FP32X-AXI4

32-Bit Single-Precision

IEEE-754--"like" Compatible

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

Revision History

Date	Revision	Author	Comments
August 13, 2015	1.1	Jerry D. Harthcock	First draft with corrections
August 21, 2015	1.2	Jerry D. Harthcock	Inclusion of FloPoCo and their floating-point operator library; added section regarding FloPoCo operators; added table showing FloPoCo VHDL floating-point operators implemented in this design.
August 22, 2015	1.3	Jerry D. Harthcock	Updated result buffer graphic to show FloPoCo operator latency.
August 28, 2015	1.4	Jerry D. Harthcock	Revised graphics to show scalablity, by adding one to n AXI4 Shaders at a time

Preface

The purpose of this document is to provide an overview and designer reference for the SYMPL FP32X-AXI4 Quad-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--"like" compatible), 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 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--("like" compatible), fully pipelined, floating-point operators including: FADD, FSUB, FMUL, FDIV, SQRT, 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
FDIV	11
SQRT	12
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. The 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 0x7FFFFFFF.
- 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.
- Three interrupt sources per thread: non-maskable interrupt (NMI), maskable floating-point exception (EXC) for NaN, INFand DML, 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.
- IEEE-754—(“like” compatible) flags include normal signed zero (SZ), normal negative (NN), normal (NML), infinity (INF), not-a-number (NaN) and denormal (DML), which are automatically updated in a thread's STATUS register when a MOV instruction successfully reads. Other subnormals are presently not available.
- 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.

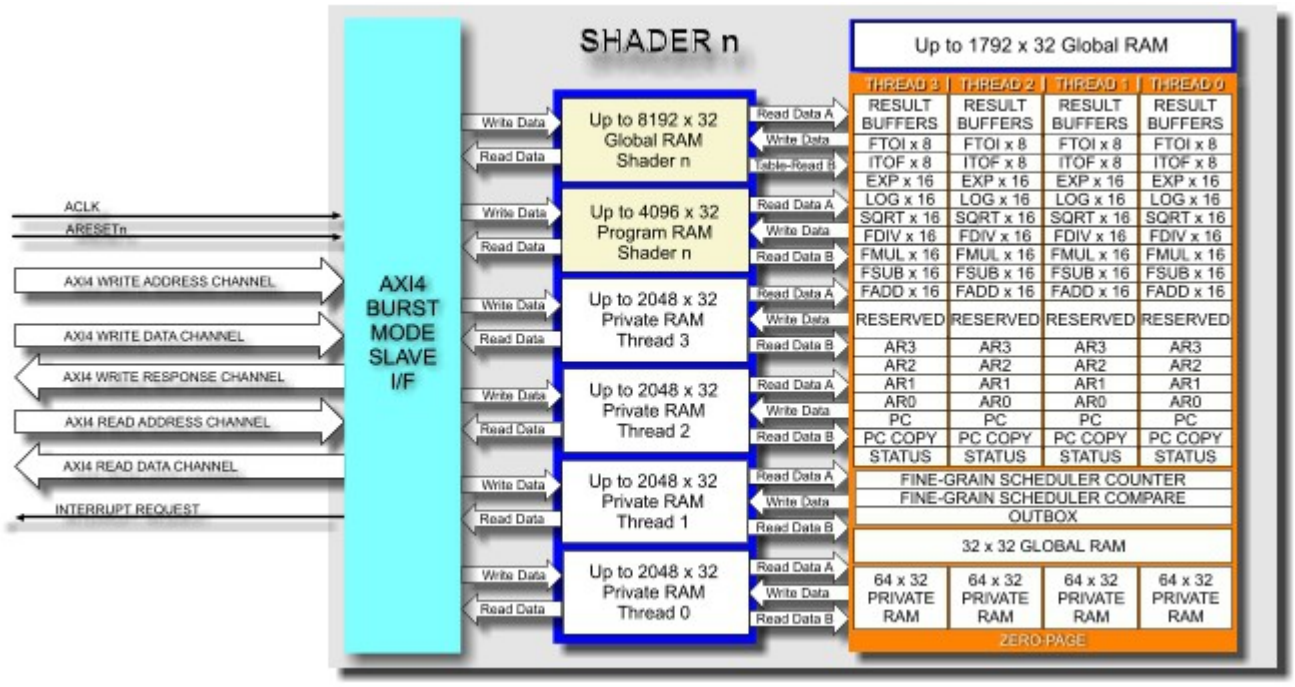
1.2 FP32X-AXI4 Block Diagram—Single-Shader Implementation

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



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. Scaleable Shader Diagram

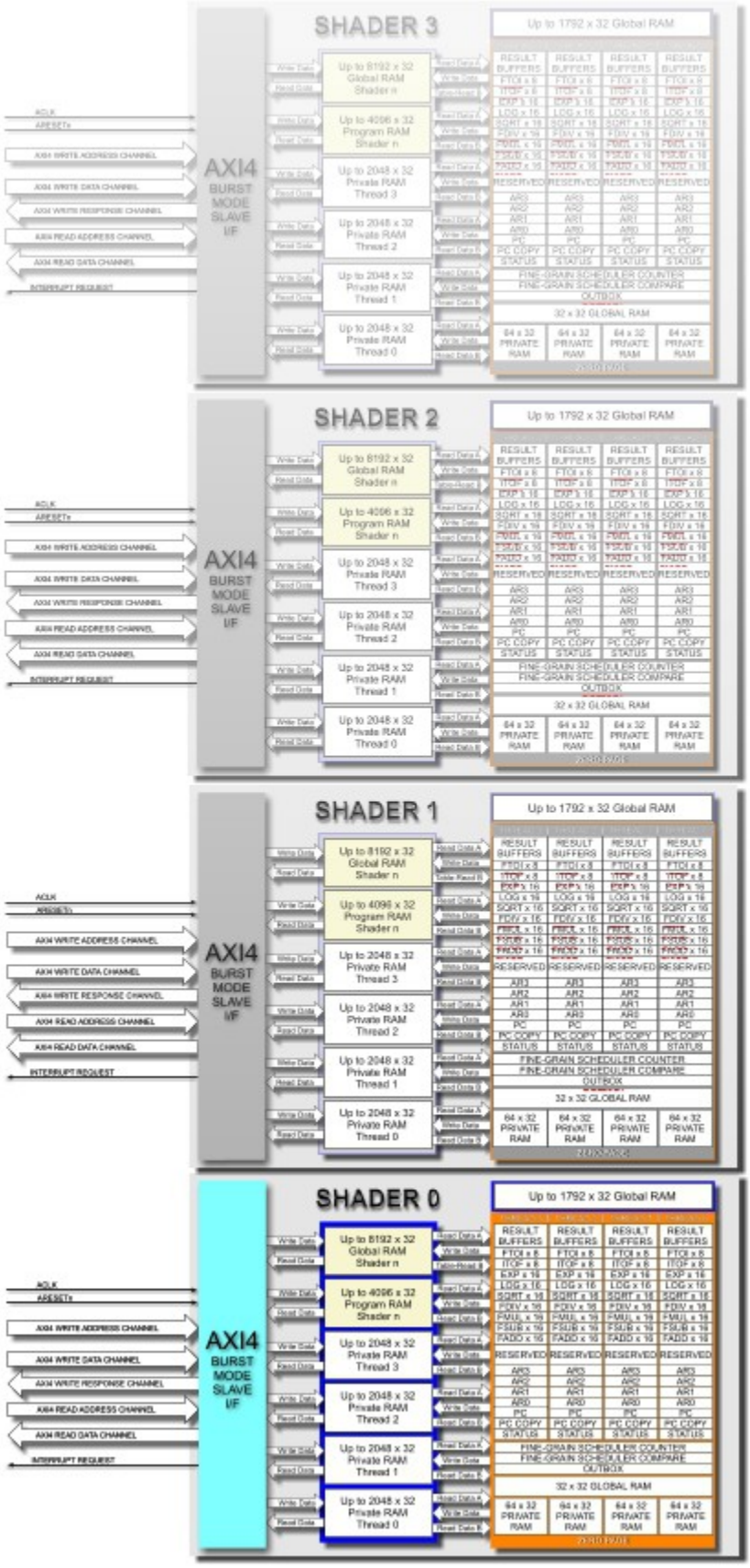
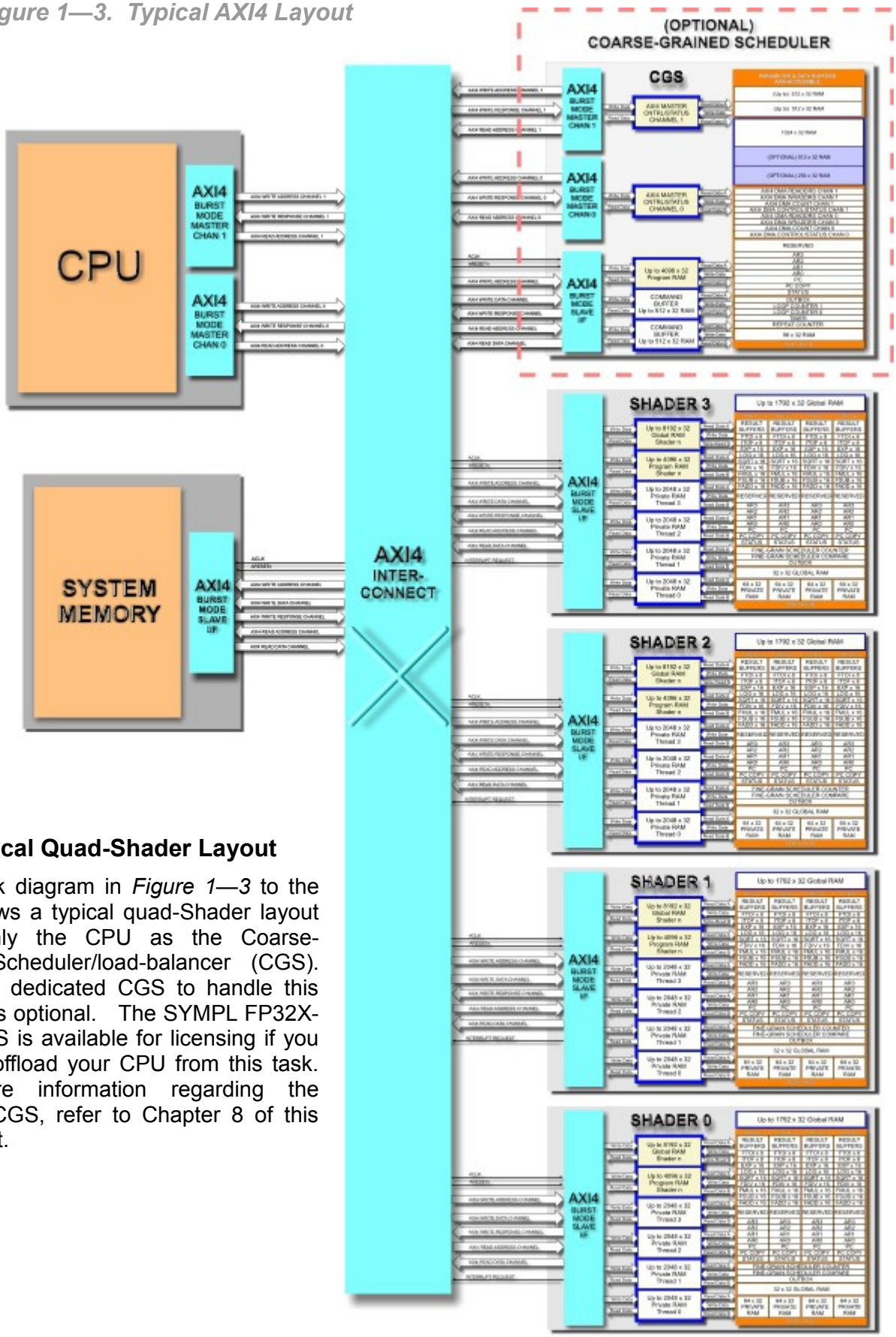


Figure 1—3. Typical AXI4 Layout



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
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.vhd	FADD operator; 3-stage pipe	func_add.v
Mul_Clk.vhd	FMUL operator; 1-stage pipe	func_mul.v
Div_Clk.vhd	FDIV operator; 10-stage pipe	func_div.v
Sqrt_Clk.vhd	SQRT operator; 11-stage pipe	func_sqrt.v
Log_Clk.vhd	LOG operator; 8-stage pipe	func_log.v
Exp_Clk.vhd	EXP operator; 6-stage pipe	func_exp.v
FP_To_FXP.vhd	FTOI operator; 1-stage pipe	func_ftoi.v
FXP_To_FP.vhd	ITOF operator; 1-stage pipe	func_itof.v
IEEE754_To_FP.vhd	IEEE754-to-FloPoCo format; combinatorial	Most of the above
FP_To_IEEE754.vhd	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.

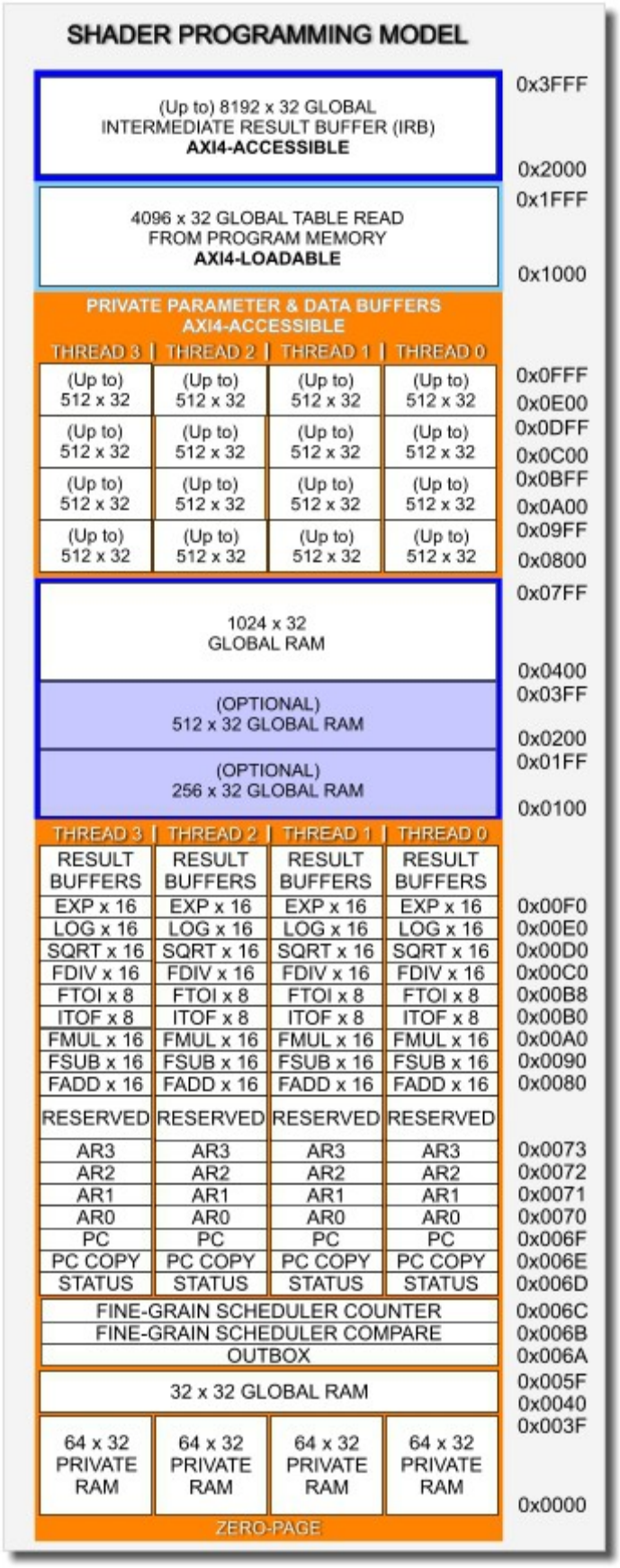
With all four threads of a given Shader scheduled for one clock each (out of four), the PCs of each thread appear “as if” there is no latency when a branch is taken, thereby hiding the fact that the instruction pipeline is three levels deep, meaning that the two instructions of a given thread that immediately 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



2.3 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”.

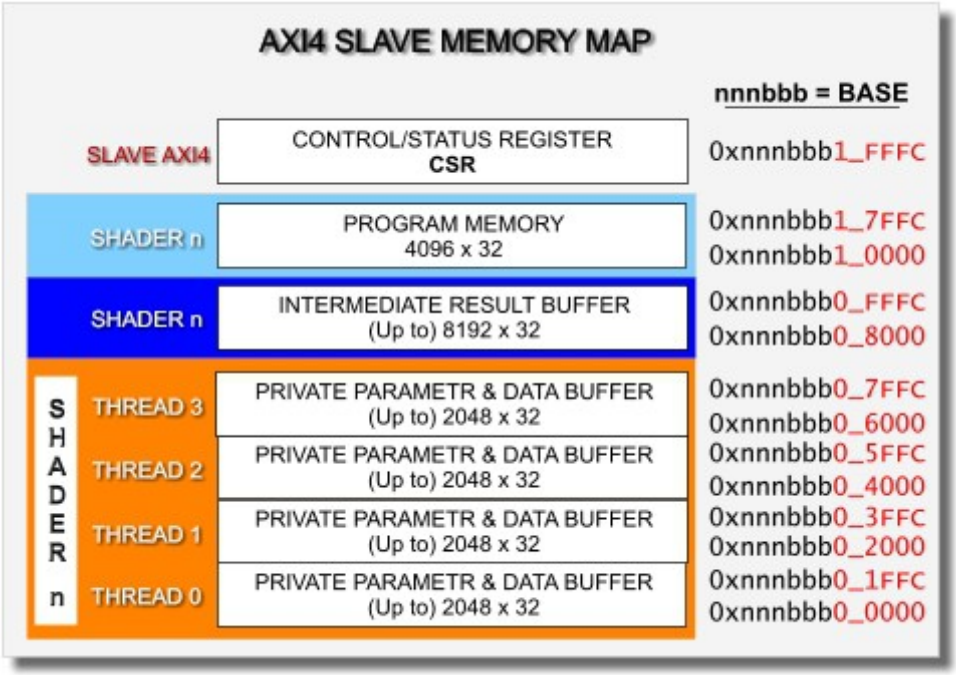
To extend address reach beyond zero-page, the indirect addressing mode is employed by using a given thread's auxiliary registers as indirect pointers. Each thread has four of them: AR3 to AR0.

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.

2.4 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.

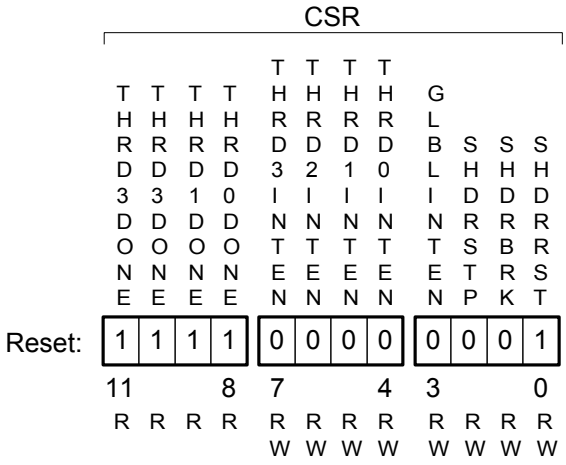
2.4.1 AXI4 Access to Shader Program Memory

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.

2.4.2 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



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

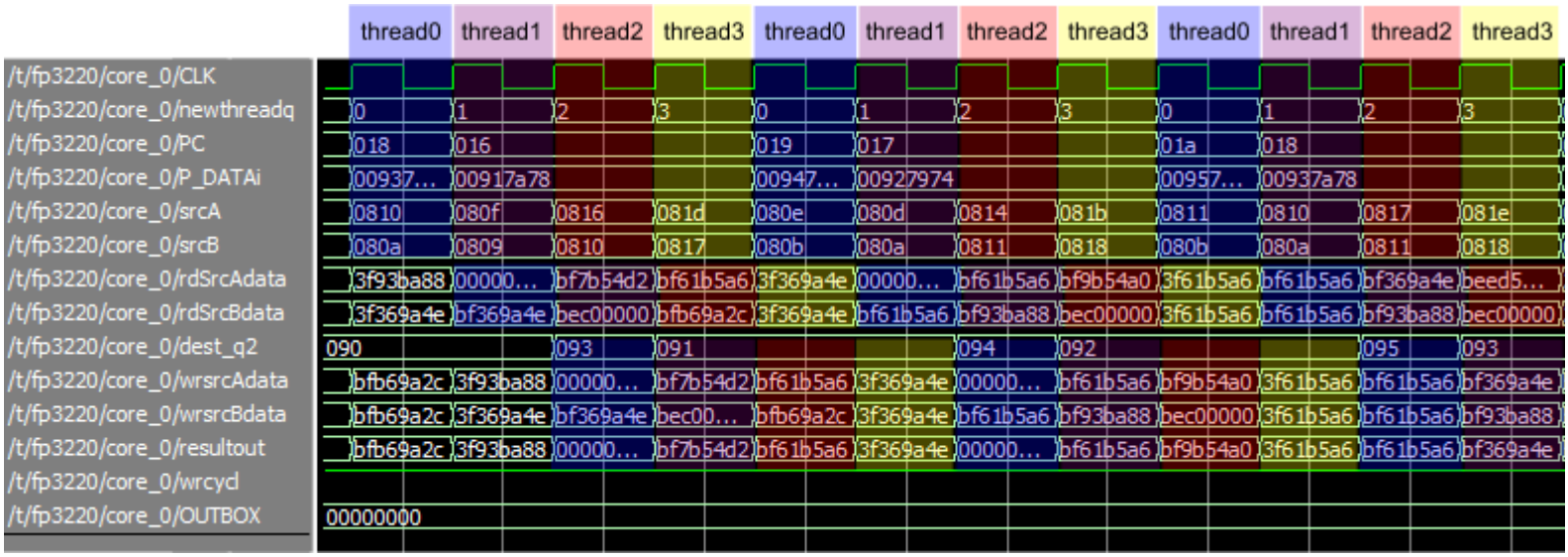
becomes active and the time slot for the originating thread will eventually work its way back around to the original thread that executed the soft-schedule and continue in the locked state until another soft-schedule occurs or the lock bit is clear, at which time interleave resumes.

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

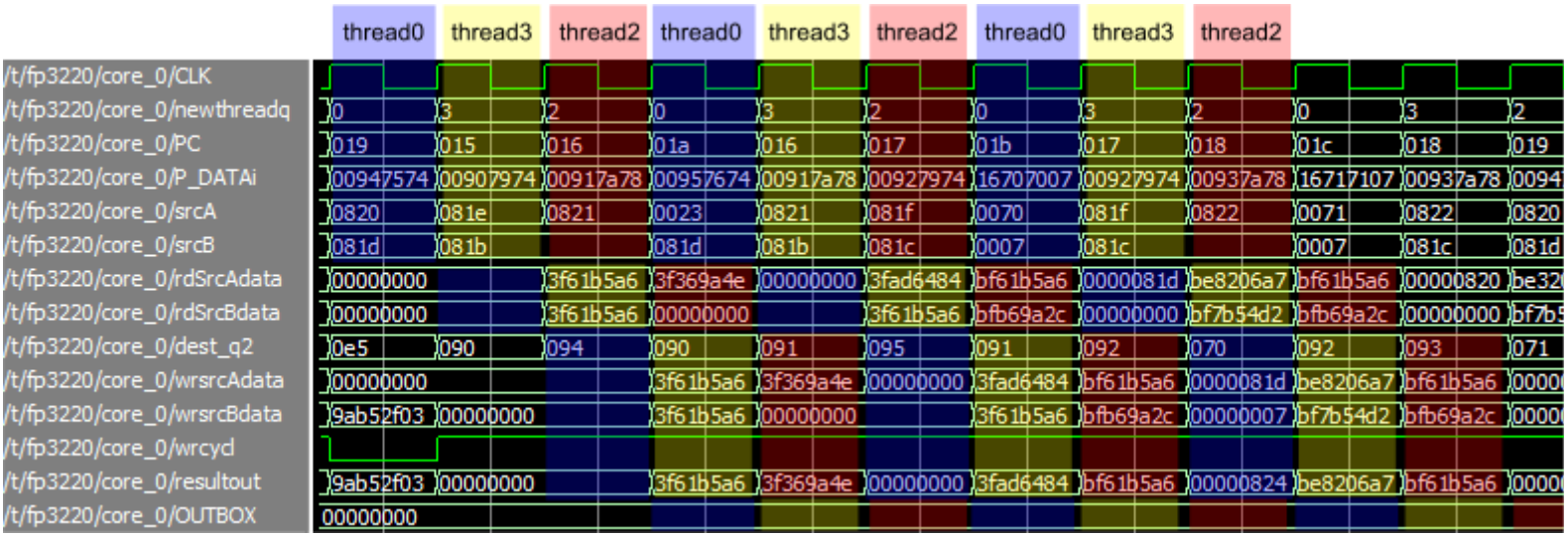


The example below shows actual scheduling behavior of a single Shader running just two threads (threads 0 and 2) with a interleave granularity of one clock each. To configure threads 0 and 2 for one-clock interleave, load the value 0x00020002 into the scheduling register.

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

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

Programmer's Reference

Chapter 4

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.

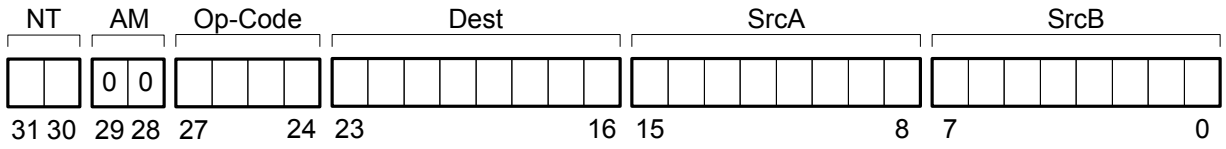
4.2 Instruction Word Format

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.

4.31 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".

4.32 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.

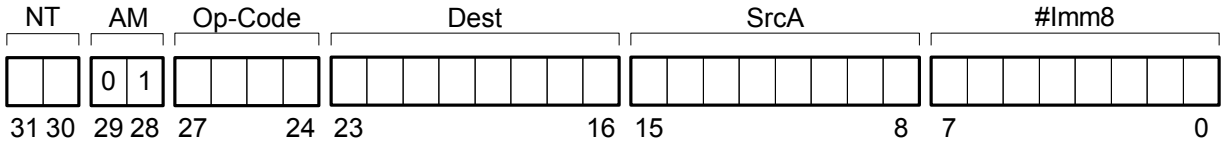
Table 4—1. Auxiliary Register Address-Modifier Translation

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).

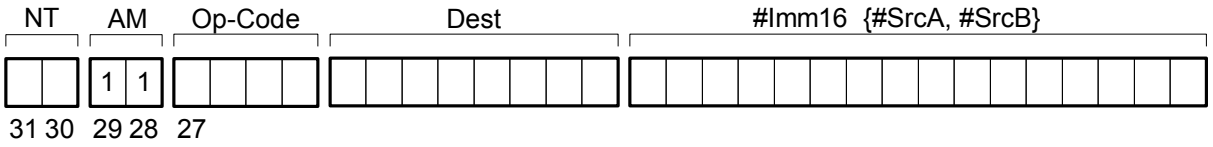
8-bit immediate mode utilizes the 8-bit constant constituting the immediately available last eight bits of the current instruction (i.e., the SrcB field only) without having to first retrieve it from data memory. The 8-bit value appearing in the #Imm8 field (SrcB position) is automatically zero-extended 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.

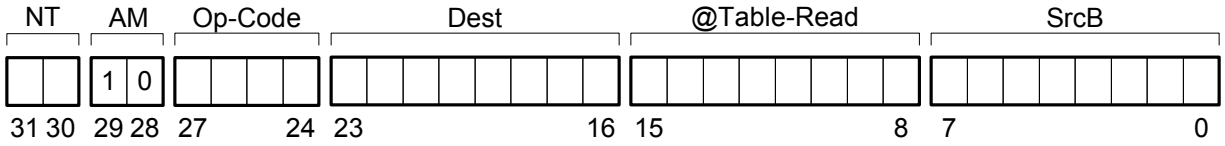
16-bit immediate mode utilizes the 16-bit constant constituting the immediately available last sixteen bits of the current instruction word without having to first retrieve it from data memory. The 16-bit value is automatically zero-extended to 32 bits before use.

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

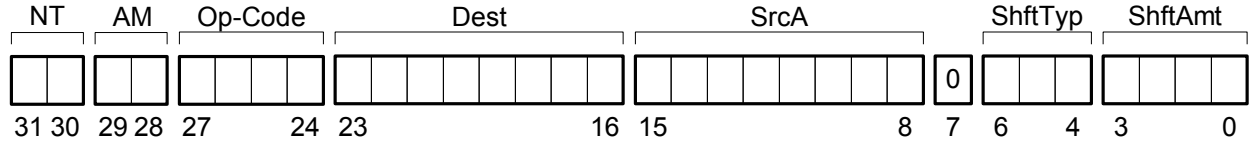
Table-read-indirect mode works just like the indirect addressing mode described previously, except it is used to access constants (or copy entire threads to data memory, for example) that may be present at any location in program memory. To employ this feature, set bit 12 of the auxiliary register used 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

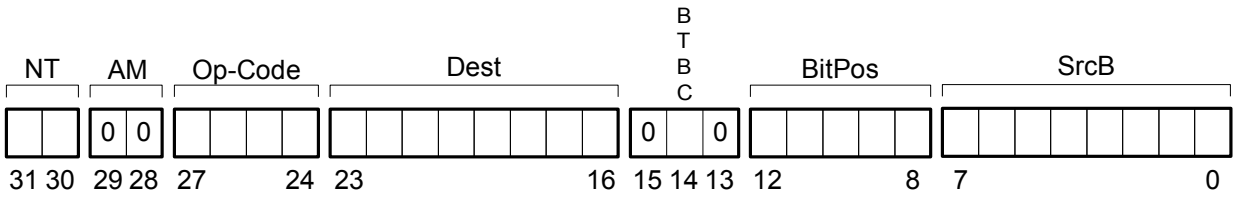


*Applies to SHFT only.

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

Next Thread **Description**

D[31:30] The “Next Thread” field is 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.
```

In the above example, the assembler will place a “10” in the “Next Thread” field of the current instruction word. If no soft-scheduled thread switch is specified next to the mnemonic, the value for the NT field will remain “00”, indicating that no soft-schedule thread switch is to take place 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 D[27:24]	Mnem	Description
"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 0x7FFFFFFF. 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

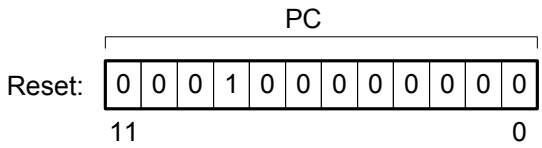
Chapter 5

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



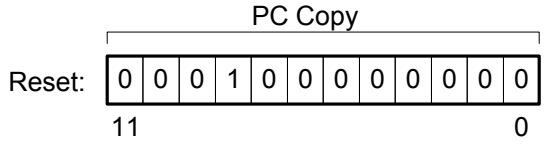
Logical Address: 0x006F

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



Logical Address: 0x006E

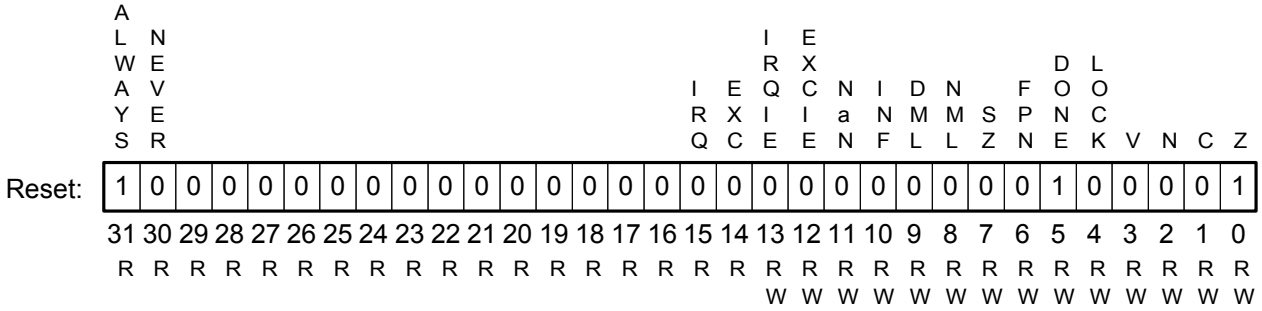
Each core has four, 12-bit PC Copy registers, one for each thread, residing in each thread’s private memory map at location 0x06E. It is both readable and writable. Anytime a thread’s PC register is written to by any instruction, a copy of that thread’s PC (+ 1) at the time the instruction was fetched 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 0x0020 for NMI interrupts, 0x0021 for floating-point exception EXC interrupts, and 0x0022 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.

5.1.3 STATUS Register

Figure 5—3. STATUS Register



Logical Address: 0x006D

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

Table 5—1. 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.
FPN	6	(Normal) floating-point negative flag—set if result is negative, reset otherwise.
SZ	7	(Normal) floating-point signed zero flag—set if result is zero, reset otherwise. Sign bit (MSB of result) can be 1 or 0 and still be normal.
NML	8	Floating-point normal flag—set if the result is normal, reset otherwise.
DML	9	Floating-point denormal flag—set if exponent of result is zero and mantissa of result is not zero, reset otherwise.

INF	10	Floating-point infinity flag—set if result is infinite, reset otherwise.
NaN	11	Floating-point not-a-number flag—set if result is not a number, reset otherwise.
EXCIE	12	Floating-point exception interrupt enable—1 enables EXC interrupts, 0 disables EXC interrupts.
IRQIE	13	General-purpose IRQ interrupt enable—1 enables IRQ interrupts, 0 disables IRQ interrupts.
EXC	14	(Read-only) floating-point exception—1 indicates result is either a NaN, INF, or DML, 0 otherwise.
IRQ	15	(Read-only) general-purpose IRQ input state—1 indicates an IRQ is being requested, 0 indicates no IRQ is being requested.
not used	16-29	(Read-only) these bits are presently unused and are read as 0.
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. All bits are readable, but only the lower fourteen bits are also writable. 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, #00100000b              ;set the DONE bit
```

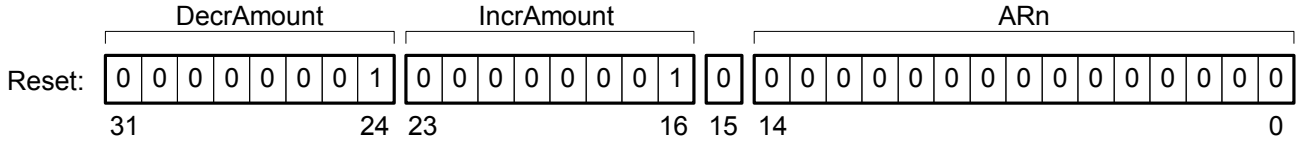
With the Cross-32 Meta Assembler, the instruction table can be easily enhanced to include aliases for the above operations to make code more readable and easier to code in assembly language. With the aliases, the assembler will generate the exact same machine code as above. Examples:

```
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 25 = 1)
BLE     dest        ;branch if less than or equal (STATUS bit 24 = 1)
BGZ     dest        ;branch if greater than 0 (STATUS bit 28 = 1)
BLZ     dest        ;branch if less than 0 (STATUS bit 26 = 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



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-execute cycle (q0), such that, if the very next instruction that is fetched following the instruction-fetch that contained the Dest ARn post-modifier operator contains expects to be able 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:

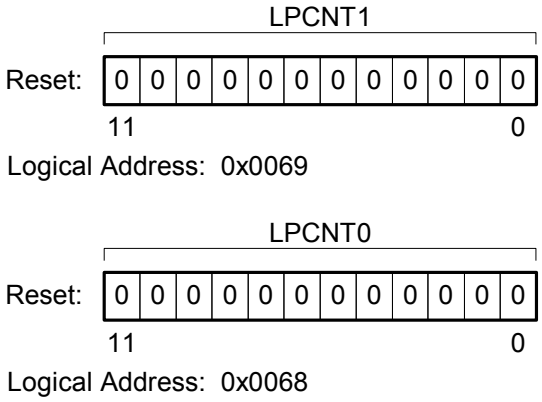
```
ADD    *AR1++, work1, *AR2++     ;the contents of *AR2 are added to the contents of work1
                                 ;and stored in the memory location pointed to by the contents
                                 ;of AR1.  AR1 and AR2 are then post-incremented to point to
                                 ;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 Hardware Loop-Counters LPCNT1 and LPCNT0

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



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.

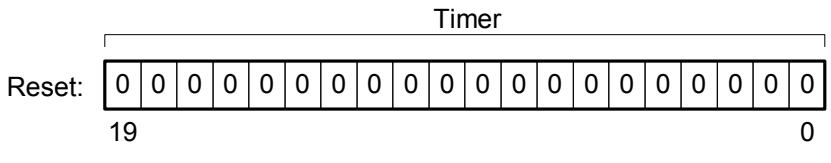
In tight loops where the segment (not including the twp separate decrement and BTBC instructions) is only one or two instructions, at least half the processing cycles is devoted to branching. In such scenarios, this can be expensive, especially when large data sets are being operated on.

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.

5.1.6 Timer Register

Figure 5—6. Timer Register



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.

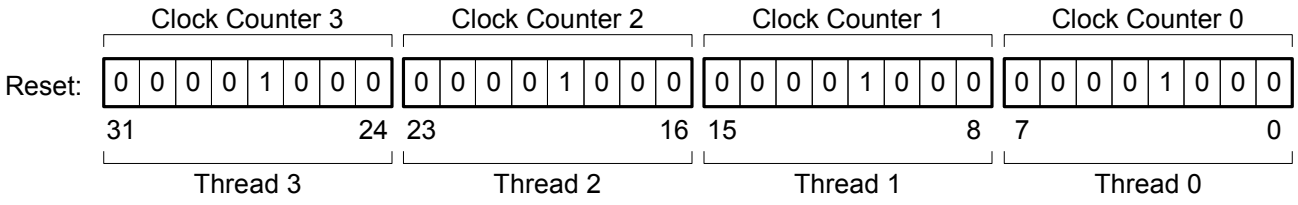
Consequently, since the zero-count output of the timer is hard-wired to the thread's NMI line, one of the first operations a thread must perform before clearing its DONE bit (signaling it is now busy), is load the timer with a cycle count value that exceeds the estimated number of clock cycles needed 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.

5.2 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.

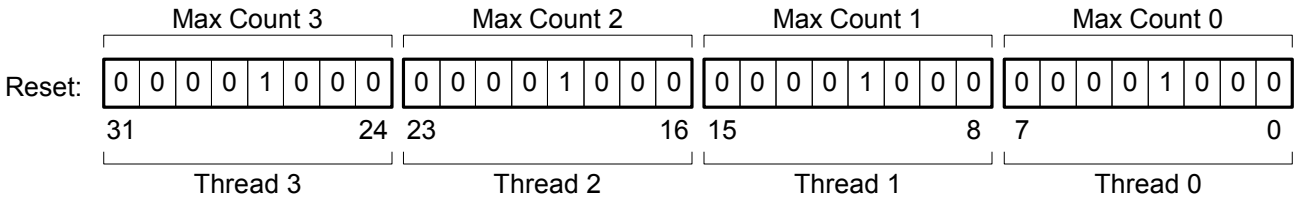
5.2.1 Fine-Grained Scheduler Counter and Compare Registers

Figure 5—7. Fine-Grained Scheduler Counter Register



Logical Address: 0x006C

Figure 5—8. 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—7* 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.

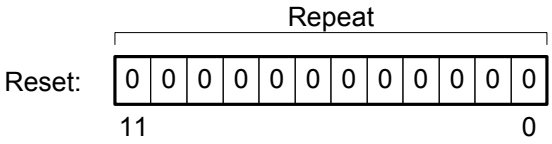
Writing a 0x00 to a given thread's hardware fine-grained scheduler counter register position will effectively remove that thread from the fine-grained scheduling queue, but still may be soft-scheduled by a different thread. If the value 0x00000000 is written to the fine-grained scheduler counter register, 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—8. 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.

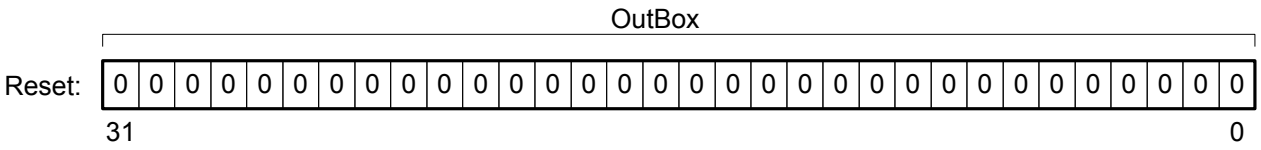
Example:

;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)
```

5.2.3 OutBox Register

Figure 5—9. OutBox Register



Logical Address: 0x006A

Each Shader has one, 32-bit , globally-mapped output register residing at location 0x006A. It is both readable and writable. The OutBox register is general-purpose and may be used by any thread of a given Shader, for, among other things, signaling to the outside world. It can come in handy for use in a test-bench. On reset, the contents of OutBox is cleared to 0x00000000.

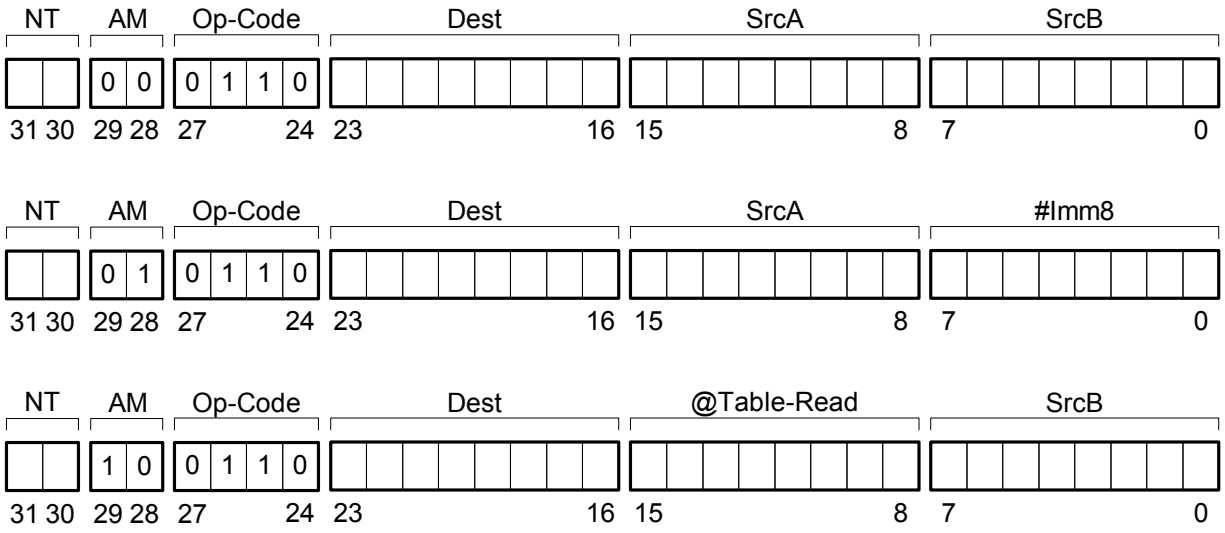
Presently, there are no module ports in the test-bench to accommodate each Shader's OutBox and, consequently, the register will dissolve away during synthesis unless you add a module port to accommodate them.

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—10. ADD Instruction



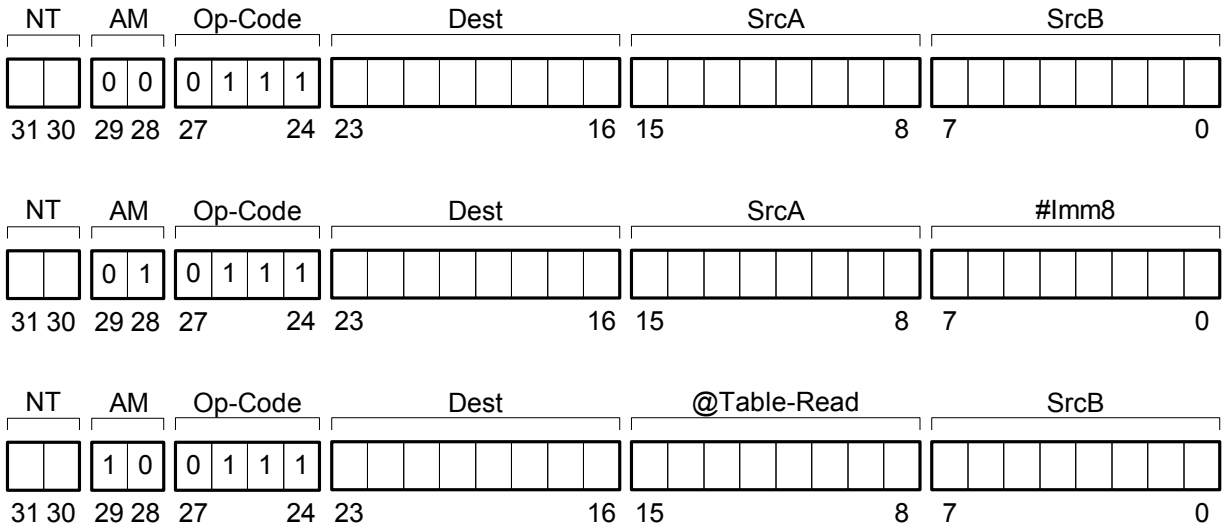
ADD adds (without carry flag) the contents SrcB to the contents of SrcA and then stores the results in Dest. Permitted addressing modes for SrcA include direct, indirect, or table-read from program memory. Permitted addressing modes for SrcB include direct, indirect, or 8-bit immediate. If SrcB is 8-bit 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—11. ADDC Instruction



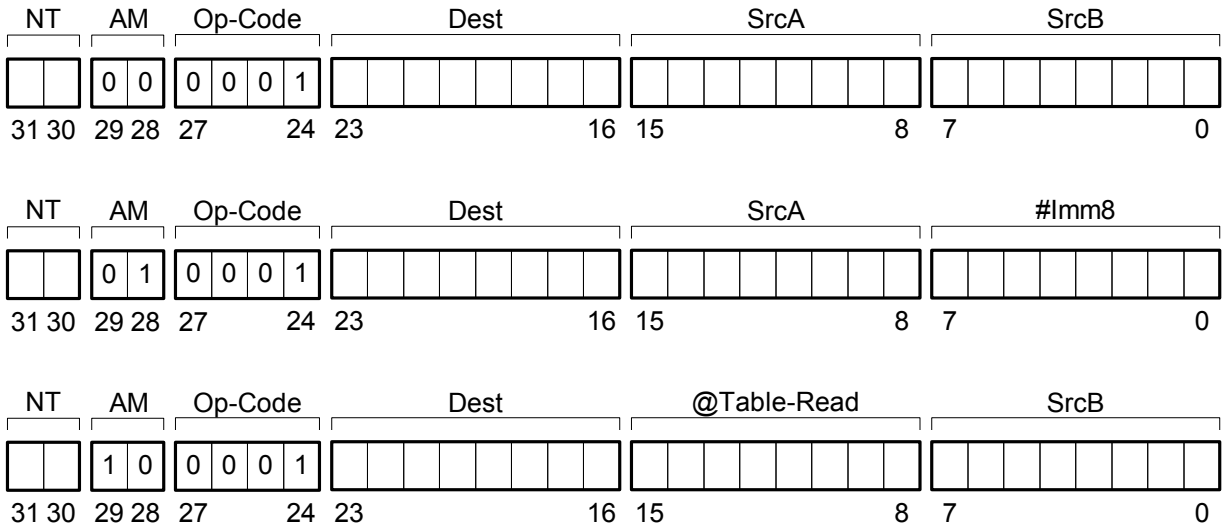
ADDC adds the contents SrcB and the carry flag to the contents of SrcA. Permitted addressing modes for SrcA include direct, indirect, or table-read from program memory. Permitted addressing modes for SrcB include direct, indirect, or 8-bit immediate. If SrcB is 8-bit immediate, it is zero-extended to 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—12. AND Instruction



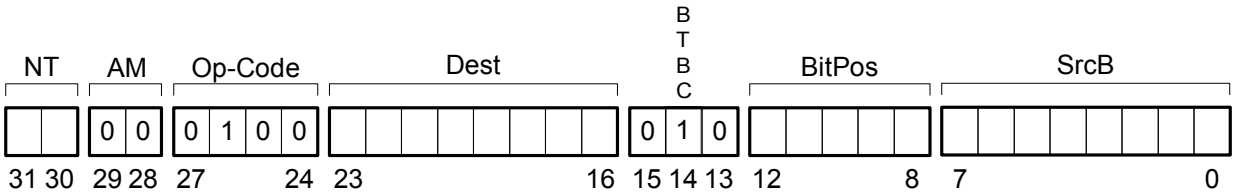
AND performs a bit-wise AND of the contents of SrcA and the contents of SrcB. Permitted addressing modes for SrcA include direct, indirect, or table-read from program memory. Permitted addressing modes for SrcB include direct, indirect, or 8-bit immediate. If SrcB is 8-bit immediate, it is zero-extended 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—13. BTBC Instruction



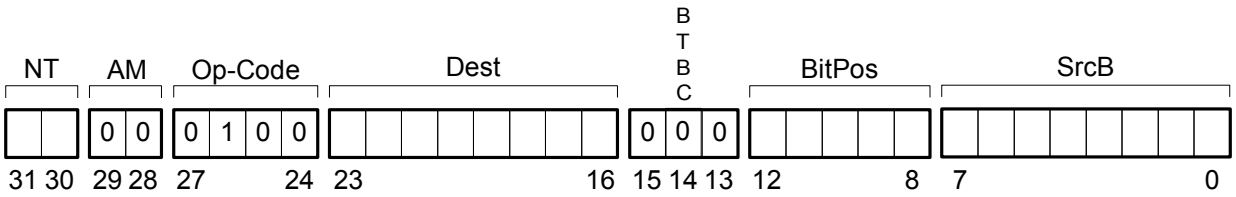
Bit-test-and-branch-if-clear (BTBC) tests for logic 0 the bit position specified in BitPos with the contents of SrcB. If the result of the test is 0 (true), then the PC is loaded with an offset relative to the program address of the BTBC fetch, such offset being in the range +127 to -128. Permitted addressing 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—14. BTBS Instruction



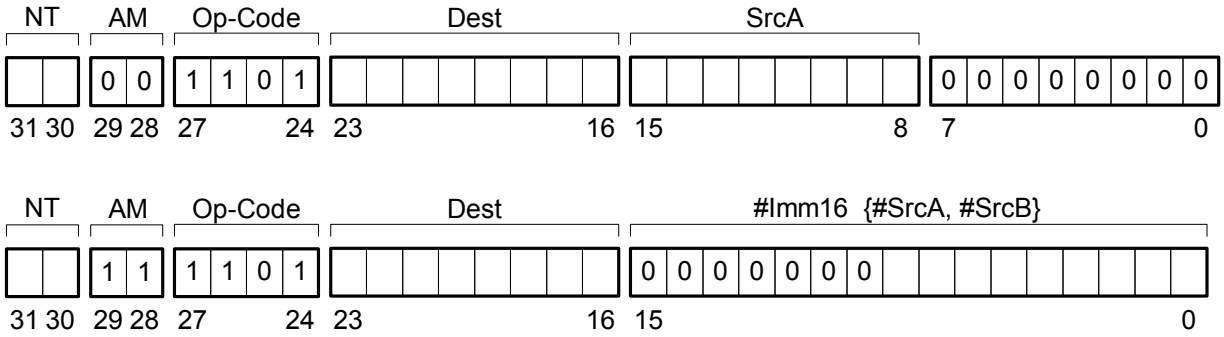
Bit-test-and-branch-if-set (BTBS) tests for logic 1 the bit position specified in BitPos with the contents of SrcB. If the result of the test is 1 (true), then the PC is loaded with an offset relative to the program address of the BTBS fetch, such offset being in the range +127 to -128. Permitted addressing 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—15. COS Instruction



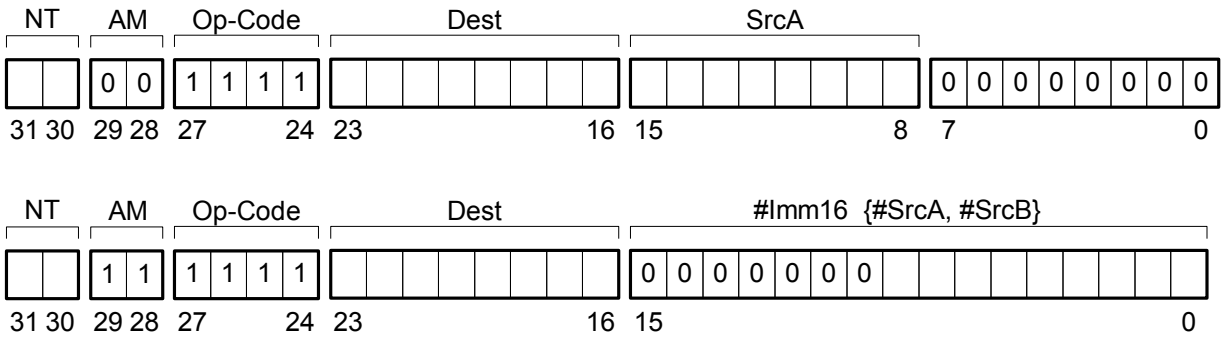
COS returns the 32-bit, floating-point cosine of the 9-bit, signed integer contained in SrcA or 16-bit immediate value and stores the result in Dest. The integer contained in SrcA or the 16-bit immediate value must be in the range of +/- 360 degrees. STATUS flags Z and N are affected as shown below.

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—16. COT Instruction



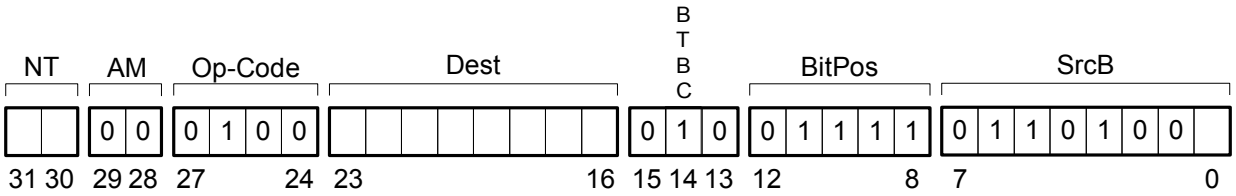
COT returns the 32-bit, floating-point cosine of the 9-bit, signed integer contained in SrcA or 16-bit immediate value and stores the result in Dest. The integer contained in SrcA or the 16-bit immediate value must be in the range of +/- 360 degrees. STATUS flags Z and N are affected as shown below.

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—17. DBNZ Instruction



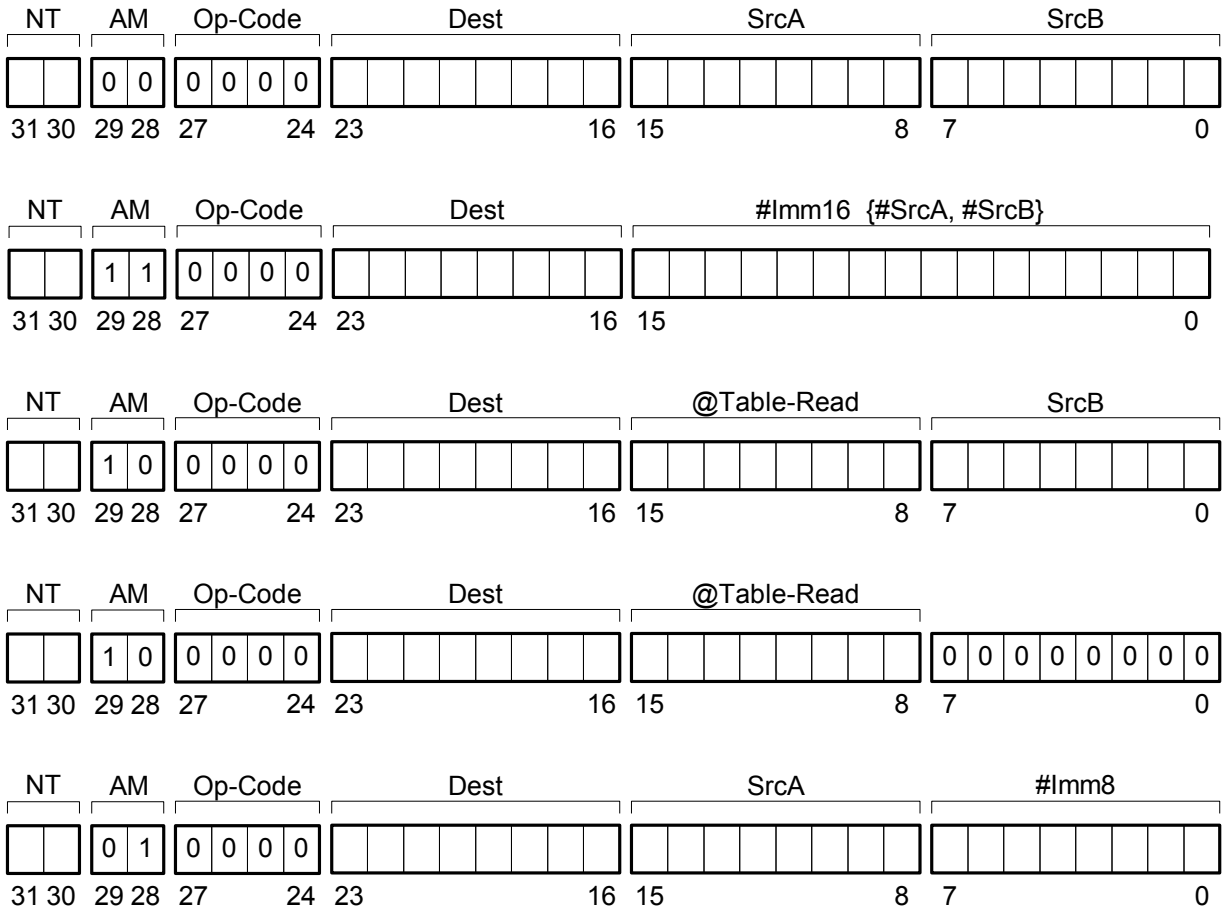
Decrement-and-branch-if-result-not-zero (DBNZ) first decrements the hardware loop-counter (LPCNTn) specified in SrcB then tests bit position 15 of the result for logic 0. If the result of the test is 0 (true), then the PC is loaded with an offset relative to the program address of the DBNZ fetch, such 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—18. MOV Instruction



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

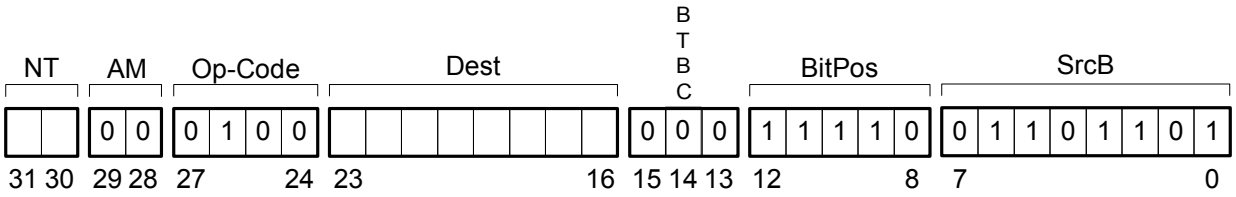
5.3.10 MOV (Special Feature)

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—19. NOP Instruction

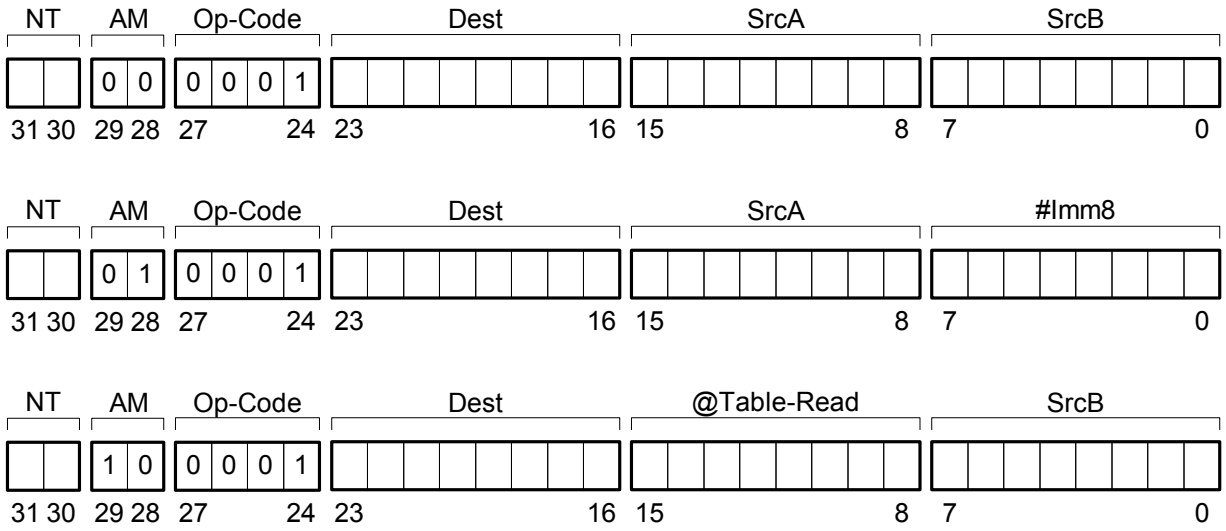


NOP is actually an alias of the bit-test-and-branch-if-set (BTBS) instruction, except the cross-assembler automatically fills in the SrcB field with the address of the STATUS register and specifies bit-30 (labeled “NEVER” and hard-wired to “0”) as the bit to be tested for logic 1. Since the test is never 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—20. OR Instruction



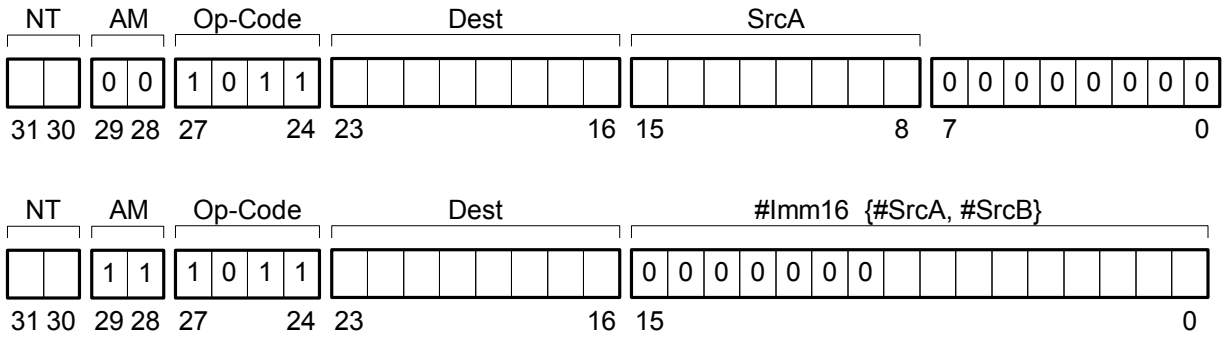
OR performs a bit-wise OR of the contents of SrcA and the contents of SrcB and then stores the results in Dest. Permitted addressing modes for SrcA include direct, indirect, or table-read from program memory. Permitted addressing modes for SrcB include direct, indirect, or 8-bit immediate. If SrcB is 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—21. RCP Instruction



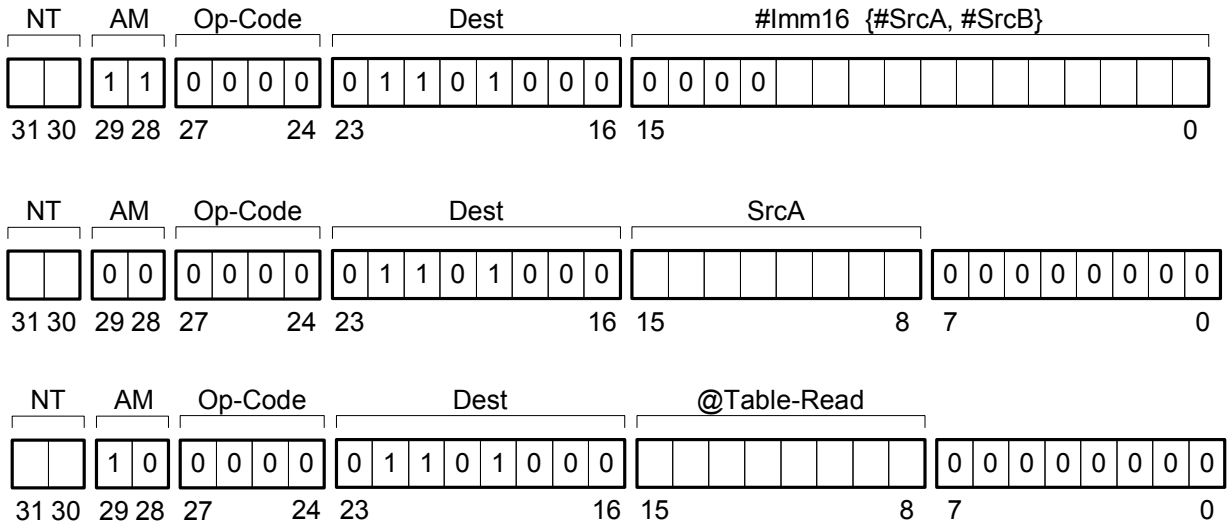
RCP returns the 1/n (reciprocal) of the 8-bit SrcA (signed integer) and stores the 32-bit floating-point result in Dest. SrcA must be a signed integer in the range +127 to -128. Data bits D[31:8] of SrcA are ignored. Input of SrcA == 0 returns 0x7FFFFFFF. Results of the RCP can be used in combination 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

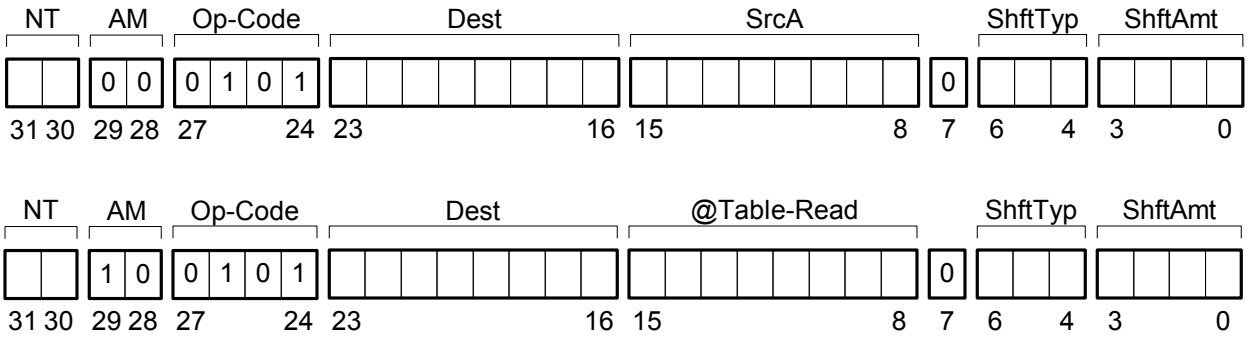
Figure 5—22. RPT Instruction



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—23. SHFT Instruction



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:

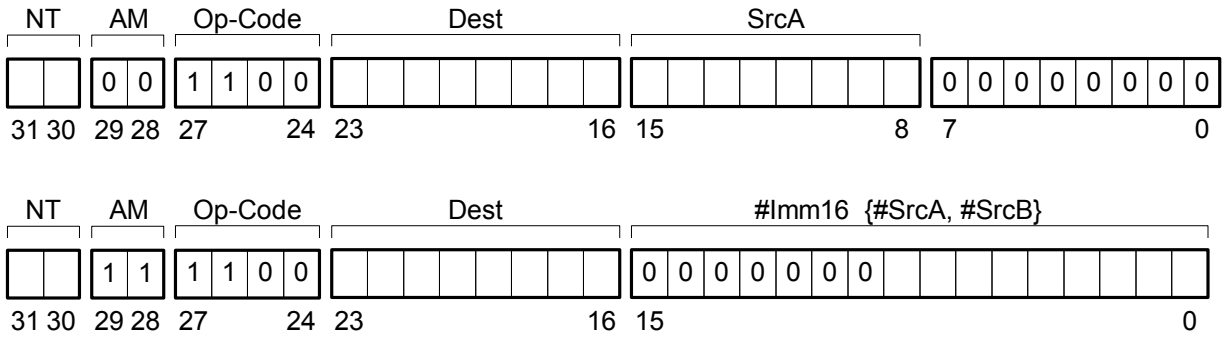
D[6:4]	**ShftType**	**Description**
"000"	LEFT	Carry flag is unaffected. LSBs filled with "0". Same as ASL.
"001"	LSL	Logical shift left through Carry. "0" shifted in through LSB.
"010"	ASL	Arithmetic shift left. LSBs filled with "0". Same as LEFT above.
"011"	ROL	True barrel shift left. Bits shifted 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" 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, copying Carry into MSB each time
```

5.3.16 SIN

Figure 5—24. SIN Instruction



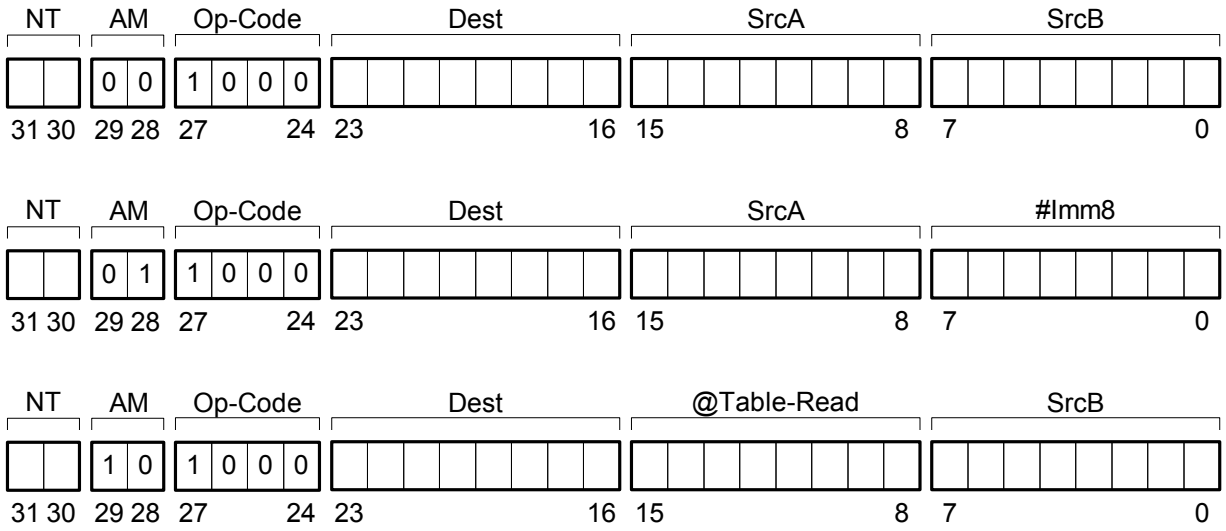
SIN returns the 32-bit, floating-point sine of the 9-bit, signed integer contained in SrcA, or 16-bit immediate value, and stores the result in Dest. The integer contained in SrcA or the 16-bit immediate value must be in the range of +/- 360 degrees. STATUS flags Z and N are affected as shown below.

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—25. SUB Instruction



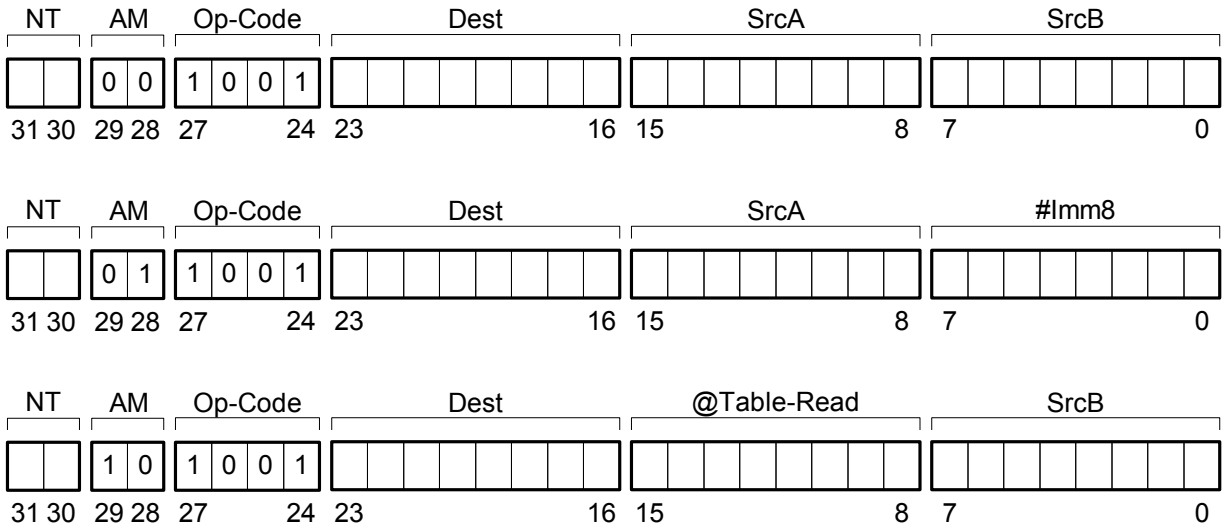
SUB subtracst (without barrow from carry flag) the contents SrcB from the contents of SrcA and then stores the results in Dest. Permitted addressing modes for SrcA include direct, indirect, or table-read from program memory. Permitted addressing modes for SrcB include direct, indirect, or 8-bit immediate. 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—26. SUBB Instruction



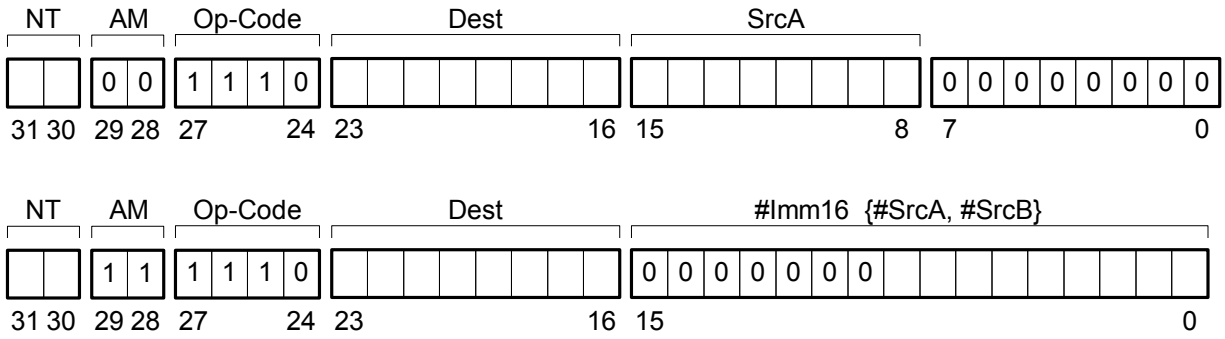
SUBB subtracts (with barrow from carry flag) the contents SrcB from the contents of SrcA and then stores the results in Dest. Permitted addressing modes for SrcA include direct, indirect, or table-read from program memory. Permitted addressing modes for SrcB include direct, indirect, or 8-bit immediate. 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—27. TAN Instruction



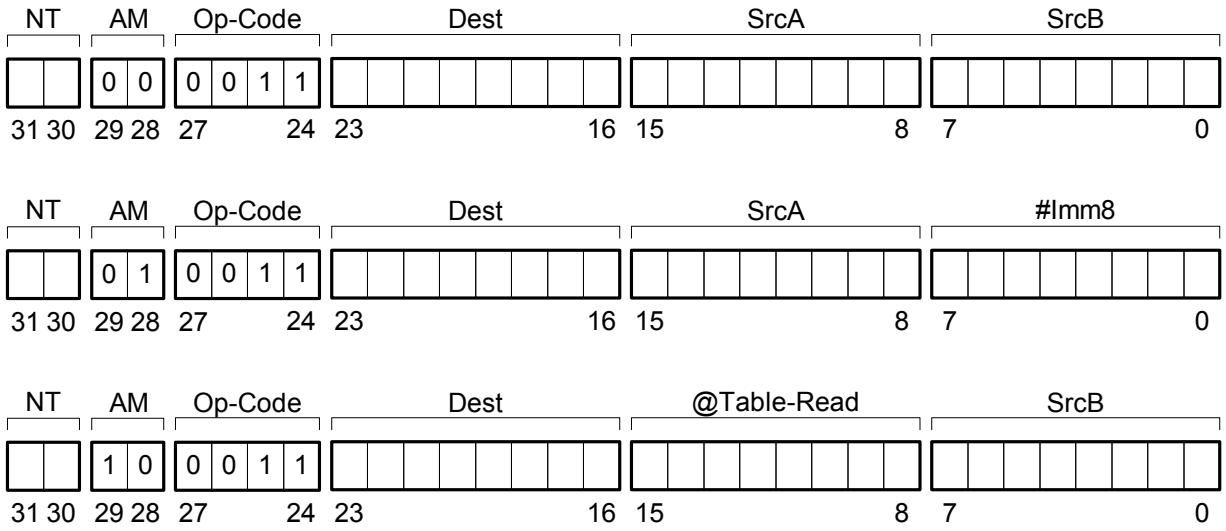
TAN returns the 32-bit, floating-point tangent of the 9-bit, signed integer contained in SrcA, or 16-bit immediate value, and stores the result in Dest. The integer contained in SrcA or the 16-bit immediate value must be in the range of +/- 360 degrees. STATUS flags Z and N are affected as shown below.

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—28. XOR Instruction



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:

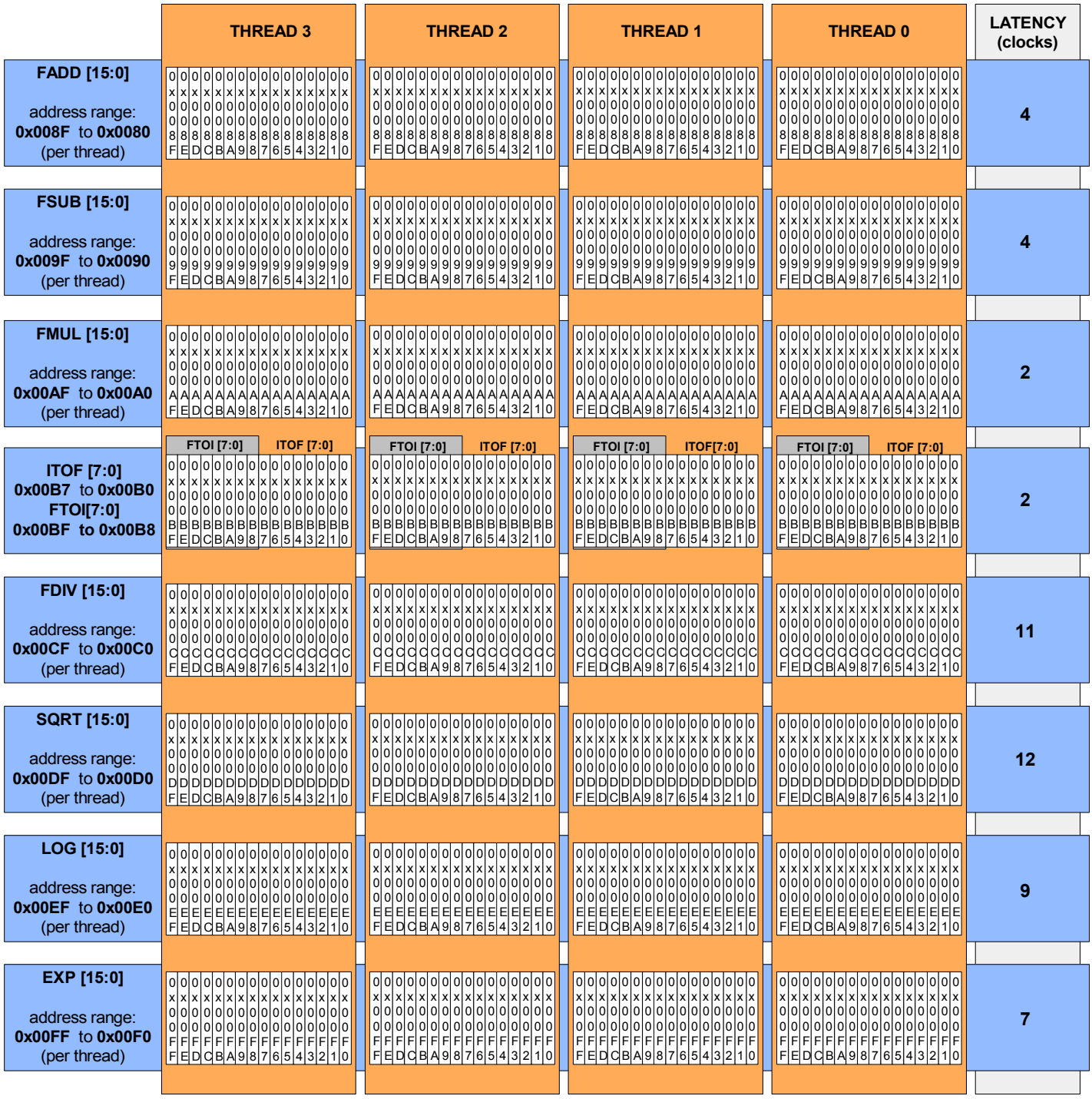
N—set if MSB of result is 1 (negative), reset otherwise
C—not affected
V—not affected
Z—set if result is zero, reset otherwise

Chapter 6

Floating-Point Operators

6.1 Memory-Mapped Floating-Point Operators

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



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.

6.1.1 Floating-Point Exception Codes

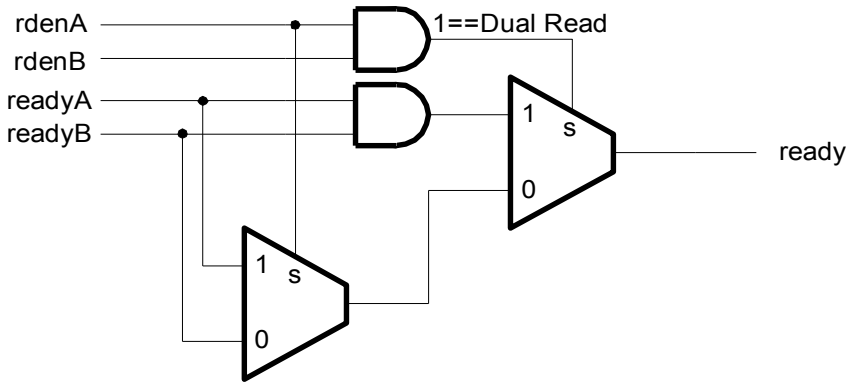
The two most-significant bits of each result buffer contain an encoded floating-point exception code that is not directly visible at the time of a given read. Instead, anytime a result is successfully read out, the exception code is decoded by the hardware and registered in that thread's STATUS register during the execute cycle of the instruction doing the read. If the exception code indicates an floating-point exception such as NaN, INF or DML, and the EXC interrupt enable bit (EXCIE) is set, then an EXC interrupt is immediately generated.

6.1.2 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—2* 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—2. Floating-Point Result-Ready Logic

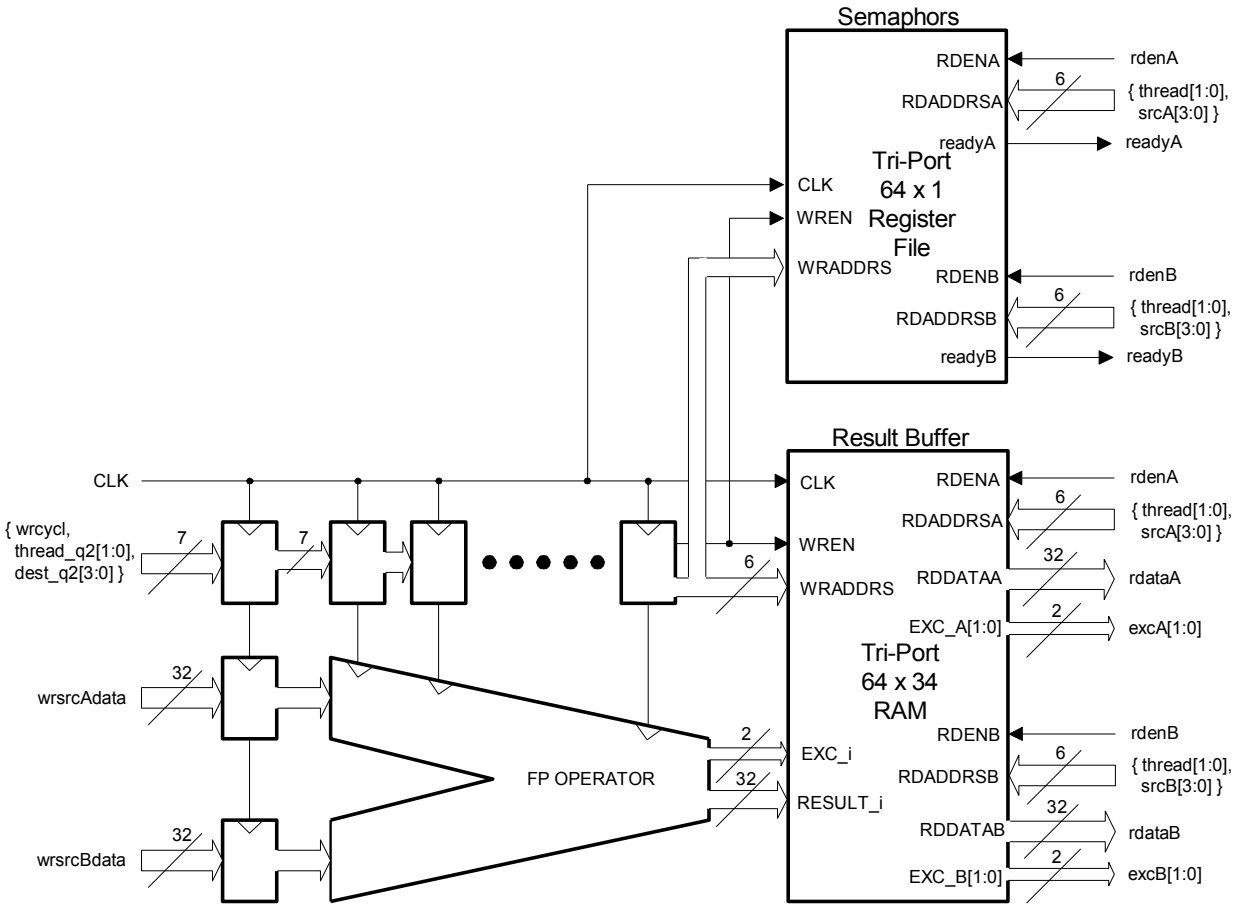


6.2 Floating-Operator Pipeline

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—3* below shows a the arrangement of one of the dual-operand, floating-point pipeline, corresponding semaphore and result buffer memory.

Figure 6—3. Floating-Point Operator Pipeline and Result Buffers



6.3 Registering Floating-Point Exception Flags

As mentioned previously, each floating-point result buffer location is actually 34-bits wide, wherein the two MSBs are used to store a 2-bit corresponding floating-point exception code, which are decoded and are automatically registered in a given thread's STATUS register during the accessing instruction's execution cycle.

Presently, only the SrcA side exception flags of a dual-operand read from result buffer memory are registered in this way. Stated another way, if you are using a dual-operand MOV instruction to read two result buffer locations simultaneously, only the SrcA-side result

exception flags are registered in that thread's STATUS register. If your application requires that all floating-point results be sanitized for floating-point exceptions, there are two options available to you.

The first option is to use two single-operand read instructions (such as a single-operand MOV) to retrieve your results (so that the flags will always be registered in that thread's STATUS register) and assemble the single results in RAM before use as dual-operands. The second option is to modify/expand the core logic for the STATUS register of each thread to include a second set of floating-point exception flags, such that both sets of flags are automatically registered in cases of dual-operand MOV accesses from the result buffer memory space.

If you would like a version of the Shader core that incorporates a dual set of floating-point exception flags in the STATUS register, please contact me at: SYMPL.GPU@gmail.com

6.4 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.

6.5 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.

6.6 Floating-Point Operator Descriptions

As mentioned above, the SYMPL FP32X-AXI4 RTL includes wrappers for FADD, FSUB, FMUL, FDIV, SQRT, 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.

6.7 How to Use the Floating-Point Operators

With exception to the ITOF and FTOI 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 three 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 threads 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 interrupt (EXC), and general-purpose interrupt (IRQ) vector locations are at 0x0101, 0x0102, and 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 interrupt (EXC), 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 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 0x0020, 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, 0x0020 instruction, which it treats as an NMI-RETI instruction, which clears the NMI-in-service state.

7.2.2 Floating-Point Exception Interrupt (EXC)

Each thread also has its own maskable, EXC interrupt, which the second-highest priority of the three interrupt sources. At least three conditions must occur before an EXC goes active: first, NMI must not be active or in-service; second, the EXC interrupt enable mask (bit 12 of that thread's STATUS register) must be set to 1; and third, the NaN (not-a-number), INF (infinity), or DML (denormal) floating-point exception flag in that thread's STATUS register must be active.

If the three conditions listed above exist, an EXC will cause the current thread to suspend execution of the current routine and automatically load its PC with 0x0102, where it fetches its respective EXC vector, which is MOV PC, #EXC_SERVC.

Since EXC is lower priority than NMI, NMI can interrupt an EXC interrupt service routine should a time-out occur. If this should occur, the EXC service routine will resume once the NMI has completed its service routine. The same is true regarding EXC and IRQ, since EXC is higher priority than IRQ.

After entering the thread's EXC service routine, among the first operations that must be performed is to save the PC COPY register to that threads private memory location 0x0021, 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, 0x0021 instruction, which it treats as an EXC-RETI instruction, which clears the EXC-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 three available interrupt sources and, therefore, its service routine may be interrupted by an NMI or EXC interrupt. An IRQ interrupt cannot interrupt an active NMI or EXC interrupt while either of them are active or in-service.

Since IRQ is maskable, its respective interrupt enable (bit 13 of that thread's STATUS register) must be set to 1 before a vectored IRQ service routine can be initiated. However, both the EXC interrupt input and the IRQ input can be polled while their respective interrupt enables are cleared by reading bits 15 and 14 respectively 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 0x0022, 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, 0x0022 instruction, which it treats as an IRQ-RETI instruction, which clears the IRQ-in-service state.

Coarse-Grained Scheduler (CGS)

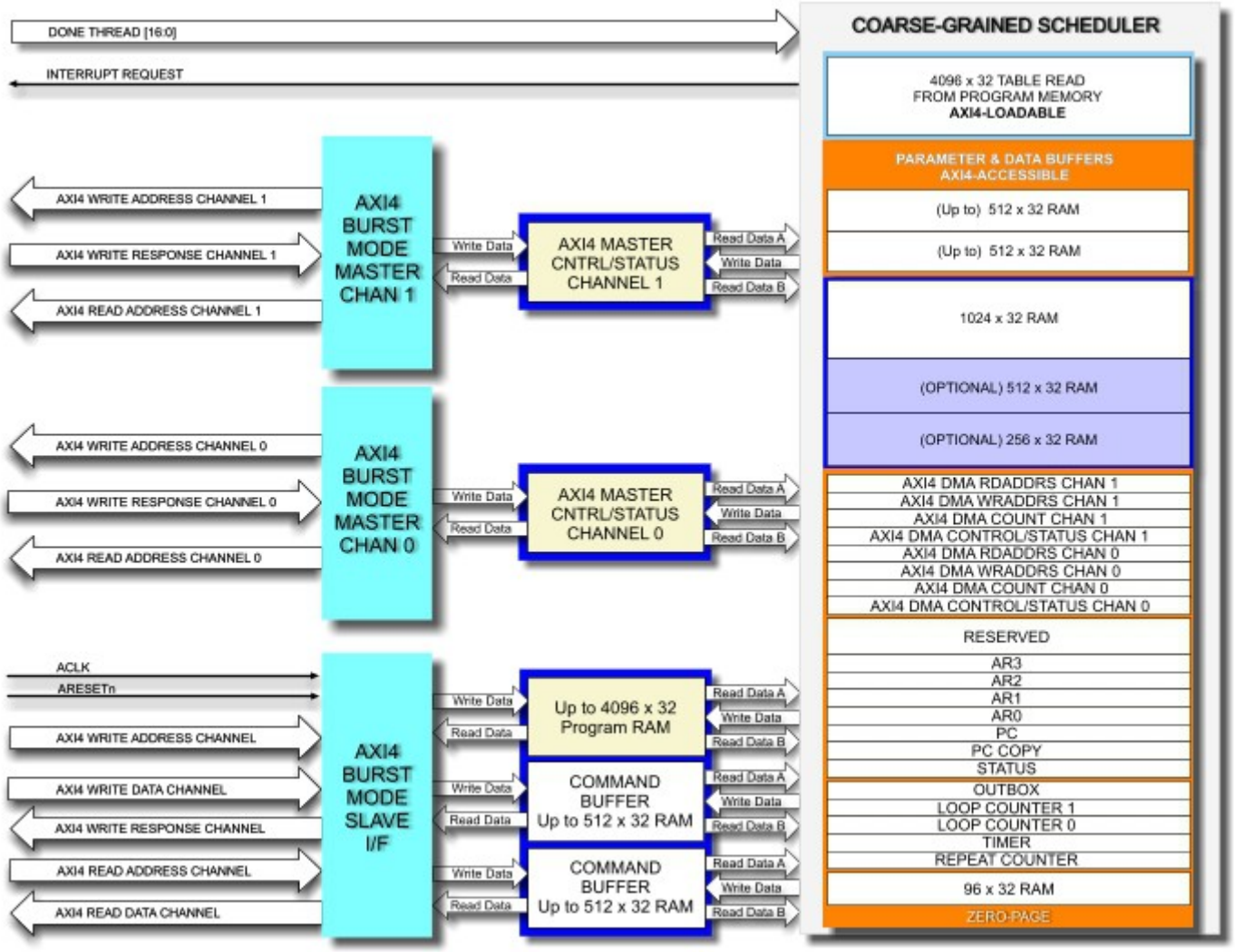
Chapter 8

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.

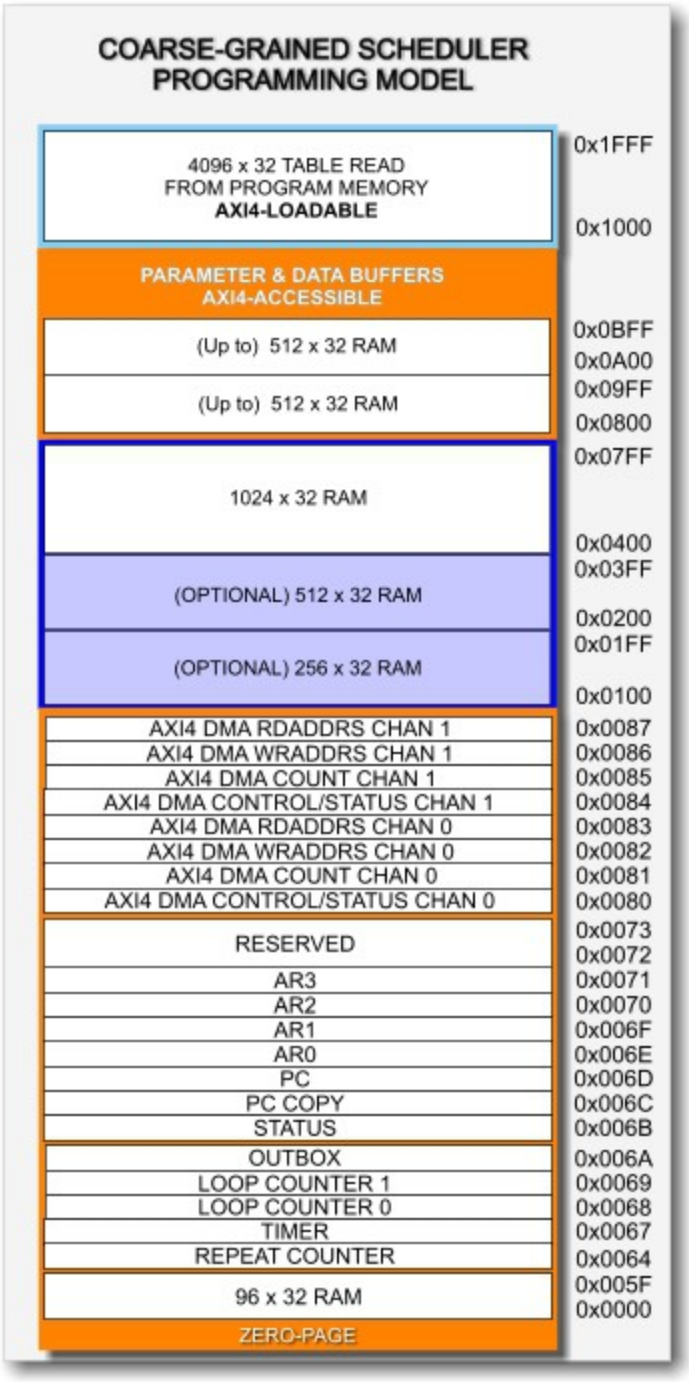
Figure 8—1. Coarse-Grained Scheduler Block Diagram



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

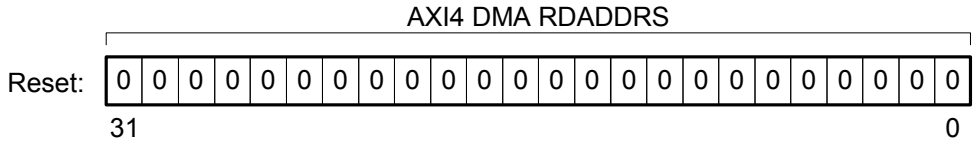
Registers unique to the SYML CGS include registers associated with the two AXI4 DMA master channels and a single, 32-bit register to enable monitoring interrupt requests from up to sixteen Shader threads. These registers are described below.

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



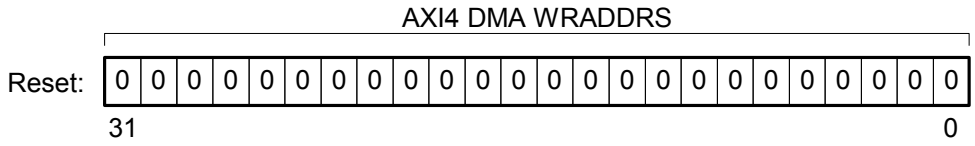
Logical Address (Channel 1): 0x0087
Logical Address (Channel 0): 0x0083

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)

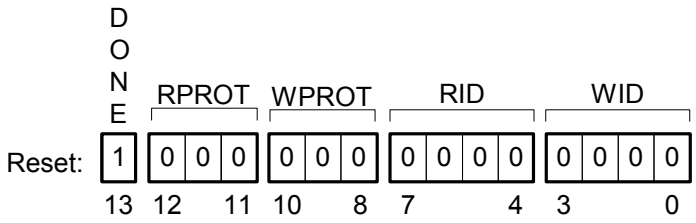
The DCSRs (one for each DMA channel) are used to configure each AXI4 DMA channel's read and write IDs as well as their read and write protection modes. Bit 13 of the DCSR is used to monitor the state of the transfer once initiated. On reset and on completion of a transfer, the DONE bit goes to logic 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



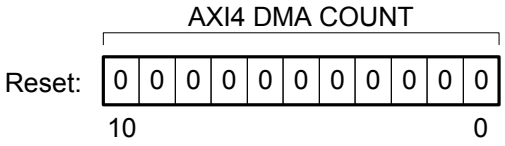
Logical Address (Channel 1): 0x0084

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.

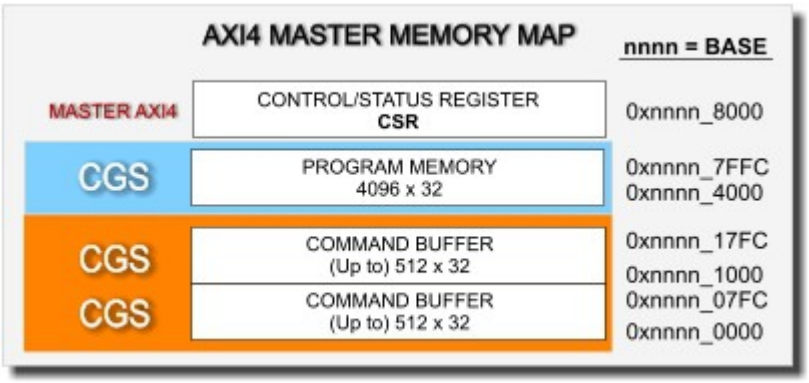
For a more detailed explanation on how just one example protocol might be implemented, refer to the previous section covering this topic for the Shader core AXI4 slave DMA interface.

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

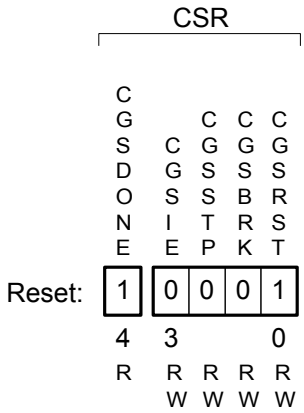
The diagram in *Figure 8—8* below shows the memory-mapped AXI4 Control/Status Register (CSR). The CPU must use this register to reset and/or launch the CGS. On system reset, the CGSRST line is set to one and remains in that state until the CPU writes a 0 to that bit position.

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



nnnn = Base Address
Logical Address: 0xnnnn_8000