Convey Spat Users Guide

October 2009
Version 1.1
901-000004-000

1302 East Collins
Richardson, TX 75081

Revisions

Version	Description
1.0	April 2009. Original printing
1.1	October 2009. Added Double Precision

Table of Contents

1 Introduction

Please note that this document contains clickable hyperlinks. If you are browsing this document on Convey's web site, or downloaded ALL of Convey's documents to the same directory, those clickable hyperlinks will work. If you only download a single document, clicking on the hyperlinks will fail.

This Convey Spat Users Guide describes how to use the Simulator Performance Analysis Tool (Spat) to:

- Generate execution traces from the Convey Simulator
- Use Spat to generate plot data of the execution
- Use the `spat.viewer` program to view a plot
- Use Spat to interactively trace the execution stream
- Use the plot and the interactive trace to enhance performance

The latest revision of this document is always available at **www.conveycomputer.com**.

2 Intended Audience

This guide is recommended reading for users who are developing or porting software for a Convey hybrid-core server and need to tune the performance of that software.

2.1 Other Suggested Documents

Convey Reference Manual – describes the Convey coprocessor architecture and instruction sets

Convey Mathematical Libraries Guide – describes the higher level mathematical functions available in the Convey Math Libraries which are optimized to take advantage of the Convey coprocessor when using the Convey provided vector personalities

Convey System Administration Guide – describes Convey SW installation/packaging, HW installation, customer support procedures

Convey PDK Reference Manual – for users of the Convey Personality Development Kit

Convey Programmers Guide – describes Convey compilers

All the above documents are available at www.conveycomputer.com, and in `/opt/convey/doc` on systems where the corresponding software has been installed.

2.2 Typographical conventions used in this guide

`Dark orange fixed width font` is used to indicate shell commands a user or system administrator might use.

`Fixed width font` is used to identify program names, command flags, filenames, file contents …

Italicized text is used to identify key concepts.

Bold text is used to draw attention to paragraph and section headings, and when also italicized, to mark the first time a key concept or idea is presented.

`Dark red text` is used for Convey-specific language extensions, such as Convey specific compiler directives or flags.

Light gray text is used to mark functionality that is not yet available in the alpha or beta test versions of various Convey products.

Underlined blue text is used for clickable hyperlinks. Convey's online documentation is typically provided in Adobe's® Portable Document Format (PDF). Most web browsers with the Adobe Reader® plugin installed can display Convey manuals and follow the embedded hyperlinks.

3 Overview

3.1 Definitions

Convey Server Terminology

hybrid-core server	The Convey hybrid-core server combines a host x86-64 processor and a Convey coprocessor. The host processor and coprocessor share a cache coherent global memory address space. A program can execute code on both processors concurrently.
coprocessor	The Convey *coprocessor* is an attached processor that can be loaded with different personalities, including both Convey-defined and user-defined personalities.
personality	A *personality* is an instruction set that is implemented on the Convey coprocessor. Personalities may be swapped during execution of a program. All personalities, including user-defined personalities, share a common core scalar instruction set. Convey currently provides single precision vector personality and plans to release other personalities including several double precision vector personalities.
signature	A *signature* identifies one specific version of a personality, and a corresponding coprocessor firmware image. Signatures contain a personality name/number, major and minor version numbers, and a coprocessor hardware model number. Signatures may be fully resolved or partially resolved. A fully resolved signature is a signature that has non-zero version numbers, and is installed and enabled. A partially resolved signature is converted into a fully resolved signature as described later in this document. The following are some sample signatures:

Single	A partial signature selecting the single precision vector instruction set
Sp	Another nickname for **single**
double.2.1	A partial signature selecting the double precision vector instruction set, version 2.1
2.1.1.3	A complete (fully resolved) signature, specifying personality number 2, version 1.1, and hardware model number 3.

coprocessor image	A coprocessor image is the FPGA instruction image that is downloaded into the Convey coprocessor whenever coprocessor code for the corresponding signature is about to be executed. The image implements the desired instruction set. Each installed signature has its own coprocessor image.
signature resolution	When a user compiles a routine and specifies a partial signature, that signature will undergo two levels of *signature resolution* to select an actual installed signature, controlled by various system defaults and environment variables. Compile time signature resolution resolves a partial signature to select a particular signature for the compiler and assembler to use, but permits a different but compatible signature to be used when executing the program. Runtime signature resolution allows programs to be compiled using a particular signature, and then later to be executed using a compatible signature, such as a newer bug fix version, different coprocessor hardware model number, or a system administrator 'approved' version. Runtime signature resolution also allows a single runtime signature to be used instead of several different but compatible compile time signatures, reducing the time spent loading coprocessor images.

Convey Coprocessor/Compiler Terminology

Attached coprocessor	The Convey coprocessor must be attached to a process before it can be used. Only one process or an MPI process group can attach the coprocessor at a time. All the threads in such a process are free to use the coprocessor also. Convey's Linux operating system typically attaches the coprocessor (if it is available) when an application that expects to use the coprocessor starts execution. Some applications that utilize the Convey Mathematical Libraries don't attach to the coprocessor until the first library routine that uses the coprocessor is called. The Convey coprocessor is part of every Convey Hybrid-Core Server.
Coprocessor code	A sequence of Convey coprocessor instructions. Coprocessor code can only be executed on the Convey coprocessor or the Convey coprocessor simulator. Coprocessor code may be dispatched from host code, or called directly from other coprocessor code.

Coprocessor region	A block of source code within a routine that is (typically) compiled for **both** the host processor (an x86-64 processor) and the Convey coprocessor. At runtime, when a coprocessor region is executed by the host processor, the host code will first check to see if the Convey coprocessor is attached to this process and (optionally) determine if executing that region on the coprocessor is expected to be profitable (faster). If the coprocessor is attached and executing the region on the coprocessor is expected to be profitable, the coprocessor version of that region is dispatched to the coprocessor. Otherwise, the equivalent host code for that region is executed. Coprocessor regions may be created explicitly with Convey provided directives/pragmas and compiler flags, or implicitly with the **`-mcny_auto_vector`** flag.
Coprocessor routine	An entire routine can be compiled for the coprocessor with the **`-mcny_dual_target`**, **`-mcny_dual_target_nowrap`**, or **`-mcny_pure`** flags. The assembler can also be used to create a routine that is entirely coprocessor code.
Host code	X86-64 executable code that executes on the host processor.
Host processor	The x86-64 processor. Refers to both the x86-64 processor on a Convey hybrid-core server as well as the x86-64 processor on any generic x86-64 Linux system.
Coprocessor dispatch	The host processor can *dispatch* the coprocessor version of the code region, that is, initiate execution of coprocessor instructions. Coprocessor code will only be dispatched when the coprocessor is currently attached to the current process or process group. If the coprocessor is attached to the current process (or process group) but is busy, the dispatch is queued, and will start as soon as the coprocessor is free.
Simulator	The Convey Simulator is a functional simulator that emulates executing instructions on the Convey coprocessor. It directly supports the Convey provided personalities (single precision vector, double precision vector …). It also provides hooks for simulating user defined instructions, as defined by a custom personality, using user provided emulation code. The Convey Simulator can be used to generate execution traces of code being developed. If the environment variable **`CNY_SIM_THREAD`** is set appropriately, the Convey simulator is used instead of the coprocessor.
Spat	The Convey program Spat is a timing simulator that emulates the instruction timing of programs executing on the Convey coprocessor. It takes instruction traces from the Convey Simulator and generates plot data and visual displays showing instruction flow thru the coprocessor.

4 Spat Operating Modes, Options, Output and Commands

4.1 Spat Operating Modes

The Spat program operates in two modes, interactive and plot. In the interactive mode spat reads the input file, accepts user input commands and visually displays the internal state on the coprocessor. In the plot mode spat reads the input file (generated by the Convey hardware simulator) and produces an output file that contains plot data of the various execution units within a coprocessor. This output file is used for processing by the spat.viewer program.

4.2 Spat Options

The following user controllable options are used by spat. More than one option may be given on the command line.

-H	Print version/date/time of the spat executable and all valid options.
-F <file>	Replace the default input file (spat.dat) with <file>.
-SUMM	When the spat execution is complete, print the number of s, a and ae instructions cracked.
-PLOT [<plot_file_name>]	Generate the plot data file. If <plot_file_name> is entered, replace the default output file (spat.plot) with <plot_file_name>. No -Wxxx options should be used with this option.
-MEMT	When the spat execution is complete, print a summary of the memory transactions generated for each dispatch.
-E <file>	The <file> should be the executable used to generate the spat.dat file. This file is required to resolve any tag names in the breakpoint command.
-STAT	At the completion of each dispatch print a summary of clocks, flops, GF/s and GB/s for the dispatch.
-WAEx	Start interactive mode and display a window showing the internal state of the scalar unit and ae unit x (0<= x < 4). The environment variable **SPAT_AEx_GEOMETRY** controls the geometry of the display window being created (example for ae0 **setenv SPAT_AE0_GEOMETRY 90x31+0+0**).
-WAEA	Start interactive mode and display windows showing the internal state of all ae units.

-WAMx	Start interactive mode and display a window showing the internal state of the memory system for ae unit x (0<= x < 4). The environment variable **SPAT_AMx_GEOMETRY** controls the geometry of the display window being created (example for ae1 **setenv SPAT_AM1_GEOMETRY 90x28+200+200**).
-WAMA	Start interactive mode and display windows showing the internal state of the memory system for all ae units.
-WMCx	Start interactive mode and display a window showing the internal state of memory controller x (0<= x < 8). The environment variable **SPAT_MCx_GEOMETRY** controls the geometry of the display window being created (example for mc3 **setenv SPAT_MC3_GEOMETRY 90x60+300+300**).

4.3 Spat Output

The example below shows the output for any spat execution. The statistics are for the entire execution.

- Clocks The number of coprocessor clocks used. This does not include any coprocessor idle time between dispatches.
- Records The number of records processed by spat. A record can be a bundle of two instructions, a dispatch record or a memory access record.
- Cache Hits The number of instruction cache hits.
- Cache Misses The number of instruction cache misses.
- Branch Prediction Hits The number of correct branch predictions.
- Branch Prediction Misses The number of incorrect branch predictions.

```
$  spat
clocks 53727    records 3518   cache hits 1740   cache misses 6
    branch prediction   hits 20      misses 13
```

4.3.1 Option –SUMM Output

The example below shows a typical spat output for the **–SUMM** option. The counts show the number of instructions cracked and simulated of each type.

```
$  spat -SUMM
Cracked  a - 1748    s - 967    ae – 803
```

4.3.2 Option -MEMT Output

The example below shows a typical spat output for the **–MEMT** option. The columns show the types of loads and stores for each dispatch.

- Dsp Dispatch number
- ld4 Number of load instructions that load 4 bytes in one operation
- ld8 Number of load instructions that load 8 bytes in one operation
- st4 Number of store instructions that store 4 bytes in one operation
- st8 Number of store instructions that store 8 bytes in one operation
- glrmw Number of load instructions generated to complete a 4 byte store (read 8 bytes, modify 4 bytes, write 8 bytes)
- cmdst32 Number of instances that two 4 byte store instructions are combined to make an 8 byte store operation
- uncst32 Number of instances that two 4 byte store instructions cannot be combined to make an 8 byte store operation

```
$  spat -MEMT
```

Dsp	ld4	ld8	st4	st8	glrmw	cmbst32	uncst32
1	8	65784	16	6	8	4	4

4.3.3 Option –STAT Output

The example below shows a typical spat output for the **–STAT** option.

- Dsp Dispatch number
- clocks Number of coprocessor clocks this dispatch took to execute
- flops Number of flops executed in this dispatch
- Gf/s GigaFlops per second (calculated)
- GB/s GigaBytes transferred per second (calculated)

```
$  spat -STAT
```

Dsp	clocks	flops	Gf/s	GB/s
1	5386	2048	0.13	32.54

4.4 Spat Commands

If **spat** is started in the interactive mode (any –Wxxx option) **spat** will initialize and display the windows specified by the options and present a prompt to the user (**spat:**). The commands that may be entered at the prompt are described below. Commands may be executed by typing just the first letter of the command.

4.4.1 h[elp]

Show all the valid spat commands.

4.4.2 q[uit]

Print any statistics and exit the spat program.

4.4.3 s[tep] [stepCnt]

Step the spat analysis **stepCnt** clocks and update all windows when the command completes. If **stepCnt** is not specified the default is 1. If update is enabled (see the **update** command below), the windows are updated at the specified frequency. If all dispatches and instructions have retired before **stepCnt** is reached, spat will terminate the step command. The step command is a sticky command. If the user hits enter (null command) after a **step 10** command, the **step 10** command will be repeated.

4.4.4 r[un]

Run until finished. If update is enabled (see the **update** command below), the windows are updated at the specified frequency. When all dispatches and instructions have retired, print any statistics and exit.

4.4.5 c[lock] clockValue

Run until the clock reaches **clockValue**. The **clockValue** must be greater than the current clock. If update is enabled (see the **update** command below), the windows are updated at the specified frequency. If all dispatches and instructions have retired before **clockValue** is reached, spat will terminate the **clock** command.

4.4.6 u[pdate] [freq]

Update the windows every **freq** clocks. If **freq** is not specified, the default is 1.

4.4.7 b[reak] –options addr

Set a breakpoint at addr to stop spat when the conditions in **-options** are met. The **addr** may be a hex value or a symbolic tag of a bundle. Tags in dynamically linked libraries will not be resolved. If a tag is entered, the –E <file> option must be used when starting spat. If no options are entered **–csavX** is used as the default. You must enter at least one of the options **cer** and at least one of the options **sav**. The valid options are shown below.

c	Stop when the bundle is cracked by the scalar unit. Bundles contain two instructions, an S and an A instruction, an S and a V instruction or an A and a V instruction.
e	Stop when the instruction enters an execution unit. Some instructions may enter multiple execution units at different times. Some instructions may be dropped early and will break when they are dropped.
r	Stop when the instruction retires. The instruction must completely retire from all execution units where it is scheduled to execute.
s	The breakpoint is set on the s instruction in the bundle.
a	The breakpoint is set on the a instruction in the bundle.

v	The breakpoint is set on the v instruction in the bundle.
0,1,2,3,A	For vector instructions this option is the enable for the ae units. The A option enables all the ae units. A breakpoint will not stop unless the enable is set for the ae unit that is executing the instruction. If none of the enables are specified in the option field, the enables are taken from the spat startup option **-WAEx**. The x will be used for the enable option.

When the command completes the breakpoint number and options used will be printed.

4.4.8 l[ist]

List all the breakpoints currently valid showing the address, name (it entered), options and the number of times the breakpoint has been hit.

```
$ l
Num   Address              Name           Options   Hits
  1   0x0000000000800000   start_test     -csav0    0
```

4.4.9 d[elete] n

Delete breakpoint number n. If n == -1, delete all breakpoints.

5 Generating Instruction Traces using the Convey Simulator

Instruction traces are generated by the Convey Simulator and placed in a file during the execution of the simulator. The generation of the trace records is controlled by setting one or more of the following environment variables before running the Convey Simulator:

- **CNY_SPAT_CMD**
- **CNY_SPAT_FILE**
- **CNY_SPAT_MAX**

The **CNY_SPAT_CMD** environment variable controls which instructions are captured in the trace file during the simulator run:

- **trace_all** – Capture all coprocessor dispatches, memory accesses and instructions in the trace file
- **trace_func_all 0xaaaaa** – Capture all coprocessor dispatches, memory accesses and instructions starting at address 0xaaaaa including all subroutine calls
- **trace_func_only 0xaaaaa** – Capture all coprocessor dispatches, memory accesses and instructions starting at address 0xaaaaa but do not trace any subroutine calls
- **trace_range 0xaaaaa 0xbbbbb** – Capture all coprocessor dispatches, memory accesses and instructions that are 0xaaaaa <= insAddr < 0xbbbbb

The environment variables accept address or names for the trace_func forms and accept only hex addresses for the trace_range form. These addresses may be found using gdb or nm. Examples of tracing an address range (in csh or tcsh) are shown below:

```
setenv CNY_SPAT_CMD "trace_range 0x1000 0x2000"
setenv CNY_SPAT_CMD "trace_func_all routine_name"
```

The default file name for the trace file output is spat.dat. It may be overridden by setting the environment variable **CNY_SPAT_FILE**.

```
setenv CNY_SPAT_FILE <new_output_file>
```

The output may be limited to x records to prevent extremely large output files by setting the environment variable **CNY_SPAT_MAX**. The record size is currently 40 bytes.

```
setenv CNY_SPAT_MAX 100000
```

CAUTION, because of large queue depths in the hardware, tracing should begin at least 1000 instructions before the area of interest (or by using the **trace_all** method which starts tracing at the beginning of the coprocessor dispatch).

Once these environment variables have been set, the user program is run under simulation and the simulator will generate the trace output file.

6 Use Spat to Generate Plot Data of the Executable

The plot data is generated by using the **–PLOT** option to spat:

```
spat –PLOT
spat –PLOT <user_plot_file>
spat –PLOT –F <user_input_file>
spat –PLOT <user_plot_file> -F <user_input_file>
```

The first example will generate a plot file named spat.plot using the default input file spat.dat.

The second example will generate a plot file named <user_plot_file> using the default input file spat.dat.

The third example will generate a plot file named spat.plot using <user_input_file> as input.

The fourth example will generate a plot file named <user_plot_file> using <user_input_file> as input.

There are no other options used with the **–PLOT** option.

7 Using the spat.viewer Program to View a Plot

The program **spat.viewer** is used to graphically display the internal state of the coprocessor as instructions flow through the scalar fma, misc and ldst units. The user has the ability to control what data is displayed in the plot, the time scale of the plot and other elements of the display. The **spat.viewer** requires the Perl TK module be installed on the system. To install the Perl TK module, as root, either install from cpan.org:

```
# perl -MCPAN -e 'CPAN::Shell->install( "Tk" )'
```

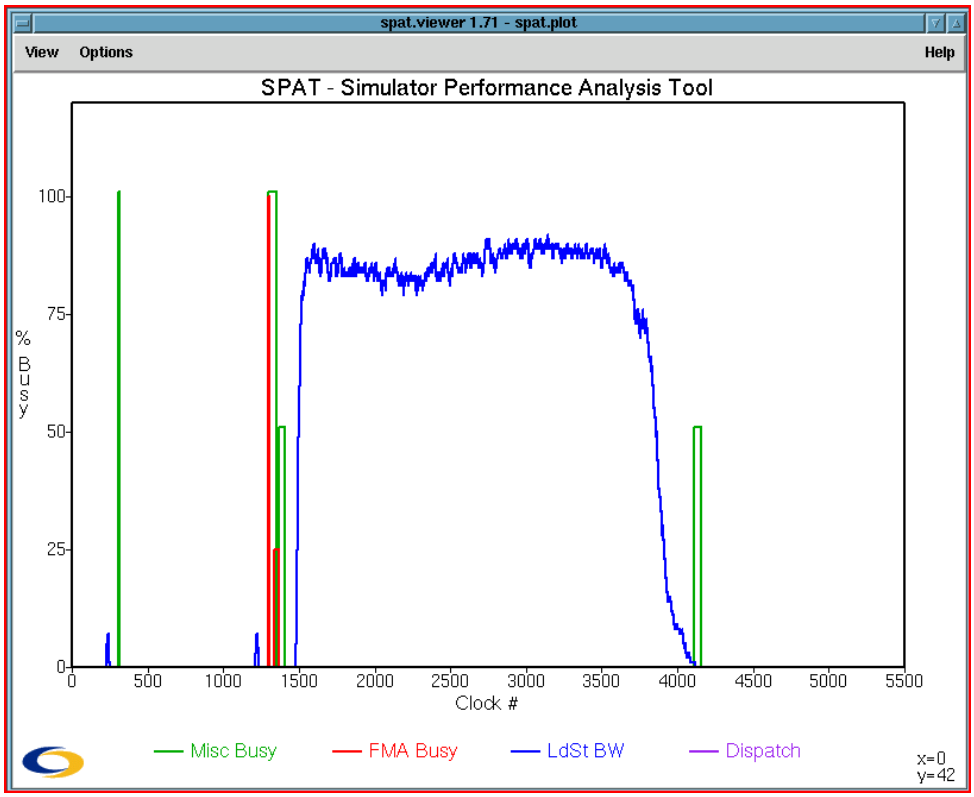
or install from a RPM with a command such as:

```
# rpm -Uhv /vol/utilities/admin/rh-sw/perl-Tk-804.027-1.x86_64.rpm
```

The simplest command to view a plot is:

```
spat.viewer
```

This will read the file spat.plot and generate a window like:



7.1 The Plot Window

The plot window consists of the view pull down menu, the options pull down menu, the help pull down menu, title bar, plot area, left scale (0-100)% busy, right scale (0-n) queue depth, x scale (in Coprocessor Interval Timer Ticks), label area and the cursor position (lower right corner). These may be modified using the options and controls described below.

7.2 Starting the spat.viewer

The command to start the **spat.viewer** is **spat.viewer [options] [<file>]**. The options are in four categories.

7.2.1 Data Selection Options

<file>	Override the default input file spat.dat with <file>. Supported input file types are uncompressed, .gz and .bz2
-a[ll]	Display all data present in the input file.
-b[usy]	Display the percent busy data for the Misc unit and Fma unit. Display the load store bandwidth and the dispatch marker (default).
-sb[usy]	Add the Scalar Busy to the plot.
-q[ueue]	Add the queue depth to the plot. This adds the Misc Fifo, Fma Fifo, Ae Fifo and the Ias Fifo queue depths to the plot.
-o[ther]	Add all other data to the plot.

7.2.2 Plot Settings

-title <text>	Replace the default title with <text>
-fg <color>	Change the foreground color to <color> default is black
-bg <color>	Change the background color to <color> default is white
-bgdark	Use image with dark background
-bglight	Use image with light background

7.2.3 Viewer Controls

-quiet	Disable all output to the controlling terminal
-terse	Enable output to the controlling terminal to show plot progress
-verbose	Enable output to the controlling terminal for all messages

-t[ime]	Show the time each plot step takes
-[no]step	[Don't] draw data as steps (default is step)
-[no]skip	[Don't] skip data points (default is skip). Make large plots faster
-[no]rotate	[Don't] rotate y-axis labels (default is norotate)
-[no]gflops	[Don't] show the Gflops/s data label (default is nogflops)
-[no]gbytes	[Don't] show the GBytes/s data label (default is nogbytes)
-l	Show all data labels

7.2.4 Other Options

-g <geom>	Use the specified geometry (WxH, WxH+X+Y)
-[no]pack	[Don't] pack/conserve memory (default is pack)
-[no]sum	[Don't] summarize data (default is sum)
-h	Print usage
-help	Print detailed help
-version	Print the version

7.3 Controlling the Plot Window

The **spat.viewer** allows the user control of the plot window.

7.3.1 Help Pull Down Menu

The Help pull down menu allows the user to zoom into and out of a selected area of the plot, pan across the plot and control data labels. These controls are executed using the mouse or keyboard and are documented in the help pull down menu in the upper right corner of the plot window.

7.3.2 Options Pull Down Menu

The options pull down menu allows the user to control settings such as plot title, color of each data set displayed, titles and ranges of the axes, fonts and zoom rates. These controls are executed by clicking on the command in the pull down option menu.

7.3.3 View Pull Down Menu

The view pull down menu allows the user to control the viewable data sets, save the plot in different formats and exit the **spat.viewer**. These controls are executed by clicking on the command in the pull down window.

8 Using Spat to Interactively Trace the Execution Stream

There are three types of windows used to display the internal state of the coprocessor:

- Scalar/Ae Display
- Ae Memory Subsystem Display
- Memory Controller Display

The displays are updated at the completion of the **step** command. If the **update** command has been entered, the displays are updated every freq clocks during a **run** or **step count** command. If text is displayed in red, there is at least one hazard that prevents that section of the coprocessor from processing more instructions.

8.1 The Scalar/Ae Window Display

The scalar/ae display window is generated when spat is started with either the **–WAEx** (show scalar and ae unit x) or the **–WAEA** option. If more than one scalar/ae display window is displayed, the scalar information is the same in all windows (because there is only one scalar unit) but because the ae units execute asynchronously the ae information displayed may be different in each scalar/ae window. This difference is usually caused by the execution timing changing depending ae memory accesses completion timing.

A brief description of each sub window:

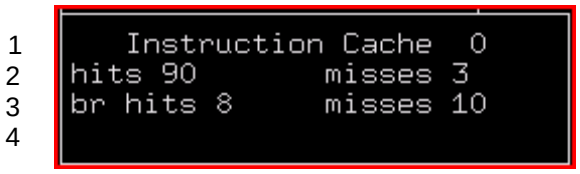
8.1.1 Clock

The count is the number of coprocessor clock ticks. The clock is advanced at the same rate as the hardware CIT register.

8.1.2 Instruction Cache

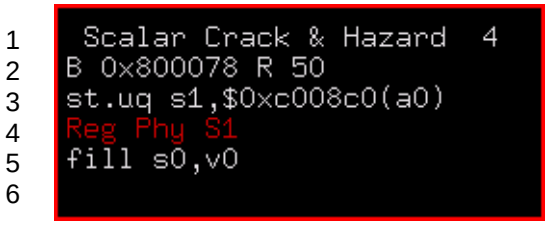
The instruction cache count (first line) is the number of outstanding cache line reads currently in progress. The hits counter (second line) is the number of instruction cache hits that have occurred. The misses counter is the number of cache line misses that have occurred. The br hits counter (third line) is the number of branch prediction hits that have occurred. The misses counter is the number of branch prediction misses that have occurred. The fourth line displays the current bundle ready for the cracker or the hazard that is stalling this unit. The hazards are:

- Dsptch — Waiting for a dispatch cycle to complete. All instructions and Memory transactions must complete. The signature is displayed.
- Instruction Cache Miss — Waiting for an instruction cache line fetch to complete.
- Flow Control Hold — Pipeline stalled for a branch prediction miss.



8.1.3 Scalar Crack & Hazard

The scalar crack and hazard count (first line) is the number of bundles currently in the queue. The second line shows the current cracked bundle address and the spat record number of that bundle. The third line shows the assembly language disassembly of the left half of the bundle. The fourth line shows any hazard preventing the instruction in the left half of the bundle from being executed. Note that for registers, the assembly language register number may not match the hazard register number because of window base registers, register renaming or register rotation. The fifth and sixth lines show the same information for the right half of the bundle. If either half of the bundle is blocked because of a hazard the bundle cannot be released to the execution units.



8.1.4 Scalar Exec

The scalar execution unit consists of eight subunits.

- A register arithmetic/logical unit

- S register arithmetic/logical unit
- S register single precision float unit
- S register single precision high latency unit(divide, square root)
- S register double precision float unit
- S register double precision high latency unit(divide, square root)
- Ae Pipe (instructions going to the ae units)
- Scalar load/store pipe.

The second line shows the A register Alu progress, the count is the number of instructions in the alu pipeline. The progress field (the 15 periods) shows the progress of an instruction from entry into the pipeline (signified by a capital A for one clock, lower case a as the instruction progresses) until the instruction is scheduled into an output time slot in the execution unit (right hand side of the periods). If a "<" is in the leftmost entry of the progress field, there are instructions that will complete in more than 15 clocks. There is a delay from instruction pipeline completion until hazard clearing. An M in the progress field is used for a load instruction returning memory data. If "HL" is displayed to the left of the progress field, the high latency portion of the A register alu is being used (scalar divide).

The third line shows the A register Alu instruction that will retire next. If an instruction is being displayed and there are no letters in the progress field, the instruction is complete and the hazards associated with the instruction are being cleared.

The fourth line is the S register Alu progress and is identical to the A register Alu on the second line except the valid letters in the progress field are C for the S register Alu output, S for the single precision output and D for the double precision output.

The fifth line shows the S register Alu instruction that will retire next. If an instruction is being displayed and there are no letters in the progress field, the instruction is complete and the hazards associated with the instruction are being cleared.

The sixth line shows the S register single precision functional unit. The count is the number of single precision instructions being executed in the different parts of the functional unit. If HL is displayed on the line, the S register high latency unit is being used (divide or sqrt).

The seventh line displays the oldest single precision instruction currently being executed.

The eighth and ninth lines are identical to the sixth and seventh but display for the S register double precision unit.

The tenth, eleventh and twelfth lines display information about the ae pipe. The count is the number of instructions in the queue destined for the ae unit. The Wait Rtn count is the number of outstanding ae instructions that return data to the scalar unit (mov V0, A4, S5).

Lines eleven and twelve show the head and tail of the ae pipe. The head of the pipe is the next instruction to be sent to the ae unit. The tail of the pipe is the newest entry into the pipe.

Lines thirteen, fourteen and fifteen show the scalar unit load store pipe. The count field is the number of outstanding loads and stores. The A count field shows the number of outstanding loads to the a registers. The S count field shows the number of outstanding loads to the s registers.

Lines fourteen and fifteen show the head and tail of the load store pipe. The head of the pipe is the next instruction to be sent to the memory control unit. The tail of the pipe is the newest entry into the pipe.

```
1   Scalar Exec
2  A Alu  1    ...............
3  or a0,$0,a1
4  S Alu  1    ...............
5  or s0,$0,s1
6  S Sp   0
7
8  S Dp   0
9
10 Ae Pipe   1     Wait Rtn  0
11 mov $8,VS
12
13 Ld St Pipe  0    A  0  S  0
14
15
```

8.1.5 The Ae Input Fifo

The AE input fifo receives instructions from the scalar unit and distributes them to the correct ae functional unit fifos (fma, misc, st, ld) when all hazards are clear.

The count on line one is the number of instructions in the fifo.

Line two shows the newest instruction that has entered the fifo.

Line three shows the pc and spat record number of the next instruction to be pulled from the fifo.

Line four shows the disassembly of the next instruction that will be pulled from the fifo.

Line five shows the current hazard blocking the instruction from being pulled from the fifo and sent to the functional unit fifo. Note that an instruction may go to multiple functional unit fifos. The instruction may be blocked by multiple hazards (register and functional unit fifo full for example) but only one is shown. The register shown in the disassembled instruction (line four) may not match the register shown in the hazard because of the window base, register renaming or register rotation.

```
1     Ae_0 Input Fifo  1
2  mov $1024,VL
3  pc 0x800000 R 1
4  mov $1024,VL
5  SpAe CrkRn  298
```

8.1.6 Fma Fifo

The Fma input fifo receives instructions from the ae input fifo and distributes them to the correct fma functional element when all hazards are clear.

The count on line one is the number of instructions in the fifo.

Line two shows the newest instruction that has entered the fifo.

Line three shows the pc and spat record number of the next instruction to be pulled from the fifo.

Line four shows the next instruction that will be pulled from the fifo.

Line five shows the current hazard blocking the instruction from being pulled from the fifo and sent to the fma functional element. Some instructions are not sent to the fma functional elements and retire when pulled from the fma fifo.

8.1.7 Misc Fifo

The Fma input fifo receives instructions from the ae input fifo and distributes them to the correct misc functional element when all hazards are clear. Add and subtract instructions may be sent to the fma functional element if the misc functional elements are unable to accept the instruction at the head of the fifo and the fma functional element has a free functional element.

The count on line one is the number of instructions in the fifo.

Line two shows the newest instruction that has entered the fifo.

Line three shows the pc and spat record number of the next instruction to be pulled from the fifo.

Line four shows the next instruction that will be pulled from the fifo.

Line five shows the current hazard blocking the instruction from being pulled from the fifo and sent to the misc functional element. Some instructions are not sent to the misc functional elements and retire when pulled from the fma fifo.

8.1.8 Store Fifo

The Store fifo receives instructions from the ae input fifo and sends them to the load store functional unit when all hazards are clear. The store fifo only sends to the load store functional unit when the load fifo is empty or blocked on a hazard.

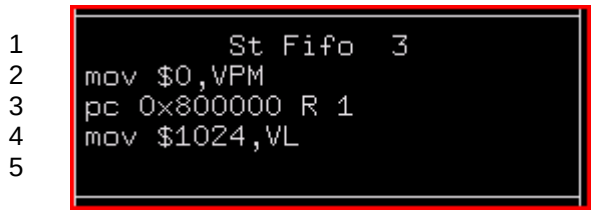
The count on line one is the number of instructions in the fifo.

Line two shows the newest instruction that has entered the fifo.

Line three shows the pc and spat record number of the next instruction to be pulled from the fifo.

Line four shows the next instruction that will be pulled from the fifo.

Line five shows the current hazard blocking the instruction from being pulled from the fifo and sent to the load store functional unit. Some instructions are not sent to the load store functional unit and retire when pulled from the fma fifo.



8.1.9 Load Fifo

The Load fifo receives instructions from the ae input fifo and sends them to the load store functional unit when all hazards are clear. The load fifo has priority over the store fifo.

The count on line one is the number of instructions in the fifo.

Line two shows the newest instruction that has entered the fifo.

Line three shows the pc and spat record number of the next instruction to be pulled from the fifo.

Line four shows the next instruction that will be pulled from the fifo.

Line five shows the current hazard blocking the instruction from being pulled from the fifo and sent to the load store functional unit. Some instructions are not sent to the load store functional unit and retire when pulled from the fma fifo.

8.1.10 Register Rename

The register rename window shows the availability of registers for renaming in the single precision ae. The Vleft field shows the number of single precision left vector registers available. The Vright field shows the number of single precision right registers available. A single precision vector instruction that requires a left and a right vector register (mul.cs) takes a register from the left and right registers. The Vmask field shows the available vector mask registers available for renaming.

8.1.11 Fma Exec

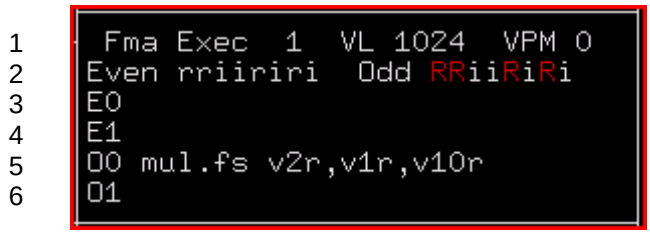
The fma execution unit has four sub units, each capable of processing a single precision fma operation (fma.fs). The units are set in groups (even units 0, 1 and odd units 0, 1) to allow for fma instructions that require two single precision fma operations (fmac.cs). All of the units may be used for instructions that require four single precision operations (mul.cs).

The counter in line one of the display shows the number of instructions currently being processed by the fully pipelined sub units. The current vector length and vector partition mode for the unit

are displayed in the VL and VPM fields. Because of fifo depths, the fma, misc and load store execution units may all have different VL, VPM, VS, and VPS values.

The information displayed in line two shows the register data path usage for the even and odd sub unit pairs. Each sub unit pair has four inputs and two outputs. The first four letters are the inputs (rrii), the r denotes a right vector register, and the i denotes a left vector register. If the letter is capital and red, it shows that input is being used. If the letter is lower case, it is available for use by either unit 0 or unit 1. The last two letters (ri) denote the output data path needed. A new instruction cannot be started until the inputs and outputs (needed later in time) are available.

The third through sixth lines show the instruction currently being executed by the sub unit. Since the sub units are pipelined, more than one instruction may be executing in each sub unit.



8.1.12 Misc Exec

The misc execution unit has two sub units (upper and lower) that may operate independently or as a single unit.

The counter in line one of the display shows the number of instructions currently being processed by the sub units. The current vector length and vector partition mode for the unit are displayed in the VL and VPM fields. Because of fifo depths, the fma, misc and load store execution units may all have different VL, VPM, VS, and VPS values.

The information displayed in line two shows the register data path usage for the sub units. Each sub unit has two inputs and one output. The first four letters are the inputs (rrii), the r denotes a right vector register, and the i denotes a left vector register. If the letter is capital and red, it shows that input is being used. If the letter is lower case, it is available. The last two letters (ri) denote the output data path needed. A new instruction cannot be started until the inputs and outputs (needed later in time) are available.

The third and forth lines show the instruction currently being executed by the sub unit. Since the sub units are pipelined, more than one instruction may be executing in each sub unit.

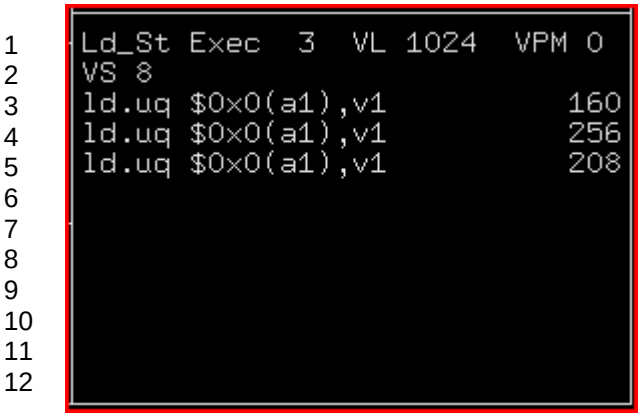
8.1.13 Ld_St Exec

The ld_st execution unit generates load, store and fence memory transactions and dispatches them to the ae memory subsystem. It can under certain conditions combine two 32 bit loads to form a single 64 bit load.

The counter in the first line shows the number of instructions generating transactions and waiting for load data to return. The current vector length, vector partition mode and vector stride are

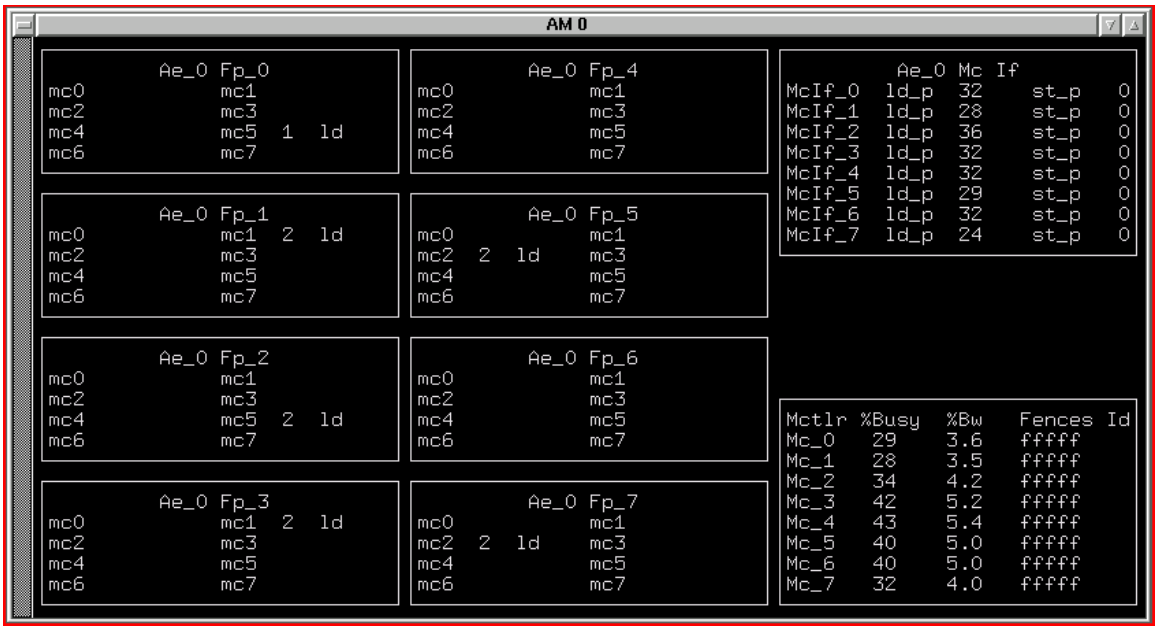
displayed in the VL, VPM and VS fields. Because of fifo depths, the fma, misc and load store execution units may all have different VL, VPM, VS, and VPS values.

The remaining lines in the ld_st exec sub window display the instructions generating transactions and waiting for load data to return (there may be more than can be displayed). The number on the right is the number of outstanding memory transactions for that instruction. Normally, this number will count up from 0 to the max number of memory transactions this ae will issue (normally this number is VL / 4 but VPM and VPA can change this) and then count back to 0 as the memory transactions retire. Store and fence instructions count up and then retire as there is no return values associated with them. If any ld/st line is displayed is red, the instruction is blocked from generating any memory transactions until the Ae Fp fifo has room to accept a new transaction.



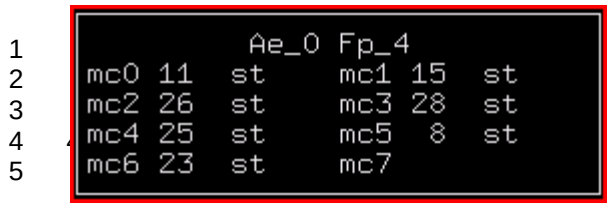
8.2 The AE Memory Subsystem Display

The ae memory subsystem display window is generated when spat is started with either the **-WAMx** (show ae memory subsystem x) or the **-WAMA** option. The memory controller information (lower right corner) is the same in all ae memory subsystem windows.



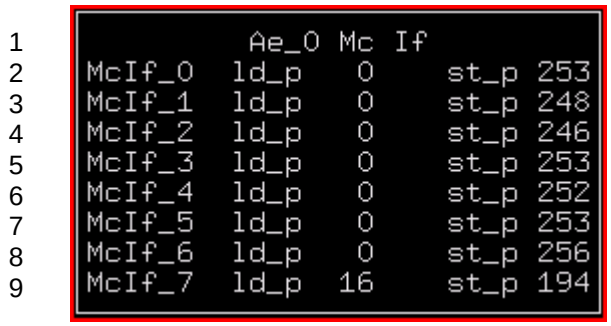
8.2.1 Ae Fp

There is one Ae_X Fp_Y sub window for each function pipe in the ae memory subsystem being displayed. The X is the ae number (0-3) and the Y is the function pipe number (0-7). Within the Ae Fp sub window are eight sections, one for each destination memory controller (mc0-mc7). Each section displays a count of the number of memory transactions currently in the fifo destined to that memory controller from this function pipe and the type of memory transaction (load, store or fence). If the entry is red the maximum number of memory transactions allowed in the fifo has been reached and input to this fifo is stalled.



8.2.2 Ae Mc If

The sub window Ae_X Mc If displays information for each interface to the eight memory controllers. The X is the ae number (0-3). The ld_p counter shows the number of outstanding loads to that memory controller from this ae. The st_p counter shows the number of outstanding stores to that memory controller from this ae. If the entry is red, the number of outstanding memory transactions has reached the maximum and output to the memory controller is stalled.



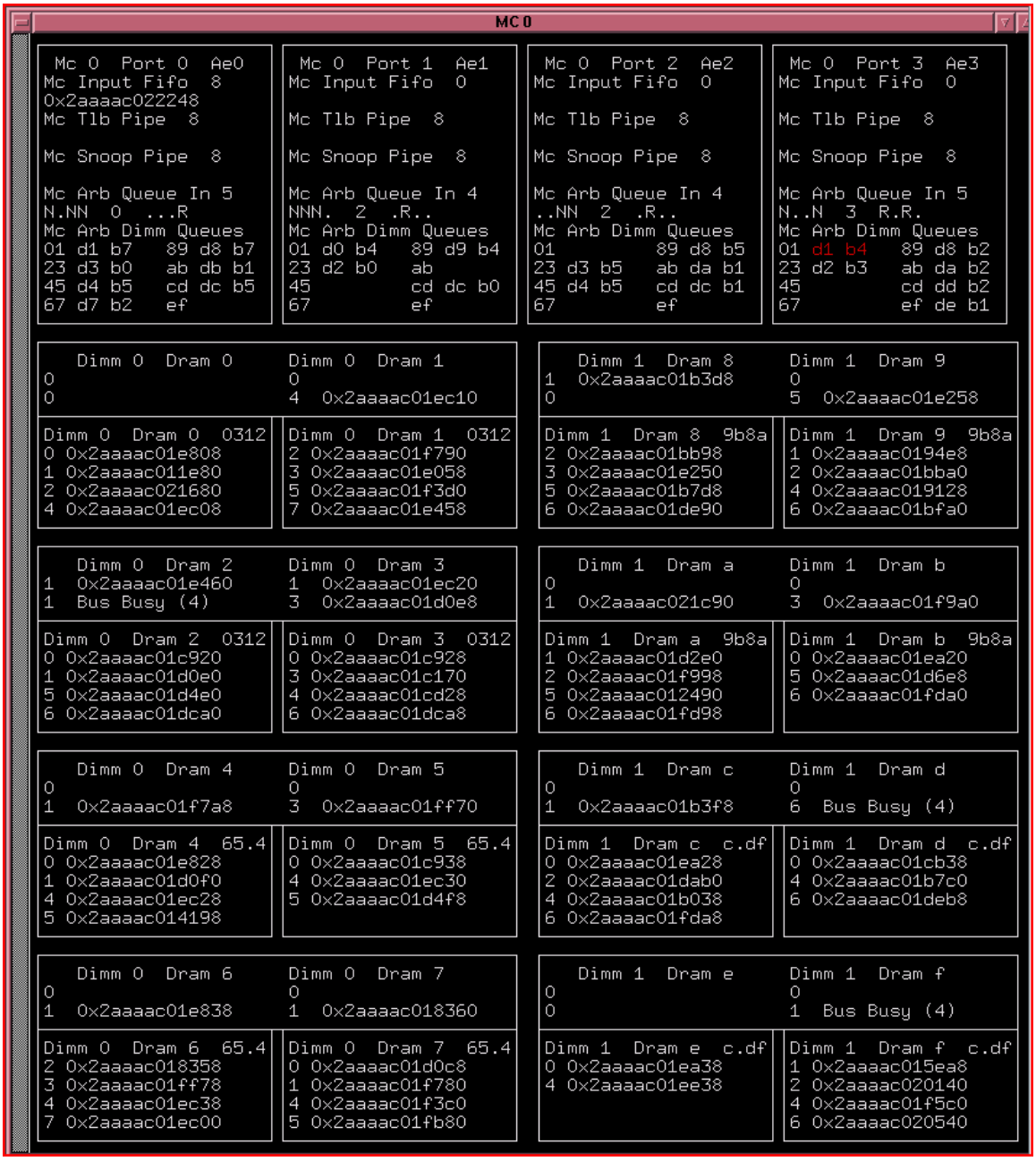
8.2.3 Mem Ctlr

The mem ctlr sub window is common across all ae memory subsystem displays and shows the status of all eight memory controllers. For each memory controller there are four items displayed.

The %Busy field shows the % busy for that memory controller (100% is max). The %Bw field shows the bandwidth percentage that that controller contributes to the overall bandwidth of the system (12.5% is max). Note that the busy field may be at 100% and the bandwidth field may be only 6.2%. This will be the case for four byte accesses with a stride not equal to four. The Fences field displays the status of fences as they progress through the system. The first four f's represent the fence state from each of the ae units (left f is ae 0). The fifth f is the fence from the scalar unit. If the f is lower case there is no fence operation in progress. If the f is upper case the memory controller has received a fence from the ae or scalar unit. If a fence operation is in progress in a memory controller, the memory controller will block all future memory transactions until it has received a fence from all the ae units and the scalar unit. The Id field is used to display the (spat generated) id of the current fence.

8.3 The Memory Controller Display

The memory controller display window is generated when spat is started with either the **–wMCx** (show memory controller x) or the **–wMCA** option. This window shows the four inputs, one from each ae (the scalar input is injected into the ae0 port below the tlb), the snoop, tlb, arb, dram queue and each dram.



8.3.1 Memory Controller Input Port

The memory controller input port accepts memory transactions from the ae memory controller interface (ae mc if). Port 0 on each memory controller also accepts memory transactions from the scalar unit. The first line of the memory controller port identifies which memory controller (0-7), which port (0-3) and which ae sends to the port (0-3).

The second line shows the input fifo for the port. The count is the number of memory transactions in the fifo. The third line shows the address of the newest entry in the fifo.

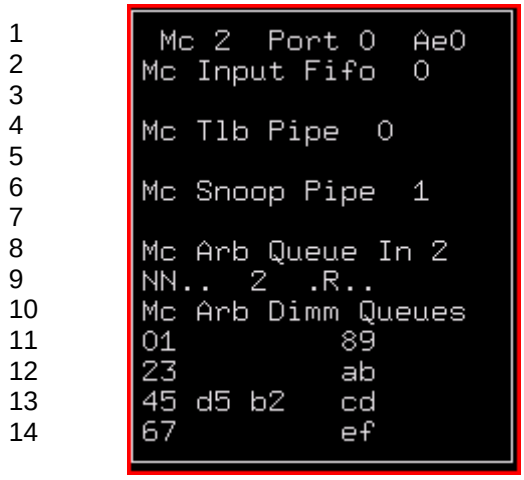
The fourth line shows the number of memory transactions in the tlb translate pipe. The fifth line shows any hazards preventing the tlb pipe from feeding the snoop pipe.

The sixth line shows the number of memory transactions in the snoop pipe. The scalar memory transactions are injected here (only on port 0). The seventh line shows any hazards preventing the snoop pipe from feeding the arb stage.

The eighth line shows the number of memory transactions in the arb queue.

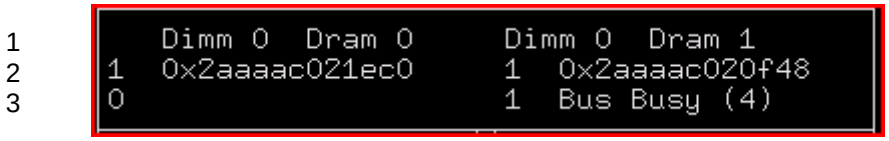
The ninth line shows the state of the arbiter. The left four period field displays the four inputs that will be used at the next arbitration cycle. The first period is for arb queues 0145, the second for 2378, the third 89cd and the fourth is for abef. The queue is selected based on the memory transaction dram destination (0-f). If a period is present, there is no memory transaction for that set of queues. If an N is present there is a memory transaction for that set of queues. The count field is the round robin value that points to the next set of queues to test (0-3). The right four period field shows the output of the arbitration process with the same queues as the left field. If a D or B is displayed the arbiter is blocked (B) or delayed (D) because of timing restrictions.

The eleventh through fourteenth lines show the output queues being fed by the arbiter. These queues (01, 23 … ef) accept memory transactions destined to the drams. The d and b fields are the destination dram and bank in the dram. These queues feed the dimm controllers.



8.3.2 Dimm Ctlr

The dim controller sub window shows the memory transactions destined to each dram. The window shows the inputs for a pair of drams. The second line shows the number of memory transactions in the input queue and the address the transaction is accessing. The third line shows the number of transactions destined to the dram pipeline. If the dram cannot accept the memory transaction the reason is displayed on the third line.

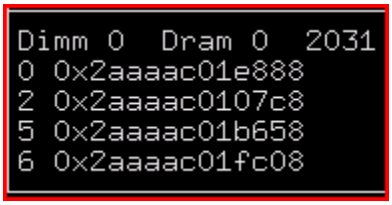


8.3.3 Dram

The dram sub window displays the state of a dram. The first line displays the Dimm number (0-1) the dram is associated with, the dram number in hex (0-f) and a four stage pipe showing the bank number of memory transactions destined to this dram. Lines 2-5 show the active banks of the

dram, the first number is the bank number, and the second number is the address being accessed (in hex).

```
1
2
3
4
5
```



9 Using the Plot and the Interactive Trace to Enhance Performance

The performance of a program can be decreased by a variety of factors. One of the major performance inhibitors is a loop that did not vectorize as expected. Another inhibitor is not using all available hardware resources.

9.1 Using Spat to Find Non-Vectorized Code

This example shows how to use spat to locate code that should vectorize but did not. In this example the vectorizer was turned off by a pragma for one loop of the code to simulate a routine that did not vectorize. The file work.c is shown below and is called from the main program vm_v.

```
void
work_p(float *fp1, float *fp2, float *prod, int size)
{
    int         i;

    for(i=0; i<size; i++) {
        prod[i] = fp1[i] * fp2[i];
    }
}

void
work_s(float *fp1, float *fp2, float *sum, int size)
{
    int         i;

#pragma cny no_vector
    for(i=0; i<size; i++) {
        sum[i] = fp1[i] + fp2[i];
    }
}

void
work_d(float *fp1, float *fp2, float *diff, int size)
{
    int         i;

    for(i=0; i<size; i++) {
        diff[i] = fp1[i] - fp2[i];
    }
}
```

The steps to compile, setup spat tracing, generate the trace file, generate the plot file and view the plot file are shown below. They are run using tcsh.

```
$  /opt/convey/bin/cnycc -O3 -c -mcny_vector -mcny_dual_target
      mcny_vec_warn work.c -o work_v.o
Vectorizing cny_work_p
VECTOR LOOP in cny_work_p at 6: L 2 S 1 B 1 U 0 I 0 M 0
Vectorizing cny_work_s
```

```
Vectorizing cny_work_d
VECTOR LOOP in cny_work_d at 27: L 2 S 1 B 1 U 0 I 0 M 0
$  /opt/convey/bin/cnycc -O3 -mcny vec_mul.c work_v.o -o vm_v
```

The section above will compile the file work.c and enable the compiler to generate vector code. The output shows two loops were vectorized, one at line 6, the other at line 27. In this simple case it is easy to look at the compiler output and see only two of the three loops vectorized. In normal code the user may have tens or hundreds of loops vectorized and finding one not vectorized would be difficult.

```
$  nm vm_v | grep work
0000000000c00544 D cny$vec_ok_cny_work
0000000000800000 W cny_work_d
00000000004058b0 T work_d
$  setenv CNY_SPAT_CMD trace_func_all cny_work_d
```

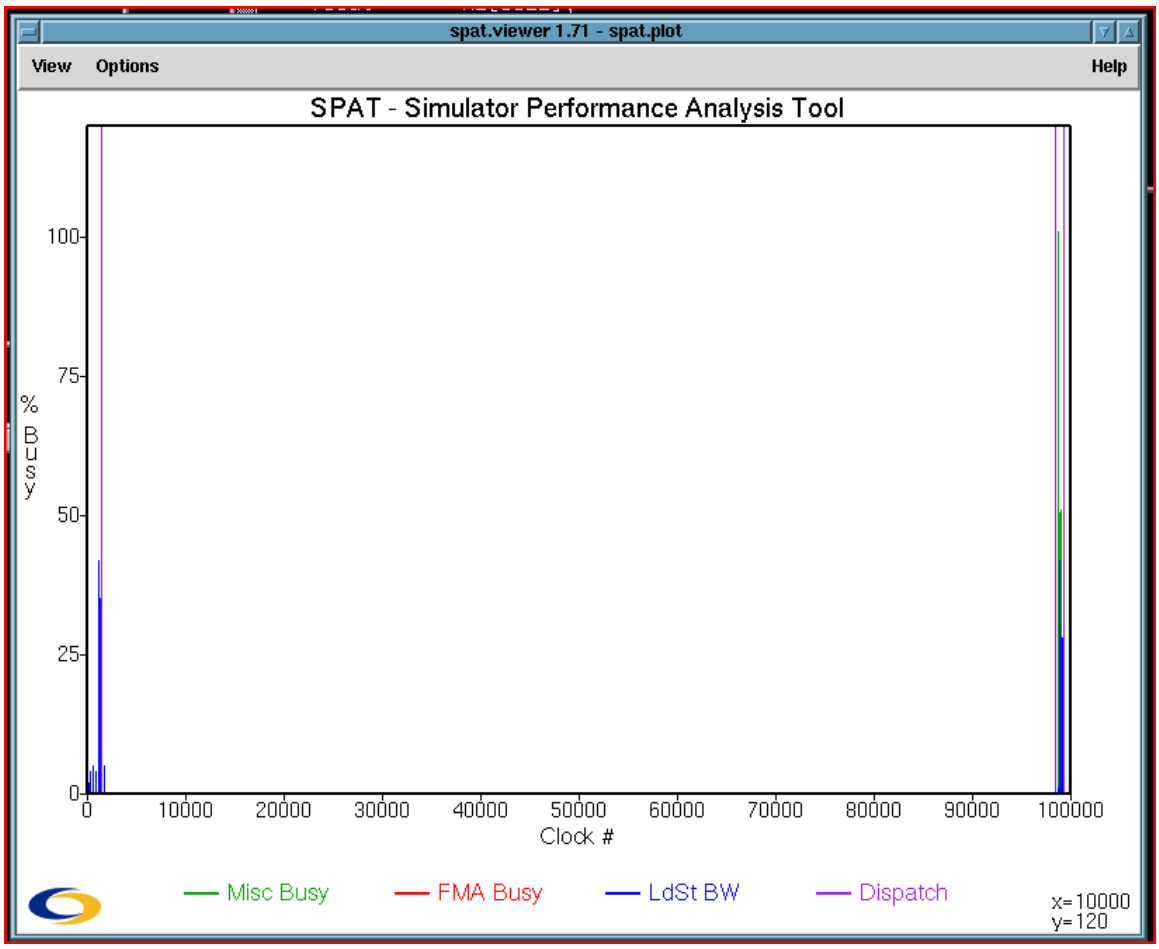
The section above shows one method to setup spat to trace the function work_d by using the nm tool. If the routine was compiled with the –mcny_dual_target option each routine will have two names, work_d and cny_work_d. The routine work_d is the native cpu code and the routine cny_work_d is the same routione running on the coprocessor. This command will cause the simulator to generate trace records for all instructions in the routine cny_work_d and all routines that cny_work_d calls. In this example we want all the routines traced so the trace_all option is used.

```
$  setenv CNY_SPAT_CMD trace_all
$  setenv CNY_SIM_THREAD cpSimLib2.so
$  setenv LD_LIBRARY_PATH /opt/convey/lib
```

The commands above are used to setup the simulator to execute the code.

```
$  vm_v
$  spat –PLOT
$  spat.viewer
```

The commands above execute the program vm_v to generate the trace file spat.dat. The spat command will read the spat.dat file and generate the spat.plot file. The spat.viewer will read the spat.plot file and generate the plot file below.



Looking at the plot will show that there is a large gap between clocks 1600 and 98000 where there is no vector processing being done. If the Scalar Busy data trace is enabled, it will show that this time is being spent in the scalar unit. To find the routine that did not vectorize, use a time in the suspected range of 1600 to 98000 (10000 in this example) and run the spat program again using the –WAE0 option.

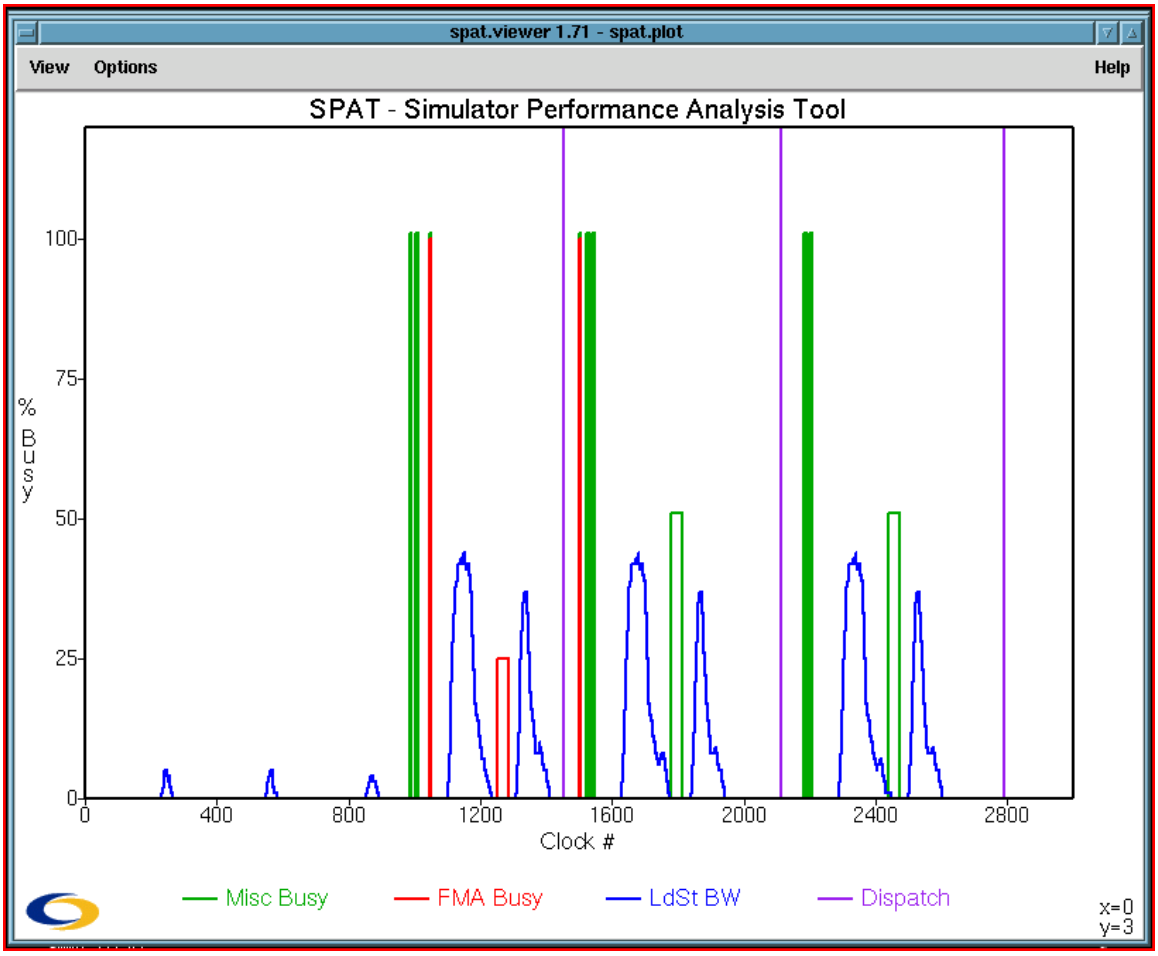
```
$  spat -WAE0
spat: c 10000
stopping on clock 10000
```

The clock was advanced to 10000 using the clock command and the AE0 window should be:

The scalar unit is cracking the instruction bundle at address 0x800380 for execution next. Use the command **nm vm_v -v** to show the routine addresses (partial output below). The output shows that the routine work_s contains the bundle at address 0x800380.

```
$  nm vm_v -v
0000000000800000 T __cny_ctext_start
0000000000800000 W cny_work_p
00000000008001b0 W cny_work_s
00000000008003e8 W cny_work_d
0000000000800598 T cny_pdk_dispatch_coproc
```

Now that the non-vectorized code has been identified, the user may fix the reason it was not vectorized (in this case removing the #pragma cny no_vector). Re-compile, execute and generate the plot file will result in a plot:



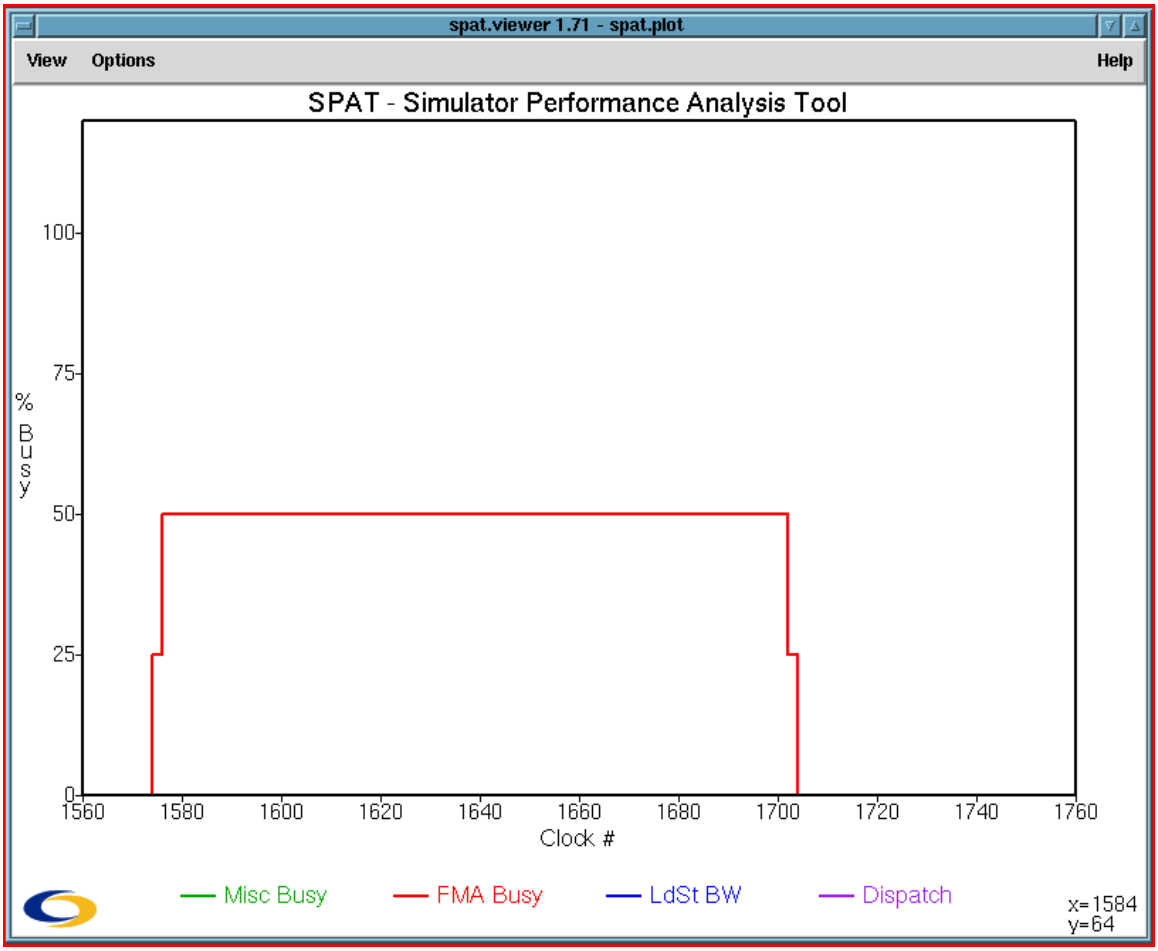
This plot shows the three dispatch regions with vector processing in each region.

9.2 Using Spat to Optimize Assembly Code

This example shows how to use **spat** to optimize assembly language code to use all functional units of the coprocessor. The subroutine fma_code does a series of fma.fs instructions using v0r through v15r and s3 as inputs.

```
fma_code_slow:
        fma.fs  %v0r, %s3, %v1r, %v16r
        fma.fs  %v2r, %s3, %v3r, %v17r
        fma.fs  %v4r, %s3, %v5r, %v18r
        fma.fs  %v6r, %s3, %v7r, %v19r
        fma.fs  %v8r, %s3, %v9r, %v20r
        fma.fs  %v10r, %s3, %v11r, %v21r
        fma.fs  %v12r, %s3, %v13r, %v22r
        fma.fs  %v14r, %s3, %v15r, %v23r
        rtn
```

Use **spat** and **spat.viewer** to generate a plot file and expand the time scale for the fma_code subroutine to give this plot. Note that the FMA Busy line is at 50% which means that only half of the fma units are being used. The total time for the subroutine is 131 clocks.



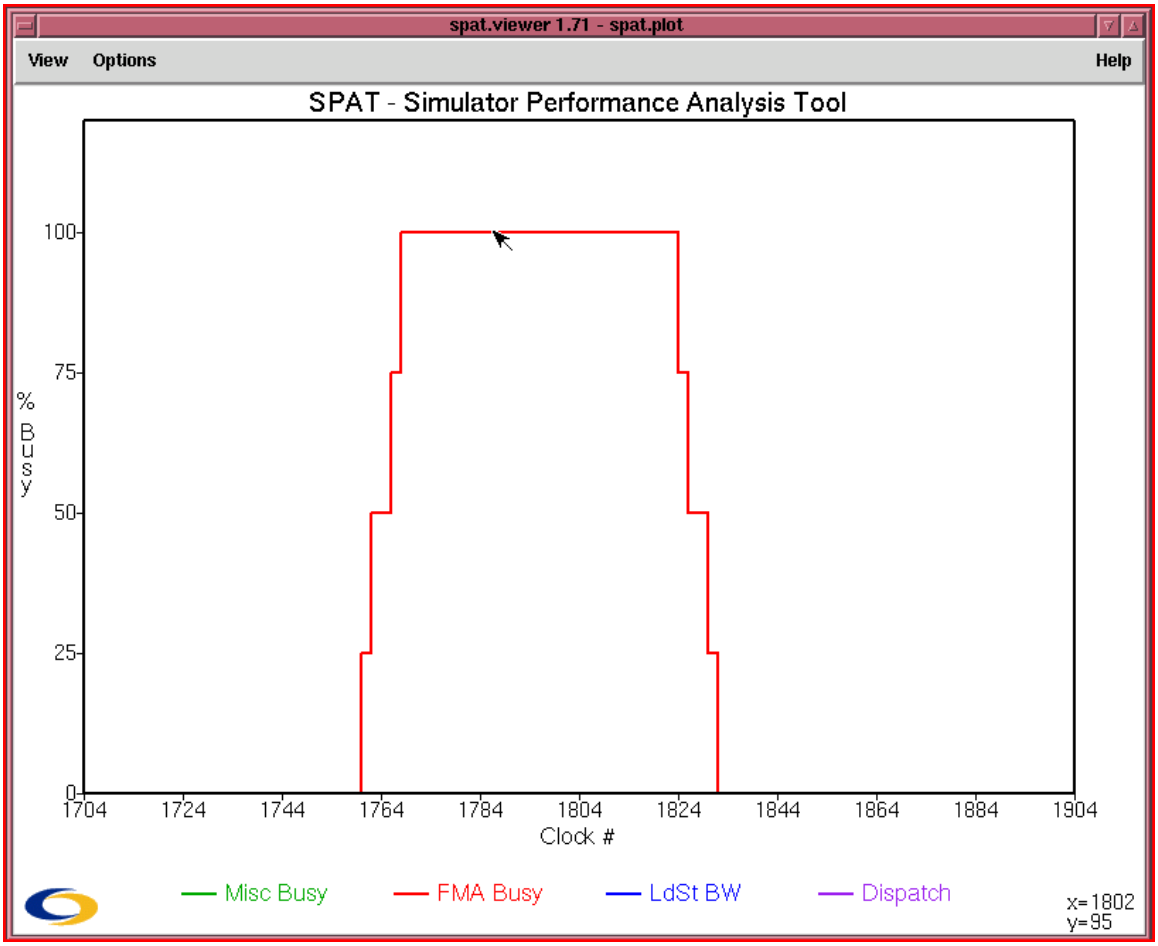
Using the **`spat -WAE0`** command and stopping at a time while the FMA Busy is at 50% (1620 for this example) will display this AE window.

The Fma Exec sub window shows two instructions are currently executing (on the E1 and O1 units), and all the R inputs and outputs are being used. The Fma Fifo sub window shows four instructions currently in the fifo and the **`fma.fs v9r, s3, v20r`** has a hazard on the **`Fma In R`** preventing it from starting execution. This code only uses the right registers, the left register inputs and outputs of the fma are not used which limits the number of fma instructions that can be executed in parallel.

To optimize the code it should be modified to use both left and right registers as shown.

```
fma_code_fast:
        fma.fs  %v0r, %s3, %v1r, %v16r
        fma.fs  %v0l, %s3, %v1l, %v16l
        fma.fs  %v2r, %s3, %v3r, %v17r
        fma.fs  %v2l, %s3, %v3l, %v17l
        fma.fs  %v4r, %s3, %v5r, %v18r
        fma.fs  %v4l, %s3, %v5l, %v18l
        fma.fs  %v6r, %s3, %v7r, %v19r
        fma.fs  %v6l, %s3, %v7l, %v19l
        rtn
```

Running the new code gives the plot below showing the total time for the subroutine is now 71 clocks.



Running the `spat -WAE0` command and stopping on clock 1790 shows all four of the fma units are now being used.


AE 0
Clock 1790
Instruction Cache 0
hits 90 misses 3
br hits 8 misses 10
Scalar Crack & Hazard 17
B 0x800138 R 154
br.f sc0.eq,$-0x10
Reg Phy C1000
nop
Scalar Exec
A Alu 0 .....
S Alu 1 .....
cmp.uq s4,s5,sc0
S Sp 0
S Dp 0
Ae Pipe 0 Wait Rtn 0
Ld St Pipe 0 A 0 S 0
Ae_0 Input Fifo 0
Fma Fifo 4
fma.fs v6l,s3,v7l,v19l
pc 0x8001b0 R 137
fma.fs v4r,s3,v5r,v18r
FMA 1 Units
Misc Fifo 0
St Fifo 0
Ld Fifo 0
Register Rename
Vleft 64 Vright 64 VMask 16
Fma Exec 4 VL 1024 VPM 0
Even RRIIRIRI Odd RRIIRIRI
E0 fma.fs v0r,s3,v1r,v16r
E1 fma.fs v2l,s3,v3l,v17l
O0 fma.fs v0l,s3,v1l,v16l
O1 fma.fs v2r,s3,v3r,v17r
Misc Exec 0 VL 1024 VPM 0
rriiri
U
L
Ld_St Exec 0

10 Personality Support

Spat supports the single precision and double precision personalities but does not support the custom personality. The scalar unit is identical for single and double precision personalities and feeds the ae fifo in the same manner. The ae fma and misc execution units for single precision and double precision are different; the load/store unit is the identical. The double precision fma execution unit has two fma units (even and odd) and the misc execution unit has a single misc unit. The vector registers are 64 bits wide and do not have a left and right half. The hazards and timing of the double precision units are the same as the single precision units.

The ae memory subsystem and the memory controller are identical in the single and double precision personalities.

Index