SYMPL

FP32X-AXI4

32-Bit Single-Precision

IEEE-754-2008 Compliant

Simultaneous Multi-Thread Multi-Processing

***n*-Shader GP-GPU-Compute Accelerator**

with AXI4 Burst-Mode Slave I/F

featuring: FloPoCo-Generated Floating-Point Operators

Email: sympl.gpu@gmail.com
Verilog RTL source-code repository: http://github.com/jerry-d

NOTICE OF DISCLAIMER: JERRY D. HARTHCOCK, DESIGNER AND EXCLUSIVE OWNER OF THIS IP, IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS." BY PROVIDING THE DESIGN, CODE, OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION, OR STANDARD, SAID **DESIGNER MAKES NO REPRESENTATION THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT. YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE FOR YOUR IMPLEMENTATION. FURTHERMORE, SAID DESIGNER EXPRESSLY DISCLAIMS ANY WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF INFRINGEMENT AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. THIS SPECIFICATION IS SUBJECT TO CHANGE WITHOUT NOTICE TO ANYONE**.

NOTICE REGARDING IMAGINATION TECHNOLOGIES' US PATENT NO. US8046761:
THERE EXISTS NO LOGIC WITHIN THE SYMPL FP342-AXI4 DESIGN THAT MAKES A DETERMINATION WHETHER A THREAD WILL STALL PRIOR TO A SWITCH INTO IT DURING CONTINUOUS INTERLEAVE OF THREADS OR OTHERWISE, REGARDLESS OF THE TYPE OF THREAD INTERLEAVE BEING USED FOR THE SWITCH, WHETHER SCHEDULED BY THE HARDWARE FINE-GRAINED SCHEDULER, SOFT-SCHEDULED, OR OTHERWISE. MOREOVER, THE SYMPL FP343-AXI4 SHADER INSTRUCTION PIPELINE, AS IMPLEMENTED IN THE INSTANT DESIGN, NEVER STALLS.

NOTICE REGARDING FLOPOCO FLOATING-POINT LIBRARY:
FLOPOCO FLOATING-POINT LIBRARY, GENERATOR AND WEBSITE (“THE FLOPOCO LIBRARY”) ARE THE EXCLUSIVE PROPERTY OF THIER OWNERS AND JERRY D. HARTHCOCK EXPRESSLY DISCAIMS ANY RIGHT, TITLE OR INTEREST TO/IN THEM OTHER THAN THOSE RIGHTS GRANTED AT THEIR FLOPOCO WEBSITE, http://flopoco.gforge.inria.fr, SUCH RIGHTS AND THIS DISCLAIMER BEING SUBJECT TO CHANGE AT ANYTIME AND WITHOUT NOTICE TO ANYONE. FURTHERMORE, ANY AND ALL FLOPOCO OPERATORS PROVIDED WITH THE SYMPL FP32X-AXI RTL LIBRARY ARE PROVIDED “AS-IS”, **WITHOUT ANY WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE FLOPOCO LIBRARY, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR REPRESENTATIONS THAT THE FLOBPOCO LIBRARY IS FREE FROM CLAIMS OF INFRINGEMENT AND FURTHER DISCLAIMS ANY IMPLIED WARRANTIES OF THE FLOPOCO LIBRARY'S MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.**

Revision History

Date	Revision	Author	Comments
Sept. 20, 2015	1.1	Jerry D. Harthcock	First draft of IEEE754-2008 compliant SYMPL FP32X

Preface

The purpose of this document is to provide an overview and designer reference for the SYMPL FP32X-AXI4 *n*-Shader architecture and software programming model. It is not within the scope of this guide to provide specific design details for integrating on-chip peripherals or interfacing the resulting design to external devices.

Provided in Verilog RTL source code, the SYMPL FP32X-AXI4 is the result of several attempts to create a very easy to implement and use Shader engine. To simplify scalability, it now relies on an AXI4 burst-mode slave interface and bus-master CPU of your choosing to push data and parameters from system memory into each thread's dedicated data/parameter buffer and pull results back into system memory when processing is completed, normally signaled by the respective thread asserting an interrupt to the CPU, which can be anything from a FPGA with integrated industry-standard 32-bit hard-core CPU, to various popular FPGA 32-bit, soft-cores, or even your own home-brew 32-bit soft-core.

The SYMPL FP32X-AXI4 provides a really convenient, ready-made platform for experimenting with the FloPoCo floating-point library for virtually any FPGA platform.

For more information regarding the FloPoCo library and generator, read the article at the link provided below:

> Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4):18—27, July 2011.
>
> http://perso.citi-lab.fr/fdedinec/recherche/publis/2011-DaT-FloPoCo.pdf

Table of Contents

Chapter 1

Overview

This chapter provides an overview of the SYMPL **FP32X-AXI4** simultaneous multi-thread RISC core.

1.1 Key Features

Designed for implementation in mainstream FPGAs, the SYMPL FP32X-AXI4 is a general-purpose, 32-bit floating-point (IEEE-754-2008 compliant), multi-thread RISC IP core in Verilog RTL that can be quickly customized and scaled to meet application-specific requirements for low-power, GP-GPU-compute accelerator applications (*see* block diagram of the FP32X-AXI4 configured as a single, dual or quad Shader on Pages 3 and 4):

- Easily scalable from one to *n* Shaders.
- Four threads per Shader core, for a maximum of (*n* x 4) simultaneous threads, with *n* being the maximum number of slave channels your AXI4 interconnect can accomodate. Each thread has its own program counter, index and status registers, including approximately 64 words of private, zero-page SRAM and 32 words of global, zero-page SRAM.
- Fetch cycles of each thread can be made to interleave so as to avoid/hide latency. Interleaving is accomplished using a globally mapped, fine-grained scheduler register.
- Programmable fine-grained scheduler register enables specific number of clocks/fetches in the current thread before switching to the next in the queue, round-robin. Thread granularity individually programmable from 1-255 clocks.
- Each thread has access of up to 2,048 (32-bit) words of its own **private** parameter/data SRAM, the contents of which are pushed into it by a system CPU acting as a load-balancer/course-grained scheduler, via the integral AXI4 slave interface. In addition, all four threads of a Shader have access to up to 8,192 words of **global** intermediate result buffer SRAM block (one per Shader core), also accessible to the CPU via AXI4 slave interface.
- Addressing modes include direct (zero page), indirect, indexed with variable auto-post-increment/decrement, immediate, and table-read from program memory.
- Sixteen native atomic op-codes with multiple alias capability for integer and logic operations, which execute in one clock cycle without stalls.
- Nine IEEE-754-2008 Compliant, fully pipelined, floating-point operators including: FADD, FSUB, FMUL, FDIV, FMA, SQRT, DOT, LOG, EXP, ITOF and FTOI.

Atomic Instructions/Clocks :

MOV	1
AND	1
OR	1
XOR	1
ADD	1
ADDC	1
SUB	1
SUBB	1
BCND	1
BTBS	1
BTBC	1
DBNZ	1
SHFT (barrel)	1
MUL	1
RCP	1
SIN	1
COS	1
TAN	1
COT	1

Floating-Point Operators/Clocks:

FADD	4
FSUB	4
FMUL	2
FMA	8
FDIV	11
SQRT	12
DOT	9
LOG	9
EXP	7
ITOF	2
FTOI	2

Key Features (continued)

- Each floating-point operator has its own 16-word, randomly addressable, read-only result buffer with semaphore, mapped into each thread's private memory space. Except for DOT, operators can accept an input every clock cycle, such that when a given operator's pipe is full, it can automatically write a result to a thread's private floating-point result buffer every clock cycle.
- Each randomly accessible floating-point operator result buffer (there are 128 of them per thread) has its own semaphore (ready flag) which is automatically tested whenever a MOV instruction attempts to read a result buffer location. If not ready, the MOV instruction will automatically rewind the PC to the original MOV instruction program fetch location and re-fetch, all in one clock cycle. Consequently, a given Shader core's instruction pipeline never stalls.
- Relatively small AXI4 memory footprint of only 32-k words (128k bytes).
- Optional look-up table instructions include: SIN, COS, TAN, COT and RCP (reciprocal). Trig functions have a resolution of +/- one degree. Reciprocal range is +127 to -128, with 0 returning 0x7F80000 (infinity).
- Each thread has two dedicated hardware loop counters that can be used in combination with the DBNZ instruction to decrement-and-branch-if-zero in one execution clock cycle, which is especially useful for tight inner-looping.
- Single/dual-operand integer maths, logical, branch and move instructions include: MOV, AND, OR, XOR, ADD, ADDC, SUB, SUBB, SHFT, BTBC, and BTBS. There are many possible aliases of the above instructions, including: BCND, BCC, BCS, BZ, BNZ, DBNZ and RPT, just to name a few.
- Each thread has four indirect pointer registers, AR3 : AR0, which can be programmed to automatically post-increment/decrement in steps of 1 to 255.
- Seven interrupt sources per thread: non-maskable interrupt (NMI), maskable floating-point exception interrupts for invalid operation, divide-by-zero, overflow, underflow-inexact, inexact, and maskable, general-purpose interrupt (IRQ), each with its own vector.
- Each thread has its own 20-bit, programmable timer with time-out flag, presently connected to the thread's NMI input.
- RPT instruction “repeats” the immediately following instruction *n* times (after the first execute) and automatically places a lock on the fine-grained scheduler so that, while in repeat mode, the thread that placed the lock consumes all available clocks (i.e., temporarily disables thread interleaving) until the RPT is completed.
- All registers are memory-mapped.
- Modified, dual-operand, Harvard memory model with table-read operand from program memory capability.
- Very simple to implement, program and use.
- Shader core (minus floating-point operators) has a relatively small logic footprint when FPGA embedded RAM blocks are employed (rather than LUTs) for memory.

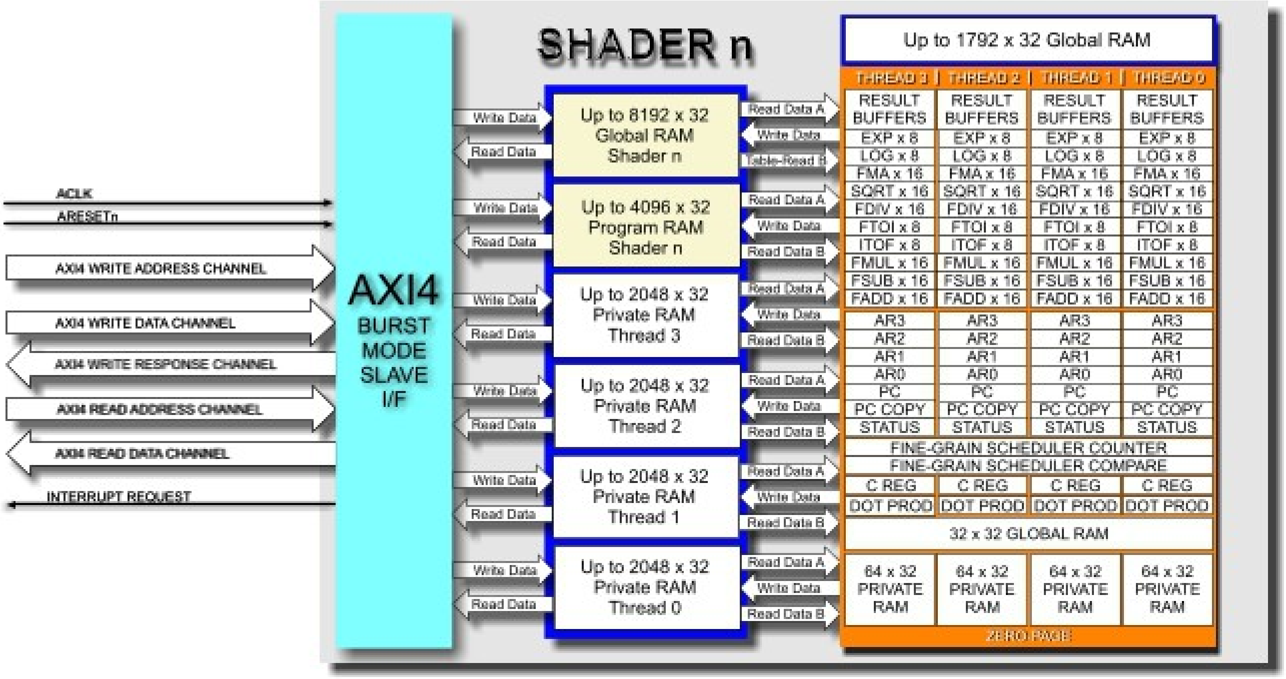
1.2 FP32X-AXI4 Block Diagram—Single-Shader Implementation

The block diagram in *Figure 1—1* below shows a single-Shader core coupled to a AXI4-compliant, burst-mode, slave DMA interface. Note that the core employs tri-ported SRAMs (two read-side and one write-side address/data buses) for dual-operand read operations. This allows the core to read two operands and write them to a given memory-mapped, floating-point operator using a single MOV instruction and, when the required floating-point operations are complete, simultaneously read two results out of their respective result buffers and immediately write both results as operands back into the same or different operators' input register, again with a single MOV instruction. Each operator has its own pipeline and, in this respect, is decoupled from the core's instruction pipeline. All threads in a given Shader core share the same memory-mapped operators, but results bin-out to randomly addressable result buffers that are private to a given thread and correspond to the same operator address written to.

Each thread has its own private, 2,048-input SRAM that the CPU pushes parameters and data into for processing by way of an AXI4 burst-mode slave interface. This is done by the CPU when it sees that a respective thread is spinning idle in the “DONE” state by reading the AXI4 Control Status Register (CSR). When the required parameters are pushed in, the CPU writes a non-zero semaphore to a predetermined location in the buffer to signal such thread that parameters and data are available. The semaphore written also happens to be the program/thread entry-point to the routine/thread needed to process the data according to the submitted parameters.

While spinning idle and DONE, the thread is sampling the semaphore location, testing for non-zero. When it sees that the semaphore is non-zero, the thread loads its PC with the semaphore, causing a jump to the corresponding instruction sequence needed to process the data according to the parameters. Upon entry, the thread clears the its DONE flag, to signal the CPU that it is now BUSY processing the data. When processing is complete, the thread re-asserts the DONE bit, causing a CPU interrupt request (if enabled), signaling that results are available and that it is ready to receive and process the next packet.

Figure 1—1. Single-Shader Block Diagram



THREAD 3	THREAD 2	THREAD 1	THREAD 0
RESULT BUFFERS	RESULT BUFFERS	RESULT BUFFERS	RESULT BUFFERS
EXP x 8	EXP x 8	EXP x 8	EXP x 8
LOG x 8	LOG x 8	LOG x 8	LOG x 8
FMA x 16	FMA x 16	FMA x 16	FMA x 16
SQRT x 16	SQRT x 16	SQRT x 16	SQRT x 16
FDIV x 16	FDIV x 16	FDIV x 16	FDIV x 16
FTOI x 8	FTOI x 8	FTOI x 8	FTOI x 8
ITOF x 8	ITOF x 8	ITOF x 8	ITOF x 8
FMUL x 16	FMUL x 16	FMUL x 16	FMUL x 16
FSUB x 16	FSUB x 16	FSUB x 16	FSUB x 16
FADD x 16	FADD x 16	FADD x 16	FADD x 16
AR3	AR3	AR3	AR3
AR2	AR2	AR2	AR2
AR1	AR1	AR1	AR1
AR0	AR0	AR0	AR0
PC	PC	PC	PC
PC COPY	PC COPY	PC COPY	PC COPY
STATUS	STATUS	STATUS	STATUS
FINE-GRAIN SCHEDULER COUNTER			
FINE-GRAIN SCHEDULER COMPARE			
C REG	C REG	C REG	C REG
DOT PROD	DOT PROD	DOT PROD	DOT PROD
32 x 32 GLOBAL RAM			
64 x 32 PRIVATE RAM	64 x 32 PRIVATE RAM	64 x 32 PRIVATE RAM	64 x 32 PRIVATE RAM
ZERO PAGE			

1.3 FP32X-AXI4 Block Diagram

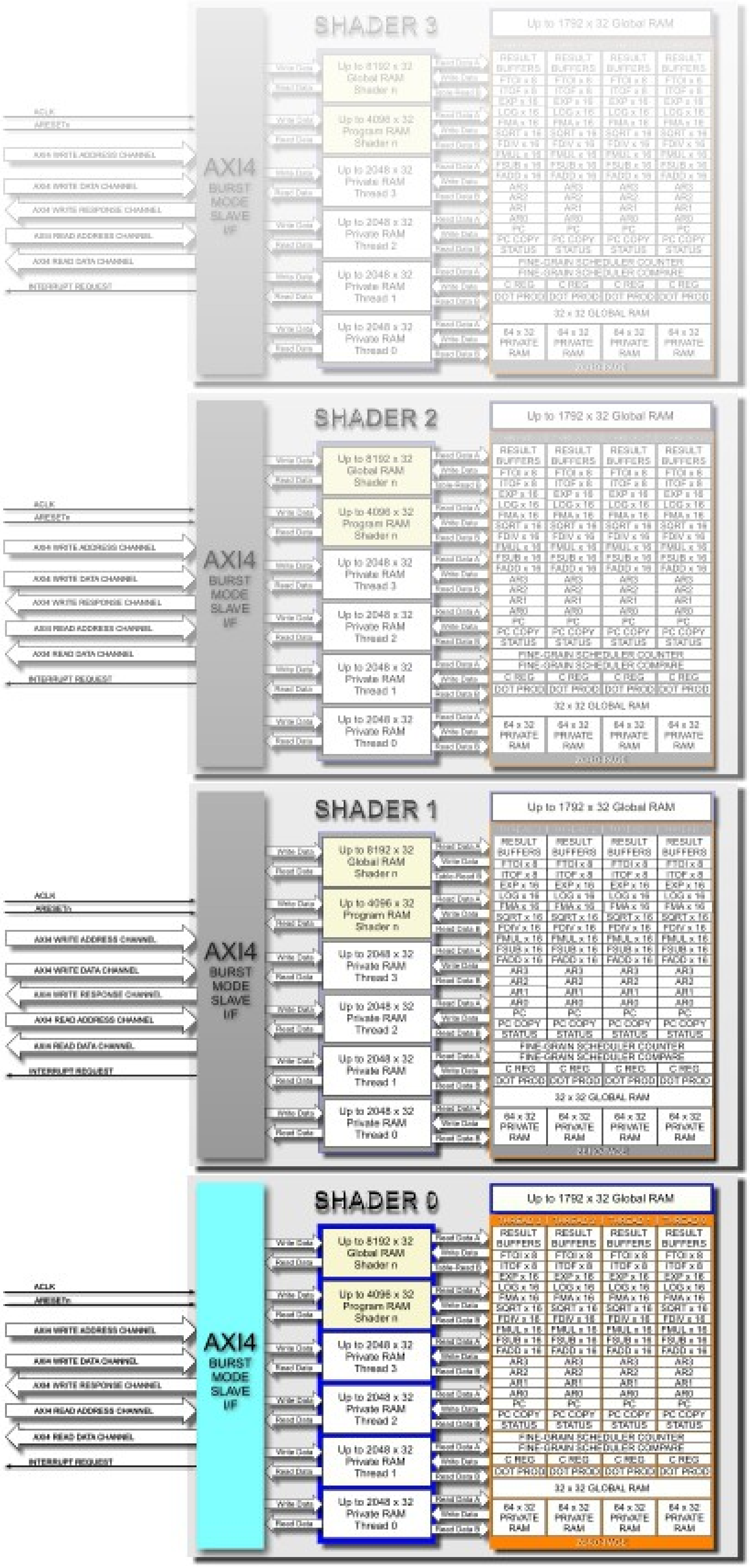
The block diagram in *Figure 1—2* on the right shows how easy it is to scale your custom GP-GPU-Compute design to virtually any number of Shaders by simply dropping more instantiations of the Shader into your design. The limit is determined by the amount of available fabric and memory in your FPGA and by the maximum number of slave AXI4 channels available in your AXI4 interconnect.

A 15-bit module port named "BASE" allows you to strap the base address of where you want a given Shader to reside you your AXI4 memory space, with a granularity of only 32k words (128k bytes), 32-bit aligned.

For your first project using the SYMPL FP32X-AXI4, it is recommended that you start with just one Shader. Once you've run a few simulations and want to try simulating and/or synthesizing more than one core with floating-point operators enabled, simply remove the comments from around the Shader and desired floating-point operator instantiations you want to expose and implement.

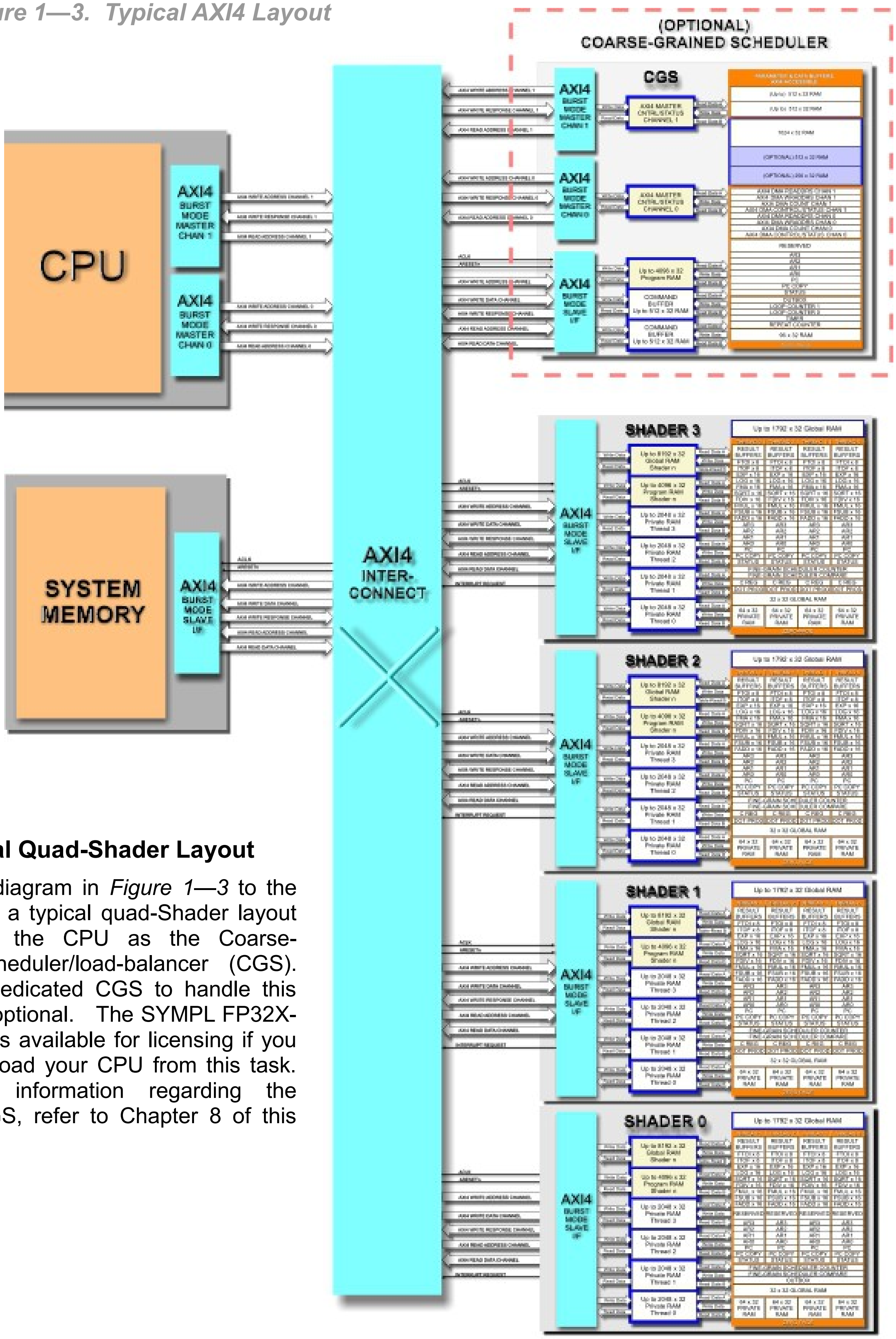
All the required code for implementing a single Shader core is in the posted RTL. Scaling to more than one Shader is a simple matter of dropping another instantiation into your design and modifying the example test-bench provided with the other sources at the SYMPL repository at GitHub.

Figure 1—2. Scalable Shader Diagram



SHADER 3
SHADER 2
SHADER 1
SHADER 0
ACLK
ARESETn
AXI4 WRITE ADDRESS CHANNEL
AXI4 WRITE DATA CHANNEL
AXI4 WRITE RESPONSE CHANNEL
AXI4 READ ADDRESS CHANNEL
AXI4 READ DATA CHANNEL
INTERRUPT REQUEST
AXI4
BURST
MODE
SLAVE
I/F
Write Data
Read Data
Up to 8192 x 32 Global RAM Shader n
Up to 4096 x 32 Program RAM Shader n
Up to 2048 x 32 Private RAM Thread 3
Up to 2048 x 32 Private RAM Thread 2
Up to 2048 x 32 Private RAM Thread 1
Up to 2048 x 32 Private RAM Thread 0
Read Data A
Read Data B
Up to 1792 x 32 Global RAM
RESULT BUFFERS
FTOI x 8
ITOF x 8
EXP x 16
LOG x 16
FMA x 16
SQRT x 16
FDIV x 16
FMUL x 16
FSUB x 16
FADD x 16
AR3
AR2
AR1
AR0
PC
PC COPY
STATUS
FINE-GRAIN SCHEDULER COUNTER
FINE-GRAIN SCHEDULER COMPARE
C REG
DOT PROD
32 x 32 GLOBAL RAM
64 x 32 PRIVATE RAM

Figure 1—3. Typical AXI4 Layout


(OPTIONAL)
COARSE-GRAINED SCHEDULER
CGS
AXI4 BURST MODE MASTER CHAN 1
AXI4 BURST MODE MASTER CHAN 0
AXI4 BURST MODE SLAVE I/F
AXI4 MASTER CNTRL/STATUS CHANNEL 1
AXI4 MASTER CNTRL/STATUS CHANNEL 0
Up to 4096 x 32 Program RAM
COMMAND BUFFER Up to 512 x 32 RAM
COMMAND BUFFER Up to 512 x 32 RAM
CPU
AXI4 BURST MODE MASTER CHAN 1
AXI4 BURST MODE MASTER CHAN 0
SYSTEM MEMORY
AXI4 BURST MODE SLAVE I/F
AXI4 INTER-CONNECT
SHADER 3
SHADER 2
SHADER 1
SHADER 0
AXI4 BURST MODE SLAVE I/F
Up to 1792 x 32 Global RAM
Up to 8192 x 32 Global RAM Shader n
Up to 4096 x 32 Program RAM Shader n
Up to 2048 x 32 Private RAM Thread 3
Up to 2048 x 32 Private RAM Thread 2
Up to 2048 x 32 Private RAM Thread 1
Up to 2048 x 32 Private RAM Thread 0
32 x 32 GLOBAL RAM
64 x 32 PRIVATE RAM

1.4 Typical Quad-Shader Layout

The block diagram in *Figure 1—3* to the right shows a typical quad-Shader layout using only the CPU as the Coarse-Grained-Scheduler/load-balancer (CGS). Use of a dedicated CGS to handle this function is optional. The SYMPL FP32X-AXI4-CGS is available for licensing if you need to offload your CPU from this task. For more information regarding the SYMPL CGS, refer to Chapter 8 of this document.

FP32X-AXI4 Mixed-Mode RTL Library

Chapter 2

2.1 Verilog RTL Library Description

Table 2—1 lists the various Verilog RTL modules that comprise the FP32X-AXI4.

Table 2—1. SYMPL FP32X-AXI4 Synthesizable Verilog RTL Source Code Library

File Name	Description	Used by/for
fp321_axi.v	FP321-AXI4 top-level design in Verilog RTL	This is the top level
shader.v	Single shader core plus axi4 interface and SRAM	fp321_axi.v
axi4_slave_if.v	AXI4 slave interface	fp321_axi.v
core.v	Single GP-GPU-compute engine	shader.v
RAM_tp.v	Parameterized tri-port SRAM with collision R/W	shader.v and core.v
aSYMPL_func.v	Wrapper for the func_*.* FP operators	core.v
sched_stack.v	Fine-grained scheduler stack	core.v
arn_sel.v	Auxiliary register selector, indirect address mode	core.v
adder_32.v	ALU 32-bit adder	core.v
int_cntrl.v	Prioritized interrupt controller	core.v
func_trig.v	Wrapper for optional trig look-up tables	core.v
rcp.v	Optional reciprocal look-up table	core.v
func_add.v	Wrapper for FADD & FSUB FP operators	aSYMPL_func.v
func_mul.v	Wrapper for FMUL FP operator	aSYMPL_func.v
func_div.v	Wrapper for FDIV FP operator	aSYMPL_func.v
func_sqrt.v	Wrapper for SQRT FP operator	aSYMPL_func.v
func_log.v	Wrapper for LOG operator	aSYMPL_func.v
func_exp.v	Wrapper for EXP operator	aSYMPL_func.v
func_itof.v	Wrapper for ITOF operator	aSYMPL_func.v
func_ftoi.v	Wrapper for FTOI operator	aSYMPL_func.v
func_fma.v	Wrapper for FMA operator	aSYMPL_func.v
func_dot.v	Wrapper for DOT operator	aSYMPL_func.v
Dot_Clk.v	Sub-wrapper for built DOT operator	func_dot.v
Fma_Clk.v	Sub-wrapper for built FMA operator	func_fma.v
IEEE754_To_FP_filtered.v	Wrapper for FloPoCo IEEE754_to_FP converter	Most operator wrappers
round_sel.v	Part of FP directed rounding function	Most operator wrappers
exc_capture.v	FP exception diagnostic information capture	core.v
FP321_axi_tf.v	Verilog test-fixture for fp321_axi.v	Stimulus

aSYMPL32.tbl	SYMPL FP321-AXI4 instruction table	Cross-32 assembler
FP321_test1.asm	Example assembly language thread source-code	Stimulus after assembly
FP321_test1.v	Verilog program memory load file	FP321_axi_tf.v
FP321_test1.LST	Assembled listing for extracting FP321_test1.v	FP321_test1.v

2.2 FloPoCo 32-bit VHDL Floating-Point Library

Table 2—2 lists the various VHDL RTL modules presently employed by the FP32X-AXI4.

Table 2—2. Synthesizable VHDL FloPoCo Floating-Point Operators

File Name	Description	Used by/for
Add_Clk.vhdl	FADD operator; 3-stage pipe	func_add.v
Mul_Clk.vhdl	FMUL operator; 1-stage pipe	func_mul.v
Div_Clk.vhdl	FDIV operator; 10-stage pipe	func_div.v
Sqrt_Clk.vhdl	SQRT operator; 11-stage pipe	func_sqrt.v
Log_Clk.vhdl	LOG operator; 8-stage pipe	func_log.v
Exp_Clk.vhdl	EXP operator; 6-stage pipe	func_exp.v
FusedADD38.vhdl	49-bit FP adder with no rounding	Dot_Clk.v and Fma_Clk.v
Mul_Clk_expert.vhdl	49-bit FP multiplier with no rounding	Dot_Clk.v and Fma_Clk.v
Mult_X1_rnd.vhdl	49-bit x 1 multiplier (33-bit result) with rounding	Dot_Clk.v and Fma_Clk.v
FP_To_FXP.vhdl	FTOI operator; 1-stage pipe	func_ftoi.v
FXP_To_FP.vhdl	ITOF operator; 1-stage pipe	func_itof.v
IEEE754_To_FP.vhdl	IEEE754-to-FloPoCo format; combinatorial	Most of the above
FP_To_IEEE754.vhdl	FloPoCo-to-IEEE754 format; combinatorial	Most of the above

The SYMPL FP321-AXI4 presently employs eight, fully pipelined and/or combinatorial, floating-point operators (in VHDL) generated by ***FloPoCo version 3.0, Beta-5*** release, which include the following:

The “**Flo**ating-**Po**int **Co**res” generator software version 3.0 can be downloaded from the following FloPoCo website, which includes additional links to installation instructions and user's manual: http://flopoco.gforge.inria.fr

32-bit, single-precision implementations of all the above-listed operators can get quite large in terms LUTs and registers, especially with four SYMPL FP321-AXI4 Shader cores instantiated in your design. If you've never tried working with floating-point operators and/or multi-core processor designs before, it is recommended you try a few simulations with just one Shader core instantiated and no floating-point operators exposed to your compiler at first, just to familiarize yourself with core architecture and target FPGA tool-set. Then, when you are ready, start instantiating floating-point operators, one at a time.

Among FloPoCo's many features is the ability to easily tweak the operational clock frequency (and consequently, latency) of each of the primary floating-point operators. However, when doing so, bear in mind that the FP321-AXI4 is presently a single-clock design and you will eventually reach a point of diminishing returns, due to the fact that an increase in maximum operational speed of an operator set will not necessarily result in an increase in the underlying Shader engine's maximum operational speed. Accordingly, the first step is to determine what the maximum clock speed of the Shader core is for a given FPGA family, and then use those numbers as parameters for generating the FloPoCo cores. The easiest way to do that is by performing an initial place and route with the floating-point math block commented out.

2.3 Shader Programming Model and Architectural Overview

The SYMPL FP32X can be described as being very much like four, 32-bit RISC cores, joined at the hip, sharing the same ALU and some shared (globally-mapped) memory, but also having their own program counter, status register, index registers and a relatively large amount of private, zero-page (directly-addressable) SRAM, private parameter/data buffers and private, randomly addressable, floating-point result buffers, sixteen per operator (except ITOF and FTOI).

The advantage to this approach is that the instruction fetch cycle of each thread can be made to interleave, with a granularity down to one clock cycle, by properly configuring the globally-mapped, fine-grained scheduler.

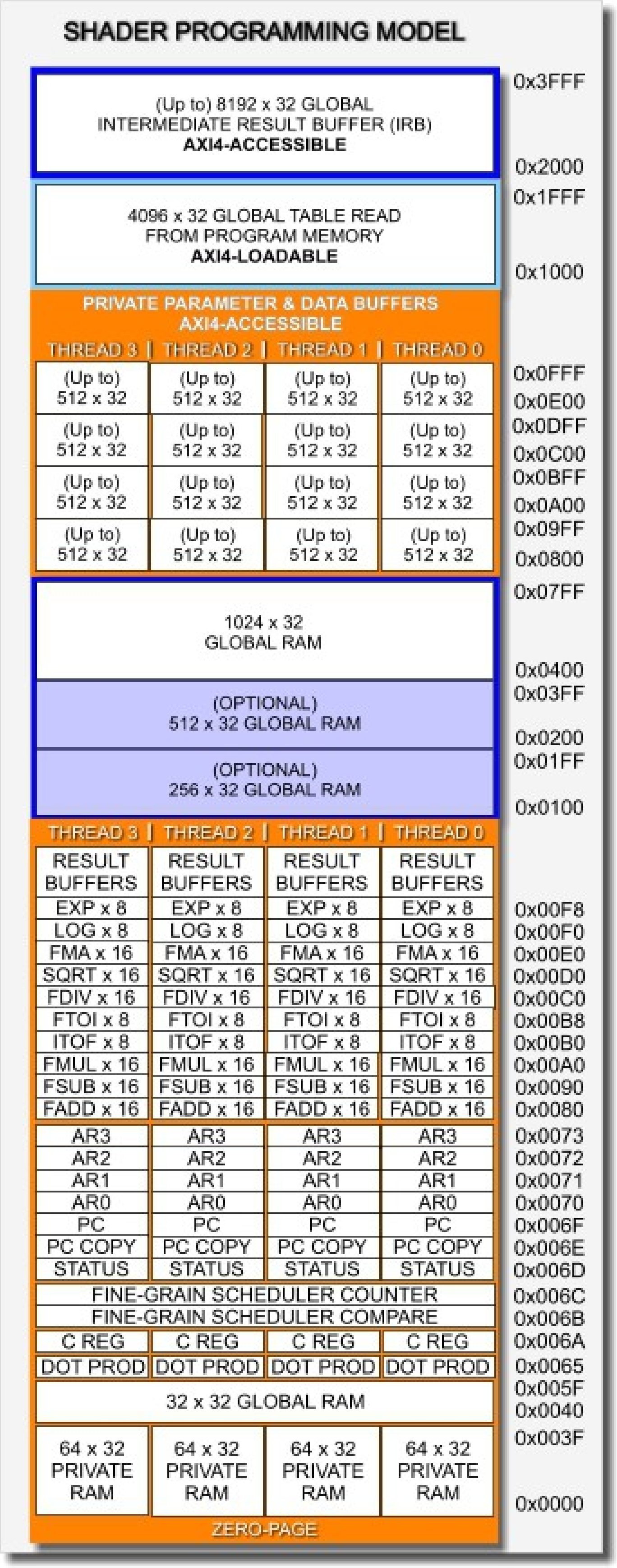
With all four threads of a given Shader scheduled for one clock each (out of four), the PCs of each thread appear "as if" there is no latency when a branch is taken, thereby hiding the fact that the instruction pipeline is three levels deep, meaning that the two instructions of a given thread that immediately follow a branch instruction and would otherwise be fetched and discarded when a branch is taken, does not appear to occur with at least three threads scheduled with interleave of one clock each.

Consequently, the above-mentioned configuration is much more efficient at processing large packets of data than conventional microprocessor architectures.

Conversely, if a given situation requires employment of only one thread, meaning that only one thread is scheduled for want of work to do, then performance is diminished somewhat, in that there are no other threads running that can be used to interleave fetches and hide latency. In such scenarios, the two instructions in a given thread following a branch instruction are fetched and discarded when the branch is taken.

*Note: due to lack of space, not all registers are shown in the programming model to the right.

Figure 2—1. Shader Programming Model



SHADER PROGRAMMING MODEL

(Up to) 8192 x 32 GLOBAL INTERMEDIATE RESULT BUFFER (IRB) **AXI4-ACCESSIBLE**				0x3FFF 0x2000
4096 x 32 GLOBAL TABLE READ FROM PROGRAM MEMORY **AXI4-LOADABLE**				0x1FFF 0x1000
PRIVATE PARAMETER & DATA BUFFERS AXI4-ACCESSIBLE				
THREAD 3	THREAD 2	THREAD 1	THREAD 0	
(Up to) 512 x 32	(Up to) 512 x 32	(Up to) 512 x 32	(Up to) 512 x 32	0x0FFF 0x0E00
(Up to) 512 x 32	(Up to) 512 x 32	(Up to) 512 x 32	(Up to) 512 x 32	0x0DFF 0x0C00
(Up to) 512 x 32	(Up to) 512 x 32	(Up to) 512 x 32	(Up to) 512 x 32	0x0BFF 0x0A00
(Up to) 512 x 32	(Up to) 512 x 32	(Up to) 512 x 32	(Up to) 512 x 32	0x09FF 0x0800
1024 x 32 GLOBAL RAM				0x07FF 0x0400
(OPTIONAL) 512 x 32 GLOBAL RAM				0x03FF 0x0200
(OPTIONAL) 256 x 32 GLOBAL RAM				0x01FF 0x0100
THREAD 3	THREAD 2	THREAD 1	THREAD 0	
RESULT BUFFERS	RESULT BUFFERS	RESULT BUFFERS	RESULT BUFFERS	
EXP x 8	EXP x 8	EXP x 8	EXP x 8	0x00F8
LOG x 8	LOG x 8	LOG x 8	LOG x 8	0x00F0
FMA x 16	FMA x 16	FMA x 16	FMA x 16	0x00E0
SQRT x 16	SQRT x 16	SQRT x 16	SQRT x 16	0x00D0
FDIV x 16	FDIV x 16	FDIV x 16	FDIV x 16	0x00C0
FTOI x 8	FTOI x 8	FTOI x 8	FTOI x 8	0x00B8
ITOF x 8	ITOF x 8	ITOF x 8	ITOF x 8	0x00B0
FMUL x 16	FMUL x 16	FMUL x 16	FMUL x 16	0x00A0
FSUB x 16	FSUB x 16	FSUB x 16	FSUB x 16	0x0090
FADD x 16	FADD x 16	FADD x 16	FADD x 16	0x0080
AR3	AR3	AR3	AR3	0x0073
AR2	AR2	AR2	AR2	0x0072
AR1	AR1	AR1	AR1	0x0071
AR0	AR0	AR0	AR0	0x0070
PC	PC	PC	PC	0x006F
PC COPY	PC COPY	PC COPY	PC COPY	0x006E
STATUS	STATUS	STATUS	STATUS	0x006D
FINE-GRAIN SCHEDULER COUNTER				0x006C
FINE-GRAIN SCHEDULER COMPARE				0x006B
C REG	C REG	C REG	C REG	0x006A
DOT PROD	DOT PROD	DOT PROD	DOT PROD	0x0065
32 x 32 GLOBAL RAM				0x005F 0x0040
64 x 32 PRIVATE RAM	64 x 32 PRIVATE RAM	64 x 32 PRIVATE RAM	64 x 32 PRIVATE RAM	0x003F 0x0000
ZERO-PAGE				

2.4 Basic Architecture

Having separate program /data memory address and data buses, the SYMPL FP32X-AXI4 follows a modified Harvard memory model, with enhancements that include tri-ported memory/registers and table-read addressing mode for reading tables and constants from program memory “as if” it were data memory. The use of tri-ported data memory is necessary for dual-operand-read, single-result-write operations, which is one of the hallmarks of register-based load-store models. For operations involving floating-point operators, the SYMPL model also has the ability to not only read two 32-bit operands, but also write two 32-bit operands within a given memory-mapped operator's input register address range, all with a single MOV instruction.

To develop a better appreciation for dual-operand read/write operations using a single MOV instruction, consider the following routine that computes the NORMs of a list of 32 (X, Y) entries —without a single stall. To accomplish the task in the shortest time possible (excepting maybe unrolled loops) , all four threads of a single Shader are employed, meaning that the 32 entries are divided into packets of 8 each and submitted to each thread, wherein each thread executes the same identical code from the same program memory, but on different data:

```
list_start:     equ     0x0820                  ;start of list in packet memory x x x x … y y y y etc
result_start:   equ     0x0900                  ;start of result buffer memory

                ; initialize indirect pointers
                mov     AR0, #list_start        ;point to first x
                mov     AR1, #list_start+8      ;point to first y
                mov     AR2, #FMUL_0            ;point to first FMUL operator (there are 16 of them)

sqr_terms:      rpt     #7                      ;”repeat” next instruction 7 times (executed 8 times)
                mov     *AR2++, *AR0++, *AR0            ;calculate square of x
                rpt     #7
                mov     *AR2++, *AR1++, *AR1            ;calculate square of y

                mov     AR0, #FMUL_0            ;point to first x result
                mov     AR1, #FMUL_8            ;point to first y result
                mov     AR2, #FADD_0            ;point to first FADD operator (there are 16 of them)

fadd_sqrs:      rpt     #7                      ;”repeat” next instruction 7 times (executed 8 times)
                mov     *AR2++, *AR0++, *AR1++          ;FADD x^2 + y^2

                mov     AR0, #FADD_0            ;point to first FADD result
                mov     AR2, #SQRT_0            ;point to first SQRT operator input register

get_sqrt:       rpt     #7
                mov     *AR2++, AR0++           ;calculate square root for each

                mov     AR0, #SQRT_0            ;point to first square root result
                mov     AR1, #result_start      ;point to first result location in packet memory

load_pckt:      rpt     #7
                mov     *AR1++, *AR0++          ;copy results to result buffer in packet memory
```

Note that in the above example, there is in reality only one op-code used to perform the entire sequence, in that RPT (repeat) is actually an alias of “MOV”, because the RPT register is memory-mapped. Also note that as long as the contents of the RPT register is not zero, a lock is automatically placed on the fine-grained scheduler, preventing interleaving of threads during such time. When the RPT sequence is completed, the lock is automatically removed, allowing the fine-grained scheduler to advance to the next thread, round-robin.

In the above example, the fine-grained scheduler is configured for one clock per thread, with all four threads in the scheduler. Although the FMUL operator pipeline requires two clocks to complete and the SQRT operator requires twelve, it appears "as if" these operations complete in just one clock each. This is what is referred to as hidden latency. The interleaving of threads in combination with the automatic lock caused by the RPT instruction provide the required time to complete the operation before the current thread's time slot comes back around for the fetch of the next instruction, without stalling at any point in the instruction sequence for the above example.

Stated another way, since all four threads execute the same instruction sequence, which includes a lock on the fine-grained scheduler for eight clocks during RPT, the first SQRT result is available for reading by the first thread (thread0) well before the scheduler comes back around with the next time slot for thread0, with the same being true for the remaining threads.

The main difference between this architecture and other industry standard RISC models is that it does not employ a "register file" typically found in "load-store" models. Instead, this architecture makes use of FPGA embedded RAM and a direct addressing mode to do memory-memory and memory to register transfers within the zero-page (direct address mode) range from 0x0000 to 0x00FF. In this respect, the SYMPL core is more aptly described as a "mover" architecture rather than "load-store".

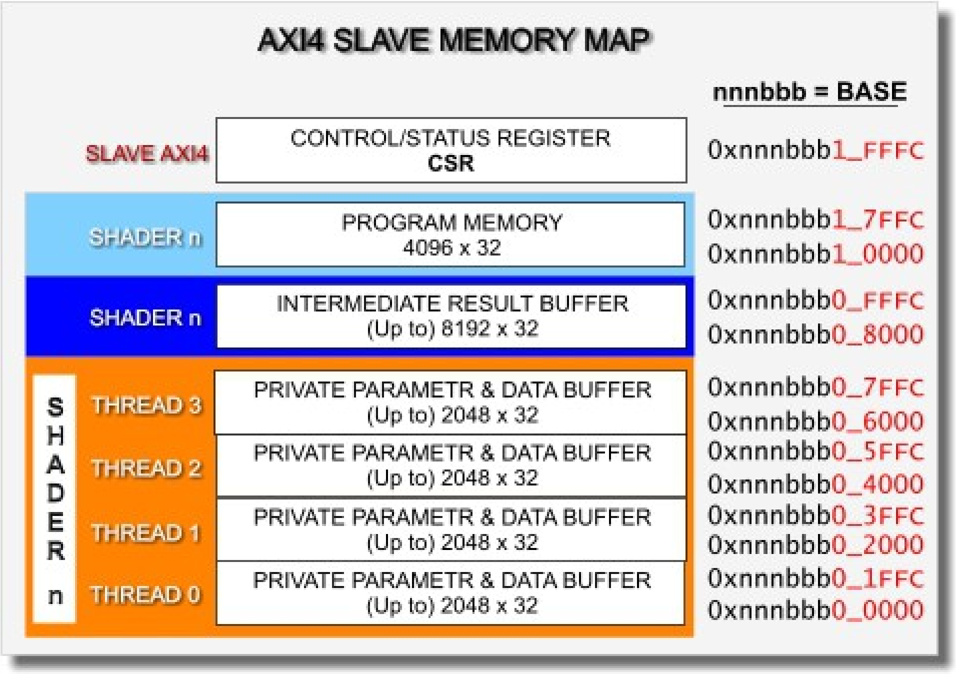
To extend address reach beyond zero-page, the indirect addressing mode is employed by using a given thread's auxiliary registers as indirect pointers. Each thread has four of them: AR3 to AR0.

Another feature of the SYMPL FP32X architecture typically not found in other industry standard RISC architectures is the fact that all of its registers are memory-mapped, with each register residing at a unique, directly-addressable location within zero-page. One of the main advantages to memory-mapped registers is that there is no need for unique op-codes to access/employ them, enabling, in this instance, use of a relatively short op-code field of only four bits, thereby freeing up more bits in the instruction word for increased direct addressing mode reach.

2.5 AXI4 MEMORY MAP

The memory map shown in *Figure 2—2* shows what a system CPU would see, looking in from the outside, via an AXI4 slave interface. The addresses listed at the far right correspond to the addresses to each memory block described in the graphic and defines the address range the CPU would write/read to/from to access a given thread's program memory or parameter/data buffer via the AXI4 interface. Note that the two lower address lines are not used. This is because the Shader core requires 32-bit aligned data, in other words, all transfers from system memory to any of these blocks are four bytes—32-bit aligned.

Figure 2—2. AXI4 Slave Memory Map



Each Shader core has four threads. Each thread employs one or more multi-ported parameter/data buffers for receiving parameters and data (“packets”) from a system CPU, which are “pushed-in” via the integral AXI4 burst-mode slave interface. When processing is completed, a given thread will then set its DONE bit, thereby generating a CPU interrupt, signaling that results are available, at which time the results are “pulled-out” by the CPU via AXI4.

Also available for each AXI4 Shader is a relatively large, globally-mapped, Intermediate Result Buffer (IRB), shared by all four Shader threads. The IRB ais general-purpose and is especially useful for storing intermediate results involving processing of several packets.

For example, if a particular task requires the use of a texture image in the form of a compressed thumbnail, the CPU can push the compressed image into the corresponding IRB, then push parameters into the corresponding thread's parameter buffer instructing such thread that there is a compressed thumb in the IRB waiting to be decompressed and used by the other threads.

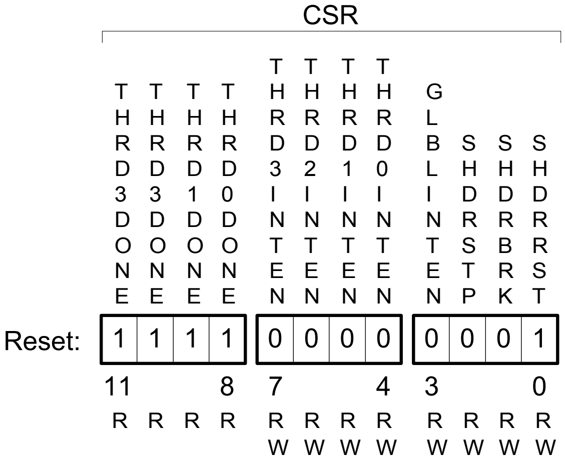
2.5.1 AXI4 Access to Shader Program Memory

Each Shader core has 4,096 (32-bit) words of program memory that can either be initialized by way of a given FPGA configuration memory, or loaded and/or modified by the CPU via the AXI4 interface as part of its system initialization. One strategy is to have all the routines that a Shader needs to perform any task requested of it already in its program memory. Another strategy is to initialize the program memory with only the routines used most often, minus 1k words, for example. With this later approach, the CPU has the flexibility to push the required routines into such memory, on-the-fly, if the required routine is not already in a Shader's program memory.

2.5.2 AXI4 Control/Status Register

Mapped at AXI4 system memory location 0xnnnbbb1_FFFC and depicted below is the AXI4 Control/Status Register (CSR).

Figure 2—3. AXI4 Control/Status Register



CSR

	THRD3DONE	THRD3DONE	THRD1DONE	THRD0DONE	THRD3INTEN	THRD2INTEN	THRD1INTEN	THRD0INTEN	GLBLINTEN	SHDRSTP	SHDRBRK	SHDRRST
Reset:	1	1	1	1	0	0	0	0	0	0	0	1
	11			8	7			4	3			0
	R	R	R	R	RW	RW	RW	RW	RW	RW	RW	RW

nnnbbb = Base Address

Logical Address: 0xnnnbbb1_FFFC

The CSR not only provides a means for the CPU to reset and/or release from reset the Shader core via the AXI4 interface, but also provides the CPU with a means for enabling/disabling interrupts in response to a given thread entering a DONE state. On power-up, the reset line of the Shader is set and held active until the CPU clears it. The CSR is presently only 12 bits wide, but when read by the CPU, it is read zero-extended to 32-bits. *Table 2—3* below gives a description for each bit in the AXI4 CSR register.

Table 2—3. Shader AXI4 Control Status Register (CSR)

Bit Position	Description
0	Shader reset, active high ("1").
1	Shader force break-point. Not yet implemented.
2	Shader single-step. Not yet implemented.
3	Shader global interrupt enable, active high. When set active high, will generate an active high Interrupt Request to the CPU when a thread's DONE flag goes

	active AND its corresponding Interrupt Enable bit is also set.
4	Thread 0 Interrupt Enable, active high.
5	Thread 1 Interrupt Enable, active high.
6	Thread 2 Interrupt Enable, active high.
7	Thread 3 Interrupt Enable, active high.
8	Thread 0 DONE status input, read-only.
9	Thread 1 DONE status input, read-only.
10	Thread 2 DONE status input, read-only.
11	Thread 3 DONE status input, read-only.

Chapter 3

Scheduling Overview

3.1 Thread Scheduling

As mentioned earlier, coarse-grained scheduling is performed by the CPU (preferably any mainstream RISC of your choice) acting as a load-balancer specifying, as a set of parameters included in the parameter/data buffer it pushes per transaction, precisely the fine-grained interleave granularity, program entry point for the required task, and other parameters a thread requires to carry out a task.

Fine-grained scheduling can be accomplished by either the hardware fine-grained scheduler or by the use of soft/self-scheduling capability built into each instruction when a given thread's LOCK bit is set, or by using the soft/self-scheduling capability in combination with the hardware fine-grained scheduling register. The hardware fine-grained scheduler has priority over soft scheduling, but can be disabled by setting the LOCK bit in a given thread's STATUS register, which disables the hardware fine-grained scheduler until the LOCK bit is cleared by the thread that set it as part of its routine, as might be specified in the parameters passed to it by the CPU.

If a given thread's LOCK bit is set, thread interleave is disabled until such time that the thread that set it, clears it. In such scenarios where a branch instruction is subsequently encountered in the instruction stream of a thread that has set its LOCK bit, the two instructions that follow the branch will be fetched and discarded when the branch is taken. To avoid these two fetches and discards of these two instructions, the soft-scheduling feature built into each instruction may be employed to switch to another thread on the very next clock cycle even though there is a lock on the present thread.

To illustrate this, consider the following instruction sequence, assuming the LOCK bit of that thread has previously been set.

```
bcnd    tr0_loop, !Z                            ;branch relative if not zero
add     work_A, work_A, #0x37
shft    work_C, @_max, RIGHT, 5
```

In the above example, if the fine-grained scheduler is disabled when thread0 fetches the BCND instruction, the ADD and SHFT instructions will be fetched before the branch is taken. This is due to the fact that thread interleave has been disabled by setting the thread's LOCK bit. Stated another way, all PC discontinuities are delayed, due to the pipeline and the fact that the branch instruction doesn’t actually execute until two clocks after the branch instruction is fetched.

In the above example, if the hardware scheduler were enabled (and LOCK bit cleared) with a granularity of one clock for thread0, this would not be a problem because the next thread in the interleave queue would automatically switch in to avoid the fetch of the ADD and SHFT in the current thread, which is one of the main ideas behind interleaving threads.

To employ soft/self-scheduling (when the LOCK bit is set), simply do this:

```
bcnd.2    tr0_loop, !Z                            ;branch relative if not zero
add       work_A, work_A, #0x37
shft      work_C, @_max, RIGHT, 5
```

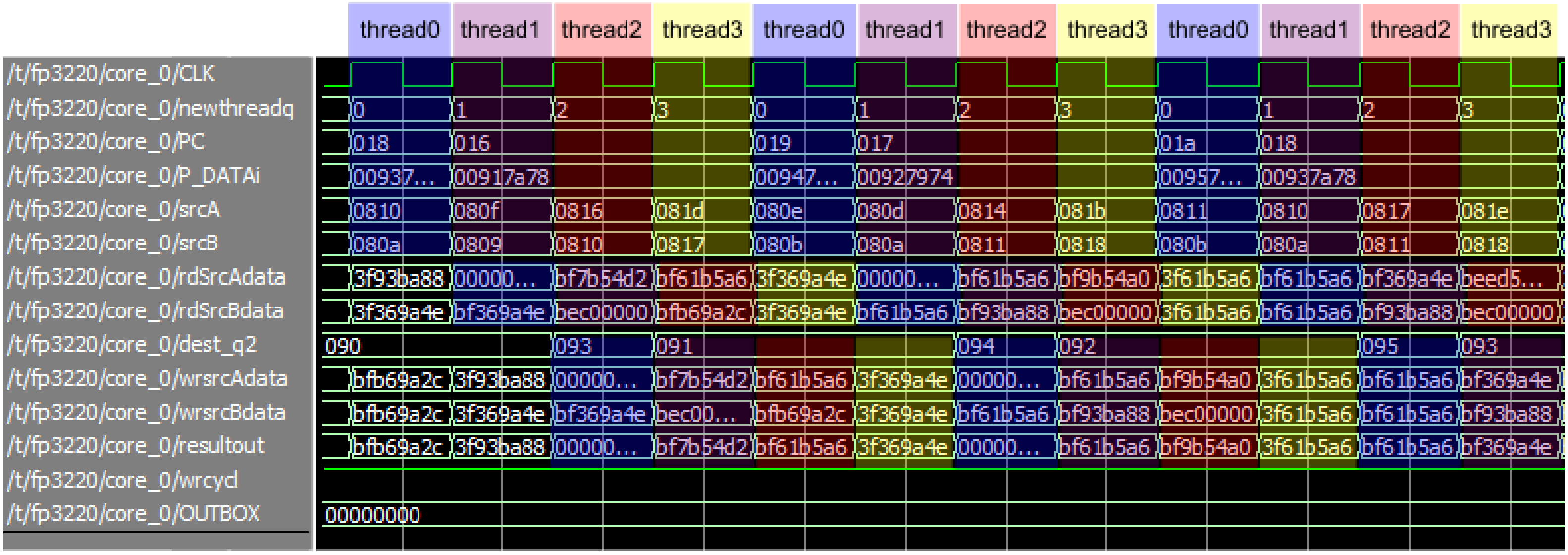
In the above example, the “.2” tells the core to switch to thread2 for the next fetch. When executed, the next instruction is fetched by thread2 instead of the current thread. If the new thread and the other remaining threads do not also have their LOCK bits set, interleave again becomes active and the time slot for the originating thread will eventually work its way back around to the original thread that executed the soft-schedule and continue in the locked state until another soft-schedule occurs or the lock bit is clear, at which time interleave resumes.

The example in *Figure 3--1* shows actual scheduling behavior of a single Shader running four threads (threads 0 through 3) with a interleave granularity of one clock each. Notice the multistage pipeline, which comprises a instruction fetch, operand read, and result write cycle. Note that without a write feed-though on the destination register or memory location, the data being written will not be available for reading until the third clock after the fetch of the instruction that caused the write. To be on the safe side, use write feed-through on your memory and registers. The original library published at the SYMPL FP32X-AXI4 repository at GitHub includes generic synchronous memory block modules that can be easily configured with or without write feed-through, at your option.

In the following examples, "newthreadq" is the current thread and "P_DATAi" is the instruction being fetched. "PC" is the fetch address. "srcA" and "srcB" are the operand addresses that are registered into the synchronous RAMs or register at the end of the respective instruction fetch cycle.

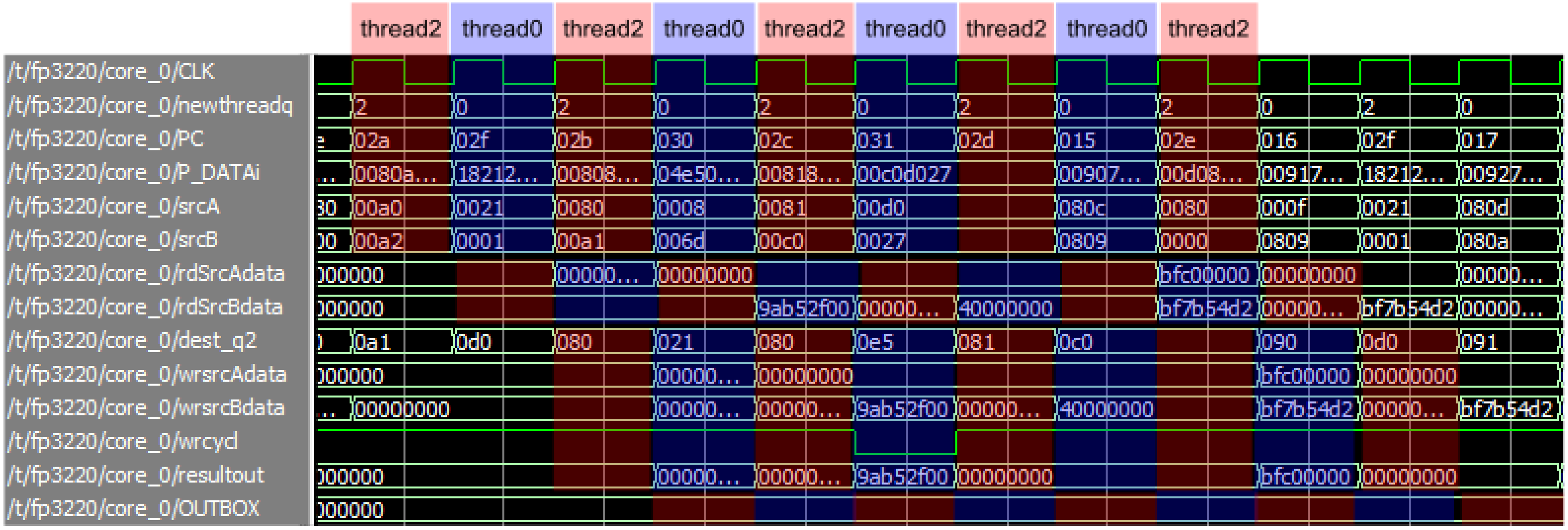
To configure a given Shader for four threads with interleave granularity of one clock, simply load the hardware fine-grained scheduling register at location 0x06C with the value 0x04040404. This programs each thread's clock counter to 04, which is the number of clocks that must transpire before the corresponding thread's next time slot becomes available.

Figure 3—1. Interleave Behavior for Four Threads with One-Clock Granularity


thread0 thread1 thread2 thread3 thread0 thread1 thread2 thread3 thread0 thread1 thread2 thread3
/t/fp3220/core_0/CLK
/t/fp3220/core_0/newthreadq 0 1 2 3 0 1 2 3 0 1 2 3
/t/fp3220/core_0/PC 018 016 019 017 01a 018
/t/fp3220/core_0/P_DATAi 00937... 00917a78 00947... 00927974 00957... 00937a78
/t/fp3220/core_0/srcA 0810 080f 0816 081d 080e 080d 0814 081b 0811 0810 0817 081e
/t/fp3220/core_0/srcB 080a 0809 0810 0817 080b 080a 0811 0818 080b 080a 0811 0818
/t/fp3220/core_0/rdSrcAdata 3f93ba88 00000... bf7b54d2 bf61b5a6 3f369a4e 00000... bf61b5a6 bf9b54a0 3f61b5a6 bf61b5a6 bf369a4e beed5...
/t/fp3220/core_0/rdSrcBdata 3f369a4e bf369a4e bec00000 bfb69a2c 3f369a4e bf61b5a6 bf93ba88 bec00000 3f61b5a6 bf61b5a6 bf93ba88 bec00000
/t/fp3220/core_0/dest_q2 090 093 091 094 092 095 093
/t/fp3220/core_0/wrsrcAdata bfb69a2c 3f93ba88 00000... bf7b54d2 bf61b5a6 3f369a4e 00000... bf61b5a6 bf9b54a0 3f61b5a6 bf61b5a6 bf369a4e
/t/fp3220/core_0/wrsrcBdata bfb69a2c 3f369a4e bf369a4e bec00... bfb69a2c 3f369a4e bf61b5a6 bf93ba88 bec00000 3f61b5a6 bf61b5a6 bf93ba88
/t/fp3220/core_0/resultout bfb69a2c 3f93ba88 00000... bf7b54d2 bf61b5a6 3f369a4e 00000... bf61b5a6 bf9b54a0 3f61b5a6 bf61b5a6 bf369a4e
/t/fp3220/core_0/wrcycl
/t/fp3220/core_0/OUTBOX 00000000

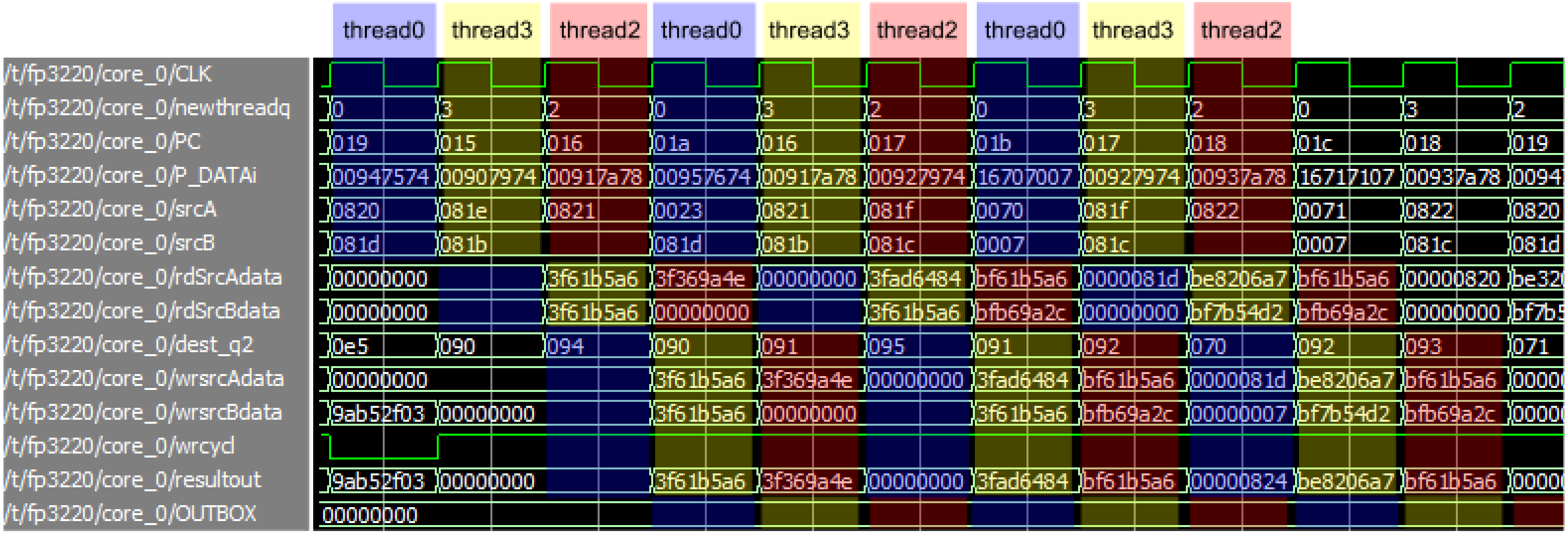
The example below shows actual scheduling behavior of a single Shader running just two threads (threads 0 and 2) with a interleave granularity of one clock each. To configure threads 0 and 2 for one-clock interleave, load the value 0x00020002 into the scheduling register.

Figure 3—2. Interleave Behavior for Three Threads with One-Clock Granularity



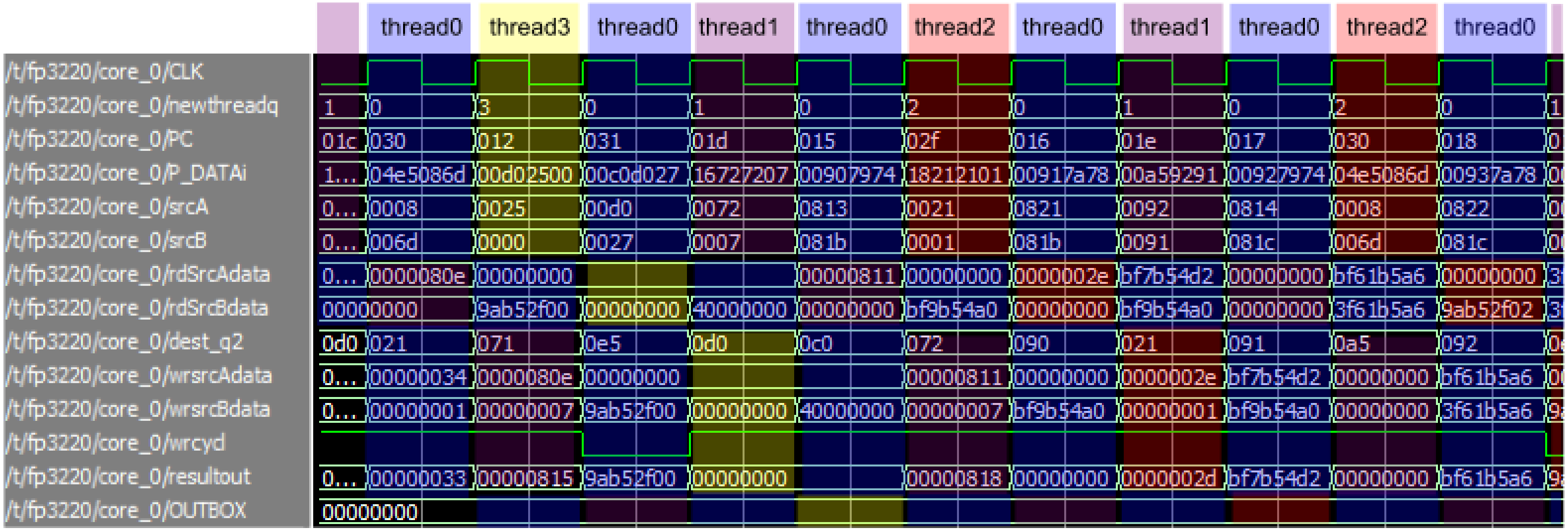
Threads 0, 2 and 3 with one-clock interleave: load scheduling register with 0x03030003. Note that thread1 has been completely de-scheduled by writing "00" to its fine-grained scheduler compare register.

Figure 3—3. Non-Sequential Interleave with One-Clock Granularity



For four-thread asymmetric interleave with one-clock granularity, with thread 3 (5% duty-cycle), thread 2 (20%), thread 1 (25%), and :thread 0 (50%): load scheduling register with 0x14050402.

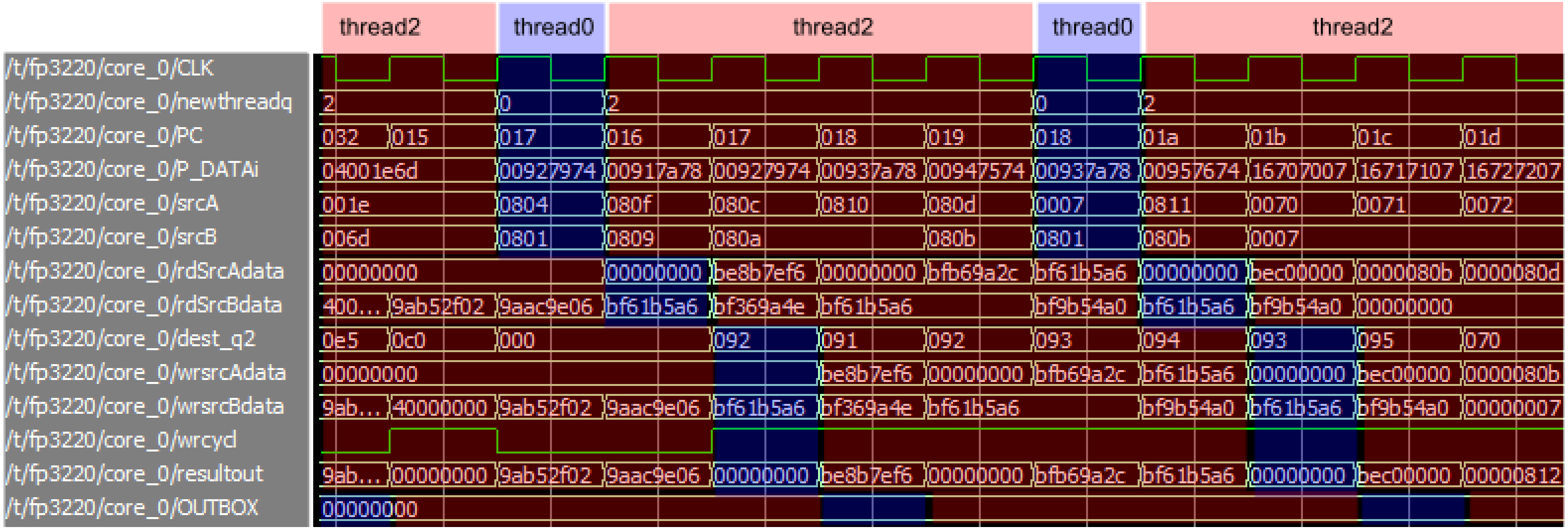
Figure 3—4. Asymmetric Interleave for Three Threads with One-Clock Granularity



```
                          thread0  thread3  thread0  thread1  thread0  thread2  thread0  thread1  thread0  thread2  thread0
/t/fp3220/core_0/CLK
/t/fp3220/core_0/newthreadq  1  0  3  0  1  0  2  0  1  0  2  0  1
/t/fp3220/core_0/PC  01c  030  012  031  01d  015  02f  016  01e  017  030  018  0
/t/fp3220/core_0/P_DATAi  1...  04e5086d  00d02500  00c0d027  16727207  00907974  18212101  00917a78  00a59291  00927974  04e5086d  00937a78  0
/t/fp3220/core_0/srcA  0...  0008  0025  00d0  0072  0813  0021  0821  0092  0814  0008  0822  0
/t/fp3220/core_0/srcB  0...  006d  0000  0027  0007  081b  0001  081b  0091  081c  006d  081c  0
/t/fp3220/core_0/rdSrcAdata  0...  0000080e  00000000  00000811  00000000  0000002e  bf7b54d2  00000000  bf61b5a6  00000000  3
/t/fp3220/core_0/rdSrcBdata  00000000  9ab52f00  00000000  40000000  00000000  bf9b54a0  00000000  bf9b54a0  00000000  3f61b5a6  9ab52f02  3
/t/fp3220/core_0/dest_q2  0d0  021  071  0e5  0d0  0c0  072  090  021  091  0a5  092  0
/t/fp3220/core_0/wrsrcAdata  0...  00000034  0000080e  00000000  00000811  00000000  0000002e  bf7b54d2  00000000  bf61b5a6  0
/t/fp3220/core_0/wrsrcBdata  0...  00000001  00000007  9ab52f00  00000000  40000000  00000007  bf9b54a0  00000001  bf9b54a0  00000000  3f61b5a6  9
/t/fp3220/core_0/wrcycd
/t/fp3220/core_0/resultout  0...  00000033  00000815  9ab52f00  00000000  00000818  00000000  0000002d  bf7b54d2  00000000  bf61b5a6  9
/t/fp3220/core_0/OUTBOX  00000000
```

For two-thread asymmetric interleave with five-clock granularity, with thread 0 (20% duty-cycle), thread 2 (80%), load the fine-grained scheduler with 0x00020005.

Figure 3—5. Asymmetric Interleave for Three Threads with One-Clock Granularity



```
                          thread2  thread0  thread2  thread0  thread2
/t/fp3220/core_0/CLK
/t/fp3220/core_0/newthreadq  2  0  2  0  2
/t/fp3220/core_0/PC  032  015  017  016  017  018  019  018  01a  01b  01c  01d
/t/fp3220/core_0/P_DATAi  04001e6d  00927974  00917a78  00927974  00937a78  00947574  00937a78  00957674  16707007  16717107  16727207
/t/fp3220/core_0/srcA  001e  0804  080f  080c  0810  080d  0007  0811  0070  0071  0072
/t/fp3220/core_0/srcB  006d  0801  0809  080a  080b  0801  080b  0007
/t/fp3220/core_0/rdSrcAdata  00000000  00000000  be8b7ef6  00000000  bfb69a2c  bf61b5a6  00000000  bec00000  0000080b  0000080d
/t/fp3220/core_0/rdSrcBdata  400...  9ab52f02  9aac9e06  bf61b5a6  bf369a4e  bf61b5a6  bf9b54a0  bf61b5a6  bf9b54a0  00000000
/t/fp3220/core_0/dest_q2  0e5  0c0  000  092  091  092  093  094  093  095  070
/t/fp3220/core_0/wrsrcAdata  00000000  be8b7ef6  00000000  bfb69a2c  bf61b5a6  00000000  bec00000  0000080b
/t/fp3220/core_0/wrsrcBdata  9ab...  40000000  9ab52f02  9aac9e06  bf61b5a6  bf369a4e  bf61b5a6  bf9b54a0  bf61b5a6  bf9b54a0  00000007
/t/fp3220/core_0/wrcycd
/t/fp3220/core_0/resultout  9ab...  00000000  9ab52f02  9aac9e06  00000000  be8b7ef6  00000000  bfb69a2c  bf61b5a6  00000000  bec00000  00000812
/t/fp3220/core_0/OUTBOX  00000000
```

Chapter 4

Programmer's Reference

4.1 Overview

The SYMPL FP32X-AXI4 presently employs five addressing modes for most of its instructions: direct (zero-page), indirect with variable auto post-increment/decrement, 8-bit immediate, 16-bit immediate, table-read-direct, and table-read-indirect from program memory. All floating-point operators (if present in a given implementation) are memory-mapped and are accessible via a given thread's private zero-page memory space (locations 0x0080 through 0x00FF) using the dual-operand MOV instruction for operators that have dual inputs.

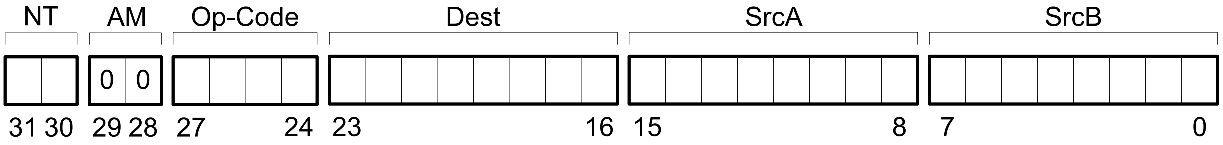
4.2 Instruction Word Format

All instructions are 32-bits wide. Presently, there are six different formats for use by specific instructions. The formats, along with descriptions of the instructions that apply, are shown in the following diagrams.

4.3 Direct and Indirect Addressing Mode

Direct and Indirect addressing mode use the same field format, comprising both SrcA and SrcB 8-bit fields as the source addresses for dual-operand read, along with an 8-bit Dest field as the destination address. For single-source, direct or indirect read operations, only the SrcA field is used. The diagram in *Figure 4—1* below shows the 32-bit instruction word format for direct and indirect addressing modes.

Figure 4—1. Direct/Indirect Addressing Mode Field Formatting



*Applies to: MOV, AND, OR, XOR, ADD, ADDC, SUB, SUBB, and MUL.
With SrcA only: MOV, RCP, SIN, COS, TAN and COT.

4.3.1 Direct Addressing Mode

Direct (zero-page) addressing may be used for accessing any location within the range 0x0000 through 0x00FF and is available for use in the fields labeled “Dest”, “SrcA”, and “SrcB” in the instruction word diagram in *Figure 4—1* above. Direct mode may be used on the same assembly line in any combination with other addressing modes, but only in fields designated “Dest”, “SrcA”, or “SrcB”.

4.3.2 Indirect Addressing Mode

Indirect addressing mode employs a given thread's auxiliary registers (presently AR0 through AR3) to access any location within a given thread's entire memory space, including zero-page, private and global, AXI4 parameter/data buffer, and program table-read. Each of the auxiliary registers may be automatically post-incremented or post-decremented.

Indirect addressing is available for use in the fields labeled “Dest”, “SrcA”, and “SrcB” in the instruction word diagrams in the Instruction Word Format section above. Prefixing any of the auxiliary register identifiers with the “*” symbol signals the assembler to use indirect mode. Example: *AR0.

Postfixing any of the auxiliary register identifiers with either the “++” or “--” (along with the “*” prefix) will cause that particular ARn to be automatically incremented or decremented by 1 after use. For example, *AR2++.

Indirect mode may be used on the same assembly line in any combination with other addressing modes, but only in fields designated “Dest”, “SrcA”, or “SrcB” as shown in the above diagrams.

If SrcA and or SrcB are within the range of 0x7F to 0x74, then this specifically implies that the indirect addressing mode is being used for the read operation. This is also true for the Dest (destination) field for the write cycle.

The table below shows the indirect addressing mode mnemonics and corresponding direct addresses associated with them that both signal the assembler that the indirect addressing mode is to be used and the type of indirect addressing that is to take place for either SrcA and/or SrcB.

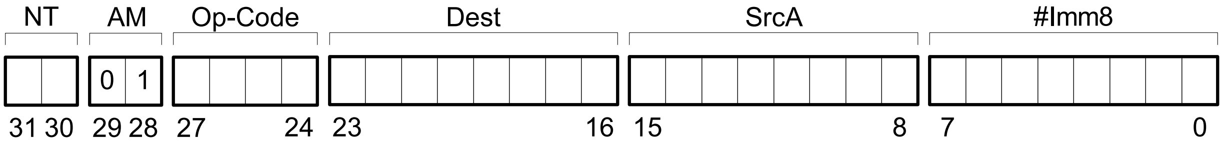
Table 4—1. Auxiliary Register Address-Modifier Translation

SrcA/SrcB/Dest Field Before Translation	SrcA/SrcB/Dest Mnemonic	Description
0x74	*AR0	AR0 is indirect pointer with no automatic post-modification
0x75	*AR0++	AR0 is indirect pointer with automatic post-increment
0x76	*AR0--	AR0 is indirect pointer with automatic post-decrement
0x77	*AR1	AR1 is indirect pointer with no automatic post-modification
0x78	*AR1++	AR1 is indirect pointer with automatic post-increment
0x79	*AR1--	AR1 is indirect pointer with automatic post-decrement
0x7A	*AR2	AR2 is indirect pointer with no automatic post-modification
0x7B	*AR2++	AR2 is indirect pointer with automatic post-increment
0x7C	*AR2--	AR2 is indirect pointer with automatic post-decrement
0x7D	*AR3	AR3 is indirect pointer with no automatic post-modification
0x7E	*AR3++	AR3 is indirect pointer with automatic post-increment
0x7F	*AR3--	AR3 is indirect pointer with automatic post-decrement

4.4 8-Bit Immediate Addressing Mode

The diagram below shows the instruction word format for 8-bit immediate addressing mode.

Figure 4—2. 8-Bit Immediate Addressing Mode Field Formatting



*Applies to: MOV, AND, OR, XOR, ADD, ADDC, SUB, SUBB, and MUL (dual operands only).

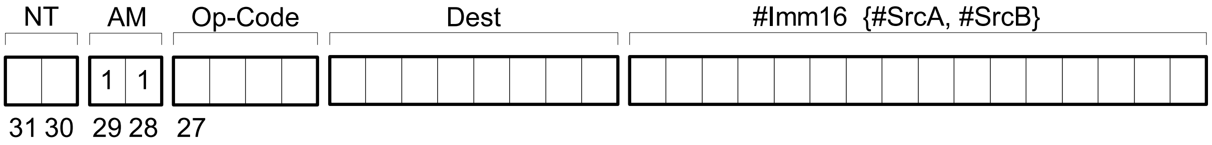
8-bit immediate mode utilizes the 8-bit constant constituting the immediately available last eight bits of the current instruction (i.e., the SrcB field only) without having to first retrieve it from data memory. The 8-bit value appearing in the #Imm8 field (SrcB position) is automatically zero-extended to 32 bits before use. Prefixing the immediate literal or label with the "#" sign signals the assembler to use 8-bit immediate mode for SrcB. 8-bit immediate mode is only available for use in the SrcB field.

Attempts to use 8-bit immediate mode in the SrcA field along with a SrcB value, will flag an error by the assembler. 8-bit immediate mode may not be used in combination with table-read-direct addressing mode described below. Stated another way, use of both the "#" and the "@" sign on the same assembly line is not permitted. 8-bit immediate mode for SrcB signals the assembler to set bit 28 of the instruction word to "1".

4.5 16-Bit Immediate Addressing Mode

The diagram below shows the instruction field format for 16-bit immediate addressing mode.

Figure 4—3. 16-Bit Immediate Addressing Mode Field Formatting



*Applies to: MOV, RCP, SIN, COS, TAN and COT only.

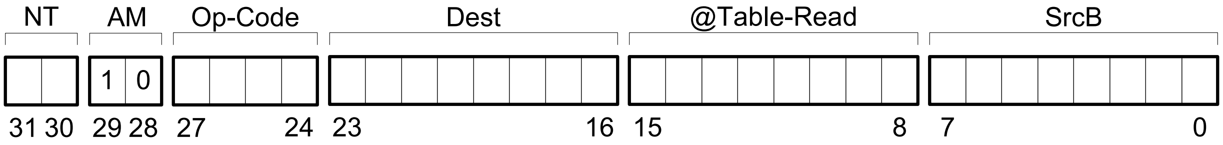
16-bit immediate mode utilizes the 16-bit constant constituting the immediately available last sixteen bits of the current instruction word without having to first retrieve it from data memory. The 16-bit value is automatically zero-extended to 32 bits before use.

Preceding the immediate literal or label with the "#" sign signals the assembler to set both bits 29 and 28 of the instruction word to "11". 16-bit immediate mode is only available for use with the MOV, RCP, SIN, COS, TAN and COT instructions only. Attempts to use 16-bit immediate mode with any other instruction will flag an error by the assembler.

4.6 Table-Read-Direct Addressing Mode

The diagram below shows the instruction field format for table-read-direct addressing mode.

Figure 4—4. Table-Read-Direct Addressing Mode Field Formatting



*Applies to: MOV, AND, OR, XOR, ADD, ADDC, SUB, SUBB, and MUL.
With SrcA only: MOV, RCP, SIN, COS, TAN and COT.

Table-read-direct mode works just like the direct mode described previously, except it is used to access constants that may be present in the first 256 locations of program memory. Preceding the location literal or label with the “@” sign signals the assembler that SrcA is a table-read from program memory and will cause the assembler to set bit 29 of the instruction word. Table-read-direct addressing mode is only available for use in the SrcA field. Table-read-direct addressing mode may not be used in combination with the 8-bit immediate addressing mode described above. Stated another way, use of both the “@” and the “#” sign on the same assembly line is not permitted. “@” in SrcA field may be used in combination with direct or indirect in SrcB field.

4.7 Table-Read-Indirect Addressing Mode

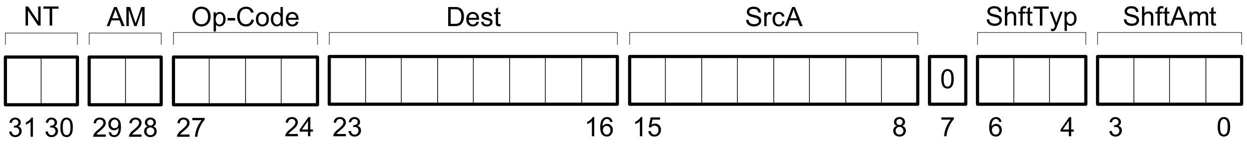
Table-read-indirect mode works just like the indirect addressing mode described previously, except it is used to access constants (or copy entire threads to data memory, for example) that may be present at any location in program memory. To employ this feature, set bit 12 of the auxiliary register used to access program memory. Stated another way, a given core's entire program memory is mapped into the core's data memory map starting at location 0x1000 and continues up to 0x1FFF and the only way to reach program memory locations above 0x00FF is by using the indirect addressing mode. The “@” symbol is reserved for use by the program memory (zero-page) table-read-direct mode only.

The table-read-indirect mode may be used on the same assembly line in any combination with other addressing modes, but only in the field designated “SrcA”.

4.8 SHFT Instruction Field Format

The diagram below shows the field formatting for use by the SHFT instruction only. For more information on the SHFT instruction, refer to Section 5.3.15, which describes it in more detail.

Figure 4—5. SHFT Instruction Field Formatting

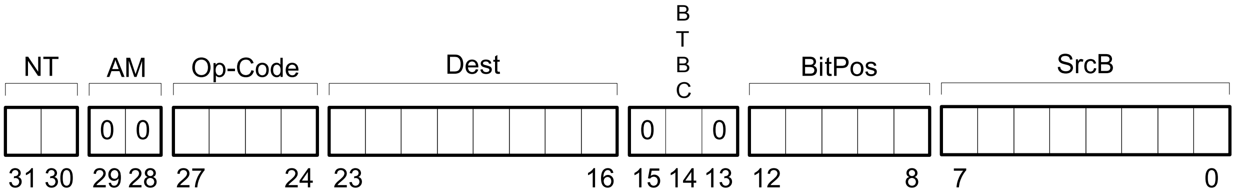


*Applies to SHFT only.

4.9 BTBS, BTBC and DBNZ Instruction Field Format

Figure 4—6 below shows the field formatting used by the bit-test-and-branch-if-set (BTBS), bit-test-and-branch-if-clear (BTBC), and decrement-and-branch-if-zero (DBNZ) instructions. For more detailed information about these two instructions, refer to Section 5.3 respective sections that describes them. If bit 14 of the bit-test-and-branch instruction is set to 1, then the instruction tests the bit specified in BitPos for the cleared state rather than the set state for the "true" condition under which a branch will be taken during execution.

Figure 4—6. BTBS, BTBC and DBNZ Instruction Field Formatting



4.10 Instruction Word Dis-Assembled

4.10.1 Next-Thread Field Description

NT / RM **Description**

D[31:30] The “NT” field is dual-purpose. For all instructions other than MOV, the NT field means “next thread”, which is used for soft-scheduling. For MOV instructions whose destination address lies outside the floating-point operator input register range (0x00FF through 0x0080), these bits serve the same purpose as all the other rinstructions. Conversely, if the destination address lies within and inclusive of said range, these two bits become the “RoundMode” field used for specifying which of the four rounding modes to use for a given floating-point operation. By default, the assembler fills these two bits with “00” to specify the default rounding mode of “nearest”. Ordinarily, the MOV instruction is used to write the operand(s) to the desired floating-point operator's input register(s). To specify a rounding mode other than the default nearest rounding mode, simply place a “.n” next to the MOV instruction, wherein “n” is either “.1” (for positive infinity), “.2” (for negative infinity or “.3” (for round to zero). Here are some examples:

```
MOV     FDIV_3, oprndA, oprndB     ;divide oprndA by oprndB and round to nearest
MOV.1   SQRT_1, oprndA             ;square root oprndA and round to positive infinity
MOV.2   FADD_4, oprndA, oprndB     ;add oprndB to oprndA and round to negative infinity
MOV.3   FMUL_7, oprndA, oprndB     ;mult oprndA by oprndB and round to zero
```

When these two bits are not used for specifying rounding mode by the MOV instruction, they can be used for soft-scheduling. These two bits are normally filled with “00” to show that there will NOT be a soft-schedule thread change on the next clock cycle. These two bits do not indicate the currently executing thread. Instead, they indicate which thread will become active for the next instruction fetch (next clock). If the value is “00” in the current instruction, no soft-schedule thread switch will occur on the next clock cycle.

The assembler permits specifying on the assembly line which, if any, soft-scheduled thread switch will become active on the next clock cycle. If the hardware scheduler is enabled and a hardware scheduler event is pending, the hardware scheduler will have priority over soft-scheduling and will over-ride any soft-schedule event occurring on the same fetch.

For example, if thread 0 is the current thread, and a “11”, for example, appears in the NT field, then this will cause the Shader to instantaneously switch to thread 3 on the next clock cycle. A “10” will cause the Shader to switch to thread 2, and so on, assuming no hardware fine-grained-scheduled thread switch is pending at that time. If thread 1, 2, or 3 is the current thread, to switch back to thread 0, a 2-bit value equal to the current thread will cause the Shader to switch back to thread 0. For example, say the current thread is thread 2, a value of “10” appearing in the NT field indicates that the core will switch to thread 0 on the next clock cycle. The value “00” in the NT field has no effect on soft-scheduling of threads, regardless of which thread is currently active. Example:

```
ADD.2 result1, *AR1++, #5        ;add 8-bit immediate value of 5 to the contents of
                                 ;the address pointed to by AR1 then post-
                                 ;increment the contents of AR1 by 1 and store
                                 ;result of the addition in direct memory address
                                 ;result1, then, if not already in thread 2, soft-
                                 ;schedule a switch to thread 2 for the next clock
                                 ;cycle, otherwise, if already in thread 2, soft-
                                 ;schedule a switch back to thread 0 for the next
                                 ;instruction fetch.
```

4.10.2 Addressing Mode Field Description

AddrsMode **Description**

D[29:28] The “Address Mode” field indicates which address mode is to be used for the current instruction. They are given as follows:

“00” Direct or indirect SrcA (and SrcB if present). Example:

```
ADD     blue, green, red          ;add red to green and store result in blue
SUB     count, count, *AR3        ;subtract from the contents of count the
                                  ;contents of the address pointed to by AR3 and
                                  ;store result in count
```

“01” Direct or indirect SrcA and 8-bit immediate as SrcB. Example:

```
XOR   *AR3++, *AR3, #mask1          ;xor contents pointed to by AR3 with mask1
```

“10” Table-Read-Direct from page-zero of program memory for SrcA only and direct or indirect for SrcB (if present). Use of 8-bit immediate #SrcB on the same assembly line with @SrcA is not permitted. Valid example:

```
AND    test1, @constant, *AR1         ;AND the constant in ROM with the  contents
                                      ;of the address pointed to by AR1, store
                                      ;result in test1
```

“11” 16-bit immediate SrcA. Only available for use with the MOV instruction. Example:

```
MOV   private1, #0x1234
```

4.10.3 Op-Code Field Description

The Shader cores presently have only sixteen actual op-codes. The op-codes and their respective function are described below:

OpCode **Mnem** **Description**

D[27:24]

“0000” MOV Move SrcA (and SrcB if present) to Dest. If SrcB is present, this implies that the MOV is being used for a floating-point operation that requires two operands, OprndA and OprndB. If SrcB is absent “and” SrcA is immediate, the immediate value is taken as #immed16 instead of #immed8, zero-extended to 32 bits.

“0001” AND Logically AND SrcA with SrcB, store result in Dest.

“0010” OR Logically OR SrcA with SrcB, store result in Dest.

“0011” XOR Logically XOR SrcA with SrcB, store result in Dest.

“0100” BTB Bit-test specified bit postion (0 - 31) of the contents of SrcB and branch relative (+127 to -128) from the location where the BTB instruction was fetched if set. Note that BCND (branch conditionally) is an alias of BTB and implies SrcB is the current thread’s STATUS register. Aliases BTBS and BTBC use the same op-code as BTB, but for BTBC (bit-test and branch if Cleared), bit 14 of the instruction is set (so it can be XORed by internal logic to achieve the “if cleared” condition). Decrement-and-branch-if-not-zero (DBNZ) is also an alias of BTB. Example BTBs/BCNDs:

```
wait:   BTBS   wait, 29, contents     ;test bit 29 of contents and wait if bit = 1
        BCND   zoom, ALWAYS           ;unconditional relative branch
        BCND   wait, NEVER            ;functionally same as NOP
zoom:   BCND   somewhere, Z           ;branch if Z flag set
        MOV    LPCNT0, #33
delay:  DBNZ   delay, LPCNT0          ;decrement LPCNT0 by 1 until zero
```

“0101” SHFT Barrel-shift SrcA from 1 to 16 bits, using both shift-type specifier and shift-amount specifier, then store result in destination. Except where noted, Z, N, and C flags are affected. V flag remains unchanged. SHFT provides seven different types of shifts shown as follows:

D[6:4]	**ShfTyp**	
“000”	LEFT	Carry flag is unaffected. LSBs filled with “0”. Same as ASL.
“001”	LSL	Logical shift left through Carry. “0” shifted in through LSB.
“010”	ASL	Arithmetic shift left. LSBs filled with “0”. Same as LEFT above.
“011”	ROL	True barrel shift left. Bits shifted out of MSB are shifted in through LSB. Carry flag is unaffected.
“100”	RIGHT	Shift right. Carry flag is unaffected. MSBs filled with “0”.
“101”	LSR	Logical shift right. LSB shifted through Carry. “0” shifted through MSB.
“110”	ASR	Arithmetic shift right. MSB does not change and is copied ShftAmt times. Carry is unaffected.
“111”	ROR	True barrel shift right. Bits shifted out of LSB are shifted in through MSB. Carry is unaffected.

D[3:0] **ShftAmt**

“0000” through “1111” Encoded shift amount. “0000” means shift by 1. “1111” means shift by “16”. The assembler encodes ShftAmt by subtracting “1” from the value appearing on the assembly line. Example:

```
SHFT   result2, vectx, ASR, 3       ;divide vectx by 8 and sign-extend
```

“0110” ADD Integer add (without carry) SrcB to SrcA, store result in Dest.

“0111” ADDC Integer add (with carry) SrcB to SrcA, store result in Dest.

“1000” SUB Integer subtract (without barrow) SrcB from SrcA, store result in Dest.

“1001” SUBB Integer subtract (with barrow) SrcB from SrcA , store result in Dest.

“1010” MUL Perform 16 x 16 integer multiply (SrcA x SrcB), store result in Dest.

“1011” RCP Performs 1/n (reciprocal) of SrcA (integer) and returns 32-bit floating-point result. SrcA must be a signed integer in the range +127 to -128. Data bits D[31:8] of SrcA are ignored. Input of SrcA == 0 returns 0x7F800000. Results of the RCP can be used in combination with the FMUL operator to perform quick floating-point divides in five clocks in lieu of the FDIV operator, which takes 15 clocks to complete.

"1100" SIN Accepts integer SrcA in range of +/- 360 degrees and stores sine (32-bit float) in Dest.

```
SIN   rot_x, #25          ;get sine of 25 degrees and store in rot_x
```

"1101" COS Accepts integer SrcA in range of +/- 360 degrees and stores cosine in Dest.

"1110" TAN Accepts integer SrcA in range of +/- 360 degrees and stores tangent in Dest.

"1111" COT Accepts integer SrcA in range of +/- 360 degrees and stores cotangent in Dest.

Shader Register Set

Chapter 5

The SYMPL FP32X comprises four threads, each with its own set of private registers and private memory mapped in its zero-page (address range 0x00FF to 0x0000). The threads also have access to (share) a few global registers. These registers are listed in the table below and are described in the Sections that follow.

Table 5—1. Thread Register Set

Name	Address	Description
AR3	0x0073	Auxiliary register 3—used for indirect addressing mode
AR2	0x0072	Auxiliary register 2—used for indirect addressing mode
AR1	0x0071	Auxiliary register 1—used for indirect addressing mode
AR0	0x0070	Auxiliary register 0—used for indirect addressing mode
PC	0x006F	Program counter
PC Copy	0x006E	Holds a copy of the latest PC increment (but not write)--for return address
Status	0x006D	Status register
FGS	0x006C	Fine-grained scheduler clock counter
FGSC	0x006B	Fine-grained scheduler counter compare (max-count) register
C	0x006A	Fused-multiply-add (FMA) “C” register must be loaded with C before writing (X, Y) to FMA operator input register
LPCNT1	0x0069	Loop-counter 1 register for use with DBNZ instruction
LPCNT0	0x0068	Loop-counter 0 register for use with DBNZ instruction
TIMER	0x0067	Timer for limiting the amount of time a thread has available before timing out and generating a non-maskable interrupt
QOS	0x0066	Quality-of-service floating-point exception counter counts the number of times invalid, divide-by-zero, overflow and underflow are signaled
DOT	0x0065	Dot (sum-of-products) operator dual-operand input register and accumulator output register
RPT	0x0064	Repeat-counter
CAPT3	0x0063	Capture register 3—captures rounding mode, thread number, PC, and destination address during initiation of alternate delayed exception handling
CAPT2	0x0062	Capture register 2—captures exception code B, source address B, exception code A, and source address B during initiation of alternate delayed exception
CAPT1	0x0061	Capture register 1—captures floating-point result buffer data B during initiation of alternate delayed exception handling
CAPT0	0x0060	Capture register 0——captures floating-point result buffer data A during initiation of alternate delayed exception handling

5.1 Private Registers

Each of the four threads of a given Shader has its own program counter (PC), status register, auxiliary registers, loop-counters and timer, each memory-mapped in the threads' private, zero-page memory space. These private registers are accessible only by the thread to which they belong and are described below.

5.1.1 Program Counter

Figure 5—1. Program Counter

	PC											
Reset:	0	0	0	1	0	0	0	0	0	0	0	0
	11											0

Logical Address: 0x006F

Each Shader has four, 12-bit program counters (”PC”s), one for each thread, for a total program reach of 4,096 words each. Each thread’s PC resides at private memory location 0x06F and is both readable and writable. On reset, the PCs are reset to 0x100, which the thread's reset vector location where it makes its first instruction fetch. Ordinarily, the instruction residing at the reset vector location should contain a MOV PC, #Start instruction, where Start is the program address of that thread's “spin-idle” routine that polls its parameter/data buffer semaphore, which indicates that the CPU has pushed a packet into the buffer for processing.

While any instruction may be used to load the PC register (and thereby effectuate a jump, branch, call, return from subroutine or interrupt), it is best practice to simply use the MOV instruction for that purpose, especially when returning from an interrupt service routine. For more information regarding this, refer to Chapter 7 of this document, which covers interrupt vectors and service routines.

5.1.2 PC Copy Register

Figure 5—2. PC Copy Register

	PC Copy											
Reset:	0	0	0	1	0	0	0	0	0	0	0	0
	11											0

Logical Address: 0x006E

Each core has four, 12-bit PC Copy registers, one for each thread, residing in each thread’s private memory map at location 0x06E. It is both readable and writable. Anytime a thread’s PC register is written to by any instruction, a copy of that thread’s PC (+ 1) at the time the instruction was fetched is stored in that thread’s PC Copy register. Ordinarily, PC Copy is used for restoring the return address at the end of subroutine calls and interrupt service routines and therefore, should be among the first items saved to memory if nested calls and interrupts are employed.

When employed during an interrupt service routine, the best practice to immediately save the PC Copy register's contents at the locations specifically reserved for it in private memory, starting at location 0x0001 for NMI interrupts, 0x0002 through 0x0006 for invalid operation, divide-by-zero, overflow, underflow-inexact and inexact respectively, and 0x0007 for general-purpose IRQ interrupts. The main reason for this that there is no dedicated return-from-interrupt (RETI) instruction in the Shader's repertoire. To overcome this, there is dedicated logic in the interrupt controller circuit that specifically monitors a read from the corresponding private RAM address to clear the corresponding interrupt in-service signal that is used by the interrupt prioritizer. For more information regarding interrupts, refer to Section 7.2.

5.1.3 STATUS Register

Figure 5—3. STATUS Register

	ALWAYS	NEVER	Z||N							IRQPEND	IRQEN	DELNXCT	DELUNFL	DELOVFL	DELDIV0	DELNVLD
Reset:	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	R	R	R	R	R	R	R	R	R	R	R W	R W	R W	R W	R W	R W

	ALTNXCT	ALTUNFL	ALTOVFL	ALTDIV0	ALTNVLD	INEXCT	UNDERFLW	OVERFLW	DIVBY0	INVALID	DONE	LOCK	V	N	C	Z
Reset:	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W	R W

Logical Address: 0x006D

Each bit in the STATUS register is described in the following table.

Table 5—2. STATUS Register Bit Descriptions

Mnemonic	Bit Position	Description
Z	0	Zero flag—set to "1" if result is 0, reset to "0" if result is not 0.
C	1	Carry flag—set if carry occurred out of MSB of result, reset otherwise.
N	2	Negative flag—set if MSB of result is 1, reset otherwise.
V	3	Overflow flag—set if overflow occurred, reset otherwise.
LOCK	4	Thread lock bit. If set, no interleave will occur while in current thread.
DONE	5	Thread Done bit. Used to signal CGS/CPU done or busy state.
INVALID	6	Floating-point "invalid" operation exception flag. Under default handling, this flag is automatically set to "1" whenever an invalid operation is attempted and remains set until cleared by implementer. Under alternate exception handling, this flag is not affected unless maybe by the implementer's alternate exception handling routine.
DIVBY0	7	Floating-point "divide-by-zero" exception flag. Under default handling, this flag is automatically set to "1" whenever a divide-non-zero-by-zero is attempted (such as FDIV(n/0) or LOG(0)) and remains set until cleared by implementer. Under alternate exception handling, this flag is not affected unless maybe by the implementer's alternate exception handling

		routine.Floating-point “invalid” operation flag. Under default handling, this flag is automatically set to “1” whenever an invalid operation is attempted and remains set until cleared by implementer. Under alternate exception handling, this flag is not affected unless maybe by the implementer's alternate exception handling routine.
OVERFLW	8	Floating-point “overflow” exception flag. Under default handling, this flag is automatically set to “1” whenever floating-point operations on finite operands produce infinite results and remains set until cleared by implementer. Under alternate exception handling, this flag is not affected unless maybe by the implementer's alternate exception handling routine.
UNDRFLW	9	Floating-point “underflow” exception flag. Under default handling, this flag is automatically set to “1” whenever floating-point operations result in the production of a subnormal number that is **inexact** and remains set until cleared by implementer. Under alternate exception handling, this flag is not affected unless maybe by the implementer's alternate exception handling routine.
INEXCT	10	Floating-point “iinexact” exception flag. Under default handling, this flag is never automatically set to “1” under any condition. Under alternate exception handling, this flag is not affected unless maybe by the implementer's alternate exception handling routine.
ALTNVLD	11	Floating-point alternate exception handler interrupt enable for “invalid” exception. If set, an interrupt will immediately be generated upon detection of an invalid operation exception.
ALTDIV0	12	Floating-point alternate exception handler interrupt enable for “divide-by-0” exception. If set (and corresponding DELDIV0 bit is “0”), the interrupt will immediately be generated upon detection of an attempt to divide a non-zero number by zero. If set (and corresponding DELDIV0 bit is “1”), an interrupt will be delayed until the results of the operation are actually read out the corresponding operator's result buffer.
ALTOVFL	13	Floating-point alternate exception handler interrupt enable for “overflow” exception. If set (and corresponding DELOVFL bit is “0”), an interrupt will immediately be generated upon detection of an operation that results in an overflow condition. If set (and corresponding DELDIV0 bit is “1”), the interrupt will be delayed until the results of the operation are actually read out the corresponding operator's result buffer.
ALTUNFL	14	Floating-point alternate exception handler interrupt enable for “underflow” exception. If set (and corresponding DELUNFL bit is “0”), the interrupt will immediately be generated upon detection of an operation that results in an underflow condition (**except of for results that are exact**, as exact underflows are never signaled). If set (and corresponding DELUNFL bit is “1”), the interrupt will be delayed until the results of the operation are actually read out the corresponding operator's result buffer.
ALTNXCT	15	Floating-point alternate exception handler interrupt enable for “inexact” exception. If set (and corresponding DELNXCT bit is “0”), an interrupt will immediately be generated upon detection of an operation that produces inexact results. If set (and corresponding DELNXCT bit is “1”), an interrupt will be delayed until the results of the operation are actually read out the corresponding operator's result buffer.

DELNVLD	16	This bit can be tested by the alternate immediate exception handler for “invalid” operation to determine if such handling should be delayed or immediate.
DELDIV0	17	If set to “1”, alternate exception handling interrupt for “divide-by-zero” will be delayed until the results of the corresponding operation signaling the exception are actually read out of its corresponding result buffer. If “0”, enabled alternate exception handling interrupt for “divide-by-zero” is immediate.
DELOVFL	18	If set to “1”, alternate exception handling interrupt for “overflow” will be delayed until the results of the corresponding operation signaling the exception are actually read out of its corresponding result buffer. If “0”, enabled alternate exception handling interrupt for “overflow” is immediate.
DELUNFL	19	If set to “1”, alternate exception handling interrupt for “**inexact** underflow” will be delayed until the results of the corresponding operation signaling the exception are actually read out of its corresponding result buffer. If “0”, enabled alternate exception handling interrupt for “inexact underflow” is immediate.
DELNXCT	20	If set to “1”, alternate exception handling interrupt for “inexact” results will be delayed until the results of the corresponding operation signaling the exception are actually read out of its corresponding result buffer. If “0”, enabled alternate exception handling interrupt for “inexact” is immediate.
IRQEN	21	If set to “1”, general-purpose interrupt is enabled
IRQPEND	22	Read-only. If “1” general-purpose interrupt is pending/being requested
not used	23-28	(Read-only) these bits are presently unused and are read as 0.
Z || N	29	Z or N flag is set.
NEVER	30	(Read-only) this bit is used with BTBS instruction to create a NOP (branch never). Since it never evaluates as true, a branch is never taken.
ALWAYS	31	(Read-only) this bit is used with BTBS instruction to create a BRA (branch always). Since it always evaluates as true, a branch is always taken.

Each thread has its own 32-bit Status register. While most of the bits in the Status register are both readable and writable, some are read-only. Examples:

```
OR      STATUS, STATUS, #00000010b          ;set the carry flag
AND     STATUS, @clr_exc_mask, STATUS       ;clear the floating-point EXC interrupt enable bit
                                            ;requires 32-bit mask using table-read
OR      STATUS, STATUS, #00100000b          ;set the DONE bit
```

With the Cross-32 Meta Assembler, the instruction table can be easily enhanced to include aliases for the above operations to make code more readable and easier to code in assembly language. With the aliases, the assembler will generate the exact same machine code as above. Examples:

```
SETC            ;set carry flag
CLRC            ;clear carry flag
```

Here are just a few other alias examples directed at the STATUS register:

```
BNZ     dest        ;branch if not zero
BZ      dest        ;branch if zero
BC      dest        ;branch if carry
BNC     dest        ;branch if no carry
BRA     dest        ;branch ALWAYS uses STATUS bit 31
NOP                 ;branch NEVER uses STATUS bit 30
BGT     dest        ;branch if greater than (STATUS bit 29 = 0)
BLE     dest        ;branch if less than or equal (STATUS bit 29 = 1)
BGZ     dest        ;branch if greater than 0 (test for N flag = 0)
BLZ     dest        ;branch if less than 0 (test for N flag = 1)
BEQ     dest         ;branch if equal
BN      dest         ;branch if negative
BP      dest         ;branch if positive
BV      dest         ;branch if overflow
BNV     dest         ;branch if no overflow
BCND    dest,  Z     ;branch if zero
BCND    dest, DONE   ;branch if STATUS bit 5 (DONE bit) is set
LOCK                 ;set the LOCK bit (STATUS bit 4)
UNLOCK               ;clear the LOCK bit
etc., etc.
```

5.1.4 Auxiliary Registers AR3 - AR0

Figure 5—4. Auxiliary Registers AR3 – AR0

	DecrAmount	IncrAmount		ARn
Reset:	0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 1	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	31 24	23 16	15	14 0

Logical Address AR0: 0x0070

Logical Address AR1: 0x0071

Logical Address AR2: 0x0072

Logical Address AR3: 0x0073

Bit 15 is read-only.

Each thread has has four auxiliary registers located in their private memory maps with AR0 located at 0x070, AR1 at 0x071, AR2 at 0x072 and AR3 at 0x073. The auxiliary registers are used primarily for indirect addressing mode, wherein their contents are used as indirect pointers to either data memory or for table-read operations from program memory, which is also globally mapped to thread data memory in the range 0x1FFF to 0x1000.

The auxiliary registers have the ability to be automatically post-incremented or post-decremented immediately after use and come in handy for performing floating-point operations on medium to large data sets. For example:

```
MOV    AR0, #FMUL_0          ;load AR0 with pointer to first bin of floating-point operator buffer
MOV    AR1, xvector          ;load AR1 with xvector location
MOV    AR2, yvector          ;load AR2 with yvector location
MOV   AR3, outbuf             ;load AR3 with first location of outbuf

RPT   #11                     ;execute the next instruction 12 times
MOV   *AR0++, *AR1++, *AR2++    ;vector operation using two operands

MOV   AR0, #FMUL_0            ;point to first result bin of FPMUL operator result buffer

RPT   #11                     ;execute the next instruction 12 times
MOV   *AR3++, *AR0++          ;perform the transfer
BCND  done, ALWAYS            ;signal supervisor task completed
```

In the above example (assuming FMUL latency is four clocks and hardware scheduling is enabled with one-clock granularity for all four threads), some of the FMUL operations are still executing while the transfer is taking place. This is fine as long as a given read of a result buffer does not take place before that respective operation is completed. If that does happen, the PC for that thread will automatically be rewound and the MOV instruction re-fetched. In the above example, the first result and all subsequent results are available by the time of the first read (and subsequent reads) of the result buffer actually take place.

5.1.4.1 IncrAmount and DecrAmount

IncrAmount and DecrAmount occupy bits [32:24] and [23:16] (respectively) of a given auxiliary register and can be used to vary the automatic post-increment and/or post-decrement step amount when such addressing modes are used. On reset, the increment and decrement amounts are both set to 0x0001. Writing any non-zero value to these fields will change these amounts to the value written for that field. Writing 0x00 to either or both fields will have no effect on them, respectively.

5.1.4.2 Potential Hazard Using ARn Post-Modification Feature

In all cases other than four-thread interleave with one-clock granularity each, if the ARn post-modification feature is in the Dest field of an instruction, the value in that ARn will be post-modified immediately after the instruction-fetch cycle (q0), such that, if the very next instruction that is fetched following the instruction-fetch that contained the Dest ARn post-modifier operator expects to use the newly modified value, the value will not have had time to be modified because the post-modification of the ARn in the Dest field will not have been executed yet. This is because any ARn used in the SrcA or SrcB field as an indirect pointer are translated and registered into the RAM or registers being accessed at the end of that instruction's instruction fetch cycle, two clocks ahead of the Dest ARn update.

To illustrate this potential hazard, consider the following instruction sequence:

```
ADD    *AR1++, work1, *AR2++      ;the contents of *AR2 are added to the contents of work1
                                  ;and stored in the memory location pointed to by the contents
                                  ;of AR1.  AR1 and AR2 are then post-incremented to point to
                                  ;the next Dest location and next SrcB location.
MOV    work2, *AR1                ;the contents of the memory location pointed to by the
                                  ;contents of AR1 are copied to work2
```

In the above example, if the current thread is locked, or unlocked with an interleave less than four threads, the *AR1++ of the ADD instruction will not have had time to execute because the Dest ARn post-modification does not happen until after the execute cycle of that instruction. Moreover, the SrcA *AR1 of the immediately following MOV instruction needs the newly modified value immediately, during that instruction's instruction fetch cycle.

If interleave is set for four threads with one-clock granularity per thread, the foregoing potential hazard is not a problem, because the immediately-following MOV instruction will not be fetched until four clocks after the ADD fetch (due to the interleave), giving the post-modification of the Dest *AR1++ adequate time to execute.

5.1.5 Fused-Multiply-Add "C" Register

The FMA operator requires three operands but the SYMPL FP32X architecture does not presently support "directly" writing three operands to the three FMA operator input registers simultaneously. To overcome this limitation, the "C" register is provided, such that FMA operations require two write operations, the first write being to the C register, and the second to the FMA X and Y input registers using a dual-operand read/write MOV instruction. When operands X and Y are written to the FMA operator, the contents of the C register is also written, implicitly. Consequently, FMA presently cannot be used with the RPT instruction.

Figure 5—5. "C" Register

"C" Register

Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	31																															0

Logical Address: 0x006A

5.1.6 Hardware Loop-Counters LPCNT1 and LPCNT0

Figure 5—6. Hardware Loop-Counters LPCNT1 and LPCNT0

LPCNT1

Reset:	0	0	0	0	0	0	0	0	0	0	0	0
	11											0

Logical Address: 0x0069

LPCNT0

Reset:	0	0	0	0	0	0	0	0	0	0	0	0
	11											0

Logical Address: 0x0068

Each thread has two, memory-mapped loop-counters implemented in hardware that are used in combination with the DBNZ instruction to facilitate efficient looping. Generally speaking, looping a fixed number of times can be carried out in a variety of ways. The most common method is to initialize a memory location with the number of times a thread is to execute a given segment of code, begin executing it, and then, for the final two instructions of the segment, perform a decrement of the loop count value followed by a bit-test-and-branch-if-clear (BTBC) of the zero (Z) flag to the entry point of the segment if the Z flag is still clear.

In tight loops where the segment (not including the twp separate decrement and BTBC instructions) is only one or two instructions, at least half the processing cycles is devoted to branching. In such scenarios, this can be expensive, especially when large data sets are being operated on.

To help minimize the expense involved in tight looping, two, 12-bit loop-counters have been implemented as hardware registers so they can be used in combination with the DBNZ instruction. The DBNZ instruction combines the decrement and the branch into a single instruction, thereby reducing said cost by 50%.

If a particular looping situation involves nesting of loops greater than two, then the most economical approach is to employ the hardware loop-counters for the innermost loops and for the remaining outer loops, employ the conventional method that comprises the two separate decrement and BTBC instructions.

5.1.7 Timer Register

Figure 5—7. Timer Register

	Timer																			
Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	19																			0

Logical Address: 0x0067

Each thread has its own 20-bit timer whose zero-count output is tied directly to that thread's NMI input, such that, if the timer ever counts down to zero before a given task completes, that thread's NMI line will be asserted, forcing entry into that thread's NMI service routine as a type of safety-net.

The DONE bit of given thread is used as a gate/qualifier for the timer, such that only when the DONE bit is cleared to 0 (i.e., the thread is BUSY) does the timer register decrement. The DONE bit is also gated with the NMI line, such that if DONE is active high, the NMI line is disabled.

Consequently, since the zero-count output of the timer is hard-wired to the thread's NMI line, one of the first operations a thread must perform before clearing its DONE bit (signaling it is now busy), is load the timer with a cycle count value that exceeds the estimated number of clock cycles needed to complete the routine. If the timer ever reaches the value of 0x00000, a non-maskable interrupt will be generated and the thread encountering it will automatically be vectored to its NMI in-service routine, where it can then recover from an excessive time exception and alert the CPU by writing an appropriate message into its respective packet memory and asserting the DONE bit/flag active high, which in turn should generate a CPU interrupt, if enabled.

5.1.8 Quality of Service (QOS) Register

The QOS register comprises four 8-bit counters, one each for invalid operation, divide-by-zero, overflow, and underflow-inexact exception signals. The counters count the number of times their respective exception is signaled during the time a given thread's DONE bit is “0”, i.e, the thread is busy processing. After processing, the QOS contents can be copied to the respective thread's parameter buffer, along with the results, as a final step, which the CPU can then use as a indicator of quality of service after it pulls said results into system memory. While the DONE bit is set to “1”, the QOS counter registers are reset to 0x00 and remain that way until DONE is cleared to “0”. The QOS register is readable but not writeable.

Figure 5—8. QOS Register

	Inexact Underflow Counter	Overflow Counter	Divide-By-Zero Counter	Invalid Operation Counter
Reset:	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
	31 24	23 16	15 8	7 0

Logical Address: 0x0066

5.1.9 Dot Product Register

The SYMPL FP32X Dot-product operator is mapped at location 0x0065 (assuming it is included in a particular implementation). To employ it, simply perform a dual-operand-write to location 0x0065 with the two desired operands. Because the ADD portion of the operation requires five clocks to complete, the RPT (repeat) may not be used with the Dot operator. Instead, use a loop for multiple writes. Always bear in mind that the Dot-product operator results will not be available until after seven clocks have transpired. As such, try to order your instructions such that there is always at least one instruction between operand writes to the Dot-product dual-operand input registers and at least one other instruction before attempting to read the Dot-product results from the Dot-product output register, which is mapped at the same location. The foregoing assumes all four threads are scheduled with one clock granularity each.

Figure 5—9. Dot Product Register

	DOT Product
Reset:	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	31 0

Logical Address: 0x0065

Example Dot-product operations:

```
        MOV    AR0, #X_list          ;load AR0 with pointer to list of X operands
        MOV    AR1, #Y_list          ;load AR1 with pointer to list of Y operands
        MOV    LPCNTR0, #10          ;load LPCNTR0 with the number of X,Y operands to compute
loop:   MOV    DOT, *AR0++, *AR1++           ;compute first sum of products
        DBNZ   loop, LPCNTR0         ;decrement LPCNTR0 and branch if not zero
        MOV    result, DOT           ;copy results of sum-of-products to final result location
```

Note that in the above example, the DBNZ instruction satisfies the requirement of having at least one instruction between operand writes to the Dot-product operand input registers and read of the accumulated result. The Dot-product accumulator is automatically cleared to zero each time results are read from the result register at location 0x0065.

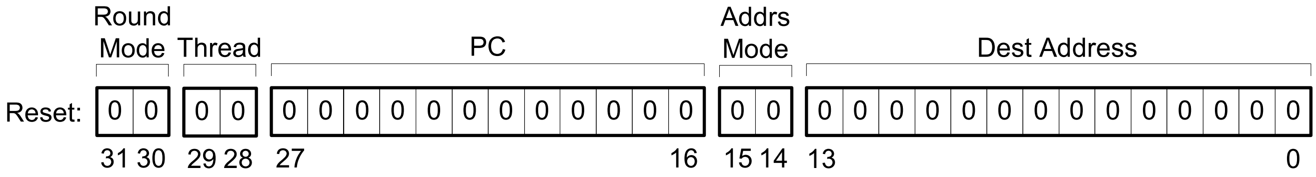
5.1.10 Alternate-Delayed Exception Capture Registers

To facilitate alternate delayed exception handling, four capture registers are provided and are described below. For more information regarding their use, refer to Section 6.2.

5.1.10.1 Capture Register 3

Whenever alternate-delayed exception handling is enabled for a given exception and that exception is signaled, upon reading a given operator's result buffer containing the results of the operation that signaled such exception, Capture Register 3 will automatically capture certain information that can be used in the exception handler interrupt service routine as shown in the following diagram and table.

Figure 5—10. Capture Register 3



Logical Address: 0x0063

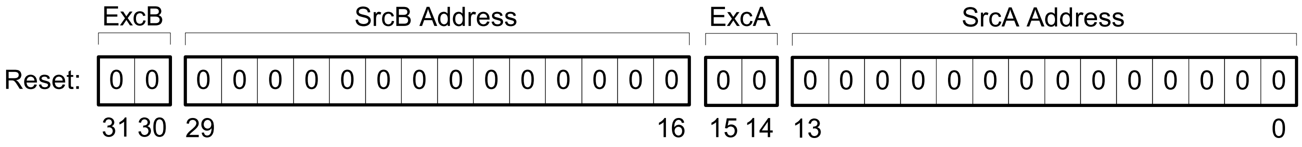
Table 5—2. Capture Register 3 Field Descriptions

Name	Bit-Position	Description
Round Mode	[31:30]	Rounding mode used during the floating-point operation that resulted in the exception
Thread	[29:28]	Thread number to which the results belong
PC	[27:16]	Program counter address that originally retrieved the results and initiated the alternate delayed exception interrupt due to the floating-point exception being previously raised
Address Mode	[15:14]	Addressing mode of the instruction at the above program counter address
Dest Address	[13:0]	Destination address to where the result that were read from the result buffer would have been written but for the alternate delayed exception interrupt

5.1.10.2 Capture Register 2

Whenever alternate-delayed exception handling is enabled for a given exception and that exception is signaled, upon reading a given operator's result buffer containing the results of the operation that signaled such exception, Capture Register 2 will automatically capture certain information that can be used in the exception handler interrupt service routine as shown in the following diagram and table.

Figure 5—11. Capture Register 2



Logical Address: 0x0062

Table 5—3. Capture Register 2 Field Descriptions

Name	Bit-Position	Description
Except B	[31:30]	Exception Code for data bus B results. If exception was divide-by-zero (code 01), overflow (code 10), or underflow-inexact (code 11), then this field will contain such code if the data bus B results belong to the operation that was excepted. Note that alternate delayed exception handling for invalid operation contains no code other than “00” (if not excepted) or “01” if excepted for invalid operation, because invalid operation has its own vector and is of only one type and is higher priority than the others.
SrcB Address	[29:16]	Source address B of the instruction used to read the result buffer (in the case of dual-operand read) and trigger the alternate delayed exception interrupt, such being triggered only if the corresponding results were excepted
Except A	[15:14]	Exception Code for data bus A results. If exception was divide-by-zero (code 01), overflow (code 10), or underflow-inexact (code 11), then this field will contain such code if the data bus A results belong to the operation that was excepted. Note that alternate delayed exception handling for invalid operation contains no code other than “00” (if not excepted) or “01” if excepted for invalid operation, because invalid operation has its own vector, is of only one type, and is higher priority than the others.
SrcA Address	[13:0]	Source address A of the instruction used to read the result buffer and trigger the alternate delayed exception interrupt, such being triggered only if the corresponding results were excepted, but always captured nonetheless during alternate-delayed exception handling

5.1.10.3 Capture Register 1

Whenever alternate-delayed exception handling is enabled for a given exception and that exception is signaled, upon reading a given operator's result buffer containing the results of the operation that signaled such exception, Capture Register 1 will automatically capture the results read from **read-data-bus B** of the floating-point result buffer, assuming it is a dual-operand (i.e., dual-result read). Care must be taken not to perform a dual-result/operand read from the floating-point result buffer region if the result attempting to be read were never written with operands to be computed in the first place, because this will cause the thread to time-out, due to the corresponding "operation-complete" semaphore not being set.

Figure 5—12. Capture Register 1

Floating-Point Diagnostics Capture Register (FPDCR) 1

Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	31																															0

Read Data Bus B

Logical Address: 0x0061

5.1.10.4 Capture Register 0

Whenever alternate-delayed exception handling is enabled for a given exception and that exception is signaled, upon reading a given operator's result buffer containing the results of the operation that signaled such exception, Capture Register 1 will automatically capture the results read from **read-data-bus A** of the floating-point result buffer, assuming it is a dual-operand (i.e., dual-result read). Care must be taken not to perform a dual-result/operand read from the floating-point result buffer region if the result attempting to be read were never written with operands to be computed in the first place, because this will cause the thread to time-out, due to the corresponding "operation-complete" semaphore not being set.

Figure 5—13. Capture Register 0

Floating-Point Diagnostics Capture Register (FPDCR) 0

Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	31																															0

Read Data Bus A

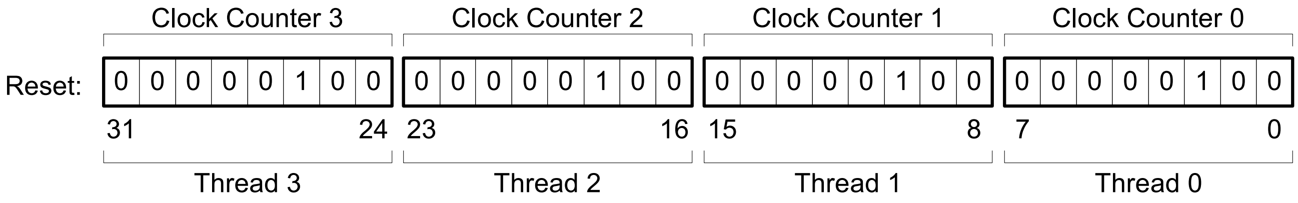
Logical Address: 0x0060

5.2 Global Registers

In addition to the private memory-mapped registers described above, each Shader employs several globally-mapped registers that are shared by all of its four threads. These globally-mapped registers include the fine-grained scheduler counter and compare (CGS) registers, the REPEAT register, and the OutBox register as described below.

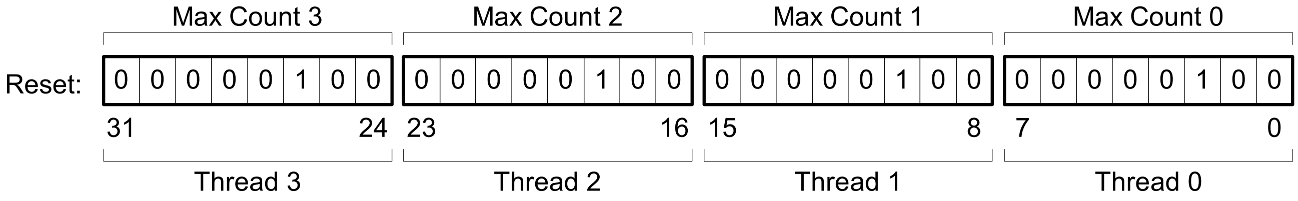
5.2.1 Fine-Grained Scheduler Counter and Compare Registers

Figure 5—14. Fine-Grained Scheduler Counter Register



Logical Address: 0x006C

Figure 5—15. Fine-Grained Scheduler Compare Register



Logical Address: 0x006B

Each multi-thread Shader core has one, global, fine-grained scheduler up-counter register at location 0x06C and a counter compare register at location 0x06B that can be used to schedule automatic thread switches at pre-defined time intervals. On reset, the default setting is for a granularity of one clock per thread, with all threads having one time-slot per clock, with thread 0 having the first slot, such that the default thread interleave sequence is thread 0 [one clock], thread 1 [one clock], thread 2 [one clock], and thread 3 [one clock], in continuous round-robin fashion.

When the fine-grained scheduler counter register at location 0x06C is written to, the exact value containing the max count for each thread being written is simultaneously copied directly into the scheduler compare register at location 0x06B. This value in the compare register does not change, unless the scheduler counter register at location 0x06C is written to again, but with a different value than before.

Referring to *Figure 5—14* above, it can be seen that there are a total of four, 8-bit up-counters, one for each thread. On reset, these counters are initialized for a count of four each (i.e., 0x04040404), which will produce a one-clock interleave granularity on all four-threads, provided that the LOCK bit in all the threads' status registers are cleared. If any of the threads' LOCK bit is set, then the fine-grained scheduler for that thread's time-slot will have no effect and the core will remain in the current thread until a soft-schedule event occurs or the LOCK bit is cleared by the thread that set it.

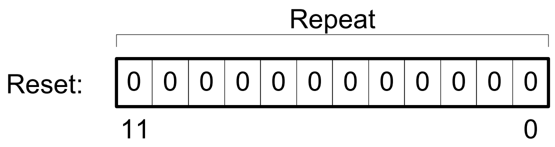
Writing a 0x00 to a given thread's hardware fine-grained scheduler counter register position will effectively remove that thread from the fine-grained scheduling queue, but still may be soft-scheduled by a different thread. If the value 0x00000000 is written to the fine-grained scheduler counter register, then hardware fine-grained scheduling will not take place for any thread, regardless of the state of any thread's LOCK bit.

The fine-grained scheduler counter "count==compare" outputs are prioritized, with thread0 having the highest priority and thread3 having the lowest. Consequently, if your application requires asymmetric threading in terms of duty cycle and one-clock granularity, for best results, ensure that you do your duty cycle calculations such that all the employed threads duty cycles add up to 100%.

For example, say you want thread 0 to only occupy 10% of the cycles and thread 3, 90%, with the remaining two threads removed from the queue, load thread 0's fine-grained counter with the value 0x09 and thread 3's counter with the value 0x01. In this instance, thread 3's scheduler event output will always be active (true), but since thread 0's output is higher priority, when it reaches its max count, it will override thread 3's output, causing it to switch to thread 0 for one clock before switching back to thread 3. Conversely, if the duty cycles were swapped, this would not work, because thread 0 is higher priority.

5.2.2 Repeat Register

Figure 5—16. Repeat Register



Logical Address: 0x0064

Each thread has access to a single, globally-mapped Repeat Register located at 0x064. When loaded with a value other than 0x000, execution of the immediately following instruction will be "repeated" the specified number of times, i.e., it will execute 1 + "repeat" number specified. The repeat function can be used with all the atomic instructions except BTB (including its aliases).

RPT may be used for, among other things, filling up a given floating-point operator's pipe, but care must be taken not to overflow the operator's result buffer, such that, by the time all sixteen operands have been written, a result is available for reading by a MOV instruction. Anytime RPT becomes active, a temporary lock is automatically placed on the hardware fine-grained scheduler until the instruction being repeated has completed. Any soft-schedule specification appearing next to the mnemonic that loads the repeat will be ignored by the hardware and flagged by the assembler if the RPT alias is used for assembly.

Example:

```
                                        ;scale vector by 1/3 then calculate SQRT of each
MOV     AR1, #vect_x                    ;load AR1 with pointer to first x value
MOV     AR0, #FMUL_0                    ;load AR0 with pointer to first input register for FMUL
                                        ;floating-point operator
RCP     work_1, #3                      ;get reciprocal of 3 and store in work_1

RPT  #15                                ;execute the next instruction 16 times
                                        ;(i.e., 1 plus the RPT #n specified)
MOV     *AR0++, *AR1++, work_1          ;multiply the vector
                                        ;upon completion, the result for the first
                                        ;(and subsequent) FMUL(s) are now immediately
                                        ;available for reading well in advance (since FMUL pipe is
                                            ;only four clocks deep)
MOV       AR0, #FMUL_0                      ;point to first FMUL result buffer
RPT       #15
MOV       *AR1++, AR0++                     ;square all 16 FMUL results.  Upon completion of RPT
                                            ;all SQRT results will be available for reading (in sequence)
```

5.3 Instruction Set Descriptions

This section describes the SYMPL FP32X-AXI4 instruction set. With exception to the reciprocal and trig instructions, they are the same for both the Shader core and the coarse-grained scheduler core. All instructions are 32-bits in length and execute in one clock cycle, without stalls.

5.3.1 ADD

Figure 5—17. ADD Instruction

NT	AM	Op-Code	Dest	SrcA	SrcB
	0 0	0 1 1 0			
31 30	29 28	27 24	23 16	15 8	7 0

NT	AM	Op-Code	Dest	SrcA	#Imm8
	0 1	0 1 1 0			
31 30	29 28	27 24	23 16	15 8	7 0

NT	AM	Op-Code	Dest	@Table-Read	SrcB
	1 0	0 1 1 0			
31 30	29 28	27 24	23 16	15 8	7 0

ADD adds (without carry flag) the contents SrcB to the contents of SrcA and then stores the results in Dest. Permitted addressing modes for SrcA include direct, indirect, or table-read from program memory. Permitted addressing modes for SrcB include direct, indirect, or 8-bit immediate. If SrcB is 8-bit immediate, it is zero-extended to 32 bits by the hardware before the addition takes place. Dest addressing mode is always either direct or indirect. STATUS flags Z, C, V and N are affected as shown below.

STATUS flags:

N—set if MSB of result is 1 (negative), reset otherwise
C—set if a carry occurred out of MSB, reset otherwise
V—set if XOR of the result's two MSBs = 1, reset otherwise
Z—set if result is zero, reset otherwise

5.3.2 ADDC

Figure 5—18. ADDC Instruction

NT	AM	Op-Code	Dest	SrcA	SrcB
	0 0	0 1 1 1			
31 30	29 28	27 24	23 16	15 8	7 0

NT	AM	Op-Code	Dest	SrcA	#Imm8
	0 1	0 1 1 1			
31 30	29 28	27 24	23 16	15 8	7 0

NT	AM	Op-Code	Dest	@Table-Read	SrcB
	1 0	0 1 1 1			
31 30	29 28	27 24	23 16	15 8	7 0

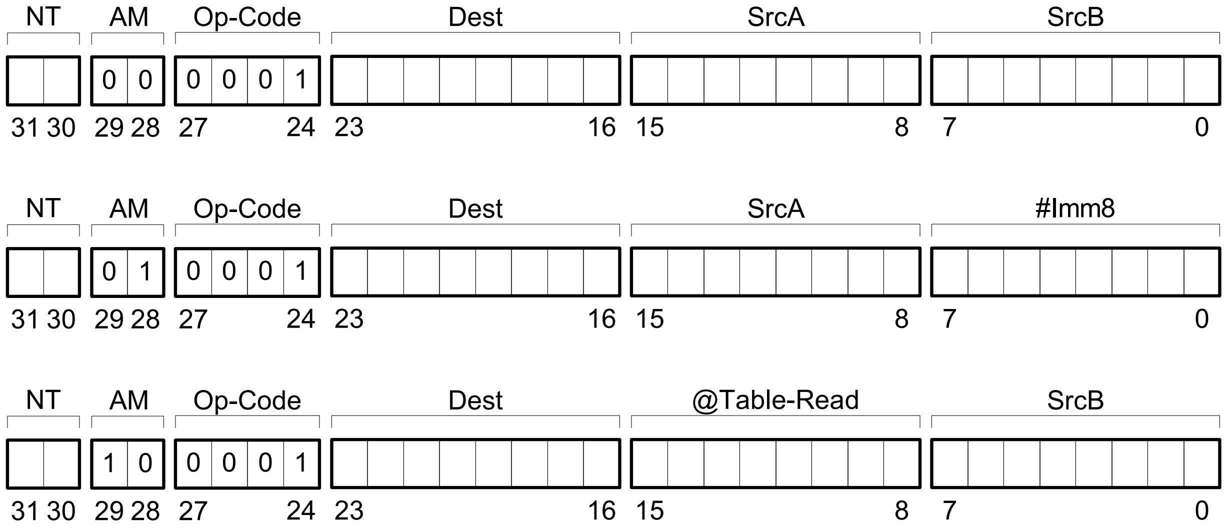
ADDC adds the contents SrcB and the carry flag to the contents of SrcA. Permitted addressing modes for SrcA include direct, indirect, or table-read from program memory. Permitted addressing modes for SrcB include direct, indirect, or 8-bit immediate. If SrcB is 8-bit immediate, it is zero-extended to 32 bits by the hardware before the addition takes place. Dest addressing mode is always either direct or indirect. STATUS flags Z, C, V and N are affected as shown below.

STATUS flags:

N—set if MSB of result is 1 (negative), reset otherwise
C—set if a carry occurred out of MSB, reset otherwise
V—set if XOR of the result's two MSBs = 1, reset otherwise
Z—set if result is zero, reset otherwise

5.3.3 AND

Figure 5—19. AND Instruction



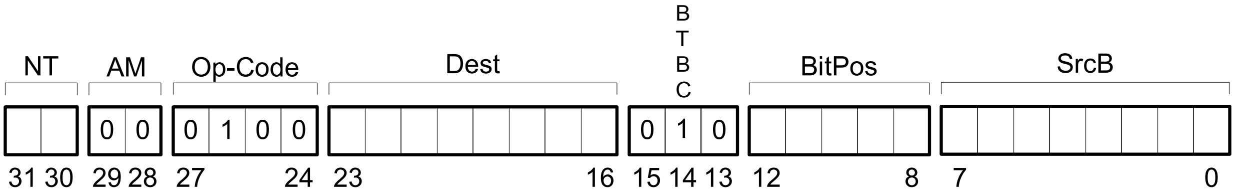
AND performs a bit-wise AND of the contents of SrcA and the contents of SrcB. Permitted addressing modes for SrcA include direct, indirect, or table-read from program memory. Permitted addressing modes for SrcB include direct, indirect, or 8-bit immediate. If SrcB is 8-bit immediate, it is zero-extended to 32 bits by the hardware before the addition takes place. Dest addressing mode is always either direct or indirect. STATUS flags Z and N are affected as shown below.

STATUS flags:

N—set if MSB of result is 1 (negative), reset otherwise
C—not affected
V—not affected
Z—set if result is zero, reset otherwise

5.3.4 BTBC

Figure 5—20. BTBC Instruction



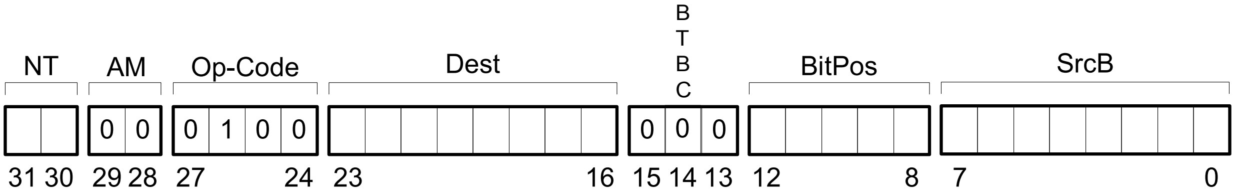
Bit-test-and-branch-if-clear (BTBC) tests for logic 0 the bit position specified in BitPos with the contents of SrcB. If the result of the test is 0 (true), then the PC is loaded with an offset relative to the program address of the BTBC fetch, such offset being in the range +127 to -128. Permitted addressing modes for SrcB being direct or indirect.

If at least three threads are active with interleave granularity of one clock each, then that thread's two following instructions will not be fetched. Otherwise, any instructions fetched by the same thread during the following two clock cycles will be discarded if the branch is taken.

Z, C, V and N flags remain unaffected.

5.3.5 BTBS

Figure 5—21. BTBS Instruction



Bit-test-and-branch-if-set (BTBS) tests for logic 1 the bit position specified in BitPos with the contents of SrcB. If the result of the test is 1 (true), then the PC is loaded with an offset relative to the program address of the BTBS fetch, such offset being in the range +127 to -128. Permitted addressing modes for SrcB being direct or indirect.

If at least three threads are active with interleave granularity of one clock each, then that thread's two following instructions will not be fetched. Otherwise, any instructions fetched by the same thread during the following two clock cycles will be discarded if the branch is taken.

Z, C, V and N flags remain unaffected.

5.3.6 COS

Figure 5—22. COS Instruction

NT	AM	Op-Code	Dest	SrcA	
	0 0	1 1 0 1			0 0 0 0 0 0 0 0
31 30	29 28	27 24	23 16	15 8	7 0

NT	AM	Op-Code	Dest	#Imm16 {#SrcA, #SrcB}
	1 1	1 1 0 1		0 0 0 0 0 0 0
31 30	29 28	27 24	23 16	15 0

COS returns the 32-bit, floating-point cosine of the 9-bit, signed integer contained in SrcA or 16-bit immediate value and stores the result in Dest. The integer contained in SrcA or the 16-bit immediate value must be in the range of +/- 360 degrees. STATUS flags Z and N are affected as shown below.

STATUS flags:

N—set if MSB of result is 1 (negative), reset otherwise
C—not affected
V—not affected
Z—set if result is zero, reset otherwise

5.3.7 COT

Figure 5—23. COT Instruction

NT	AM	Op-Code	Dest	SrcA	
	0 0	1 1 1 1			0 0 0 0 0 0 0 0
31 30	29 28	27 24	23 16	15 8	7 0

NT	AM	Op-Code	Dest	#Imm16 {#SrcA, #SrcB}
	1 1	1 1 1 1		0 0 0 0 0 0 0
31 30	29 28	27 24	23 16	15 0

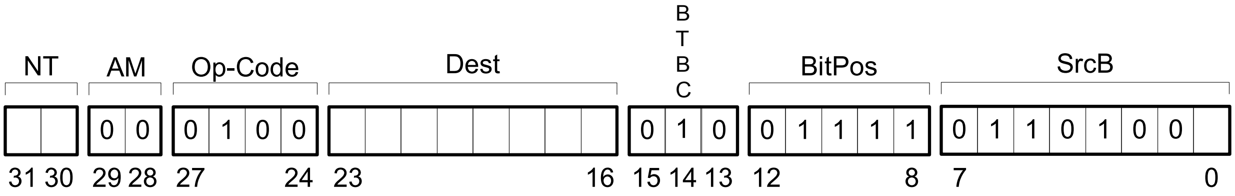
COT returns the 32-bit, floating-point cosine of the 9-bit, signed integer contained in SrcA or 16-bit immediate value and stores the result in Dest. The integer contained in SrcA or the 16-bit immediate value must be in the range of +/- 360 degrees. STATUS flags Z and N are affected as shown below.

STATUS flags:

N—set if MSB of result is 1 (negative), reset otherwise
C—not affected
V—not affected
Z—set if result is zero, reset otherwise

5.3.8 DBNZ

Figure 5—24. DBNZ Instruction



Decrement-and-branch-if-result-not-zero (DBNZ) first decrements the hardware loop-counter (LPCNTn) specified in SrcB then tests bit position 15 of the result for logic 0. If the result of the test is 0 (true), then the PC is loaded with an offset relative to the program address of the DBNZ fetch, such offset being in the range +127 to -128. Permitted addressing modes for SrcB being direct or indirect.

If at least three threads are active with interleave granularity of one clock each, then that thread's two following instructions will not be fetched. Otherwise, any instructions fetched by the same thread during the following two clock cycles will be discarded if the branch is taken.

Z, C, V and N flags remain unaffected.

5.3.9 MOV

Figure 5—25. MOV Instruction

NT	AM	Op-Code	Dest	SrcA	SrcB
	0 0	0 0 0 0			
31 30	29 28	27 24	23 16	15 8	7 0

NT	AM	Op-Code	Dest	#Imm16 {#SrcA, #SrcB}
	1 1	0 0 0 0		
31 30	29 28	27 24	23 16	15 0

NT	AM	Op-Code	Dest	@Table-Read	SrcB
	1 0	0 0 0 0			
31 30	29 28	27 24	23 16	15 8	7 0

NT	AM	Op-Code	Dest	@Table-Read	
	1 0	0 0 0 0			0 0 0 0 0 0 0 0
31 30	29 28	27 24	23 16	15 8	7 0

NT	AM	Op-Code	Dest	SrcA	#Imm8
	0 1	0 0 0 0			
31 30	29 28	27 24	23 16	15 8	7 0

Move SrcA (and SrcB if present) to Dest. If SrcB is present and either (or both) SrcA and/or SrcB reference any location within the floating-point result buffer (0x0080 to 0x00FF), this implies that the MOV is being used for a floating-point operation that requires two operands, OprndA and OprndB, in which case, both operands will be read from memory and written to that floating-point operator's dual-input registers during execution.

If SrcB is absent “and” SrcA is immediate, the immediate value is taken as #immed16 instead of #immed8. Both 16-bit immediate and 8-bit immediate values are automatically zero-extended to 32 bits by the hardware. Dest addressing mode is always either direct or indirect. STATUS flags Z and N are affected as shown below.

STATUS flags:

N—set if MSB of result is 1 (negative), reset otherwise
C—not affected
V—not affected
Z—set if result is zero, reset otherwise

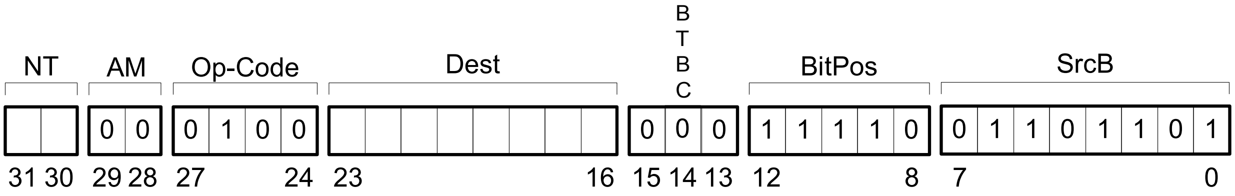
5.3.10 MOV (Special Feature)

MOV has a special feature when using it to read single or dual results from any location within a thread's floating-point result buffer area (private RAM locations 0x00FF through 0x0080). If SrcA and/or SrcB of the MOV instruction references any location within the floating-point result buffer, the MOV instruction automatically tests the corresponding semaphore for the contents of SrcA (and/or SrcB if present) to see if the result(s) is(are) “ready.”

If ready, execution of the MOV instruction proceeds as usual. If not ready, then that thread's PC is re-wound to the program memory location of the MOV instruction and re-fetched. This re-winding of the PC will continue until ready becomes active high. Consequently, the core's instruction pipeline never actually stalls, in that, under these conditions, the MOV instruction is actually simultaneously acting like both a MOV instruction and BTBC (or conditional MOV) instruction combined.

5.3.11 NOP

Figure 5—26. NOP Instruction

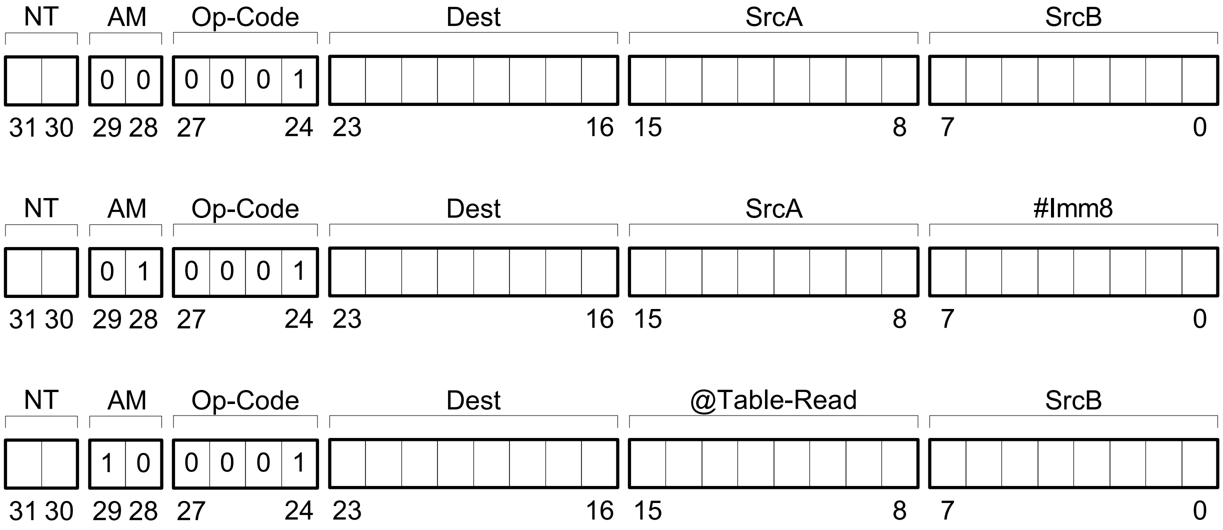


NOP is actually an alias of the bit-test-and-branch-if-set (BTBS) instruction, except the cross-assembler automatically fills in the SrcB field with the address of the STATUS register and specifies bit-30 (labeled “NEVER” and hard-wired to “0”) as the bit to be tested for logic 1. Since the test is never true, a branch is never taken and falls right through, acting like a NOP.

Z, C, V and N flags remain unaffected.

5.3.12 OR

Figure 5—27. OR Instruction



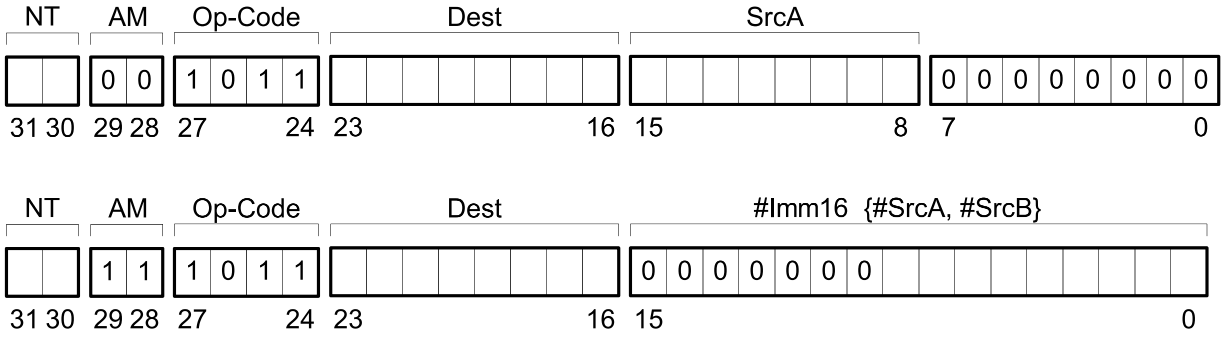
OR performs a bit-wise OR of the contents of SrcA and the contents of SrcB and then stores the results in Dest. Permitted addressing modes for SrcA include direct, indirect, or table-read from program memory. Permitted addressing modes for SrcB include direct, indirect, or 8-bit immediate. If SrcB is 8-bit immediate, it is zero-extended to 32 bits by the hardware before the addition takes place. Dest addressing mode is always either direct or indirect. STATUS flags Z and N are affected as shown below.

STATUS flags:

N—set if MSB of result is 1 (negative), reset otherwise
C—not affected
V—not affected
Z—set if result is zero, reset otherwise

5.3.13 RCP

Figure 5—28. RCP Instruction



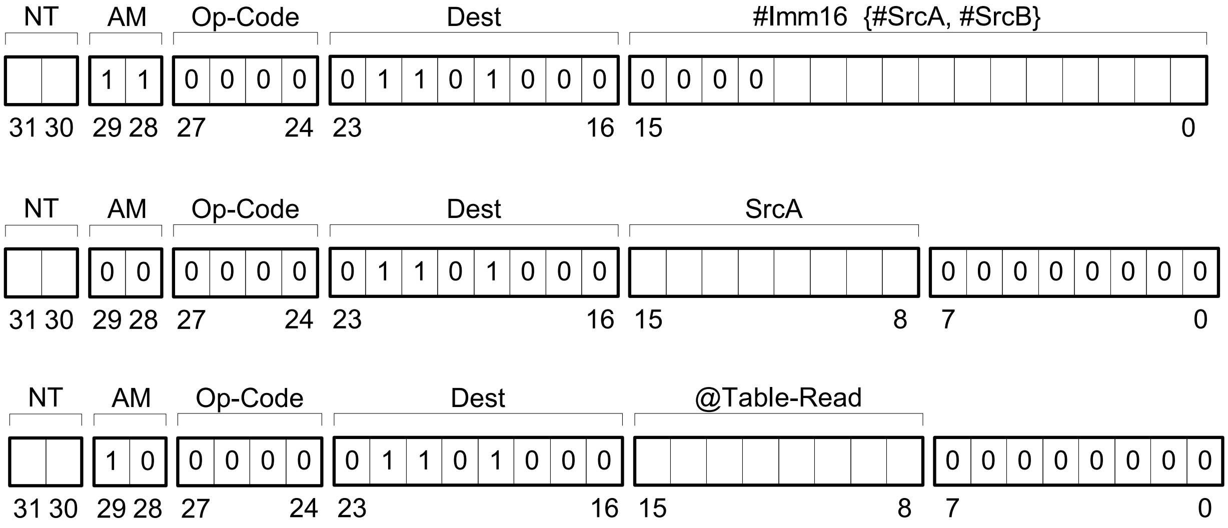
RCP returns the 1/n (reciprocal) of the 8-bit SrcA (signed integer) and stores the 32-bit floating-point result in Dest. SrcA must be a signed integer in the range +127 to -128. Data bits D[31:8] of SrcA are ignored. Input of SrcA == 0 returns 0x7F800000. Results of the RCP can be used in combination with the FMUL operator to perform quick floating-point divides in five clocks in lieu of the FDIV operator, which takes 15 clocks to complete. STATUS flags Z and N are affected as shown below.

STATUS flags:

N—set if MSB of floatingpoint result is 1 (negative), reset otherwise
C—not affected
V—not affected
Z—always cleared to 0

5.3.14 RPT

Figure 5—29. RPT Instruction



RPT is an alias of the MOV instruction, which loads the 12-bit, hardware repeat-counter with the contents of SrcA or 12-bit immediate value. When the contents of the repeat-counter is non-zero, the immediately following instruction is “repeated” the specified number of times. A non-zero value in the repeat-counter will automatically place a lock on the current thread until said next instruction repeats the specified number of times. Any soft-schedule specifier appearing next to the RPT mnemonic or the immediately following instruction will flag an error by the assembler.

5.3.15 SHFT

Figure 5—30. SHFT Instruction

NT	AM	Op-Code	Dest	SrcA		ShftTyp	ShftAmt
	0 0	0 1 0 1			0		
31 30	29 28	27 24	23 16	15 8	7	6 4	3 0

NT	AM	Op-Code	Dest	@Table-Read		ShftTyp	ShftAmt
	1 0	0 1 0 1			0		
31 30	29 28	27 24	23 16	15 8	7	6 4	3 0

SHFT barrel-shifts SrcA from 1 to 16 bits, using both the shift-type specifier and shift-amount specifier, then stores the result in Dest. Except where noted, Z, N, and C flags are affected. V flag remains unchanged. SHFT provides seven different types of shifts shown as follows:

D[6:4]	ShftType	Description
“000”	LEFT	Carry flag is unaffected. LSBs filled with “0”. Same as ASL.
“001”	LSL	Logical shift left through Carry. “0” shifted in through LSB.
“010”	ASL	Arithmetic shift left. LSBs filled with “0”. Same as LEFT above.
“011”	ROL	True barrel shift left. Bits shifted out of MSB are shifted in through LSB. Carry flag is unaffected.
“100”	RIGHT	Shift right. Carry flag is unaffected. MSBs filled with “0”.
“101”	LSR	Logical shift right. LSB shifted through Carry. “0” shifted through MSB.
“110”	ASR	Arithmetic shift right. MSB does not change and is copied ShftAmt times. Carry is unaffected.
“111”	ROR	True barrel shift right. Bits shifted out of LSB are shifted in through MSB. Carry is unaffected.
D[3:0]	**ShftAmt**	
“0000” “1111”	through	Encoded shift amount. “0000” means shift by 1. “1111” means shift by “16”. The assembler encodes ShftAmt by subtracting “1” from the value appearing on the assembly line.

Example:

```
SHFT   result2, vectx, ASR, 3            ;divide vectx by 8, copying Carry into MSB each time
```

5.3.16 SIN

Figure 5—31. SIN Instruction

NT	AM	Op-Code	Dest	SrcA	
	0 0	1 1 0 0			0 0 0 0 0 0 0 0
31 30	29 28	27 24	23 16	15 8	7 0

NT	AM	Op-Code	Dest	#Imm16 {#SrcA, #SrcB}
	1 1	1 1 0 0		0 0 0 0 0 0 0
31 30	29 28	27 24	23 16	15 0

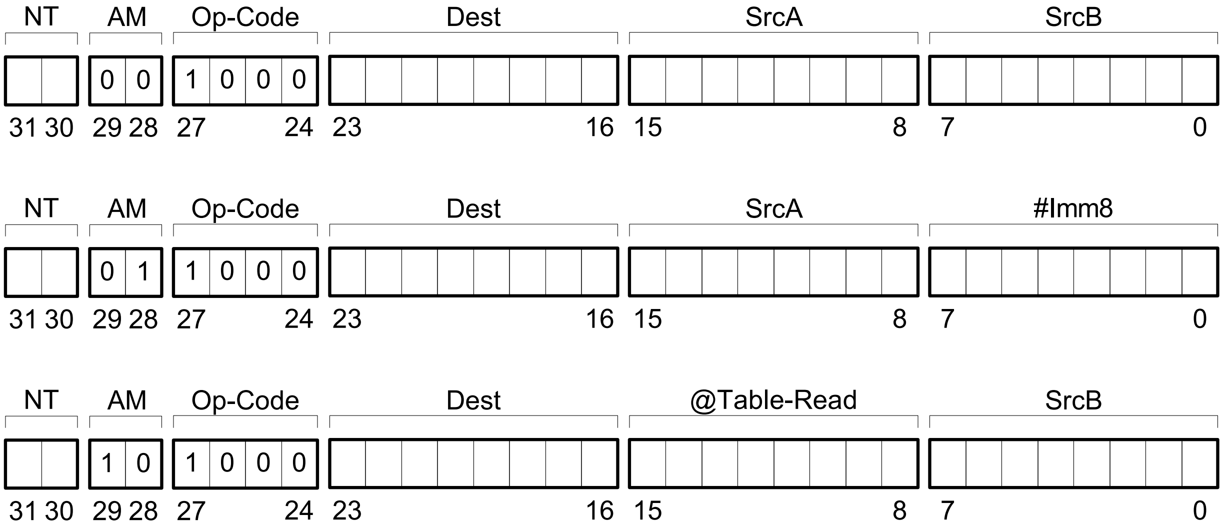
SIN returns the 32-bit, floating-point sine of the 9-bit, signed integer contained in SrcA, or 16-bit immediate value, and stores the result in Dest. The integer contained in SrcA or the 16-bit immediate value must be in the range of +/- 360 degrees. STATUS flags Z and N are affected as shown below.

STATUS flags:

N—set if MSB of result is 1 (negative), reset otherwise
C—not affected
V—not affected
Z—set if result is zero, reset otherwise

5.3.17 SUB

Figure 5—32. SUB Instruction



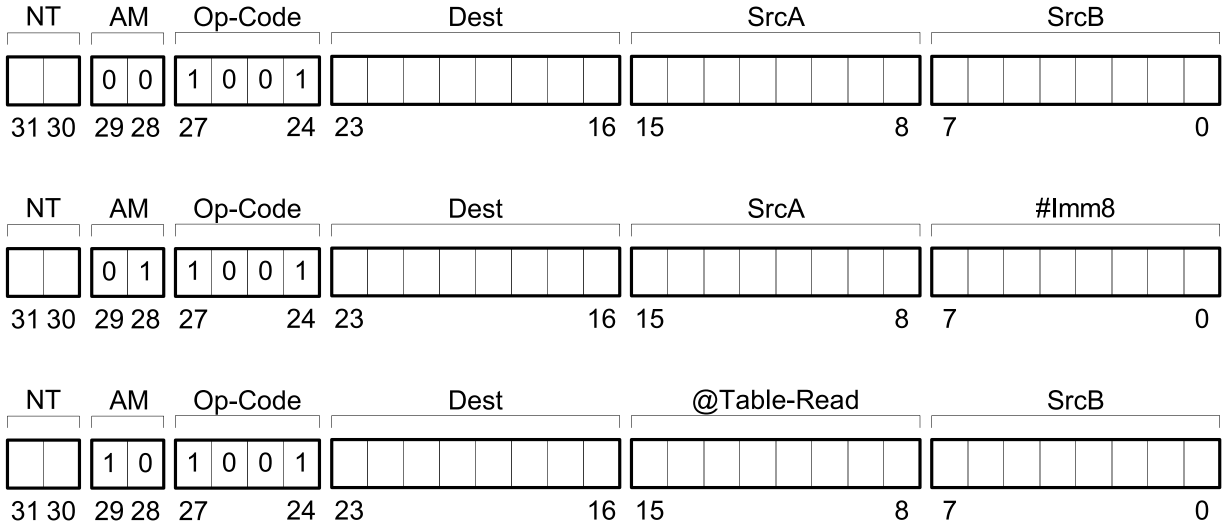
SUB subtracst (without barrow from carry flag) the contents SrcB from the contents of SrcA and then stores the results in Dest. Permitted addressing modes for SrcA include direct, indirect, or table-read from program memory. Permitted addressing modes for SrcB include direct, indirect, or 8-bit immediate. If SrcB is 8-bit immediate, it is zero-extended to 32 bits by the hardware before the addition takes place. Dest addressing mode is always either direct or indirect. STATUS flags Z, C, V and N are affected as shown below.

STATUS flags:

N—set if MSB of result is 1 (negative), reset otherwise
C—cleared if a barrow of Carry flag occured, otherwise unchanged
V—set if XOR of the result's two MSBs = 1, reset otherwise
Z—set if result is zero, reset otherwise

5.3.18 SUBB

Figure 5—33. SUBB Instruction



SUBB subtracts (with barrow from carry flag) the contents SrcB from the contents of SrcA and then stores the results in Dest. Permitted addressing modes for SrcA include direct, indirect, or table-read from program memory. Permitted addressing modes for SrcB include direct, indirect, or 8-bit immediate. If SrcB is 8-bit immediate, it is zero-extended to 32 bits by the hardware before the addition takes place. Dest addressing mode is always either direct or indirect. STATUS flags Z, C, V and N are affected as shown below.

STATUS flags:

N—set if MSB of result is 1 (negative), reset otherwise
C—cleared if a barrow of Carry flag occured, otherwise unchanged
V—set if XOR of the result's two MSBs = 1, reset otherwise
Z—set if result is zero, reset otherwise

5.3.19 TAN

Figure 5—34. TAN Instruction

NT	AM	Op-Code	Dest	SrcA	
	0 0	1 1 1 0			0 0 0 0 0 0 0 0
31 30	29 28	27 24	23 16	15 8	7 0

NT	AM	Op-Code	Dest	#Imm16 {#SrcA, #SrcB}
	1 1	1 1 1 0		0 0 0 0 0 0 0
31 30	29 28	27 24	23 16	15 0

TAN returns the 32-bit, floating-point tangent of the 9-bit, signed integer contained in SrcA, or 16-bit immediate value, and stores the result in Dest. The integer contained in SrcA or the 16-bit immediate value must be in the range of +/- 360 degrees. STATUS flags Z and N are affected as shown below.

STATUS flags:

N—set if MSB of result is 1 (negative), reset otherwise
C—not affected
V—not affected
Z—set if result is zero, reset otherwise

5.3.20 XOR

Figure 5—35. XOR Instruction

NT	AM	Op-Code	Dest	SrcA	SrcB
	0 0	0 0 1 1			
31 30	29 28	27 24	23 16	15 8	7 0

NT	AM	Op-Code	Dest	SrcA	#Imm8
	0 1	0 0 1 1			
31 30	29 28	27 24	23 16	15 8	7 0

NT	AM	Op-Code	Dest	@Table-Read	SrcB
	1 0	0 0 1 1			
31 30	29 28	27 24	23 16	15 8	7 0

XOR performs a bit-wise eXclusive-OR of the contents of SrcA and the contents of SrcB. Permitted addressing modes for SrcA include direct, indirect, or table-read from program memory. Permitted addressing modes for SrcB include direct, indirect, or 8-bit immediate. If SrcB is 8-bit immediate, it is zero-extended to 32 bits by the hardware before the addition takes place. Dest addressing mode is always either direct or indirect. STATUS flags Z and N are affected as shown below.

STATUS flags:

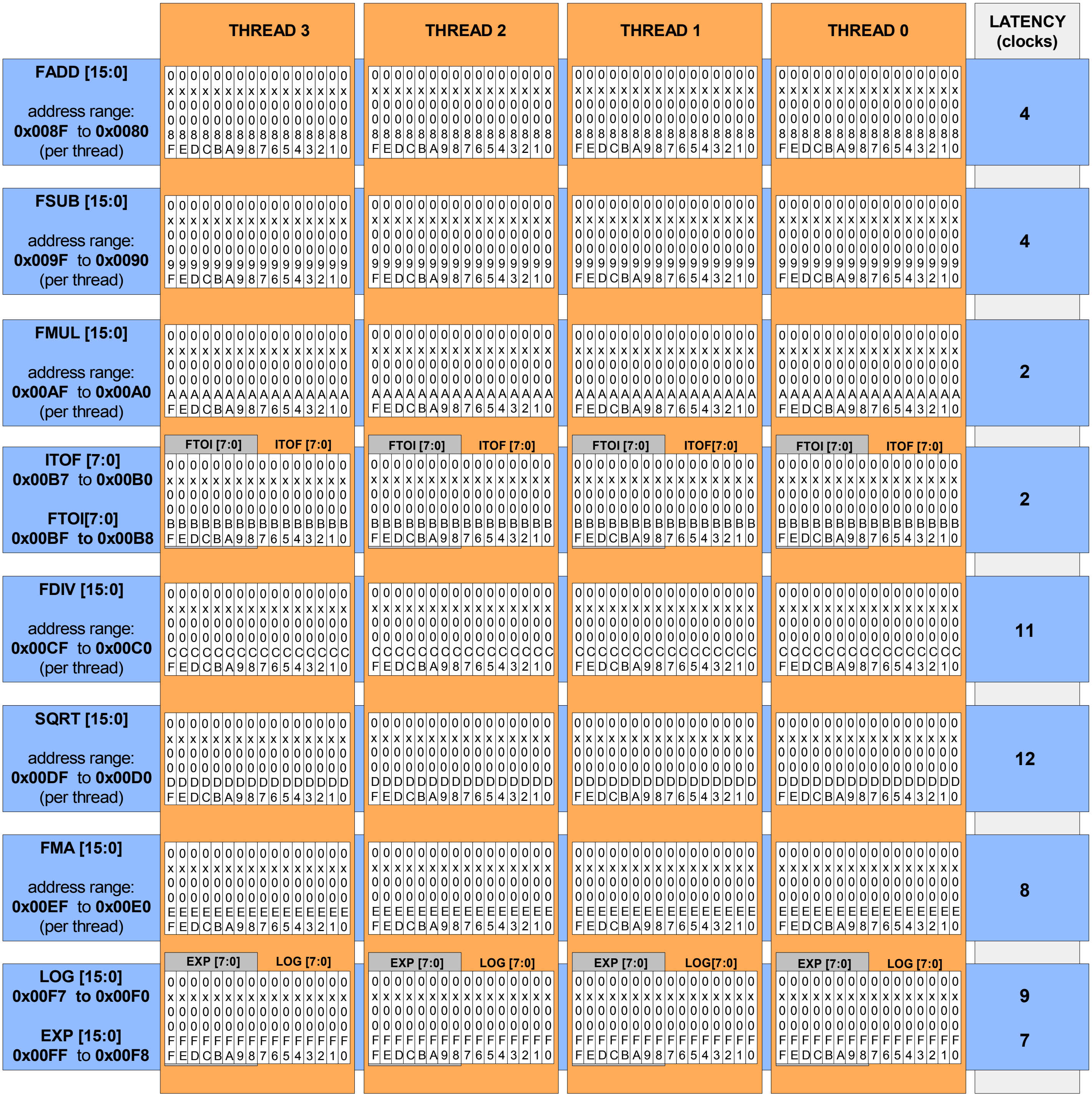
N—set if MSB of result is 1 (negative), reset otherwise
C—not affected
V—not affected
Z—set if result is zero, reset otherwise

Chapter 6

Floating-Point Operators

6.1 Memory-Mapped Floating-Point Operators

Figure 6—1. Floating-Point Operator Memory-Map



	THREAD 3	THREAD 2	THREAD 1	THREAD 0	LATENCY (clocks)
FADD [15:0] address range: 0x008F to 0x0080 (per thread)	0x008F 0x008E 0x008D 0x008C 0x008B 0x008A 0x0089 0x0088 0x0087 0x0086 0x0085 0x0084 0x0083 0x0082 0x0081 0x0080	0x008F 0x008E 0x008D 0x008C 0x008B 0x008A 0x0089 0x0088 0x0087 0x0086 0x0085 0x0084 0x0083 0x0082 0x0081 0x0080	0x008F 0x008E 0x008D 0x008C 0x008B 0x008A 0x0089 0x0088 0x0087 0x0086 0x0085 0x0084 0x0083 0x0082 0x0081 0x0080	0x008F 0x008E 0x008D 0x008C 0x008B 0x008A 0x0089 0x0088 0x0087 0x0086 0x0085 0x0084 0x0083 0x0082 0x0081 0x0080	4
FSUB [15:0] address range: 0x009F to 0x0090 (per thread)	0x009F 0x009E 0x009D 0x009C 0x009B 0x009A 0x0099 0x0098 0x0097 0x0096 0x0095 0x0094 0x0093 0x0092 0x0091 0x0090	0x009F 0x009E 0x009D 0x009C 0x009B 0x009A 0x0099 0x0098 0x0097 0x0096 0x0095 0x0094 0x0093 0x0092 0x0091 0x0090	0x009F 0x009E 0x009D 0x009C 0x009B 0x009A 0x0099 0x0098 0x0097 0x0096 0x0095 0x0094 0x0093 0x0092 0x0091 0x0090	0x009F 0x009E 0x009D 0x009C 0x009B 0x009A 0x0099 0x0098 0x0097 0x0096 0x0095 0x0094 0x0093 0x0092 0x0091 0x0090	4
FMUL [15:0] address range: 0x00AF to 0x00A0 (per thread)	0x00AF 0x00AE 0x00AD 0x00AC 0x00AB 0x00AA 0x00A9 0x00A8 0x00A7 0x00A6 0x00A5 0x00A4 0x00A3 0x00A2 0x00A1 0x00A0	0x00AF 0x00AE 0x00AD 0x00AC 0x00AB 0x00AA 0x00A9 0x00A8 0x00A7 0x00A6 0x00A5 0x00A4 0x00A3 0x00A2 0x00A1 0x00A0	0x00AF 0x00AE 0x00AD 0x00AC 0x00AB 0x00AA 0x00A9 0x00A8 0x00A7 0x00A6 0x00A5 0x00A4 0x00A3 0x00A2 0x00A1 0x00A0	0x00AF 0x00AE 0x00AD 0x00AC 0x00AB 0x00AA 0x00A9 0x00A8 0x00A7 0x00A6 0x00A5 0x00A4 0x00A3 0x00A2 0x00A1 0x00A0	2
ITOF [7:0] 0x00B7 to 0x00B0 FTOI[7:0] 0x00BF to 0x00B8	FTOI [7:0] 0x00BF 0x00BE 0x00BD 0x00BC 0x00BB 0x00BA 0x00B9 0x00B8 ITOF [7:0] 0x00B7 0x00B6 0x00B5 0x00B4 0x00B3 0x00B2 0x00B1 0x00B0	FTOI [7:0] 0x00BF 0x00BE 0x00BD 0x00BC 0x00BB 0x00BA 0x00B9 0x00B8 ITOF [7:0] 0x00B7 0x00B6 0x00B5 0x00B4 0x00B3 0x00B2 0x00B1 0x00B0	FTOI [7:0] 0x00BF 0x00BE 0x00BD 0x00BC 0x00BB 0x00BA 0x00B9 0x00B8 ITOF[7:0] 0x00B7 0x00B6 0x00B5 0x00B4 0x00B3 0x00B2 0x00B1 0x00B0	FTOI [7:0] 0x00BF 0x00BE 0x00BD 0x00BC 0x00BB 0x00BA 0x00B9 0x00B8 ITOF [7:0] 0x00B7 0x00B6 0x00B5 0x00B4 0x00B3 0x00B2 0x00B1 0x00B0	2
FDIV [15:0] address range: 0x00CF to 0x00C0 (per thread)	0x00CF 0x00CE 0x00CD 0x00CC 0x00CB 0x00CA 0x00C9 0x00C8 0x00C7 0x00C6 0x00C5 0x00C4 0x00C3 0x00C2 0x00C1 0x00C0	0x00CF 0x00CE 0x00CD 0x00CC 0x00CB 0x00CA 0x00C9 0x00C8 0x00C7 0x00C6 0x00C5 0x00C4 0x00C3 0x00C2 0x00C1 0x00C0	0x00CF 0x00CE 0x00CD 0x00CC 0x00CB 0x00CA 0x00C9 0x00C8 0x00C7 0x00C6 0x00C5 0x00C4 0x00C3 0x00C2 0x00C1 0x00C0	0x00CF 0x00CE 0x00CD 0x00CC 0x00CB 0x00CA 0x00C9 0x00C8 0x00C7 0x00C6 0x00C5 0x00C4 0x00C3 0x00C2 0x00C1 0x00C0	11
SQRT [15:0] address range: 0x00DF to 0x00D0 (per thread)	0x00DF 0x00DE 0x00DD 0x00DC 0x00DB 0x00DA 0x00D9 0x00D8 0x00D7 0x00D6 0x00D5 0x00D4 0x00D3 0x00D2 0x00D1 0x00D0	0x00DF 0x00DE 0x00DD 0x00DC 0x00DB 0x00DA 0x00D9 0x00D8 0x00D7 0x00D6 0x00D5 0x00D4 0x00D3 0x00D2 0x00D1 0x00D0	0x00DF 0x00DE 0x00DD 0x00DC 0x00DB 0x00DA 0x00D9 0x00D8 0x00D7 0x00D6 0x00D5 0x00D4 0x00D3 0x00D2 0x00D1 0x00D0	0x00DF 0x00DE 0x00DD 0x00DC 0x00DB 0x00DA 0x00D9 0x00D8 0x00D7 0x00D6 0x00D5 0x00D4 0x00D3 0x00D2 0x00D1 0x00D0	12
FMA [15:0] address range: 0x00EF to 0x00E0 (per thread)	0x00EF 0x00EE 0x00ED 0x00EC 0x00EB 0x00EA 0x00E9 0x00E8 0x00E7 0x00E6 0x00E5 0x00E4 0x00E3 0x00E2 0x00E1 0x00E0	0x00EF 0x00EE 0x00ED 0x00EC 0x00EB 0x00EA 0x00E9 0x00E8 0x00E7 0x00E6 0x00E5 0x00E4 0x00E3 0x00E2 0x00E1 0x00E0	0x00EF 0x00EE 0x00ED 0x00EC 0x00EB 0x00EA 0x00E9 0x00E8 0x00E7 0x00E6 0x00E5 0x00E4 0x00E3 0x00E2 0x00E1 0x00E0	0x00EF 0x00EE 0x00ED 0x00EC 0x00EB 0x00EA 0x00E9 0x00E8 0x00E7 0x00E6 0x00E5 0x00E4 0x00E3 0x00E2 0x00E1 0x00E0	8
LOG [15:0] 0x00F7 to 0x00F0 EXP [15:0] 0x00FF to 0x00F8	EXP [7:0] 0x00FF 0x00FE 0x00FD 0x00FC 0x00FB 0x00FA 0x00F9 0x00F8 LOG [7:0] 0x00F7 0x00F6 0x00F5 0x00F4 0x00F3 0x00F2 0x00F1 0x00F0	EXP [7:0] 0x00FF 0x00FE 0x00FD 0x00FC 0x00FB 0x00FA 0x00F9 0x00F8 LOG [7:0] 0x00F7 0x00F6 0x00F5 0x00F4 0x00F3 0x00F2 0x00F1 0x00F0	EXP [7:0] 0x00FF 0x00FE 0x00FD 0x00FC 0x00FB 0x00FA 0x00F9 0x00F8 LOG[7:0] 0x00F7 0x00F6 0x00F5 0x00F4 0x00F3 0x00F2 0x00F1 0x00F0	EXP [7:0] 0x00FF 0x00FE 0x00FD 0x00FC 0x00FB 0x00FA 0x00F9 0x00F8 LOG [7:0] 0x00F7 0x00F6 0x00F5 0x00F4 0x00F3 0x00F2 0x00F1 0x00F0	9 7

Each SYMPL FP32X-AXI4 Shader core presently includes eight memory-mapped floating-point operators shared by all of its four threads. All but the ITOF and FTOI operators occupy sixteen memory locations in a given thread's zero-page memory map at locations 0x080 through 0x0FF. The floating-point result buffer memory map is shown in *Figure 6—1* above.

For each floating-point operator that requires two operands, there are two memory-mapped, 32-bit, (write-only) input registers that must be written to simultaneously using the MOV instruction only. Single-operand operators have one memory-mapped, (write-only) input register and thus may be written-to using any instruction. For every memory-mapped input register there is a corresponding 34-bit (read-only) result buffer residing at the same location.

6.2 IEEE754-2008 Compliance

One of the design goals of the SYMPL FP32X is to provide a simple way to carry out GP-GPU-compute operations that are fully IEEE754-2008 compliant. This section describes the key aspects of the SYMPL FP32X core design that enable such compliance and lists any known exclusions/non-compliance with the standard. With exception to remainder and binary <—> decimal operators (which the implementer must perform in software), all operators are implemented in hardware.

6.2.1 Default Exception Handling

The memory-mapped Status register for each of the four threads contain, among others, the five exception flags specified in the IEEE754-2008 specification: invalid ("INVALID"), divide-by-zero ("DIVBY0"), overflow ("OVERFLW", underflow-inexact("UNDRFLW", and inexact (INEXCT"). Each floating-point operator contains logic that signals if one of these exceptions arise. Except for the inexact exception, under default exception handling, the corresponding above-named flag is automatically set (and remains set until explicitly cleared by the implementer) whenever such exception is detected and signaled.

6.2.2 Default Exception Handling for Invalid Operation

Under default exception handling, attempts to write out-of-bounds operands or signaling NaNs to a floating-point operator (except for ITOF) will signal an invalid operation exception and automatically set the INVALID flag in the Status register and deliver a non-signaling ("quiet") NaN with diagnostic payload as described below. Additionally, the INVALID flag, once set, will remain set unless explicitly cleared by the implementer.

6.2.3 Non-Signaling (Quiet) NaNs

The following diagram shows the bit fields ("payload") of the quiet NaN delivered under default exception handling for invalid operation exception.

Figure 6—2. Quite NaN with Diagnostic Payload

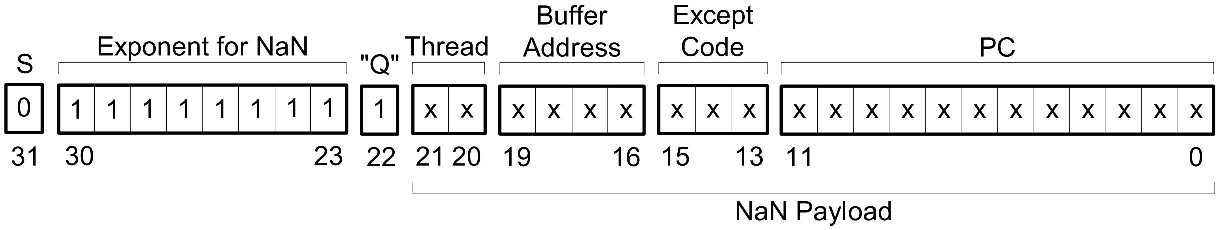


Table 6—1 below gives a description of the fields that make up the quiet NaN delivered under default exception handling for invalid operation exceptions.

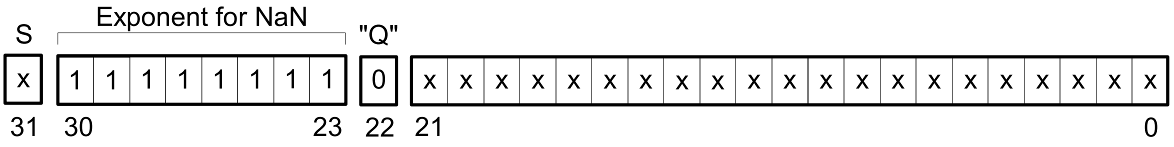
Table 6—1. Quiet NaN Field Description

Bit-position	Field Name	Description
[31]	Sign	Sign bit cleared to zero
[30:23]	Exponent	8'hFF
[22]	Quiet	1'b1
[21:20]	Thread	Thread number that wrote the operands to the signaling operator
[19:16]	Buffer Address	The operator input register address written to and result address read from. For example: FDIV_6 means FDIV address 6
[15:13]	Exception Code	Exception code indicates the type of invalid operation signaled
	3'b000	Signaling NaN
	3'b001	Multiply operands out of bounds, i.e., mult(0, INF) or mult(INF, 0)
	3'b010	Fused-multiply (FMA) operands out of bounds
	3'b011	Add or subract (or FMA-add) operands out of bounds
	3'b100	Division operands out of bounds, div(0, 0) or div(INF, INF)
	3'b101	Remainder operands out of bounds, rem(x, y), when y is zero or x is infinite (and neither is NaN)
	3'b110	Square-root operand out of bounds, operand is less than zero
	3'b111	Conversion result does not fit in destination format, or a converted finite number yields (or would yield) infinite result
[12]	Reserved	This bit is reserved for future PC expansion
[11:0]	PC	Program counter of the instruction that wrote the invalid operand(s)

6.2.4 Signaling NaNs

Signaling NaNs are NaNs that have their “Q” bit (bit-position [22]) cleared to “0”. They are useful for testing/exercising alternate immediate/delayed exception handling routines. If written to any floating-point operator (except ITOF) input register, such will induce an invalid operation exception and will be handled according to the default, alternate-immediate, or alternate-delayed handler in place at the time written. The diagram in *Figure 6—3* below shows the bit-fields of the signaling NaN. Fields with “x” in them mean such bits are “don't care”.

Figure 6—3. Signaling NaN



6.2.5 Alternate Immediate Exception Handling

Alternate-immediate exception handling is enabled by setting its corresponding enable bit in the Status register. During reset, all alternate exception handler enable bits in the Status register of each thread are cleared to “0”, thereby enabling **default** handling for all exceptions. If a given exception has its corresponding alternate exception handling enable bit set “**and”** the corresponding alternate-**delayed** exception enable bit is **cleared**, anytime such exception is thereafter signaled, an interrupt will be “**immediately**” generated and vectored to that exception's handler routine anytime the corresponding exception is signaled. “Immediate” means that as soon as the exception is detected/signaled, an interrupt is generated right then, before the results of the operation are actually written into its corresponding result buffer.

Depending on the strategy employed by the implementer, the corresponding exception flag might or might not be set in the Status register because such flag is not automatically set during alternate exception handling, whether it be immediate or delayed, because it is up to the implementer to set it in the handler routine.

6.2.6 Alternate Delayed Exception Handling

Alternate-delayed exception handling is enabled by setting both its corresponding alternate exception handling enable **and** alternate-delayed exception handling enable bits in the Status register. During reset, all alternate exception handler enable bits in the Status register of each thread are cleared to “0”, thereby enabling **default** handling for all exceptions. If a given exception has **both** its corresponding alternate exception handling enable bit and the corresponding alternate-**delayed** exception enable bit set to “1”, anytime such exception is thereafter signaled, an interrupt will be generated only during the time the corresponding results are read from its respective result buffer, at which time the thread is vectored to that exception's handler routine.

In the context of the SYPML FP32X, alternate “delayed” exception handling means that such exception handling does not take place until after the excepted results are read from its corresponding result buffer even though such exception was signaled (but not actually acted upon) before the results of the operation is actually written into its corresponding result buffer.

Depending on the strategy employed by the implementer, the corresponding exception flag might or might not be set in the Status register because such flag is not automatically set during alternate exception handling, whether it be immediate or delayed, because it is up to the implementer to set it in the handler routine.

6.2.7 Alternate Delayed Exception Capture Registers

Provided as a means for processing alternate delayed exception results for proper delivery, four capture registers (Capture registers 0 through 3) can be read during the interrupt service routine to retrieve information related to the exception. During issuance of a given alternate delayed exception interrupt, the following information is automatically captured off their respective buses and registered for retrieval during that exception's handler (note, for more information about these registers, refer to Section 5.1.10 :

a) Capture Register 0 will contain the “A” side results read from the corresponding operator result buffer.

b) Capture Register 1 will contain the “B” side result read from the corresponding operator result buffer, assuming that the read cycle that initiated the alternate delayed exception was a dual-result read/write operation.

c) Capture Register 2 will contain both source address A (and source address B if dual-operand read) along with the corresponding two-bit, prioritized exception code for for each in the case of divide-by-zero, overflow or underflow-inexact.

d) Capture Register 3 will contain the program counter (PC) of the instruction that was fetched and executed that performed the read of the result buffer that caused the alternate delayed exception, along with the thread number, rounding mode, addressing mode, and destination address.

The purposed of the alternate delayed exception capture registers is to provide the information necessary to properly handle the alternate delayed exception and, if the implementer so desires, deliver an substituted result to the original destination address.

6.2.8 Default and Directed Rounding Modes

SYMPL FP32X supports all four rounding modes specified by the standard, with round-to-nearest being the default. Directed rounding is accomplished with the MOV instruction used for writing to a given operator's operand input register. For example"

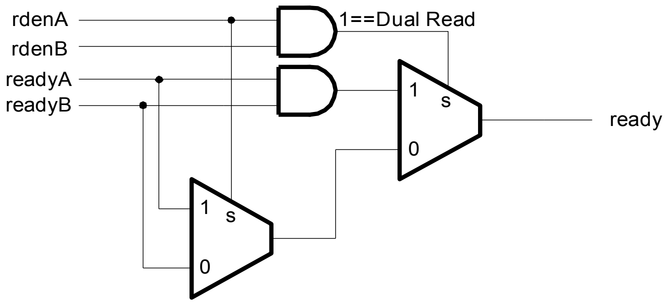
```
MOV     FDIV_3, oprndA, oprndB      ;divide oprndA by oprndB and round to nearest
MOV.1  SQRT_1, oprndA               ;square root oprndA and round to positive infinity
MOV.2  FADD_4, oprndA, oprndB       ;add oprndB to oprndA and round to negative infinity
MOV.3  FMUL_7, oprndA, oprndB       ;mult oprndA by oprndB and round to zero
```

6.3 Floating-Point Result Buffer Semaphores

Each result buffer location also has its own semaphore that indicates whether results are ready. When a given result location is accessed by the BTBC or BTBS instruction, the semaphore is read (in bit-position [15] of the word being read) instead of the actual result. If the semaphore tests a logic “1”, this indicates that the result is ready, if “0” it is not ready. Once it is determined that the result is ready, then it may be retrieved with any instruction. Care must be taken not to attempt to read a result buffer location that never had a write to its corresponding input register(s), because the semaphore in such cases will never be set and, consequently, a TIMER time-out will eventually generate a non-maskable interrupt.

The MOV instruction has a special mode for accessing the result buffer region that automatically tests a given result buffer semaphore without the use of the above-mentioned BTBC/BTBS instructions. If a single-operand or dual-operand MOV instruction is used to read a result buffer location, the PC will automatically rewind to the MOV instruction fetch address and re-fetch the MOV instruction if the semaphore (ready flag) is not set. Such will continue until the ready flag is set. Consequently (and technically speaking) the Shader's instruction pipeline never stalls, in that it is always fetching and executing an instruction. *Figure 6—4* below shows the logic employed by the hardware to determine the ready state of a given result buffer location during a read operation.

Figure 6—4. Floating-Point Result-Ready Logic

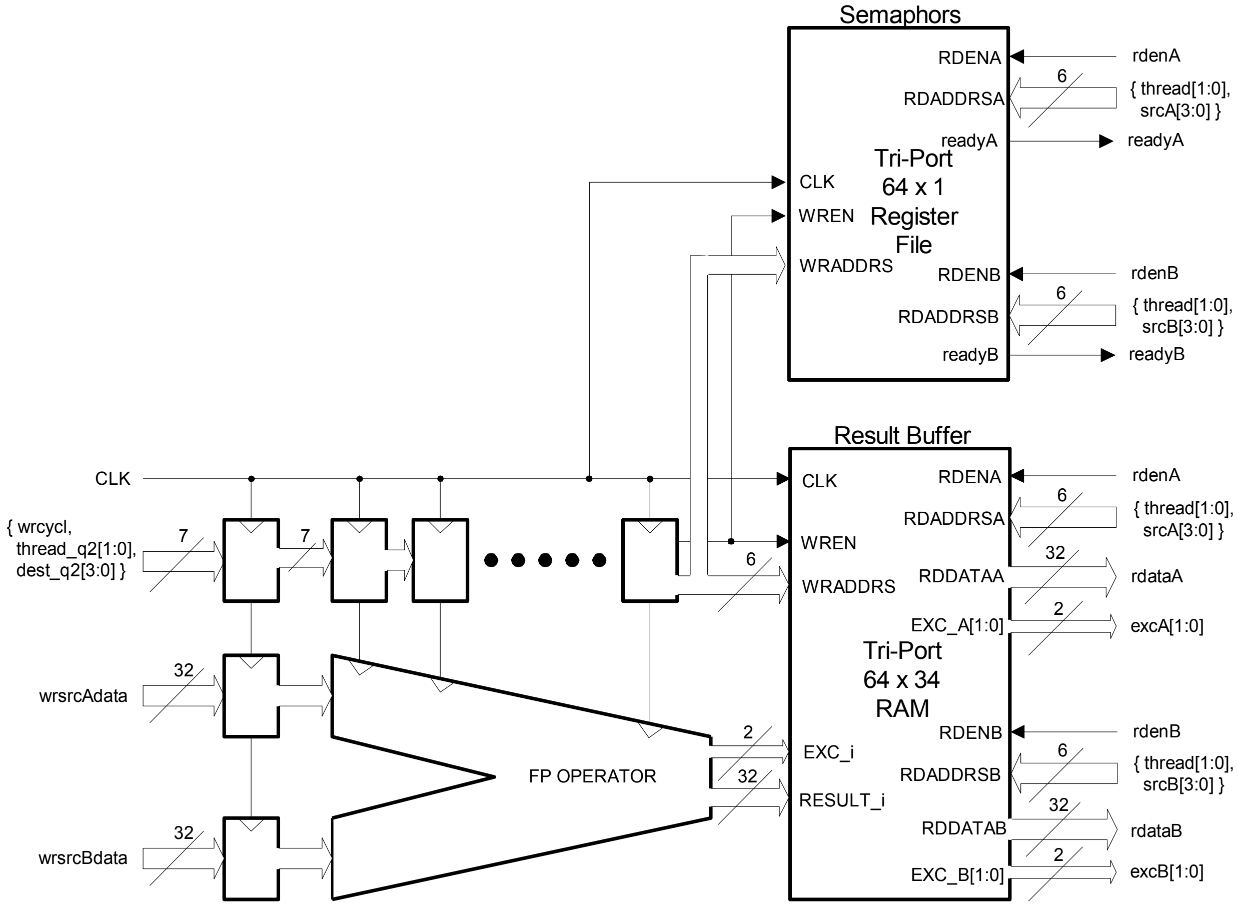


6.4 Floating-Operator Pipeline

Each floating-point operator has their own pipeline which can be from two to fifteen stages deep, depending on the operator, and are decoupled from a given Shader core's instruction pipeline. For instance, the ITOF operator is only two stages deep and the SQRT operator is fifteen stages deep, taking two clocks and fifteen clocks respectively to complete.

For floating-point operators that accept only one operand, any instruction can be used to store a result directly into a given floating-point operator's input register. For floating-point operators that accept two operands, only the MOV instruction can be used to load that floating-point operator's input register. *Figure 6—5* below shows a the arrangement of one of the dual-operand, floating-point pipeline, corresponding semaphore and result buffer memory.

Figure 6—5. Floating-Point Operator Pipeline and Result Buffers



6.5 Customized Floating-Point Operators

While the SYMPL FP32X-AXI4 design presently employs FloPoCo operators from the very popular open-source FloPoCo, VHDL floating-point library, the wrappers for corresponding operators can be easily modified so that floating-point operators from various other IP providers can be used in lieu of FloPoCo. Below are links to various other floating-point operator providers you may be interested in experimenting with. However, with that stated, at the end of the day, you will most likely conclude that you will not be able to find operators that out-perform FloPoCo.

http://flopoco.gforge.inria.fr

The LIBHDLFLTP VHDL library can be downloaded for free at the following website and freely used under the GNU General Public License version 2.0 (GPLv2) or later version:

http://sourceforge.net/projects/libhdlfltp/

VFLOAT ("The Northeastern Variable precision Floating-point library") is another open-source floating-point library (in VHDL) containing at least a few operators that might be suitable for this application and can be found at the following site, or at least re-directed to a site where it can be downloaded, is at the following link:

http://www.coe.neu.edu/Research/rcl/projects/floatingpoint/index.html

If you are not comfortable with using an open-source VHDL floating-point library, there are a number of other IP providers available where you can obtain at least some of the required operators. Here are links to a few of them (not listed in any order of preference):

http://www.dcd.pl

http://www.hitechglobal.com

http://www.think-silicon.com

http://www.zipcores.com

http://www.synopsys.com/dw/buildingblock.php

When choosing a floating-point operator library, bear in mind that the operators must be fully pipelined, meaning that such operators must be able to accept a new input every clock cycle. From the perspective of a given Shader engine, the only module ports that are required are clock, operand inputs and result output, with reset and enables being optional.

6.6 Using A Different IP Provider's Floating-Point Operators

In the event that you need to employ a different provider's floating-point library for whatever reason, it is very likely that the latencies for their library will not match that of the FloPoCo library. In such cases, simply add or subtract the number of delay registers that make up the write-address/write-enable FIFOs mentioned above, so that the total equals the latency of the operator you intend to use.

The FloPoCo operators do not include an enable input or functional reset input (although a reset module port is present). If the operator you intend to use does have these, simply add a reset module port to the wrapper and connect it to the operator's reset and the Shader global reset line and tie the enable to the write enable of the wrapper module port.

6.7 Floating-Point Operator Descriptions

As mentioned above, the SYMPL FP32X-AXI4 RTL includes wrappers for FADD/FSUB, FMUL, FMA, FDIV, SQRT, DOT, EXP, LOG, ITOF and FTOI. Each of these operators employ pipelines of different length, the longest being the SQRT operator at eleven clocks and shortest being the ITOF and FTOI operators at one clock each. FADD and FSUB share the same operator circuit and is three clocks deep. FMUL is just one clock deep, FDIV is ten clocks deep and EXP is six clocks deep, while LOG is eight clocks deep. The above-stated numbers do not include the extra clock required to write results in each operator's memory-mapped result buffer.

6.8 How to Use the Floating-Point Operators

With exception to the ITOF, FTOI, LOG and EXP operators, each floating-point operator occupies sixteen consecutive locations in the Shader’s zero-page memory map starting at location 0x080 and continuing up to 0x0FF as shown in *Figure 6—1* above. To employ a given operator that requires two operands, use the MOV instruction and store the dual operands in the desired input register according to the map in *Figure 6--1*. Once written, the results for that operation will be available for reading at that same location it was written to once the number of delay/pipe clocks have transpired. For example:

```
MOV    FADD_0, oprndA, oprndB        ;move both oprndA and oprndB to FADD_0 input register
MOV    FADD_1, oprndC, oprndD        ;move both oprndC and oprndD to FADD_1 input register
MOV    FADD_2, oprndE, oprndF        ;move both oprndE and oprndF to FADD_2 input register
MOV    FMUL_0, FADD_0, FADD_1        ;multiply results of first two FADDs
```

In the example above, the MOV instruction is used to store operands oprndA and oprndB at the FADD0 input register address. Once three clocks have transpired from the write cycle of the MOV instruction, results are available for reading at the same physical address.

```
MOV    AR0, #FADD_0                  ;load AR0 with pointer to FADD base address
MOV    AR1, #xvector                 ;load AR1 with xvector base address
MOV    AR2, #yvector                 ;load AR2 with yvector base address
RPT    #7                            ;execute 8 times (i.e., 1 plus the RPT #n specified)
MOV    *AR0++, *AR1++, *AR2++        ;FADD the vector
MOV    FMUL_0, FADD_0, FADD_1        ;multiply results of FADD_0 and FADD_1
MOV    FMUL_1, FADD_2, FADD_3
MOV    FMUL_2, FADD_4, FADD_5
MOV    FMUL_3, FADD_6, FADD_7
```

In the above example, auto-post-increment indirect addressing is used in combination with RPT and MOV to operate on multiple operands in a vector with the results of those operations being immediately multiplied. Because the latency for FADD is only five clocks, by the time the RPT/MOV is done, the first results, as well as successive results, are (or will be) available for reading by the time their respective read cycle comes along.

Reset, Initialization and Interrupts

Chapter 7

7.1 Reset and Initialization

The FP32X-AXI core employs asynchronous registers that follow the Verilog RTL asynchronous coding as follows:

```
reg [31:0] somereg;
always@(posedge CLK or posedge RESET) begin
      somereg <= 32'h0000_0000;
end
else begin
      somereg <= somereg + 1'b1;
end
```

Almost all core registers employ asynchronous resets (or none at all) mainly to make it easier to move from memory-based FPGAs to custom ASIC if desired. Register resets also facilitate running simulations during development, in that it places the core in a known state prior to the release of the reset input.

The FP32X-AXI4 takes its reset from the active low, AXI4 “ARESETn” input, where it is immediately inverted to active high and re-named, “RESET” before being used in the circuit.

During RESET, most of the core's internal registers are cleared to 0, while some registers are initialized to some other value. The Program Counter (PC) for each thread is just one of these, in that each of the PCs is pre-set to the value 0x0100, which is the vector location for each thread's initialization sequence.

Upon release of RESET, each thread begins fetching from location 0x0100 in program memory. Since the non-maskable interrupt (NMI), floating-point exception interrupts (invalid operation, divide-by-zero, overflow, underflow-inexact and inexact), and general-purpose interrupt (IRQ) vector locations are at 0x0101 through 0x0103 respectively, the instruction at location 0x0100 should contain a MOV PC, #SPIN_IDLE instruction, wherein SPIN_IDLE is the address of each thread's SPIN_IDLE routine.

While in SPIN_IDLE, a thread should test its parameter/data memory semaphore for non-zero to see if the Coarse-Grained Scheduler/CPU has pushed a new packet in for processing. If a non-zero semaphore is read, the semaphore should be loaded into the PC, as the semaphore itself is the entry-point address to the routine that is to do the processing for that particular parameter/data packet.

As a first step in the routine, the thread should clear its DONE flag in its STATUS register to signal the CGS/CPU that the thread is now busy processing the data. Upon completion of the processing, the thread should then clear the original semaphore location back to zero and set its DONE flag back to 1 to signal the CGS/CPU that processing is complete and the thread is now available to process another packet.

For an example of the foregoing, refer to the listing file named “FP321_test1.LST”, which is included in the RTL sorce-code package at the GitHub repository for this core.

7.2 Interrupts

Each of the four threads of a given Shader core have three prioritized, vectored, active high, interrupt sources (listed in the order of highest priority): non-maskable interrupt (NMI), floating-point exception interrupts (invalid operation, divide-by-zero, overflow, underflow-inexact and inexact), and general-purpose interrupt (IRQ). These are described below.

7.2.1 Non-Maskable Interrupt (NMI)

An NMI going high will cause a given thread to temporarily suspend execution of the current routine and automatically load its PC with 0x0101, where it fetches its respective NMI vector to begin processing its NMI service routine. The NMI input, like its name suggests, means it is non-maskable and is the highest priority, which itself cannot be interrupted.

Presently, the zero-count output of each thread's 20-bit TIMER output is tied directly to that thread's NMI input, such that, if the timer ever counts down to zero before a given task completes, that thread's NMI line will be asserted, forcing entry into that thread's NMI service routine as a type of safety-net.

The DONE bit of a given thread is used as a gate/qualifier for the timer, such that only when the DONE bit is cleared to 0 (i.e., the thread is BUSY) does the timer register decrement. The DONE bit is also gated with the NMI line, such that if DONE is active high, the NMI line is disabled.

Consequently, since the zero-count output of the timer is hard-wired to the thread's NMI line, one of the first operations a thread must perform before clearing its DONE bit (signaling it is now busy), is load the timer with a cycle count value that exceeds the estimated number of clock cycles needed to complete the routine. If the timer ever reaches the value of 0x00000, a non-maskable interrupt will be generated and the thread encountering it will automatically be vectored to its NMI in-service routine, where it can then recover from an excessive time exception and alert the CSG/CPU by writing an appropriate message into its respective packet memory and asserting the DONE bit/flag active high, which in turn should generate a CPU interrupt, if enabled.

After entering the thread's NMI service routine, among the first operations that must be performed is to save the PC COPY register to that threads private memory location 0x0001, which is expressly reserved for that purpose and must not be used by any other routine. The reason for this is two-fold. First, the thread must be able to restore its PC to the location where the interrupt occurred, in that any PC discontinuity that occurs while in-service (such as a branch, for example) will overwrite the PC COPY register. Second, there is internal in-service logic within the interrupt controller module that detects when an Return from Interrupt (RETI) occurs (since there is no specific RETI instruction for this purpose). The logic looks for a MOV PC, 0x0001 instruction, which it treats as an NMI-RETI instruction, which clears the NMI-in-service state.

7.2.2 Floating-Point Exception Interrupts

Each thread also has five prioritized, maskable floating-point exception interrupts (complete with their own vectors) for the five floating-point exceptions listed in the order of highest priority here: invalid operation, divide-by-zero, overflow, underflow-inexact and inexact.. At least three conditions must occur before any of these goes active: first, NMI must not be active or in-service; second, there must be a floating-point exception signaled that corresponds to one of the named exceptions; and third, the corresponding alternate exception handling bit in the Status register must be set to enable the interrupt.

For alternate-immediate exception handling, only the alternate exception handling enable bit need be set. For alternate-delayed exception handling, both the alternate exception handling and corresponding alternate-delayed exception handling bits need to be set. The main difference between alternate-immediate and alternate-delayed exception handling is that for alternate-immediate exception handling, the interrupt is asserted immediately, as soon as the exception is signaled and before results are automatically written into the operator's result buffer, whereas, for alternate-delayed exception handling, the interrupt is not asserted until the results are read out of the result buffer by the routine employing the operator in its normal course of processing. For information regarding these two alternate exception handling modes, refer to Section 6.2.

If the three conditions listed above exist, an appropriately-signaled floating-point exception will cause the current thread to suspend execution of the current routine and automatically load its PC with the vector for that exception's handler/service routine. The table below lists all the available interrupts, along with corresponding vector and reserved PC_COPY save addresses that must be used to save and restore the PC upon entry/exit.

Table 7—1. Interrupt Vectors and Corresponding Reserved PC_COPY Locations

(Prioritized) Interrupt Name	Vector Address	PC_Copy Save Address
RESET	0x0100	N/A
NMI	0x0101	0x0001
Invalid operation	0x0102	0x0002
Divide-by-zero	0x0103	0x0003
Overflow	0x0104	0x0004
Underflow-inexact	0x0105	0x0005
Inexact	0x0106	0x0006
IRQ	0x0107	0x0007

After entering the thread's interrupt service routine, among the first operations that must be performed is to save the PC COPY register to that threads private memory location expressly reserved for that purpose and must not be used by any other routine. The reason for this is two-fold. First, the thread must be able to restore its PC to the location where the interrupt occurred, in that any PC discontinuity that occurs while in-service (such as a branch, for example) will overwrite the PC COPY register. Second, there exists internal in-service logic within the interrupt controller module that detects when an Return from Interrupt (RETI) occurs (since there is no specific RETI instruction for this purpose). The logic looks for MOV PC, 0x00x instruction, which it treats as an RETI instruction, which automatically clears the in-service state.

7.2.3 General-Purpose Interrupt Request (IRQ)

All threads have their own maskable, general-purpose interrupt request input. IRQ is the lowest priority of the seven available interrupt sources and, therefore, any pending IRQ might be interrupted by an NMI or floating-point exception interrupt. An IRQ interrupt cannot interrupt an active NMI or floating-point exception interrupt while either of them are pending, active or in-service.

Since IRQ is maskable, its respective interrupt enable (bit 21 of that thread's STATUS register) must be set to 1 before a vectored IRQ service routine can be initiated. However, interrupt input and the IRQ input can be polled while their respective interrupt enables are cleared by reading bit 22 in that thread's STATUS register.

After entering the thread's IRQ service routine, among the first operations that must be performed is to save the PC COPY register to that threads private memory location 0x0007, which is expressly reserved for that purpose and must not be used by any other routine. The reason for this is two-fold. First, the thread must be able to restore its PC to the location where the interrupt occurred, in that any PC discontinuity that occurs while in-service (such as a branch, for example) will overwrite the PC COPY register. Second, there is internal in-service logic within the interrupt controller module that detects when an Return from Interrupt (RETI) occurs (since there is no specific RETI instruction for this purpose). The logic looks for MOV PC, 0x0007 instruction, which it treats as an IRQ-RETI instruction, which clears the IRQ-in-service state.

Coarse-Grained Scheduler (CGS)

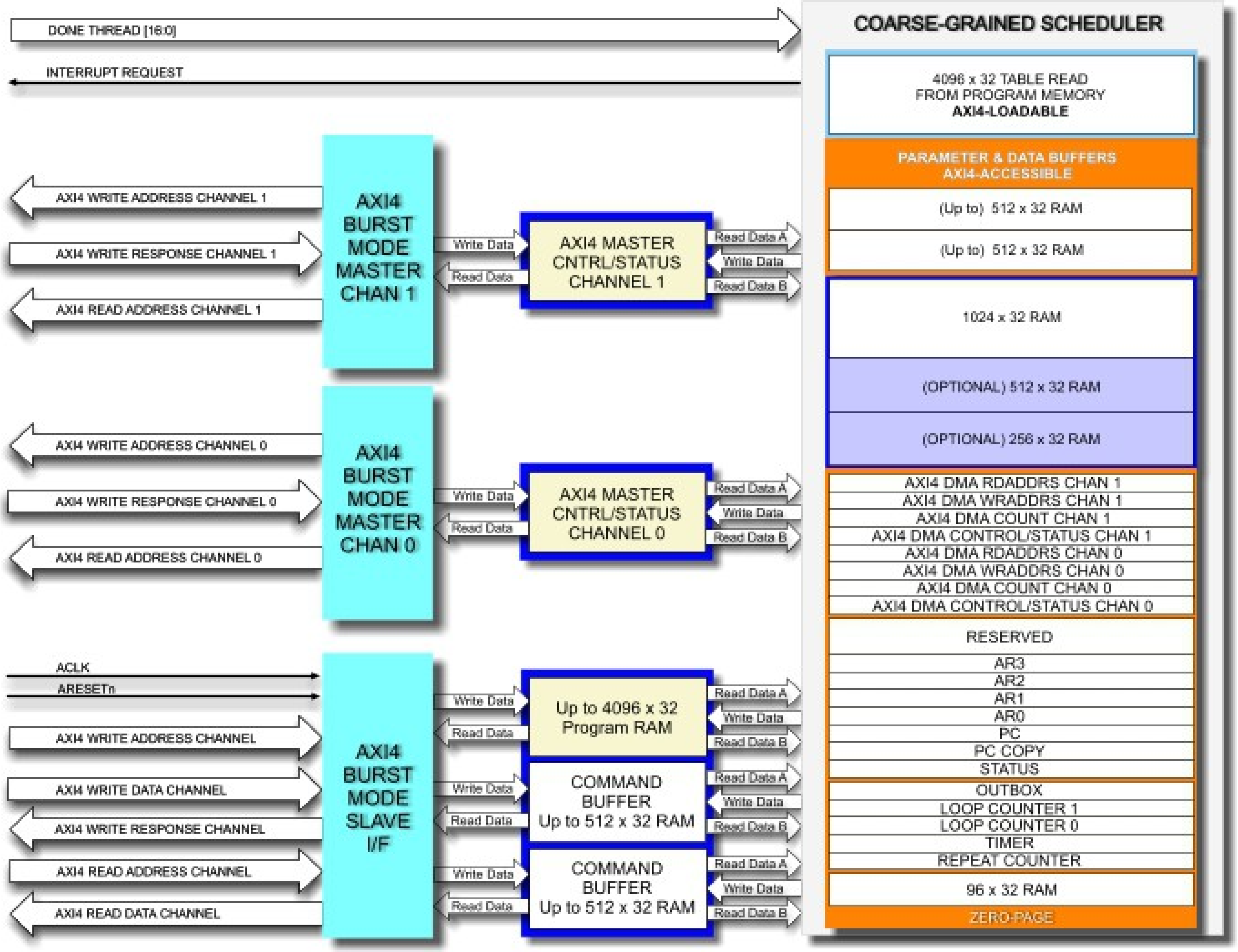
Chapter 8

8.1 (Optional) Coarse-Grained Scheduler/Load-Balancer

The SYMPL FP32X-AXI4 floating-point, quad-Shader engine is designed so that it can be easily bolted onto the FPGA embedded soft or hard core CPU of your choice, by way of existing AXI4 interconnect in your FPGA design. If after doing so you determine that your CPU is overburdened with the Course-Grained Scheduling/load-balancer task, an easy solution is to simply drop in the SYMPL FP32X-AXI4-CGS core to take over that function and thereby relieve your CPU of that function.

The SYMPL CGS is basically a "sawed-off" version of the a single Shader engine, with essentially the same native instruction-set as the SYMPL Shader engine, except it has no floating-point operators and is not multi-threaded like the SYMPL Shader core. It's basically a very fast, general-purpose, 32-bit RISC that includes a dual-channel AXI4 master controller coupled to an AXI4 slave interface so that commands can be pushed into its command buffer by the CPU, much like the parameter/data buffers of the Shader engine.

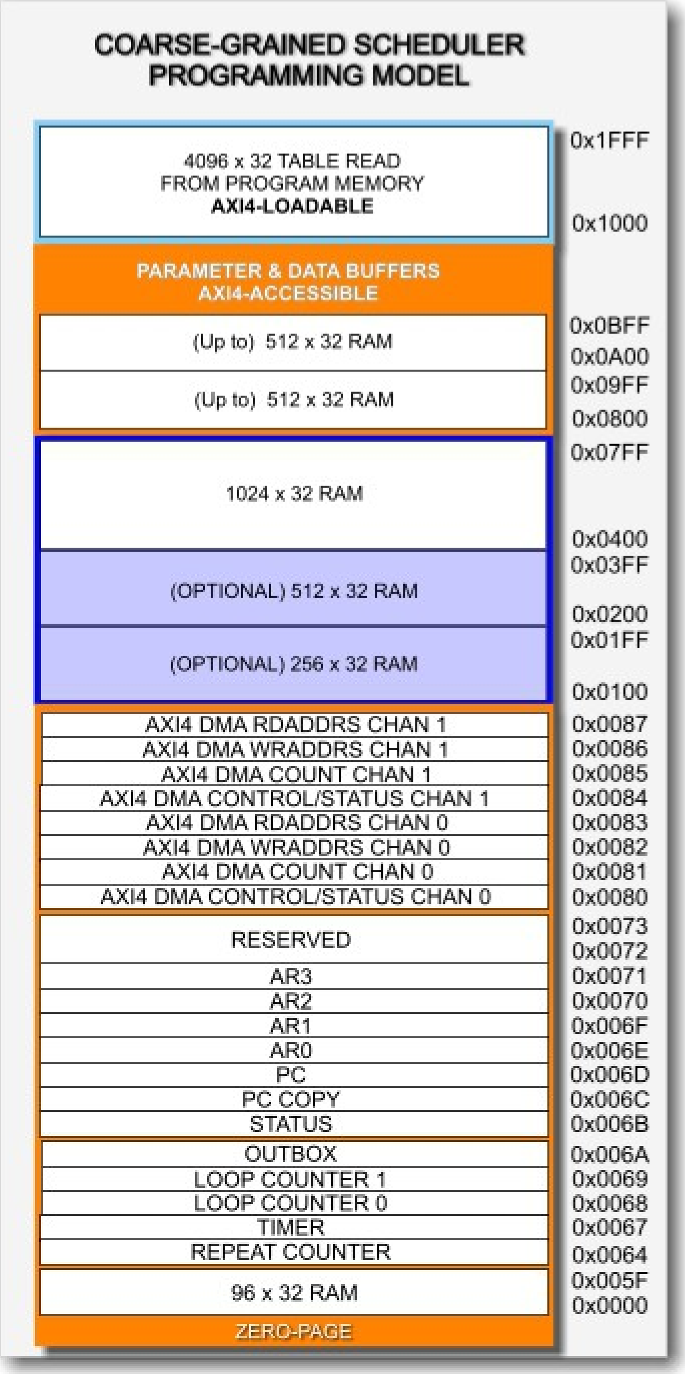
Figure 8—1. Coarse-Grained Scheduler Block Diagram



The block diagram in *Figure 8—1* above shows the arrangement of the AXI4 master and slave interfaces coupled to the SYMPL CGS, all of which are provided in the SYMPL FP32X-AXI4 CGS package.

This CGS package is presently not available for download at the SYMPL FP32X-AXI4 repository at GitHub. If you would like to add the SYMPL CGS to your design, please contact me at SYMPL.GPU@gmail.com and I will be happy to provide you with a license and Verilog RTL source-code under reasonable terms. The dual-channel AXI4 master DMA controller enables the SYMPL CGS to push a new packet into a target thread's parameter/data buffer while simultaneously pulling the results from previous processing transaction, thereby giving it the ability to minimize as much as possible initial latency. But in order to do that, the target system memory must either be dual-ported and/or comprise two separate memories that have their own read and write address and data buses.

Figure 8—2. CGS Programming Model



8.2 SYMPL CGS Programming Model

The SYMPL CGS's programming model is virtually identical to the SYMPL Shader core's programming model, except the SYMPL CGS has no floating-point operators or fine-grained scheduler. Also absent are the floating-point exception flags and LOCK bit in its STATUS register.

Instead, the SYMPL CGS has extra registers to handle dual-channel AXI4 master DMA transactions, as well as interrupt control and status register to enable setting up and servicing interrupts from up to sixteen of the Shader threads under its supervision.

Since the SYMPL CGS utilizes an instruction subset of the SYMPL Shader engine, the same assembler and debugger can be used to develop your custom CGS algorithm.

8.3 SYMPL FP32X-AXI4-CGS Registers

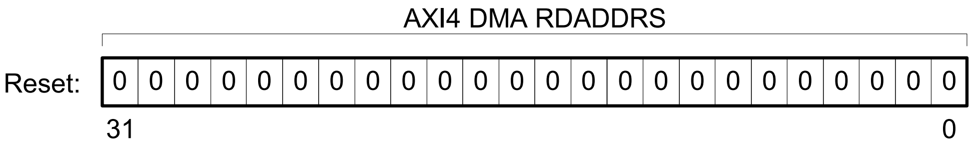
Registers unique to the SYML CGS include registers associated with the two AXI4 DMA master channels and a single, 32-bit register to enable monitoring interrupt requests from up to sixteen Shader threads. These registers are described below.

8.4 AXI4 DMA RDADDRS Registers

The DMA RDADDR registers are used to program the start address for a given master channel's DMA transfer. For the channel being employed for the transfer, the RDADDRS register's and WRADDRS register's start addresses, along with that channel's AXI4 configuration register must be programmed before that channels COUNT register is loaded with the count value. This is because writing the count value into the COUNT register automatically triggers the transfer. The RDADDRS register is both readable and writable.

The RDADDRS registers, along with the addresses where they reside, is shown in *Figure 8—3* below.

Figure 8—3. AXI4 DMA RDADDRS Registers



Logical Address (Channel 1): 0x0087

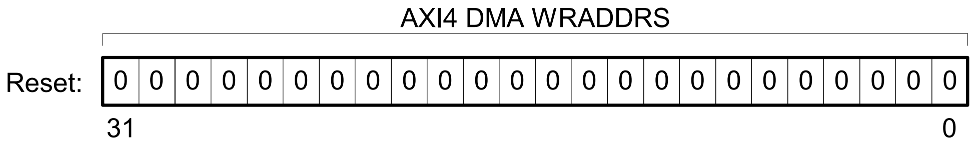
Logical Address (Channel 0): 0x0083

8.5 AXI4 DMA WRADDRS Registers

The DMA WRADDR registers are used to program the start address for a given master channel's DMA transfer. For the channel being employed for the transfer, the WRADDRS register's and WRADDRS register's start addresses, along with that channel's AXI4 configuration register must be programmed before that channels COUNT register is loaded with the count value. This is because writing the count value into the COUNT register automatically triggers the transfer. The WRADDRS register is both readable and writable.

The WRADDRS registers, along with the addresses where they reside, is shown in *Figure 8—4* below.

Figure 8—4. AXI4 DMA WRADDRS Registers



Logical Address (Channel 1): 0x0086

Logical Address (Channel 0): 0x0082

8.6 AXI4 DMA Configuration/Status Registers (DCSR)

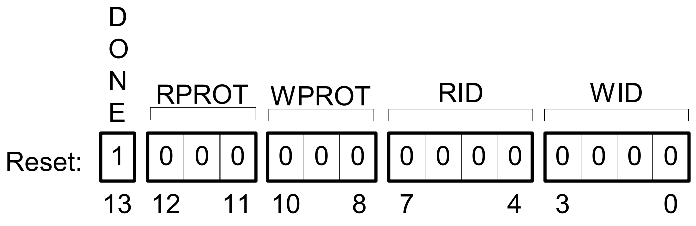
The DCSRs (one for each DMA channel) are used to configure each AXI4 DMA channel's read and write IDs as well as their read and write protection modes. Bit 13 of the DCSR is used to monitor the state of the transfer once initiated. On reset and on completion of a transfer, the DONE bit goes to logic 1 to indicate completion of the previously initiated transfer. When the COUNT register is loaded with a non-zero count value, the DONE bit automatically goes to logic 0 to indicate that a transfer is in progress and returns to logic 1 when complete. The DONE bit is read-only. All others are read-only when DONE is logic 0 (i.e., while a transfer is in progress), and both readable and writable when DONE is logic 1.

Like the RDADDRS and WRADDRS registers described earlier, the DCSR must be configured prior to writing the count value to the COUNT register, because writing to the COUNT register a non-zero value will automatically trigger the transfer.

Note: For more information regarding the definitions of the RPROT, WPROT, RID and WID parameters, refer to the official "AXI4 Protocol v1.0 Specification", which can be downloaded in .pdf form from the Internet.

Figure 8—5 below shows the DCSRs and respective addresses where they reside.

Figure 8—5. AXI4 DMA DCSR Registers



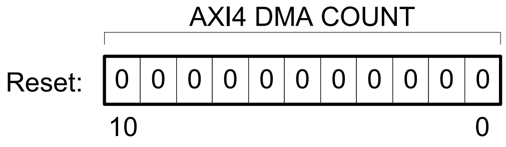
Logical Address (Channel 1): 0x0084

Logical Address (Channel 0): 0x0080

8.7 AXI4 DMA COUNT Register (DCR)

The DCR is used to specify the number of aligned 32-bit words to be transfer for a given transaction. Each DMA channel has one. The DCR must be the last register written-to for a given DMA transfer, because doing so automatically triggers the transaction, provided that the value written is non-zero. Writing a zero value to the DCR has no effect. The DCR is both readable and writable and may be sampled anytime a transfer is in progress. The DCR as well as the addresses where they reside are provided in *Figure 8—6* below.

Figure 8—6. AXI4 DMA Count Registers



Logical Address (Channel 1): 0x0085

Logical Address (Channel 0): 0x0081

8.8 SYMPL CGS AXI-4 DMA Slave Interface

The SYMPL CGS memory-mapped slave interface gives the CPU the ability to push commands into one of the CGS's two command buffers. An example command comprises at least a program entry-point for the routine or task that the CGS is to use to process the requested task, the address in system memory where the parameters and data are located as well as the length or word-count of the data and and some kind of ID or code identifying the source of the parameters and data that is requesting the processing task.

Typically, the entry-point or PC value submitted also acts as the semaphore that CPU signals the CGS with to indicate that a command is available for processing. Because of this, and from the standpoint of the CPU, the entry-point location (i.e., semaphore) in the command buffer should be initialized to the value, 0x0000 as a first step, and thereafter, should be the last item written into the command buffer with the actual entry-point value that the CGS uses to load its PC with to begin processing the command.

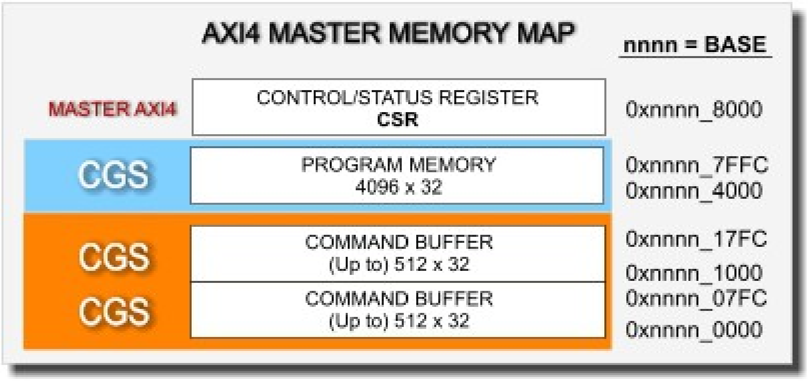
For a more detailed explanation on how just one example protocol might be implemented, refer to the previous section covering this topic for the Shader core AXI4 slave DMA interface.

8.9 CGS AXI4 MEMORY MAP

The CGS AXI4 memory-map is shown in *Figure 8—7* below. It comprises two, 512-word command buffers, (write-only) direct access to the CGS' program memory space, and a Control/Status Register (CSR) used primarily by the CPU for resetting the CGS, enabling interrupts and monitoring the state of the CGS' DONE flag. Write-only access to the CGS' program memory space gives the CPU the ability load and modify the routines used by the CGS, which can take place during initialization and/or on-the-fly.

To retrieve or dump the contents of the CGS' program memory, the CPU must issue a command to the CGS to do so, and have a routine in the CGS' program memory to carry out the transfer.

Figure 8—7. AXI4 DMA Master Memory-Map



8.10 CGS AXI4 Slave DMA Interface CSR

The diagram in *Figure 8—8* below shows the memory-mapped AXI4 Control/Status Register (CSR). The CPU must use this register to reset and/or launch the CGS. On system reset, the CGSRST line is set to one and remains in that state until the CPU writes a 0 to that bit position.

The CGS interrupt enable (bit-3 of the CSR) is used by the CPU to enable generation of a system interrupt when the CGS' DONE bit goes active high. On reset, it is cleared to 0, disabling such interrupts.

The CGSDONE bit is read-only and gives the CPU the ability to monitor the state of the CGS' DONE flag without enabling an interrupt.

The other bits in the CGS CSR are reserved for debugging purposes.

Figure 8—8. AXI4 Slave DMA CSR Register

CSR

	CGSDONE	CGSIE	CGSSTP	CGSBRK	CGSRST
Reset:	1	0	0	0	1
	4	3			0
	R	RW	RW	RW	RW

nnnn = Base Address
Logical Address: 0xnnnn_8000