

VM LABS



NUONTM Multi-Media Architecture

Programmer's Guide

Revision 26

October 24th, 2001

VM Labs, Inc.
520 San Antonio Road
Mountain View, CA 94040
Tel: (650) 917 8050
Fax: (650) 917 8052

This version of the documentation is for programmers using the NUON system. It omits some hardware details, as these are accessed through the BIOS.

**Copyright © 1995-2001 VM Labs, Inc.
All Rights Reserved.**

The information contained in this document is confidential and proprietary to VM Labs, Inc.

It may not be distributed or copied in any form whatsoever without the prior written permission of VM Labs.

NUON, NUON Multi-Media Architecture, the NUON logo, and the VM Labs logo are trademarks of VM Labs, Inc.

This is a preliminary specification. VM Labs reserves the right to make changes to any information in this document.

CONTENTS

Introduction	4	DMA Commands	148
Overview	4	DMA Command Fields	151
Internal Architecture	5	MPE DMA Control and Status	
Overview	6	Register	157
Memory Map	7	Backward Pixel Transfers	159
Differences between Aries 2 and		DMA Pixel Types	159
Aries 3	7		
Differences between Aries 1 and		Communications bus	161
Aries 2	9	Communication Bus Identification	
Compatibility with future NUON		Codes	161
Architectures	10	Data Transfer Protocol	162
Conventions	11	Data Flow Control	163
		Communication Bus Control Flags	163
Media Processor Elements	12	IO Devices on the Communication	
Overview	12	Bus	164
Memory maps	13	Other Bus	165
Instruction and Data Cache	14	Command Format	165
Register File	15	Restrictions on Other Bus DMA	166
Instruction Flow	17	Control Registers	166
MPE Function Units	19	System Bus	167
Arithmetic Logic Unit (ALU)	19	Communication with an External	
Multiply Unit (MUL)	20	Host Processor	172
Execution Control Unit (ECU)	22		
Register Control Unit (RCU)	26	ROM Bus	179
Memory Unit (MEM)	27	ROM Communication Bus	
		Interface	179
MPE Register Set Reference	39		
MPE Instruction Set Reference	60	Video Output & Display Time-Base	180
		Video Data Flow, Filtering, and	
Main Bus	146	Scaling	180
Arbitration	146	Video Data Encoding	185
Main Bus DMA Controller	147		

INTRODUCTION

The NUON Aries 3 described in this document is the chip at the heart of the NUON Multi-Media Architecture (the MMA). The hardware engineers who created the NUON chip wrote this document, so this is the definitive reference work describing it. However, this document does not describe the associated software, or any particular implementation of NUON.

The variants of the NUON device are currently:

- Aries 1 (MMP-L3B) is the first production NUON device. It is the successor to the pre-production prototype known as Oz (MMP-L3A).
- Aries 2 (MMP-L3C) is the successor to Aries 1. The few differences are described below. Functionally Aries 1 and 2 are largely identical.
- Aries 3 is the most recent version at the time of writing. Aries 3 offers faster operation and larger on-chip memories, so has increased functionality.

Note that systems based around Oz are obsolete prototypes, and applications written for the NUON Architecture do not have to be compatible with them.

The NUON Architecture was developed to provide a high performance yet very cost-effective solution to the processing and content requirements of the next generation of consumer multimedia systems.

Content developers can target their interactive applications to a single development platform. NUON compatible applications can then run on any product that incorporates the NUON Architecture, as long as they conform to some specific rules. These rules are beyond the scope of this document.

Overview

This table summarizes the most significant Aries 3 changes relative to Aries 2 and Aries 1.

	Aries 1	Aries 2	Aries 3
Process	0.35u 5M	0.25u 5M	0.18u 6M
Package	356 TEBGA (256+100)	272 BGA (256+16)	208 PQFP (option for 256 BGA)
VDD Core	2.5 \pm 5% Volts	2.5 \pm 5% Volts	1.8 \pm 5% Volts
VDD I/O	3.3 \pm 5% Volts	3.3 \pm 5% Volts (5 VT)	3.3 \pm 5% Volts (5 VT)
Power	3.1 Watts max 2.3 Watts typical for DVD	2.8 Watts max 2.2 Watts typical for DVD	1.8 Watts max (108 MHz) 1.4 Watts typical for DVD
MPE Ram	Mpe0 8/8 cached Mpe1 4/4 Mpe2 4/4 Mpe3 4/4 cached	Mpe0 8/8 cached Mpe1 4/4 Mpe2 4/4 Mpe3 4/4 cached	Mpe0 26/20 cached Mpe1 16/16 Mpe2 16/16 Mpe3 20/20 cached
MPE Speed	54 MHz	54 MHz	54 and 108 MHz modes
Mainbus DRAM	108 MHz SDRAM x16 (2 bank only)	108 MHz SDRAM x16 (2 bank or 4 bank)	108 MHz SDRAM x16
Sysbus DRAM	27 MHz EDO x32	54 MHz SDRAM x16 or 27 MHz EDO x32	54 MHz SDRAM x16
Other Features	+ SD MPEG2 decode + DVD CSS-1 + DVD subpicture + CCIR 656 video out + video overlays, scaling + CCIR 656 video in + 6 audio out channels + 2 audio in channels + NUON Controller Ports + I2C master / slave I/F	+ split master / slave I2C + sysbus enhancements	+ 2 audio out ch. (8 total) + Dual SIO ports + Larger audio-out FIFO + Glueless ROM I/F + Glueless 8bit flash I/F + 108 MHz PLL + Audio PLL

Internal Architecture

At the heart of the NUON architecture are four processors, known as MPEs (or NUON Media Processor Elements). These are VLIW processors with five function units, and each processor has its own private program and instruction memory. They run at up to 108 MHz, and can execute a maximum of five instructions per clock cycle, although because you can actually independently decrement two counters as well, we claim that we can execute seven instructions per clock cycle. The MPEs are described in great detail later in this document.

Each of the four MPEs, referred to as MPE0 to MPE3, has the same processor core, and all can run the same code. However, MPEs 0 and 3 also contain cache controllers, so that one of them (or, rarely, both) can execute larger programs than will fit in the program memory.

Three busses run between the MPEs, allowing them to talk to each other and to external memory. These busses are:

1. The Main Bus – this is a 32-bit bus with a maximum data transfer rate of 216 Mbytes/sec either between MPE memory and external SDRAM, or from one MPE to another. This bus is optimized for transferring bursts of data, and has extensive support for pixel transfer, including bi-linear addressing and Z-buffer compares. It is also used for video and audio output. All NUON systems will have a minimum of 8 Mbytes of SDRAM on this bus.

2. The Communication Bus – this is another 32-bit bus, with a maximum data transfer rate of around 172 Mbytes/sec, and is used for transferring 128-bit packets either between the MPEs, or to allow MPEs to talk to peripheral devices. This is a very low latency bus, and is ideal for inter-processor communication.
3. The Other Bus – this is a 16-bit bus, and is like a simpler slower version of the Main Bus. It is used to talk to System Bus memory. It can only perform linear data transfers, at a maximum rate of 108 Mbytes/sec. All NUON systems will have a minimum of 8 Mbytes of DRAM on this bus.

These busses may all be used by explicitly requesting transfers, and the cached MPEs may also implicitly use the Main and Other busses to execute code and transfer data.

The major blocks of NUON device are summarized in this diagram:

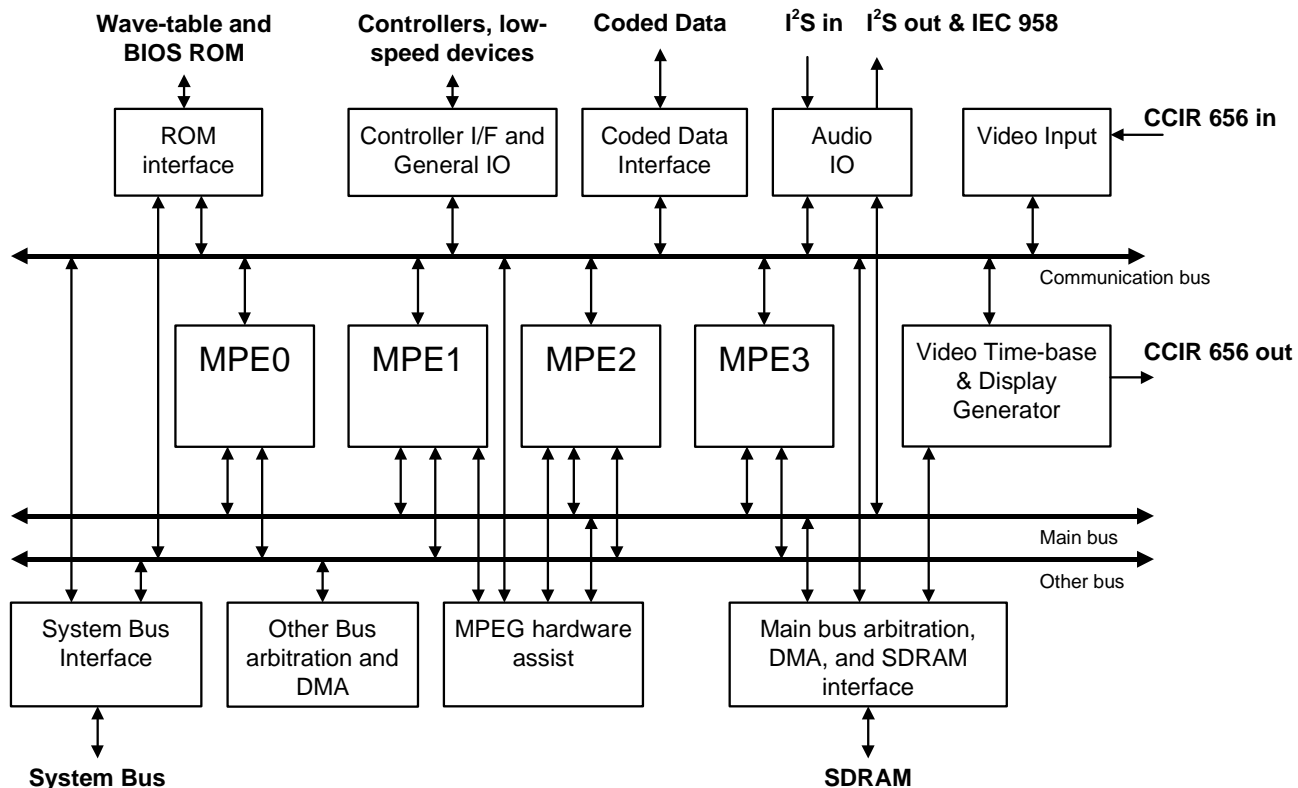


Figure 1. NUON Internal Architecture

Overview

The NUON Media Processor is designed to be a high performance, low cost, interactive alternative to the audio, video, graphics and processor requirements of a consumer MPEG-2 product.

In order to achieve these goals, we chose a parallel processing architecture. A set of four Media Processor Elements (MPEs) provides the performance necessary for high-end media applications (such as MPEG-2 and 3D graphics).

The MPEs are each fully programmable, very-long-instruction-word (VLIW) processors. Each MPE can independently saturate the memory busses if necessary, replacing the need for custom Bit-Blit functions. MPEs each contain a scalar and vector register set, a 32 x 32 bit multiplier, a 64-bit ALU and barrel shifter, linear and bi-linear address generation and a powerful execution control unit. Up to seven

operations using these function units can operate in parallel, resulting in extremely efficient inner loops. Code compression and efficient execution control units allow MPEs to also perform complex outer loop operations. This level of programmability provides the opportunity to avoid the unnecessary calculations and bus operations often associated with SIMD or hardwired architectures.

MPEs can efficiently perform 3D geometry operations (such as vector arithmetic), image operations (such as texture mapping, filtering and shading), data transformations (such as Huffman coding and decoding, and Cosine Transforms), data sorting and complex decision making, all in software.

While the instruction set has been optimized for these types of operations, we made great efforts to retain generality. This generality provides developers with the opportunity to experiment with new types of low-level algorithms, as well as at higher levels.

The problems of image manipulation and 3D graphics are very amenable to a parallel architecture. Many good algorithms exist which can make use of multiple parallel processors (MPEs in this case). First generation MMP devices will contain four MPEs. Future versions may contain many more than this. The architecture and BIOS contain specific features to support upward compatibility. Indeed, correctly written applications will actually take advantage of the additional processors in future versions.

Memory Map

The memory map is a single 32-bit byte-address space from the perspective of the MPEs. Two distinct hardware mechanisms are provided for accessing this memory, the Main Bus and the Other Bus. The mechanism required to access each space is shown in this table.

Address	Size	Main Bus	Other Bus	Description
\$0000 0000 - \$1FFF FFFF	512M	–	–	Reserved
\$2000 0000 - \$2FFF FFFF	32x8M	✓	✓	MPE Memory and I/O spaces
\$3000 0000 - \$3FFF FFFF	256M	–	–	Reserved
\$4000 0000 - \$7FFF FFFF	1024M	✓		Main Bus DRAM (media RAM)
\$8000 0000 - \$8FFF FFFF	256M		✓	System Bus DRAM
\$9000 0000 - \$9FFF FFFF	256M		✓	System Bus ROM / SRAM 0
\$A000 0000 - \$AFFF FFFF	256M		✓	System Bus ROM / SRAM 1
\$B000 0000 - \$EFFF FFFF	1024M	–	–	Reserved
\$F000 0000 - \$F0FF FFFF	16M		✓	ROM – BIOS & Audio wave-tables
\$F100 0000 - \$FFFF FFFF	239M	–	–	Reserved
\$FFF0 0000 - \$FFFF FFFF	1M		✓	Other Bus IO

Differences between Aries 2 and Aries 3

QFP-208 and BGA-256 Package Options

Aries 3 will be available in a QFP-208 package, as well as in a BGA-256 package that is nearly drop-in compatible with current Aries 2 systems (the only required changes are to lower the core voltage supply as shown above, and to update the boot ROM/FLASH program).

Integrated PLL for 108 MHz Clock Generation

Aries 3 integrates a PLL which allows a cheaper 27 MHz external crystal to be used instead of the 108MHz crystal needed with Aries 2. This PLL generates the internal 108 MHz and 54 MHz and 27

MHz clocks. For drop-in compatibility with Aries 2 systems, the PLL is bypassed so that an external 108 MHz crystal can feed the clock input of the chip.

Integrated PLL for Audio Clock Generation

Aries 3 integrates a second PLL which is used to generate a master audio clock phase-locked to the video clock. This allows the use of cheaper audio DACs that do not include the PLL needed for Aries 2 systems. The Aries 3 audio PLL can be bypassed so that an external audio clock can be input as it is in Aries 2 systems.

Increased MPE RAM

The amount of local Data-RAM / Instruction-RAM for each MPE in Aries 3 has been increased significantly, to 26KBytes / 20KBytes for MPE0, 16KBytes / 16KBytes for MPE1 and MPE2, and 20KBytes / 20KBytes for MPE3. Both MPE0 and MPE3 have dcache / icache support for up to 16KBytes / 16KBytes.

Increased MPE Speed

The MPEs in Aries 3 run at either 108 MHz, for high-performance applications, or at 54 MHz for backward compatibility with existing Aries 2 applications.

Integrated SIO Ports

Two SIO ports have been added for communication with micro-controllers that do not have I2C. These ports can optionally be enabled on certain GPIO pins.

Additional I2S Audio Output Pair

Aries 3 integrates an additional pair of I2S audio outputs, which supports down-mixed audio at the same time as 5.1 audio output, or any other combination of 8 channels.

Enhanced SPDIF and Deeper Audio Buffers

Aries 3 SPDIF can run at a half or a quarter of the I2S sample rate, using its own DMA buffer. Also the audio output buffers have been increased in size to reduce the demands placed on the dma subsystem.

Glueless Interface to External ROM and FLASH

Aries 3 supports a glue-less interface to external ROM, by multiplexing the Boot ROM address and data lines onto the pins used for the system bus. This eliminates the need for the external latches required on Aries 2 systems. Also, Aries 3 supports a glue-less interface to NAND flash memory parts (SmartMedia type).

Other Changes

1. Remove the MPE 0 ROM. The equivalent function can be obtained by using the additional data RAM in MPE 0 to store the tables formerly held in ROM.

2. Several other detailed changes in the audio output hardware are described in the audio section of this document.
3. Provide additional GPIO functions on the unused balls of the BGA package option.

RTL Bug Fixes - The following bugs have been fixed (refer to the Aries 2 bugs list):

4. *The I2C controller cannot send a start code in the middle of a transfer.*
An additional master type has been added to allow this.

Differences between Aries 1 and Aries 2

1. Support System Bus SDRAM in internal mode. Aries 2 supports one or two banks of 54 MHz 16-bit SDRAM in internal mode. This is designed to closely match the performance of the 32-bit EDO DRAM. A wide variety of 16, 64, 128 and 256 Mbit SDRAMs are supported in 2 or 4 bank configurations.
2. Support 4-bank 64 Mbit SDRAM on the Main Bus. This change allows 4 bank SDRAMs to be used on the Main Bus.
3. Allow optional Separation of SPB master and slave. On Aries 1 the Serial Peripheral Bus master and slave were combined inside the chip, and available as a single bus on GPIO3-2. For Aries 2 they may be optionally separated, with the master on GPIO11-10 and the slave on GPIO3-2. This is achieved by setting the slaveAlone bit in the spbSlaveStatus register in the Serial Peripheral Bus controller, and the gp11mode and gp10mode bits in the gpioSpec register in the Miscellaneous IO controller.
4. Remove dead cycles during external mode System Bus ownership. This change improves the System Bus performance in external mode by increasing the internal FIFO size to allow closely packed bursts. The burst-to-burst delay is reduced to one clock cycle; and the bus is relinquished immediately after the end of the last burst of the Other Bus DMA. This will roughly double the DMA transfer rate in external mode.
5. Modify external mode System Bus arbitration to prevent the external host starving for bandwidth. The external host currently only gets a few percent of the bus bandwidth while NUON is doing back-to-back DMA transfers. The intention of this change is to force re-arbitration more frequently, so that the host can maintain enough real-time performance. The mechanism for this is two programmable length loop counters, so that re-arbitration can be forced in the middle of a NUON DMA block. Every time the first counter reaches zero, NUON will retract bus busy at the end of the current burst, wait the second programmed count length, and then re-arbitrate
6. Support external mode System Bus single cycle burst transfers. If SDRAM is used as the external DRAM, 32-bits of data can be read every clock cycle. This allows (in theory, anyway) burst transfer timing of 4-1-1-1 or similar.
7. Host readable version number. The top byte of the host interrupt control register, which reads zero in Aries 1, is now an architecture version number. It reads \$02 in Aries 2.
8. Tri-state on the System Bus DRAM control lines. The DRAM control lines may be tri-stated to support DRAM sharing in internal mode.
9. Expand the SYSCSB address spaces. This allows a split bus architecture to have more than 16MB addressable on the host side by NUON.

10. Modify the CDI. Extra logic has been added to the CDI to support CD-DA mode with certain DVD drives.

RTL Bug Fixes - The following bugs have been fixed (refer to the Aries bugs list):

11. *The I2C slave cannot flag that it is empty to an external master.*
The slave will not acknowledge (NACK) its own address under two conditions: for write if the receive buffer is full, for read if the transmit buffer is empty.
12. *General IO register reads may conflict with Serial Device Bus input data.*
The interaction is now handled properly and no packets are lost.
13. *The debug controller system reset and watchdog functions do not reset the MPEs.*
This reset now occurs correctly.
14. *The host reset function does not work.*
This reset now occurs correctly.
15. *Data transfers can be corrupted in the chip select address spaces.*
The System Bus has been significantly changed for external mode, including fixing this problem
16. *The dataDelay flag for audio out delays by the wrong clock.*
This is corrected, allowing more flexibility for the choice of audio DACs for Aries 2. It has no effect on current systems, as this bit is never set.
17. *Audio register reads may conflict with audio input data.*
The interaction is now handled properly and no packets are lost.
18. *Audio input clock polarity is not programmable in master mode.*
A new control bit allows the capture clock polarity to be programmed independently of the output clock.
19. *Video clock relationship to video data is not defined.*
Already fixed by a metal change in Aries 1.1, this is now reflected in the source RTL.
20. *Sub-picture does not work at some horizontal alignments.*
Again, already fixed by a metal change in Aries 1.1, this is now reflected in the source RTL.
21. *Vertical filtering of progressive MPEG data with downscaling is wrong.*
The filter now operates correctly.
22. *Video Output can be shifted right by 16 pixels*
The DMA will no longer lock out the VDG for too long.
23. *Last Block Element Truncation “White Dots Bug”*
The truncation error in the BDU is fixed.

Compatibility with future NUON Architectures

It is beyond the scope of this document to discuss future generations of the NUON Architecture. However, we want to set some guidelines for you to follow, so that your software will be compatible with small variations of the architecture, and will be either compatible with or easily ported to major new versions of the architecture.

The golden rules are:

1. Do not address peripheral hardware directly. This includes the video input and output channels, the audio input and output channels, the joystick interface, the coded data interface, and any other external devices. All of these may change, and you must always use VM Labs supplied drivers.
2. Do not assume the speed of operations is fixed. Future versions of the system may well be clocked faster, and have faster memory interfaces. You should also note that memory devices attached to NUON will also vary in speed in production. Different SDRAM devices may vary in performance by a few percent, and System Bus memory may be significantly slower in some applications.
3. Try to avoid using undocumented hardware behavior. Do not assume hardware register bits that currently read zero will always do so, or that you can get away with writing non zero values to unused locations. Don't make assumptions about minimum or maximum response times, e.g. for DMA transfers. If something is not clear, ask our technical support staff to clarify it for you.

Conventions

MMA refers to the NUON Multi-Media Architecture

MMP refers to a NUON Multi-Media Processor device, which may be used in NUON compatible systems, and possibly incompatible systems too.

Data elements

Notation

Byte	8 bit value	+	Logical OR
Word	16 bit value	.	Logical AND
Scalar / Long	32 bit value	/	Logical NOT
Half-vector	64 bit value (8 bytes)	\$xx	Hexadecimal value xx
Vector	128 bit value (16 bytes)	%bbbb	Binary value bbbb

The NUON chip is a big-endian device, in the style of the Motorola 68000 family. Strictly speaking this is big-endian byte, word, long, and so on ordering, but little-endian bit ordering. This implies that bit 31 is the most significant bit in a long, and byte 0 is the most significant byte in a long. Compare this to the more sensible Intel style, where the highest numbered bit or byte is always the most significant; or to the Power PC style, where the lowest number is always most significant.

MEDIA PROCESSOR ELEMENTS

Overview

The NUON Media Processor Elements (MPEs) are each fully independent, variable-length very-long-instruction-word processors. This diagram gives an overview of the MPE internal architecture.

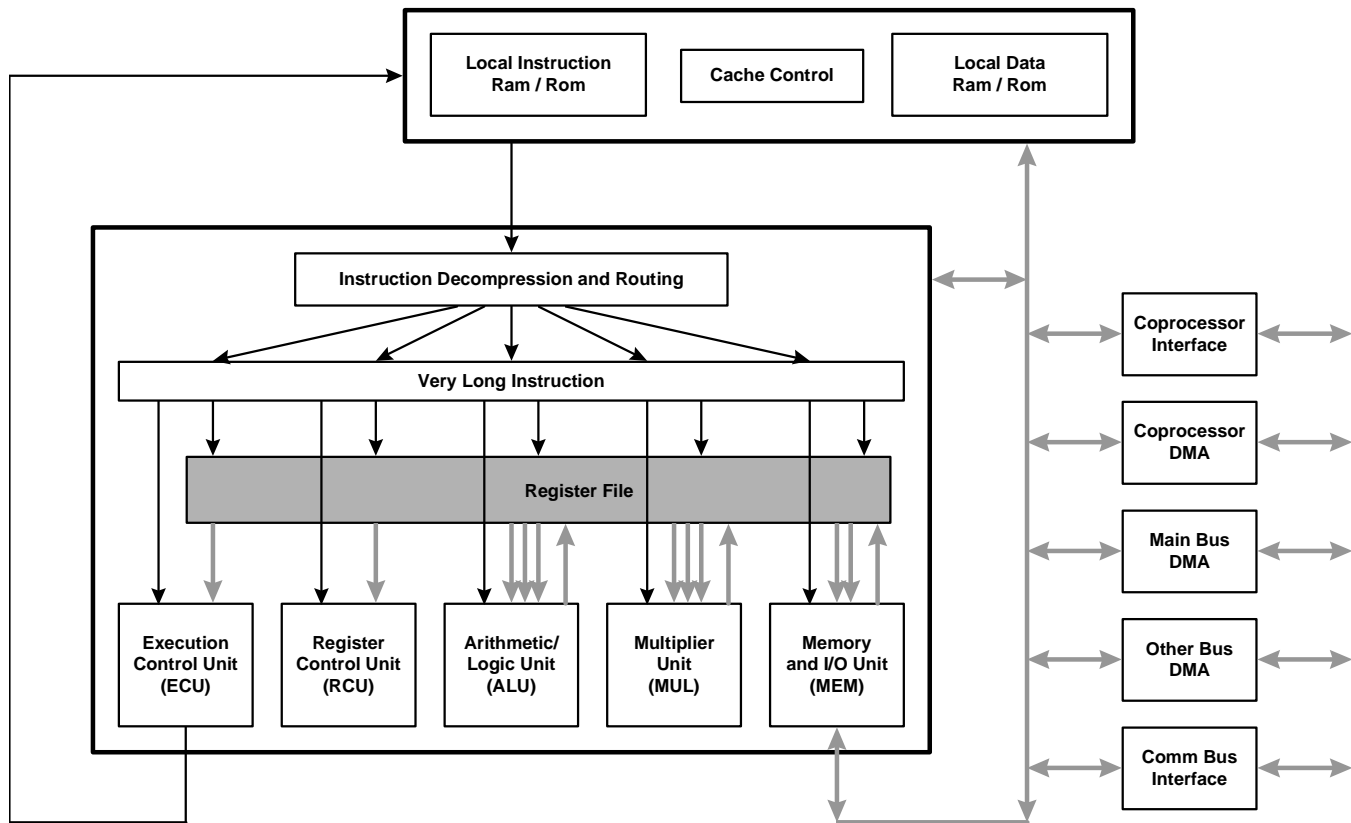


Figure 2 – MPE Internal Architecture

Each MPE has five distinct function units:

ECU – Execution Control Unit

RCU – Register Unit

ALU – Arithmetic Logic Unit

MUL – Multiply Unit

MEM – Memory Unit

The VLIW architecture allows all five function units to operate in parallel, without the complex dynamic instruction scheduling hardware of super-scalar processors. The scheduling is already given in the instruction stream. In essence, the scheduling task is moved from the hardware to the programmer and optimizing tools.

Instructions are encoded into *instruction packets*. Each instruction packet contains from one to five instructions, each for a different function unit. A packet may contain instructions for any combination of

function units. Instruction packets are therefore of variable length, from a minimum of sixteen bits to a maximum of one hundred and twenty-eight bits (see below for restrictions on large packets).

For example, the following set of instructions comprises one instruction packet and will be issued to the function units in one clock cycle:

```
{
    add    r1,r2,r3      ; ALU operation
    mul    r4,r5         ; Multiply operation
    ld_s   (r6),r7       ; Memory operation
    addr   #4,rx         ; RCU operation
    dec    rc0           ; RCU operation
    dec    rc1           ; RCU operation
    bra    eq,loop       ; Execution Control Unit operation
}
```

The RCU decrement instructions may be encoded in parallel with any other RCU operation, allowing it to execute three instructions per clock cycle.

Each MPE can therefore perform 7 independently programmable instructions in every clock cycle, for a very theoretical maximum execution rate of 378 million instructions per second at 54 MHz.

Typically, the inner loops of performance critical applications will be tuned to use as many function units as possible in every clock cycle. However this is not always possible in the outer loops or in general code streams, and in outer loop code most instruction packets may contain no more than one or two instructions and so will be as compact as more standard microprocessor code.

The NUON assembler, compiler and optimizer tools help automate this packing, in addition to optimizing register usage and critical paths.

MPEs generally execute code from on-chip instruction RAM or ROM, and access local data RAM and ROM, which are on a separate bus to the program memory and so may be accessed in parallel. MPEs 1 to 2 cannot execute code from off-chip memory, but MPE 0 and 3 contain instruction and data caches, and can execute code and access data from any memory space on the Main Bus or Other Bus.

MPEs are fundamentally big-endian, although this has little meaning, since we refer to more abstract data type objects in general, rather than bytes or words. The big-endian style is the perverse form of the Motorola 68000 family, i.e. big-endian byte ordering, and little-endian bit ordering.

Memory maps

Each MPE has an identical view of its memory map. This allows MPE code to run on any MPE in the system.

From the point of view of the Main Bus and Other Bus memory map, this corresponds to each MPE thinking that internally it is MPE 0.

Label	Address	Maximum Size	Description
dtrom	\$20000000 - \$200FFFFFFF	1 MByte	Data ROM
dtram	\$20100000 - \$201FFFFFFF	1 MByte	Data RAM
itrom	\$20200000 - \$202FFFFFFF	1 MByte	Instruction ROM
iram	\$20300000 - \$203FFFFFFF	1 MByte	Instruction RAM
dtags	\$20400000 - \$2047FFFFF	0.5 Mbyte	Data tag RAM
itags	\$20480000 - \$204FFFFFFF	0.5 Mbyte	Instruction tag RAM
ctlreg	\$20500000 - \$205FFFFFFF	1 MByte	Control register space
	\$20600000 - \$207FFFFFFF	2 Mbyte	reserved

MPE Memory Sizes

In Aries 1 and 2, the MPE program and data RAM sizes are as follows:

	Data RAM	Program RAM
MPE 0	8 Kbytes	8 Kbytes
MPE 1	4 Kbytes	4 Kbytes
MPE 2	4 Kbytes	4 Kbytes
MPE 3	4 Kbytes	4 Kbytes

In Aries 3, the MPE program and data RAM sizes are as follows:

	Data RAM	Program RAM
MPE 0	26 Kbytes	20 Kbytes
MPE 1	16 Kbytes	16 Kbytes
MPE 2	16 Kbytes	16 Kbytes
MPE 3	20 Kbytes	20 Kbytes

In addition to these, in Aries 1 and 2 MPE 0 also contains 16 Kbytes of data ROM. This function is replaced by additional data RAM in Aries 3.

MPEs 0 and 3 contain cache tag memory as follows:

	Data Tag RAM	Program Tag RAM
MPE 0	1024 bytes	1024 bytes
MPE 3	512 bytes	512 bytes

MPE 0 Local ROM memory map

Address	Comments
0x20000000	Recip LUT
0x20000200	Sine LUT
0x20000604	RSqrt LUT
0x20000940	AC3 Tables
0x20002ff0	MPEG Audio Tables

Instruction and Data Cache

MPE 0 and MPE 3 both have the ability to directly access data and code outside their local space through both a data and instruction cache mechanism. The intention behind the cache is to allow one MPE to be designated as the C-language “main” processor, and the others to be considered co-processors to it, although other uses are clearly possible.

MPE 0 will have better performance due to its larger RAM sizes, but sometimes it may be necessary to use MPE 0 as a co-processor (e.g. when decoding a compressed audio stream), in which case MPE3 may be used as the cached processor.

Cache Setup

Both the instruction and data caches are configured by a control register with the following fields:

- cWayAssoc** This gives the number of cache ways, for multi-way set associative caching. Values of 1-8 way set associative may be set. Pseudo multi-way set associative caching is achieved in the MPE with what is effectively a direct-mapped cache hardware by searching each of the ways in turn. One clock cycle per way is required to search, with the first way to be searched being the last one on which a hit was made.
- cWaySize** This gives the size of each cache way, so the total memory used by the cache is the product of this and the number of ways. Allowable way sizes are 1024, 2048, 4096 or 8192 bytes. This means that not all of the instruction or data memory needs to be assigned to the cache, allowing a mixture of resident and cached instructions or data to be present in the MPE. The cache will use MPE memory starting from the lowest address.
- cBlockSize** This gives the size of the block fetched on a cache miss. The block size may be set to 16, 32, 64 or 128 bytes. The optimum block size is dependent on the program being executed, and the bus latencies when it is executing, but generally the optimum is in the middle of this set, at either 32 or 64 bytes.

Cache initialization

Before the cache can be used the tag RAM must be cleared to zero, to mark all the cache lines as invalid. Once this has been done, and the cache control registers set up, then the cache may be directly used.

Tag RAMs

The tag RAMs are 32-bit memories that may be used for data storage when the cache is not in use. As they are only 32-bit, they may not be used for vector loads and stores, or for pixels larger than 32-bit.

As tags, the entry format is:

Bit	Function
31-4	This is the upper part of the address of the cached entry. When read out for compares, it will be properly masked based on the way size. When read out for use as the write back address a different masking is done based on the block size.
3-2	Unused
1	This flags that the entry is dirty, and only applies to the data cache. This means that the block data will be written out to main memory before the cache block can be re-used.
0	This flags that the cache entry is valid.

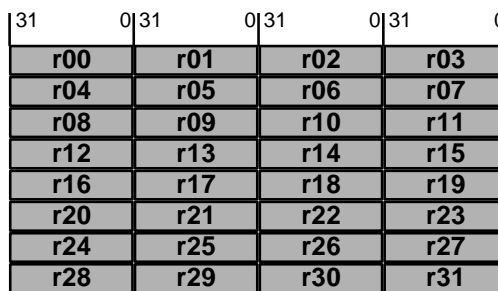
Register File

Each MPE has a general-purpose register file that may be accessed in different modes depending on the instruction.

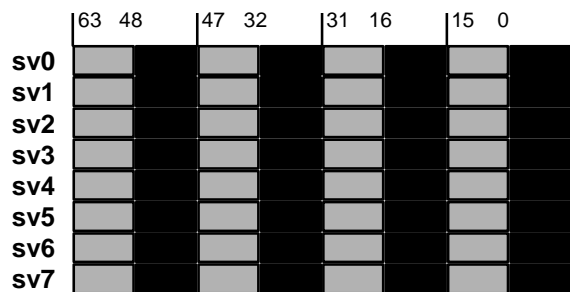
1024 Storage Elements
Simultaneously accessible in different modes (instruction dependent)



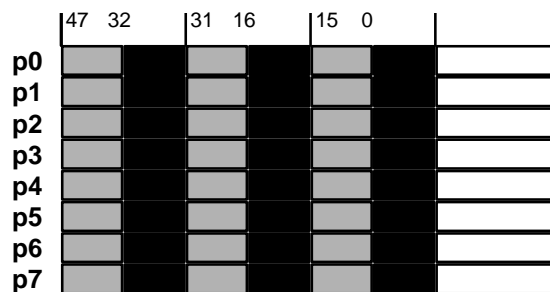
Accessed as Vector Registers
(8 Vectors, each 128 bits)



Accessed as Scalar Registers
(32 Scalars, each 32 bits)



Accessed as Small-Vector Registers
(8 Small-Vectors, each 4x16 bits)



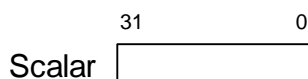
Accessed as Pixel Registers
(8 Pixels, each 3x16 bits)

MPEs have been optimized for certain specific data types, while remaining as general as possible (in the belief that we cannot foresee all future applications). In particular, MPEs are efficient at all types of graphics operations, including geometry and rendering.

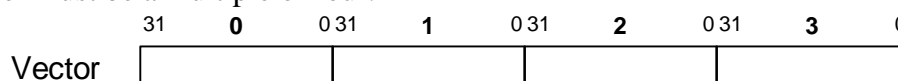
For geometry, we typically use signed 32-bit numbers. MPEs have 32-bit scalar registers, and vector registers which hold four 32-bit scalars. For rendering, we require three color elements (such as Red, Green, Blue, or Y, U and V). Intermediate results often require several fraction bits, and need to be signed. MPEs can reuse vector registers to efficiently store 16 bit signed pixel data. Each vector register holds one pixel in this orientation.

For the remainder of this document, we will use the following syntax for data types, and their register representation:

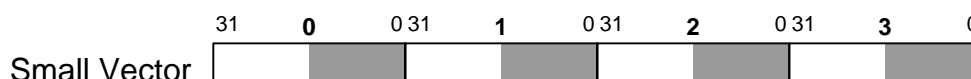
- Scalars, which are a 32-bit signed number, with arbitrary binary point position. Any 32-bit register can hold a Scalar.



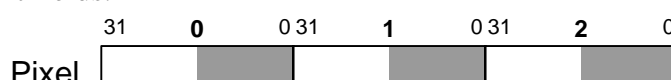
- Vectors, which are a group of four scalars in four consecutive registers, where the first register number must be a multiple of four.



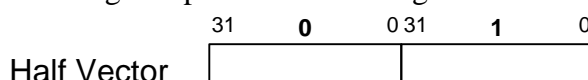
- Small vectors, which are like vectors, except that operations with them are only on the 16 most significant bits of each register. When a small vector is written, the 16 LSBs are usually set to zero.



- Pixels, which are like small vectors, except that they only use registers 0-2. Any instruction that writes a pixel will clear the low 16 bits of the three scalars to zero, and leaves the fourth scalar unchanged. A pixel with an associated Z value is actually a (small) vector, as it has four significant fields.



- Half Vectors, which are two 32-bit Scalars. Used for the butterfly instructions to represent a scalar register pair on an even register boundary



Instruction Flow

The MPE units have a three to four stage pipeline. Instruction packets are dispatched into this pipeline every clock cycle, with a few exceptions, which are described below. These pipeline stages are:

1. Instruction route and decode
2. Instruction fetch
3. Instruction operand fetch, execute, and write-back

In addition to these three cycles, some classes of operation take longer:

- Multiply operations take two cycles to complete, this includes scalar multiply, small vector multiply, and the dot product instructions.
- Load operations from data RAM do not write back the loaded data until the end of the clock cycle which follows the execute cycle.

The main effect of this pipeline is that the two instruction packets after a jump, branch or return from subroutine instruction are always executed, whether the branch is taken or not. This is referred to as the instructions in the *delay slots*. In the cycle after those two packets, either the branch target or the next instruction is executed. This means that no cycles are wasted, if you can find something to do in the delay slots. A special form of the **jmp** instruction forces two empty delay slots, if the jump is taken.

- The pipeline will stall under the following conditions:
- When in single-step mode, the MPE will stall after every instruction.
- When an exception occurs. This is a debug condition that halts the MPE and interrupts the debug control module.

- When a DMA data transfer operation conflicts with the MEM unit instruction about to be executed.

Instruction Packet Restrictions

The following instruction combinations are not allowed, and should be flagged as an error by the assembler:

1. A memory load instruction may not be followed in the next instruction packet by any of the following instruction types:
 - ◆ a MEM unit register to register move
 - ◆ a move immediate

Memory loads complete in two clock cycles, whereas register to register moves complete in one, so there is a conflict for the register write port.
2. A multiply unit **mul**, **mul_sv**, **mul_p** or **dotp** instruction may not be followed in the next instruction packet by an **addm** or **subm** MUL unit arithmetic operation. Multiplies complete in two clock cycles, whereas the arithmetic operations complete in one, so there is a conflict for the register write port.
3. Any instruction in the packet after a memory load or a two-cycle multiply unit instruction (**mul**, **mul_sv**, **mul_p** or **dotp**), must not reference the target register. This is because if an interrupt or pipeline stall occurs between the two instructions, then the two cycle instruction will have completed and the new data will be present; but if no pause occurs then the old data will still be present.
4. A single register port is shared by the following instructions; only one of these instructions may be present in a given instruction packet.


```
ECU      jmp/jsr cc,(Si) | cc,(Si),nop
RCU      mvr/addr Si,RI
ALU      and/or/eor/ftst Si,>>Sj,Sk | Si,<>Sj,Sk
MUL      mul Si,Sk,>>Sq,Sk | #n,Sk,>>Sq,Sk
```
5. Any instruction combination that would imply two simultaneous writes to the same register in the register file, including scalar and vector registers that overlap, must not occur. Register writes can be performed by the ALU, MUL and MEM units. For example, an ALU instruction and a register-to-register move must not target the same register if they are in the same packet, and a load instruction must not be followed by an ALU instruction in the next packet would write to the same register. This also applies to scalar and vector registers that are physically the same register.
6. No instruction packet may span more than two consecutive aligned 64-bit blocks. This puts an upper limit of 80 bits on arbitrarily aligned packets, and an absolute upper limit of 128 bits on any instruction packet. A special pad instruction form exists for padding out packets larger than 80 bits to put the next one on any desired boundary.

MPE FUNCTION UNITS

Arithmetic Logic Unit (ALU)

Overview

The Arithmetic Logic Unit essentially consists of a 32-bit Arithmetic Operation Unit (AOU), with a variety of pre-processing options on the source data.

The source data may include immediate data, scalar registers, vector registers or pixel registers.

ALU Instruction set summary

The ALU instruction set includes both 16 and 32-bit instruction forms. Refer to the Instruction Set Reference starting on page 60 for more details.

Mnemonic	Description
abs	Convert the signed integer to its unsigned absolute value
add	Arithmetic addition
add_sv	Add small vector
and	32-bit logical AND of A and B
as	Arithmetic shift
asl	Arithmetic shift left (also used for lsl)
asr	Arithmetic shift right
bclr	Clear a bit in a register
bits	Bit field extraction
bset	Set a bit in a register
btst	Test a bit in a register
butt	Butterfly operation (sum and difference) of two scalar values
cmp	Arithmetic compare
copy	Register to register move through the ALU
eor	32-bit logical EOR of A and B
ftst	Test a bit field
ls	Logical shift
lsr	Logical shift right
msb	Find the MSB function of the input value.
neg	Arithmetic complement
nop	Null operation
not	Logical complement
or	32-bit logical OR of A and B
sat	Arithmetic saturation
sub	Arithmetic subtraction
sub_sv	Subtract small vectors

A register port is shared by the ALU, for 3 register operand instructions; by the RCU, for **addr** instruction with a register operand; and by the ECU for **jmp** instructions with the jump address held in a register. Only one of these instruction forms may be present in any instruction packet.

Shift

The ALU shifter can perform rotation or arithmetic shifts in either direction, of up to 32 bits of input data. The ALU shifter should not be confused with the bit-extraction units in the multiplier.

Arithmetic shifts to the right maintain the sign of the original value. Arithmetic shifts to the left shift in zeros.

In the instruction set summaries, an arithmetic shift is denoted by the \gg symbol. Rotations are denoted by the $\langle \rangle$ symbol. Positive values are considered to be right shifts or rotates. Negative values are left shifts or rotates.

Sign Extend

Extends the sign of the input data to a 32-bit two's complement number. In the case of positive numbers, the most significant bits are filled with zeros. In the case of negative numbers, they are filled with ones.

MSB

This unit extracts the number of significant bits of a signed number. The result is in the range 0 to 31. Refer to the MSB instruction description for more details.

Flags

The ALU has the following flags:

Name	Description
z	Zero Flag. Set if the result of a scalar arithmetic operation was zero.
c	Carry / Borrow Flag. Set if there is a carry from an addition or a borrow from a subtraction.
v	Overflow Flag. Set if the sign of the result of a scalar add or subtract operation was incorrect. This is signed arithmetic overflow.
n	Negative Flag. Set if the result of a scalar arithmetic operation was negative.

Refer to the individual ALU instruction definitions for their exact effect on flags. The flags are valid in the clock cycle after an arithmetic unit instruction.

Multiply Unit (MUL)

The multiplier can perform two fundamental operations.

1. 32x32 signed multiply, with a sign extended 32-bit result extracted by an appropriate arithmetic shift,
2. Four independent 16x16 signed multiplies, with four 32-bit results with a limited range of shift options.

All multiplies are signed.

The shift operation necessary to extract these results is controlled by either the **acshift** and **svshift** registers, or by the operands of the **mul**, **mul_sv**, **mul_p** and **dotp** instructions.

All multiply operations take two clock cycles to complete. However, you cannot rely on the destination register containing the old value in the clock cycle which follows the multiply instruction.

Arithmetic Operations

The MUL unit can also be used as a simple ALU with a limited range of functions. It can do scalar addition and subtraction. These may be used to augment the main ALU in functions that are limited by the ALU.

The **addm** and **subm** MUL unit arithmetic operations complete in one clock cycle, and may therefore not be used in the clock cycle after a **mul**, **mul_sv**, **mul_p** or **dotp**.

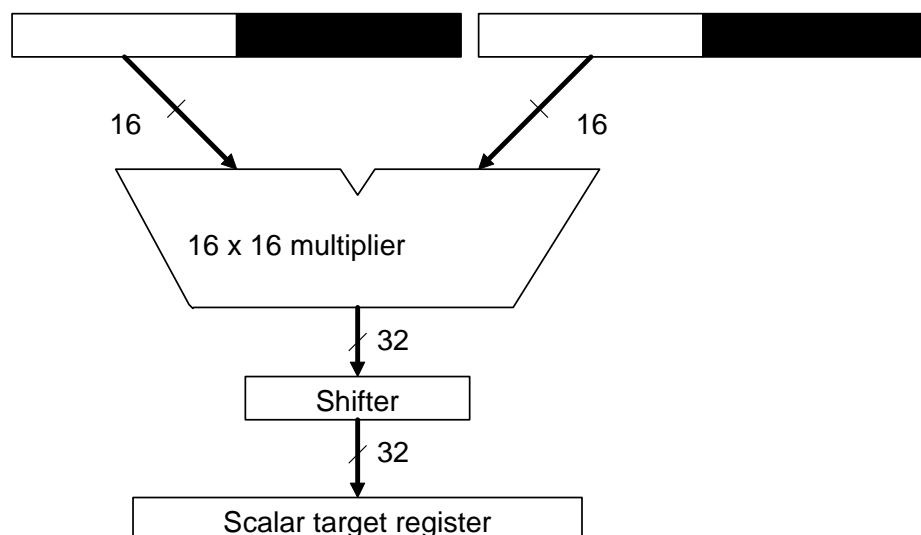
MUL Instruction set summary

Mnemonic	Description
mul	Multiply two (32-bit) scalars
mul_p	Multiply all elements of a pixel
mul_sv	Multiply all elements of a small vector
dotp	Multiply all elements of a small vector, and produce their sum
addm	Arithmetic addition using the MUL unit
subm	Arithmetic subtraction using the MUL unit

Small Vector Shifts

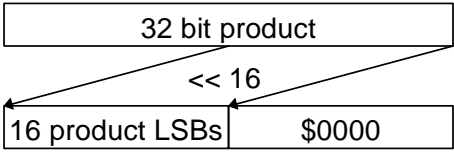
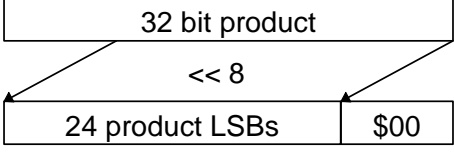
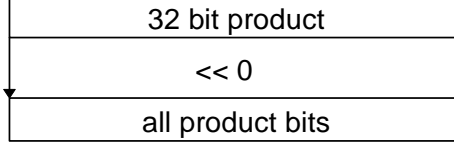
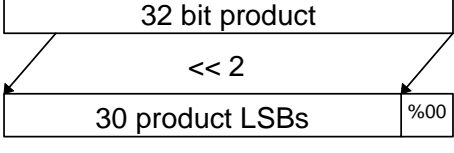
Small vector or pixel multiply results are shifted by one of four values. To understand the shift amounts, you have to understand what the hardware does. For a small vector or pixel multiply, or a dot product, the data flow through the multiplier is something like this:

Scalar source values



The shifter actually performs one of four shift left amounts. However, the programmer's view of these shifts is a shift right, because when the shifter shifts by zero, you can see that there is an effective shift right of 16 through this arrangement, because the *top* 16 bits of the source values are used.

Shift values are therefore encoded like this:

svshift value	Hardware shifts by	Effective shift right	Small vector product definition
0		0	for the product of 16.0 values as a 16.0 small vector value
1		8	for the product of 8.8 values as an 8.8 small vector value (8.16 is actually written)
2		16	for the full 32-bit product
3		14	for the product of 2.14 values as a 2.14 small vector value (2.28 is actually written)

Execution Control Unit (ECU)

Overview

The ECU is responsible for controlling the program counter and execution pipeline. By default, the ECU will advance the program counter after every instruction packet to the start of the next packet, while monitoring exception and interrupt conditions. In addition, a number of instructions are available which directly control the ECU.

ECU instructions execute in parallel with all the other function units. For example, an RTS instruction may coexist with a POP instruction from the memory unit.

A register port is shared by the ALU, for 3 register operand instructions; by the RCU, for **addr** instruction with a register operand; and by the ECU for **jmp** instructions with the jump address held in a register. Only one of these instruction forms may be present in any instruction packet.

ECU Instruction set summary

Mnemonic	Description
bra	branch conditionally; target address is a relative offset to the current instruction packet address
jmp	jump conditionally; target address is an absolute value
jsr	jump conditionally to subroutine; target address is an absolute value
rts	jump conditionally to absolute address in register rz

Branch optimization

The two instruction packets following any ECU instruction are usually executed whether the instruction flow is changed or not. These two instruction packets are known as the delay slots. After the delay slots, execution either continues or branches to a new address.

The only exception to this rule is a special form of the jump instruction with a **nop** operand. When a jump instruction has a **nop** operand, the processor is idle in the two clock cycles that follow if the jump is taken (these are known as dead cycles); if it is not taken then execution always continues normally. These dead cycles form may be used to save code space if there is no useful function which can be performed when the jump is taken.

ECU instructions may follow each other in successive instruction packets. If the first one causes the program counter to change, i.e. the **bra**, **jmp** or **rts** is taken, then any ECU instructions in the two instruction packets that follow it will be ignored. If the first ECU instruction is not taken, then an ECU instruction that follows it will be evaluated normally. ECU instructions may follow each other repeatedly in this manner.

Relative branches are calculated from the address of the instruction packet which follows the packet containing the branch instruction.

The register **rz** is used for sub-routine calls. The **jsr** instruction copies the correct return address to **rz** register. This is correct both for normal branches, where it is the address of the packet of instructions three packets on from the current one, that is after the delay slot packets; and for the special **jsr** form with implied **nop**, when it is the address of the packet that follows the current one.

The implied **nop** form is therefore more efficient if the branch is not taken, because execution continues immediately with code for that path, and it may be useful when the branch is not normally taken to use this form when the delay slot cannot be filled.

Condition codes

Condition code flags are generally set at the end of the current instruction, and remain valid until updated. The point at which the flag may be tested varies with the type, as follows:

1. The ALU condition codes may be tested by a branch instruction in the cycle after they are generated, i.e. at the same time that the ALU result may be used.
2. The multiplier **mv** flag may be tested two cycles after the multiplier instruction, again at the same time that the result is valid. In the cycle in between the flag state is not defined.
3. The counter flags, **c0z** and **c1z** may be tested by a branch instruction in the cycle after they are decremented.

The following condition code flags are defined:

Flag	Name	Description
z	ALU Zero	Set if the result of a scalar ALU operation is zero
c	ALU Carry / Borrow	Set if there is a carry from an addition or a borrow from a subtraction.
n	ALU Negative	Set if bit 31 of a scalar ALU operation is set
v	ALU Overflow	Set if there is an overflow from a scalar ALU operation
mv	MUL Overflow	Set if any significant bits are lost as a result of the shift in a scalar multiply instruction
c0z	Counter rc0 zero	Set if counter register rc0 is zero
c1z	Counter rc1 zero	Set if counter register rc1 is zero

Flag	Name	Description
modge	RCU range high	Set if a modulo or range instruction found the value greater than or equal to the range specified.
modmi	RCU range low	Set if a modulo or range instruction found the value less than zero.
cf0	Coprocessor 0	Used for expansion hardware
cf1	Coprocessor 1	Used for expansion hardware

Note that there are no flags for the **rx**, **ry**, **ru**, and **rv** registers. There is also no overflow detection for the small vector accumulator multiplies. There are no flags for pixels, vectors or small vectors.

Branch instructions can test combinations of these flags. A full list of available tests is given below.

Mnemonic	Condition	Test
ne	Not equal	/z
eq	Equal	z
lt	Less than	(n./v) + (/n.v)
le	Less than or equal	z + (n./v) + (/n.v)
gt	Greater than	(n.v./z) + (/n.v./z)
ge	Greater than or equal	(n.v) + (/n./v)
c0ne	rc0 not equal to zero	/c0z
c1ne	rc1 not equal to zero	/c1z
c0eq	rc0 equal to zero	c0z
c1eq	rc1 equal to zero	c1z
cc (hs)	Carry clear (High or same)	/c
cs (lo)	Carry set (Low)	c
vc	Overflow clear	/v
vs	Overflow set	v
mvc	Multiply overflow clear	/mv
mvs	Multiply overflow set	mv
hi	High	/c./z
ls	Low or same	c + z
pl	Plus	/n
mi	Minus	n
t	True	1
modmi	modulo RI was < zero	modmi
modpl	modulo RI was >= zero	/modmi
modge	modulo RI was >= range	modge
modlt	modulo RI was < range	/modge
cf0lo	Coprocessor flag 0 low	/cf0
cf0hi	Coprocessor flag 0 high	cf0
cf1lo	Coprocessor flag 1 low	/cf1
cf1hi	Coprocessor flag 1 high	cf1

Note that the 16-bit form of the branch instruction can only use the first eight conditions from the condition code table (**ne**, **eq**, **lt**, **le**, **gt**, **ge**, **c0ne**, **c1ne**).

Subroutines and Interrupts

The MPE has some very simple instructions and registers to assist subroutine calling. The guiding philosophy for subroutines has been to make calls and returns extremely fast, and to efficiently reduce overall code size.

Interrupts must be responded to within certain maximum time limits in a real-time system. Therefore the interrupt philosophy is low-latency, and fast return.

The inventory of instructions (not all of which are ECU instructions) for subroutines and interrupts includes:

Instruction	Description
push ...	Pushes a vector register (128 bits) on a stack in data RAM. Special forms save rz and other hardware state. sp is pre-decremented.
pop ...	Pops a vector register from the stack, the reverse of push. sp is post-incremented.
jsr ...	Copy the address of the first un-executed instruction packet into the special register rz and transfer control to the specified address
rts cc	Conditionally jump to register rz indirect.
rti cc	Conditionally return from interrupt by using the rzi registers to restore the fetch pipe-line.

Subroutines

Simple sub-routine calling may be performed by the **jsr** instruction. The two following instruction packets in the two delay slots will be executed unless the special implied **nop** form of **jsr** is used, and then control will be transferred to the subroutine. At the end of the subroutine the **rts** instruction returns control to the calling code, once again after executing the instruction packets in the delay slots after it.

```
        jsr      subroutine          ; transfer control and set up rz
        nop                      ; delay slot 1
        nop                      ; delay slot 2
; instruction flow will return here
. . .
subroutine:
. . .
rts
        nop                      ; delay slot 1
        nop                      ; delay slot 2
```

If recursion to further levels of sub-routine call is required, then the **rz** register will need to be preserved with the appropriate form of **push** before re-using it to call a subroutine at a deeper level.

Interrupts

Interrupts will cause a transfer of execution control to the interrupt service routine at the earliest possible moment. The interrupt service routine address is defined by one of two **intvec** registers, and is a location in physical memory.

Two level of interrupt are generated, level 1 which is used for the majority of interrupt servicing, and level 2 which allows one interrupt source to be selected as a higher priority interrupt. A level 2 interrupt can interrupt the interrupt service routine of a level 1 interrupt.

When an interrupt is recognized, the execution control unit performs an implied branch to the interrupt service routine address, and copies program address return information to the appropriate interrupt registers **rzi1** or **rzi2**.

On entry to the interrupt service routine, all interrupts at that level are masked by corresponding the **imaskHw** hardware interrupt mask bit, which is set when the interrupt is recognized. It is cleared again by the **rti** return from interrupt, and may be left set throughout the interrupt service routine if desired.

If software masking of interrupts is required, the appropriate **imaskSw** bit may be set to mask all interrupts.

Individual interrupt sources may be independently enabled in the MPE interrupt control register, by setting the appropriate interrupt enable bits. When an interrupt occurs, this register gives the source of the interrupt, and can be used to clear the interrupt hardware.

A non-masked interrupt to the MPE effectively causes a forced branch to the interrupt handler located at the **intvec** address stored in the interrupt vector register. As with a normal branch, there are 2 “delay slots” before the first instruction of the interrupt routine actually arrives in the MPE execute stage.

How Interrupts work

When a level-1 interrupt is true, enabled, and not masked, the ECU saves **pcroute** into **rzi1**, sets **imaskHw1** high, forces a jump to **intvec1**, and kills the execution of the packets that were in the route and fetch stages of the pipeline. When a level-2 interrupt is true, enabled, and not masked, the ECU saves **pcroute** into **rzi2**, sets **imaskHw2** high, forces a jump to **intvec2**, and kills the execution of the packets that were in the route and fetch stages of the pipeline.

Both level-1 and level-2 interrupts are temporarily blocked during the execution of any “taken-jump” instruction and its first delay slot. This includes taken **bra**, **jmp**, **jts**, **rts**, and **rti** instructions, as well as the “interrupt-jumps” themselves. If both level-1 and level-2 interrupts are true, enabled, and not masked, then the level-1 interrupt is temporarily blocked while the level-2 interrupt jump is taken.

Assuming the MPE is not already handling a level-2 interrupt, there is a maximum latency of 5 ticks from the time a level-2 interrupt is captured until the MPE is executing the first instruction packet of the level-2 interrupt service routine (ignoring stalls caused by DMA). Like any other ECU jump, the interrupt-jump itself takes 3 ticks, and before the interrupt-jump starts, there may be up to 2 ticks in which the level-2 interrupt is being automatically blocked if the ECU is executing a taken-jump.

Software must handle the “clearing” of pulse-style interrupt sources differently from level-style interrupt sources in order to guarantee no lost or spurious interrupts. For pulse-style interrupts, first the local MPE **IntSrc** register bit is cleared, and then the source logic can be informed that the last interrupt has been handled and another may now be issued. For level-style interrupts, first the source logic is informed that the interrupt has been handled so that it can remove its interrupt (or keep it asserted if it has another interrupt), and then the local MPE **IntSrc** register bit can be cleared (which will have no effect if the source logic kept the interrupt asserted).

Register Control Unit (RCU)

The register unit (RCU) is responsible for control of the special purpose registers **rx**, **ry**, **ru**, **rv**, **rz**, **rc0** and **rc1**.

Registers **rx**, **ry**, **ru**, and **rv** are used for bilinear (pixel) address generation, and are normally used as 16.16 fixed point fractions. The **ADDR** instruction allows a scalar or an immediate integer value to be added to one of these registers.

A register port is shared by the ALU, for 3 register operand instructions; by the RCU, for **addr** instruction with a register operand; and by the ECU for **jmp** instructions with the jump address held in a register. Only one of these instruction forms may be present in any instruction packet.

The register unit is responsible for copying the program counter into the register **rz**.

Registers **rc0** and **rc1** are 12-bit programmable down-counters, which stop on zero. The instructions that decrement these are not in fact actual instructions, but are actually encoded as a bit-field in any other RCU instruction, and therefore one or both counters may be decremented in parallel with any other RCU instruction. The **addr #0,ri** form is used as an RCU null operation when only decrement instructions are encoded.

RCU Instruction set summary

Mnemonic	Description
dec	Decrement rc0 or rc1 register, unless it is zero
addr	Add to index register
modulo	Range limit index register
range	Range check index register

Memory Unit (MEM)

The memory unit is responsible for data transfers between internal MPE registers and data memory. These include loads and stores of scalars, small vectors, vectors and pixels. It also supports vector stack operations. By utilizing the same data paths, the memory unit also supports register-to-register transfers.

The memory unit is supplemented by a programmable DMA engine. The DMA engine provides the only way that MPEs can access external memory. Since external bus(???) bandwidth is a precious commodity, the programmer needs to understand the memory unit and DMA function in some detail, in order to be able to configure it effectively for each algorithm.

MEM Instruction set summary

Mnemonic	Description
ld_b	Load Byte
ld_w	Load word
ld_p	Load Pixel
ld_s	Load Scalar
ld_sv	Load Small vector
ld_v	Load Vector
mirror	Reverse the bit order of a scalar
mv_s	Move Scalar
mv_v	Move Vector
pop	Pop data from stack
push	Push data on to stack
st_p	Store Pixel
st_s	Store Scalar
st_sv	Store Small vector
st_v	Store Vector

The move instructions provide a convenient way to transfer data between internal registers.

The remaining load and store instructions provide a variety of addressing modes and data paths for transferring data between registers and memory.

Data can be loaded from memory in a wide variety of forms, including 4, 8, 16, 32, 64 and 128-bit quantities. Memory is physically organized as 32-bit wide RAMs, therefore it is only possible to store in multiples of 32-bit words.

Several addressing modes are available, which can be split into two categories: linear addressing and bilinear addressing. Various forms of linear addressing are available for loading and storing scalars, vectors and small vectors. Please refer to the instruction reference for details of which modes are available to which instructions. Examples are:

<code>ld_s</code>	<code>(Si),Sj</code>	Indirect. Load from address <code>Si</code> , to register <code>Sj</code>
<code>ld_s</code>	<code>(#nn),Sj</code>	Absolute. Load from address <code>#nn</code> in local data RAM or local data ROM
<code>ld_s</code>	<code>(xy / uv),Sj</code>	Bilinear indirect. Form an address from the <code>xy</code> or <code>uv</code> pair, along with their associated base and flags.

The scalar registers `r0-r31`, when used as indirect address pointer, are considered unsigned 32.0 format, i.e. whole numbers of bytes. All addressing forms can only reference internal data RAM or data ROM.

Stack Operations

Stack operations always push or pop 16 bytes of data. This can be a vector register or a specified set of the special purpose registers. The stack pointer is a special purpose register which always points at a 16-byte boundary in RAM. The stack grows down through memory, so a push operation pre-decrements the stack pointer, and a pop operation post-increments it.

Accessing Mpe Control Registers

Since most of the MPE Control Registers are scalars on vector-aligned addresses, they are normally accessed with the direct-absolute-addressing forms of the `ld_s` and `st_s` instructions. The `commrecv` and `commxmit` registers are exceptions since they are each made up of 4 scalars on successive scalar addresses, and they may be accessed as vectors with the `ld_v` and `st_v` instructions. (Access to the Control Registers may also be done with indirect addressing modes, but this normally isn't very useful.)

The table below shows the restrictions on address values and immediate values for the different instruction forms used to access MPE Control Registers. (The set of restrictions is a bit odd, but that's the price we pay for instruction compression and getting the most out of the bits available at each instruction length.) Source code normally specifies `st_s` or `ld_s` along with the Control Register name and the register file register or immediate value, and leaves the assembler to choose the shortest instruction form that can handle the operands.

Length	Instruction	Operand size	Address Range in CTL REG Space (Alignment)
*(16)	<code>ld_s (#base,#offset),Sk</code>	offset[8:4]	\$2050_0000:\$2050_01F0 (Vector-aligned)
*(16)	<code>st_s Sj, (#base,#offset)</code>	offset[8:4]	\$2050_0000:\$2050_01F0 (Vector-aligned)
@(32)	<code>ld_s (#base,#offset),Sk</code>	offset[12:2]	\$2050_0000:\$2050_1FFC (Scalar-aligned)

Length	Instruction	Operand size	Address Range in CTL REG Space (Alignment)
@(32)	st_s Sj, (#base, #offset)	offset[12:2]	\$2050_0000:\$2050_1FFC (Scalar-aligned)
@(32)	ld_v (#base, #offset), Vk	offset[14:4]	\$2050_0000:\$2050_7FF0 (Vector-aligned)
@(32)	st_v Vj, (#base, #offset)	offset[14:4]	\$2050_0000:\$2050_7FF0 (Vector-aligned)
*(32)	st_s #immu, (#base, #offset)	offset[12:4] immu[9:0]	\$2050_0000:\$2050_1FF0 (Vector-aligned)
@(64)	st_s #immu, (#base, #offset)	offset[13:2] immu[31:0]	\$2050_0000:\$2050_3FFC (Scalar-aligned)

@ means the 2-bit base value is encoded in the instruction as one of:

00	DTROM	\$2000_0000
01	DTRAM	\$2010_0000
10	CTLREG	\$2050_0000
01	reserved	

* means the base value is implicitly:

10	CTLREG	\$2050_0000
----	--------	-------------

immu[9:0] is zero-filled to the left to create a scalar.

Bilinear addressing

Bilinear addressing is only available to the load and store instructions that transfer pixel, scalar, and small vector data. It is normally used for pixel input data such as texture maps, and rendered pixel output, but may find uses in other one or two dimensional structures as well. This is an example of a bilinear load instruction:

ld_p (xy), Vi Bilinear indexed load from address formed by the xy pair

Two bilinear register pairs are available – **rx** and **ry**, which are referred to as **(xy)** when used as an operand; and **ru** and **rv**, which are referred to as **(uv)**. Each pair has an associated set of IO registers which define the structure it is referencing. To reference a new data structure, these IO registers will need to be modified.

The index register pairs are normally used in unsigned 16.16 format. When a register pair is used to reference corresponding bit-maps at different resolutions (normally for MIP-mapping), then the position of the binary point may be changed via the **##_MIPMAP** register.

The fractional part of the register will be truncated for address generation, and the upper bits of the integer part may be masked to support low overhead 2^n modular arithmetic (see the TILE values below).

The target address for **(xy)** addressed loads and stores is formed by the following formula.

$$XY_BASE + \left[\begin{array}{c} \text{pixel width} \\ \text{of} \\ XY_TYPE \end{array} \right] \times \left[\begin{array}{l} ((Y \gg XY_MIPMAP) \& (YTILEMASK \gg XY_MIPMAP)) \times (\text{width} \gg XY_MIPMAP) \\ + ((X \gg XY_MIPMAP) \& (XTILEMASK \gg XY_MIPMAP)) \end{array} \right]$$

Normal or bit-reverse X/Y or U/V

XTILEMASK is the mask produced by taking the value \$FFFF0000 << (16-x_tile). The right shift performed on it by **mipmap** is an arithmetic shift. YTILEMASK is calculated similarly.

X and Y are the integer parts (16 MSBs) of **rx** and **ry**.

The other values used in this calculation are defined below.

The bilinear store instruction looks like this (xy is used as an example, and may be replaced by uv):

st_p Vi,(xy) Bilinear Indexed. Store to address formed by the xy pair
Store instructions are subject to the limitation that only 32 and 64 bit pixel types can be stored.

The control values for bilinear addressing are as follows:

Register Label	Width in bits	Description																																																				
x_rev u_rev	1	If this bit is set, the integer part of rx / ru is mirrored prior to address generation. See below.																																																				
y_rev v_rev	1	If this bit is set, the integer part of ry / rv is mirrored prior to address generation. See below.																																																				
xy_chnorm uv_chnorm	1	If set, 128 is subtracted from chrominance fields during ld_p and 128 is added to chrominance fields during st_p when this bilinear pair is used. Sign extension is performed on the ld_p operation.																																																				
xy_type uv_type	3	Defines the data type in internal RAM. Only Pixel data types 1-6 and Small Vector are legal in this field. Some modes are illegal for store pixel. Refer to the MPE Data Types section. <table><tr><th>Type</th><th>Mapping</th><th>Bits</th><th>Note</th></tr><tr><td>0</td><td>MPEG pixel</td><td>24</td><td>see notes on storage format</td></tr><tr><td>1</td><td>Pixel data type 1</td><td>4</td><td>for CLUT lookup</td></tr><tr><td>2</td><td>Pixel data type 2</td><td>16</td><td></td></tr><tr><td>3</td><td>Pixel data type 3</td><td>8</td><td>for CLUT lookup</td></tr><tr><td>4</td><td>Pixel data type 4</td><td>32</td><td></td></tr><tr><td>5</td><td>Pixel data type 5</td><td>32</td><td></td></tr><tr><td>6</td><td>Pixel data type 6</td><td>64</td><td></td></tr><tr><td>8</td><td>Byte</td><td>8</td><td>not valid as a pixel load/store type</td></tr><tr><td>9</td><td>Word</td><td>16</td><td>not valid as a pixel load/store type</td></tr><tr><td>A</td><td>Scalar</td><td>32</td><td>not valid as a pixel load/store type</td></tr><tr><td>C</td><td>Small vector</td><td>64</td><td>not valid as a pixel load/store type</td></tr><tr><td>D</td><td>Vector</td><td>128</td><td>not valid as a pixel load/store type</td></tr></table>	Type	Mapping	Bits	Note	0	MPEG pixel	24	see notes on storage format	1	Pixel data type 1	4	for CLUT lookup	2	Pixel data type 2	16		3	Pixel data type 3	8	for CLUT lookup	4	Pixel data type 4	32		5	Pixel data type 5	32		6	Pixel data type 6	64		8	Byte	8	not valid as a pixel load/store type	9	Word	16	not valid as a pixel load/store type	A	Scalar	32	not valid as a pixel load/store type	C	Small vector	64	not valid as a pixel load/store type	D	Vector	128	not valid as a pixel load/store type
Type	Mapping	Bits	Note																																																			
0	MPEG pixel	24	see notes on storage format																																																			
1	Pixel data type 1	4	for CLUT lookup																																																			
2	Pixel data type 2	16																																																				
3	Pixel data type 3	8	for CLUT lookup																																																			
4	Pixel data type 4	32																																																				
5	Pixel data type 5	32																																																				
6	Pixel data type 6	64																																																				
8	Byte	8	not valid as a pixel load/store type																																																			
9	Word	16	not valid as a pixel load/store type																																																			
A	Scalar	32	not valid as a pixel load/store type																																																			
C	Small vector	64	not valid as a pixel load/store type																																																			
D	Vector	128	not valid as a pixel load/store type																																																			
x_tile u_tile	4	Defines a mask of the upper n bits of the register value before it is used in the address calculation. The most significant x_tile or y_tile bits of the register value are forced to zero, primarily to allow tiling of texture maps. For example, a value of 0 does not mask any bits, and a value of 15 masks Bit 31 to Bit 17.																																																				
y_tile v_tile	4	Defines a mask of the upper n bits of the register value before it is used in the address calculation. The most significant y_tile or v_tile bits of the register value are forced to zero, primarily to allow tiling of texture maps. For example, a value of 0 does not mask any bits, and a value of 15 masks Bit 31 to Bit 17.																																																				
xy_width uv_width	11	The width of the two-dimensional structure, measured in pixels. Legal values are 0, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 and 1024. A value of zero means that the Y index register is not used as part of the address (it is multiplied by zero, effectively).																																																				
xy_mipmap uv_mipmap	3	Defines the position of the binary point, relative to 16.16. This is necessary for MIP Mapping. The position of the binary point is considered to be 16+(##_MIPMAP) by the Memory Unit address generator and the mul_sv and mul_p instructions. Valid values are 0-4.																																																				
xybase uvbase	30	The memory base address of the top left pixel of the entire image. It must be on a scalar boundary.																																																				
rx	32	32-bit X index register. The X part of the address is formed from the integer part of																																																				

Register Label	Width in bits	Description
ru		rx or ru , where the binary point can be moved left from 0-4 positions by the ##_MIPMAP value.
ry rv	32	32-bit Y index register. The Y part of the address is formed from the integer part of ry or rv , where the binary point can be moved left from 0-4 positions by the ##_MIPMAP value.

The IO address and bit-field assignments for these are shown in the MPE IO memory map section of this document.

Range Limiting Index Registers

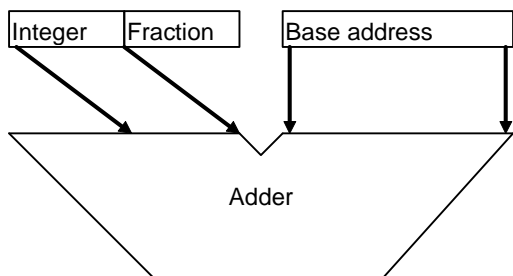
The range registers allow the index register **rx**, **ry**, **ru** and **rv** to have a maximum range defined for them. The **modulo** instruction will take one of the index registers, and if it is greater than corresponding range value will subtract the range; and if less then zero will add the range. If it is out of range by more than the range value, this operation will not work correctly. The **range** instruction performs the same comparison, but only sets the flags.

Register Label	Width in bits	Description
xrange urange	10	Gives the range of rx / ru for the modulo and range instructions.
yrange vrage	10	Gives the range of ry / rv for the modulo and range instructions.

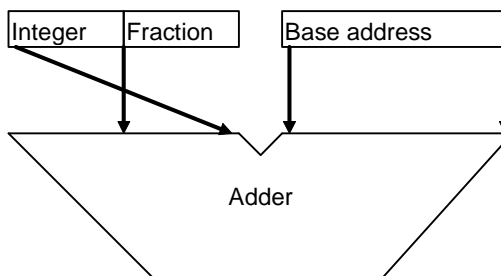
FFT Address Generation

The **xy_xrev** and **uv_xrev** flags are useful for FFT address generation. It reverses the bit order of the integer part of **rx** / **ru** before it is used. This means that the system can support a 2^n sized buffer for FFT operations, with the same effect as the 56000 “reverse carry propagation” in the address. This simplified diagram shows the effect.

Normal indexed address generation



Mirrored indexed address generation



Note that this means that the integer bits used start from the *MSB* of **rx** / **ru**. An increment value of one will require suitable left shifting. For example, for a $2^7 = 128$ sized buffer, the $(16-7) = 9$ low integer bits are not used, so to add one it should be shifted left 9.

Linear Indexed Addressing

It is quite feasible to use the bilinear addressing form as a linear indexed addressing form, although the index will be limited to 64K data elements. The bilinear address form can be used with load and store scalar, and with load and store small vector. For scalar loads and stores, the **##_type** field should be set to type 5; for small vector loads and stores it should be set as shown in the table above.

Normally **ry** will be set to zero in this mode, so the width field is not important.

Either the **x_tile / y_tile / y_tile / v_tile** field or the **modulo** operation can be used to implement circular buffers. The **x_tile / y_tile / y_tile / v_tile** mechanism is the simplest mechanism; it forces addresses to wrap within a 2^n area. The **modulo** operation gives more flexibility to the circular buffer size, at the cost of greater program overhead.

Pixel Data Types

Scalars and vectors are stored in MPE data memory as they are in registers. Scalars in data memory have to be on a long boundary, vectors in data memory have to be on a vector boundary. These can be transferred by linear DMA between MPE data RAM and external DRAM, or between MPEs. Linear DMA is always performed in long-words on long-word boundaries, so the only alignment restriction on vectors in DRAM is to be on long boundaries.

Load pixel operations only affect the first three elements of the target vector. Load small-vector operations affect all four elements. Otherwise their operations are identical, and the actual data type referenced in MPE data RAM is given by the appropriate type field.

The MPE MEM Unit supports a number of data types for loads from Memory into the RegFile, and for stores from the RegFile to Memory. There are different forms of load and store for dealing with the different data types: byte, word, scalar, small-vector, vector, and 7 different pixel types. The supported transfers are summarized in the table below, followed by a brief description of the data transformations performed for each pixel type.

The data type number is used in the **xyctl** and **uvctl** registers to specify the **xy_type** and **uv_type** for bilinear addressing. It is also used in the **linpixctl** register to specify the **linpix_type** for linear addressing with **st_p**, **st_pz**, **ld_p**, and **ld_pz**.

Data Type #	Name	Store Data Size To Memory	Store Form	Load Form	Load Data Size Into Register File
0	pixel	MPEG 16 bits	NA NA	ld_p ld_pz	¾ vector vector
1	pixel	4 bits	NA NA	ld_p ld_pz	¾ vector vector
2	pixel	16 bits	NA NA	ld_p ld_pz	¾ vector vector
3	pixel	8 bits	NA NA	ld_p ld_pz	¾ vector vector
4	pixel	24+8 bits	st_p st_pz	ld_p ld_pz	¾ vector vector
5	pixel	16+16 bits	st_p st_pz	ld_p ld_pz	¾ vector vector
6	pixel	24+8+32 bits	st_p	ld_p	¾ vector

			st_pz	ld_pz	vector
7	reserved				
8	byte	8 bits	NA	ld_b	scalar (msb aligned) (byte,24'b0)
9	word	16 bits	NA	ld_w	scalar (msb aligned) (word,16'b0)
A	scalar	32 bits	st_s	ld_s	scalar
B	reserved				
C	small-vector	64 bits	st_sv	ld_sv	sm-vector (msb aligned) (word,16'b0,word,16'b0, word,16'b0,word,16'b0)
D	vector	128 bits	st_v	ld_v	vector
E	reserved				
F	reserved				

Data Type 0 – MPEG pixels

The format for this pixel type is derived from the format used to store decoded MPEG video, and is meant to allow efficient use of MPEG video as a texture. The main-bus DMA is used to move the data from external SDRAM into local MPE DTRAM. This is accomplished by at least two separate DMA operations, one for the luma data and one for the chroma data. Details of these DMA operations are described in the Main Bus section. Once the DMA transfers are complete, the layout of the Type 0 pixel data within a DTRAM vector is pictured here.

Long Word offset	0				1			
Byte offset	0	1	2	3	4	5	6	7
Bit address	127-120	119-112	111-104	103-96	95-88	87-80	79-72	71-64
	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7

Long Word offset	2				3			
Byte offset	8	9	A	B	C	D	E	F
Bit address	63-56	55-48	47-40	39-32	31-24	23-16	15-8	7-0
	CR0	CR1	CR2	CR3	CB0	CB1	CB2	CB3

The upper bits of the address formed by a **ld_p** or **ld_pz** instruction are used to access the vector pictured above, and then bits [3:1] are used to select one of the following 8 pixels:

Y0,CR0,CB0
Y1,CR0,CB0
Y2,CR1,CB1
Y3,CR1,CB1
Y4,CR2,CB2
Y5,CR2,CB2
Y6,CR3,CB3
Y7,CR3,CB3

The load pixel instruction will map a type 0 pixel into a vector register as follows:

	Bits	31-30	29-22	21-16	15-0
Register Vn[0] - Y		0	Yx	0	0
Register Vn[1] - Cr		SS	CRy	0	0
Register Vn[2] - Cb		SS	CBy	0	0
Register Vn[3]		left unchanged			

The load and store pixel with Z (**ld_pz** and **st_pz**) instructions will map a type 0 pixel into a vector register as follows:

	Bits	31-30	29-22	21-16	15-0
Register Vn[0] - Y		0	Yx	0	0
Register Vn[1] - Cr		SS	CRy	0	0
Register Vn[2] - Cb		SS	CBy	0	0
Register Vn[3] - control		0			

For loads, if the appropriate **chnorm** bit is set, then 128 is subtracted from the values placed in Vn[1] and Vn[2]. Since these are assumed to be signed 2.14 numbers, the value %100000 00000000 is actually subtracted from bits 29-16. When this is done, sign extension is performed into bits 30-31 (SS in the table above).

Data Type 1 – 4 bit pixels

Type 1 pixels are four bits. The value represents an index into an arbitrary look-up table, and so it has no fixed relationship with the physical appearance. These are sometimes useful for memory efficient texture maps, and can be used for very memory efficient display buffers. They can be used by load pixel instructions, but may not be directly stored.

Type 1 pixels are always stored together in groups of four in a 16-bit word. This represents a horizontal strip of four pixels. All DMA operations on type 1 pixels must be defined with their X position and length as multiples of four pixels.

Byte address	0		1	
Bit address	15-8		7-0	
	0	1	2	3

The load pixel instruction will load a 4-bit pixel into a vector register as follows:

	Bits	31-6	5-2	1-0
Register Vn[0]		CLUTBASE(31:6)	P[3:0]	0
Register Vn[1]		0	0	0
Register Vn[2]		0	0	0
Register Vn[3]		left unchanged		

Note that the lowest element of the vector Vn[0] is set up ready for a subsequent indexing operation by inserting the CLUTBASE base address of a color lookup table, which must be 64-byte aligned.

Typically, Pixel Map type 1 pixel reading code will look something like this:

```
ld_p      (uv),v1      load 4 bit texture value, in a form
                        which can be used for table lookup
nop
ld_p      (R4),v1      allow the load to complete
                        load indexed value from CLUT. CLUT is
                        sixteen 32-bit packed elements.
                        linpix_type must be data type 4.
```

Data Type 2 – 16 bit pixels

Type 2 pixels are 16 bits per pixel. They represent a physical color, thus:

Y	Cr	Cb
15 10 9	5 4	0

When these pixels are used for display generation zeroes are added in the least significant positions to increase them to 8 bits per field.

They can be used by load pixel instructions, but may not be stored. However, type 4 pixels in MPE RAM may be converted to type 2 in DRAM by DMA transfer.

Byte address	0	1
Bit address	15-8	7-0
	P[15:0]	

The load pixel instruction will load a 16-bit pixel into a vector register as follows:

Bits	31-30	29-25	24	23-16	15-0
Register Vn[0] - Y	0	P[15:11]	P[10]	0	0
Register Vn[1] - Cr	SS	P[9:5]	0	0	0
Register Vn[2] - Cb	SS	P[4:0]	0	0	0
Register Vn[3]	left unchanged				

For loads, if the appropriate **chnorm** bit is set, then 128 is subtracted from the values placed in Vn[1] and Vn[2]. Since these are assumed to be signed 2.14 numbers, the value %100000 00000000 is actually subtracted from bits 29-16. When this is done, sign extension is performed into bits 30-31 (SS in the table above).

For stores, the pixel value is saturated from the 2.14 bit representation in the small vector, so that the Y value lies in the range 0-1 (\$00 to \$FF), and so the Cr and Cb values lie in the range -½ to +½ (\$80 to \$7F). If the **chnorm** bit is set then 128 is added to the saturated chrominance values before storing them.

Data Type 3 – 8 bit pixels

Type 3 pixels are eight bits. The value represents an index into an arbitrary look-up table, and so it has no fixed relationship with the physical appearance. These are sometimes useful for memory efficient texture maps, and can be used for very memory efficient display buffers. They can be used by load pixel instructions, but may not be directly stored.

Type 3 pixels are always stored together in groups of two in a 16-bit word. This represents a horizontal strip of two pixels. All DMA operations on type 3 pixels must be defined with their X position and length as multiples of two pixels.

Byte address	0	1
Bit address	15-8	7-0
	1	2

The load pixel instruction will load an 8-bit pixel into a vector register as follows:

Bits	31-10	9-2	1-0
Register Vn[0]	CLUTBASE(31:10)	P[7:0]	0
Register Vn[1]	0	0	0
Register Vn[2]	0	0	0

Register Vn[3]	left unchanged
----------------	----------------

Note that the lowest element of the vector Vn[0] is set up ready for a subsequent indexing operation by inserting the CLUTBASE base address of a color lookup table, which must be 1024-byte aligned.

Typically, Pixel Map type 3 pixel reading code will look something like this:

```
ld_p      (uv),v1      load 4 bit texture value, in a form
                        which can be used for table lookup
nop
ld_p      (r4),v1      allow the load to complete
                        load indexed value from CLUT. CLUT is
                        256 32-bit packed elements. linpix_type
                        must be data type 4.
```

Data type 4 – 32-bit pixels

Type 4 pixels are 32 bits per pixel. They represent a physical color, thus:

Y	Cr	Cb	control
31	24 23	16 15	8 7 0

They can be used by load and store pixel instructions, and can be present in DRAM and in MPE RAM.

The load pixel instruction will map a 32-bit element P[31:0] into a vector register as follows:

	Bits	31-30	29-22	21-16	15-0
Register Vn[0] - Y		0	P[31:24]	0	0
Register Vn[1] - Cr		SS	P[23:16]	0	0
Register Vn[2] - Cb		SS	P[15:8]	0	0
Register Vn[3]		left unchanged			

The load and store pixel with Z (**ld_pz** and **st_pz**) instructions will map a 32-bit element P[31:0] into a vector register as follows:

	Bits	31-30	29-22	21-16	15-0
Register Vn[0] - Y		0	P[31:24]	0	0
Register Vn[1] - Cr		SS	P[23:16]	0	0
Register Vn[2] - Cb		SS	P[15:8]	0	0
Register Vn[3] - control		P[7:0] + 00		0	0

For loads, if the appropriate **chnorm** bit is set, then 128 is subtracted from the values placed in Vn[1] and Vn[2]. Since these are assumed to be signed 2.14 numbers, the value %100000 00000000 is actually subtracted from bits 29-16. When this is done, sign extension is performed into bits 30-31 (SS in the table above).

For stores, the pixel value is saturated from the 2.14 bit representation in the small vector, so that the Y value lies in the range 0-1 (\$00 to \$FF), and so the Cr and Cb values lie in the range -½ to +½ (\$80 to \$7F). If the **chnorm** bit is set then 128 is added to the saturated chrominance values before storing them.

Data type 5 – 16 bit pixels with 16 bit Z

Type 5 pixels are 16 bits per pixel, with an associated 16-bit control value, usually used for a Z-buffer depth. The 16 pixel bits represent a physical color, thus:

Y	Cr	Cb	Z
31 26	25 21	20 16	15 0

When these pixels are used for display generation zeroes are added in the least significant positions to increase them to 8 bits per field.

They can be used by load and store pixel and small vector instructions. Type 5 pixels in MPE RAM can be converted to types 7-B by DMA transfer to DRAM, or can be transferred directly.

The load pixel instruction will map a type 5 element to a vector register as follows:

Bits	31-30	29-25	24	23-16	15-0
Register Vn[0] - Y		P[15:11]	P[10]	0	0
Register Vn[1] - Cr	SS	P[9:5]	0	0	0
Register Vn[2] - Cb	SS	P[4:0]	0	0	0
Register Vn[3] - Z	left unchanged				

The load and store pixel with Z (**ld_pz** and **st_pz**) instructions will map a type 5 element to a vector register as follows:

Bits	31-30	29-25	24	23-16	15-0
Register Vn[0] - Y		P[15:11]	P[10]	0	0
Register Vn[1] - Cr	SS	P[9:5]	0	0	0
Register Vn[2] - Cb	SS	P[4:0]	0	0	0
Register Vn[3] - Z	Z[15:0]				0

For loads, if the appropriate **chnorm** bit is set, then 128 is subtracted from the values placed in Vn[1] and Vn[2]. Since these are assumed to be signed 2.14 numbers, the value %100000 00000000 is actually subtracted from bits 29-16. When this is done, sign extension is performed into bits 30-31 (SS in the table above).

For stores, the pixel value is saturated from the 2.14 bit representation in the small vector, so that the Y value lies in the range 0-1 (\$00 to \$FF), and so the Cr and Cb values lie in the range -½ to +½ (\$80 to \$7F). If the **chnorm** bit is set then 128 is added to the saturated chrominance values before storing them.

Data Type 6 – 32-bit pixels with 32-bit Z

Type 6 pixels are 32 bits per pixel, with an associated 32-bit control value, usually used for a Z-buffer depth. They represent a physical color, thus:

Y	Cr	Cb	unused	Z
63 56	55 48	47 40	39 32	31 0

They can be used by load and store pixel and small vector instructions, and can be present in DRAM and in MPE RAM.

The load pixel instruction will store to a type 6 element from a pixel register as follows:

Bits	31-30	29-22	21-0
Register Vn[0] - Y	0	P[31:24]	0
Register Vn[1] - Cr	SS	P[23:16]	0
Register Vn[2] - Cb	SS	P[15:8]	0

Register Vn[3] - Z	left unchanged
--------------------	----------------

The load and store pixel with Z (**ld_pz** and **st_pz**) instructions will map a type 6 element to a vector register as follows:

Bits	31-30	29-22	21-0
Register Vn[0] - Y	0	P[31:24]	0
Register Vn[1] - Cr	SS	P[23:16]	0
Register Vn[2] - Cb	SS	P[15:8]	0
Register Vn[3] - Z	Z[31:0]		

For loads, if the appropriate **chnorm** bit is set, then 128 is subtracted from the values placed in Vn[1] and Vn[2]. Since these are assumed to be signed 2.14 numbers, the value %100000 00000000 is actually subtracted from bits 29-16. When this is done, sign extension is performed into bits 30-31 (SS in the table above).

For stores, the pixel value is saturated from the 2.14 bit representation in the small vector, so that the Y value lies in the range 0-1 (\$00 to \$FF), and so the Cr and Cb values lie in the range -½ to +½ (\$80 to \$7F). If the **chnorm** bit is set then 128 is added to the saturated chrominance values before storing them.

MPE REGISTER SET REFERENCE

This section defines the MPE's internal registers. Bits not defined here are reserved. Reserved bits may read as 0 or 1 (undefined), and should always be written as 0. However, it is permissible to write back unmodified any value read from a read / write register. These registers are accessed with the normal load and store scalar instructions, except the Comm Bus receive and transmit registers, which may also be accessed with the load and store vector instructions.

mpectl MPE Control Register

Address: \$2050_0000
Read / Write

This register controls the basic operation of the MPE processor. It is modified by writing a pattern to it with the appropriate set or clear bits set to one, and all other bits zero. This allows atomic modifications, so that read-modify-write operations are normally unnecessary.

Bit	“Write” value	“Read” value	Description
31-28	(reserved)	(reserved)	
27-24	cycleType	cycleType	Internal state for use by the debugger only.
23	cycleType_wren	0	This bit must be set for the cycleType bits to be written. Writing a zero has no effect.
22-16	(reserved)	(reserved)	
15	(reserved)	mpeWasReset	This bit will be set whenever the MPE comes out of reset.
14	mpeWasReset_clr	0	Writing a one clears mpeWasReset . Writing a zero has no effect.
13	resetMpe_set	resetMpe	Writing a one causes the MPE to be reset. Writing a zero has no effect. Reset clears this bit.
12	(reserved)	0	
11	intToHost_set	intToHost	When this bit is set, an exception interrupt will be generated from this MPE to the debug control module. Writing a zero has no effect.
10	intToHost_clr	0	Writing a one clears the intToHost register. Writing a zero has no effect.
9	mpeIs2x_set	mpeIs2X	Writing a one to this bit sets the control bit that allows the MPE to run at 2X speed (108 MHz instead of 54 MHz). Writing a zero has no effect. When read, this bit gives the state of this control bit. <i>See note below.</i> <i>Aries 3 and up only.</i>
8	mpeIs2x_clr	0	Writing a one to this bit clears the control bit that allows the MPE to run at 2X speed. Writing a zero has no effect. <i>See note below.</i> <i>Aries 3 and up only</i>
7	daWrBrkEn_set	daWrBrkEn	Writing a one to this bit enables the data address write breakpoint. Writing a zero has no effect. When read, this bit gives the state of this enable. See the dabreak register description below.
6	daWrBrkEn_clr	0	Writing a one clears the data address write breakpoint enable. Writing a zero has no effect.

Bit	“Write” value	“Read” value	Description
5	daRdBrkEn_set	daRdBrkEn	Writing a one enables the data address read breakpoint. Writing a zero has no effect. When read, this bit gives the state of this enable. See the dabreak register description below.
4	daRdBrkEn_clr	0	Writing a one clears the data address read break point. Writing a zero has no effect.
3	singleStep_set	singleStep	While singleStep is set, the MPE is in single step mode. In this mode, each time mpeGo is set only one instruction packet will be executed, then mpeGo will be cleared. Writing a one sets the singleStep bit. Writing a zero has no effect.
2	singleStep_clr	0	Writing as one clears the singleStep bit. Writing a zero has no effect.
1	mpeGo_set	mpeGo	While mpeGo is set, the MPE is enabled and will execute instructions (but see singleStep above). Writing a one sets the mpeGo bit. Writing a zero has no effect.
0	mpeGo_clr	0	Writing as one clears the mpeGo bit. Writing a zero has no effect.

Note: You can only change the MPE speed when you are certain that there is no activity on the main bus, other bus, or comm. bus for that MPE. This means all these interfaces must be idle, there is no cache activity, you must be certain no other device is trying to send you comm. bus packets, and no other device is making you the source or destination of a DMA. After the speed change you should allow a few cycles (say five) before doing anything. This means that the code making the speed change must be resident.

The safe way to do this is to use the BIOS `_CompatibilityMode` call, and not to change these bits directly.

excepsrc Exception Source Register

Address: \$2050_0010
Read / Write

The hardware sets these “exception source” bits when the corresponding hardware exception occurs, whether or not the MPE is attempting to set or clear the bit in the same tick. For each bit, writing a zero has no effect, while writing a one sets the bit.

Bit	Name	Description
12	excepSrc_copr_error	coprocessor error
11	excepSrc_cdma_error	coprocessor dma error
10	excepSrc_odma_error	otherbus dma error
9	excepSrc_mdma_error	mainbus dma error
8	excepSrc_iport_address_error	iport-address error
7	excepSrc_dbus_address_error	dbus-address error
6	excepSrc_bilin_addr_error	bilinear address error
5	excepSrc_rfmulport_error	regfile mul-write port conflict
4	excepSrc_rfmemport_error	regfile mem-write port conflict
3	excepSrc_da_breakpoint	data-address breakpoint

Bit	Name	Description
2	excepSrc_breakpointnow	breakpoint instruction
1	excepSrc_singleStep	single-step break
0	excepSrc_halt	“halt” instruction

excepcr **Exception Clear**

Address: \$2050_0020
Read / Write

Writing a 1 to any bit in this register clears the corresponding bit in the excepsrc register (unless the hardware is capturing that exception into excepsrc in that same tick). Writing a 0 has no effect. Always reads as zero.

excephalten **Exception Halt Enable Register**

Address: \$2050_0030
Read / Write

There are 13 conditions that are classified as “exceptions”, including error conditions and debug conditions. Normally, when an exception is captured in the excepsrc register, it causes the MPE to halt and raise its outgoing exception signal so that some other MPE or debug host can deal with the situation. However, this behavior may be disabled for each exception condition by setting the corresponding excepHaltEn bit to zero. Then if this exception occurs, the MPE will cause the “exception” bit in its own intsrc register to be set.

Bit	Name	Description
12	excepHaltEn_copr_error	coprocessor error
11	excepHaltEn_cdma_error	coprocessor dma error
10	excepHaltEn_odma_error	otherbus dma error
9	excepHaltEn_mdma_error	mainbus dma error
8	excepHaltEn_oport_address_error	oport-address error
7	excepHaltEn_dbus_address_error	dbus-address error
6	excepHaltEn_bilin_addr_error	bilinear address error
5	excepHaltEn_rfmultiport_error	regfile mul-write port conflict
4	excepHaltEn_rfmemport_error	regfile mem-write port conflict
3	excepHaltEn_da_breakpoint	data-address breakpoint
2	excepHaltEn_breakpointnow	breakpoint instruction
1	excepHaltEn_singleStep	single-step break
0	excepHaltEn_halt	“halt” instruction

cc **Condition Code Register**

Address: \$2050_0040
Read / Write

As described in the MPE instruction reference section, condition code flags are set or cleared or unchanged by each instruction. ECU instructions can test combinations of these flags for conditional branches and jumps.

Bit	Name	Description
10	cf1	Co-processor flag 1. Used by MPE coprocessor hardware.
9	cf0	Co-processor flag 0. Used by MPE coprocessor hardware, e.g. MPE 2 interface to BDU.
8	modmi	RCU range low flag. Indicates whether the source operand of a modulo or range instruction is less than zero
7	modge	RCU range high flag. Indicates whether the source operand of a modulo or range instruction is greater than or equal to the relevant range register
6	c1z	RCU rc0 register zero flag. Indicates whether the rc0 counter is zero.
5	c0z	RCU rc1 register zero flag. Indicates whether the rc1 counter is zero.
4	mv	MUL overflow flag. Indicates whether significant bits are lost as a result of the shift in a scalar multiply instruction.
3	n	ALU negative flag. Indicates whether a scalar ALU result is negative.
2	v	ALU overflow flag. Indicates whether a scalar ALU operation overflows.
1	c	ALU carry / borrow flag. Indicates whether there is a carry from an ALU scalar addition or a borrow from an ALU scalar subtraction or compare.
0	z	ALU zero flag. Indicates whether a scalar ALU result is zero.

pcfetch

Program Counter at Fetch stage

Address: \$2050_0050
Read / Write

When read, this location reflects the current state of the program counter. It may only be written when the **mpeGo** bit in **mpectl** is cleared, and it must always be written to with a valid program address before setting that bit.

pcroute

Program Counter at Route stage

Address: \$2050_0060
Read / Write

This location gives the program address of the instruction currently at the routing stage of the pipeline. This is a full 32-bit address on a word boundary, i.e. bit 0 is always clear. It is not normally written to.

pcexec

Program Counter at Execute stage

Address: \$2050_0070
Read / Write

This location gives the program address of the instruction currently at the execute stage of the pipeline. It is not normally written to.

rz

Sub-routine return address

Address: \$2050_0080
Read / Write

This register is used for sub-routine calls to hold the return program address. Refer to the ECU description.

rzi1

Level 1 interrupt return address

Address: \$2050_0090
Read / Write

This register is used to return from level 1 interrupts, and holds the program fetch address to be restored. When taking a level-1 interrupt, the value in **pcroute** is loaded into **rzi1**. Then, when an **rti cc,rzi1** or **rti cc,rzi1,nop** instruction is taken, **rzi1** is used as the jump destination. Refer to the ECU description.

rzi2

Level 2 interrupt return address

Address: \$2050_00A0
Read / Write

This register is used to return from level 2 interrupts, and holds the program fetch address to be restored. When taking a level-2 interrupt, the value in **pcroute** is loaded into **rzi2**. Then, when an **rti cc,rzi2** or **rti cc,rzi2,nop** instruction is taken, **rzi2** is used as the jump destination. Refer to the ECU description.

intvec1

Level 1 interrupt vector

Address: \$2050_00B0
Read / Write

This location holds the program address of the level 1 interrupt service routine. Control is transferred to this location in instruction memory when a level 1 interrupt occurs, by forcing a jump.

intvec2

Level 2 interrupt vector

Address: \$2050_00C0
Read / Write

This location holds the program address of the level 2 interrupt service routine. Control is transferred to this location in instruction memory when a level 2 interrupt occurs, by forcing a jump.

intsrc

Interrupt source

Address: \$2050_00D0
Read / Write

The hardware sets these “interrupt source” bits when the corresponding hardware interrupt signal is high, whether or not the corresponding enable bit in the **intctl** register is set, whether or not the MPE is halted, and whether or not the MPE is attempting to set or clear that bit in the same tick. For each bit, writing a zero has no effect, while writing a one sets the bit. The bits correspond to interrupts as follows:

Bit	Interrupt	Description
31	vidtimer	VDG beam position interrupt
30	systimer1	System timer 1 interrupt
29	systimer0	System timer 0 interrupt
28	gpio	GPIO IO pin combined interrupt
27	audio	Audio system interrupt
26	host	External host (System Bus) interrupt
25	debug	Debug control unit interrupt
24	mcumbdone	MCU macro-block done interrupt
23	mcudctdone	MCU DCT done interrupt
22	bdumbdone	BDU macro-block done interrupt (MPE 2 only)
21	bduerror	BDU error flag (MPE 2 only)
20	iicperiph	Serial Peripheral Bus interrupt
19	mdmafinish	Main Bus DMA finish interrupt (for debug)
18	mdmadump	Main Bus DMA dump interrupt (for debug)
17	mdmaotf	Main Bus DMA otf interrupt (for debug)
16	systimer2	System timer 2 interrupt
13	vdmaready	VLD DMA ready interrupt (MPE 1 only)
12	vdmadone	VLD DMA done interrupt (MPE 1 only)
9	odmaready	Other Bus DMA ready interrupt
8	odmadone	Other Bus DMA done interrupt
7	mdmaready	Main Bus DMA ready interrupt
6	mdmadone	Main Bus DMA done interrupt
5	commxmit	Comm Bus transmit buffer empty interrupt
4	commrecv	Comm Bus receive buffer full interrupt
1	software	Software generated interrupt
0	exception	Local MPE exception interrupt

intclr

Interrupt clear

Address: \$2050_00E0
Read / Write

Writing a one to any bit in this register clears the corresponding bit in the **intsrc** register, while writing a zero has no effect. Always reads as zero. The bits correspond to interrupts as follows:

intctl

Interrupt control register

Address: \$2050_00F0
Read / Write

This register is used to control the masking of interrupts. The **imaskHw** bits are set by the hardware whenever the corresponding interrupt occurs, and are cleared by the hardware when the **rti** from the ISR is executed. They may be left set throughout the ISR, or cleared and set appropriately to allow re-entrant interrupt behavior. Never set or clear the **imaskHw1** bit unless the **imaskSw1** bit is already set. Never set or clear the **imaskHw2** bit unless the **imaskSw2** bit is already set.

The **imaskSw** bits also provide a mechanism for masking interrupts in software.

This register may be modified by writing a pattern to it with the appropriate set or clear bits set to one, and all other bits zero. This allows atomic modifications to this register, so that read-modify-write operations are normally unnecessary.

imaskSw2 masks level-2 interrupts

imaskHw2 masks both level-1 and level-2 interrupts

imaskSw1 masks level-1 interrupts

imaskHw1 masks level-1 interrupts

Bit	Name	Description
7	imaskSw2_set	Writing a one to this location sets the imaskSw2 bit, writing a zero has no effect. The imaskSw2 bit can be read from here. Software can set the imaskSw2 bit to mask the level 2 interrupt and clear it to enable whichever interrupt source is selected. The imaskSw2 bit is not changed by hardware, other than forcing it to zero on reset.
6	imaskSw2_clr	Writing a one to this location clears the imaskSw2 bit, writing a zero has no effect. This bit is always read as zero.
5	imaskHw2_set	Writing a one to this location sets the imaskHw2 bit, writing a zero has no effect. The imaskHw2 bit can be read from here. The imaskHw2 bit is set when taking a level 2 interrupt branch, is cleared by the rti from a level 2 ISR, and may be set or cleared by software just like the imaskSw2 bit.
4	imaskHw2_clr	Writing a one to this location clears the imaskHw2 bit, writing a zero has no effect. This bit is always read as zero.
3	imaskSw1_set	Writing a one to this location sets the imaskSw1 bit, writing a zero has no effect. The imaskSw1 bit can be read from here. Software can set the imaskSw1 bit to mask the level 1 interrupt and clear it to enable whichever interrupt sources are selected. The imaskSw1 bit is not changed by hardware, other than forcing it to zero on reset.
2	imaskSw1_clr	Writing a one to this location clears the imaskSw1 bit, writing a zero has no effect. This bit is always read as zero.
1	imaskHw1_set	Writing a one to this location sets the imaskHw1 bit, writing a zero has no effect. The imaskHw1 bit can be read from here. The imaskHw1 bit is set when taking a level 1 interrupt branch, is cleared by the rti from a level 1 ISR, and may be set or cleared by software just like the imaskSw1 bit.
0	imaskHw1_clr	Writing a one to this location clears the imaskHw1 bit, writing a zero has no effect. This bit is always read as zero.

inten1

Level 1 interrupt enables

Address: \$2050_0100
Read / Write

This register defines which interrupts are enabled as level 1 interrupts. Any number of level 1 interrupts may be simultaneously enabled. The bits correspond to interrupts as follows:

Bit	Enbale	Description
31	vidtimer	VDG beam position interrupt enable
30	systimer1	System timer 1 interrupt enable
29	systimer0	System timer 0 interrupt enable
28	gpio	GPIO IO pin combined interrupt enable
27	audio	Audio system interrupt enable
26	host	External host (sSstem Bus) interrupt enable
25	debug	Debug control unit interrupt enable
24	mcumbdone	MCU macro-block done interrupt enable
23	mcudctdone	MCU DCT done interrupt enable
22	bdumbdone	BDU macro-block done interrupt enable (MPE 2 only)
21	bduerror	BDU error flag interrupt enable (MPE 2 only)
20	iicperiph	Serial Peripheral Bus interrupt enable
19	mdmafinish	Main Bus DMA finish interrupt enable (for debug)
18	mdmadump	Main Bus DMA dump interrupt enable (for debug)
17	mdmaotf	Main Bus DMA otf interrupt enable (for debug)
16	systimer2	System timer 2 interrupt enable
13	vdmaready	VLD DMA ready interrupt enable (MPE 1 only)
12	vdmadone	VLD DMA done interrupt enable (MPE 1 only)
9	odmaready	Other Bus DMA ready interrupt enable
8	odmadone	Other Bus DMA done interrupt enable
7	mdmaready	Main Bus DMA ready interrupt enable
6	mdmadone	Main Bus DMA done interrupt enable
5	commxmit	Comm Bus transmit buffer empty interrupt enable
4	commrecv	Comm Bus receive buffer full interrupt enable
1	software	Software generated interrupt enable
0	exception	Local MPE exception interrupt enable

inten1set

Level 1 interrupt enables set

Address: \$2050_0110
Read / Write

Writing a 1 to any bit in this register sets the corresponding bit in the inten1 register, while writing a 0 has no effect. Always reads the same as inten1.

inten1clr

Level 1 interrupt enables clear

Address: \$2050_0120
Read / Write

Writing a 1 to any bit in this register clears the corresponding bit in the inten1 register, while writing a 0 has no effect. Always reads as zero.

inten2sel

Level 2 interrupt enable

Address: \$2050_0130
Read / Write

This register defines which interrupt is enabled as a level 2 interrupt. Only one level 2 interrupt may be enabled at once. The selection of the interrupt source is programmed as follows:

Bits	Value	Enables	Description
4-0	31	vidtimer	The VDG beam position interrupt.
	30	systimer1	System timer 1.
	29	systimer0	System timer 0.
	28	gpio	The GPIO IO pin combined interrupt.
	27	audio	The audio system interrupt.
	26	host	The external host (System Bus) interrupt.
	25	debug	The debug control unit interrupt.
	24	mcumbdone	The MCU macro-block done interrupt.
	23	mcudctdone	The MCU DCT done interrupt.
	22	bdumbdone	The BDU macro-block done interrupt (MPE 2 only).
	21	bduerror	The BDU error interrupt (MPE 2 only).
	13	vdmaready	The VLD DMA ready interrupt (MPE 1 only).
	12	vdmadone	The VLD DMA done interrupt (MPE 1 only).
	9	odmaready	The Other Bus DMA ready interrupt.
	8	odmadone	The Other Bus DMA done interrupt.
	7	mdmaready	The Main Bus DMA ready interrupt.
	6	mdmadone	The Main Bus DMA done interrupt.
	5	commxmit	The Comm Bus transmit buffer empty interrupt.
	4	commrecv	The Comm Bus receive buffer full interrupt.
	1	software	Software generated interrupt.
	0	exception	Local MPE exception

rc0

Counter register rc0

Address: \$2050_01E0
Read / Write

This register is the first of two hardware loop counters in the RCU.

Bit	Name	Description
15-0	rc0	Loop counter 0

rc1 Counter register rc1

Address: \$2050_01F0
Read / Write

This register is the second of two hardware loop counters in the RCU.

Bit	Name	Description
15-0	rc1	Loop counter 1

rx Register rx

Address: \$2050_0200
Read / Write

This 32-bit register forms the X part of the XY address generator. It is normally a 16.16 bit number. See the memory unit description for more details.

ry Register ry

Address: \$2050_0210
Read / Write

This 32-bit register forms the Y part of the XY address generator. It is normally a 16.16 bit number. See the memory unit description for more details.

xyrange rx / ry range values

Address: \$2050_0220
Read / Write

Sets the range of **rx** and **ry** for the **modulo** and **range** instructions.

Bit	Description
25-16	X range
9-0	Y range

xybase XY address generator base address

Address: \$2050_0230
Read / Write

Bit	Name	Description
31-2	xybase	Address of XY map in physical memory

xyctl

XY control flags

Address: \$2050_0240
Read / Write

Bit	Name	Description																																																				
30	x_rev	Bit reverse X for FFT addressing																																																				
29	y_rev	Bit reverse Y for FFT addressing																																																				
28	xy_chnorm	Flags chrominance normalization																																																				
26-24	xy_mipmap	Binary point position of X and Y																																																				
23-20	xy_type	Pixel map type of XY image, described in full in the Memory Unit section. <table><tr><th>Type</th><th>Mapping</th><th>Bits</th><th>Note</th></tr><tr><td>0</td><td>MPEG pixel</td><td>24</td><td>see notes on storage format</td></tr><tr><td>1</td><td>Pixel data type 1</td><td>4</td><td>for CLUT lookup</td></tr><tr><td>2</td><td>Pixel data type 2</td><td>16</td><td></td></tr><tr><td>3</td><td>Pixel data type 3</td><td>8</td><td>for CLUT lookup</td></tr><tr><td>4</td><td>Pixel data type 4</td><td>32</td><td></td></tr><tr><td>5</td><td>Pixel data type 5</td><td>32</td><td></td></tr><tr><td>6</td><td>Pixel data type 6</td><td>64</td><td></td></tr><tr><td>8</td><td>Byte</td><td>8</td><td>not valid as a pixel load/store type</td></tr><tr><td>9</td><td>Word</td><td>16</td><td>not valid as a pixel load/store type</td></tr><tr><td>A</td><td>Scalar</td><td>32</td><td>not valid as a pixel load/store type</td></tr><tr><td>C</td><td>Small vector</td><td>64</td><td>not valid as a pixel load/store type</td></tr><tr><td>D</td><td>Vector</td><td>128</td><td>not valid as a pixel load/store type</td></tr></table>	Type	Mapping	Bits	Note	0	MPEG pixel	24	see notes on storage format	1	Pixel data type 1	4	for CLUT lookup	2	Pixel data type 2	16		3	Pixel data type 3	8	for CLUT lookup	4	Pixel data type 4	32		5	Pixel data type 5	32		6	Pixel data type 6	64		8	Byte	8	not valid as a pixel load/store type	9	Word	16	not valid as a pixel load/store type	A	Scalar	32	not valid as a pixel load/store type	C	Small vector	64	not valid as a pixel load/store type	D	Vector	128	not valid as a pixel load/store type
Type	Mapping	Bits	Note																																																			
0	MPEG pixel	24	see notes on storage format																																																			
1	Pixel data type 1	4	for CLUT lookup																																																			
2	Pixel data type 2	16																																																				
3	Pixel data type 3	8	for CLUT lookup																																																			
4	Pixel data type 4	32																																																				
5	Pixel data type 5	32																																																				
6	Pixel data type 6	64																																																				
8	Byte	8	not valid as a pixel load/store type																																																			
9	Word	16	not valid as a pixel load/store type																																																			
A	Scalar	32	not valid as a pixel load/store type																																																			
C	Small vector	64	not valid as a pixel load/store type																																																			
D	Vector	128	not valid as a pixel load/store type																																																			
19-16	x_tile	Defines the mask for X for tiling source bit-maps																																																				
15-12	y_tile	Defines the mask for Y for tiling source bit-maps																																																				
10-0	xy_width	Width of XY image along Y dimension																																																				

ru

Register ru

Address: \$2050_0250
Read / Write

This 32-bit register forms the U part of the UV address generator. It is normally a 16.16 bit number. See the memory unit description for more details.

rv

Register rv

Address: \$2050_0260
Read / Write

This 32-bit register forms the V part of the UV address generator. It is normally a 16.16 bit number. See the memory unit description for more details.

uvrange

ru / rv range values

Address: \$2050_0270
Read / Write

Sets the range of **ru** and **rv** for the **modulo** and **range** instructions.

Bit	Description
25-16	U range
9-0	V range

uvbase UV address generator base address

Address: \$2050_0280

Read / Write

Bit	Name	Description
31-2	uvbase	Address of UV map in physical memory

uvctl UV control flags

Address: \$2050_0290

Read / Write

Bit	Name	Description																																																				
30	u_rev	Bit reverse U for FFT addressing																																																				
29	v_rev	Bit reverse V for FFT addressing																																																				
28	uv_chnorm	Flags chrominance normalization																																																				
26-24	uv_mipmap	Binary point position of U and V																																																				
23-20	uv_ttype	Pixel map type of UV image, described in full in the Memory Unit section. <table><tr><th>Type</th><th>Mapping</th><th>Bits</th><th>Note</th></tr><tr><td>0</td><td>MPEG pixel</td><td>24</td><td>see notes on storage format</td></tr><tr><td>1</td><td>Pixel data type 1</td><td>4</td><td>for CLUT lookup</td></tr><tr><td>2</td><td>Pixel data type 2</td><td>16</td><td></td></tr><tr><td>3</td><td>Pixel data type 3</td><td>8</td><td>for CLUT lookup</td></tr><tr><td>4</td><td>Pixel data type 4</td><td>32</td><td></td></tr><tr><td>5</td><td>Pixel data type 5</td><td>32</td><td></td></tr><tr><td>6</td><td>Pixel data type 6</td><td>64</td><td></td></tr><tr><td>8</td><td>Byte</td><td>8</td><td>not valid as a pixel load/store type</td></tr><tr><td>9</td><td>Word</td><td>16</td><td>not valid as a pixel load/store type</td></tr><tr><td>A</td><td>Scalar</td><td>32</td><td>not valid as a pixel load/store type</td></tr><tr><td>C</td><td>Small vector</td><td>64</td><td>not valid as a pixel load/store type</td></tr><tr><td>D</td><td>Vector</td><td>128</td><td>not valid as a pixel load/store type</td></tr></table>	Type	Mapping	Bits	Note	0	MPEG pixel	24	see notes on storage format	1	Pixel data type 1	4	for CLUT lookup	2	Pixel data type 2	16		3	Pixel data type 3	8	for CLUT lookup	4	Pixel data type 4	32		5	Pixel data type 5	32		6	Pixel data type 6	64		8	Byte	8	not valid as a pixel load/store type	9	Word	16	not valid as a pixel load/store type	A	Scalar	32	not valid as a pixel load/store type	C	Small vector	64	not valid as a pixel load/store type	D	Vector	128	not valid as a pixel load/store type
Type	Mapping	Bits	Note																																																			
0	MPEG pixel	24	see notes on storage format																																																			
1	Pixel data type 1	4	for CLUT lookup																																																			
2	Pixel data type 2	16																																																				
3	Pixel data type 3	8	for CLUT lookup																																																			
4	Pixel data type 4	32																																																				
5	Pixel data type 5	32																																																				
6	Pixel data type 6	64																																																				
8	Byte	8	not valid as a pixel load/store type																																																			
9	Word	16	not valid as a pixel load/store type																																																			
A	Scalar	32	not valid as a pixel load/store type																																																			
C	Small vector	64	not valid as a pixel load/store type																																																			
D	Vector	128	not valid as a pixel load/store type																																																			
19-16	u_tile	Defines the mask for U for tiling source bit-maps																																																				
15-12	v_tile	Defines the mask for V for tiling source bit-maps																																																				
10-0	uv_width	Width of UV image along V dimension																																																				

linpixctl Linear address pixel transfer control flags

Address: \$2050_02A0

Read / Write

This register sets the data type and chrominance normalization of **ld_p**, **ld_pz**, **st_p** and **st_pz** instructions that have a linear address. See the memory unit description for more details.

Bit	Name	Description
28	linpix_chnorm	Flags chrominance normalization for type 2 small vectors.
23-20	linpix_type	Defines the data mapping used for the linear forms of load and store pixel,

		and load and store pixel with Z. The meaning of the bits in this field is identical to xy_type , described in full in the Memory Unit section.		
Type	Mapping	Bits	Note	
0	MPEG pixel	24	see notes on storage format	
1	Pixel data type 1	4	for CLUT lookup	
2	Pixel data type 2	16		
3	Pixel data type 3	8	for CLUT lookup	
4	Pixel data type 4	32		
5	Pixel data type 5	32		
6	Pixel data type 6	64		
8	Byte	8	not valid as a pixel load/store type	
9	Word	16	not valid as a pixel load/store type	
A	Scalar	32	not valid as a pixel load/store type	
C	Small vector	64	not valid as a pixel load/store type	
D	Vector	128	not valid as a pixel load/store type	

clutbase

Base address of color lookup table

Address: \$2050_02B0

Read / Write

This is the base address of the color lookup table, which is used for Pixel Map types 1 and 3 (see Memory Unit for details). This table is only used for load pixel.

Bit	Name	Description
31-6	clutbase	Address of the color lookup table in physical memory

svshift

Small Vector Multiply Shift Control

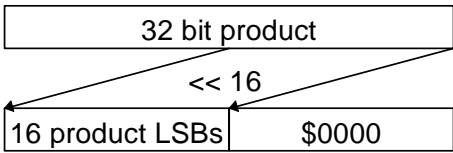
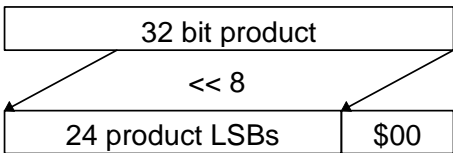
Address: \$2050_02C0

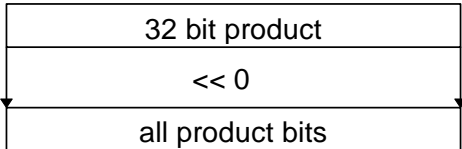
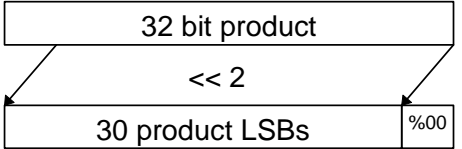
Read / Write

Defines the amount by which the 32-bit result of the **mul_sv** and **mul_p** instructions is shifted right.

Bit	Name	Description
1-0	svshift	Shift amount, see below.

The table of possible values for the right shift amount is:

Value	Shift by	Small vector product definition
0		for the product of 16.0 values as a 16.0 small vector value
1		for the product of 8.8 values as an 8.8 small vector value

Value	Shift by	Small vector product definition
2	 <pre> graph TD A[32 bit product] -- "<< 0" --> B[all product bits] </pre>	for the full 32-bit products
3	 <pre> graph TD A[32 bit product] -- "<< 2" --> B[30 product LSBs] B -- "%00" --> C[] </pre>	for the product of 2.14 values as a 2.14 small vector value

acshift Scalar Multiply Shift Control

Address: \$2050_02D0
Read / Write

Default right shift value used by some MUL unit instructions. This is a 7 bits, two's complement number, the valid range is +63 to -32. The value in here is sign extended to 32 bits on a read.

Bit	Name	Description
7-0	acshift	Shift amount, see below.

sp MPE Stack pointer

Address: \$2050_02E0
Read / Write

This is the stack pointer used by **push** and **pop** instructions. It must always lie on a vector boundary.

Bit	Name	Description
31-4	sp	Stack pointer.

dabreak Data Breakpoint Address

Address: \$2050_02F0
Read / Write

This is the 32-bit internal MPE Data Port address to break on. It works in conjunction with the **daRdBrkEn** and **daWrBrkEn** bits in the **mpectl** register.

When enabled, the data address breakpoint for read or write should trigger if the **dabreak** address is accessed (for read or write respectively) by any executed mem-unit operation. This is true even if the mem-unit address does not exactly match the data breakpoint address, so long as any part of the accessed data value is at the **dabreak** address. For example, even if the **dabreak** register has non-zero bits 3-0, a **ld_v** address (bits 3-0 are zero) which matches bits 31-4 of **dabreak** should trigger a da-read-breakpoint, while a **ld_b** address which matches bits 31-4 but doesn't match bits 3-0 should not trigger.

Bit	Name	Description
31-0	dabreak	Data breakpoint address.

r0-r31 Registers r0-r31

Address: \$2050_0300, \$2050_0310,, \$2050_04f0
Read / Write

Register File 32-bit scalar registers r0 to r31. The addresses above are for debugger DMA access only, and must not be used as the memory address operand of store and load instructions.

odmactl Other Bus DMA control and status register

Address: \$2050_0500
Read / Write

See the Other Bus section of this document for a description of MPE Other Bus DMA.

Bit	Description
6-5	Other Bus DMA bus priority, valid values are 1-2, with 2 being the highest priority. Default value is 1. 0 disables Other Bus DMA, and 3 is reserved for future use.
4	Other Bus DMA command pending, this flag means that the DMA command pointer must not be written to. This bit is read only.
3-0	Other Bus DMA active level, 0 indicates no activity, 1 means that a single DMA transfer is active, 2 means that a data transfer is active and a command is pending, higher values will not occur in Aries 1, 2 and 3. These bits are read only.

odmacptr Other Bus DMA command pointer

Address: \$2050_0510
Read / Write

The address of a valid Other Bus DMA command structure may be written to this register when the DMA pending flag is clear. Writing this register initiates the DMA process. The address written here must lie on a vector address boundary within the local MPE space.

Bit	Description
22-4	Other bus DMA command address.

Note: Application software should not normally perform any direct DMA operations, but should instead call the appropriate BIOS calls. There are potential interactions with the cache mechanism that make directly using this hardware dangerous to correct operation.

mdmactl Main Bus DMA control and status register

Address: \$2050_0600
Read / Write

See the Main Bus section of this document for a description of MPE Main Bus DMA.

Bits	Name	Description
31-24	done_cnt_wr	Write done count. This counter is used only when trying to set up multiple overlapping read and write transfers. It is incremented by the hardware whenever a write transfer completes, and is decremented by software. Valid values are between 0 and \$1D. Two error conditions may also exits, \$FE for overflow, and \$FF for underflow.
23-16	done_cnt_rd	Read done count. This counter is used only when trying to set up multiple overlapping read and write transfers. It is incremented by the hardware whenever a read transfer completes, and is decremented by software. Valid values are between 0 and \$1D. Two error conditions may also exits, \$FE for overflow, and \$FF for underflow.
15	cmd_error	Command error. The DMA controller has found an error while processing a command. There are Comm Bus registers in the DMA controller to determine what was wrong in detail.
14	dmpe_error	Command pointer error. One of the following things has occurred: <ul style="list-style-type: none"> • The command pointer write was to an invalid range • The command pointer incremented past a 32K byte range. • The command pointer was written while a transfer was pending.
11	done_cnt_wr_dec	Decrement write done count. When a one is written to this bit the write done counter is decremented. This should be performed when necessary to clear the interrupt condition.
10	done_cnt_rd_dec	Decrement read done count. When a one is written to this bit the read done counter is decremented. This should be performed when necessary to clear the interrupt condition.
9	done_cnt_enable	Done count enable. Writing a one to this bit enables the read and write done counter mechanism. This has a variety of effects, discussed below. When read this bit returns the enable status.
8	done_cnt_disable	Done count disable. Writing a one to this bit disables the read and write done counter mechanism. This has a variety of effects, discussed below. This bit always reads as zero.
6-5	priority	Bus priority. Sets the bus priority for MPE DMA transfers. Valid values are 1-3, with 3 being the highest priority.
4	pending	Command pending. This flag means that the DMA command pointer must not be written to. This bit is read only.
3-0	active	DMA active level, this give the number of DMA commands that have been accepted by the DMA controller, but whose data transfer is not yet complete. In theory, this can reach a level of around 6 or 7. These bits are read only.

mdmacptr

Main Bus DMA command pointer

Address: \$2050_0610

Read / Write

The address of a valid DMA command structure may be written to this register when the DMA pending flag is clear. Writing this register initiates the DMA process. The address written here must lie on a vector address boundary within the local MPE space.

Bit	Description
22-4	Main Bus DMA command address.

Note: Application software should not normally perform any direct DMA operations, but should instead call the appropriate BIOS calls. There are potential interactions with the cache mechanism that make directly using this hardware dangerous to correct operation.

comminfo **Communication Bus transfer information**

Address: \$2050_07E0
Read / Write

This register is used to pass an additional eight bits of data in each Comm Bus packet. This is only possible between the MPEs, and on packets sent from the coded data interface.

Bit	Description
23-16	Extended receive data (read only)
7-0	Extended transmit data (read / write)

commctl **Communication Bus status and control**

Address: \$2050_07F0
Read / Write

Bit	Description
31	Receive buffer full (read only)
30	Receive disable (read / write)
23-16	Received source ID (read only)
15	Transmit buffer full (read only)
14	Transmit failed (read only)
13	Transmit retry flag (read / write)
12	Transmit bus lock flag (read / write)
7-0	Transmit target ID (read / write)

commxmit **Communication Bus 128-bit transmit packet**

Address: \$2050_0800
Read / Write

This 128 bit register holds the Comm Bus transmit data. A write to the highest address triggers the transmit. However, this register will normally be written to all at once with a vector store, triggering the transmit. The four scalar registers are accessible as register **commxmit0** - **commxmit3**.

commrcv **Communication Bus 128-bit receive packet**

Address: \$2050_0810
Read Only

This 128 bit register holds the Comm Bus receive data. A read from the highest address clears the receive buffer full flag. However, this register will normally be read all at once with a vector load, clearing the flag. The four scalar registers are accessible as register **commrcv0** - **commrcv3**.

configa

Configuration a

Address: \$2050_0FF0
Read only

Bit	Name	Description
31-24	mmp_release	NUON release \$00 - \$01 Oz / Aries1 \$02 - \$03 Aries2 \$04 Aries3
23-16	mpe_release	MPE Release \$00 reserved for emulator \$01 Oz / Aries \$02 - \$03 Aries2 \$04 Aries3
15-8	mpe_identifier	MPE Identification Number \$00 mpe-0 \$01 mpe-1 \$02 mpe-2 \$03 mpe-3 \$99 reserved
7-2	reserved	
1-0	where_on_reset	MPE behavior coming out of reset 00 halted 01 executing from base of irom 1x executing from base of iram

configb

Configuration b

Address: \$2050_0FF4
Read only

reserved

dcachectl

Data Cache Control

Address: \$2050_0FF8
Read / Write
MPes 0 and 3 only

This register controls the operation of the data cache. It must be initialized, and the data tag memory cleared, before making any cacheable memory references.

Bit	Name	Description
30-28	cState	The MPE does not automatically stall after a dcacheable write access, even if it misses, unless another dcacheable access is done while the dcacheable write is still being handled. This means that when any piece of

		code is about to write the m/macptr or o/macptr registers, if the environment is such that a Cacheable-Write access might be in process, then either of two things must be done to avoid a possible collision: (1) Issue a cacheable read to the appropriate m/odma space, then check that the corresponding m/odma "pending" bit is clear, then go ahead and write the m/macptr register; or (2) Repeatedly check whether these cState bits are 0, and when they are, check that the m/odma "pending" bit is clear, then go ahead and write the m/macptr register.
26-24	cWaysTried	For diagnostic purposes only.
18-16	cCurrentWay	For diagnostic purposes only.
10-8	cWayAssoc	Number of cache ways, for multiple way set associative caching. Values 0-7 correspond to 1-8 way set associative, respectively.
5-4	cWaySize	Cache way size, this gives the size of each cache way, so the total memory used by the cache is the product of this and the number of ways. Value Way size 00 1024 bytes 01 2048 bytes 10 4096 bytes 11 8192 bytes
1-0	cBlockSize	Cache block size, this gives the size of the block fetched on a cache miss. Value Block size 00 16 bytes 01 32 bytes 10 64 bytes 11 128 bytes

icachectl

Instruction Cache Control

Address: \$2050_0FFC

Read / Write

MPEs 0 and 3 only

This register controls the operation of the instruction cache. It must be initialized, and the instruction tag memory cleared, before making any cacheable memory references.

Bit	Name	Description
30-28	cState	For diagnostic purposes only.
26-24	cWaysTried	For diagnostic purposes only.
18-16	cCurrentWay	For diagnostic purposes only.
10-8	cWayAssoc	Number of cache ways, for multiple way set associative caching. Values 0-7 correspond to 1-8 way set associative, respectively.
5-4	cWaySize	Cache way size, this gives the size of each cache way, so the total memory used by the cache is the product of this and the number of ways. Value Way size 00 1024 bytes 01 2048 bytes 10 4096 bytes 11 8192 bytes
1-0	cBlockSize	Cache block size, this gives the size of the block fetched on a cache miss. Value Block size

		00	16 bytes
		01	32 bytes
		10	64 bytes
		11	128 bytes

vdmactla **VLD DMA Control Register A**

Address: \$2050_1100

Read / Write

MPE 1 only

This register controls DMA channel A between MPE 1 and the VLD unit in the BDU. This hardware is specific to the MPEG decode function.

Bit	Name	Description
24	a_active	This bit should be set to initiate the transfer. It is cleared by hardware when transfer completes.
19-0	a_count	Number of bytes still to be transferred. Valid values are even and between \$00000 and \$03FFE.

vdmactlb **VLD DMA Control Register B**

Address: \$2050_1110

Read / Write

MPE 1 only

This register controls DMA channel B between MPE 1 and the VLD unit in the BDU. This hardware is specific to the MPEG decode function.

Bit	Name	Description
24	b_active	This bit should be set to initiate the transfer. It is cleared by hardware when transfer completes.
19-0	b_count	Number of bytes still to be transferred. Valid values are even and between \$00000 and \$03FFE.

vdmaptra **VLD DMA Pointer Register A**

Address: \$2050_1120

Read / Write

MPE 1 only

This register gives the address of the next vector to be read by VLD DMA channel A from MPE 1 memory. This hardware is specific to the MPEG decode function.

Bit	Name	Description
22-4	Aptr	Vector address in MPE memory.

vdmmaptrb

VLD DMA Pointer Register B

Address: \$2050_1130

Read / Write

MPE 1 only

This register gives the address of the next vector to be read by VLD DMA channel B from MPE 1 memory. This hardware is specific to the MPEG decode function.

Bit	Name	Description
22-4	Bptr	Vector address in MPE memory.

vld and bdu control registers

\$20501200 to \$20501320

Read / Write

MPE 2 only

See the BDU section of this document.

MPE INSTRUCTION SET REFERENCE

Instruction set summary

This list summarizes all the instructions available from all the function units. Each instruction typically offers several versions of pre-processing or effective addressing of the source data.

Mnemonic	Function unit	Description
abs	ALU	Convert the signed integer to its unsigned absolute value
add	ALU	Arithmetic addition
add_p	ALU	Add pixel values
add_sv	ALU	Add small vector
addm	MUL	Arithmetic addition using the MUL unit
addr	RCU	Special purpose register addition
addwc	ALU	Arithmetic addition with carry
and	ALU	32-bit logical AND of A and B
as	ALU	Arithmetic shift
asl	ALU	Arithmetic shift left
asr	ALU	Arithmetic shift right
bclr	ALU	Clear a bit in a register
bits	ALU	Bit field extraction
bra	ECU	conditional branch to an offset relative to the program counter
breakpoint	none	Debug breakpoint
bset	ALU	Set a bit in a register
btst	ALU	Test a bit in a register
butt	ALU	Butterfly operation (sum and difference) of two scalar values
cmp	ALU	Arithmetic compare
cmpwc	ALU	Arithmetic compare with carry
copy	ALU	Register to register move through the ALU
dec	RCU	Decrement rc0 or rc1 register, unless it is zero
dotp	MUL	Multiply all elements of a small vector, and produce their sum
eor	ALU	32-bit logical EOR of A and B
ftst	ALU	Bit field test
halt	ECU	Halt program execution
jmp	ECU	Conditional jump to an absolute address
jsr	ECU	Conditional jump to subroutine at an absolute address
ld_b	MEM	Load byte
ld_io	MEM	Obsolete instruction form, equivalent to load scalar
ld_p	MEM	Load pixel
ld_pz	MEM	Load pixel plus Z data
ld_s	MEM	Load scalar
ld_sv	MEM	Load small vector
ld_v	MEM	Load vector
ld_w	MEM	Load word
ls	ALU	Logical shift
lsl	ALU	Logical shift left
lsr	ALU	Logical shift right

Mnemonic	Function unit	Description
mirror	MEM	Reverse the bit order of a scalar
modulo	RCU	Range limit index register
msb	ALU	Find the MSB of the source operand
mul	MUL	Multiply two (32-bit) scalars
mul_p	MUL	Multiply all elements of a pixel
mul_sv	MUL	Multiply all elements of a small vector
mv_s	MEM	Move Scalar
mv_v	MEM	Move Vector
mvr	RCU	Move scalar data to index register
neg	ALU	Arithmetic complement
nop	ALU	Null operation
not	ALU	Logical complement
or	ALU	32-bit logical OR of A and B
pad	none	Instruction packet alignment padding
pop	MEM	Pop data from stack
push	MEM	Push data on to stack
range	RCU	Range check index register
rot	ALU	Rotate scalar
rti	ECU	Return from interrupt
rts	ECU	Return from subroutine
sat	ALU	Arithmetic saturation
st_io	MEM	Obsolete instruction form, equivalent to store scalar
st_p	MEM	Store pixel
st_pz	MEM	Store pixel plus Z data
st_s	MEM	Store scalar
st_sv	MEM	Store small vector
st_v	MEM	Store vector
sub	ALU	Arithmetic subtraction
subwc	ALU	Arithmetic subtraction with carry
sub_p	ALU	Subtract pixels
sub_sv	ALU	Subtract small vectors
subm	MUL	Arithmetic subtraction using the MUL unit

Function Unit: ALU

Operation: absolute_value (Scalar Register) \Rightarrow Scalar Register

Description: Convert the input signed integer to an unsigned integer by negating it if it is negative, and leaving it unchanged if it is positive. The carry flag reflects the sign of the value prior to this operation.

Note that this function might be considered to fail if the input value is \$80000000 as there is no positive signed integer equivalent. Of course, if the output value is considered to be an unsigned integer, then the result is correct. The n or v flag may be used to detect this condition.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
16-bit forms	
abs Sk	take the absolute value of Sk, writing the result to Sk

Operand Values: Sk is any scalar register r0-r31.

Condition Codes: z : set if the result is zero, cleared otherwise.
n : set if the result is negative, cleared otherwise.
c : set if source operand was negative, cleared otherwise.
v : set if the result is negative, cleared otherwise.
Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: Scalar + Scalar \Rightarrow Scalar Register

Description: Compute the thirty-two bit sum of two scalar sources, writing the result to a scalar register.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
16-bit forms		
add S_i, S_k	add S_i to S_k , writing the result to S_k	
add $\#n, S_k$	add $\#n$ to S_k , writing the result to S_k	$0 \leq n \leq 31$
32-bit forms		
add S_i, S_j, S_k	add S_i to S_j , writing the result to S_k	
add $\#n, S_j, S_k$	add $\#n$ to S_j , writing the result to S_k	$0 \leq n \leq 31$
add $\#nn, S_k$	add $\#nn$ to S_k , writing the result to S_k	$0 \leq nn \leq 1023$
add $\#n, >>\#m, S_k$	add $\#n$ arithmetically shifted right by $\#m$, to S_k , writing the result to S_k	$0 \leq n \leq 31$ $-16 \leq m \leq 0$
add $S_i, >>\#m, S_k$	add S_i arithmetically shifted right by $\#m$, to S_k , writing the result to S_k	$-16 \leq m \leq 15$
48-bit forms		
add $\#nnnn, S_k$	add $\#nnnn$ to S_k , writing the result to S_k	$-(2^{31}) \leq nnnn \leq (2^{31}) - 1$
64-bit forms		
add $\#nnnn, S_j, S_k$	add $\#nnnn$ to S_j , writing the result to S_k	$-(2^{31}) \leq nnnn \leq (2^{31}) - 1$

Operand Values:

- S_i any scalar register r0-r31.
- S_j any scalar register r0-r31.
- S_k any scalar register r0-r31.
- $\#n$ 5-bit immediate value, zero extended to 32 bits.
- $\#nn$ 10-bit immediate value, zero extended to 32 bits.
- $\#nnnn$ 32-bit immediate value.
- $\#m$ immediate shift value.
- $>>$ shifts are arithmetic, right for positive values, left for negative values, and overflow from shift out is not detected.

Condition Codes:

- z : set if the result is zero, cleared otherwise.
- n : set if the result is negative, cleared otherwise.
- c : set if there is a carry out of the addition, cleared otherwise.
- v : set if there is signed arithmetic overflow (sign is invalid), cleared otherwise.

Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: Pixel + Pixel \Rightarrow Pixel Register (first 3 scalars of a vector)

Description: (This instruction behaves identically to the **add_sv** instruction, except that only the three lowest numbered scalars of the vector register destination are written.)

Add two pixels. Pixels consist of three 16 bit elements, taken from the 16 MSBs of the first three scalars in a vector register. Each 16 bit element of the first source is independently added to the corresponding element in the other source, and the result is written to the destination vector register in the same format. The lower 16 bits of each of the first three scalars in the vector destination are written with zeros.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
32-bit forms	
add_p <i>Vi,Vj,Vk</i>	add pixel Vi to pixel Vj, writing the result to Vk

Operand Values: Vi any vector register v0-v7, as a pixel.
Vj any vector register v0-v7, as a pixel.
Vk any vector register v0-v7, as a pixel.

Condition Codes: Unchanged by this instruction.

Function Unit: ALU

Operation: Small vector Source A + Small vector Source B \Rightarrow Vector Destination

Description: Add two small vectors. Small vectors consist of four 16 bit elements, taken from the 16 MSBs of the four scalars in a vector register. Each 16 bit element of the first source is independently added to the corresponding element in the other source, and the result is written to the destination vector register in the same format. The lower 16 bits of each scalar element of the vector destination are written with zeros.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
16-bit forms	
add_sv Vi,Vk	add small-vector Vi to small-vector Vk, writing the result to Vk
32-bit forms	
add_sv Vi,Vj,Vk	add small-vector Vi to small-vector Vj, writing the result to Vk

Operand Values: Vi any vector register v0-v7, as a small-vector.
Vj any vector register v0-v7, as a small-vector.
Vk any vector register v0-v7, as a small-vector.

Condition Codes: Unchanged by this instruction.

Function Unit: MUL

Operation: Scalar + Scalar \Rightarrow Scalar Destination

Description: Use the MUL unit to add two scalars, writing the result to a scalar destination register.

Unlike some MUL unit operations, this instruction completes in one tick, so the result may be used in the immediately following packet. Note that the MUL unit has only one write port into the register file, so there is a conflict if this instruction executes in a packet immediately following a packet which executed any of the 2-tick MUL unit instructions (**mul**, **mul_p**, **mul_sv**).

Assembler Syntax:

INSTRUCTION	DESCRIPTION
32-bit forms	
addm <i>Si, Sj, Sk</i>	add <i>Si</i> to <i>Sj</i> writing the result to <i>Sk</i>

Operand Values: *Si* any scalar register r0-r31.
Sj any scalar register r0-r31.
Sk any scalar register r0-r31.

Condition Codes: Unchanged by this instruction.

Function Unit: RCU

Operation: Data + Index \Rightarrow Index

Description: Add register or immediate data to an index register. This instruction directly uses the value of the index register and ignores the settings in the **xyctl** / **uvctl** registers.

Up to two **dec** instructions may also be encoded as bit-fields in this instruction, so that up to three RCU operations may be executed in one cycle.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
16-bit forms		
addr Si,RI *	add Si to index register RI. All 32 bits of Si and RI are used in this operation	
addr #(n<<16),RI	add #n to the integer part of index register RI, treating RI as a 16.16 number	$-16 \leq n \leq 15$
48-bit forms		
addr #nnnn,RI	add #nnnn to index register RI. All 32 bits of #nnnn and RI are used in this operation	$-(2^{31}) \leq \text{nnnn} \leq (2^{31})-1$

Restricted Forms: * Instructions marked with an asterisk share a register file read port with other function units. Only one of these instructions may be present in a given packet:

ECU jmp/jsr cc,(Si) | cc,(Si),nop

RCU mvr/addr Si,RI

ALU and/or/eor/ftst Si,>>Sj,Sk | Si,<>Sj,Sk

MUL mul Si,Sk,>>Sq,Sk | #n,Sk,>>Sq,Sk

Operand Values:

Si any scalar register r0-r31.

RI any index register rx, ry, ru, or rv.

#n 5-bit immediate value, sign extended to 16 bits.

#nnnn 32-bit immediate value.

Condition Codes: Unchanged by this instruction.

Function Unit: ALU

Operation: Scalar + Scalar + Carry Flag \Rightarrow Scalar Register

Description: Compute the thirty-two bit sum of two scalar sources along with the current value of the carry flag, writing the result to a scalar register.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
32-bit forms		
addwc <i>Si, Sj, Sk</i>	add c to Si to Sj, writing the result to Sk	
addwc <i>#n, Sj, Sk</i>	add c to #n to Sj, writing the result to Sk	$0 \leq n \leq 31$
addwc <i>#nn, Sk</i>	add c to #nn to Sk, writing the result to Sk	$0 \leq nn \leq 1023$
addwc <i>#n, >>#m, Sk</i>	add c to #n arithmetically shifted right by #m, to Sk, writing the result to Sk	$0 \leq n \leq 31$ $-16 \leq m \leq 0$
addwc <i>Si, >>#m, Sk</i>	add c to Si arithmetically shifted right by #m, to Sk, writing the result to Sk	$-16 \leq m \leq 15$
64-bit forms		
addwc <i>#nnnn, Sj, Sk</i>	add c to #nnnn to Sj, writing the result to Sk	$-(2^{31}) \leq nnnn \leq (2^{31}) - 1$

Operand Values:

- c* current value of the c flag in the cc register, zero extended to 32 bits.
- Si* any scalar register r0-r31.
- Sj* any scalar register r0-r31.
- Sk* any scalar register r0-r31.
- #n* 5-bit immediate value, zero extended to 32 bits.
- #nn* 10-bit immediate value, zero extended to 32 bits.
- #nnnn* 32-bit immediate value.
- #m* immediate shift value.
- >>* shifts are arithmetic, right for positive values, left for negative values, and overflow from shift out is not detected.

Condition Codes:

- z* : unchanged if the result is zero, cleared otherwise.
- n* : set if the result is negative, cleared otherwise.
- c* : set if there is a carry out of the addition, cleared otherwise.
- v* : set if there is signed arithmetic overflow (sign is invalid), cleared otherwise.

Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: Scalar AND Scalar \Rightarrow Scalar Register

Description: Bit-wise logical AND of two 32-bit sources, writing the result to a scalar register.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
16-bit forms		
and Si, Sk	AND Si with Sk , writing the result to Sk	
32-bit forms		
and Si, Sj, Sk	AND Si with Sj , writing the result to Sk	
and $\#n, Sj, Sk$	AND $\#n$ with Sj , write result to Sk . Useful for Lisp.	$-16 \leq n \leq 15$
and $\#n, <>\#m, Sk$	AND $\#n$ rotated right by $\#m$, with Sk , writing the result to Sk . Useful for masking in or out, a bit field.	$-16 \leq n \leq 15$ $-\infty \leq m \leq \infty$
and $\#n, >>Sj, Sk$	AND $\#n$ logically shifted right by Sj , with Sk , writing the result to Sk	$-16 \leq n \leq 15$ $-32 \leq Sj \leq 31$
and $Si, >>\#m, Sk$	AND Si logically shifted right by $\#m$, with Sk , writing the result to Sk	$-16 \leq m \leq 15$
and $Si, >>Sj, Sk *$	AND Si logically shifted right by Sj , with Sk , writing the result to Sk	$-32 \leq Sj \leq 31$
and $Si, <>Sj, Sk *$	AND Si rotated right by Sj , with Sk , writing the result to Sk	all Sj are valid
64-bit forms		
and $\#nnnn, Sj, Sk$	AND $\#nnnn$ with Sj , writing the result to Sk	$-(2^{31}) \leq nnnn \leq (2^{31})-1$
and $\#nnnn, >>Sj, Sk$	AND $\#nnnn$ logically shifted right by Sj , with Sk , writing the result to Sk	$-(2^{31}) \leq nnnn \leq (2^{31})-1$ $-32 \leq Sj \leq 31$

Restricted Forms: * Instructions marked with an asterisk share a register file read port with other function units. Only one of these instructions may be present in a given packet:

ECU jmp/jsr cc,(Si) | cc,(Si),nop

RCU mvr/addr Si,RI

ALU and/or/eor/ftst $Si, >>Sj, Sk$ | $Si, <>Sj, Sk$

MUL mul $Si, Sk, >>Sq, Sk$ | $\#n, Sk, >>Sq, Sk$

Operand Values:

- Si any scalar register r0-r31.
- Sj any scalar register r0-r31. For shifts, bits 5-0 are used, bits 31-6 ignored.
- Sk any scalar register r0-r31.
- $\#n$ 5-bit immediate value, sign extended to 32 bits.
- $\#nnnn$ 32-bit immediate value.
- $\#m$ immediate shift or rotate value.
- $>>$ shifts are logical, right for positive values, left for negative values.
- $<>$ rotates are right for positive values, left for negative values.

...continued

Condition Codes: z : set if the result is zero, cleared otherwise.
 n : set if the result is negative, cleared otherwise.
 c : unchanged.
 v : cleared.

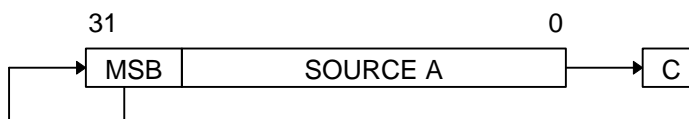
Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: Scalar Source A \gg Source B \Rightarrow Scalar Destination

Description: Arithmetically shift source A either left or right by source B, setting flags appropriately, and writing the result to destination. Only the bottom six bits of Source B are used, the high-order 26 bits are ignored. A positive shift value implies a right shift; a negative shift value implies a left shift

For right shifts, the MSB (sign bit) is shifted into the most significant bit, as shown below:



Shift into carry is always from bit 0, in other words, this flag-setting function is only valid for a shift by one. It may be used for any shift, but will always be the contents of bit 0.

For left shifts, a zero is shifted in from the right, as shown below:



Shift into carry is always from bit 31, in other words, this flag-setting function is only valid for a shift by one. It may be used for any shift, but will always be the contents of bit 31.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
32-bit forms		
as $\gg S_j, S_i, S_k$	arithmetical shift right of S_i by S_j , writing the result to S_k .	$-32 \leq S_j \leq 31$

Operand Values:

- S_i any scalar register r0-r31.
- S_j any scalar register r0-r31. Bits 5-0 are used, bits 31-6 are ignored.
- S_k any scalar register r0-r31.

Condition Codes:

- z : set if the result is zero, cleared otherwise.
- n : set if the result is negative, cleared otherwise.
- c : for right shifts ($S_j \geq 0$), c takes the value of bit 0 of Source A; for left shifts ($S_j < 0$), c takes the value of bit 31 of Source A.
- v : cleared.

Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: Scalar Source A \ll Source B \Rightarrow Scalar Destination

Description: Arithmetically shift left source A by source B, setting flags appropriately, and writing the result to destination. Arithmetic and logical shifts are identical for a left shift, and this is the same instruction as **lsl**. A zero is shifted in from the right, as shown below:



Shift into carry is always from bit 31, in other words, this flag-setting function is only valid for a shift by one. It may be used for any shift, but will always be the contents of bit 31.

A register shift-control version of **asl** is available through the **as** instruction.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
16-bit forms		
asl #m, Sk	arithmetic shift left of Sk by #m, writing the result to Sk	$0 \leq m \leq 31$
32-bit forms		
asl #m, Si, Sk	arithmetic shift left of Si by #m, writing the result to Sk	$0 \leq m \leq 31$

Operand Values:

- Si any scalar register r0-r31.
- Sk any scalar register r0-r31.
- #m immediate shift value.

Condition Codes:

- z : set if the result is zero, cleared otherwise.
- n : set if the result is negative, cleared otherwise.
- c : bit 31 of source A.
- v : cleared.

Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: Scalar Source A >> Source B \Rightarrow Scalar Destination

Description: Arithmetically shift right source A by source B, setting flags appropriately, and writing the result to destination. The MSB (sign bit) is shifted into the most significant bit, as shown below:



Shift into carry is always from bit 0, in other words, this flag-setting function is only valid for a shift by one. It may be used for any shift, but will always be the contents of bit 0.

A register shift-control version of **asr** is available through the **as** instruction.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
16-bit forms		
asr #m, Sk	arithmetic shift right of Sk by #m, writing the result to Sk	$0 \leq m \leq 31$
32-bit forms		
asr #m, Si, Sk	arithmetic shift right of Si by #m, writing the result to Sk	$0 \leq m \leq 31$

Operand Values:

- Si any scalar register r0-r31.
- Sk any scalar register r0-r31.
- #m immediate shift value.

Condition Codes:

- z : set if the result is zero, cleared otherwise.
- n : set if the result is negative, cleared otherwise.
- c : bit 0 of source A.
- v : cleared.

Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: Scalar Destination AND Mask \Rightarrow Scalar Destination

Description: Logical AND of the destination register with a bit mask which has all bits set except the one selected by the immediate operand.

This instruction is equivalent to the instruction form: **and #-2,<>#-n,Sk**.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
32-bit forms		
bclr #n,Sk	clear the selected bit in Sk, writing the result to Sk	$0 \leq n \leq 31$

Operand Values: Sk any scalar register r0-r31.
#n 5-bit immediate value.

Condition Codes: z : set if the result is zero, cleared otherwise.
n : set if the result is negative, cleared otherwise.
c : unchanged.
v : cleared.

Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: Logical shift Scalar Destination right by m or Source i, mask above bit n
 \Rightarrow Scalar Destination

Description: This instruction is used to extract an arbitrary length bit-field at any bit position of a scalar register and write the bit field back into that register aligned to bit zero.

To achieve this, the register value is logic shifted right by an immediate or scalar source register value. Then a second immediate value is used to generate a mask to clear the high bits. The result is an arbitrarily aligned, arbitrary length field which is written to the destination aligned to bit zero.

Note that the value for the mask is one less than the number of bits extracted, i.e. 0 yields one bit, 1 yields two bits, up to 31 which gives thirty-two bits.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
32-bit forms		
bits #n,>>Si,Sk	logical shift right of Sk by Si, then mask out any bits above bit n, writing the result to Sk	$0 \leq n \leq 31$ $0 \leq Si \leq 31$
bits #n,>>#m,Sk	logical shift right of Sk by #m, then mask out any bits above bit n, writing the result to Sk	$0 \leq n \leq 31$ $0 \leq m \leq 31$

Operand Values:

- Si any scalar register r0-r31. Bits 4-0 are used, bits 31-5 are ignored.
- Sk any scalar register r0-r31.
- #n 5-bit immediate value, used to generate a mask.
- #m immediate shift value.
- >> shifts are logical, right for positive values, left for negative values.

Condition Codes:

- z : set if the result is zero, cleared otherwise.
- n : set if the result is negative, cleared otherwise.
- c : unchanged.
- v : unchanged.

Other condition codes are unchanged by this instruction.

Function Unit: ECU

Operation: Conditional branch relative to the current **pcexec** program counter

Description: If the specified condition is true, then the branch is taken, otherwise the branch is not taken. A taken branch which has a **nop** operand will force two ‘dead’ cycles after executing the branch packet, then continue execution from the target address. A taken branch which does not have a **nop** operand will have no ‘dead’ cycles—the branch packet, the next two packets, and the packet at the target address will execute on successive cycles (ignoring unrelated pipeline stalls). If a branch is not taken, whether or not it has a **nop** operand, execution will continue with the next packet.

The two instruction packets after a packet containing a branch without a **nop** operand are in what is known as the “delay slots” of the branch. If such a branch is taken, any ECU instructions (**bra**, **halt**, **jmp**, **jsr**, **rti**, **rts**) in its delay slots will not be evaluated. If the branch is not taken, the delay slots execute normally. This allows multi-way branch decisions to be made in successive instruction packets.

Normally, a programmer lets the assembler choose the shortest form that will accommodate the **bra** offset, condition, and possible use of the **nop** operand.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	RANGE RESTRICTIONS ON offset = <label> – pccur
16-bit forms		
bra cc,<label>	If the cc condition is true: execute the next two packets, then continue execution at <label>. false: continue execution with the next packet. For this form only, the cc condition is restricted to one of {ne, eq, lt, le, gt, ge, c0ne, c1ne}.	$-128 \leq \text{offset} \leq 126$
bra <label>	After executing the next two packets, continue execution at <label>.	$-1024 \leq \text{offset} \leq 1022$
32-bit forms		
bra cc,<label>	If the cc condition is true: execute the next two packets, then continue execution at <label>. false: continue execution with the next packet.	$-16484 \leq \text{offset} \leq 16382$
bra cc,<label>,nop	If the cc condition is true: force two dead cycles, then continue execution at <label>. false: continue execution with the next packet.	$-16484 \leq \text{offset} \leq 16382$
48-bit forms		
bra cc,<label>	If the cc condition is true: execute the next two packets, then continue execution at <label>. false: continue execution with the next packet.	$-(2^{31}) \leq \text{offset} \leq (2^{31})-2$

...continued

INSTRUCTION	DESCRIPTION	RANGE RESTRICTIONS ON offset = <label> – pcexec
48-bit forms		
bra cc,<label>,nop	If the cc condition is true: force two dead cycles, then continue execution at <label>. false: continue execution with the next packet.	$-(2^{31}) \leq \text{offset} \leq (2^{31})-2$

Operand Values: <label> is resolved to an address by the assembler/linker. This target address is always an even number since instructions are on 16-bit boundaries. The offset between the target address and the address of the packet that contains the branch instruction is then calculated. This offset is encoded into the branch instruction, and during program execution, the branch target address is created by adding the offset value to the value in the **pcexec** register. (Note that it is illegal for the 32-bit target address to lie within a different 1 Gbyte quarter of the address space from the 32-bit **pcexec** of the branch packet.)

cc may take on any of the following values, except as noted above (if not specified, t is assumed):

cc mnemonic	condition	test
ne	Not equal	/z
eq	Equal	z
lt	Less than	(n./v) + (/n.v)
le	Less than or equal	z + (n./v) + (/n.v)
gt	Greater than	(n.v./z) + (/n./v./z)
ge	Greater than or equal	(n.v) + (/n./v)
c0ne	rc0 not equal to zero	/c0z
clne	rc1 not equal to zero	/clz
c0eq	rc0 equal to zero	c0z
cleq	rc1 equal to zero	clz
cc (hs)	Carry clear (High or same)	/c
cs (lo)	Carry set (Low)	c
vc	Overflow clear	/v
vs	Overflow set	v
mvc	Multiply overflow clear	/mv
mvs	Multiply overflow set	mv
hi	High	/c./z
ls	Low or same	c + z
pl	Plus	/n
mi	Minus	n
t	True	1
modmi	modulo RI was < zero	modmi
modpl	modulo RI was >= zero	/modmi

...continued

cc mnemonic	condition	test
modge	modulo RI was \geq range	modge
modlt	modulo RI was $<$ range	/modge
cf0lo	Coprocessor flag 0 low	/cf0
cf0hi	Coprocessor flag 0 high	cf0
cf1lo	Coprocessor flag 1 low	/cf1
cf1hi	Coprocessor flag 1 high	cf1

Condition Codes: Unchanged by this instruction.

Function Unit: none

Operation:

Description:

Assembler Syntax:

INSTRUCTION	DESCRIPTION
16-bit forms	
breakpoint	

Condition Codes: Unchanged by this instruction.

Function Unit: ALU

Operation: Scalar Destination OR Mask \Rightarrow Scalar Destination

Description: Logical OR of the destination register with a bit mask which one bit set as selected by the immediate operand.

This instruction is equivalent to the instruction: **or #1,<>#-n,Sk**.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
32-bit forms		
bset #n,Sk	set the selected bit in Sk, writing the result to Sk	$0 \leq n \leq 31$

Operand Values: Sk any scalar register r0-r31.
#n 5-bit immediate value.

Condition Codes: z : set if the result is zero, cleared otherwise.
n : set if the result is negative, cleared otherwise.
c : unchanged.
v : cleared.

Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: Test a bit of the Scalar Source \Rightarrow Flags

Description: Logical AND of a one bit mask and a 32-bit source register, without writing a result back to a register. This instruction may be used to test a bit in any register.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
16-bit forms		
btst #m, Sj	test bit #m of register Sj and set the flags accordingly	$0 \leq m \leq 31$

Operand Values: Sj any scalar register r0-r31.
#m 5-bit immediate value.

Condition Codes: z : set if the selected bit is zero, cleared otherwise.
n : set if the selected bit was bit 31 and it was not zero.
c : unchanged.
v : cleared.
Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: Scalar Source A + Scalar Source B \Rightarrow Scalar Destination K
Scalar Source A – Scalar Source B \Rightarrow Scalar Destination K+1

Description: Compute the thirty-two bit sum and difference of the two source operands, and write these to the destination half-vector. A half-vector is a pair of scalar registers on an even register boundary. The sum is written to the first register in the half-vector pair, and the difference is written to the second.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
32-bit forms	
butt <i>Si, Sj, Hk</i>	write the butterfly function result (S_j+Si , S_j-Si) to the destination half-vector <i>Hk</i>

Operand Values: *Si* any scalar register r0-r31.
Sj any scalar register r0-r31.
Hk any scalar register r0-r31, even numbers only.

Condition Codes: *z* : set if the result of the add is zero, cleared otherwise.
n : set if the result of the add is negative, cleared otherwise.
c : set if there is a carry out of the add, cleared otherwise.
v : set if there is signed arithmetic overflow of the add, cleared otherwise.
Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: Scalar - Scalar \Rightarrow NULL

Description: Subtract one scalar value from another scalar value, setting the condition codes appropriately, without any write-back of the result.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
16-bit forms		
cmp S_i, S_j	subtract S_i from S_j	
cmp $\#n, S_j$	subtract $\#n$ from S_j	$0 \leq n \leq 31$
32-bit forms		
cmp $\#nn, S_q$	subtract $\#nn$ from S_q	$0 \leq nn \leq 1023$
cmp $\#n, >>\#m, S_q$	subtract $\#n$ arithmetically shifted right by $\#m$ from S_q	$0 \leq n \leq 31$ $-16 \leq m \leq 0$
cmp $S_i, \#n$	subtract S_i from $\#n$	$0 \leq n \leq 31$
cmp $S_i, >>\#m, S_q$	subtract S_i arithmetically shifted right by $\#m$ from S_q	$-16 \leq m \leq 15$
48-bit forms		
cmp $\#nnnn, S_j$	subtract $\#nnnn$ from S_j	$-(2^{31}) \leq nnnn \leq (2^{31})-1$
64-bit forms		
cmp $S_i, \#nnnn$	subtract S_i from $\#nnnn$	$-(2^{31}) \leq nnnn \leq (2^{31})-1$

Operand Values:

- S_i any scalar register r0-r31.
- S_j any scalar register r0-r31.
- S_q any scalar register r0-r31.
- $\#n$ 5-bit immediate value, zero extended to 32 bits.
- $\#nn$ 10-bit immediate value, zero extended to 32 bits.
- $\#nnnn$ 32-bit immediate value.
- $\#m$ immediate shift value.
- $>>$ shifts are arithmetic, right for positive values, left for negative values, and overflow from shift out is not detected.

Condition Codes:

- z : set if the result is zero, cleared otherwise.
- n : set if the result is negative, cleared otherwise.
- c : set if there is a borrow out of the subtraction, cleared otherwise.
- v : set if there is signed arithmetic overflow (sign is invalid), cleared otherwise.

Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: Scalar - Scalar - Carry Condition Code \Rightarrow NULL

Description: Subtract one scalar value from another scalar value and also subtract the current value of the carry condition code bit, setting the condition codes appropriately, without any write-back of the result.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
32-bit forms		
cmpwc S_i, S_j	subtract c and S_i from S_j	
cmpwc $\#n, S_j$	subtract c and $\#n$ from S_j	$0 \leq n \leq 31$
cmpwc $\#nn, S_q$	subtract c and $\#nn$ from S_q	$0 \leq nn \leq 1023$
cmpwc $\#n, >>\#m, S_q$	subtract c and $\#n$ arithmetically shifted right by $\#m$ from S_q	$0 \leq n \leq 31$ $-16 \leq m \leq 0$
cmpwc $S_i, \#n$	subtract c and S_i from $\#n$	$0 \leq n \leq 31$
cmpwc $S_i, >>\#m, S_q$	subtract c and S_i arithmetically shifted right by $\#m$ from S_q	$-16 \leq m \leq 15$
64-bit forms		
cmpwc $\#nnnn, S_j$	subtract c and $\#nnnn$ from S_j	$-(2^{31}) \leq nnnn \leq (2^{31})-1$
cmpwc $S_i, \#nnnn$	subtract c and S_i from $\#nnnn$	$-(2^{31}) \leq nnnn \leq (2^{31})-1$

Operand Values:

- c current value of the c flag in the cc register, zero extended to 32 bits.
- S_i any scalar register r0-r31.
- S_j any scalar register r0-r31.
- S_q any scalar register r0-r31.
- $\#n$ 5-bit immediate value, zero extended to 32 bits.
- $\#nn$ 10-bit immediate value, zero extended to 32 bits.
- $\#nnnn$ 32-bit immediate value.
- $\#m$ immediate shift value.
- $>>$ shifts are arithmetic, right for positive values, left for negative values, and overflow from shift out is not detected.

Condition Codes:

- z : unchanged if the result is zero, cleared otherwise.
- n : set if the result is negative, cleared otherwise.
- c : set if there is a borrow out of the subtraction, cleared otherwise.
- v : set if there is signed arithmetic overflow (sign is invalid), cleared otherwise.

Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: Scalar Source \Rightarrow Scalar Destination

Description: Copy the source register to the destination register through the ALU, by adding implied immediate zero to it. This instruction allows the ALU to be used to for a register copy in parallel with other operations, which could include another register to register copy through the memory unit (MEM).

This instruction is encoded in 16 bits, and is equivalent to the 32-bit instruction form **add #0,Si,Sk**. It is called COPY to distinguish it from the MV instructions, which use the memory unit, not the ALU; and do not set the flags, which this does.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
16-bit forms	
copy Si,Sk	copy register Si to register Sk

Operand Values: Si any scalar register r0-r31.
Sk any scalar register r0-r31.

Condition Codes: z : set if the register is zero, cleared otherwise.
n : set if the register is negative, cleared otherwise.
c : unchanged.
v : cleared.

Other condition codes are unchanged by this instruction.

Function Unit: RCU sub-instruction

Operation: $c - 1 \Rightarrow c$

Description: Decrement register rc0 or rc1 by 1. If the register is already zero it remains zero.

This instruction is not encoded as a full instruction. Instead, either or both of these decrement operations are encoded as a bit field in any register unit instruction. If there is no other register unit instruction, the assembler will encode a special **dec_only** form which has no other function besides encoding up to two decrement instructions.

The condition codes are valid, and may be tested, in the cycle after the decrement instruction.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
0/16-bit forms	
dec rc0	decrement register rc0, unless it is zero
dec rc1	decrement register rc1, unless it is zero

Condition Codes: c0z : set if rc0 is zero, cleared otherwise.

c1z : set if rc1 is zero, cleared otherwise.

Other condition codes are unchanged by this instruction.

Function Unit: MUL

Operation: Sum of products of (Small vector Source A * Small vector Source B)
⇒ Scalar Destination

Description: Four parallel 16x16 signed integer multiply operations are performed, followed by three additions, to give the sum of the products of each of the four elements of the small vectors. One of the sources is always a small vector. The other source may either be another small vector or a scalar. The result is shifted as defined by the instruction, and written to the scalar destination.

The 32-bit products are summed, and the sum is then shifted as defined by the instruction. Overflow, from the addition, is not detected.

This operation completes in two clock cycles, so the result is not valid during the following clock cycle. However, you cannot rely on the previous value still being in the destination register in the following clock cycle, because if the MPE stalls for any reason then it will be over-written.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
32-bit forms	
dotp $Si, Vj, >>svshift, Sk$	form a small vector by repeating the 16 most significant bits of scalar register Si four times, multiply it by all four elements of small vector Vj , sum the products, shift the result by an amount determined by the svshift register, and write the result to scalar Sk
dotp $Si, Vj, >>\#m, Sk$	form a small vector by repeating the 16 most significant bits of scalar register Si four times, multiply it by all four elements of small vector Vj , sum the products, shift the result by an amount determined by the immediate value (legal values are {16,24,32,30}, see below), and write the result to scalar Sk
dotp $Vi, Vj, >>svshift, Sk$	multiply all four elements of small vector Vi by all four elements of small vector Vj , sum the products, shift the result by an amount determined by the svshift register, and write the result to scalar Sk
dotp $Vi, Vj, >>\#m, Sk$	multiply all four elements of small vector Vi by all four elements of small vector Vj , sum the products, shift the result by an amount determined by the immediate value (legal values are {16,24,32,30}, see below), and write the result to scalar Sk

...continued

Operand Values:

- Si any scalar register r0-r31. Bits 31-16 are used, bits 15-0 are ignored.
- Sk any scalar register r0-r31.
- Vi any vector register v0-v7, as a small-vector.
- Vj any vector register v0-v7, as a small-vector.
- >> the value encoded into #m, or encoded in the the **svshift** register, determines the final shift amount, as follows:

Encoding for #m	Encoding for svshift	Description
16	0	the 32-bit sum of products value is shifted left by 16, filling with zeros. This produces a 16.16 scalar result when the input small-vector elements are considered to be in 16.0 format.
24	1	the 32-bit sum of products value is shifted left by 8, filling with zeros. This produces an 8.24 scalar result when the input small-vector elements are considered to be in 8.8 format.
32	2	the 32-bit sum of products value are used directly, and are not shifted. This produces a 0.32 scalar result when the input small-vector elements are considered to be in 0.16 format.
30	3	the 32-bit sum of products value is shifted left by 2, filling with zeros. This produces a 2.30 scalar result when the input small-vector elements are considered to be in 2.14 format.

Condition Codes: Unchanged by this instruction.

Function Unit: ALU

Operation: Scalar Exclusive-OR Scalar \Rightarrow Scalar Register

Description: Bit-wise logical exclusive-OR of two 32-bit sources, writing the result to a scalar register.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
16-bit forms		
eor Si,Sk	exclusive-OR Si with Sk, writing the result to Sk	
eor #n,Sk	exclusive-OR #n with Sk, writing the result to Sk	$-16 \leq n \leq 15$
32-bit forms		
eor Si,Sj,Sk	exclusive-OR Si with Sj, writing the result to Sk	
eor #n,Sj,Sk	exclusive-OR #n with Sj, writing the result to Sk.	$-16 \leq n \leq 15$
eor #n,<>#m,Sk	exclusive-OR #n rotated right by #m, with Sk, writing the result to Sk. May be used to mask in or out, a bit field.	$-16 \leq n \leq 15$ $-\infty \leq m \leq \infty$
eor #n,>>Sj,Sk	exclusive-OR #n logically shifted right by Sj, with Sk, writing the result to Sk	$-16 \leq n \leq 15$ $-32 \leq Sj \leq 31$
eor Si,>>#m,Sk	exclusive-OR Si logically shifted right by #m, with Sk, writing the result to Sk	$-16 \leq m \leq 15$
eor Si,>>Sj,Sk *	exclusive-OR Si logically shifted right by Sj, with Sk, writing the result to Sk	$-32 \leq Sj \leq 31$
eor Si,<>Sj,Sk *	exclusive-OR Si rotated right by Sj, with Sk, writing the result to Sk	all Sj are valid
48-bit forms		
eor #nnnn,Sk	exclusive-OR #nnnn with Sk, writing the result to Sk	$-(2^{31}) \leq nnnn \leq (2^{31})-1$
64-bit forms		
eor #nnnn,Sj,Sk	exclusive-OR #nnnn with Sj, writing the result to Sk	$-(2^{31}) \leq nnnn \leq (2^{31})-1$
eor #nnnn,>>Sj,Sk	exclusive-OR #nnnn logically shifted right by Sj, with Sk, writing the result to Sk	$-(2^{31}) \leq nnnn \leq (2^{31})-1$ $-32 \leq Sj \leq 31$

Restricted Forms: * Instructions marked with an asterisk share a register file read port with other function units. Only one of these instructions may be present in a given packet:

ECU jmp/jsr cc,(Si) | cc,(Si),nop

RCU mvr/addr Si,RI

ALU and/or/eor/ftst Si,>>Sj,Sk | Si,<>Sj,Sk

MUL mul Si,Sk,>>Sq,Sk | #n,Sk,>>Sq,Sk

Operand Values:

- Si any scalar register r0-r31.
- Sj any scalar register r0-r31. For shifts, bits 5-0 are used, bits 31-6 ignored.
- Sk any scalar register r0-r31.
- #n 5-bit immediate value, sign extended to 32 bits.

...continued

#nnnn 32-bit immediate value.

#m immediate shift or rotate value.

>> shifts are logical, right for positive values, left for negative values.

<> rotates are right for positive values, left for negative values.

Condition Codes: z : set if the result is zero, cleared otherwise.
n : set if the result is negative, cleared otherwise.
c : unchanged.
v : cleared.

Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: Scalar Source A AND Scalar Source B \Rightarrow Flags

Description: Bit-wise logical AND of two 32-bit sources, without writing the result back to a register. This instruction may be used to test a bit-range in any register, or to perform a bit compare on two registers.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
32-bit forms		
ftst Si,Sj	AND Si with Sj	
ftst #n,Sj	AND #n with Sj	$-16 \leq n \leq 15$
ftst #n,<>#m,Sq	AND #n rotated right by #m, with Sq	$-16 \leq n \leq 15$ $-\infty \leq m \leq \infty$
ftst #n,>>Sj,Sq	AND #n logically shifted right by Sj, with Sq	$-16 \leq n \leq 15$ $-32 \leq Sj \leq 31$
ftst Si,>>#m,Sq	AND Si logically shifted right by #m, with Sq	$-16 \leq n \leq 15$
ftst Si,>>Sj,Sq *	AND Si logically shifted right by Sj, with Sq	$-32 \leq Sj \leq 31$
ftst Si,<>Sj,Sq *	AND Si rotated right by Sj, with Sq	all Sj are valid
64-bit forms		
ftst #nnnn,Sj	AND #nnnn with Sj	$-(2^{31}) \leq nnnn \leq (2^{31})-1$
ftst #nnnn,>>Sj,Sq	AND #nnnn logically shifted right by Sj, with Sq	$-(2^{31}) \leq nnnn \leq (2^{31})-1$ $-32 \leq Sj \leq 31$

Restricted Forms: * Instructions marked with an asterisk share a register file read port with other function units. Only one of these instructions may be present in a given packet:

ECU jmp/jsr cc,(Si) | cc,(Si),nop

RCU mvr/addr Si,RI

ALU and/or/eor/ftst Si,>>Sj,Sk | Si,<>Sj,Sk

MUL mul Si,Sk,>>Sq,Sk | #n,Sk,>>Sq,Sk

Operand Values:

- Si any scalar register r0-r31.
- Sj any scalar register r0-r31. For shifts, bits 5-0 are used, bits 31-6 ignored.
- Sq any scalar register r0-r31.
- #n 5-bit immediate value, sign extended to 32 bits.
- #nnnn 32-bit immediate value.
- #m immediate shift or rotate value.
- >> shifts are logical, right for positive values, left for negative values.
- <> rotates are right for positive values, left for negative values.

Condition Codes:

- z : set if the result is zero, cleared otherwise.
- n : set if the result is negative, cleared otherwise.
- c : unchanged.
- v : cleared.

Other condition codes are unchanged by this instruction.

Function Unit: ECU

Operation: Halt program execution

Description: Halt the MPE, clearing the **mpeGo** bit in the **mpectl** register. When the MPE stops, **pcexec** will contain the address of the packet that would otherwise have executed if the halt instruction had not been present.

(In the unlikely event that the **excepHaltEn_halt** bit has been cleared in the **excephalten** register, then a halt instruction will not actually halt the MPE. Instead, the exception bit in the intsrc register will be set, and the MPE will continue executing.)

Assembler Syntax:

INSTRUCTION	DESCRIPTION
16-bit forms	
halt	halt program execution

Condition Codes: Unchanged by this instruction.

Function Unit: ECU

Operation: Conditional jump to an absolute address

Description: If the specified condition is true, then the jump is taken, otherwise the jump is not taken. A taken jump which has a **nop** operand will force two ‘dead’ cycles after executing the jump packet, then continue execution from the target address. A taken jump which does not have a **nop** operand will have no ‘dead’ cycles—the jump packet, the next two packets, and the packet at the target address will execute on successive cycles (ignoring unrelated pipeline stalls). If a jump is not taken, whether or not it has a **nop** operand, execution will continue with the next packet.

The two instruction packets after a packet containing a jump without a **nop** operand are in what is known as the “delay slots” of the jump. If such a jump is taken, any ECU instructions (**bra**, **halt**, **jmp**, **jsr**, **rti**, **rts**) in its delay slots will not be evaluated. If the jump is not taken, the delay slots execute normally. This allows multi-way jump decisions to be made in successive instruction packets.

For a **jmp** to an immediate <label>, the programmer normally lets the assembler choose the shortest form that will accomodate the target address.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	TARGET ADDRESS
32-bit forms		
<code>jmp cc,(Si) *</code>	If the cc condition is true: execute the next two packets, then continue execution at address Si. false: continue execution with the next packet.	32-bit absolute address
<code>jmp cc,(Si),nop *</code>	If the cc condition is true: force two dead cycles, then continue execution at address Si. false: continue execution with the next packet.	32-bit absolute address
<code>jmp cc,<label></code>	If the cc condition is true: execute the next two packets, then continue execution at <label>. false: continue execution with the next packet.	any address in the first 16K bytes of local IROM or IRAM
<code>jmp cc,<label>,nop</code>	If the cc condition is true: force two dead cycles, then continue execution at <label>. false: continue execution with the next packet.	any address in the first 16K bytes of local IROM or IRAM
64-bit forms		
<code>jmp cc,<label></code>	If the cc condition is true: execute the next two packets, then continue execution at <label>. false: continue execution with the next packet.	32-bit absolute address
<code>jmp cc,<label>,nop</code>	If the cc condition is true: force two dead cycles, then continue execution at <label>. false: continue execution with the next packet.	32-bit absolute address

...continued

Restricted Forms: * Instructions marked with an asterisk share a register file read port with other function units. Only one of these instructions may be present in a given packet:

ECU jmp/jsr cc,(Si) | cc,(Si),nop

RCU mvr/addr Si,RI

ALU and/or/eor/ftst Si,>>Sj,Sk | Si,<>Sj,Sk

MUL mul Si,Sk,>>Sq,Sk | #n,Sk,>>Sq,Sk

Operand Values: Si is any any scalar register r0-r31, as an absolute 32-bit address.

<label> is resolved to an address by the assembler/linker. This target address is always an even number since instructions are on 16-bit boundaries.

cc may take on any of the following values (if not specied, t is assumed):

cc mnemonic	condition	test
ne	Not equal	/z
eq	Equal	z
lt	Less than	(n./v) + (/n.v)
le	Less than or equal	z + (n./v) + (/n.v)
gt	Greater than	(n.v./z) + (/n./v./z)
ge	Greater than or equal	(n.v) + (/n./v)
c0ne	rc0 not equal to zero	/c0z
clne	rc1 not equal to zero	/c1z
c0eq	rc0 equal to zero	c0z
cleq	rc1 equal to zero	c1z
cc (hs)	Carry clear (High or same)	/c
cs (lo)	Carry set (Low)	c
vc	Overflow clear	/v
vs	Overflow set	v
mvc	Multiply overflow clear	/mv
mvs	Multiply overflow set	mv
hi	High	/c./z
ls	Low or same	c + z
pl	Plus	/n
mi	Minus	n
t	True	1
modmi	modulo RI was < zero	modmi
modpl	modulo RI was >= zero	/modmi
modge	modulo RI was >= range	modge
modlt	modulo RI was < range	/modge
cf0lo	Coprocessor flag 0 low	/cf0
cf0hi	Coprocessor flag 0 high	cf0
cf1lo	Coprocessor flag 1 low	/cf1
cf1hi	Coprocessor flag 1 high	cf1

Condition Codes: Unchanged by this instruction.

Function Unit: ECU

Operation: Conditional jump to an absolute address, also copying a return address to **rz**

Description: If the specified condition is true, then the **jsr** jump is taken, otherwise the jump is not taken. A taken jump which has a **nop** operand will force two ‘dead’ cycles after executing the jump packet, then continue execution from the target address. A taken jump which does not have a **nop** operand will have no ‘dead’ cycles—the jump packet, the next two packets, and the packet at the target address will execute on successive cycles (ignoring unrelated pipeline stalls). If a jump is not taken, whether or not it has a **nop** operand, execution will continue with the next packet.

The two instruction packets after a packet containing a jump without a **nop** operand are in what is known as the “delay slots” of the jump. If such a jump is taken, any ECU instructions (**bra**, **halt**, **jmp**, **jsr**, **rti**, **rts**) in its delay slots will not be evaluated. If the jump is not taken, the delay slots execute normally. This allows multi-way jump decisions to be made in successive instruction packets.

When a **jsr** is taken, the address of the next unexecuted instruction is copied to the **rz** register, so that a later **rts** instruction can return to the proper address.

For a **jsr** to an immediate <label>, the programmer normally lets the assembler choose the shortest form that will accomodate the target address.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	TARGET ADDRESS
32-bit forms		
<code>jsr cc,(Si) *</code>	If the cc condition is true: copy pcfethnext to rz , execute the next two packets, continue execution at address Si. false: continue execution with the next packet.	32-bit absolute address
<code>jsr cc,(Si),nop *</code>	If the cc condition is true: copy pcroute into rz , force two dead cycles, then continue execution at address Si. false: continue execution with the next packet.	32-bit absolute address
<code>jsr cc,<label></code>	If the cc condition is true: copy pcfethnext to rz , execute the next two packets, then continue execution at <label>. false: continue execution with the next packet.	any address in the first 16K bytes of local IROM or IRAM
<code>jsr cc,<label>,nop</code>	If the cc condition is true: copy pcroute into rz , force two dead cycles, then continue execution at <label>. false: continue execution with the next packet.	any address in the first 16K bytes of local IROM or IRAM
64-bit forms		
<code>jsr cc,<label></code>	If the cc condition is true: copy pcfethnext to rz , execute the next two packets, then continue execution at <label>. false: continue execution with the next packet.	32-bit absolute address

...continued

INSTRUCTION	DESCRIPTION	TARGET ADDRESS
64-bit forms		
jsr cc,<label>,nop	If the cc condition is true: copy pcroute into rz , force two dead cycles, then continue execution at <lable>. false: continue execution with the next packet.	32-bit absolute address

Restricted Forms: * Instructions marked with an asterisk share a register file read port with other function units. Only one of these instructions may be present in a given packet:

ECU jmp/jsr cc,(Si) | cc,(Si),nop

RCU mvr/addr Si,RI

ALU and/or/eor/ftst Si,>>Sj,Sk | Si,<>Sj,Sk

MUL mul Si,Sk,>>Sq,Sk | #n,Sk,>>Sq,Sk

Operand Values: Si is any any scalar register r0-r31, as an absolute 32-bit address.

<label> is resolved to an address by the assembler/linker. This target address is always an even number since instructions are on 16-bit boundaries.

pcfetchnext is the value that would have gone into the **pcfetch** register after the jump packet finished executing, had the jump not been taken.

cc may take on any of the following values (if not specied, t is assumed):

cc mnemonic	condition	test
ne	Not equal	/z
eq	Equal	z
lt	Less than	(n./v) + (/n.v)
le	Less than or equal	z + (n./v) + (/n.v)
gt	Greater than	(n.v./z) + (/n./v./z)
ge	Greater than or equal	(n.v) + (/n./v)
c0ne	rc0 not equal to zero	/c0z
c1ne	rc1 not equal to zero	/c1z
c0eq	rc0 equal to zero	c0z
c1eq	rc1 equal to zero	c1z
cc (hs)	Carry clear (High or same)	/c
cs (lo)	Carry set (Low)	c
vc	Overflow clear	/v
vs	Overflow set	v
mvc	Multiply overflow clear	/mv
mvs	Multiply overflow set	mv
hi	High	/c./z
ls	Low or same	c + z
pl	Plus	/n
mi	Minus	n
t	True	1
modmi	modulo RI was < zero	modmi
modpl	modulo RI was >= zero	/modmi

...continued

cc mnemonic	condition	test
modge	modulo RI was \geq range	modge
modlt	modulo RI was $<$ range	/modge
cf0lo	Coprocessor flag 0 low	/cf0
cf0hi	Coprocessor flag 0 high	cf0
cf1lo	Coprocessor flag 1 low	/cf1
cf1hi	Coprocessor flag 1 high	cf1

Condition Codes: Unchanged by this instruction.

Function Unit: MEM

Operation: Byte Data \Rightarrow Register

Description: Load a byte value into bits 31-24 of a scalar register, setting bits 23-0 to zero. The effective address for the load may be on any byte boundary.

The load instruction completes in two cycles, which means the loaded data is guaranteed to be valid for use by instructions in the second packet after the packet containing the load instruction. During execution of the first packet after the load packet, the contents of the target register cannot be relied upon and must not be referenced.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
32-bit forms	
ld_b (Si), Sk	load byte from address Si into register Sk
ld_b <label>, Sk	load byte from address <label> into register Sk
ld_b (xy), Sk	load byte from bilinear address (xy) into register Sk (only data type 8 is valid)
ld_b (uv), Sk	load byte from bilinear address (uv) into register Sk (only data type 8 is valid)

Operand Values:

- Si any scalar register r0-r31, as an absolute 32-bit address.
- Sk any scalar register r0-r31.
- (xy) bilinear address formed from the **rx**, **ry**, **xybase**, and **xyctl** registers.
- (uv) bilinear address formed from the **ru**, **rv**, **uvbase**, and **uvctl** registers.
- <label> is resolved to an address by the assembler/linker. The instruction encoding for this immediate address value restricts it to being on a byte boundary within an 11-bit offset in bytes (bits 10 to 0 of the address) above any of the following base values:
 - \$2000_0000 base of local dtrom
 - \$2010_0000 base of local dtrom
 - \$2050_0000 base of local control registers

Condition Codes: Unchanged by this instruction.

Function Unit: MEM

Operation: Packed Pixel Data \Rightarrow First 3 scalars of a Vector Register

Description: Load a pixel value into the first three scalars (three lowest numbered) of a vector register. See the ‘MPE Data Types’ section for a full discussion of the behavior of **ld_p** for each data type. The effective address for the load must be on the selected pixel size boundary, with the appropriate number of least significant bits equal to zero.

The load instruction completes in two cycles, which means the loaded data is guaranteed to be valid for use by instructions in the second packet after the packet containing the load instruction. During execution of the first packet after the load packet, the contents of the target register cannot be relied upon and must not be referenced.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
32-bit forms	
ld_p (Si),Vk	load pixel from address Si into register Vk, transforming the data according to the settings in the linpixctl register (only data types 0 to 6 are valid)
ld_p <label>,Vk	load pixel from address <label> into register Vk, transforming the data according to the settings in the linpixctl register (only data types 0 to 6 are valid)
ld_p (xy),Vk	load pixel from bilinear address (xy) into register Vk, transforming the data according to the settings in the xyctl register (only data types 0 to 6 are valid)
ld_p (uv),Vk	load pixel from bilinear address (uv) into register Vk, transforming the data according to the settings in the uvctl register (only data types 0 to 6 are valid)

Operand Values:

- Si any scalar register r0-r31, as an absolute 32-bit address.
- Vk any vector register v0-v7, as a pixel value.
- (xy) bilinear address formed from the **rx**, **ry**, **xybase**, and **xyctl** registers.
- (uv) bilinear address formed from the **ru**, **rv**, **uvbase**, and **uvctl** registers.
- <label> is resolved to an address by the assembler/linker. The instruction encoding for this immediate address value restricts it to being on a word boundary within an 11-bit offset in words (bits 11 to 1 of the address) above any of the following base values:
 - \$2000_0000 base of local dtrom
 - \$2010_0000 base of local dtram
 - \$2050_0000 base of local control registers

Condition Codes: Unchanged by this instruction.

Function Unit: MEM

Operation: Packed Pixel plus Z Data \Rightarrow Vector Register

Description: Load a pixel value with Z into the four scalars of a vector register. See the ‘MPE Data Types’ section for a full discussion of the behavior of **ld_pz** for each data type. The effective address for the load must be on the selected pixel size boundary, with the appropriate number of least significant bits equal to zero.

The load instruction completes in two cycles, which means the loaded data is guaranteed to be valid for use by instructions in the second packet after the packet containing the load instruction. During execution of the first packet after the load packet, the contents of the target register cannot be relied upon and must not be referenced.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
32-bit forms	
ld_pz (Si),Vk	load pixel plus Z from address Si into register Vk, transforming the data according to the settings in the linpixctl register (only data types 0 to 6 are valid)
ld_pz <label>,Vk	load pixel plus Z from address <label> into register Vk, transforming the data according to the settings in the linpixctl register (only data types 0 to 6 are valid)
ld_pz (xy),Vk	load pixel plus Z from bilinear address (xy) into register Vk, transforming the data according to the settings in the xyctl register (only data types 0 to 6 are valid)
ld_pz (uv),Vk	load pixel plus Z from bilinear address (uv) into register Vk, transforming the data according to the settings in the uvctl register (only data types 0 to 6 are valid)

Operand Values:

- Si any scalar register r0-r31, as an absolute 32-bit address.
- Vk any vector register v0-v7, as a pixel plus Z value.
- (xy) bilinear address formed from the **rx**, **ry**, **xybase**, and **xyctl** registers.
- (uv) bilinear address formed from the **ru**, **rv**, **uvbase**, and **uvctl** registers.
- <label> is resolved to an address by the assembler/linker. The instruction encoding for this immediate address value restricts it to being on a word boundary within an 11-bit offset in words (bits 11 to 1 of the address) above any of the following base values:
 - \$2000_0000 base of local dtrom
 - \$2010_0000 base of local dtram
 - \$2050_0000 base of local control registers

Condition Codes: Unchanged by this instruction.

Function Unit: MEM

Operation: Scalar Data \Rightarrow Register

Description: Load a scalar value into a scalar register. The effective address for the load may be on any scalar boundary, and the least significant 2 bits will be ignored.

The load instruction completes in two cycles, which means the loaded data is guaranteed to be valid for use by instructions in the second packet after the packet containing the load instruction. During execution of the first packet after the load packet, the contents of the target register cannot be relied upon and must not be referenced.

The **ld_io** instruction is a synonym for **ld_s**. The **ld_io** form may be found in some old software written when it used to be a separate form.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
16-bit forms	
ld_s (Si), Sk	load scalar from address Si, into register Sk
ld_s <labelA>, Sk	load scalar from address <labelA>, into register Sk
32-bit forms	
ld_s <labelB>, Sk	load scalar from address <labelB>, into register Sk
ld_s (xy), Sk	load scalar from bilinear address (xy), into register Sk (only data type A is valid)
ld_s (uv), Sk	load scalar from bilinear address (uv), into register Sk (only data type A is valid)

Operand Values:

- Si any scalar register r0-r31, as an absolute 32-bit address.
- Sk any scalar register r0-r31.
- (xy) bilinear address formed from the **rx**, **ry**, **xybase**, and **xyctl** registers.
- (uv) bilinear address formed from the **ru**, **rv**, **uvbase**, and **uvctl** registers.
- <labelA> is resolved to an address by the assembler/linker. The instruction encoding for this immediate address value restricts it to being on a vector boundary within a 5-bit offset in vectors (bits 8 to 4 of the address) above the following base value:
\$2050_0000 base of local control registers
- <labelB> is resolved to an address by the assembler/linker. The instruction encoding for this immediate address value restricts it to being on a scalar boundary within a 11-bit offset in scalars (bits 12 to 2 of the address) above any of the following base values:
\$2000_0000 base of local dtrom
\$2010_0000 base of local dtram
\$2050_0000 base of local control registers

Condition Codes: Unchanged by this instruction.

Function Unit: MEM

Operation: Packed Small Vector Data \Rightarrow Vector Register

Description: Load a small-vector value into a vector register. The effective address for the load may be on any 8-byte boundary, and the least significant 3 bits will be ignored.

The load instruction completes in two cycles, which means the loaded data is guaranteed to be valid for use by instructions in the second packet after the packet containing the load instruction. During execution of the first packet after the load packet, the contents of the target register cannot be relied upon and must not be referenced.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
32-bit forms	
ld_sv (Si),Vk	load small-vector from address Si, into register Vk
ld_sv <label>,Vk	load small-vector from address <label>, into register Vk
ld_sv (xy),Vk	load small-vector from bilinear address (xy), into register Vk (only data type C is valid)
ld_sv (uv),Vk	load small-vector from bilinear address (uv), into register Vk (only data type C is valid)

Operand Values:

- Si any scalar register r0-r31, as an absolute 32-bit address.
- Vk any vector register v0-v7.
- (xy) bilinear address formed from the **rx**, **ry**, **xybase**, and **xyctl** registers.
- (uv) bilinear address formed from the **ru**, **rv**, **uvbase**, and **uvctl** registers.
- <label> is resolved to an address by the assembler/linker. The instruction encoding for this immediate address value restricts it to being on a 8-byte boundary within a 11-bit offset in scalars (bits 13 to 3 of the address) above any of the following base values:
 - \$2000_0000 base of local dtrom
 - \$2010_0000 base of local dtram
 - \$2050_0000 base of local control registers

Condition Codes: Unchanged by this instruction.

Function Unit: MEM

Operation: Vector Data \Rightarrow Vector Register

Description: Load a vector value into a vector register. The effective address for the load may be on any 16-byte boundary, and the least significant 4 bits will be ignored.

The load instruction completes in two cycles, which means the loaded data is guaranteed to be valid for use by instructions in the second packet after the packet containing the load instruction. During execution of the first packet after the load packet, the contents of the target register cannot be relied upon and must not be referenced.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
32-bit forms	
ld_v (Si),Vk	load vector from address Si, into register Vk
ld_v <label>,Vk	load vector from address <label>, into register Vk
ld_v (xy),Vk	load vector from bilinear address (xy), into register Vk (only data type D is valid)
ld_v (uv),Vk	load vector from bilinear address (uv), into register Vk (only data type D is valid)

Operand Values:

- Si any scalar register r0-r31, as an absolute 32-bit address.
- Vk any vector register v0-v7.
- (xy) bilinear address formed from the **rx**, **ry**, **xybase**, and **xyctl** registers.
- (uv) bilinear address formed from the **ru**, **rv**, **uvbase**, and **uvctl** registers.
- <label> is resolved to an address by the assembler/linker. The instruction encoding for this immediate address value restricts it to being on a 16-byte boundary within a 11-bit offset in scalars (bits 14 to 4 of the address) above any of the following base values:
 - \$2000_0000 base of local dtrom
 - \$2010_0000 base of local dtram
 - \$2050_0000 base of local control registers

Condition Codes: Unchanged by this instruction.

Function Unit: MEM

Operation: Word Data \Rightarrow Register

Description: Load a word value into bits 31-16 of a scalar register, setting bits 15-0 to zero. The effective address for the load may be on any 2-byte boundary, and the least significant bit will be ignored.

The load instruction completes in two cycles, which means the loaded data is guaranteed to be valid for use by instructions in the second packet after the packet containing the load instruction. During execution of the first packet after the load packet, the contents of the target register cannot be relied upon and must not be referenced.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
32-bit forms	
ld_w (Si), Sk	load word from address Si, into register Sk
ld_w <label>, Sk	load word from address <label>, into register Sk
ld_w (xy), Sk	load word from bilinear address (xy), into register Sk (only data type 9 is valid)
ld_w (uv), Sk	load word from bilinear address (uv), into register Sk (only data type 9 is valid)

Operand Values:

- Si any scalar register r0-r31, as an absolute 32-bit address.
- Sk any scalar register r0-r31.
- (xy) bilinear address formed from the **rx**, **ry**, **xybase**, and **xyctl** registers.
- (uv) bilinear address formed from the **ru**, **rv**, **uvbase**, and **uvctl** registers.
- <label> is resolved to an address by the assembler/linker. The instruction encoding for this immediate address value restricts it to being on a 2-byte boundary within a 11-bit offset in scalars (bits 11 to 1 of the address) above any of the following base values:
 - \$2000_0000 base of local dtrom
 - \$2010_0000 base of local dtram
 - \$2050_0000 base of local control registers

Condition Codes: Unchanged by this instruction.

Function Unit: ALU

Operation: Scalar Source A >> Source B \Rightarrow Scalar Destination

Description: Logically shift Source A either left or right by Source B, setting flags appropriately, and writing the result to destination. Only the bottom six bits of Source B are used, the high-order 26 bits are ignored. A positive shift value implies a right shift, a negative shift value implies a left shift

For right shifts, zero is shifted into the most significant bit, as shown below:



Shift into carry is always from bit 0, in other words, this flag-setting function is only valid for a shift by one. It may be used for any shift, but will always be the contents of bit 0.

For left shifts, a zero is shifted in from the right, as shown below:



Shift into carry is always from bit 31, in other words, this flag-setting function is only valid for a shift by one. It may be used for any shift, but will always be the contents of bit 31.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
32-bit forms		
ls >>Sj, Si, Sk	logical shift right of Si by Sj, writing the result to Sk	$-32 \leq S_j \leq 31$

Operand Values:

- Si any scalar register r0-r31.
- Sj any scalar register r0-r31. Bits 5-0 are used, bits 31-6 are ignored.
- Sk any scalar register r0-r31.

Condition Codes:

- z : set if the result is zero, cleared otherwise.
- n : set if the result is negative, cleared otherwise.
- c : for right shifts ($S_j \geq 0$), c takes the value of bit 0 of Source A;
for left shifts ($S_j < 0$), c takes the value of bit 31 of Source A.
- v : cleared.

Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: Scalar Source A << Source B \Rightarrow Scalar Destination

Description: Logically shift left Source A by Source B, setting flags appropriately, and writing the result to destination. Arithmetic and logical shifts are identical for a left shift, and this is the same instruction as **asl**. A zero is shifted in from the right, as shown below:



Shift into carry is always from bit 31, in other words, this flag-setting function is only valid for a shift by one. It may be used for any shift, but will always be the contents of bit 31.

A register shift-control version of **lsl** is available through the **ls** instruction.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
16-bit forms		
<code>lsl #m, Sk</code>	logical shift left of Sk by #m, writing the result to Sk	$0 \leq m \leq 31$
32-bit forms		
<code>lsl #m, Si, Sk</code>	logical shift left of Si by #m, writing the result to Sk	$0 \leq m \leq 31$

Operand Values: Si any scalar register r0-r31.
Sk any scalar register r0-r31.
#m immediate shift value.

Condition Codes: z : set if the result is zero, cleared otherwise.
n : set if the result is negative, cleared otherwise.
c : bit 31 of source A.
v : cleared.

Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: Scalar Source A >> Source B \Rightarrow Scalar Destination

Description: Logically shift right Source A by Source B, setting flags appropriately, and writing the result to destination. A zero is shifted into the most significant bit, as shown below:



Shift into carry is always from bit 0, in other words, this flag-setting function is only valid for a shift by one. It may be used for any shift, but will always be the contents of bit 0.

A register shift-control version of **lsr** is available through the **ls** instruction.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
16-bit forms		
<code>lsr #m, Sk</code>	logical shift right of Sk by #m, writing the result to Sk	$0 \leq m \leq 31$
32-bit forms		
<code>lsr #m, Si, Sk</code>	logical shift right of Si by #m, writing the result to Sk	$0 \leq m \leq 31$

Operand Values:

- Si any scalar register r0-r31.
- Sk any scalar register r0-r31.
- #m immediate shift value.

Condition Codes:

- z : set if the result is zero, cleared otherwise.
- n : set if the result is negative, cleared otherwise.
- c : bit 0 of source A.
- v : cleared.

Other condition codes are unchanged by this instruction.

Function Unit: MEM

Operation: Mirror (Scalar Source) \Rightarrow Scalar Destination

Description: Move scalar data from register to register, reversing the bit ordering. Bit 31 goes to bit 0, bit 30 to bit 1, and so on to bit 0 going to bit 31.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
32-bit forms	
mirror <i>Sj,Sk</i>	mirror scalar data from <i>Sj</i> , writing the result to <i>Sk</i>

Operand Values: *Sj* any scalar register r0-r31.
Sk any scalar register r0-r31.

Condition Codes: Unchanged by this instruction.

Function Unit: RCU

Operation: IF (Index >= Range) Index – Range ⇒ Index
ELSE IF (Index < 0) Index + Range ⇒ Index
ELSE Index ⇒ Index

Description: Compare the integer part of the specified index register to its corresponding range from the **xyrange** or **uvrange** register, and also compare it to zero. If it is greater than or equal to the range, then subtract the range value from the index register. If it is less than zero, then add the range value to the index register. The fractional bits of the index register are unchanged by this instruction.

The **range** instruction performs the same operation as **modulo**, except that it only affects the condition code flags, without changing the index register.

Up to two DEC instructions may be encoded as bit-fields in this instruction, so that up to three RCU instructions may be executed in one cycle.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
16-bit forms	
modulo RI	Compare RI to its corresponding range register, subtracting the range if it is greater. Add the range if RI is less than zero.

Operand Values: RI is any index register **rx**, **ry**, **ru**, or **rv**. The value is considered to be a 16.16 number, and the **xyctl** and **uvctl** registers are ignored.

Condition Codes: modmi : set if RI was less then zero, cleared otherwise.
modge : set if RI was greater than or equal to the range, cleared otherwise.

Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: sigbits(Scalar Source) \Rightarrow Scalar Destination

Description: Find the number of significant bits in the Scalar Source, and write the result to a destination scalar register.

For positive numbers:

The number of significant bits is in the range 0-31. The significant bits function, sigbits(), is defined as the bit position of the left-most 1, plus 1. If the number is 0 then the result is 0.

For negative numbers:

The number of significant bits is in the range 0-31. The significant bits function, sigbits(), is defined as the bit position of the left-most 0, plus 1. If the number is -1 then the result is 0.

The number of significant bits therefore lies in the range 0 to 31.

In logical terms, this operation can be considered as returning the bit position of the left-most bit which is not the same as the top bit

Note – the behavior of this function for negative numbers may not be what you require for two's complement operations, in which case you should ABS the value before applying MSB. Note in particular that for powers of 2 it is not symmetrical; plus two will return 2, but minus two will return 1.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
32-bit forms	
msb Si, Sk	write sigbits(Si) to Sk

Operand Values: Si any scalar register r0-r31.
Sk any scalar register r0-r31.

Condition Codes: z : set if the result is zero, cleared otherwise.
n : unchanged.
c : unchanged.
v : unchanged.

Other condition codes are unchanged by this instruction.

Function Unit: MUL

Operation: Scalar Source A * Scalar Source B \Rightarrow Scalar Destination

Description: Signed integer multiply of two 32-bit scalar values, shifting the 64-bit product as specified, and storing the bottom thirty-two bits of the shifted result in the destination register.

The shift range is from +63 to -32, with positive numbers implying a right shift of the 64 bit multiplier result. Thus 32 bits are extracted, sign extended and rounded down (i.e. LSBs are truncated, and not rounded). If a negative shift is used (i.e. a left shift of the accumulator), then zeros are shifted in from the right.

This operation completes in two clock cycles, so the result is not valid during the following clock cycle. However, you cannot rely on the previous value still being in the destination register in the following clock cycle, because if the MPE stalls for any reason then it will be over-written.

The multiply overflow flag is valid at the same time, and subject to the same restrictions, as the multiply result.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
16-bit forms		
mul Si,Sk,>>acshift,Sk	multiply Si and Sk, arithmetically shift the product by the acshift register value, and write the result to Sk	
32-bit forms		
mul Si,Sj,>>acshift,Sk	multiply Si and Sj, arithmetically shift the product by the acshift register value, and write the result to Sk	
mul Si,Sk,>>#m,Sk	multiply Si and Sk, arithmetically shift the product by #m, and write the result to Sk	$-32 \leq m \leq 63$
mul Si,Sk,>>Sq,Sk *	multiply Si and Sk, arithmetically shift the product by Sq, and write the result to Sk	$-32 \leq Sq \leq 63$
mul #n,Sj,>>acshift,Sk	multiply #n and Sj, arithmetically shift the product by the acshift register value, and write the result to Sk	$0 \leq n \leq 31$
mul #n,Sk,>>#m,Sk	multiply #n and Sk, arithmetically shift the product by #m, and write the result to Sk	$0 \leq n \leq 31$ $-32 \leq m \leq 63$
mul #n,Sk,>>Sq,Sk *	multiply #n and Sk, arithmetically shift the product by Sq, and write the result to Sk	$0 \leq n \leq 31$ $-32 \leq Sq \leq 63$

Restricted Forms: * Instructions marked with an asterisk share a register file read port with other function units. Only one of these instructions may be present in a given packet:

ECU jmp/jsr cc,(Si) | cc,(Si),nop

RCU mvr/addr Si,RI

ALU and/or/eor/ftst Si,>>Sj,Sk | Si,<>Sj,Sk

MUL mul Si,Sk,>>Sq,Sk | #n,Sk,>>Sq,Sk

...continued

Operand Values:

Si	any scalar register r0-r31.
Sj	any scalar register r0-r31.
Sq	any scalar register r0-r31. Bits 6-0 are used, bits 31-7 are ignored.
Sk	any scalar register r0-r31.
#n	5-bit immediate value, zero extended to 32 bits.
#m	immediate shift value.
acshift	shift value from the acshift register.
>>	shifts are arithmetic, right for positive values, left for negative values.

Condition Codes:

mv : set if there are any significant two's complement bits above the 32-bit extracted result, cleared otherwise. The mv result is valid for shift values in the range 0 to 63, and is otherwise undefined.

Other condition codes are unchanged by this instruction.

Function Unit: MUL

Operation: Pixel Source A * Pixel Source B \Rightarrow Pixel Destination

Description: Three parallel 16x16 signed integer multiply operations are performed, giving three 32-bit products. One of the sources is always a pixel. The other source may either be another pixel, or a scalar, or registers **ru** or **rv**. The result is shifted left as defined by the instruction, and written to the destination pixel.

This instruction is identical in operation to **mul_sv**, with the exception that only elements 0-2 of the small vector are changed, the fourth element being entirely unchanged by this instruction.

The **ru** and **rv** forms of this are specifically useful for linear interpolation functions, such as anti-aliased textures, and tri-linear interpolation.

This operation completes in two clock cycles, so the result is not valid during the following clock cycle. However, you cannot rely on the previous value still being in the destination register in the following clock cycle, because if the MPE stalls for any reason then it will be over-written.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
32-bit forms	
mul_p Si,Vj,>>svshift,Vk	form a pixel by repeating the 16 most significant bits of scalar register Si three times, multiply it by all three elements of pixel Vj, shift the products by the amount determined by the svshift register, and write the result to pixel Vk
mul_p Si,Vj,>>#m,Vk	form a pixel by repeating the 16 most significant bits of scalar register Si three times, multiply it by all three elements of pixel Vj, shift the products by the amount determined by the immediate value (legal values are {16,24,32,30}, see below), and write the result to pixel Vk
mul_p ru,Vj,>>svshift,Vk	form a pixel by repeating the 14 most significant fractional bits of index register ru three times, multiply it by all three elements of pixel Vj, shift the products by the amount determined by the svshift register, and write the result to pixel Vk
mul_p ru,Vj,>>#m,Vk	form a pixel by repeating the 14 most significant fractional bits of index register ru three times, multiply it by all three elements of pixel Vj, shift the products by the amount determined by the immediate value (legal values are {16,24,32,30}, see below), and write the result to pixel Vk
mul_p rv,Vj,>>svshift,Vk	form a pixel by repeating the 14 most significant fractional bits of index register rv three times, multiply it by all three elements of pixel Vj, shift the products by the amount determined by the svshift register, and write the result to pixel Vk
mul_p rv,Vj,>>#m,Vk	form a pixel by repeating the 14 most significant fractional bits of index register rv three times, multiply it by all three elements of pixel Vj, shift the products by the amount determined by the immediate value (legal values are {16,24,32,30}, see below), and write the result to pixel Vk

...continued

INSTRUCTION	DESCRIPTION
32-bit forms	
mul_p Vi,Vj,>>svshift,Vk	multiply all three elements of pixel Vi by all three elements of pixel Vj, shift the products by the amount determined by the svshift register, and write the result to pixel Vk
mul_p Vi,Vj,>>#m,Vk	multiply all three elements of pixel Vi by all three elements of pixel Vj, shift the products by the amount determined by the immediate value (legal values are {16,24,32,30}, see below), and write the result to pixel Vk

Operand Values:

- Si any scalar register **r0-r31**. Bits 31-16 are used, bits 15-0 are ignored.
- ru,rv the most significant 14 bits of the fractional part of index register **ru** or **rv** are combined with 2 leading zeroes to create a positive 2.14 number. The position of the binary point in **ru** and **rv** is determined by the **uv_mipmap** field of the **uvctl** register (only values 0-4 are supported).
- Vi any vector register **v0-v7**, as a pixel.
- Vj any vector register **v0-v7**, as a pixel.
- Vk any vector register **v0-v7**, as a pixel.
- >> the value encoded into #m, or encoded in the the **svshift** register, determines the final shift amount, as follows:

Encoding for #m	Encoding for svshift	Description
16	0	the 32-bit product values are shifted left by 16, filling with zeros. This produces 16.16 pixel results when the input pixel elements are considered to be in 16.0 format.
24	1	the 32-bit product values are shifted left by 8, filling with zeros. This produces 8.24 pixel results when the input pixel elements are considered to be in 8.8 format.
32	2	the 32-bit product values are used directly, and are not shifted. This produces 0.32 pixel results when the input pixel elements are considered to be in 0.16 format.
30	3	the 32-bit product values are shifted left by 2, filling with zeros. This produces 2.30 pixel results when the input pixel elements are considered to be in 2.14 format.

Condition Codes: Unchanged by this instruction.

Function Unit: MUL

Operation: Small vector Source A * Small vector Source B \Rightarrow Vector Destination

Description: Four parallel 16x16 signed integer multiply operations are performed, giving four 32-bit products in a vector register. One of the sources is always a small vector. The other source may either be another small vector, or a scalar, or registers **ru** or **rv**. The result is shifted left as defined by the instruction, and written to the destination vector.

The **ru** and **rv** forms of this are specifically useful for linear interpolation functions, such as anti-aliased textures, and tri-linear interpolation.

This operation completes in two clock cycles, so the result is not valid during the following clock cycle. However, you cannot rely on the previous value still being in the destination register in the following clock cycle, because if the MPE stalls for any reason then it will be over-written.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
16-bit forms	
mul_sv Vi,Vk,>>svshift,Vk	multiply each element of small vector Vi by the corresponding element of Vj, shift the products by the amount determined by the svshift register, and write the result to the vector Vk.
32-bit forms	
mul_sv Si,Vj,>>svshift,Vk	form a small-vector by repeating the 16 most significant bits of scalar register Si four times, multiply it by all four elements of small-vector Vj, shift the products by the amount determined by the svshift register, and write the result to small-vector Vk
mul_sv Si,Vj,>>#m,Vk	form a small-vector by repeating the 16 most significant bits of scalar register Si four times, multiply it by all four elements of small-vector Vj, shift the products by the amount determined by the immediate value (legal values are {16,24,32,30}, see below), and write the result to small-vector Vk
mul_sv ru,Vj,>>svshift,Vk	form a small-vector by repeating the 14 most significant fractional bits of index register ru four times, multiply it by all four elements of small-vector Vj, shift the products by the amount determined by the svshift register, and write the result to small-vector Vk
mul_sv ru,Vj,>>#m,Vk	form a small-vector by repeating the 14 most significant fractional bits of index register ru four times, multiply it by all four elements of small-vector Vj, shift the products by the amount determined by the immediate value (legal values are {16,24,32,30}, see below), and write the result to small-vector Vk
mul_sv rv,Vj,>>svshift,Vk	form a small-vector by repeating the 14 most significant fractional bits of index register rv four times, multiply it by all four elements of small-vector Vj, shift the products by the amount determined by the svshift register, and write the result to small-vector Vk

...continued

INSTRUCTION	DESCRIPTION
32-bit forms	
mul_sv rv,Vj,>>#m,Vk	form a small-vector by repeating the 14 most significant fractional bits of index register rv four times, multiply it by all four elements of small-vector Vj , shift the products by the amount determined by the immediate value (legal values are {16,24,32,30}, see below), and write the result to small-vector Vk
mul_sv Vi,Vj,>>svshift,Vk	multiply all four elements of small-vector Vi by all four elements of small-vector Vj , shift the products by the amount determined by the svshift register, and write the result to small-vector Vk
mul_sv Vi,Vj,>>#m,Vk	multiply all four elements of small-vector Vi by all four elements of small-vector Vj , shift the products by the amount determined by the immediate value (legal values are {16,24,32,30}, see below), and write the result to small-vector Vk

Operand Values:

- Si** any scalar register **r0-r31**. Bits 31-16 are used, bits 15-0 are ignored.
- ru,rv** the most significant 14 bits of the fractional part of index register **ru** or **rv** are combined with 2 leading zeroes to create a positive 2.14 number. The position of the binary point in **ru** and **rv** is determined by the **uv_mipmap** field of the **uvctl** register (only values 0-4 are supported).
- Vi** any vector register **v0-v7**, as a small-vector.
- Vj** any vector register **v0-v7**, as a small-vector.
- Vk** any vector register **v0-v7**, as a small-vector.
- >>** the value encoded into **#m**, or encoded in the **svshift** register, determines the final shift amount, as follows:

Encoding for #m	Encoding for svshift	Description
16	0	the 32-bit product values are shifted left by 16, filling with zeros. This produces 16.16 small-vector results when the input small-vector elements are considered to be in 16.0 format.
24	1	the 32-bit product values are shifted left by 8, filling with zeros. This produces 8.24 small-vector results when the input small-vector elements are considered to be in 8.8 format.
32	2	the 32-bit product values are used directly, and are not shifted. This produces 0.32 small-vector results when the input small-vector elements are considered to be in 0.16 format.
30	3	the 32-bit product values are shifted left by 2, filling with zeros. This produces 2.30 small-vector results when the input small-vector elements are considered to be in 2.14 format.

Condition Codes: Unchanged by this instruction.

Function Unit: MEM

Operation: Scalar Source \Rightarrow Scalar Destination

Description: Move scalar data

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
16-bit forms		
mv_s Sj, Sk	move scalar data from register Sj into register Sk	
mv_s #n, Sk	move #n into register Sk	$-16 \leq n \leq 15$
32-bit forms		
mv_s #nnn, Sk	move #nnn into register Sk	$-2048 \leq nnn \leq 2047$
48-bit forms		
mv_s #nnnn, Sk	move #nnnn into register Sk	$-(2^{31}) \leq nnnn \leq (2^{31})-1$

Operand Values:

- Sj any scalar register r0-r31.
- Sk any scalar register r0-r31.
- #n 5-bit immediate value, sign extended to 32 bits.
- #nnn 12-bit immediate value, sign extended to 32 bits.
- #nnnn 32-bit immediate value.

Condition Codes: Unchanged by this instruction.

Function Unit: MEM

Operation: Vector Source \Rightarrow Vector Destination

Description: Move vector data from register to register

Assembler Syntax:

INSTRUCTION	DESCRIPTION
16-bit forms	
mv_v Vj, Vk	move vector data from register Vj into register Vk

Operand Values: Vj any vector register v0-v7.
Vk any vector register v0-v7.

Condition Codes: Unchanged by this instruction.

Function Unit: RCU

Operation: Scalar Source \Rightarrow Index Register

Description: Move scalar data to RCU index register.

Up to two **dec** instructions may also be encoded as bit-fields in this instruction, so that up to three register unit operations may be executed in one cycle.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
16-bit forms		
mvr Sj,RI	move data from register Sj into index register RI	
48-bit forms		
mvr #nnnn,RI	move #nnnn into register RI	$-(2^{31}) \leq \text{nnnn} \leq (2^{31})-1$

Operand Values: Sj any scalar register r0-r31.
 RI index register rx,ry,ru,rw
 #nnnn 32-bit immediate value.

Condition Codes: Unchanged by this instruction.

Function Unit: ALU

Operation: Zero minus Scalar Destination \Rightarrow Scalar Destination

Description: Subtract the destination value from zero, and write the result to the destination register. This is a short form of **sub Sk,#0,Sk**.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
16-bit forms	
neg Sk	subtract Sk from zero, writing the result to Sk

Operand Values: Sk is any scalar register r0-r31.

Condition Codes:

- z : set if the result is zero, cleared otherwise.
- n : set if the result is negative, cleared otherwise.
- c : set if there was a borrow out of the subtraction, cleared otherwise.
- v : set if there was signed arithmetic overflow (sign is invalid), cleared otherwise.

Other condition codes are unchanged by this instruction.

nop**Null Operation****nop****Function Unit:** none**Operation:** Null Operation**Description:** Do nothing.**Assembler Syntax:**

INSTRUCTION	DESCRIPTION
16-bit forms	
<code>nop</code>	

Condition Codes: Unchanged by this instruction.

not

Logical Complement

not**Function Unit:** ALU**Operation:** Scalar Destination exclusive-OR \$FFFFFFFF \Rightarrow Scalar Destination**Description:** Logical complement of all the bits of the destination , and write the result to the destination register.This instruction is equivalent to the instruction “**eor #-1,Sk**”.**Assembler Syntax:**

INSTRUCTION	DESCRIPTION
16-bit forms	
not Sk	logical complement of Sk, writing the result to Sk

Operand Values: Sk is any scalar register r0-r31.**Condition Codes:**
z : set if the result is zero, cleared otherwise.
n : set if the result is negative, cleared otherwise.
c : unchanged.
v : cleared.

Other condition codes are unchanged by this instruction.

or

Logical OR

or

Function Unit: ALU**Operation:** Scalar Source A OR Scalar Source B \Rightarrow Scalar Destination**Description:** Bit-wise logical inclusive OR of two 32-bit sources, writing the result to a scalar register.**Assembler Syntax:**

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
16-bit forms		
or Si,Sk	OR Si with Sk, writing the result to Sk	
32-bit forms		
or Si,Sj,Sk	OR Si with Sj, writing the result to Sk	
or #n,Sj,Sk	OR #n with Sj, writing the result to Sk	$-16 \leq n \leq 15$
or #n,<>#m,Sk	OR #n rotated right by #m, with Sk, writing the result to Sk. May be used to mask in or out, a bit field.	$-16 \leq n \leq 15$ $-\infty \leq m \leq \infty$
or #n,>>Sj,Sk	OR #n logically shifted right by Sj, with Sk, writing the result to Sk	$-16 \leq n \leq 15$ $-32 \leq Sj \leq 31$
or Si,>>#m,Sk	OR Si logically shifted right by #m, with Sk, writing the result to Sk	$-16 \leq m \leq 15$
or Si,>>Sj,Sk *	OR Si logically shifted right by Sj, with Sk, writing the result to Sk	$-32 \leq Sj \leq 31$
or Si,<>Sj,Sk *	OR Si rotated right by Sj, with Sk, writing the result to Sk	all Sj are valid
64-bit forms		
or #nnnn,Sj,Sk	OR #nnnn with Sj, writing the result to Sk	$-(2^{31}) \leq nnnn \leq (2^{31})-1$
or #nnnn,>>Sj,Sk	OR #nnnn logically shifted right by Sj, with Sk, writing the result to Sk	$-(2^{31}) \leq nnnn \leq (2^{31})-1$ $-32 \leq Sj \leq 31$

Restricted Forms: * Instructions marked with an asterisk share a register file read port with other function units. Only one of these instructions may be present in a given packet:

ECU jmp/jsr cc,(Si) | cc,(Si),nop

RCU mvr/addr Si,RI

ALU and/or/eor/ftst Si,>>Sj,Sk | Si,<>Sj,Sk

MUL mul Si,Sk,>>Sq,Sk | #n,Sk,>>Sq,Sk

Operand Values:

Si any scalar register r0-r31.

Sj any scalar register r0-r31. For shifts, bits 5-0 are used, bits 31-6 ignored.

Sk any scalar register r0-r31.

#n 5-bit immediate value, sign extended to 32 bits.

#nnnn 32-bit immediate value.

#m immediate shift or rotate value.

>> shifts are logical, right for positive values, left for negative values.

<> rotates are right for positive values, left for negative values.

...continued

Condition Codes: z : set if the result is zero, cleared otherwise.
 n : set if the result is negative, cleared otherwise.
 c : unchanged.
 v : cleared.

Other condition codes are unchanged by this instruction.

Function Unit: none

Operation:

Description: Pad instructions are used by the assembler to increase the size of an instruction packet so that the next instruction packet can be suitably aligned. The requirement for this operation comes from the restriction that an instruction packet must lie within 128 bits on any 64 bit boundary. This means that instruction packets larger than 80 bits may not be arbitrarily aligned. The MPE ignores these pad instructions and advances the program counter over them. Multiple padding instructions may be used in sequence.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
16-bit forms	
pad	

Condition Codes: Unchanged by this instruction.

Function Unit: MEM

Operation: pop 16 bytes of data from the stack in memory \Rightarrow Registers
 $\mathbf{sp} + 16 \Rightarrow \mathbf{sp}$

Description: Pop 16 bytes of data from the stack, using the current value of the stack pointer register **sp**, and then increase **sp** by 16.

There are four different forms of **pop** instruction, as shown in the table below, and each has a matching **push** form. The first form allows one vector register to be restored from the stack. The second form, useful for sub-routine calls, allows three scalar registers to be restored at the same time as the return address is restored from the stack. The third and fourth forms are useful for interrupt handlers.

This instruction completes in two clock cycles, so the target values may not be used until the second instruction packet after this one. In the instruction packet which follows this one, the destination contents cannot be relied upon and must not be referenced.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
16-bit forms	
pop V_k	Pop the vector register V_k from stack.
pop V_k, rz	Pop first three elements of vector V_k and rz from the stack.
pop $Sk, cc, rzi1, rz$	Pop the scalar register Sk , the condition codes, interrupt program status register rzi1 and rz from the stack. This is primarily for level 1 interrupt service routines.
pop $Sk, cc, rzi2, rz$	Pop the scalar register Sk , the condition codes, interrupt program status register rzi2 and rz from the stack. This is primarily for level 2 interrupt service routines.

Operand Values:

- V_k any vector register v0-v7.
- Sk any scalar register r0-r31.
- cc the **cc** register.
- $rzi1$ the **rzi1** register.
- $rzi2$ the **rzi2** register.
- rz the **rz** register.

Condition Codes: Restored by the **pop $Sk, cc, rzi1, rz$** and **pop $Sk, cc, rzi2, rz$** forms. Otherwise not affected.

Function Unit: MEM

Operation: $sp - 16 \Rightarrow sp$
push 16 bytes of data from registers \Rightarrow the stack in memory

Description: Increase the stack pointer register **sp** by 16, then push 16 bytes of data onto the stack.

There are four different forms of **push** instruction, as shown in the table below, and each has a matching **pop** form. The first form allows one vector register to be copied to the stack. The second form, useful for sub-routine calls, allows three scalar registers to be preserved at the same time as the return address is put on the stack. The third and fourth forms are useful for interrupt handlers.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
16-bit forms	
push Vk	Push the vector register Vk on to stack.
push Vk, rz	Push first three elements of vector Vk and rz on to the stack.
push $Sk, cc, rzi1, rz$	Push the scalar register Sk , the condition codes, interrupt program status register rzi1 and rz on to the stack. This is primarily for level 1 interrupt service routines.
push $Sk, cc, rzi2, rz$	Push the scalar register Sk , the condition codes, interrupt program status register rzi1 and rz on to the stack. This is primarily for level 2 interrupt service routines.

Operand Values:

- Vk any vector register v0-v7.
- Sk any scalar register r0-r31.
- cc the **cc** register.
- $rzi1$ the **rzi1** register.
- $rzi2$ the **rzi2** register.
- rz the **rz** register.

Condition Codes: Unchanged by this instruction.

Function Unit: RCU

Operation: IF (Index \geq Range) set **modge**
 IF (Index < 0) set **modmi**

Description: Compare the integer part of the specified index register to its corresponding range from the **xyrange** or **uvrange** register, and also compare it to zero. Change the **modmi** and **modge** condition code flags as shown below, without changing the index register.

The **modulo** instruction performs the same operation as **range**, except that it also changes the index register.

Up to two **dec** instructions may be encoded as bit-fields in this instruction, so that up to three RCU instructions may be executed in one cycle.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
16-bit forms	
range RI	Compare RI to its corresponding range register, and to zero, setting flags appropriately.

Operand Values: RI is any index register **rx**, **ry**, **ru**, or **rv**. The value is considered to be a 16.16 number, and the **xyctl** and **uvctl** registers are ignored.

Condition Codes: modmi : set if RI was less than zero, cleared otherwise.
 modge : set if RI was greater than or equal to the range, cleared otherwise.
 Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: Scalar Source B \diamond Source A \Rightarrow Scalar Destination

Description: Rotate Source B right by Source A, setting flags appropriately, and writing the result to the destination.



Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
32-bit forms		
rot >>Sj, Si, Sk	rotate Si right by Sj bits, writing the result to Sk. Negative values may be used to encode a rotate left.	all Sj are valid
rot #m, Si, Sk	rotate Si right by #m, writing the result to Sk	$-\infty \leq m \leq \infty$

Operand Values:

- Sj any scalar register r0-r31.
- Si any scalar register r0-r31.
- Sk any scalar register r0-r31.
- #m immediate shift value.

Condition Codes:

- z : set if the result is zero, cleared otherwise.
- n : set if the result is negative, cleared otherwise.
- c : unchanged.
- v : cleared.

Other condition codes are unchanged by this instruction.

Function Unit: ECU

Operation: Conditional jump to the absolute address in **rzi1** or **rzi2**, also clearing the corresponding **imaskHw1** or **imaskHw2** bit

Description: If the specified condition is true, then the **rti** jump is taken, otherwise the jump is not taken. A taken jump which has a **nop** operand will force two ‘dead’ cycles after executing the jump packet, then continue execution from the target address. A taken jump which does not have a **nop** operand will have no ‘dead’ cycles—the jump packet, the next two packets, and the packet at the target address will execute on successive cycles (ignoring unrelated pipeline stalls). If a jump is not taken, whether or not it has a **nop** operand, execution will continue with the next packet.

The two instruction packets after a packet containing a jump without a **nop** operand are in what is known as the “delay slots” of the jump. If such a jump is taken, any ECU instructions (**bra**, **halt**, **jmp**, **jsr**, **rti**, **rts**) in its delay slots will not be evaluated. If the jump is not taken, the delay slots execute normally. This allows multi-way jump decisions to be made in successive instruction packets.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	TARGET ADDRESS
16-bit forms		
<code>rti cc,(rzi1)</code>	If the cc condition is true: execute the next two packets, clear the imaskHw1 bit in the intctl register, then continue execution at address rzi1 . false: continue execution with the next packet.	32-bit absolute address
<code>rti cc,(rzi1),nop</code>	If the cc condition is true: force two dead cycles, clear the imaskHw1 bit in the intctl register, then continue execution at address rzi1 . false: continue execution with the next packet.	32-bit absolute address
<code>rti cc,(rzi2)</code>	If the cc condition is true: execute the next two packets, clear the imaskHw2 bit in the intctl register, then continue execution at address rzi2 . false: continue execution with the next packet.	32-bit absolute address
<code>rti cc,(rzi2),nop</code>	If the cc condition is true: force two dead cycles, clear the imaskHw2 bit in the intctl register, then continue execution at address rzi2 . false: continue execution with the next packet.	32-bit absolute address

...continued

Operand Values: **rzi1** and **rzi2** target addresses are always an even number since instructions are on 16-bit boundaries.

cc may take on any of the following values (if not specied, t is assumed):

cc mnemonic	condition	test
ne	Not equal	/z
eq	Equal	z
lt	Less than	(n./v) + (/n.v)
le	Less than or equal	z + (n./v) + (/n.v)
gt	Greater than	(n.v./z) + (/n./v./z)
ge	Greater than or equal	(n.v) + (/n./v)
c0ne	rc0 not equal to zero	/c0z
clne	rc1 not equal to zero	/clz
c0eq	rc0 equal to zero	c0z
cleq	rc1 equal to zero	clz
cc (hs)	Carry clear (High or same)	/c
cs (lo)	Carry set (Low)	c
vc	Overflow clear	/v
vs	Overflow set	v
mvc	Multiply overflow clear	/mv
mvs	Multiply overflow set	mv
hi	High	/c./z
ls	Low or same	c + z
pl	Plus	/n
mi	Minus	n
t	True	1
modmi	modulo RI was < zero	modmi
modpl	modulo RI was >= zero	/modmi
modge	modulo RI was >= range	modge
modlt	modulo RI was < range	/modge
cf0lo	Coprocessor flag 0 low	/cf0
cf0hi	Coprocessor flag 0 high	cf0
cf1lo	Coprocessor flag 1 low	/cf1
cf1hi	Coprocessor flag 1 high	cf1

Condition Codes: Unchanged by this instruction.

Function Unit:

ECU

Operation:

Conditional jump to the absolute address in **rz**

Description:

If the specified condition is true, then the **rts** jump is taken, otherwise the jump is not taken. A taken jump which has a **nop** operand will force two ‘dead’ cycles after executing the jump packet, then continue execution from the target address. A taken jump which does not have a **nop** operand will have no ‘dead’ cycles—the jump packet, the next two packets, and the packet at the target address will execute on successive cycles (ignoring unrelated pipeline stalls). If a jump is not taken, whether or not it has a **nop** operand, execution will continue with the next packet.

The two instruction packets after a packet containing a jump without a **nop** operand are in what is known as the “delay slots” of the jump. If such a jump is taken, any ECU instructions (**bra**, **halt**, **jmp**, **jsr**, **rti**, **rts**) in its delay slots will not be evaluated. If the jump is not taken, the delay slots execute normally. This allows multi-way jump decisions to be made in successive instruction packets.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	TARGET ADDRESS
16-bit forms		
<code>rts cc</code>	If the cc condition is true: execute the next two packets, then continue execution at address rzi1 . false: continue execution with the next packet.	32-bit absolute address
<code>rts cc,nop</code>	If the cc condition is true: force two dead cycles, then continue execution at address rzi1 . false: continue execution with the next packet.	32-bit absolute address

...continued

Operand Values: cc may take on any of the following values (if not specied, t is assumed):

cc mnemonic	condition	test
ne	Not equal	/z
eq	Equal	z
lt	Less than	(n./v) + (/n.v)
le	Less than or equal	z + (n./v) + (/n.v)
gt	Greater than	(n.v./z) + (/n./v./z)
ge	Greater than or equal	(n.v) + (/n./v)
c0ne	rc0 not equal to zero	/c0z
clne	rc1 not equal to zero	/clz
c0eq	rc0 equal to zero	c0z
cleq	rc1 equal to zero	clz
cc (hs)	Carry clear (High or same)	/c
cs (lo)	Carry set (Low)	c
vc	Overflow clear	/v
vs	Overflow set	v
mvc	Multiply overflow clear	/mv
mvs	Multiply overflow set	mv
hi	High	/c./z
ls	Low or same	c + z
pl	Plus	/n
mi	Minus	n
t	True	1
modmi	modulo RI was < zero	modmi
modpl	modulo RI was >= zero	/modmi
modge	modulo RI was >= range	modge
modlt	modulo RI was < range	/modge
cf0lo	Coprocessor flag 0 low	/cf0
cf0hi	Coprocessor flag 0 high	cf0
cf1lo	Coprocessor flag 1 low	/cf1
cflhi	Coprocessor flag 1 high	cf1

Condition Codes: Unchanged by this instruction.

Function Unit: ALU

Operation: Saturate (Scalar Source) \Rightarrow Scalar Register

Description: The signed integer scalar source is checked to see if it falls outside a defined range of significant bits, and if it does, then it is clipped to within this range. For example, a saturate to 16 bits will change any values greater than \$00007FFF to \$00007FFF, and any value less than \$FFFF8000 to \$FFFF8000.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
32-bit forms		
sat #n, Si, Sk	saturate Si to n bits, writing the result to Sk	$1 \leq n \leq 32$

Operand Values:

- Si any scalar register r0-r31.
- Sk any scalar register r0-r31.
- #n 5-bit immediate value.

Condition Codes:

- z : set if the result is zero, cleared otherwise.
- n : set if the result is negative, cleared otherwise.
- c : unchanged.
- v : unchanged.

Other condition codes are unchanged by this instruction.

Function Unit: MEM

Operation: Pixel Data \Rightarrow Memory

Description: Store a pixel value to memory. See the ‘MPE Data Types’ section for a full discussion of the behavior of **st_p** for each data type. The effective address for the store must be on the selected pixel size boundary, with the appropriate number of least significant bits equal to zero.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
32-bit forms	
st_p Vj, (Si)	store pixel from register Vj to address Si, transforming the data according to the settings in the linpixctl register (only data types 4 to 6 are valid)
st_p Vj, <label>	store pixel from register Vj to address <label>, transforming the data according to the settings in the linpixctl register (only data types 4 to 6 are valid)
st_p Vj, (xy)	store pixel from register Vj to bilinear address (xy), transforming the data according to the settings in the xyctl register (only data types 4 to 6 are valid)
st_p Vj, (uv)	store pixel from register Vj to bilinear address (uv), transforming the data according to the settings in the uvctl register (only data types 4 to 6 are valid)

Operand Values:

- Si any scalar register r0-r31, as an absolute 32-bit address.
- Vj any vector register v0-v7, as a pixel value.
- (xy) bilinear address formed from the **rx**, **ry**, **xybase**, and **xyctl** registers.
- (uv) bilinear address formed from the **ru**, **rv**, **uvbase**, and **uvctl** registers.
- <label> is resolved to an address by the assembler/linker. The instruction encoding for this immediate address value restricts it to being on a word boundary within an 11-bit offset in words (bits 11 to 1 of the address) above any of the following base values:
 - \$2000_0000 base of local dtrom
 - \$2010_0000 base of local dtrom
 - \$2050_0000 base of local control registers

Condition Codes: Unchanged by this instruction.

Function Unit: MEM

Operation: Pixel plus Z Data \Rightarrow Memory

Description: Store a pixel plus Z value to memory. See the ‘MPE Data Types’ section for a full discussion of the behavior of **st_pz** for each data type. The effective address for the store must be on the selected pixel size boundary, with the appropriate number of least significant bits equal to zero.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
32-bit forms	
st_pz Vj, (Si)	store pixel plus Z from register Vj to address Si, transforming the data according to the settings in the linpixctl register (only data types 4 to 6 are valid)
st_pz Vj, <label>	store pixel plus Z from register Vj to address <label>, transforming the data according to the settings in the linpixctl register (only data types 4 to 6 are valid)
st_pz Vj, (xy)	store pixel plus Z from register Vj to bilinear address (xy), transforming the data according to the settings in the xyctl register (only data types 4 to 6 are valid)
st_pz Vj, (uv)	store pixel plus Z from register Vj to bilinear address (uv), transforming the data according to the settings in the uvctl register (only data types 4 to 6 are valid)

Operand Values:

- Si any scalar register r0-r31, as an absolute 32-bit address.
- Vj any vector register v0-v7, as a pixel plus Z value.
- (xy) bilinear address formed from the **rx**, **ry**, **xybase**, and **xyctl** registers.
- (uv) bilinear address formed from the **ru**, **rv**, **uvbase**, and **uvctl** registers.
- <label> is resolved to an address by the assembler/linker. The instruction encoding for this immediate address value restricts it to being on a word boundary within an 11-bit offset in words (bits 11 to 1 of the address) above any of the following base values:
 - \$2000_0000 base of local dtrom
 - \$2010_0000 base of local dtrom
 - \$2050_0000 base of local control registers

Condition Codes: Unchanged by this instruction.

Function Unit: MEM

Operation: Scalar Source \Rightarrow Memory

Description: Store a scalar value to memory. The effective address for the store may be on any scalar boundary, and the least significant 2 bits will be ignored.

The **st_io** instruction is a synonym for **st_s**. The **st_io** form may be found in some old software written when it used to be a separate form.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
16-bit forms	
st_s Sj, (Si)	store scalar from register Sj to address Si
st_s Sj, <labelA>	store scalar from register Sj to address <labelA>
32-bit forms	
st_s Sj, <labelB>	store scalar from register Sj to address <labelB>
st_s Sj, (xy)	store scalar from register Sj to bilinear address (xy) (only data type A is valid)
st_s Sj, (uv)	store scalar from register Sj to bilinear address (uv) (only data type A is valid)
st_s #nn, <labelC>	store immediate data to address <labelC>
64-bit forms	
st_s #nnnn, <labelD>	store immediate data to address <labelD>

Operand Values:

- Si any scalar register r0-r31, as an absolute 32-bit address.
- Sk any scalar register r0-r31.
- (xy) bilinear address formed from the **rx**, **ry**, **xybase**, and **xyctl** registers.
- (uv) bilinear address formed from the **ru**, **rv**, **uvbase**, and **uvctl** registers.
- #nn 10-bit immediate value, zero extended to 32 bits.
- #nnnn 32-bit immediate value.
- <labelA> is resolved to an address by the assembler/linker. The instruction encoding for this immediate address value restricts it to being on a vector boundary within a 5-bit offset in vectors (bits 8 to 4 of the address) above the following base value:
\$2050_0000 base of local control registers
- <labelB> is resolved to an address by the assembler/linker. The instruction encoding for this immediate address value restricts it to being on a scalar boundary within a 11-bit offset in scalars (bits 12 to 2 of the address) above any of the three base values shown below.
- <labelC> is resolved to an address by the assembler/linker. The instruction encoding for this immediate address value restricts it to being on a vector boundary within a 9-bit offset in scalars (bits 12 to 4 of the address) above the base of local control registers shown below.

...continued

<labelD> is resolved to an address by the assembler/linker. The instruction encoding for this immediate address value restricts it to being on a scalar boundary within a 12-bit offset in scalars (bits 13 to 2 of the address) above any of the three base values shown below.

\$2000_0000 base of local dtrom

\$2010_0000 base of local dtrom

\$2050_0000 base of local control registers

Condition Codes: Unchanged by this instruction.

Function Unit: MEM

Operation: Small-Vector Data \Rightarrow Memory

Description: Store a small-vector value to memory. The effective address for the store may be on any 8-byte boundary, and the least significant 3 bits will be ignored.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
32-bit forms	
st_sv Vj, (Si)	store small-vector from register Vj to address Si
st_sv Vj, <label>	store small-vector from register Vj to address <label>
st_sv Vj, (xy)	store small-vector from register Vj to bilinear address (xy) (only data type C is valid)
st_sv Vj, (uv)	store small-vector from register Vj to bilinear address (uv) (only data type C is valid)

Operand Values:

- Si any scalar register r0-r31, as an absolute 32-bit address.
- Vj any vector register v0-v7, as a small-vector value.
- (xy) bilinear address formed from the **rx**, **ry**, **xybase**, and **xyctl** registers.
- (uv) bilinear address formed from the **ru**, **rv**, **uvbase**, and **uvctl** registers.
- <label> is resolved to an address by the assembler/linker. The instruction encoding for this immediate address value restricts it to being on an 8-byte boundary within an 11-bit offset in words (bits 13 to 3 of the address) above any of the following base values:
 - \$2000_0000 base of local dtrom
 - \$2010_0000 base of local dtram
 - \$2050_0000 base of local control registers

Condition Codes: Unchanged by this instruction.

Function Unit: MEM

Operation: Vector Register \Rightarrow memory

Description: Store a vector value to memory. The effective address for the store may be on any 16-byte boundary, and the least significant 4 bits will be ignored.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
32-bit forms	
st_v Vj, (Si)	store vector from register Vj to address Si
st_v Vj, <label>	store vector from register Vj to address <label>
st_v Vj, (xy)	store vector from register Vj to bilinear address (xy) (only data type D is valid)
st_v Vj, (uv)	store vector from register Vj to bilinear address (uv) (only data type D is valid)

Operand Values:

- Si any scalar register r0-r31, as an absolute 32-bit address.
- Vj any vector register v0-v7, as a vector value.
- (xy) bilinear address formed from the **rx**, **ry**, **xybase**, and **xyctl** registers.
- (uv) bilinear address formed from the **ru**, **rv**, **uvbase**, and **uvctl** registers.
- <label> is resolved to an address by the assembler/linker. The instruction encoding for this immediate address value restricts it to being on an 16-byte boundary within an 11-bit offset in words (bits 14 to 4 of the address) above any of the following base values:
 - \$2000_0000 base of local dtrom
 - \$2010_0000 base of local dtam
 - \$2050_0000 base of local control registers

Condition Codes: Unchanged by this instruction.

Function Unit: ALU

Operation: Scalar - Scalar \Rightarrow Scalar Register

Description: Subtract one scalar value from another scalar value, and write the result to a scalar register.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
16-bit forms		
sub S_i, S_k	subtract S_i from S_k , writing the result to S_k	
sub $\#n, S_k$	subtract $\#n$ from S_k , writing the result to S_k	$0 \leq n \leq 31$
32-bit forms		
sub S_i, S_j, S_k	subtract S_i from S_j , writing the result to S_k	
sub $\#n, S_j, S_k$	subtract $\#n$ from S_j , writing the result to S_k	$0 \leq n \leq 31$
sub $\#nn, S_k$	subtract $\#nn$ from S_k , writing the result to S_k	$0 \leq nn \leq 1023$
sub $\#n, >>\#m, S_k$	subtract $\#n$ arithmetically shifted right by $\#m$, from S_k , writing the result to S_k	$0 \leq n \leq 31$ $-16 \leq m \leq 0$
sub $S_i, \#n, S_k$	subtract S_i from $\#n$, writing the result to S_k	$0 \leq n \leq 31$
sub $S_i, >>\#m, S_k$	subtract S_i arithmetically shifted right by $\#m$, from S_k , writing the result to S_k	$-16 \leq m \leq 15$
48-bit forms		
sub $\#nnnn, S_k$	subtract $\#nnnn$ from S_k , writing the result to S_k	$-(2^{31}) \leq nnnn \leq (2^{31}) - 1$
64-bit forms		
sub $\#nnnn, S_j, S_k$	subtract $\#nnnn$ from S_j , writing the result to S_k	$-(2^{31}) \leq nnnn \leq (2^{31}) - 1$
sub $S_i, \#nnnn, S_k$	subtract S_i from $\#nnnn$, writing the result to S_k	$-(2^{31}) \leq nnnn \leq (2^{31}) - 1$

Operand Values:

- S_i any scalar register r0-r31.
- S_j any scalar register r0-r31.
- S_k any scalar register r0-r31.
- $\#n$ 5-bit immediate value, zero extended to 32 bits.
- $\#nn$ 10-bit immediate value, zero extended to 32 bits.
- $\#nnnn$ 32-bit immediate value.
- $\#m$ immediate shift value.
- $>>$ shifts are arithmetic, right for positive values, left for negative values, and overflow from shift out is not detected.

Condition Codes:

- z : set if the result is zero, cleared otherwise.
- n : set if the result is negative, cleared otherwise.
- c : set if there is a borrow out of the subtraction, cleared otherwise.
- v : set if there is signed arithmetic overflow (sign is invalid), cleared otherwise.

Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: Scalar - Scalar - Carry Condition Code \Rightarrow Scalar Register

Description: Subtract one scalar value from another scalar value and also subtract the current value of the carry condition code bit, and write the result to a scalar register.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	DATA RESTRICTIONS
32-bit forms		
subwc S_i, S_j, S_k	subtract c and S_i from S_j , writing the result to S_k	
subwc $\#n, S_j, S_k$	subtract c and $\#n$ from S_j , writing the result to S_k	$0 \leq n \leq 31$
subwc $\#nn, S_k$	subtract c and $\#nn$ from S_k , writing the result to S_k	$0 \leq nn \leq 1023$
subwc $\#n, \gg\#m, S_k$	subtract c and $\#n$ arithmetically shifted right by $\#m$, from S_k , writing the result to S_k	$0 \leq n \leq 31$ $-16 \leq m \leq 0$
subwc $S_i, \#n, S_k$	subtract c and S_i from $\#n$, writing the result to S_k	$0 \leq n \leq 31$
subwc $S_i, \gg\#m, S_k$	subtract c and S_i arithmetically shifted right by $\#m$, from S_k , writing the result to S_k	$-16 \leq m \leq 15$
64-bit forms		
subwc $\#nnnn, S_j, S_k$	subtract c and $\#nnnn$ from S_j , write the result to S_k	$-(2^{31}) \leq nnnn \leq (2^{31})-1$
subwc $S_i, \#nnnn, S_k$	subtract c and S_i from $\#nnnn$, write the result to S_k	$-(2^{31}) \leq nnnn \leq (2^{31})-1$

Operand Values:

- c current value of the c flag in the cc register, zero extended to 32 bits.
- S_i any scalar register r0-r31.
- S_j any scalar register r0-r31.
- S_k any scalar register r0-r31.
- $\#n$ 5-bit immediate value, zero extended to 32 bits.
- $\#nn$ 10-bit immediate value, zero extended to 32 bits.
- $\#nnnn$ 32-bit immediate value.
- $\#m$ immediate shift value.
- \gg shifts are arithmetic, right for positive values, left for negative values, and overflow from shift out is not detected.

Condition Codes:

- z : unchanged if the result is zero, cleared otherwise.
- n : set if the result is negative, cleared otherwise.
- c : set if there is a borrow out of the subtraction, cleared otherwise.
- v : set if there is signed arithmetic overflow (sign is invalid), cleared otherwise.

Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: Pixel Source A - Pixel Source B \Rightarrow Vector Destination (first 3 scalars)

Description: Subtract two pixels. Pixels consist of three 16 bit elements, taken from the 16 MSBs of the first three scalars in a vector register. Each 16 bit element of the first source is independently subtracted from the corresponding element in the other source, and the result is written to the destination vector register in the same format. The lower 16 bits of each of the first three scalars in the vector destination are written with zeros.

This instruction behaves identically to the **sub_sv** instruction, except that only the first three elements (the three lowest numbered scalars) of the vector register destination are written.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
16-bit forms	
sub_p V_i, V_j, V_k	subtract pixel V_i from pixel V_j , writing the result to V_k

Operand Values:

- V_i any vector register v0-v7, as a pixel.
- V_j any vector register v0-v7, as a pixel.
- V_k any vector register v0-v7, as a pixel.

Condition Codes: Unchanged by this instruction.

Function Unit: ALU

Operation: Small Vector Source A - Small Vector Source B \Rightarrow Vector Destination

Description: Subtract two small vectors. Small vectors consist of four 16 bit elements, taken from the 16 MSBs of the four scalars in a vector register. Each 16 bit element of the first source is independently subtracted from the corresponding element in the other source, and the result is written to the destination vector register in the same format. The lower 16 bits of each scalar element of the vector destination are written with zeros.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
16-bit forms	
sub_sv Vi,Vk	subtract small-vector Vi from small-vector Vj, writing the result to Vk
32-bit forms	
sub_sv Vi,Vj,Vk	subtract small-vector Vi from small-vector Vj, writing the result to Vk

Operand Values: Vi any vector register v0-v7, as a small-vector.
Vj any vector register v0-v7, as a small-vector.
Vk any vector register v0-v7, as a small-vector.

Condition Codes: Unchanged by this instruction.

Function Unit: MUL

Operation: Scalar Source A – Scalar Source B \Rightarrow Scalar Destination

Description: Compute the thirty-two bit difference of the two sources, and write this to the destination scalar register. This instruction allows the MUL unit to be used to perform some simple arithmetic tasks to augment the ALU.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
16-bit forms	
subm S_i, S_j, S_k	subtract S_i from S_j and write the result to S_k

Operand Values: S_i any scalar register r0-r31.
 S_j any scalar register r0-r31.
 S_k any scalar register r0-r31.

Condition Codes: Unchanged by this instruction.

MAIN BUS

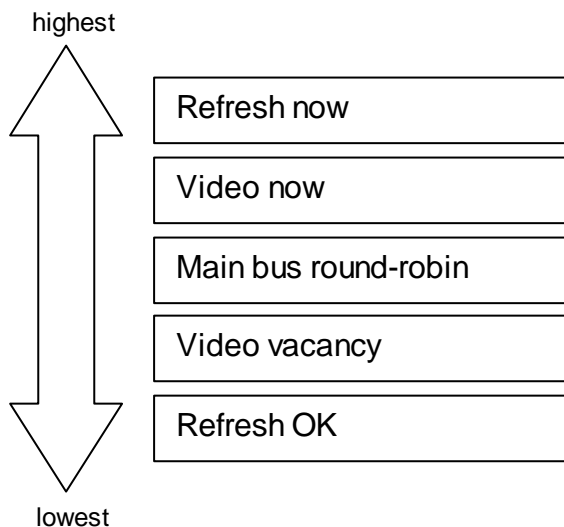
The Main Bus is used by the MPEs, the video display generator, and the audio DMA channel to transfer data to and from SDRAM, or from one location to another in internal memory. It has a 32-bit data channel transferring data at a maximum rate of one long per clock cycle.

The Main Bus is shared by arbitration between the bus masters. Each bus master supplies a priority-encoded request to the arbiter, which will eventually respond by granting the bus and accepting a command. Normally, the bus must be requested again for each DMA command.

Arbitration

Several bus masters share the Main Bus, and so an arbiter is required to handle simultaneous requests from them. Bus arbitration allows us to define maximum bus latency for all bus masters, and to prioritize requests between masters.

The bus is allocated at two levels. The primary allocation level of the bus is made as follows:



A bus owner in this hierarchy can be interrupted at any time by a higher priority. The bus owners are:

- Refresh now means that the internal refresh timer requires refreshes be carried out immediately in order to retain the DRAM contents.
- Video now means that one of the video output channels has a low FIFO level, and that it must be refilled as soon as possible to avoid underflow. The level at which this occurs is programmable. There is no priority between the four video channels.
- The Main Bus round robin is described below, and is the secondary bus allocation mechanism. MPE transfers are included in this.
- Video vacancy means that a video output channel has room in its FIFO for more data, and so a fetch may occur.
- Refresh OK means that the internal refresh timer indicates that a refresh may be carried out. The majority of refreshes will occur at this level.

Within this hierarchy lies a secondary arbitration level, shown above as the Main Bus round robin. This is where all MPE DMA transfers, and the audio output DMA, occur. At this arbitration level, DMA commands are executed atomically, with the bus being arbitrated on each command.

The arbitration scheme allows each bus master to request the bus at one of four bus priorities. Each bus priority is serviced on a round-robin basis, which may be considered to be a series of rings, thus:

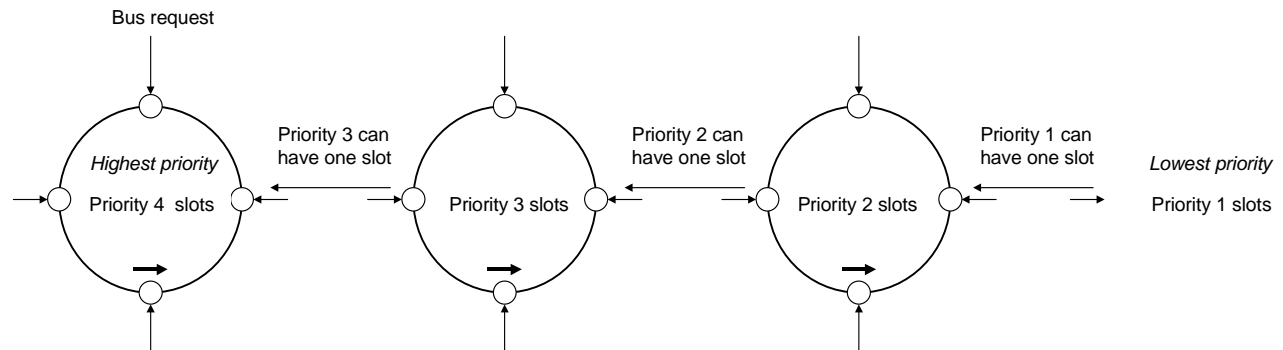


Figure 3 - Round robin prioritized bus arbitration

The bus is available to a requester for one time slot. This is the time it takes to complete the requested transfer, and you should normally keep it short to limit the maximum bus latency of requesting devices. Linear transfers cannot be particularly long, but you should be aware that while it is possible to clear the screen with a single pixel mode transfer, this will lock out all the other DMA requesters for a long time, and any real time needs, such as audio output, will not be met.

Slots are available at one of four priorities, where priority four is the highest. Priority four requesters are serviced in turn. At each pass round the allocation loop one slot can be made available to priority three bus masters, unless this feature is disabled in which case priority four masters can hog all the available bus bandwidth. Similarly, one slot in the priority three loop can be made available to priority two bus masters, and so on. These priority gateways, when open, guarantee a maximum bus latency to all bus masters, dependent on their requesting priority. Each gateway can be independently enabled.

The slots as drawn do not necessarily consume any time. If a slot is not requested it is skipped. Therefore, if there are no priority three requests, all available bus bandwidth is available at priority two, and so on.

The available priority levels vary according to the requirements of each bus master. Generally, priority four is used only for low latency real-time requirements. See the descriptions of each bus master for more details.

Main Bus DMA Controller

Introduction

Main Bus DMA is the only means of transferring data between DRAM (also referred to as SDRAM) and NUON. The DMA mechanism is exposed to MPEs, and is also built into the hardware of the video and audio output channels.

You create a Main Bus DMA command by building a DMA command structure in MPE memory, and then writing its address to the DMA command pointer register.

The following chain of events makes up a DMA transfer:

1. *Request the bus.* This is done for the MPEs when the DMA command pointer register is written. The bus arbitration mechanism and its associated latencies are described in detail on the Main Bus section.
2. *Transfer the DMA command.* For the MPEs, this is copied from MPE data RAM. The command gives the internal and external addresses of the data associated with the transfer, its length, and various other flags to control such things as linear or XY addressing. When this is complete the pending flag is cleared.
3. *Transfer DMA data.* This is a burst of data transfer at a rate of up to 216 Mbytes / second.

This section describes DMA from the perspective of the MPE programmer, as the MPE is the only device that can directly set up DMA commands under programmer control. However, some peripheral devices also use DMA to transfer data to and from SDRAM, so this section may also be relevant in that wider context.

All SDRAM transfers occur through the DMA controller. It manages the SDRAM interface and all cycles on the internal Main Bus. It is highly optimized to get the best SDRAM and Main Bus bandwidth, and can perform operations in parallel; such as fetching video data during DMA command transfers or internal to internal DMA; or reading a DMA command at the same time as writing DMA data. DMA latencies can be significant, especially when the bus is busy; and the Comm Bus is better suited to low latency inter-process communication.

DMA for the MPE

DMA transfer is the only way that MPEs can transfer data to and from SDRAM. These transfers occur over the Main Bus.

The MPEs program a DMA transfer through a command held in data RAM. The DMA transfer is initiated by writing the address of this command into the DMA command pointer register. The DMA command structure must lie on a vector (128-bit) address boundary.

DMA transfers are either linear or bi-linear:

Linear transfers copy from 1 to 128 longs of data from MPE RAM to SDRAM (writing), or from SDRAM to MPE RAM (reading). The transfer must be aligned to a long boundary in both memories.

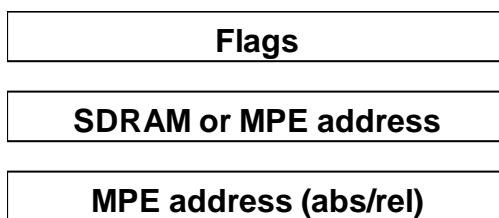
Bi-linear transfers are for pixels. They can transfer a rectangle of pixels between the MPE RAM and SDRAM, and these rectangles include both horizontal and vertical single pixel strips, so that polygon rendering can be optimized for the best DMA performance. The DRAM storage of pixels is a complex arrangement intended to optimize memory performance. The base address for bi-linear transfers must be on a 128-byte boundary. If the cluster bit in the command-mode bits is set, then the base address must be on a 512-byte boundary.

DMA Commands

DMA commands can take one of the following formats. Each of the fields in the diagrams below corresponds to one long word of data, and the individual bits within the fields are discussed in more detail further on. Note that all unused command bits **must** be written with zero.

Linear transfer command format

This format is used to transfer linear blocks of long data.



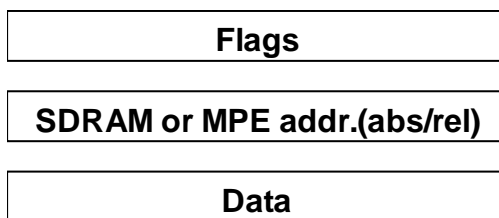
The flags control the direction of the transfer, its length, and some other special functions.

The first address field is the destination address of a write, or the source address of a read. This is sometimes called the base address. It may be anywhere in SDRAM or internal MPE memory, and is always a system address (absolute).

The second address field is the source of address of a write, or the destination address of a read. This is sometimes called the internal address. It must be in internal MPE memory, but may be specified as a system address when the REMOTE bit is set, or as a local MPE address when the REMOTE bit is clear.

Direct write command format

This format is used to write data embedded directly in the command. The DIRECT flag is set to enable this mode.



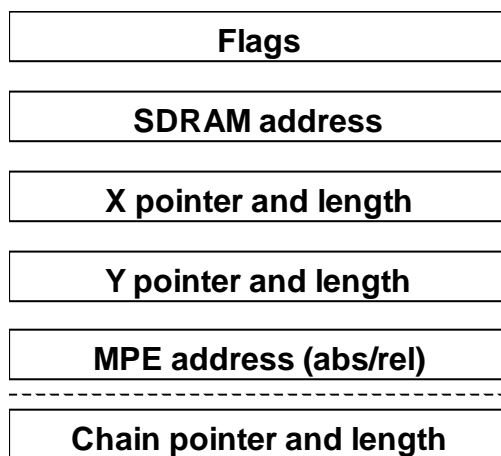
The address field is the destination address. It may be anywhere in SDRAM or internal MPE memory, and may be specified as a system address when the REMOTE bit is set, or as a local MPE address when the REMOTE bit is clear.

Normally this is used to transfer a single long of data, however if the length of the transfer is more than one, the same data will be automatically duplicated across many locations.

Pixel transfer command format

This command format is used to transfer pixels. The transfer can be a horizontal or vertical strip of pixels, or a rectangular area.

Pixel commands can be *chained*, which allows a series of adjacent horizontal or vertical strips of pixels to be transferred as a single operation, with the start position and width updated for each strip. This is useful for optimizing the transfer of polygon data.



This field only used when CHAIN is set, and is repeated for as many scan lines as required.

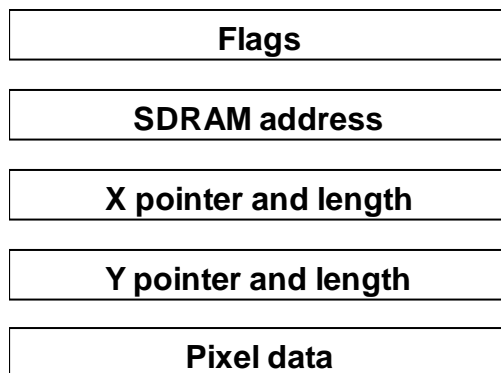
The first address field is the destination address of a write, or the source address of a read. This is sometimes called the base address. It must be on a 128-byte boundary in SDRAM. If the cluster bit in the command-mode bits is set, then the base address must be on a 512-byte boundary in SDRAM.

The X and Y fields contain a start position and a length for both X and Y.

The second address field is the source of address of a write, or the destination address of a read. This is sometimes called the internal address. It must be in internal MPE memory, but may be specified as a system address when the REMOTE bit is set, or as a local MPE address when the REMOTE bit is clear.

Direct write pixel command format

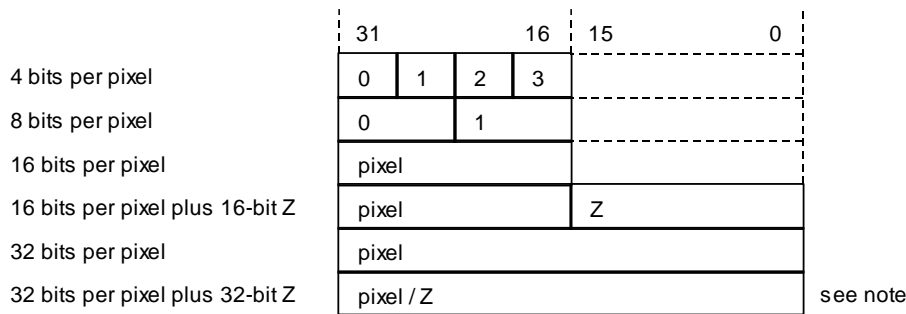
This command format is normally used as a convenient mode to directly transfer a single pixel. It can also be used to fill a number of pixels with the same value. It is more efficient than regular pixel DMA.



The address field is the destination address of a write, or the source address of a read. This is sometimes called the base address. It has the same restrictions as the base address for normal Pixel Transfers.

The X and Y fields contain a start position and a length for both X and Y.

Pixel data is stored in the command like this:



Note that the same data is used for both the pixel and the Z field in 32 bits per pixel plus Z mode. Note also that in pixel mode 8 the direct data should be 16 bits per pixel!

DMA Command Fields

These are the long words that make up the DMA command. Any unused bits must be set to zero.

DMA Command – Flags

Bits	Name	Description
31	PLAST	Pipelined end flag for chained DMA. Use of this flag is optional, and it should be set four long words before the last command for optimal efficiency. See the discussion on Chained DMA below.
30	BATCH	This flag allows multiple DMA operations to be run together without any software intervention. See the discussion of batch DMA below.
29	CHAIN	This flag allows multiple bi-linear transfers to occur as a single operation. See the discussion of DMA chaining below and the chain command long word below.
28	REMOTE	When this bit is set, the internal address field is interpreted as a Main Bus address, and therefore can lie anywhere in internal memory. The MSB of this address is implicitly zero. When this bit is clear, the internal address field is interpreted as an MPE internal address to the requesting MPE. This allows code to be written that will run on any MPE. (The internal address of all MPEs is the Main Bus address of MPE0.)
27	DIRECT	Direct mode flag. When this is set write data is part of the command itself, instead of being pointed to by the command. See the discussion above of the direct modes. When this flag is set it also has the effect of turning on the DUP flag whatever the state of the DUP bit in the command.
26	DUP	Duplicate data. When this flag is set for writes, only the first long word is read from internal memory, and this long is repeated over the entire DMA transfer. This is useful for memory clears, fills, and so on. See also direct mode.
25	TRIGGER	Can trigger the capture of a breakpoint. Normally should be set to zero.
24	ERROR	Used for debug only, set to zero.
23-16	LENGTH / XSIZE	For linear address transfers, this gives the length of the transfer in longs. Valid values are 1-127. For bi-linear address transfers, this gives the X size of the addressed pixel bit-map. The value written here is the width of the pixel-map, i.e. the number of pixels divided by eight, except for the MPEG modes, where it is the width in luminance pixels divided by sixteen (i.e. the number of macro-blocks).
15-14	TYPE	DMA type, as described below.
13	READ	Read flag, as described below.
12	DEBUG	Debug flag, must be set to zero.

Bits	Name	Description
11-0	MODE	DMA command mode, as described below.

The DMA transfer type, read flag and mode is encoded as follows. The mode bits are discussed in the section describing each mode, see below.

Type	Read	Description	Length	Address	Mode bits BA9876543210
0	0	Linear write	1-64 longs	Linear	??????ULIIS
0	1	Linear read	1-64 longs	Linear	??????ULIIS
2	0	Motion predictor write	n/a	Bi-linear	PmMmmSmRCFFF
2	1	Motion predictor read	n/a	Bi-linear	PmMmmSmRCFFF
3	0	Pixel write	X, Y size	Bi-linear	C?BVPPPPZZZA
3	1	Pixel read	X, Y size	Bi-linear	C?BVPPPPZZZA

Linear Transfer Command Mode Bits

Linear transfers are the standard means of DMA transferring linear long data. Linear transfers may be between internal memory and SDRAM, or can be from internal memory to internal memory, and so can pass data between MPEs.

Linear transfer mode can also be used to read and write single bytes and words. When the IIS field is set to 001, the UL field selects byte transfer mode.

Linear transfer mode bits are:

Bits	Name	Description																											
4-3	UL	Used for byte or word transfer when the IIS field is set to 001, these control if the transfer is byte 0, byte 1 or word. The base address is specified to a word boundary. 00 Must be set to zero when IIS is not 001, illegal when it is 01 Byte 1 only (bits 0-7) 10 Byte 0 only (bits 8-15) 11 Word transfer																											
2-0	IIS	Control writing sparse linear data, for example when building the audio output buffer with the data from one audio channel. When the S bit is set, the base address may be on any word boundary, when it is clear, the base address must lie on a long boundary. This function is only available to the MPEs. <table> <tr> <th>IIS</th><th>Action</th><th>Description</th></tr> <tr> <td>000</td><td>ULULULULULULULUL</td><td>contiguous data</td></tr> <tr> <td>010</td><td>UL--UL--UL--UL--</td><td>alternate longs</td></tr> <tr> <td>100</td><td>UL-----UL-----</td><td>every fourth long</td></tr> <tr> <td>110</td><td>UL-----</td><td>every eighth long</td></tr> <tr> <td>001</td><td></td><td>byte mode, see above</td></tr> <tr> <td>011</td><td>U-L-U-L-U-L-U-L-</td><td>alternate words</td></tr> <tr> <td>101</td><td>U---L---U---L---</td><td>every fourth word</td></tr> <tr> <td>111</td><td>U-----L-----</td><td>every eighth word</td></tr> </table>	IIS	Action	Description	000	ULULULULULULULUL	contiguous data	010	UL--UL--UL--UL--	alternate longs	100	UL-----UL-----	every fourth long	110	UL-----	every eighth long	001		byte mode, see above	011	U-L-U-L-U-L-U-L-	alternate words	101	U---L---U---L---	every fourth word	111	U-----L-----	every eighth word
IIS	Action	Description																											
000	ULULULULULULULUL	contiguous data																											
010	UL--UL--UL--UL--	alternate longs																											
100	UL-----UL-----	every fourth long																											
110	UL-----	every eighth long																											
001		byte mode, see above																											
011	U-L-U-L-U-L-U-L-	alternate words																											
101	U---L---U---L---	every fourth word																											
111	U-----L-----	every eighth word																											

Pixel Command Mode Bits

Pixel mode is generally used for all pixel transfers. Pixel transfer must be between internal memory and SDRAM. The base address for bi-linear transfers must be on a 512-byte boundary.

Bits	Name	Description
11	C	Cluster addressing.
9	B	Backwards flag B. See the discussion of backward pixel transfer below.
8	V	Transfer direction. This is used to make narrow vertical strips much more efficient in their use of DRAM. The normal pixel order of X then Y in MPE memory is reversed to be Y then X, so that a horizontal strip of pixels in MPE data RAM maps onto a vertical strip in DRAM. 0 Horizontal (normal) 1 Vertical When this bit is set the A and B flags swap function.
7-4	PPPP	Pixel types. This control the pixel type assumed for the transfer, and the mapping between MPE RAM types and DRAM types. The types are described in the table below:
3-1	ZZZ	Z comparison for writes. Compares the target pixel Z, which is the Z of the pixel already present in DRAM, with the transfer pixel Z, which is the pixel being transferred from the MPE RAM, and inhibits the write if the compare condition is met. Value Inhibit write if: 0 never 1 target pixel Z < transfer pixel Z 2 target pixel Z = transfer pixel Z 3 target pixel Z <= transfer pixel Z 4 target pixel Z > transfer pixel Z 5 target pixel Z != transfer pixel Z 6 target pixel Z >= transfer pixel Z 7 pixel only write Value 7 is a special case flag that makes the DMA write transfer write only to the pixel values and leave the Z undisturbed. No Z compare is performed.
0	A	Backwards flag A. See the discussion of backward pixel transfer below.

Motion Predictor Command Mode Bits

Motion predictor mode is specific to the MPEG decoder function, and is not available for other functions.

Bits	Name	Description
11	P	Set for pixel mode, clear for MCU mode
10	m	This flag is passed to the MCU
9	M	Set for manual length, clear for auto (implied by the RCFFF bits, below)
8-7	mm	These flags are passed to the MCU
6	S	If asserted, it indicates that 4:3 scaling needs to be done. This applies during a write-back operation only
5	m	This flag is passed to the MCU
4-3	RC	00 Luma 10 Cr/Cb (pixel mode only) 01 Cr 11 Cb
2-0	FFF	Field / frame control 000 16x16 frame 001 16x16 frame 010 16x16 field 011 16x16 field 100 16x8 field top 101 16x8 field top

Bits	Name	Description
110	16x8	field bottom
111	16x8	field bottom

DMA Command – SDRAM Address

Bits	Name	Description
30-1	BASE	SDRAM address of the transfer. This is the DRAM pointer for a linear transfer operation, or the base address of the bit-map for a bi-linear operation. The base address for linear transfers is normally long aligned, although it may be a word address for the special case word & byte transfer mode. The base address for bi-linear transfers must be on a 128-byte boundary. If the cluster bit in the command-mode bits is set, then the base address must be on a 512-byte boundary. The address must be in the range \$40000000 - \$7FFFFFFE. The bit range 30-1 implies only that the bottom bit of the address is always zero. No shifting is required on a normal byte address.
31	PLAST	Pipelined end flag for chained DMA. Use of this flag is optional, and it should be set four long words before the last command for optimal efficiency. See the discussion on Chained DMA below.

DMA Command – MPE Address

Bits	Name	Description
30-2	BASE	MPE address of the transfer. This is the MPE space pointer for a linear or a bi-linear transfer operation. It should be long aligned. If the REMOTE flag is set in the command, it may point anywhere in the MPE space, i.e. in the range \$2000 0000 - \$2FFF FFFC. If the REMOTE flag is not set it is an address within the current MPE. As all the MPEs appear to be in the MPE0 space internally, this means it must lie in the range \$2000 0000 - \$207F FFFC. The bit range 30-2 implies only that the bottom two bits of the address are always zero. No shifting is required on a normal byte address.
31	PLAST	Pipelined end flag for chained DMA. Use of this flag is optional, and it should be set four long words before the last command for optimal efficiency. See the discussion on Chained DMA below.

DMA Command – X pointer and length

Bits	Name	Description
31	PLAST	Pipelined end flag for chained DMA. Use of this flag is optional, and it should be set four long words before the last command for optimal efficiency. See the discussion on Chained DMA below.
25-16	XLEN	X length. This is the width of the bi-linear transfer, in pixels. For mode 1 this must be a multiple of 4, for mode 3 it must be a multiple of 2.
10-0	XPOS	X pointer. This is the start X position for a bi-linear transfer, in pixels. For mode 1 this must be a multiple of 4, for mode 3 it must be a multiple of 2. For a motion predictor command this is a 10.1 bit value, for pixel commands it is an 11-bit integer.

DMA Command – Y pointer and length

Bits	Name	Description
31	PLAST	Pipelined end flag for chained DMA. Use of this flag is optional, and it should be

Bits	Name	Description
		set four long words before the last command for optimal efficiency. See the discussion on Chained DMA below.
25-16	YLEN	Y length. This is the height of the bi-linear transfer, in pixels.
10-0	YPOS	Y pointer. This is the start Y position for a bi-linear transfer, in pixels. For a motion predictor command this is a 10.1 bit value, for pixel commands it is an 11-bit integer.

DMA Command – Chain pointer and length

Bits	Name	Description
31	PLAST	Pipe-lined end flag for chained DMA. Use of this flag is optional, and it should be set four long words before the last command for optimal efficiency. See the discussion on Chained DMA below.
30	LAST	Last flag. This bit should be set on the last chain command word, unless the PLAST flag was set four long words previously, in which case this does not have to be set.
25-16	ZLEN	X / Y length. This is the width or height of the next strip of a chained bi-linear transfer, in pixels. It is X if the transfer direction in the mode is horizontal, it is Y if the mode is vertical. For mode 1 if this is X then this must be a multiple of 4, for mode 3 it must be a multiple of 2.
15-14	STEP	Step. This controls how the Y pointer for horizontal DMA, or the X pointer for vertical DMA, is modified before this strip: 0 increment by 1 1 no change 2 decrement by 1
10-0	ZPOS	X / Y pointer. This is the start position for next strip of pixels. For horizontal DMA, it is the X position, for vertical DMA it is the Y position. Its is subject to the same restrictions as the X and Y pointer above, as appropriate. It is an 11-bit integer.

Chained DMA

Pixel (bi-linear) transfers may be chained. This allows multiple scan lines of a polygon, for example, to be transferred in a single DMA operation, and can be used to make much more efficient use of the bus, particularly for small polygons. A chained DMA sets up an initial transfer of a single strip of pixels, which may be either horizontal or vertical, and then the chain command long sets up a new start position and length.

As an example, if the transfer direction is set to horizontal, then the initial DMA transfers a horizontal strip of pixels. The chain command that follows supplies a new X start position, a new X width, and can either increment, decrement, or have no effect on the Y pointer. The pixel data buffer in MPE data RAM in is in one contiguous block.

This is useful for very small polygons, or for reading the data at the edges of an area for anti-alias operations. It is also very useful for line drawing.

Chaining is best suited for applications where you are limited by bus bandwidth, as it improves bus performance, at a cost of increased program overhead. If you are processor bandwidth bound you may choose to continue to use DMA transfers of single strips.

Chaining is enabled by setting the CHAIN bit in the DMA command flags. The end of the chain may be flagged in one of two ways:

1. The last chain command word should have the LAST bit set in it.
2. The command word four longs before the last long word of the chain should have the PLAST bit set. This allows the DMA to pipeline the end of the chain, and will give a significant performance boost.

If both flags are present in a chained command, then whichever would finish the chain first takes priority.

Batch DMA

Batch DMA transfers allow the programmer to set up multiple DMA transfers without having to wait for each to complete, and scheduling the next. Batch DMA transfers do not occur in the same slot, as the bus is re-arbitrated between them, but they have the advantage that no program overhead is required to schedule the next transfer.

Batch DMA command blocks should start on contiguous vectors in memory, so you may have to insert padding between them.

Pixel Transfer Types

Some type conversion can occur between the MPE memory and SDRAM. The types in DRAM are explained below; the types in MPE memory are explained in the MPE section; however where the types overlap the same number refers to the same mode.

Type	MPE mode		DRAM mode	Notes
	ZZZ=7	ZZZ≠7		
0	2	2	5Z	Allows just the Z field to be written of mode 5 DRAM data.
1	1	1	1	4 bit pixels. X position and X length must be multiples of four.
2	2	2	2	16 bit pixels.
3	3	3	3	8 bit pixels. X position and X length must be multiples of two.
4	–	4	4	32 bit pixels.
5	2	5	5	16 bit pixels with 16 bit Z. Z flags are used.
6	4	6	6	32 bit pixels with 32 bit Z. Z flags are used.
7	4	4	6Z	Allows just the Z field to be written of mode 6 DRAM data.
8	4	4	2	32 bit pixels in MPE, 16 bit pixels in DRAM
9	2	5	7	16/16 pixels in MPE, 16/16 triple buffer map C in DRAM. Z flags are used.
10	2	5	8	16/16 pixels in MPE, 16/16 triple buffer map B in DRAM. Z flags are used.
11	2	5	9	16/16 pixels in MPE, 16/16 triple buffer map A in DRAM. Z flags are used.
12	2	2	CZ	Allows just the Z field to be written of a 16/16 triple buffer map in DRAM.
13	2	5	A	16/16 pixels in MPE, 16/16 double buffer map B in DRAM
14	2	5	B	16/16 pixels in MPE, 16/16 double buffer map A in DRAM
15	2	2	DZ	Allows just the Z field to be written of a 16/16 double buffer map in DRAM.

Notes

- Types 0, 5, 6, 9-11 and 13-14 will over-write just the pixel field if the Z flags are set to 7 for a pixel only write.

MPE DMA Control and Status Register

The DMA control and status register allows each MPE to control and determine its DMA status.

Bits	Name	Description
31-24	done_cnt_wr	Write done count. This counter is used only when trying to set up multiple overlapping read and write transfers. It is incremented by the hardware whenever a write transfer completes, and is decremented by software. Valid values are between 0 and \$1D. Two error conditions may also exist, \$FE for overflow, and \$FF for underflow.
23-16	done_cnt_rd	Read done count. This counter is used only when trying to set up multiple overlapping read and write transfers. It is incremented by the hardware whenever a read transfer completes, and is decremented by software. Valid values are between 0 and \$1D. Two error conditions may also exist, \$FE for overflow, and \$FF for underflow.
15	cmd_error	Command error. The DMA controller has found an error while processing a command. There are Comm Bus registers in the DMA controller to determine what was wrong in detail.
14	dmpe_error	Command pointer error. One of the following things has occurred: <ul style="list-style-type: none">The command pointer write was to an invalid rangeThe command pointer incremented past a 32K byte range.The command pointer was written while a transfer was pending.
11	done_cnt_wr_dec	Decrement write done count. When a one is written to this bit the write done counter is decremented. This should be performed when necessary to clear the interrupt condition.
10	done_cnt_rd_dec	Decrement read done count. When a one is written to this bit the read done counter is decremented. This should be performed when necessary to clear the interrupt condition.
9	done_cnt_enable	Done count enable. Writing a one to this bit enables the read and write done counter mechanism. This has a variety of effects, discussed below. When read this bit returns the enable status.
8	done_cnt_disable	Done count disable. Writing a one to this bit disables the read and write done counter mechanism. This has a variety of effects, discussed below. This bit always reads as zero.
6-5	priority	Bus priority. Sets the bus priority for MPE DMA transfers.
4	pending	Command pending. This flag means that the DMA command pointer must not be written to. This bit is read only.
3-0	active	DMA active level, this gives the number of DMA commands that have been accepted by the DMA controller, but whose data transfer is not yet complete. In theory, this can reach a level of around 6 or 7. These bits are read only.

Simple DMA Control

Bits 6 and below are used to control the DMA at its simplest level. All the higher bits should be written as zero. See the discussion on out-of-order completion to see if this is all you need.

The DMA pending flag is used to indicate that the DMA command pointer holds the address of a DMA command that has not yet been transferred to the main DMA controller, because the MPE is still waiting to be granted the bus. Whenever it is clear, a new DMA command address may be written to the command pointer, even if previous DMA transfers have not yet completed.

The DMA active level is a counter that indicates which DMA buffers are still in use and cannot be read for reads, or re-used for writes. When the active level count is zero, all DMA data transfers are complete. It increases by one every time a command is read from this MPE by the DMA controller, and decreases by one every time a DMA transfer to or from this MPE completes.

DMA Errors

Two flags, the command error bit, and the command pointer error bit, flag errors to the MPE. When either of these bits is set the DMA error interrupt to the MPE is held high. Writing a one to the appropriate bit will reset the error.

Out-of-order DMA Completion

All DMA transfers take place in the programmed order as far as the SDRAM is concerned. However, because the DMA pipelines for both read and write are several clock cycles deep, overlapped DMA transfers can appear to complete out-of-order at the MPE. If a DMA is programmed to write to SDRAM as soon as the pending bit clears from a DMA that reads from SDRAM, then the write data can be fetched from the MPE before the read data has arrived at the MPE. Although at the SDRAM the read occurs before the write, if the write transfer was using the read data, then the wrong data would get written.

The only way for software to deal with this effectively is to maintain separate track of read transfers and write transfers. This allows you to always determine what has actually completed when the active level drops. The done counters in the MPE Main Bus DMA control register allow this more sophisticated control to be enabled when required.

Overlapped DMA Control

Writing one to the done count enable bit enables the advanced DMA control functionality. This changes the DMA done interrupt from a pulse to a level that is active whenever the read done counter or the write done counter have valid values greater than 0.

Writing a one to the done count disable bit disables the done counters. If you write a one to both the done count enable bit and the done count disable bit both the read done counter and the write done counter are flushed, any errors are cleared, and the done count enable bit is set.

After processing an interrupt, write a one to either the decrement write done count bit or the decrement read done count bit to decrement the appropriate done counter.

When reading this register:

- The done count disable bit returns zero.
- The done count enable bit returns the enable status.
- The decrement read done count bit returns one if the read done counter has valid values greater than zero.
- The decrement write done count bit returns one if the write done counter has valid values greater than zero.

When the done count enable bit is disabled counters DO still increment!

Caveats: When using batch mode the LAST command in the batch determines whether the read done counter or the write done counter is incremented. DANGER: if a command error occurs during a batch transfer the transfer will be aborted. The type (read/write) of the aborted command will determine which done counter is incremented. The command error will also be set and further information about what went wrong can be determined by querying the DMA controller.

Backward Pixel Transfers

The backward flags for pixel DMA transfers have the effect shown on pixel DMA:

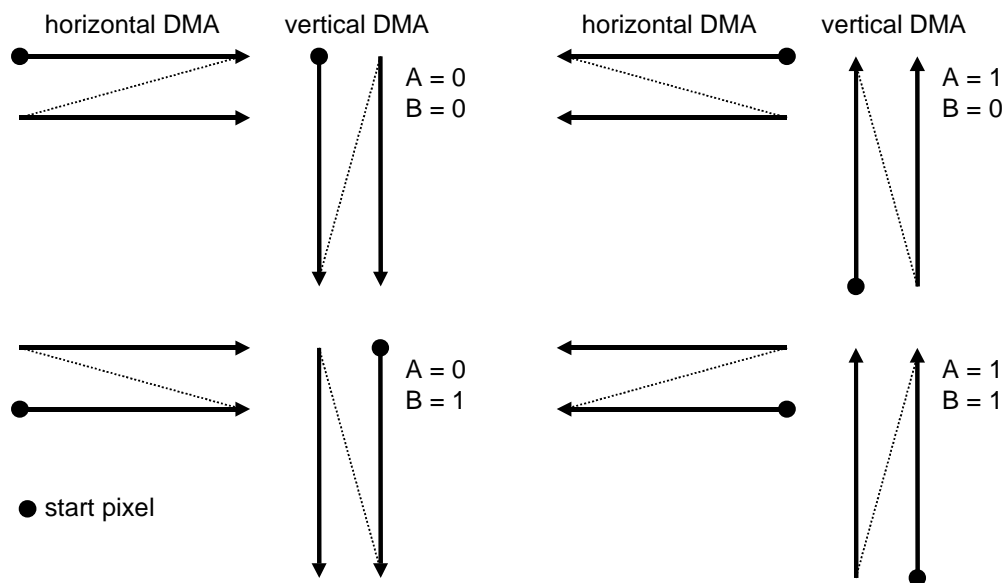


Figure 4 - Backwards DMA Types

These flags can be used to give the reflections and rotations as shown.

DMA Pixel Types

Storage Formats

Each type of pixel type, and linear data, are stored packed in memory in a format optimized to their type. This means that you should not store data in one format and expect to make sense of it in another. Memory for mixed modes should always be allocated on 512-byte boundaries, and in 512-byte increments.

Details of the storage formats are beyond the scope of this document, but if you obey the rules above then you need not be aware of the layout details.

Pixel Type 1 – 4 bit pixels

Type one pixels are four bits. The value represents an index into an arbitrary look-up table, and so have no fixed relationship with the physical appearance. These are sometimes useful for memory efficient texture maps, and can be used for memory efficient overlay display buffers.

All DMA operations on them must have both X position and X size as multiples of four.

Pixel Type 2 – 16 bit pixels

Type two pixels are sixteen bits per pixel. They represent a physical color, thus:

Y	Cr	Cb
15 10 9	5 4	0

When these pixels are used for display generation zeroes are added in the least significant positions to increase them to 8 bits per field.

Pixel Type 3 – 8 bit pixels

Type three pixels are eight bit. The value represents an index into an arbitrary look-up table, and so have no fixed relationship with the physical appearance. These are used for memory efficient overlay display buffers.

All DMA operations on them must have both X position and X size as multiples of two.

Pixel type 4 – 32 bit pixels

Type 4 pixels are 32 bits per pixel. They represent a physical color, thus:

Y	Cr	Cb	control
31 24 23	16 15	8 7	0

They can be used by load and store pixel instructions, and can be present in DRAM and in MPE RAM.

Pixel type 5 – 16 bit pixels with 16 bit Z

Type 5 pixels are 16 bits per pixel, with an associated 16-bit control value, usually used for a Z-buffer depth. The 16 pixel bits represent a physical color, thus:

Y	Cr	Cb	Z
31 26 25 21 20 16 15			0

When these pixels are used for display generation zeroes are added in the least significant positions to increase them to 8 bits per field.

Pixel Type 6 – 32 bit pixels with 32 bit Z

Type 6 pixels are 32 bits per pixel, with an associated 32-bit control value, usually used for a Z-buffer depth. The fourth byte of the pixel value is normally unused, but may be accessed. See the discussion of pixel types in the MPE section. These pixels represent a physical color and Z value, thus:

Y	Cr	Cb	unused	Z
63 56 55 48 47 40 39 32 31				0

COMMUNICATIONS BUS

The Communication Bus allows units within NUON to communicate over a low latency, high-speed bus. This is a 32-bit bus running at the full clock rate, and therefore has a bandwidth in excess of 200 Mbytes per second. This bus is quite independent from the main and other busses, and provides an alternative to passing data in memory. This bus is in many ways analogous to a simple network.

The Communication Bus is used both as means of inter-processor communication, and as a means of communication with peripherals. The following devices have a Communication Bus interface:

- The MPEs
- Video output generator
- Video input channel
- Audio output and input channels
- System IO for user interface, controllers and media control
- ROM interface
- Main Bus DMA controller
- The external host processor on the System Bus
- System debug controller
- MPEG Block Decode Unit
- Coded data interface

Each device attached to the Communication Bus has a transmit buffer, and a receive buffer. Each of these buffers can contain 128 bits of data, along with a Communication Bus address.

Communication Bus Identification Codes

Each MPE is allocated a logical identification code, so that communication is by logical device, rather than physical. This allows processes to communicate without having to be aware of the physical location of each other. All other devices have a physical identity. At power on, MPEs are assigned their MPE number as an ID by default.

Communication Bus identification numbers are allocated 7-bit values as follows:

ID dec	hex	Function
0-63	00-3F	Logical codes for MPEs
65	41	Video output controller
66	42	Video input controller
67	43	Audio interface
68	44	Debug controller
69	45	Miscellaneous IO interface
70	46	ROM Bus interface
71	47	DMA controller
72	48	External host
73	49	BDU

74	4A	Coded Data interface
75	4B	Serial Peripheral Bus

MPEs may send packets to themselves, but no other device can do this.

Data Transfer Protocol

Before you try to transmit a packet, you must first make sure that the local transmit data buffer is empty unless you can be sure that it is already empty. Then you must write the target device address into the Communication Bus control register unless it is already set up, and then the transmit data itself. The act of writing data into the transmit data buffer marks it as full, and initiates the transfer mechanism. The transmit buffer full flag is set until the hardware has transmitted the data, or the transmission fails.

The Communication Bus interface hardware will then request a transfer of data to the selected target by requesting the Communication Bus. The bus is allocated on a round-robin basis between requesting transmitters. When the transmitter is allocated the bus it presents the target ID for the transfer, and its own ID. Two things can then happen. If the target is able to receive data, that is its receive buffer is not full or disabled, then the data is transferred over the bus. If the target is unable to receive the bus transaction terminates. In either case the transaction is then complete, and the bus is re-arbitrated.

The tables below explain the Communication Bus protocol. Each line represents one clock cycle.

Transfer to receiver with buffer empty:

Requester action	Bus contents	Controller action
Bus request		
... 0- L_{max} clock cycles ...		
	<i>previous transfer</i>	Bus acknowledge
Present target ID	Target ID and Sender ID	Check for target full, so continue in this case.
Present data	Data 1	
	Data 2	
	Data 3	
	Data 4	<i>next bus acknowledge</i>
	<i>next transfer target ID</i>	

Transfer to receiver with buffer full:

Requester action	Bus contents	Controller action
Bus request		
... 0- L_{max} clock cycles ...		
	<i>previous transfer</i>	Bus acknowledge
Present target ID	Target ID and Sender ID	Check for target full, abort the transfer in this case.
	Idle	
	<i>next transfer target ID</i>	

Note the following about this protocol:

- The maximum bus latency, L_{max} , is normally given by five clock cycles times the maximum number of simultaneously requesting Communication Bus masters. This

number can be controlled in an application by suitable restrictions on use of the Communication Bus.

- The internal data bus is used to carry the target ID to the controller and the sender ID to the target. This means packets are actually five clock cycles in length.
- If the receive buffer of the target device is empty, the data will be transferred when the transmitter is allocated the bus. However, if the receive buffer is full or disabled, one of two things can happen:
 - If the transmit retry flag is set, the hardware will continue to request the bus, and every time the bus is granted to it, it will attempt to transfer the data. The transmit buffer full flag indicates that this process has not yet succeeded. This will tie up the transmit port. Transmitters waiting to transmit data can therefore occupy a significant proportion of the bus bandwidth if the receiver is slow to empty its buffer.
 - If the transmit retry flag is clear, the transmit failed flag is set, the transmit buffer full flag is cleared, and the hardware goes idle. This allows the transmitter to give up on the transfer and take some other action, such as attempting to send the data to another target.

A receiver can refuse to accept Communication Bus data by setting the receive disable flag. This means that all transmission attempts to it will fail on target full, even if its receive data buffer is empty.

When data is received a flag is set that may be polled, and an interrupt can be generated. The ID of the transmitter may be read. When the receive data is read, the receive buffer is marked as empty, and another packet can be received unless receive is disabled.

Data Flow Control

The Communication Bus can be used as a means of inter-process communication, as it can generate an interrupt when a packet is received. This allows any processor to interrupt any other, and pass it a message at the same time.

If data is being passed between processors in some multi-processor pipeline, then it is necessary to control the flow, particularly if a transmitter could send the data to one of several receivers, depending on which is able to take it. In this case a receiver can use the receive disable mechanism to flag that it is not willing to accept data. If a transmitter has the transmit retry field clear, it can then detect that a transmit has failed, and perhaps attempt to transmit the same data packet to another target.

In the reverse situation, where a receiver could receive from one of several transmitters, it could use the same mechanism to poll them in turn, or just enable reception, and wait for data. If the data set being transferred was larger than one Communication Bus packet, it would have to be prepared to receive, and deal with, a packet from another transmitter in the middle of it.

Communication Bus Control Flags

Each processor-controlled interface to the Communication Bus has the following control fields:

Communication Bus status and control:

Bits	Read / Write	Description
31	R	Receive buffer full. This flag indicates that there is a received packet in the receive data buffer and the received source ID fields. This flag is cleared (and these fields can then be

Bits	Read / Write	Description
		over-written) when the receive buffer is read.
30	RW	Receive disable. This flag should be set to prevent reception. All transmit attempts to this receiver will fail while this flag is set. If this flag is set while the receive buffer is empty, the receive buffer full flag should be checked afterwards, in case a packet was received just before this flag was set.
16–23	R	Received source ID. This indicates the Communication Bus ID of the last data packet to be received. This value should be read before the receive data buffer, as another packet might be received as soon as the receive buffer is empty.
15	R	Transmit buffer full. This flag indicates that the hardware is attempting to transmit the data in the transmit data buffer to the transmit target ID. If the retry flag is set then this bit will remain set until a successful transmission has occurred, if the retry flag is clear, then this bit will always be cleared after first transmission attempt, and the transmit failed flag will reflect what happened.
14	R	Transmit failed. When the transmit retry flag is clear, this flag will be set when a transmit attempt fails because the receive buffer is full. This flag is cleared when the transmit data register is next written.
13	RW	Transmit retry flag. When this flag is set, the hardware will continue to attempt to transmit the data until the transmission is successful. If this bit is cleared while the transmit buffer is full, then the transmit buffer full flag should be polled until it is clear indicating that the transmitter has stopped retrying. When it is clear the transmit failed flag should be tested to determine if the last transmit attempt succeeded or failed.
12	RW	Transmit bus lock flag. When a transmitter sets this bit, the Communication Bus will be locked to this transmitter until this bit is cleared. This allows one transmitter to have the maximum possible Communication Bus bandwidth available to it. (Only the MPEs have this bit.) This is potentially dangerous to performance, as all other Communication Bus traffic is locked out while this bit is set; and so this should be used with extreme care.
0–7	RW	Transmit target ID. This will be used for the next data to be written into the transmit data buffer.

IO Devices on the Communication Bus

The Communication Bus supports ‘slave’ IO devices as well as processors. These obey the normal rules of the Communication Bus, but will generally have their own particular definition of what the data packet represents. Simple IO devices may not connect to all 32 bits of the Communication Bus.

Writing to an IO device requires sending a packet that contains a register address and the write data. Generally, the register address and command type is sent in the first long word of the transfer, and the write data in the rest of the packet.

Reading from an IO device requires sending a packet that contains the register address, and then waiting for the IO device to respond with a packet containing the read data. The receiver will have to interpret this data in the context of the last command packet it sent. An IO device which receives a read command will leave its receive buffer full flag set until it has responded successfully.

OTHER BUS

The Other Bus is a DMA bus between the MPEs and external System Bus memory and ROM. It resembles the Main Bus, but is greatly simplified.

The mechanism to use it is the same as Main Bus DMA – a command is built in MPE memory and the address of that command placed in a pointer register. This causes the bus to be requested. At the end of a transfer and whenever the bus is idle the bus is arbitrated, and when granted to a master the command is read from that master by the central controller, which then executes the command.

All transfers on this bus are initiated by the central controller, and may be between any two addresses in the Other Bus address space. The restriction is that both source and destination may not lie in the same block of memory, for example they may not both be on the external System Bus, or not both in the same MPE.

Command Format

This is the format of the Other Bus DMA command. This is created in MPE memory on a vector (128 bit) boundary. Its is always three long words in length:

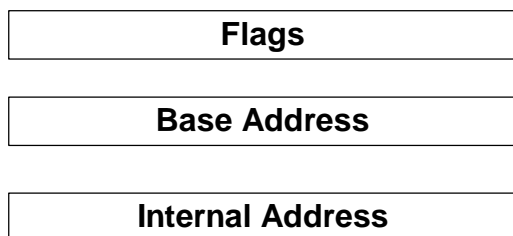


Figure 5 - Other Bus DMA Command Format

These are the long words that make up the DMA command. Any unused bits must be set to zero.

DMA Command – Flags

Bits	Name	Description
31-29	Unused	Set to zero.
28	REMOTE	When this bit is set, the internal address field is interpreted as a system address, and therefore can lie anywhere in memory that is accessible from the Other Bus. Refer to the table under Memory Map in the Introduction section. When this bit is clear, the internal address field is interpreted as an MPE internal address to the requesting MPE. This allows code to be written that will run on any MPE. (The internal address of all MPEs is the system address of MPE0.)
27-24	Unused	Set to zero.
23-16	LENGTH	This gives the length of the transfer in longs. Valid values are 1-255.
15-14	Unused	Set to zero.
13	READ	Read flag, set for transfer from base address to internal address, clear for transfer from internal address to base address.
12-0	Unused	Set to zero.

DMA Command – Base Address

Bits	Name	Description
31-0	BASE	Base address of the transfer. This is the “external” pointer for a linear transfer

		operation, and is always a system address. Normally, it must lie on a long boundary, but for transfers to ROM it may lie on any byte boundary.
--	--	--

DMA Command – Internal Address

Bits	Name	Description
31-30	Unused	These bits must be set to zero.
29-2	IA	Internal address. This is the internal pointer of the DMA transfer. Normally this will be in the data RAM of the MPE requesting the transfer, however it may actually be any valid address in internal memory if the REMOTE flag is set. It is always on a long word boundary.
1-0	Unused	These bits must be set to zero.

Restrictions on Other Bus DMA

1. Remote DMA may only be used on a remote MPE if that MPE is not using its Other Bus interface. If you break this rule the Other Bus will become unusable.
2. The source and destination of the transfer may not be the same device, i.e. an MPE cannot transfer data to itself, and the Other Bus cannot perform block copies in external memory.

Control Registers

The following two control registers are present in each MPE and control DMA.

odmactl Other Bus DMA control and status register

Read / Write

Bit	Name	Description
31-7	reserved	
6-5	odmaPriority	DMA bus priority, in the range 1-2. Default value is 1. 0 disables Other Bus DMA, and 3 is reserved for future use.
4	cmdPending	DMA command pending, this flag means that the DMA command pointer must not be written to. This bit is read only.
3-0	cmdActive	Other Bus DMA active level, 0 indicates no activity, 1 means that a single DMA transfer is active, 2 means that a data transfer is active and a command is pending, higher values will not occur in NUON. These bits are read only.

odmacptr Other Bus DMA command pointer

Read / Write

The address of a valid DMA command structure may be written to this register when the DMA pending flag is clear. Writing this register initiates the DMA process. The address written here must lie on a vector address boundary.

SYSTEM BUS

The System Bus is the expansion bus of NUON. It allows additional memory to be accessed, and can be used for communication with external processors. The MPEs access it by performing Other Bus DMA,

The System Bus operates in one of two principal modes:

Internal Mode

In internal mode the System Bus interface acts as a memory controller. “Internal” refers to the use of the on-chip memory controller. This controller can access three memory areas; one SDRAM area, one area switchable between SDRAM or a non-multiplexed bus memory type, and one fixed non-multiplexed bus memory area, these latter may be used for ROM, EPROM, flash memory, SRAM or some other external memory mapped device.

Each of these three areas may independently contain 8, 16 or 32-bit memory. The DRAM interface supports a variety of 16-bit wide SDRAM configurations, running at a 54 MHz clock speed.

External Mode

In external mode, an external device controls the system memory. NUON may request the bus, and then perform cycles to external memory. External memory is normally always 32 bits wide, but one special area may be defined as narrower.

The External System Bus provides a means by which the NUON system may be expanded. It supports both bus master and slave operations.

The Other Bus DMA channel may become a bus master on the System Bus, accessing memory and peripheral devices on the System Bus in parallel with the operation of the rest of the system. NUON will request the System Bus from an external arbiter when it is required, so the bus can be shared with an external bus master. The system interface supports byte, word and long word transfers.

A slave interface is also provided on the System Bus so that an external bus master may communicate with the NUON system.

The Memory Controller Mode is set with configuration resistors during reset.

External Memory Controller mode

In this mode, an external memory controller (such as the MPC860 SIU) is responsible for generating the control signals and strobes for System Bus memories and peripherals. The bus is synchronous to a bus clock (BCLK), and supports high speed burst data transfers. The system provides a clean interface to the Motorola MPC860, and can be modified with external logic to support Other Bus specifications.

The System Bus area memory map in this mode is:

Address	Size	Description
\$8000 0000 - \$AFFF FFFF	768M	System Bus

All this memory is treated in the same way

Internal Memory Controller mode

In this mode, NUON is responsible for generating all the strobes and control signal for System Bus peripherals and memories. The System Interface supports locally attached memories including SRAM, ROM, FLASH or DRAM.

In internal memory controller mode NUON will still accept slave transfers from an external bus master. However it will not be possible for an external bus master to access memory controlled by NUON.

The DRAM controller in NUON will support either SDRAM or EDO DRAM, with widths of 16-bits for SDRAM and 32 bits for EDO. However, EDO support is considered obsolete in Aries 3 and later devices.

The System Bus area memory map in this mode is:

Address	Size	Description
\$8000 0000 - \$8FFF FFFF	256M	System Bus DRAM
\$9000 0000 - \$90FF FFFF	16M	System Bus DRAM / ROM / SRAM 0
\$9100 0000 - \$9FFF FFFF	240M	Reserved
\$A000 0000 - \$A0FF FFFF	16M	System Bus ROM / SRAM 1
\$A100 0000 - \$AFFF FFFF	240M	Reserved

However, this memory map may be modified if using SDRAM, as follows:

dram0Enable	dram1Enable	contiguousSdram	DRAM0 Address Space	DRAM1 Address Space
0	0	X	Disabled	Disabled
1	0	X	\$8XXXXXXX	Disabled
0	1	0	Disabled	\$9XXXXXXX
0	1	1	Disabled	\$8XXXXXXX
1	1	0	\$8XXXXXXX	\$9XXXXXXX
1	1	1	\$8XXXXXXX	contiguous to DRAM0

Internal Mode Address Multiplexing

In internal mode, the EDO DRAM address lines should be hooked up as follows:

address pin	22	21	20	19	18	17	16	15	14	13	12	11
row	23	22	20	19	18	17	16	15	14	13	12	11
column	25	24	21	10	9	8	7	6	5	4	3	2

System Bus Control Registers

These registers allow the System Bus interface to be configured. They are available as part of the Miscellaneous IO controller, and you should refer to that section to see how to access them over the Communication Bus.

sysCtrl

System Bus Control

\$0030

Read / Write

This register controls the behavior of the NUON system on the System Bus.

Bit	Name	Description
31		Unused, set to zero.
30	xhostAddrLo	<p>This bit modifies to address lines used by an external master to access NUON. When set, the number of address lines is reduced. This is notmally used for the QFP package option.</p> <p>The default state is zero, and is compatible with earlier versions of Aries. The logic looks at sys_sa[23:21] for host offset, and sys_sa[19:2] to determine the register being accessed by the master. During a 16-bit master access, in addition to sys_sa[23:2], sys_sa[24] is used to determine which word of the long is being accessed (0 = 31:16, 1 = 15:0). During an 8-bit master cycle, sys_sa[24,20] are used to determine the byte being accessed (00 = 31:24, 01 = 23:16, 10 = 15:8, 11 = 7:0).</p> <p>In the QFP package, the address inputs [24:18] are forced to zero, so 16-bit master accesses will always access bits 31:15 of the NUON registers.</p> <p>If this bit is set, during a 32-bit master access, sys_sa[23:21] are used for host offset, and sys_sa[12:2] are used to determine the register being accessed. For 16-bit master cycles, sys_sa[14] determines the word being accessed, and in 8-bit master mode, sys_sa[14:13] determine the byte being accessed.</p> <p>These bits will allow a 16-bit master to work with either the QFP or BGA versions of the chip. The master will have to drive sys_sa[12:2] with the address of the NUON register, with sys_sa[14] = 0 to access bits[31:16] and sys_sa[14] = 1 to access bits[15:0]. Boot code will have to set the two new bits.</p>
29	xhostDataLo	<p>The default state for this is zero, implying that the master's data bus is aligned on sys_sd[31]. This means that during external master reads from and writes to NUON, data is used on:</p> <ul style="list-style-type: none">sys_sd[31:0] for 32-bit masterssys_sd[31:16] for 16-bit masterssys_sd[31:24] for 8-bit masters <p>If this bit is set the master's data bus is aligned on sys_sd[0]. This means that during external master reads from and writes to NUON, data is used on:</p> <ul style="list-style-type: none">sys_sd[31:0] for 32-bit masterssys_sd [15:0] for 16-bit masterssys_sd[7:0] for 8-bit masters <p><i>Aries 3 and up only.</i></p>
28	dramBank2	When this bit is set in internal mode, the chip select 0 space is used for DRAM instead of for a static address memory type.
27	xHost16	External host is a 16-bit device (default 32)
26	xHost8	External host is an 8-bit device (default 32)
25	cs1RdyEn	Require an external ready signal to terminate a CS1 cycle. This bit will enable the memory cycles done in this range to terminate on a falling edge on the SYS_RDY_B input pin.

24	cs0RdyEn	Require an external ready signal to terminate a CS0 cycle. This bit will enable the memory cycles done in this range to terminate on a falling edge on the SYS_RDY_B input pin.
23	dramX16	Internal mode System Bus DRAM is 16-bit
22	dramX8	Internal mode System Bus DRAM is 8-bit
21	saMuxEn	Enables the address multiplex function for external mode.
20	uaeTsDead	UAE to TS dead cycle. Control part of the cycle timing for external mode master cycles.
19		Unused, set to zero.
18	external	External mode. This controls the mode of the System Bus itself, and is discussed below in the System Bus section. The reset state of this is set. It should be set appropriately once after power up.
15-11	cs1Length	Chip select 1 length. This controls the timing of the System Bus internal mode chip select 1 memory area (\$A000 0000 - \$A0FF FFFF). The value of this register is the number of clock periods that a chip select will be active for. On reset the value of the length is set to all ones. Do not program a value of less than 3.
10-6	cs0Length	Chip select 0 length. This controls the timing of the System Bus internal mode chip select 0 memory area (\$9000 0000 - \$90FF FFFF). The value of this register is the number of clock periods that a chip select will be active for. On reset the value of the length is set to all ones. Do not program a value of less than 3.
5	hostInt	External host interrupt. Writing a one to this bit generates an external host interrupt. Writing a zero has no effect, and it is not necessary to clear this bit after writing a one. A zero is always read from this bit position.
4	busLock	System Bus lock. When this is set, the System Bus is requested and held until this bit is cleared. This allows atomic operations to be performed on the System Bus, but must be used with great care to avoid locking up the System Bus. You should set this bit, perform a test and set operation or whatever, and clear this bit all in the same cache line to avoid causing problems.
0-2	slaveOffset	Sets the NUON System Bus slave register address offset. NUON can appear in one of 8 locations at 2 Mbyte offsets relative to the base address decoded by CS. CS is assumed to decode a 16 Mbyte region, so address line 2 to 23 are decoded. These bits are defined at power-on by external configuration resistors.

sysMemctl

System Bus Memory Control

\$0031
Read / Write

This register controls the behavior of the memory on the System Bus.

Bit	Name	Description
14-4	refLength	Refresh length. Refreshes are performed at the clock rate divided by this value plus one.
1	slowRam	Slow DRAM flag. This bit is set to slow down some aspects of the DRAM timing.
0	edoRam	EDO DRAM flag. This bit should be set for EDO DRAM, and left clear for page mode DRAM.

sysSdramCtrl

System Bus SDRAM Control

\$0032

Read / Write

Following bits have meaning only if the System Bus is programmed in internal mode. If SDRAM is enabled either by setting **dram0Enable** or **dram1Enable**, the EDO interface will be automatically disabled.

Bits	Name	Description
31	contiguousSdram	This bit in conjunction with bit 24 determines the address space of logical bank 1 when logical bank 1 is enabled. See table below.
30	refEnable	If set to 1, sdram/edo refresh operations will be performed. On a reset, this bit is always set. If set to 0, refresh will never be performed. This bit will be set to 0 only during sdram initialization and must be set to 1 during normal operation.
29	dram1Banks	Specifies if the sdram in logical bank1 has two or four internal banks. 0 = 2 banks, 1 = 4 banks.
28-27	dram1Width	Specifies if the sdram in logical bank1 is composed of one X16, two X8, or four X4 parts. 00 = X4, 01 = X8, 10 = X16, 11 = Reserved.
26-25	dram1Tech	Specifies if the sdram in logical bank1 is composed of 16 Mbit, 64 Mbit, 128 Mbit, or 256 Mbit technology parts. 00 = 16 Mbit, 01 = 64 Mbit, 10 = 128 Mbit, 11 = 256 Mbit
24	dram1Enable	Specifies if logical bank 1 is populated or not. This bit in conjunction with bit 31 determines the address space of this bank. 1 = enable sdram, 0 = disable sdram
23	dram0Banks	Specifies if the sdram in logical bank0 has two or four internal banks. 0 = 2 banks, 1 = 4 banks.
22-21	dram0Width	Specifies if the sdram for logical bank0 is composed of one X16, two X8, or four X4 parts. 00 = X4, 01 = X8, 10 = X16, 11 = Reserved.
20-19	dram0Tech	Specifies if the sdram in logical bank0 is composed of 16 Mbit, 64 Mbit, 128 Mbit, or 256 Mbit technology parts. 00 = 16 Mbit, 01 = 64 Mbit, 10 = 128 Mbit, 11 = 256 Mbit
18	dram0Enable	Specifies if logical bank 0 is populated or not. This bank is always mapped to the 8XXXXXXXh System Bus address space. 1 = enable sdram, 0 = disable sdram
17	refreshCmd	If set, will cause the sdram controller to issue a refresh command to both the logical banks during sdram initialization. It will be automatically cleared by the sdram controller once the requested refresh has been performed. It must not be set during normal operation.
16	mrsCmd	If set, will cause the sdram controller to issue a mode register set command to both the logical banks during sdram initialization. This bit will be automatically cleared by the sdram controller once the requested mrs command has been issued. The cas latency and burst length bits in this register must be programmed before setting this bit. It must not be set during normal operation.
15	prechCmd	If set, will cause the sdram controller to issue a precharge all banks command to both the logical banks during sdram initialization. This bit will be automatically cleared by the sdram controller once the

		requested precharge command has been issued. It must not be set during normal operation.
14-13	twr	Sets the delay in clocks between the last data in and precharge command (also known as tDPL). Delay = Value + 2 clocks.
12-10	tras	Sets the delay in clocks between activate and precharge commands. Delay = Value + 2 clocks.
9-7	trc	Sets the delay in clocks between activate and refresh, and refresh and activate commands. Delay = Value + 2 clocks.
6-5	trp	Sets the delay in clocks between precharge and activate commands. Delay = Value + 2 clocks.
4-3	casLatency	These bits must be set prior to setting the MRS_CMD bit. 00 = Reserved, 01 = CAS Latency 1, 10 = CAS Latency 2, 11 = CAS Latency 3
2-1	burstLength	These bits must be set prior to setting the MRS_CMD bit. 00 = Reserved, 01 = Burst Length 2, 10 = Burst Length 4, 11 = Burst Length 8
0	tristateSdramBus	If set, the sdram address and control signals will be tri-stated if an external master performs a cycle. The data (???) bus will be tri-stated if the external master performs a read with the chip-select de-asserted, indicating that it is reading a systembus device and not the Aries registers, or if it performs a write cycle.

Note: The refresh rate is controlled by the refLength bits in the sysMemctl register.

Address space table controlled by bits 31, 24 and 18:

dram0 Enable	dram1 Enable	contiguous Sdram	DRAM0 Address Space	DRAM1 Address Space
0	0	X	Disabled	Disabled
1	0	X	8XXXXXXXXh	Disabled
0	1	0	Disabled	9XXXXXXXXh
0	1	1	Disabled	8XXXXXXXXh
1	1	0	8XXXXXXXXh	9XXXXXXXXh
1	1	1	8XXXXXXXXh	contiguous to DRAM0

Communication with an External Host Processor

The NUON System Bus allows the NUON system to act both as a peripheral device to an external host processor, and to transfer data to external memory on the System Bus.

In order that fast reliable communication can be achieved between the NUON sub-system and the external host, three mechanisms are provided. These are:

1. NUON processors can interrupt the external host, and the external host can interrupt NUON processors.
2. The external host has a Communication Bus port which allows it to quickly transfer 128-bit data packets with any Aries processor. This interface can be interrupt driven or polled.
3. The NUON Other Bus DMA channel can become a bus master using the System Bus interface, and therefore MPEs can read and write data in RAM shared with the external host processor.

External host access to the Communication Bus

The NUON host register interface provides access to the Communication Bus for an external host processor. This interface does not operate in quite the same way as the MPE Communication Bus interface. In particular, there is not access to the additional **comminfo** data, and the interrupt handling is different.

When a Comm Bus packet is sent to the System Bus interface, hostIntStat[1] gets set. This bit can only be cleared by the host processor writing zero to it. However, if you clear hostIntStat[1] before you read the data, it will be set again. So, when you receive an interrupt (at the host processor), do the following:

1. See if hostIntStat[1] = 1.
2. If so, read host_crd[127:0]. This will clear the internal interrupt signal.
3. Now set hostIntStat[1] = 0.

MMP Slave Interface

The NUON host interface can be viewed as a memory mapped peripheral device. When a bus master (such as a host processor) activates the CS signal, NUON will qualify the address(???) bus and RW signal, to allow the bus master to read a write a variety of internal registers.

The base address of these registers is programmable by power-on configuration resistors, and the offset given in this table is relative to that base address.

These registers are:

host_ctd Communication Bus 128-bit transmit data

\$0000 0000 to \$0000 000C
Write Only

This register is four 32-bit write only registers in consecutive locations that allow the external host to send communication packets. Writing the highest address initiates a transmit.

Although this shares the address range with host_crd below, these are physically separate registers, i.e. the transmit and receive functions are entirely independent.

host_crd Communication Bus 128-bit receive data

\$0000 0000 to \$0000 000C
Read Only

This register is four 32-bit read only registers in consecutive locations which allow the external host to receive Communication Bus packets. Reading the highest address clears the receive buffer.

Although this shares the address range with host_ctd above, these are physically separate registers, i.e. the transmit and receive functions are entirely independent.

host_cctl

Communication Bus status and control

\$0000 0010
Read / Write

This register controls the operation of the external host Communication Bus port, and allows its status to be determined.

Bit	Name	Description
31	sysRecFull	Receive buffer full (read only)
30	sysComDis	Receive disable (read / write)
29-24	reserved	
23-16	sysSourceID	Received source ID (read only)
15	sysTxFull	Transmit buffer full (read only)
14	sysTxFail	Transmit failed (read only)
13	sysTxRetry	Transmit retry flag (read / write)
12-8	reserved	
7-0	sysTargetID	Transmit target ID (read / write)

hostIntCtl

External Host Interrupt Control

\$0000 0014 for write
\$0000 0018 for read
Read / Write

This register allows the external host to control what interrupts it receives from the Communication Bus interface and other sources.

Bit	Name	Description
31-4	reserved	
3	debugInt	Debug interrupt enable
2	mpe2hostInt	NUON software to host interrupt enable
1	rxFullInt	Communication Bus receive buffer full interrupt enable
0	txEmptyInt	Communication Bus transmit buffer empty interrupt enable

hostIntStat

External Host Interrupt Status

\$0000 0018 for write
\$0000 0014 for read
Read /Write

This register allows the external host to determine the source of an interrupt from the NUON system.

The interrupt line to the external host assumes level sensitive interrupts. The bits below allow it to determine the source of an interrupt, and some of them may be cleared by writing a zero to the corresponding bit. Any bits that you do not want to clear should be written with a one, this will have no effect, and will prevent erroneous interrupt clears.

Bit	Name	Description
31-24	version	Hardware version number. Currently assigned codes are: \$03 Aries 3 \$02 Aries 2 (MMP-L3C) \$00 all previous versions (MMP-L3A/B a.k.a. Oz/Aries 1)
23-4	reserved	
3	debugInt	Debug interrupt. This must be cleared in the debug control unit.

2	mpe2hostInt	NUON software to host interrupt. Cleared by writing a zero to this bit.
1	rxFullInt	Communication Bus receive buffer full. Cleared by writing a zero to this bit.
0	txEmptyInt	Communication Bus transmit buffer empty. Cleared by writing a zero to this bit.

hostIntReq

External Host Interrupt & Reset Request

\$0000 001C
Write Only

This register allows the external host to request that NUON processors be interrupted. This allows communication to be established without using the Communication Bus, as this can be masked by a full receive buffer. The interrupt will go to any MPE which has this interrupt enabled.

This register also supports a reset of the NUON system. This has the same effect as a power on reset, and so should only be used in extreme circumstances.

Bit	Name	Description
31-2	reserved	
1	hostReset	Reset the NUON system (active high).
0	host2mpeInt	Writing a 1 to this bit interrupts the NUON processors. Writing a 0 has no effect. It is not necessary to clear this bit.

hostMemPrct

External Host Memory Protection

\$0000 0020
Read / Write

This register controls a memory protection for the external host. A region may be defined within the host address space, by means of upper and lower address bounds, that is allowable for NUON accesses. The region is programmable on 64 Kbytes boundaries. All transfers outside this space will fail in external mode, and can generate a NUON debug exception.

The address used for this comparison is the transfer address **after** any modification given by the upperOffset bits, which are internally programmed in the sysMemctl register.

Bit	Name	Description
31-16	upperBound	Upper address bits. This is at most the top 16-bits of the highest allowable address for NUON. This defaults to all ones implying no upper bound.
15-0	lowerBound	Lower address bound. This is the top 16-bits if the lowest allowable address for NUON. This defaults to all zeroes implying no lower bound..

host16bit0

External Host 16-bit area 0

\$0000 0024
Read / Write

This register controls a memory area that is only 16-bits wide, as opposed to the general 32-bit width of memory. The region is programmable to be any power of two bytes in size, on a corresponding power of two boundary, down to a 64 Kbyte boundary.

The address used for this comparison is the transfer address **after** any modification given by the upperOffset bits, which are internally programmed in the sysMemctl register.

Bit	Name	Description
31-16	wordAddress	Upper address bits. This is as many as the top 16-bits of the 16-bit area, as

		selected by the mask in the low bits.
15-0	wordMask	Upper address mask. A set bit here means that the corresponding bit in the high part of the register is used for comparison.

As an example, to program a 16-Mbyte area starting at \$83000000 you would set the low bits to \$FF00 so that only the top eight bits are used for the comparison, and set the high bits to \$8300.

This register defaults to all ones implying that the top 64-Kbytes of memory are 16-bit. If you plane to use this area for 32-bit memory you will have to change the setting.

host16bit1 External Host 16-bit area 1

\$0000 0028
Read / Write

This register is exactly the same as the register above, and allows a second area to be programmed.

hostBanks External Mode DRAM muxing control

\$0000 002C
Read / Write

This register determines where the boundary between bank 0 and bank1 lies for the SAMUX function in external mode only. If internal mode is selected, or the SAMUX function is not in use, it has no effect. The boundary is controlled in the same manner as the 16-bit area registers above, with the exception that only the top 8 bits of the address are used, limiting the resolution to a 16-Mbyte boundary.

This register contains address and mask bits that determine a sub-range of System Bus addresses that correspond to Bank0. Any address not in this range is assumed to reside in Bank1 or other sub-range.

The address used for this comparison is the transfer address **after** any modification given by the upperOffset bits which are internally programmed in the sysMemctl register.

Bit	Name	Description
20-19	bank1mux	Selects the multiplexing type for SAMUX in bank 1 of DRAM in external mode. See below.
17-16	bank0mux	Selects the multiplexing type for SAMUX in bank 0 of DRAM in external mode. See below.
15-8	bank1addr	Bank 1 start address. This is as many as the top 8-bits of the address of the bank 1 area, as selected by the mask in the low bits.
7-0	bank1mask	Upper address mask. A set bit here means that the corresponding bit in the start address field is used for determining the start of the bank 1 area. This is superfluous and should be set to all ones.

hostSysOffset External Mode Address Offset

\$0000 0030
Read / Write

This register determines the physical address of NUON transfers in the external CPU memory space. The MPEs see the System Bus space as starting at the logical address \$80000000, in

external mode the MSB of the logical address is ignored, and this value is then added to the top sixteen bits. i.e.

$$\text{physical address} = \text{logical address} - \$80000000 + (\text{sysOffset} \ll 16).$$

Bit	Name	Description
15-0	sysOffset	External mode address offset. This defaults to \$8000 implying no address modification.

dmaBreak

External Mode DMA Interruption

\$0000 0034
Read / Write

This register contains two fields that can be used to break a System Bus DMA transfer into smaller groups. This can allow an external device to master the bus without having to wait for the entire DMA transfer to complete.

Bit	Name	Description
31-16	breakCount	This field contains the number of clock periods that the System Bus interface will wait before it attempts to re-arbitrate for the bus after having been interrupted.
15-0	cmdCount	This is the number of individual transfers, within an entire DMA, that the interface will execute before releasing the bus.

External Mode Bank Select Bits

In External Mode, the System Bus address range can have several subdivisions. Control bits in several registers determine the type of memory expected to reside in each section. The System Bus has 2 main banks of memory. Once the location of each has been programmed, select bits for each bank indicate the type of memory placed to be accessed. When operations in these ranges occur it is expected that the external processor will produce the necessary memory strobes. The external processor can handle a wide variety of memory sizes, but it is done in a very particular manner. The System Bus implements 6 possible memory configurations. These are the likely choices for memory that NUON could encounter in a design. The selected ones are:

Bank Select Bits	Memory Size (kB)
00	1 or 2 M
01	4 or 8 M
10	16 or 32 M
11	Invalid

External Mode Address Multiplexing

The System Bus control logic can be programmed to present column and row addresses on the address pins of the System Bus during memory accesses. The address bits that make up the row and column address for the respective memory sizes follow the particular scheme that the external processor decided to use. The multiplexed address bits will appear on the lower bits of the address pins, bits 13-2, and will conform to the following scheme:

1M/2M

address pin	13	12	11	10	9	8	7	6	5	4	3	2
row	x	x	20	19	18	17	16	15	14	13	12	11
column	x	x	x	10	9	8	7	6	5	4	3	2

4M/8M

address pin	13	12	11	10	9	8	7	6	5	4	3	2
row	x	22	21	20	19	18	17	16	15	14	13	12
column	x	x	11	10	9	8	7	6	5	4	3	2

16M/32M

address pin	13	12	11	10	9	8	7	6	5	4	3	2
row	24	23	22	21	20	19	18	17	16	15	14	13
column	x	12	11	10	9	8	7	6	5	4	3	2

External Mode MMP Bus Master Interface

NUON can also arbitrate for the host bus, and become a full bus master.

As a bus master, NUON can read and write any memory on the host bus (using either the built-in memory controller in internal mode, or an external memory controller such as the MPC860 SIU in external mode).

By sharing host bus memory with a host processor, high-bandwidth data transfers can be made to and from a previously defined data transfer area, or ‘letter-box’ RAM. Typically a semaphore can be used to lock and unlock this shared memory, or transmit and receive command FIFOs can be implemented as required.

Interrupts

The MPEs can be interrupted through the MMP Slave interface described above. An interrupt can be used as a signal to them that a data block is available in shared RAM.

In addition, NUON can interrupt a host processor via one of the GPIO signals.

Through a combination of these interrupts, shared memory and semaphores, a reliable, high-bandwidth communications interface can be implemented between NUON and a host processor.

ROM BUS

NUON supports up to 16 Mbytes of ROM, EPROM, Flash or SRAM on a simple 8-bit interface. This memory is intended for the Bootstrap, API / library, and built-in application software.

This memory is available as a memory-mapped device on the Other Bus, and so is accessible to the MPEs.

An alternative means of access to this memory is provided over the Communication Bus. Any Communication Bus master can send the ROM Bus interface a packet containing up to four addresses, and the ROM Bus interface will respond with the data from those addresses, which can be byte, word or long, as specified in the request packet.

ROM Communication Bus Interface

This mechanism is now considered obsolete and should not be used.

Any Communication Bus master can send the ROM Bus interface a request packet. The ROM Bus interface will fetch the requested data, and send a response packet containing the requested data. Only read transfers are possible using this mechanism. While the ROM Bus interface is servicing a request packet it will neither accept any more request packets nor allow any Other Bus cycles to be performed to ROM.

The data in long word 0 is read first, and the top two bits of this long word indicate the size of the transfer requested for all four addresses. The addresses are read, and fetched from in sequence after that, and an address of zero is used to terminate the sequence. This causes the fetch mechanism to cease reading and return the data read so far. If the size is line, then address fields 1-3 are ignored. All addresses, for any size data, can be arbitrarily byte aligned.

The request and response packet structures are described below. These packets are transmitted in the normal manner over the Communication Bus. The Communication Bus identification numbers are defined on page 161. The communication protocol is as follows for the request packet:

Long word	Description
0	31-30 transfer size, 0 = long, 1 = byte, 2 = word, 3 = line 23-0 request address 0
1	23-0 request address 1, if zero then no further data is read
2	23-0 request address 2, if zero then no further data is read
3	23-0 request address 3, if zero then no data is read

A response packet is returned. Its format is:

Long word	Description
0	Read data for address 0. 31-0 long read data 31-16 word read data 31-24 byte read data
1	Read data for address 1 in the same format as that for address 0.
2	Read data for address 2 in the same format as that for address 0.
3	Read data for address 3 in the same format as that for address 0.

Read data is not defined if the corresponding request address, or a preceding one, was zero.

VIDEO OUTPUT & DISPLAY TIME-BASE

The video display generator creates the video display output stream from a set of DRAM images, scaling and filtering the image if appropriate. It generates its own video time-base.

The video display generator always physically outputs lines of 720 pixels. It always drives an interlaced display, and supports both 60 Hz and 50 Hz video refresh rates. Available output resolutions are as follows:

Field rate:	60 Hz	50 Hz
Interlaced TV display:	720 x 480	720 x 576

This output resolution does not have to match the internal display buffer resolution, as the display generator is capable of both horizontal and vertical scaling on the memory image to match these output resolutions. Also, graphic buffers in memory do not have to fill this pixel area; they may correspond to sub-rectangles of it, with a specified border color filling the remainder of the screen. Multiple sub-rectangles can be displayed simultaneously, subject to certain restrictions.

The display generator can also fetch data from up to three separate display buffers and overlay one over the other with controllable transparency. These buffers do not have to be in the same pixel mode, and they can be independently scaled. Their capabilities are:

Channel	Capability
Main Video Channel	16-bit pixels, 32-bit pixels or MPEG data
Overlay Video Channel	4-bit pixels, 8-bit pixels, 16-bit pixels or 32-bit pixels
Sub-picture Video Channel	Sub-picture data

These three channels are combined with this priority:

1. Overlay Video Channel data
2. Sub-picture Channel data
3. Main Video Channel data
4. Border color

Each Overlay and Sub-Picture pixel has an alpha value, which spans from transparent to opaque. Main Channel and Border are mutually exclusive.

Video Data Flow, Filtering, and Scaling

Vertical Filtering

Vertical filtering is performed by fetching data from two, three or four lines of pixels and combining them with a F.I.R. filter to give the output value. No line buffering is performed; therefore vertical filtering adds significant additional bandwidth requirements to the video output channel. For example, a 720 x 480 16-bit video display normally requires about 20 Mbytes/sec of Main Bus bandwidth, while adding a four-tap anti-flicker filter to that will increase the requirement to around 80 Mbytes/sec. This should be weighed against performing filtering in software, or not filtering at all.

The two tap vertical filter performs linear interpolation, using the fractional parts of the position field. Three and four tap filters are implemented using either a set of coefficients stored in a register, or in the

case of the four-tap filter, a set of pre-programmed coefficients selected by the fractional parts of the position. Further details of this filtering are described in the Main Bus section.

Main Video Channel Horizontal Scaling

Horizontally, pixels may be arbitrarily scaled up or down, with a 4-tap filter. This scaling includes the following required scaling abilities for MPEG-2:

Scaling ratio	Image width in DRAM
1:1	720 pixels
4:3	540 pixels
3:2	480 pixels
2:1	360 pixels

The pixel rate scaling is performed with a sophisticated horizontal re-sizing filter. This is a 4-tap F.I.R. filter with two different sets of sixteen-coefficients.

The scaling rate is actually defined by a 3.11 bit value, which is effectively the value added to the X pointer of the display fetch after each pixel. This gives sufficient precision for the scaling ratios outlined above, as well as a much more general scaling ability.

Scaling can be performed on all Main Video Channel pixel types.

This horizontal scaling mechanism cannot scale down by more than a factor of three due to FIFO bandwidth limitations. However, a four-to-one scale down of MPEG pixels in the main channel is possible using decimation.

Vertical Scaling

Vertical scaling is handled differently to horizontal scaling, as it is performed within the main DMA mechanism. The filter applied may be interpolation between two successive display lines, or a F.I.R. applied to two, three or four display lines.

Vertical scaling is controlled by a vertical scale counter and a vertical scale increment. These are both 1.11 bit values. After each display line is fetched the vertical increment is added to the vertical counter, and if the integer field advances then the display pointer is advanced one line. The fractional bits control vertical interpolation and filtering.

Vertical scaling can be applied to both the Main Video Channel and the Overlay Video Channel.

Overlay Video Channel

The Overlay Video Channel supports 16-bit or 32-bit pixels in the same manner as the main video channel, and 32-bit pixels may contain transparency or alpha-channel data, so that pixels are blended with the Main Video Channel image. 16-bit pixels have one transparent color, and an overall alpha level may be set for the remaining colors.

The Overlay Video Channel can also support 4-bit or 8-bit logical pixels, which are turned into 32-bit pixels by indexing them into a Color Look-up Table (CLUT). These may also contain alpha-channel data.

Overlay data cannot be scaled other than by 1:1 or 1:2 and is a buffer of pixels covering a defined rectangle of the screen. Multiple overlay rectangles can be achieved by re-programming these registers

as the beam advances down the display, with the restriction that there can be no vertical overlap between them.

Sub-Picture Video Channel

The Sub-Picture channel is intended to support the display of a Sub-Picture stream as described in the “DVD Specifications for Read-Only Disc Version 1.0”. Two data sets are prepared by decoding software, and are read by the display hardware from main memory. These are the sub-picture data channel and the sub-picture control channel. The sub-picture mechanism is described in more detail in the Sub-Picture section of the full OEM version of this document.

Display Data Path

The flow of video data for main memory to the video output channel is shown below. The pointers and Main Bus DMA block shown here are all within the Main Bus DMA logic, and are described in that section. The VDG logic, described here, is the functionality from the Video FIFO onwards in the display path.

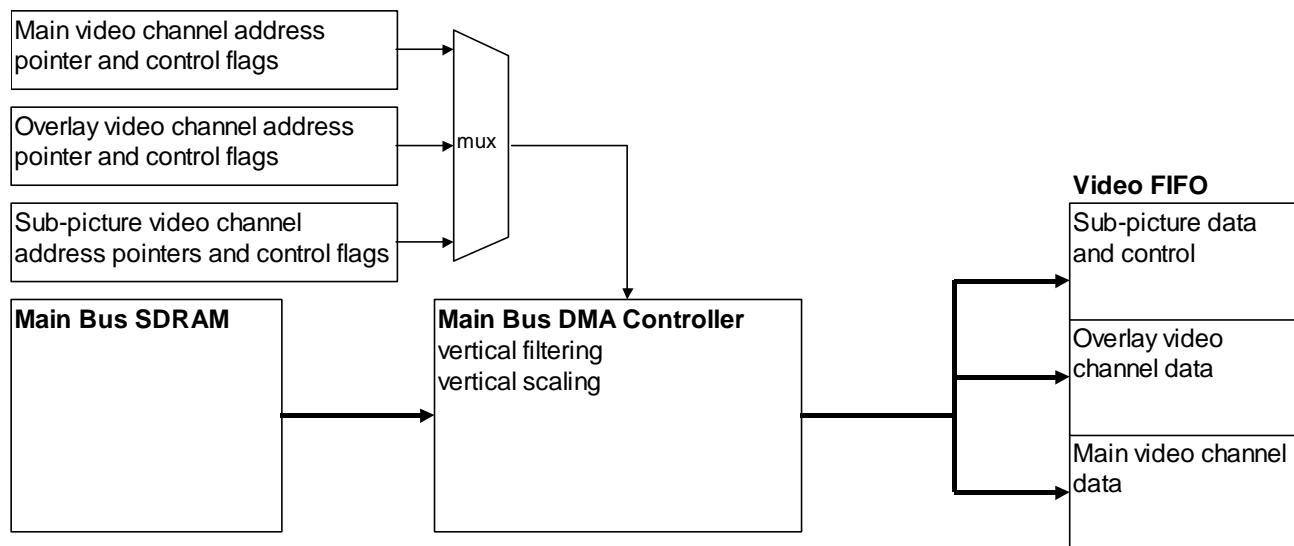


Figure 6 - Data Flow From Main Memory To The Video FIFO

Fetch of video data is controlled from within the Main Bus DMA Controller, and the pointers and flags shown above are actually part of the DMA unit. Vertical filtering and scaling is performed within the DMA controller.

This diagram shows how the pixel data is processed after being fetched from main memory, but before it is output:

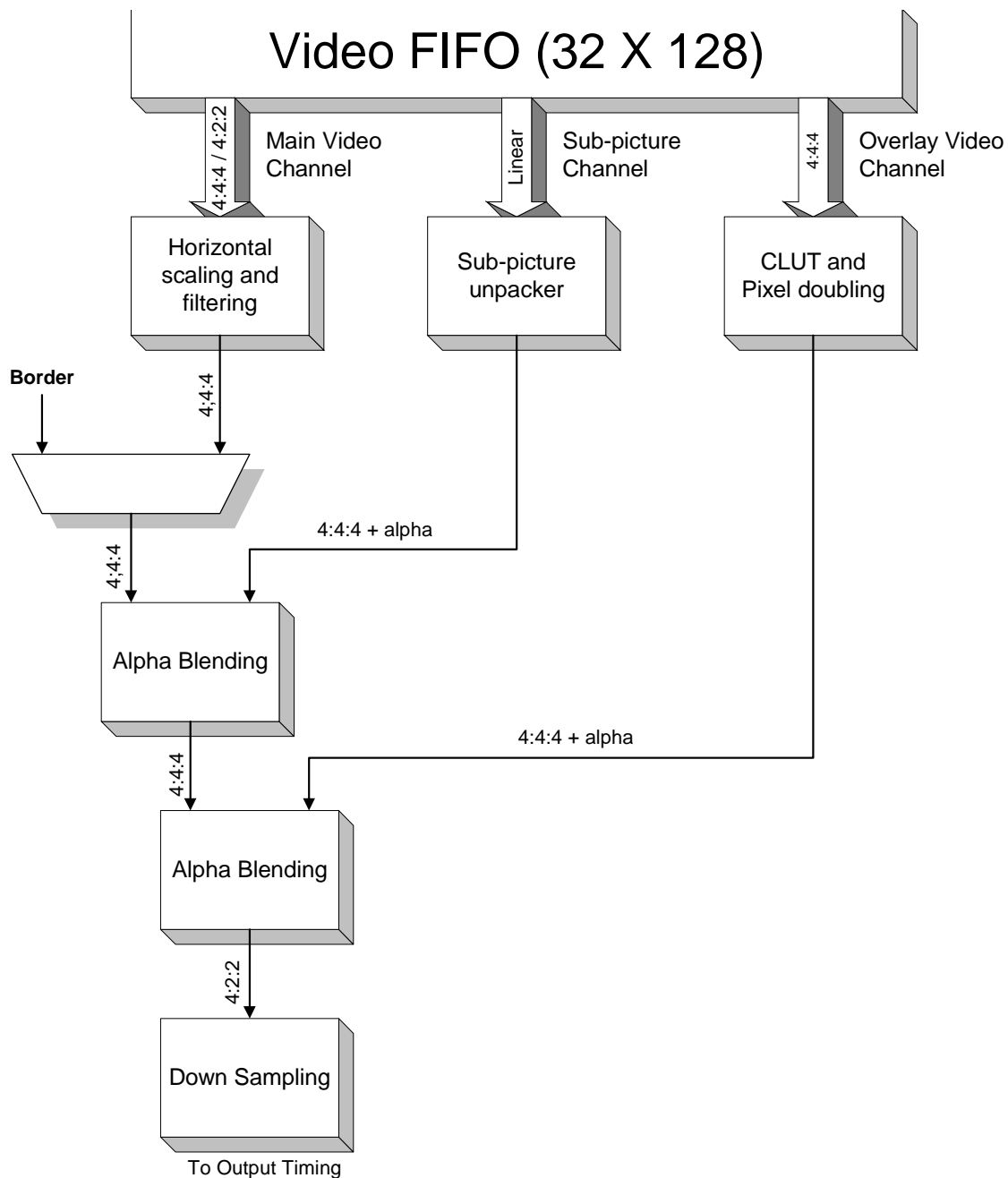


Figure 7 - data flow from the video FIFO to the video output

The three data channels are fetched from Main Bus memory, and any vertical scaling is performed on the main video channel data prior to the video FIFO.

The main video channel data is in one of the packed pixel modes except 4 bits or 8 bits per pixel; or is in the MPEG display data format. This data may be horizontally scaled and filtered.

Overlay channel data is either logical pixel data (4 bits or 8 bits used to index a CLUT) or physical pixel data in any of the packed pixel modes, i.e. a 4:4:4 mode. The pixels may be doubled in width, and may

include alpha channel data, either from the CLUT or from the 8 control bits of 32 bit pixels. Outside of the active overlay pixel area, totally transparent pixels are generated.

Alpha values of zero are opaque, so that if the overlay channel has its alpha at \$00 then it is opaque, if it is \$FF it is almost transparent. Color \$00, \$00, \$00 is always treated as transparent.

The sub-picture decoder is a special purpose unit used for DVD applications. It outputs 4:4:4 data with an alpha value. It shares a RAM with the overlay channel to hold its CLUT.

Address Generation

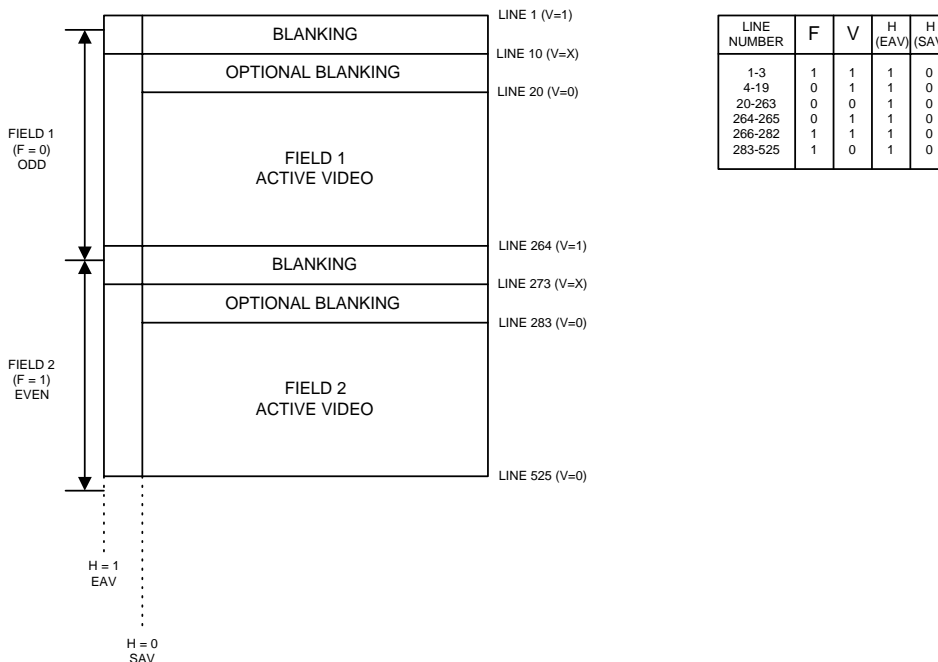
The display address generator will always fetch lines of pixel running from left to right upwards through memory. The choice of data formats available is described below. The base address of the display map is programmable, as is the amount to add to the base address for each successive line of the display. Display line data does not therefore have to be contiguous.

Video Time-Base

Two counters control the video time-base generator. One counts in clock cycles to give the video line timing, and the other counts in lines to give the field and frame timing.

A series of control values determine where in the count value various display functions are enabled, such as blanking, border, active video and the overlay data. The default values in these registers are the NTSC settings. Once a new value is written into these registers, the value will remain in effect until the next system reset (system power up or hard/soft reset).

The following picture will help in understanding the default settings:



The conversion to 8 bit integers uses the following formulae, where the computed value is rounded to the nearest integer.

$$\begin{aligned}Y &= 219 E_Y + 16 \\CR &= 224 E_{CR} + 128 \\CB &= 224 E_{CB} + 128\end{aligned}$$

Note that this implies the following about YCrCb data:

- Luminance occupies only 220 levels, with black being level 16.
- The color difference signals occupy 225 levels, with zero being level 128.
- Certain valid combinations of YCrCb coefficients do not correspond to a physical (RGB) color, and are not allowed (see below).
- MPE software performing color space conversion from RGB will have to perform the offset of 16 on the Y value, but not the 128 offset for Cr and Cb which is performed by store pixel, if the corresponding **chnorm** bit is set in the MPE **xyctl** or **uvctl** registers.
- Cr and Cb are (R-Y) and (B-Y) scaled to have a range of $\pm\frac{1}{2}$.

The hardware will strip out the illegal values 0 and 255 from the pixel stream, if they arise, and replace them with 1 and 254, respectively. Other illegal values are passed through to the digital video encoder, which may give unpredictable and non-linear results. You should make sure that your software does not generate out-of-range values.

Note that this encoding corresponds to the MPEG-2 video bit-stream sequence extension matrix coefficients value 6. If a different encoding is present in the MPEG-2 stream, then appropriate action (or inaction) will have to be taken.

Illegal Color Values

Some combinations of Y, Cr and Cb do not correspond to valid NTSC and PAL values, and must be avoided. This will not be a problem for artwork that is generated in RGB and converted, but can be a problem for algorithmically generated colors, and also potentially for some lighting models.

You can calculate the range of strictly legal values by transforming the RGB cube into NTSC space using the formulae above, thus:

R	G	B	Y	Cr	Cb
0	0	0	16	128	128
0	0	1	41	110	240
0	1	0	145	34	54
0	1	1	170	16	166
1	0	0	81	240	90
1	0	1	106	222	202
1	1	0	210	146	16
1	1	1	235	128	128

In theory only values within this transformed cube are allowed. However, we can stand a large range outside this, but we do have a particular problem with large Cr and Cb deviations from 128 when Y is approaching its minimum of 16, as these can appear to be sync signals to some TV sets.

VM Labs will screen NUON applications for this issue, and if it is found we will require it to be corrected for NUON applications.