will follow the register write messages, indicating that an event has reached the top of the shader stage.

It is possible for waves from multiple draw calls to execute simultaneously, but there will be no more WAVE_START events associated with a previous dispatch occurring in that pipeline stage after the event marker passes that stage. (A WAVE_END message as well as other events associated with an earlier draw call may be placed in the thread trace buffer as that work “drains” from the pipeline, but no more WAVE_START messages from a draw will occur after an event marker passes that stage.)

The {SQTt}_MSG_EVENT message format has an EVENT_TYPE field, and the register documentation for VGT_EVENT_INITIATOR lists the possible values for EVENT_TYPE. (The event type values have been reproduced in [REF APPENDIX_A \h] for convenience.) The THREAD_TRACE_MARKER event type is a non-functional marker event that will show up in the thread trace as both a register write and an event, enabling correlation between draw calls and traced waves. (Documentation for the VGT_EVENT_INITIATOR register indicates that the THREAD_TRACE_MARKER events should be inserted by the driver or the Command Processor microcode.)

6 Visualizing thread tracing results

The contents of the thread trace buffer can be manipulated and viewed in different ways to provide different perspectives on the observed performance. It is convenient to organize the visualization and analysis options in to different slices or *views* of the data. These views reduce the data by holding one or more observation parameters (wavefront ID, SIMD ID, draw context ID, shader, time) in the thread trace output streams constant.

AMD1044_00048710

 AMD	ORIGINATE [CREATEDATE	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	---------------------------	---	---------------------------	-------------------------------------

6.1 Wavefront View

When looking at the thread trace with a view of extracting data about the execution of waves in the graphics core the following information can be obtained.

- Wavefront lifetime
- Wavefront startup latency
- Unallocated/stalled time
- Wavefront population (threads/wave)
- Wavefront instructions issued (by type)
- Wavefront memory operations (by type – V-MEM, GDS, EXPORT, LDS, constant, instruction)
- Average/min/max instruction latency (by type)

The wavefront view allows the state of the shader core to be examined.

6.2 SIMD View

In a SIMD-centric view, the following information can be extracted.

- Wave population
- Instruction issue FIFO levels (by type)
- Instruction latency (by type)

Data could be easily presented in a graph form as shown below.

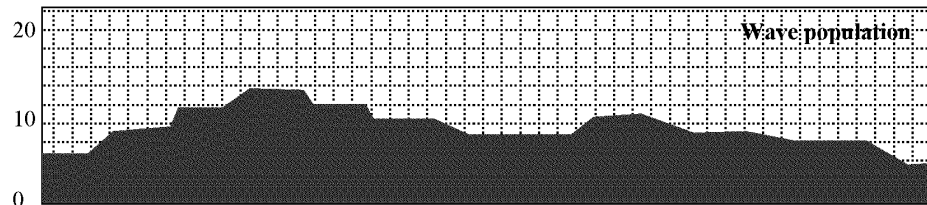


Figure [STYLEREF 2 \s],[SEQ Figure * ARABIC \s 2]: Example wave population time series visualization

6.3 Per draw-context/ per-shader data

At the coarser engine level it might be interesting to take a look at the data per draw (or actually per draw context). This allows examination of interactions of the draw calls as well as the interaction of the data at the different shader stages. The data could be presented as a graph, either per whole engine or per SIMD, and in a similar fashion to the SIMD-centric presentation it would show the number of active waves, but separately for each shader type. The contexts could be color coded for easy identification. Draw events would also be marked on the graph.

AMD1044_00048711

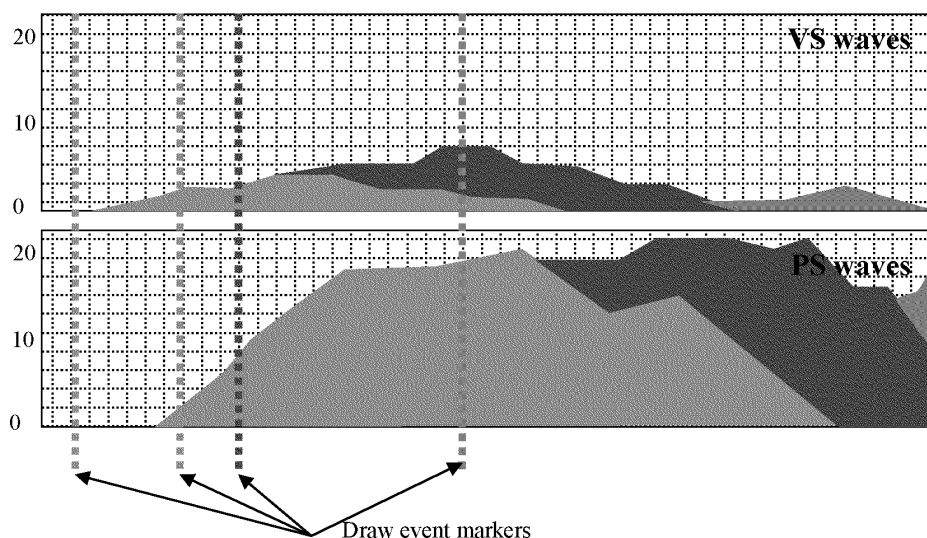


Figure [STYLEREF 2 \s].[SEQ Figure * ARABIC \s 2]: Example visualization of wave population by shader type with draw event reference times

Other data: other information could be extracted to be shown per context in the textual form

- Number of draw calls
- Start-up latency of the first call
- Number of waves (possibly by type)
- Number of threads (possibly by type)
- Total execution time
- Wave summary
- Average/min/max wave lifetime
- Average/min/max issue/execution latencies

The context data could also be presented as a summary for multiple draw contexts – e.g. min/max/average start-up latency, etc.

6.4 Thread Trace Visualization Tool Example

A screen shot from the thread trace visualization tool under development for TT2.0 is shown in [REF _Ref258929154 \h]. The top pane of the visualization tool shows a single row per wavefront, with color-coded execution information to display the wavefront state as time evolved over the course of the trace. (Additional modes that display hardware states and graphics contexts in the top pane exist as well.) The lower half of the screen shot in [REF _Ref258929154 \h] shows the number of waves active in the shader engine as well as in each CU individually (only the first 2 are shown).

AMD1044_00048712



Figure [STYLEREF 2 \s],[SEQ Figure * ARABIC \s 2]: Thread tracing visualization tool

Appendix A – VGT_EVENT_INITIATOR events

The table below lists the possible event types for the EVENT_TYPE field of the VGT_EVENT_INITIATOR. (Writes to the VGT_EVENT_INITIATOR register will result in the placement of [REF SQ_TTRACE_MSG_REG \h] messages in the thread trace buffer. In addition, [REF SQ_TTRACE_MSG_EVENT \h] messages will be placed when the register write event reaches the top of each stage.) The list below was generated from the register description file vgt.blk, version 31. Consult the latest register specification for the VGT module for the most up-to-date list.

VGT_EVENT_TYPE	Description
Reserved_0x00	Reserved -- available
SAMPLE_STREAMOUTSTATS1	Sample Streamout1 Statistics counters -- Inserted by the driver to request the GPU to sample counters associated with streamout. The CP will subsequently write them to memory.
SAMPLE_STREAMOUTSTATS2	Sample Streamout2 Statistics counters -- Inserted by the driver to request the GPU to sample counters associated with streamout. The CP will subsequently write them to memory.
SAMPLE_STREAMOUTSTATS3	Sample Streamout3 Statistics counters -- Inserted by the driver to request the GPU to sample counters associated with streamout. The CP will subsequently write them to memory.

AMD1044_00048713

 AMD	ORIGINATE [CREATEDATE]	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *]	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	-----------------------------	---	---------------------------	----------------------------------

CACHE_FLUSH_TS	Destination Cache Flush with Timestamp -- Inserted by the driver to cause the CBs, DBs, and SX to flush all prior rendering in any destination cache, wait for write confirm, then signal the CP.
CONTEXT_DONE	GFXDEC Context Done -- Inserted by the CP on the first GFXDEC state update after a draw. The 8 contexts are now shared with compute shaders and therefore, the done applies to the most recent state used for GFX, as opposed to being used for CS (compute shaders).
CACHE_FLUSH	Destination Caches Flushed -- Inserted by the driver to cause the CBs, DBs, and SX to flush all prior rendering in any destination cache to memory (No Timestamp is Generated).
CS_PARTIAL_FLUSH	Used to flush Compute Shader work after the CP in the VGT, SPI, SQ such that all previous CS work launched prior to this event will complete execution in the shader core and free all shader resources.
VGT_STREAMOUT_SYNC	Generated and consumed by the VGT for use when syncing shader engines. The driver should not insert this event.
Reserved_0x09	Reserved -- was used for SC_WAIT_WC
VGT_STREAMOUT_RESET	Resets internal streamout related registers and should be sent prior to a draw that has reprogrammed streamout registers.
END_OF_PIPE_INCR_DE	End Of Pipe event used to increment the Draw Engine Counter.
END_OF_PIPE_IB_END	End Of Pipe event used to indicate when the backend has finished processing the command buffer.
RST_PIX_CNT	Reset SPI's auto Pixel Counter -- Inserted by the driver.
Reserved_0x0E	Reserved -- was RST_VXT_CNT
VS_PARTIAL_FLUSH	Used to flush all work between the CP and the ES, GS, VS shaders including the VGT.
PS_PARTIAL_FLUSH	Used to flush all work between the CP and the ES, GS, VS, PS shaders including scan conversion, primitive assembly, and VGT.
FLUSH_HS_OUTPUT	Flush Hull Shader Output -- Sent by the VGT after an HS thread group. Used to make sure all HS thread group data is processed before the corresponding DS thread group begins.
FLUSH_LS_OUTPUT	Flush Local Shader Output -- Sent by the VGT after an LS thread group. Used to make sure all LS thread group data is processed before the corresponding HS thread group begins.
Reserved_0x13	Reserved -- available
CACHE_FLUSH_AND_INV_TS_EVENT	Destination Cache Flush and Invalidate with Timestamp -- Inserted by the driver to cause the CBs, DBs, and SX to flush and invalidate all prior rendering in any destination cache, wait for write confirm, then signal the CP.
ZPASS_DONE	Write ZPASS counts to memory -- Inserted by the driver to instruct the DBs to write out the ZPASS counters to memory. Used to support DX10 occlusion queries.
CACHE_FLUSH_AND_INV_EVENT	Destination Cache Flush and Invalidate -- Inserted by the driver to cause the CBs, DBs, and SX to flush and invalidated all prior rendering in any destination cache to memory (No Timestamp is Generated).
PERFCOUNTER_START	Start enabled event based Performance counters -- Inserted by the driver.
PERFCOUNTER_STOP	Stop enabled event based Performance counters that are event-enabled -- Inserted by the driver.
PIPELINESTAT_START	Start pipeline/streamout stat -- Inserted by the driver.
PIPELINESTAT_STOP	Stop pipeline/streamout stat -- Inserted by the driver.
PERFCOUNTER_SAMPLE	Sample the performance counters of all blocks -- Inserted by the driver to read the performance counters.
FLUSH_ES_OUTPUT	Flush Export Shader Output -- Inserted by the VGT to instruct the SX to flush all the ES output to memory.
FLUSH_GS_OUTPUT	Flush Geometry Shader Output -- Inserted by the VGT to instruct the SX to flush all the GS output to memory.

AMD1044_00048714

 AMD	ORIGINATE [CREATEDATE]	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *]	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	-----------------------------	---	---------------------------	----------------------------------

SAMPLE_PIPELINESTAT	Sample Pipeline Statistics counters -- Inserted by the driver to request the GPU to sample counters associated with pipeline stats. The CP will subsequently write them to memory.
SO_VGTSTREAMOUT_FLUSH	VGT Streamout Flush -- This event will cause VGT to update the read only offsets registers and then send a VGT_CP_strmout_flushed to instruct the CP to read the offsets.
SAMPLE_STREAMOUTSTATS	Sample Streamout0 Statistics counters -- Inserted by the driver to request the GPU to sample counters associated with streamout. The CP will subsequently write them to memory.
RESET_VTX_CNT	Reset Vertex Count -- Inserted by the driver to reset the auto index count for vertex count. There are two counters: one for GS and one for non-GS, and these should be reset separately
BLOCK_CONTEXT_DONE	Block Managed State (SQCONSDEC) Context Done -- Inserted by the CP on the first SQCONSDEC constant update after a draw.
CS_CONTEXT_DONE	GFXDEC Context Done for CS (compute shaders) -- Converted to CONTEXT_DONE event by the VGT before it sends it as an event down the pipe. Therefore, for R8xx, only the CP and VGT must be aware of this event. Inserted by the CP on the first GFXDEC state update for CS after a draw that is being used to run compute shaders. This applies to the same 8 context states as the CONTEXT_DONE event, except that it applies to the most recent context that is being used for running compute shaders
VGT_FLUSH	VGT Flush - Inserted by the driver to cause the VGT to be flushed. Used when GS ring buffer sizes are changed
Reserved_0x25	Reserved -- not available
SQ_NON_EVENT	Deprecated. This event does not do anything, and never has. SQ Non-Event -- This event is reserved for SQ
SC_SEND_DB_VPZ	SC Send Depth Block VPort Z -- Inserted by the SC when it sends the vport array Zmin and Zmax values to the DBs.
BOTTOM_OF_PIPE_TS	Bottom of the Pipe Timestamp -- Inserted by the driver to request a bottom of pipe timestamp be sent to memory, no flushing required.
FLUSH_SX_TS	Flush SX Timestamp - Inserted by the driver to cause the SX to flush its caches, then signal the CP. The other destination caches must also signal the CP for this event. All responses to the CP must be in the order the TS were received, regardless if the cache is required to otherwise act upon it. (DEPRECATED -- unavailable)
DB_CACHE_FLUSH_AND_INV	DB Flush and Invalidate - Inserted by the driver when the depth surface is paged out of memory.
FLUSH_AND_INV_DB_DATA_TS	Flush and Invalidate DB's Data Cache Only - Inserted by the driver to cause the DB to flush and invalidate only its data cache, wait for write confirm, then signal the CP. The other destination caches must also signal the CP for this event. All responses to the CP must be in the order the TS were received, regardless if the cache is required to otherwise act upon it.
FLUSH_AND_INV_DB_META	Flush and Invalidate DB's Meta (htile) Only - Inserted by the driver to cause the DB to flush and invalidate only its Meta cache.
FLUSH_AND_INV_CB_DATA_TS	Flush and Invalidate CB's Data Cache Only - Inserted by the driver to cause the CB to flush and invalidate only its data cache, wait for write confirm, then signal the CP. The other destination caches must also signal the CP for this event. All responses to the CP must be in the order the TS were received, regardless if the cache is required to otherwise act upon it.
FLUSH_AND_INV_CB_META	Flush and Invalidate CB's Meta (cmask/fmask) Only - Inserted by the driver to cause the CB to flush and invalidate only its Meta cache.
CS_DONE	Inserted by the driver using an EVENT_WRITE_EOS packet. The SQ, in response, will generate a signal to indicate that all CS work prior to this point has completed.
PS_DONE	Inserted by the driver using an EVENT_WRITE_EOS packet. The SQ, in

AMD1044_00048715

	response, will generate a signal to indicate that all PS work prior to this point has completed.
FLUSH_AND_INV_CB_PIXEL_DATA	Flush and invalidate CB's pixel (render target) data in color cache. Does not guarantee UAV(RAT) flush-and-inv, and does not flush the cmask/fmask cache either. Typically would be inserted by the driver before resolving or expanding an MSAA buffer. No wait-idle is necessary between this flush and the subsequent resolve/expand draw command.
SX_CB_RAT_ACK_REQUEST	HW generated event by the SX to signify an ack request to the CB. (DEPRECATED – unavailable)
THREAD_TRACE_START	Enable thread trace in SQ. Inserted by the driver.
THREAD_TRACE_STOP	Disable thread trace in SQ. Inserted by the driver.
THREAD_TRACE_MARKER	A non-functional marker event that will show up in the thread trace as both a register write and an event, enabling correlation between draw calls and traced waves. Inserted by the driver or the CP µcode.
THREAD_TRACE_FLUSH	Flush the thread trace buffer to memory. The flush is not guaranteed to have completed until either - the GUI is idle, or - BOTH a subsequent timestamp have been returned and {SQTT}_STATUS.BUSY reads 0
THREAD_TRACE_FINISH	Flush the thread trace buffer to memory and reset the memory write address to the value last written to {SQTT}_BASE (which may change the destination buffer). The flush is not guaranteed to have completed until either - the GUI is idle, or - BOTH a subsequent timestamp has been returned and {SQTT}_STATUS.BUSY reads 0. Only one of these events may be present in the pipeline at a given time.

Appendix B -- Performance counters for the {SQTT}_TOKEN_PERF token

See the SQ Performance Counter Users Guide.

Appendix C – Diagram of Thread Trace Token Formats

[EMBED Visio.Drawing.11]
[EMBED Visio.Drawing.11]
[EMBED Visio.Drawing.11]
[EMBED Visio.Drawing.11]

Appendix D – Token Hardware Timing Model

It is important to understand how the hardware generates these tokens and how they are sampled in order to understand and properly interpret ISSUE and INST tokens. The SQ block generates ISSUE and INST tokens at different times for the same instruction, and the SQG block collects them, adds a timestamp (time-delta) and writes them to memory. Since SQG only reports time in units of (clocks/4), and because different INST tokens arrive at different times relative to their ISSUE token, timing interpretation is complicated.

SQ reports trace events to SQG over 7 busses (expreq is simulation-only):

TTrace Channel	Bus Function	Instructions	Inst delay after Issue
0	SALU	SALU, SMEM	5
1	VALU	VALU	2
2	Misc	branch, msg, endpgm	-1

AMD1044_00048716

 AMD	ORIGINATE [CREATEDATE	EDIT DATE [SAVEDATE \@ "d-MMM-yy" *	DOCUMENT-VER. NUM. 3.2	PAGE [PAGE] of [NUMPAGES]
--	---------------------------	---	---------------------------	----------------------------------

3	SRC-C	vmem, flat	0
4	SRC-D	vmem, flat, export-grant	0 (exp-gnt has no issue)
5	Exp-req	export requests: sim only	0
ISSUE	ISSUE	Issue tokens	-

The ISSUE token identifies when an instruction starts, and the INST token indicate when it ends. For a single-cycle instruction (4 clock, e.g. a V_ADD_F32) ISSUE and INST would ideally occur in the same cycle.

[EMBED Visio.Drawing.11]

There are three fundamental problems here which make token timing difficult to correlate:

1. INST tokens come out at varying times with respect to their ISSUE tokens.
Inst tokens should all come out at the same time, AND that time should be a multiple of 4 cycles from the ISSUE token (preferably zero).
2. It is unknown how the SQG's cycle counter is synchronized to the SQ's phase
3. The presence of repeaters between SQ and SQG changes #2 above because SQG does not account for these. Repeaters between SQ0 and SQG:
all chips have zero repeaters, except these have 1:
Gfx9: Greenland (not raven1x)
Gfx8: bermuda, ellesmere, fiji, tonga
Gfx7: gladius (not milos), hawaii (has 2)

Because of #1, the time from Issue to Inst varies and means something different for each instruction type. By itself, this is easy to compensate for. It does mean that we also need to consider which SIMD and which CU a wave is running on to know how SQG will group the instructions (when the clock/4 increments).

Each CU is 2 clocks farther from SQG than the next. The phase is off by 1, and there is one cycle of TTRACE data travel time for each CU it goes through on the way to SQG.

Because the SQ's phase is not sync'd to the SQG's clock, it is unknown how SQG will group tokens into sets of 4 cycles with the same timestamp. This is extra hard to deal with.

[EMBED Visio.Drawing.11]

AMD1044_00048717