VM LABS

NUON™ Multi-Media Architecture

Aries 3 Specifications

Full OEM Version

Revision 26

September 26th, 2001

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

This full version of the documentation is for VM Labs internal use and hardware OEM use only. It describes many hardware details which are normally concealed by the BIOS and HAL software, as these may change in future versions of the NUON architecture.

The information contained in this document is confidential and proprietary to VM Labs, Inc. and is provided pursuant to a Non-Disclosure agreement between VM Labs, Inc. and the recipient.

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.

Revision History – Last Updated: October 19, 2001;

V7 RM hand-over to JM. Introduced short instruction forms. Revised memory map.

V8 Update interrupts, exceptions, busses. Video generator. Revised MUL.

V9 L3B specific. Audio output. Controller bus.

V10 DMA updated, Cache removed. Arithmetic shifter simplified.

V11 Audio in, video in. More bus info.

V12 MPE DMA modified, instructions updated.

V14 Extra MPE instruction pipeline stage. Interrupts, DMA, MPE regs. modified. JSR added.

V15 Alpha silicon notes. Fuller coverage of hardware. Farewell, BUTTM. First Beta stuff.

V16 Last release of alpha documentation.

V17 Beta documentation interim release, not complete yet.

V18 More info. in Appendix A. Still not complete for beta.

V19 Beta MPE instruction set and registers. More MPEG details.

V20 Corrections. Last release of Oz (beta) documentation.

V21 Aries changes, preliminary release.

V22 First full Aries version, preliminary release.

V23 First Aries 2 release. Now called NUON, not Merlin.

V24 Beta Aries 3 release. No Aries 3 MPE or PLL details yet.

V25 First full Aries 3 release.

CONTENTS

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 ± 5% Volts	2.5 ± 5% Volts	1.8 ± 5% Volts
VDD I/O	3.3 ± 5% Volts	3.3 ± 5% Volts (5 VT)	3.3 ± 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:

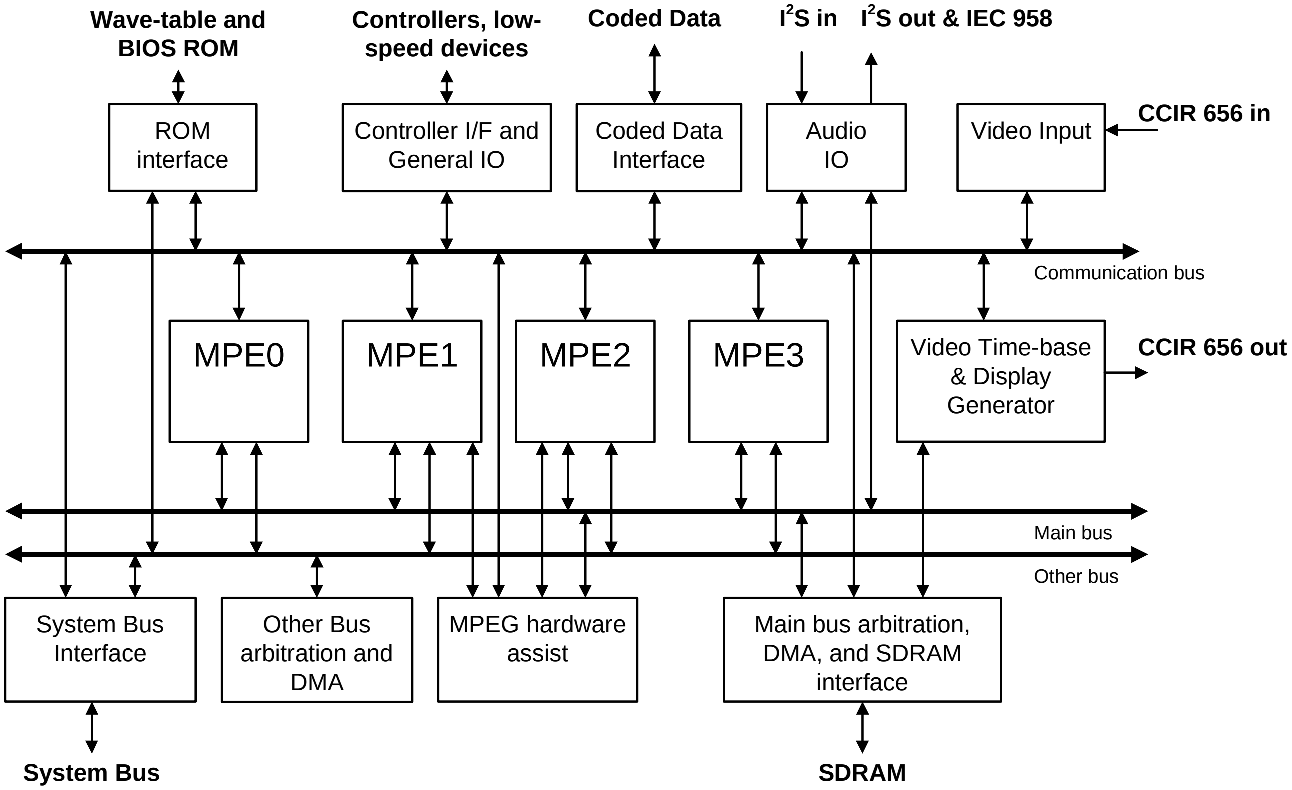


Figure 1. NUON Internal Architecture

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

Byte	8 bit value
Word	16 bit value
Scalar / Long	32 bit value
Half-vector	64 bit value (8 bytes)
Vector	128 bit value (16 bytes)

Notation

+	Logical OR
.	Logical AND
/	Logical NOT
$xx	Hexadecimal value xx
%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.

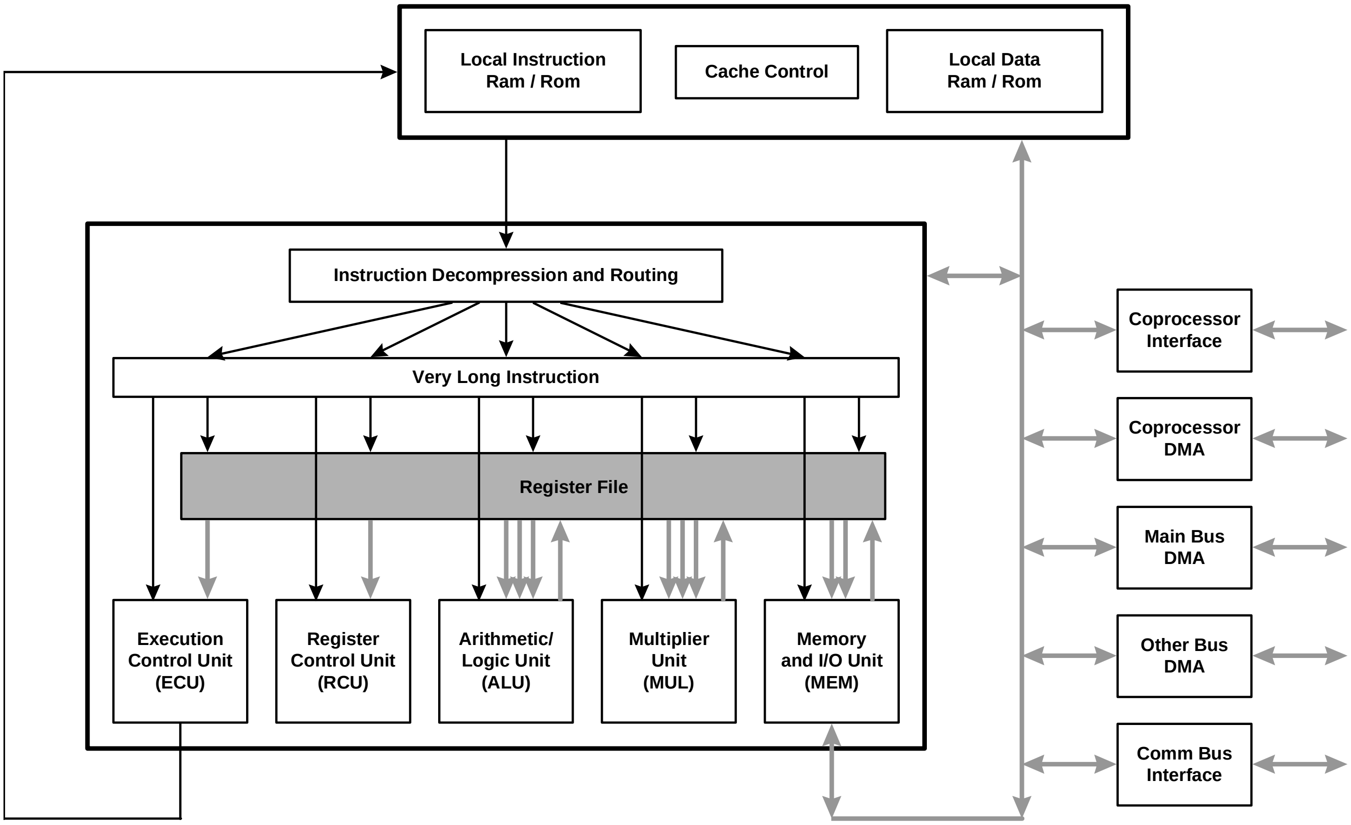

Local Instruction Ram / Rom
Cache Control
Local Data Ram / Rom
Instruction Decompression and Routing
Very Long Instruction
Register File
Execution Control Unit (ECU)
Register Control Unit (RCU)
Arithmetic/ Logic Unit (ALU)
Multiplier Unit (MUL)
Memory and I/O Unit (MEM)
Coprocessor Interface
Coprocessor DMA
Main Bus DMA
Other Bus DMA
Comm Bus Interface

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 then 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 - $200FFFFF	1 MByte	Data ROM
dtram	$20100000 - $201FFFFF	1 MByte	Data RAM
irom	$20200000 - $202FFFFF	1 MByte	Instruction ROM
iram	$20300000 - $203FFFFF	1 MByte	Instruction RAM
dtags	$20400000 - $2047FFFF	0.5 Mbyte	Data tag RAM
itags	$20480000 - $204FFFFF	0.5 Mbyte	Instruction tag RAM
ctlreg	$20500000 - $205FFFFF	1 MByte	Control register space
	$20600000 - $207FFFFF	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.

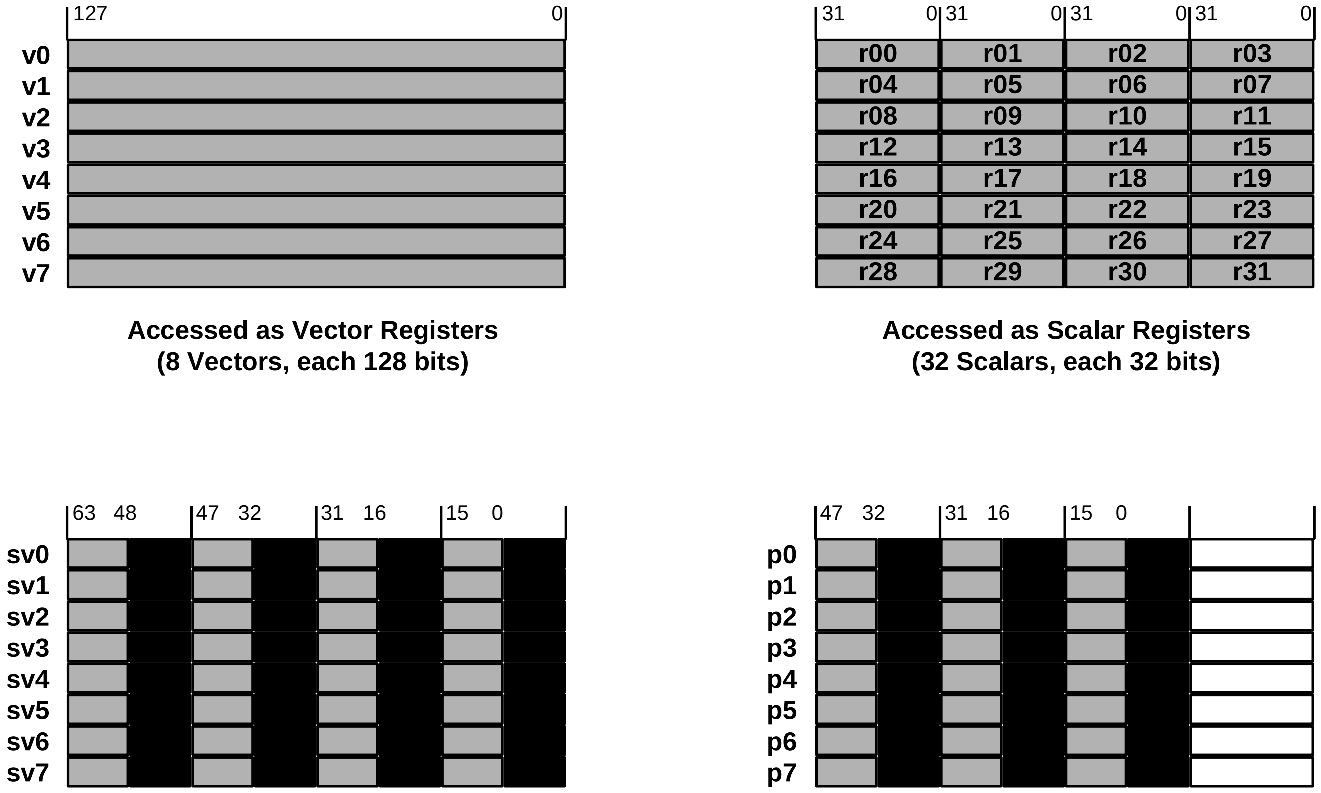
As tags, the entry format is:

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 (instuction 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)

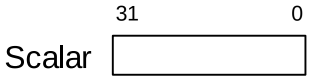
Accessed as Pixel Registers
(8 Pixels, each 3x16 bits)

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.

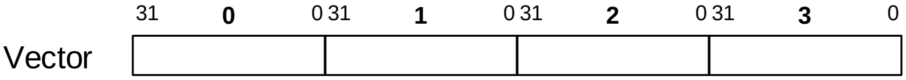
For geometry, we typically use signed 32-bit numbers. MPEs have 32-bit scalar registers, and vector registers which hold four 32-bit scalars. For rendering, we require three color elements (such as Red, Green, Blue, or Y, U and V). Intermediate results often require several fraction bits, and need to 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:

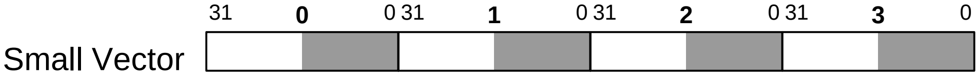
- Scalars, which are a 32-bit signed number, with arbitrary binary point position. Any 32-bit register can hold a Scalar.



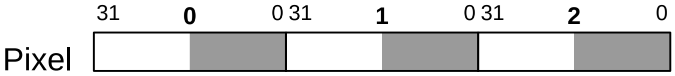
- Vectors, which are a group of four scalars in four consecutive registers, where the first register number must be a multiple of four.



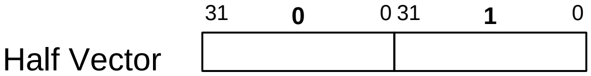
- Small vectors, which are like vectors, except that operations with them are only on the 16 most significant bits of each register. When a small vector is written, the 16 LSBs are usually set to zero.



- 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.

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 >> symbol. Rotations are denoted by the <> 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.

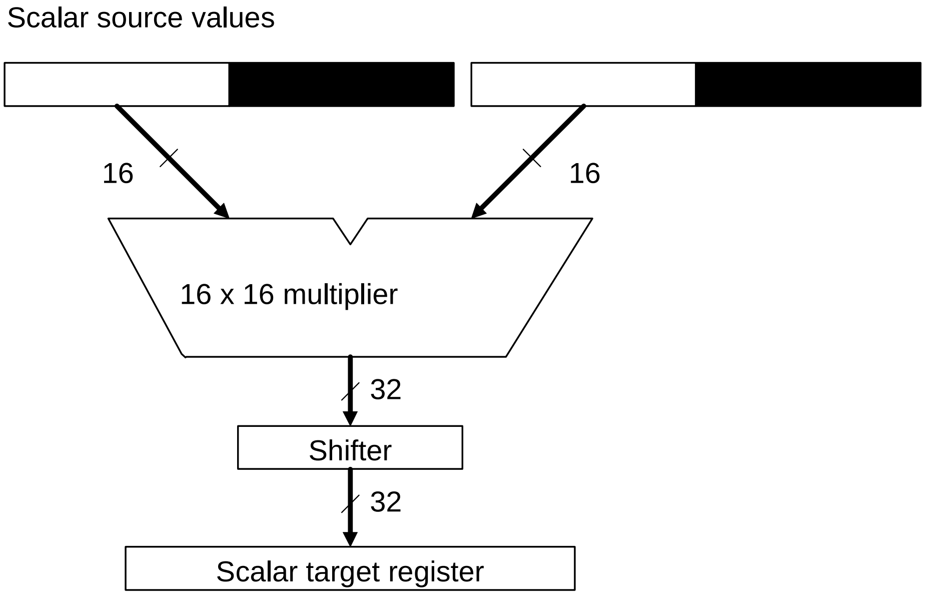
The **addm** and **subm** MUL unit arithmetic operations complete in one clock cycle, and may therefore not be used in the clock cycle after a **mul**, **mul_sv** , **mul_p** or **dotp**.

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

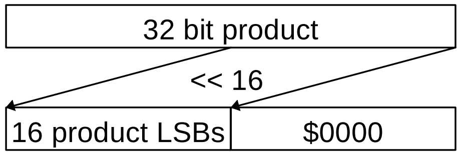
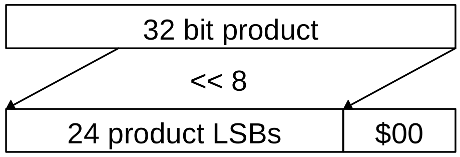
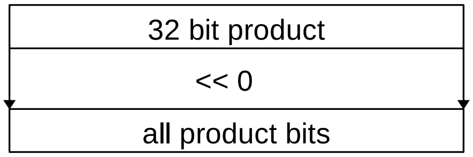
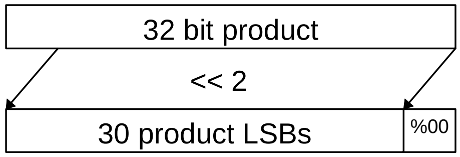
Small Vector Shifts

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:



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
`modge`	RCU range high	Set if a modulo or range instruction found the value greater than or equal to

Flag	Name	Description
		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:

```
ld_s        (Si),Sj           Indirect. Load from address Si, to register Sj
ld_s        (#nn),Sj          Absolute. Load from address #nn in local data RAM or
                              local data ROM
ld_s        (xy / uv),Sj      Bilinear indirect. Form an address from the xy or uv
                              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)	ld_s (#base,#offset),Sk	offset[8:4]	$2050_0000:$2050_01F0 (Vector-aligned)
*(16)	st_s Sj,(#base,#offset)	offset[8:4]	$2050_0000:$2050_01F0 (Vector-aligned)
@(32)	ld_s (#base,#offset),Sk	offset[12:2]	$2050_0000:$2050_1FFC (Scalar-aligned)
@(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.

```
        ┌pixel width┐   ┌ ((Y>>XY_MIPMAP)&(YTILEMASK>>XY_MIPMAP))x(width>>XY_MIPMAP)┐
XY_BASE+│    of     │ x │                                                           │
        └  XY_TYPE  ┘   └+((X>>XY_MIPMAP)&(XTILEMASK>>XY_MIPMAP))                   ┘
```

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`	1	If this bit is set, the integer part of **rx** / **ru** is mirrored prior to address generation. See

Register Label	Width in bits	Description
u_rev		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. 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 ru	32	32-bit X index register. The X part of the address is formed from the integer part of **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.

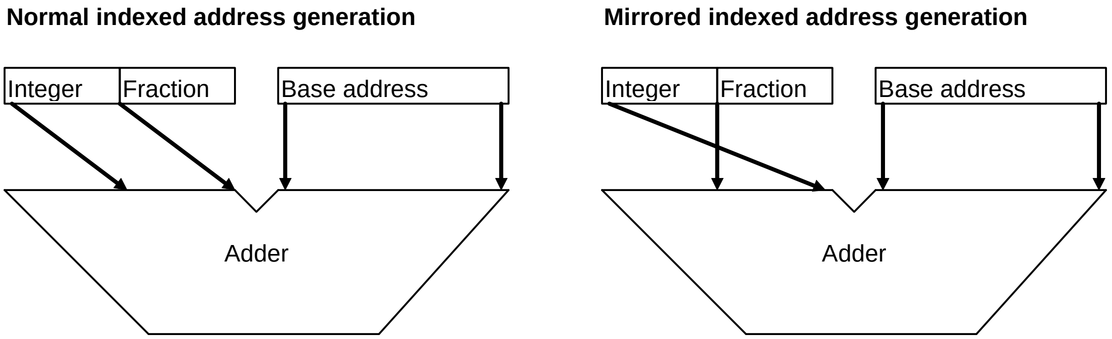
Range Limiting Index Registers

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` `vrange`	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
Integer
Fraction
Base address
Adder
Mirrored indexed address generation
Integer
Fraction
Base address
Adder

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		Store Data Size	Store	Load	Load Data Size
#	Name	To Memory	Form	Form	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 st_pz	ld_p ld_pz	¾ vector 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,16’0, 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                            allow the load to complete
ld_p    (R4),v1                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                             allow the load to complete
ld_p    (r4),v1                 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. ***See note below***. *Aries 3 and up only*.
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. ***See note below***. *Aries 3 and up only*
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 break point enable. Writing a zero has no effect.
5	**daRdBrkEn_set**	**daRdBrkEn**	Writing a one enables the data address read

Bit	“Write” value	“Read” value	Description
			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
2	**excepSrc_breakpointnow**	breakpoint instruction
1	**excepSrc_singleStep**	single-step break

Bit	Name	Description
0	**excepSrc_halt**	“halt” instruction

excepclr 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_iport_address_error**	iport-address error
7	**excepHaltEn_dbus_address_error**	dbus-address error
6	**excepHaltEn_bilin_addr_error**	bilinear address error
5	**excepHaltEn_rfmulport_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.

Bit	Name	Description
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 then 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	Enbable	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. Type Mapping Bits Note 0 MPEG pixel 24 see notes on storage format

Bit	Name	Description			
		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_tvpe**	Pixel map type of UV image, 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
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

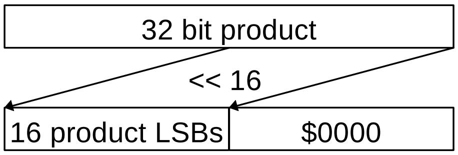
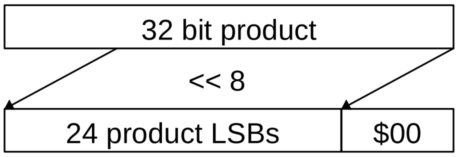
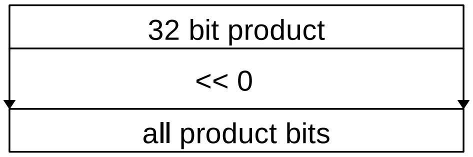
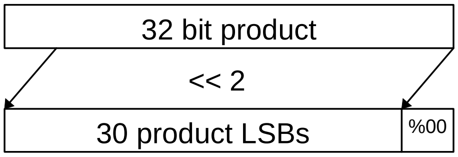
svshift Small Vector Multiply Shift Control

```
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
2		for the full 32-bit products
3		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.

Bits	Name	Description
14	**dmpe_error**	Command pointer error. One of the following things has occurred: • 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**.

commrecv 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 **commrecv0 - commrecv3**.

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 dcachable write is still being handled. This means that when any piece of code is about to write the mdmacptr or odmacptr 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/odmacptr 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/odmacptr 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.

vdmaptrb 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.

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
mirror	MEM	Reverse the bit order of a scalar

Mnemonic	**Function unit**	**Description**
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

Absolute Value

Function Unit: ALU

Operation: absolute_value (Scalar Register) ⇒ 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.

Scalar Addition

Function Unit: ALU

Operation: Scalar + Scalar ⇒ 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  Si,Sk`	add Si to Sk, writing the result to Sk	
`add  #n,Sk`	add #n to Sk, writing the result to Sk	`0 ≤ n  ≤ 31`
32-bit forms		
`add  Si,Sj,Sk`	add Si to Sj, writing the result to Sk	
`add  #n,Sj,Sk`	add #n to Sj, writing the result to Sk	`0 ≤ n  ≤ 31`
`add  #nn,Sk`	add #nn to Sk, writing the result to Sk	`0 ≤ nn ≤ 1023`

`add  #n,>>#m,Sk`	add #n arithmetically shifted right by #m, to Sk, writing the result to Sk	`0 ≤ n  ≤ 31`
		`-16 ≤ m  ≤ 0`

`add  Si,>>#m,Sk`	add Si arithmetically shifted right by #m, to Sk, writing the result to Sk	`-16 ≤ m  ≤ 15`
48-bit forms		
`add  #nnnn,Sk`	add #nnnn to Sk, writing the result to Sk	`-(2^31) ≤ nnnn ≤ (2^31)-1`
64-bit forms		
`add  #nnnn,Sj,Sk`	add #nnnn to Sj, writing the result to Sk	`-(2^31) ≤ nnnn ≤ (2^31)-1`

Operand Values:

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 : 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.

Pixel Addition

Function Unit: ALU

Operation: Pixel + Pixel ⇒ 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  Vi,Vj,Vk`	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.

add_sv Small Vector Addition add_sv

Function Unit: ALU

Operation: Small vector Source A + Small vector Source B ⇒ 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.

addm Scalar Addition using the MUL unit addm

Function Unit: MUL

Operation: Scalar + Scalar ⇒ 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  Si,Sj,Sk`	add Si to Sj writing the result to Sk

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.

Scalar Addition To RCU Index Register

Function Unit: RCU

Operation: Data + Index ⇒ 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 ≤ n ≤ 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) ≤ nnnn ≤ (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.

Scalar Addition With Carry

Function Unit: ALU

Operation: Scalar + Scalar + Carry Flag ⇒ 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  Si,Sj,Sk`	add c to Si to Sj, writing the result to Sk	
`addwc  #n,Sj,Sk`	add c to #n to Sj, writing the result to Sk	`0 ≤ n  ≤ 31`
`addwc  #nn,Sk`	add c to #nn to Sk, writing the result to Sk	`0 ≤ nn ≤ 1023`

`addwc  #n,>>#m,Sk`	add c to #n arithmetically shifted right by #m, to Sk, writing the result to Sk	`0 ≤ n  ≤ 31`
		`-16 ≤ m  ≤ 0`

`addwc  Si,>>#m,Sk`	add c to Si arithmetically shifted right by #m, to Sk, writing the result to Sk	`-16 ≤ m  ≤ 15`
64-bit forms		
`addwc  #nnnn,Sj,Sk`	add c to #nnnn to Sj, writing the result to Sk	`-(2^31) ≤ nnnn ≤ (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.

Logical AND

Function Unit: ALU

Operation: Scalar AND Scalar ⇒ 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 ≤ n  ≤ 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 ≤ n  ≤ 15`
		`-∞ ≤ m  ≤ ∞`

`and  #n,>>Sj,Sk`	AND #n logically shifted right by Sj, with Sk, writing the result to Sk	`-16 ≤ n  ≤ 15`
		`-32 ≤ Sj ≤ 31`

`and  Si,>>#m,Sk`	AND Si logically shifted right by #m, with Sk, writing the result to Sk	`-16 ≤ m  ≤ 15`
`and  Si,>>Sj,Sk` ∗	AND Si logically shifted right by Sj, with Sk, writing the result to Sk	`-32 ≤ Sj ≤ 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) ≤ nnnn ≤ (2^31)-1`

`and  #nnnn,>>Sj,Sk`	AND #nnnn logically shifted right by Sj, with Sk, writing the result to Sk	`-(2^31) ≤ nnnn ≤ (2^31)-1`
		`-32 ≤ Sj ≤ 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.

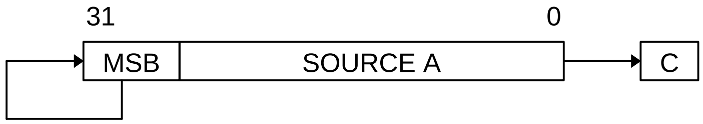
as Arithmetic Shift as

Function Unit: ALU

Operation: Scalar Source A >> Source B ⇒ Scalar Destination

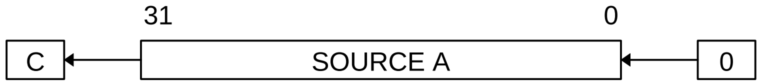
Description: Arithmetically shift source A either left or right by source B, setting flags appropriately, and writing the result to destination. Only the bottom six bits of Source B are used, the high-order 26 bits are ignored. A positive shift value implies a right shift; a negative shift value implies 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 >>Sj,Si,Sk`	arithmetical shift right of Si by Sj, writing the result to Sk.	`-32 ≤ Sj ≤ 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 (Sj ≥ 0), c takes the value of bit 0 of Source A;
for left shifts (Sj < 0), c takes the value of bit 31 of Source A.
v : cleared.

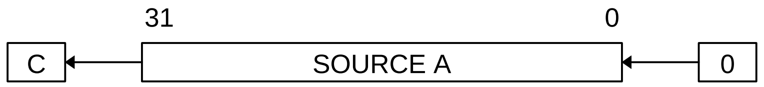
Other condition codes are unchanged by this instruction.

asl Arithmetic Shift Left asl

Function Unit: ALU

Operation: Scalar Source A << Source B ⇒ 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 ≤ m ≤ 31`
32-bit forms		
`asl  #m,Si,Sk`	arithmetic shift left of Si by #m, writing the result to Sk	`0 ≤ m ≤ 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.

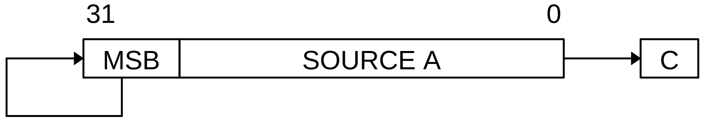
Other condition codes are unchanged by this instruction.

asr Arithmetic Shift Right asr

Function Unit: ALU

Operation: Scalar Source A >> Source B ⇒ 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 ≤ m ≤ 31`
32-bit forms		
`asr  #m,Si,Sk`	arithmetic shift right of Si by #m, writing the result to Sk	`0 ≤ m ≤ 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.

Clear a Bit in a Register

Function Unit: ALU

Operation: Scalar Destination AND Mask ⇒ 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 ≤ n ≤ 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.

Bit Field Extraction

Function Unit: ALU

Operation: Logical shift Scalar Destination right by m or Source i, mask above bit n ⇒ 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 ≤ n  ≤ 31`
		`0 ≤ Si ≤ 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 ≤ n  ≤ 31`
		`0 ≤ m  ≤ 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.

bra Branch To Relative Address bra

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 accomodate the **bra** offset, condition, and possible use of the **nop** operand.

Assembler Syntax:

INSTRUCTION	DESCRIPTION	RANGE RESTRICTIONS ON offset = <label> – pcexec
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 ≤ offset ≤ 126`
`bra  <label>`	After executing the next two packets, continue execution at <label>.	`-1024 ≤ offset ≤ 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 ≤ offset ≤ 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 ≤ offset ≤ 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) ≤ offset ≤ (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) ≤ offset ≤ (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
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 >= 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.

Set a Bit in a Register

Function Unit: ALU

Operation: Scalar Destination OR Mask ⇒ 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 ≤ n ≤ 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.

Bit Test

Function Unit: ALU

Operation: Test a bit of the Scalar Source ⇒ 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 ≤ m ≤ 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.

Butterfly Operation

Function Unit: ALU

Operation: Scalar Source A + Scalar Source B ⇒ Scalar Destination K
Scalar Source A – Scalar Source B ⇒ 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  Si,Sj,Hk`	write the butterfly function result (Sj+Si, Sj-Si) to the destination half-vector Hk

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.

Scalar Compare

Function Unit: ALU

Operation: Scalar - Scalar ⇒ 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  Si,Sj`	subtract Si from Sj	
`cmp  #n,Sj`	subtract #n from Sj	`0 ≤ n  ≤ 31`
32-bit forms		
`cmp  #nn,Sq`	subtract #nn from Sq	`0 ≤ nn ≤ 1023`

`cmp  #n,>>#m,Sq`	subtract #n arithmetically shifted right by #m from Sq	`0 ≤ n  ≤ 31`
		`-16 ≤ m  ≤ 0`

`cmp  Si,#n`	subtract Si from #n	`0 ≤ n  ≤ 31`
`cmp  Si,>>#m,Sq`	subtract Si arithmetically shifted right by #m from Sq	`-16 ≤ m  ≤ 15`
48-bit forms		
`cmp  #nnnn,Sj`	subtract #nnnn from Sj	`-(2^31) ≤ nnnn ≤ (2^31)-1`
64-bit forms		
`cmp  Si,#nnnn`	subtract Si from #nnnn	`-(2^31) ≤ nnnn ≤ (2^31)-1`

Operand Values:

Si	any scalar register r0-r31.
Sj	any scalar register r0-r31.
Sq	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.

Scalar Compare With Carry

Function Unit: ALU

Operation: Scalar - Scalar - Carry Condition Code ⇒ 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  Si,Sj`	subtract c and Si from Sj	
`cmpwc  #n,Sj`	subtract c and #n from Sj	`0 ≤ n ≤ 31`
`cmpwc  #nn,Sq`	subtract c and #nn from Sq	`0 ≤ nn ≤ 1023`

`cmpwc  #n,>>#m,Sq`	subtract c and #n arithmetically shifted right by #m from Sq	`0 ≤ n ≤ 31`
		`-16 ≤ m ≤ 0`

`cmpwc  Si,#n`	subtract c and Si from #n	`0 ≤ n ≤ 31`
`cmpwc  Si,>>#m,Sq`	subtract c and Si arithmetically shifted right by #m from Sq	`-16 ≤ m ≤ 15`
64-bit forms		
`cmpwc  #nnnn,Sj`	subtract c and #nnnn from Sj	`-(2^31) ≤ nnnn ≤ (2^31)-1`
`cmpwc  Si,#nnnn`	subtract c and Si from #nnnn	`-(2^31) ≤ nnnn ≤ (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.
Sq	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.

copy Copy Register to Register through the ALU copy

Function Unit: ALU

Operation: Scalar Source ⇒ Scalar Destination

Description: Copy the source register to the destination register though 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.

Decrement counter

Function Unit: RCU sub-instruction

Operation: c - 1 ⇒ 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.

Small Vector Dot Product

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.

eor Logical Exclusive OR eor

Function Unit: ALU

Operation: Scalar Exclusive-OR Scalar ⇒ 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 ≤ n  ≤ 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 ≤ n  ≤ 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 ≤ n  ≤ 15`
		`-∞ ≤ m  ≤ ∞`

`eor  #n,>>Sj,Sk`	exclusive-OR #n logically shifted right by Sj, with Sk, writing the result to Sk	`-16 ≤ n  ≤ 15`
		`-32 ≤ Sj ≤ 31`

`eor  Si,>>#m,Sk`	exclusive-OR Si logically shifted right by #m, with Sk, writing the result to Sk	`-16 ≤ m  ≤ 15`
`eor  Si,>>Sj,Sk` ✻	exclusive-OR Si logically shifted right by Sj, with Sk, writing the result to Sk	`-32 ≤ Sj ≤ 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) ≤ nnnn ≤ (2^31)-1`
64-bit forms		
`eor  #nnnn,Sj,Sk`	exclusive-OR #nnnn with Sj, writing the result to Sk	`-(2^31) ≤ nnnn ≤ (2^31)-1`

`eor #nnnn,>>Sj,Sk`	exclusive-OR #nnnn logically shifted right by Sj, with Sk, writing the result to Sk	`-(2^31) ≤ nnnn ≤ (2^31)-1`
		`-32 ≤ Sj ≤ 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.

Bit-Field Test

Function Unit: ALU

Operation: Scalar Source A AND Scalar Source B ⇒ 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 ≤ n  ≤ 15`

`ftst  #n,<>#m,Sq`	AND #n rotated right by #m, with Sq	`-16 ≤ n  ≤ 15`
		`-∞ ≤ m  ≤ ∞`

`ftst  #n,>>Sj,Sq`	AND #n logically shifted right by Sj, with Sq	`-16 ≤ n  ≤ 15`
		`-32 ≤ Sj ≤ 31`

`ftst  Si,>>#m,Sq`	AND Si logically shifted right by #m, with Sq	`-16 ≤ n  ≤ 15`
`ftst  Si,>>Sj,Sq` ✻	AND Si logically shifted right by Sj, with Sq	`-32 ≤ Sj ≤ 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) ≤ nnnn ≤ (2^31)-1`

`ftst  #nnnn,>>Sj,Sq`	AND #nnnn logically shifted right by Sj, with Sq	`-(2^31) ≤ nnnn ≤ (2^31)-1`
		`-32 ≤ Sj ≤ 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.

Halt Operation

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.

jmp Jump to Absolute Address jmp

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		
`jmp  cc,(Si) ✻`	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`
`jmp  cc,(Si),nop ✻`	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`
`jmp  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.	`any address in the first 16K bytes of local IROM or IRAM`
`jmp  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.	`any address in the first 16K bytes of local IROM or IRAM`
64-bit forms		
`jmp  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.	`32-bit absolute address`
`jmp  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.	`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`
`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`

Condition Codes: Unchanged by this instruction.

jsr Jump to Subroutine at Absolute Address jsr

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		
`jsr  cc,(Si) *`	If the cc condition is true: copy **pcfetchnext** to **rz**, execute the next two packets, continue execution at address Si. false: continue execution with the next packet.	`32-bit absolute address`
`jsr  cc,(Si),nop *`	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`
`jsr  cc,<label>`	If the cc condition is true: copy **pcfetchnext** 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`
`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.	`any address in the first 16K bytes of local IROM or IRAM`
64-bit forms		
`jsr  cc,<label>`	If the cc condition is true: copy **pcfetchnext** 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 >= 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.

ld_b Load byte data into register ld_b

Function Unit: MEM

Operation: Byte Data ⇒ 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 dtram
$2050_0000	base of local control registers

Condition Codes: Unchanged by this instruction.

Load pixel data into registers

Function Unit: MEM

Operation: Packed Pixel Data ⇒ 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.

ld_pz Load pixel plus Z data into registers ld_pz

Function Unit: MEM

Operation: Packed Pixel plus Z Data ⇒ 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.

ld_s Load scalar data into register ld_s

Function Unit: MEM

Operation: Scalar Data ⇒ 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.

Load small vector data into register

Function Unit: MEM

Operation: Packed Small Vector Data ⇒ 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.

Load vector data into register

Function Unit: MEM

Operation: Vector Data ⇒ 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.

Load word data into scalar register

Function Unit: MEM

Operation: Word Data ⇒ 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.

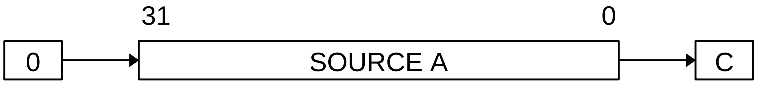
ls Logical Shift ls

Function Unit: ALU

Operation: Scalar Source A >> Source B ⇒ Scalar Destination

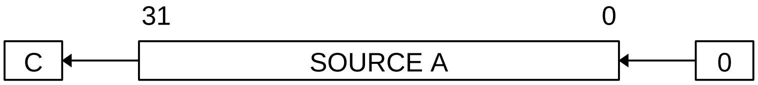
Description: Logically shift Source A either left or right by Source B, setting flags appropriately, and writing the result to destination. Only the bottom six bits of Source B are used, the high-order 26 bits are ignored. A positive shift value implies a right shift, a negative shift value implies a 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 ≤ Sj ≤ 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 (Sj ≥ 0), c takes the value of bit 0 of Source A;
for left shifts (Sj < 0), c takes the value of bit 31 of Source A.
v : cleared.

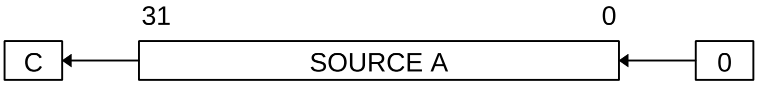
Other condition codes are unchanged by this instruction.

Logical Shift Left

Function Unit: ALU

Operation: Scalar Source A << Source B ⇒ 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		
`lsl  #m,Sk`	logical shift left of Sk by #m, writing the result to Sk	`0 ≤ m ≤ 31`
32-bit forms		
`lsl  #m,Si,Sk`	logical shift left of Si by #m, writing the result to Sk	`0 ≤ m ≤ 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.

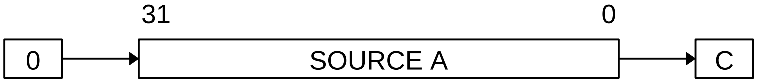
Other condition codes are unchanged by this instruction.

lsr Logical Shift Right lsr

Function Unit: ALU

Operation: Scalar Source A >> Source B ⇒ 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		
`lsr  #m,Sk`	logical shift right of Sk by #m, writing the result to Sk	`0 ≤ m ≤ 31`
32-bit forms		
`lsr  #m,Si,Sk`	logical shift right of Si by #m, writing the result to Sk	`0 ≤ m ≤ 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.

Reverse the Bits of a Register

Function Unit: MEM

Operation: Mirror (Scalar Source) ⇒ 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  Sj,Sk`	mirror scalar data from Sj, writing the result to Sk

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.

Find the Number of Significant Bits

Function Unit: ALU

Operation: sigbits(Scalar Source) ⇒ Scalar Destination

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

F*or 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.

F*or 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

N*ote* – 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.

Scalar Integer Multiplication

Function Unit: MUL

Operation: Scalar Source A * Scalar Source B ⇒ 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 ≤ m  ≤ 63`
`mul  Si,Sk,>>Sq,Sk  ✻`	multiply Si and Sk, arithmetically shift the product by Sq, and write the result to Sk	`-32 ≤ Sq ≤ 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 ≤ n  ≤ 31`

`mul  #n,Sk,>>#m,Sk`	multiply #n and Sk, arithmetically shift the product by #m, and write the result to Sk	`0 ≤ n  ≤ 31`
		`-32 ≤ m  ≤ 63`

`mul  #n,Sk,>>Sq,Sk  ✻`	multiply #n and Sk, arithmetically shift the product by Sq, and write the result to Sk	`0 ≤ n  ≤ 31`
		`-32 ≤ Sq ≤ 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.

Pixel Integer Multiplication

Function Unit: MUL

Operation: Pixel Source A * Pixel Source B ⇒ 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.

Small Vector Integer Multiplication

Function Unit: MUL

Operation: Small vector Source A * Small vector Source B ⇒ 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 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.

Move Scalar Data in Register File

Function Unit: MEM

Operation: Scalar Source ⇒ 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 ≤ n ≤ 15`
32-bit forms		
`mv_s  #nnn,Sk`	move #nnn into register Sk	`-2048 ≤ nnn ≤ 2047`
48-bit forms		
`mv_s  #nnnn,Sk`	move #nnnn into register Sk	`-(2^31) ≤ nnnn ≤ (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.

Move Vector Data in Register File

Function Unit: MEM

Operation: Vector Source ⇒ 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.

Move Scalar Data to Index Register

Function Unit: RCU

Operation: Scalar Source ⇒ 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) ≤ nnnn ≤ (2^31)-1`

Operand Values:

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

Condition Codes: Unchanged by this instruction.

neg Negate neg

Function Unit: ALU

Operation: Zero minus Scalar Destination ⇒ 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.

Null Operation

Function Unit: none

Operation: Null Operation

Description: Do nothing.

Assembler Syntax:

INSTRUCTION	DESCRIPTION
16-bit forms	
`nop`	

Condition Codes: Unchanged by this instruction.

not Logical Complement not

Function Unit: ALU

Operation: Scalar Destination exclusive-OR $FFFFFFFF ⇒ 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 ⇒ 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 ≤ n  ≤ 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 ≤ n  ≤ 15`
		`-∞ ≤ m  ≤ ∞`

`or  #n,>>Sj,Sk`	OR #n logically shifted right by Sj, with Sk, writing the result to Sk	`-16 ≤ n  ≤ 15`
		`-32 ≤ Sj ≤ 31`

`or  Si,>>#m,Sk`	OR Si logically shifted right by #m, with Sk, writing the result to Sk	`-16 ≤ m  ≤ 15`
`or  Si,>>Sj,Sk` ✻	OR Si logically shifted right by Sj, with Sk, writing the result to Sk	`-32 ≤ Sj ≤ 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) ≤ nnnn ≤ (2^31)-1`

`or  #nnnn,>>Sj,Sk`	OR #nnnn logically shifted right by Sj, with Sk, writing the result to Sk	`-(2^31) ≤ nnnn ≤ (2^31)-1`
		`-32 ≤ Sj ≤ 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.

Pad Instruction Packet

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.

POP data from stack

Function Unit: MEM

Operation: pop 16 bytes of data from the stack in memory ⇒ Registers
sp + 16 ⇒ **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  Vk`	Pop the vector register Vk from stack.
`pop  Vk,rz`	Pop first three elements of vector Vk 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:

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

PUSH data on to stack

Function Unit: MEM

Operation: **sp** - 16 ⇒ **sp**
push 16 bytes of data from registers ⇒ 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.

Index Register Range Check

Function Unit: RCU

Operation: IF (Index >= 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.

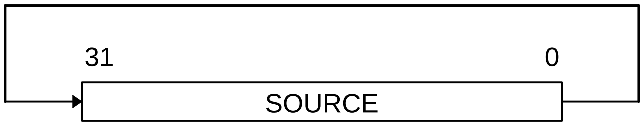
Condition Codes: modmi : set if RI was less then zero, cleared otherwise.
modge : set if RI was greater than or equal to the range, cleared otherwise.

Other condition codes are unchanged by this instruction.

Function Unit: ALU

Operation: Scalar Source B <> Source A ⇒ 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	-∞ ≤ m ≤ ∞

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		
`rti  cc,(rzi1)`	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`
`rti  cc,(rzi1),nop`	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`
`rti  cc,(rzi2)`	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`
`rti  cc,(rzi2),nop`	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`
`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`

Condition Codes: Unchanged by this instruction.

rts Return From Subroutine rts

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		
`rts  cc`	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`
`rts  cc,nop`	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
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

Condition Codes: Unchanged by this instruction.

Saturate

Function Unit: ALU

Operation: Saturate (Scalar Source) ⇒ 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 ≤ n ≤ 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.

st_p Store pixel data st_p

Function Unit: MEM

Operation: Pixel Data ⇒ 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 dtram
$2050_0000	base of local control registers

Condition Codes: Unchanged by this instruction.

Store pixel plus Z data

Function Unit: MEM

Operation: Pixel plus Z Data ⇒ 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 dtram
$2050_0000	base of local control registers

Condition Codes: Unchanged by this instruction.

Store scalar data to memory

Function Unit: MEM

Operation: Scalar Source ⇒ 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.
<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 dtram $2050_0000 base of local control registers

…continued

Condition Codes: Unchanged by this instruction.

Store small vector data

Function Unit: MEM

Operation: Small-Vector Data ⇒ 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.

st_v Store vector data to memory st_v

Function Unit: MEM

Operation: Vector Register ⇒ 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 dtram
$2050_0000	base of local control registers

Condition Codes: Unchanged by this instruction.

Scalar Subtraction

Function Unit: ALU

Operation: Scalar - Scalar ⇒ 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  Si,Sk`	subtract Si from Sk, writing the result to Sk	
`sub  #n,Sk`	subtract #n from Sk, writing the result to Sk	`0 ≤ n  ≤ 31`
32-bit forms		
`sub  Si,Sj,Sk`	subtract Si from Sj, writing the result to Sk	
`sub  #n,Sj,Sk`	subtract #n from Sj, writing the result to Sk	`0 ≤ n  ≤ 31`
`sub  #nn,Sk`	subtract #nn from Sk, writing the result to Sk	`0 ≤ nn ≤ 1023`

`sub  #n,>>#m,Sk`	subtract #n arithmetically shifted right by #m, from Sk, writing the result to Sk	`0 ≤ n  ≤ 31`
		`-16 ≤ m  ≤ 0`

`sub  Si,#n,Sk`	subtract Si from #n, writing the result to Sk	`0 ≤ n  ≤ 31`
`sub  Si,>>#m,Sk`	subtract Si arithmetically shifted right by #m, from Sk, writing the result to Sk	`-16 ≤ m  ≤ 15`
48-bit forms		
`sub  #nnnn,Sk`	subtract #nnnn from Sk, writing the result to Sk	`-(2^31) ≤ nnnn ≤ (2^31)-1`
64-bit forms		
`sub  #nnnn,Sj,Sk`	subtract #nnnn from Sj, writing the result to Sk	`-(2^31) ≤ nnnn ≤ (2^31)-1`
`sub  Si,#nnnn,Sk`	subtract Si from #nnnn, writing the result to Sk	`-(2^31) ≤ nnnn ≤ (2^31)-1`

Operand Values:

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 : 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.

Scalar Subtraction With Carry

Function Unit: ALU

Operation: Scalar - Scalar - Carry Condition Code ⇒ 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  Si,Sj,Sk`	subtract c and Si from Sj, writing the result to Sk	
`subwc  #n,Sj,Sk`	subtract c and #n from Sj, writing the result to Sk	`0 ≤ n  ≤ 31`
`subwc  #nn,Sk`	subtract c and #nn from Sk, writing the result to Sk	`0 ≤ nn ≤ 1023`

`subwc  #n,>>#m,Sk`	subtract c and #n arithmetically shifted right by #m, from Sk, writing the result to Sk	`0 ≤ n  ≤ 31`
		`-16 ≤ m  ≤ 0`

`subwc  Si,#n,Sk`	subtract c and Si from #n, writing the result to Sk	`0 ≤ n  ≤ 31`
`subwc  Si,>>#m,Sk`	subtract c and Si arithmetically shifted right by #m, from Sk, writing the result to Sk	`-16 ≤ m  ≤ 15`
64-bit forms		
`subwc  #nnnn,Sj,Sk`	subtract c and #nnnn from Sj, write the result to Sk	`-(2^31) ≤ nnnn ≤ (2^31)-1`
`subwc  Si,#nnnn,Sk`	subtract c and Si from #nnnn, write the result to Sk	`-(2^31) ≤ nnnn ≤ (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 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.

Pixel Subtraction

Function Unit: ALU

Operation: Pixel Source A - Pixel Source B ⇒ 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  Vi,Vj,Vk`	subtract pixel Vi from 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.

Small Vector Subtraction

Function Unit: ALU

Operation: Small Vector Source A - Small Vector Source B ⇒ 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.

Arithmetic Subtraction using the MUL unit

Function Unit: MUL

Operation: Scalar Source A – Scalar Source B ⇒ 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  Si,Sj,Sk`	subtract Si from Sj and write the result to Sk

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.

MAIN BUS

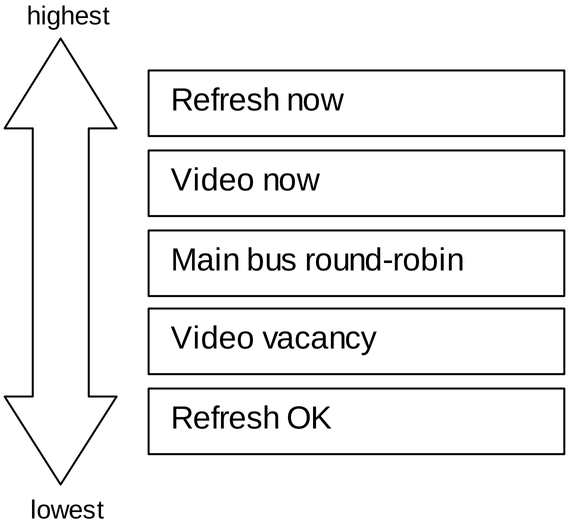
The Main Bus is used by the MPEs, the video display generator, and the audio DMA channel to transfer data to and from SDRAM, or from one location to another in internal memory. It has a 32-bit data channel transferring data at a maximum rate of one long per clock cycle.

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:

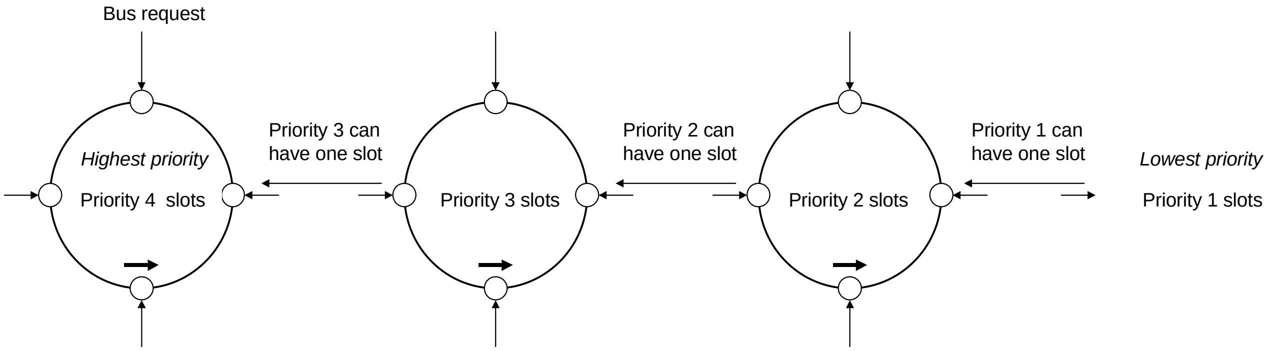


Figure 3 - Round robin prioritized bus arbitration

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 isbusy; 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. The 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.

Flags

SDRAM or MPE address

MPE address (abs/rel)

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.

Flags

SDRAM or MPE addr.(abs/rel)

Data

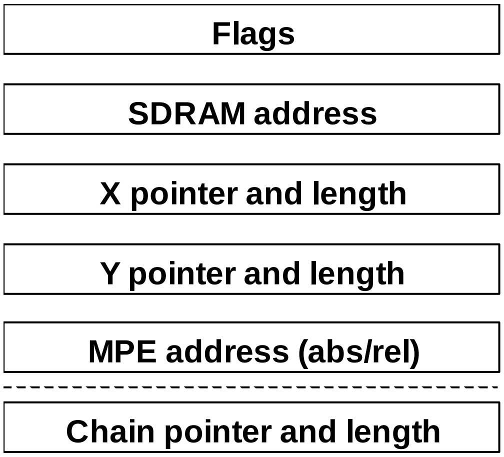
The address field is the destination address. It may be anywhere in SDRAM or internal MPE memory, and may be specified as a system address when the REMOTE bit is set, or as a local MPE address when the REMOTE bit is clear.

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.



Flags

SDRAM address

X pointer and length

Y pointer and length

MPE address (abs/rel)

Chain pointer and length

This field only used when CHAIN is set, and is repeated for as many scan lines as required.

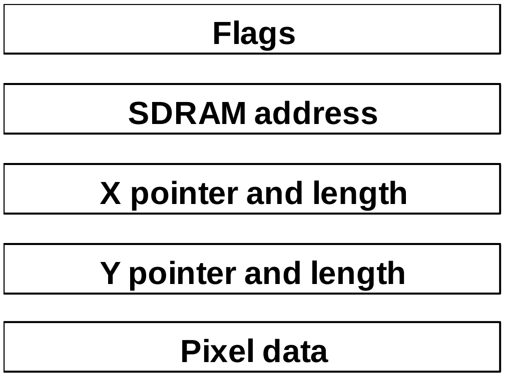
The first address field is the destination address of a write, or the source address of a read. This is sometimes called the base address. It must be on a 128-byte boundary in SDRAM. If the cluster bit in the command-mode bits is set, then the base address must be on a 512-byte boundary in SDRAM.

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.



Flags

SDRAM address

X pointer and length

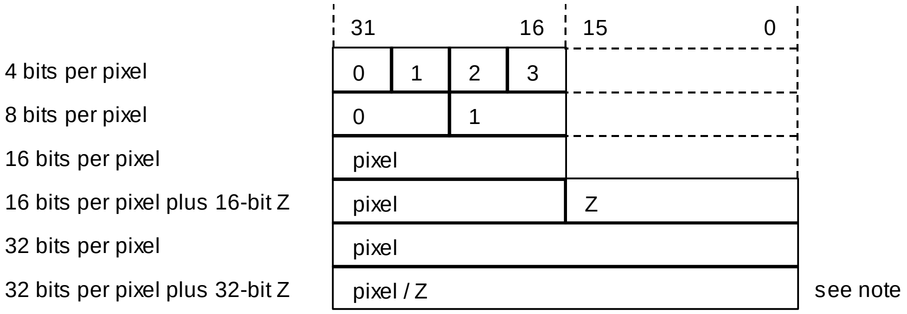
Y pointer and length

Pixel data

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.
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. 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

Bits	Name	Description
		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 **`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 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

Bits	Name	Description
		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 a16/16 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 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: • 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.
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.

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 of 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:

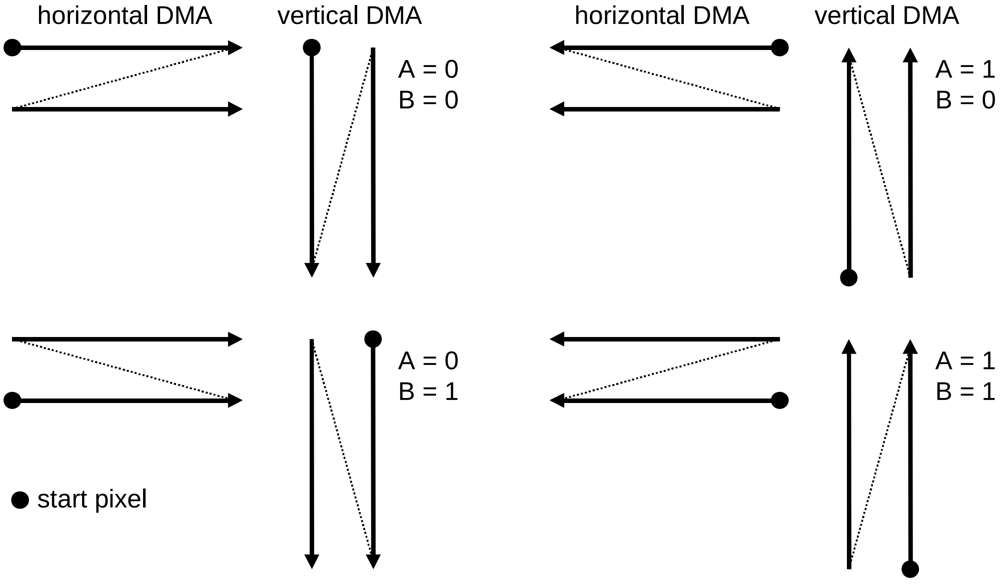


Figure 4 - Backwards DMA Types

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 …	
	previous transfer	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	*next bus acknowledge*
	next transfer target ID	

Transfer to receiver with buffer full:

Requester action	**Bus contents**	**Controller action**
Bus request		
	… 0-L_{max} clock cycles …	
	previous transfer	Bus acknowledge
Present target ID	Target ID and Sender ID	Check for target full, abort the transfer in this case.
	Idle	
	next transfer target ID	

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 busis 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 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

Bits	Read / Write	Description
		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:

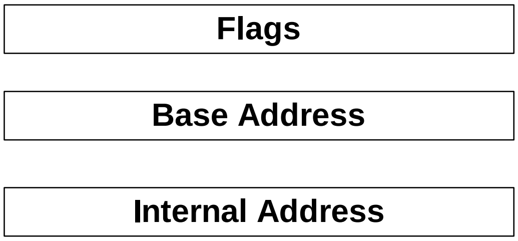


Figure 5 - Other Bus DMA Command Format

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 supportsa variety of 16-bit wide SDRAM configurationss, 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**	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. 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). 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. 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. 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.
29	**xhostDataLo**	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: sys_sd[31:0] for 32-bit masters sys_sd[31:16] for 16-bit masters sys_sd[31:24] for 8-bit masters 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: sys_sd[31:0] for 32-bit masters sys_sd [15:0] for 16-bit masters sys_sd[7:0] for 8-bit masters *Aries 3 and up only*.
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	8XXXXXXXh	Disabled
0	1	0	Disabled	9XXXXXXXh
0	1	1	Disabled	8XXXXXXXh
1	1	0	8XXXXXXXh	9XXXXXXXh
1	1	1	8XXXXXXXh	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)

Bit	Name	Description
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.

physical address = logical address - $80000000 + (**sysOffset** << 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 this spec.

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.

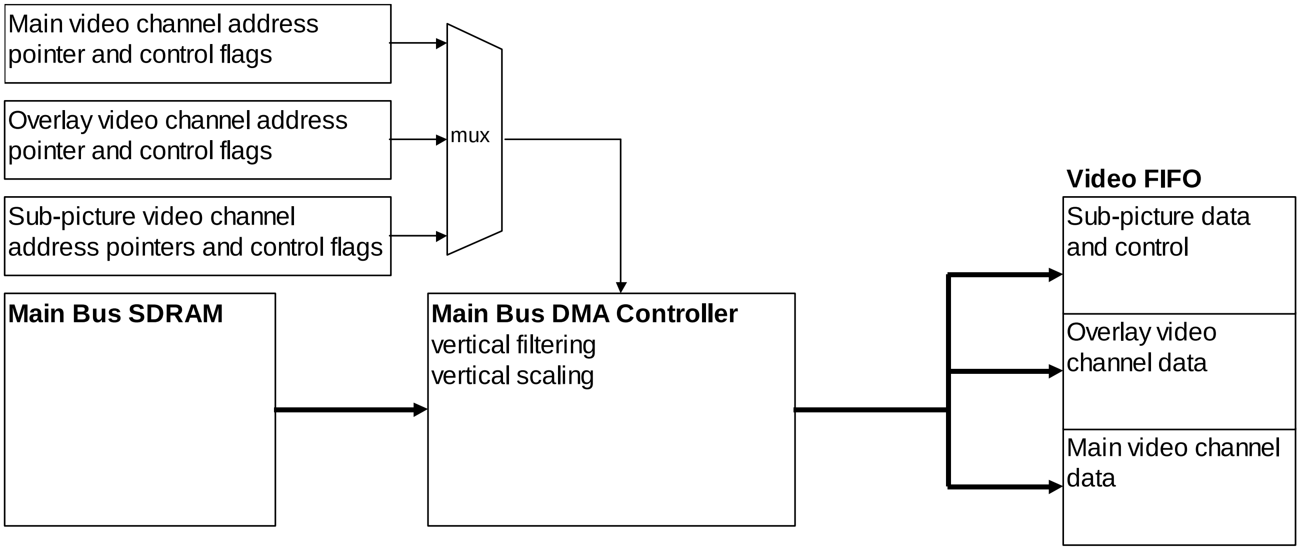

Main video channel address pointer and control flags
Overlay video channel address pointer and control flags
Sub-picture video channel address pointers and control flags
mux
Main Bus SDRAM
Main Bus DMA Controller
vertical filtering
vertical scaling
Video FIFO
Sub-picture data and control
Overlay video channel data
Main video channel data

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:

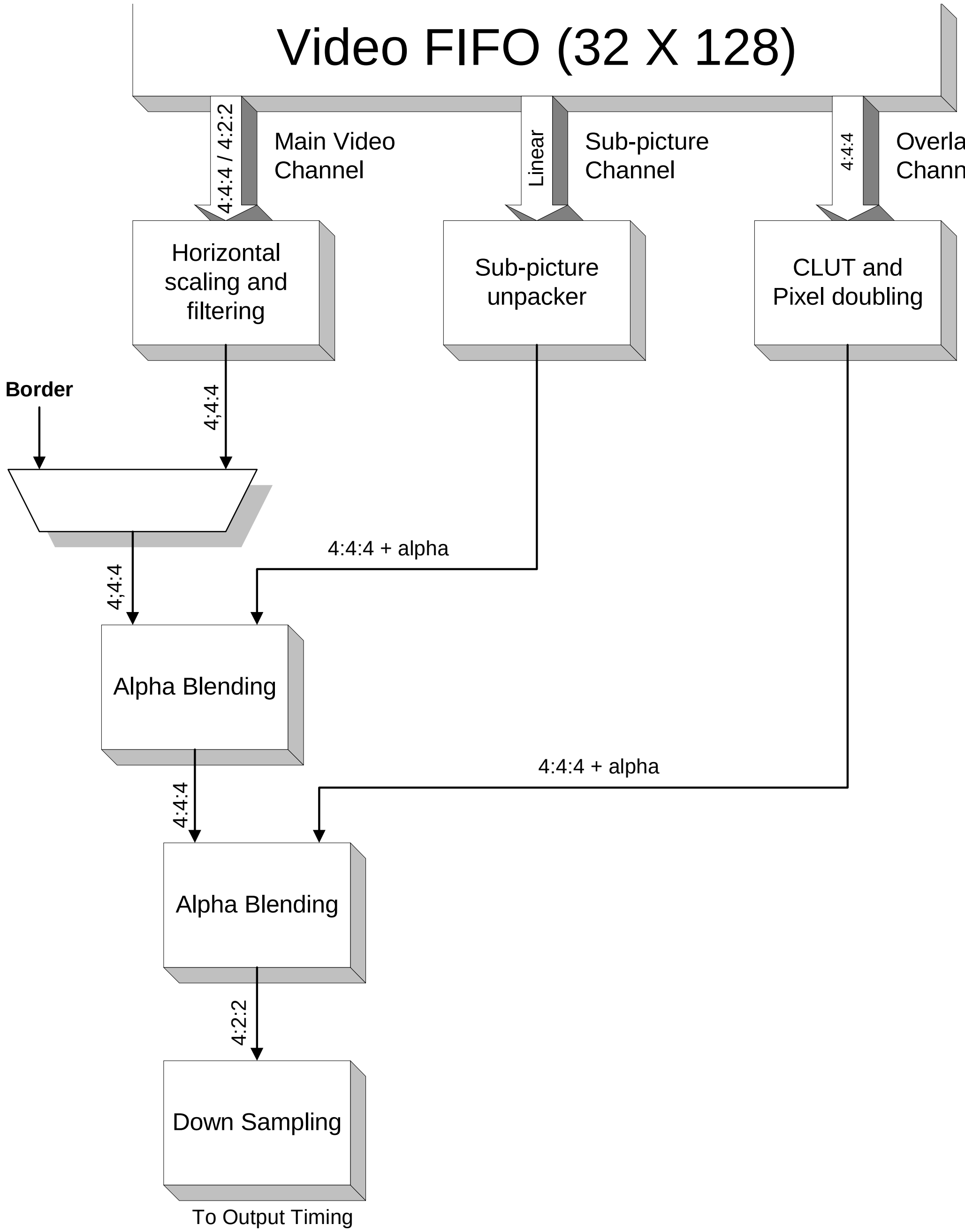


Figure 7 - data flow from the video FIFO to the video output

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

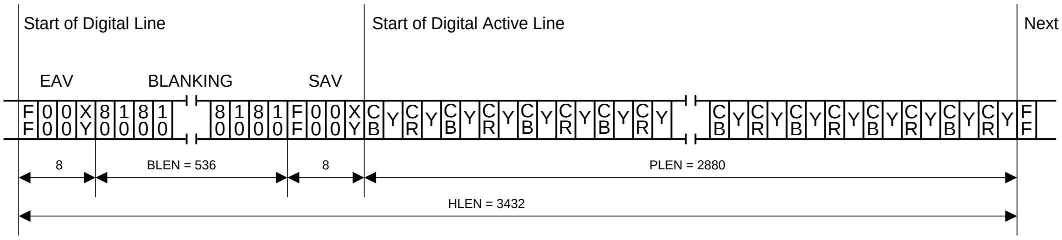
The display address generator will always fetch lines of pixel running from left to right upwards through memory. The choice of data formats available is described below. The base address of the display map is programmable, as is the amount to add to the base address for each successive line of the display. Display line data does not therefore have to be contiguous.

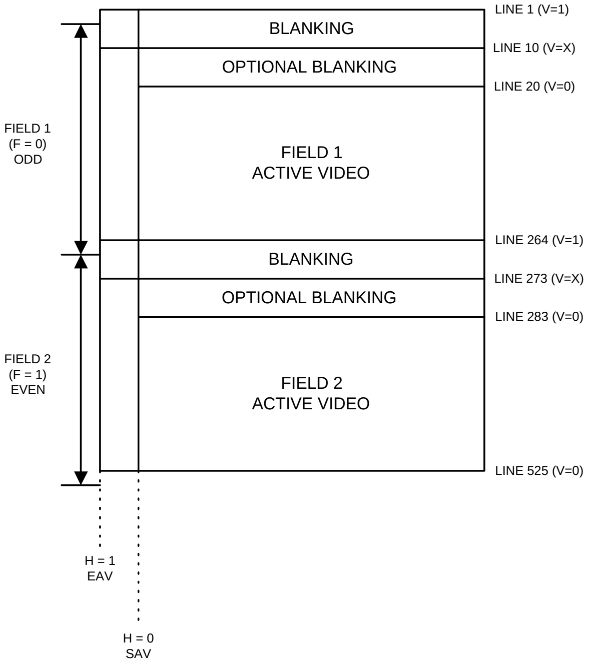
Video Time-Base

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:


Start of Digital Line
Start of Digital Active Line
Next Line
EAV
BLANKING
SAV
FF 00 00 XY 80 10 80 10
80 10 80 10 FF 00 00 XY CB Y CR Y CB Y CR Y CB Y CR Y CB Y CR Y
CB Y CR Y CB Y CR Y CB Y CR Y CB Y CR Y FF
8
BLEN = 536
8
PLEN = 2880
HLEN = 3432



LINE NUMBER	F	V	H (EAV)	H (SAV)
1-3	1	1	1	0
4-19	0	1	1	0
20-263	0	0	1	0
264-265	0	1	1	0
266-282	1	1	1	0
283-525	1	0	1	0

Gen-lock to external Syncs

The display engine is capable of gen-locking to external syncs. As described above, the video time base generation is controlled by two counters, namely, the horizontal counter and the vertical counter. By programming the display engine into synchronization mode (10 - gen-lock to sync. Input), these counters will be reset based on the falling edge of incoming syncs (HSYNC and FIELD). Specifically, the Horizontal Counter will reset to “1” one tick after the falling edge of HSYNC. Meanwhile, the vertical counter will reset to the value programmed in the “vidFst ($0140)” register two ticks after the falling edge of FIELD.

Video Data Encoding

The video data output and input formats supported by Aries conforms to CCIR 656 and CCIR 601. Refer to these standards for further details.

The output data stored in Main Bus DRAM, or the corresponding CLUT entries, are always in YCrCb form. The relationship between this and RGB is defined as follows by CCIR 601:

If R, G and B are values between 0 and 1, then

$$E_Y = 0.299\ R + 0.587\ G + 0.114\ B$$
$$E_{CR} = 0.713(R - E_Y) = 0.500\ R - 0.419\ G - 0.081\ B$$
$$E_{CB} = 0.564(B - E_Y) = -\ 0.169\ R - 0.331\ G + 0.500\ B$$

The conversion to 8 bit integers uses the following formulae, where the computed value is rounded to the nearest integer.

$Y = 219 E_{Y} + 16$
$CR = 224 E_{CR} + 128$
$CB = 224 E_{CB} + 128$

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 ±½.

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.

Control Registers

The registers described below control the display fetch mechanism and the video time-base generator. These registers are programmed over the Communication Bus. . The VDG has Communication Bus ID 65 ($41).

The communication protocol is as follows for the command packet. Note that there are two write modes and only one read mode:

Mode 1

Long word	Description
0	0-15 register address 31 set for write, clear for read
1	0-31 write data if a write command
2	unused
3	unused

Mode 2

Long word	Description
0	31 must be set to 0 30 must be set to 1 29-24 register address [5:0] 23-0 write data
1	29-24 register address [5:0] 23-0 write data
2	29-24 register address [5:0] 23-0 write data
3	29-24 register address [5:0] 23-0 write data

A response packet is returned if the operation was a read. Its format is:

Long word	Description
0	unused
1	0-31 read data
2	unused
3	unused

List of registers

vidCtrl (0) Display Control (0)

```
mode 1: $0000
mode 2: $00
Read / Write
```

Display control register.

Mode 1

Bit	Description
31	Reserved, write zero.
30	Active low Horizontal Sync. When this bit is set HSYNC will be active low.
28-25	Offset in 4bpp overlay. These four bits will be used as upper address bits when accessing the CLUT in 4 bits per pixel mode.
24	Sub-picture overlay enable.
20-16	Overlay display mode
15	Overlay enable
12-11	Synchronization mode. 00 free running (VDG generates a CCIR656 stream) 01 reserved 10 gen-lock to sync inputs 11 reserved
10	Reserved, write zero.
9	Expand – expand each of the color components to include illegal CCIR656 (1 to 254) values. In other words, if this bit is NOT set, the output CCIR656 streams will be truncated into the valid range (see above).
8	Video enable. When this is clear, the video generator outputs synchronization pulses, but the display is blanked.
7	Main display enable
4-0	Display mode. See the list below.

Mode 2

Bit	Description
5	Expand – expand each of the color components to include illegal CCIR656 (1 to 254) values. In other words, if this bit is NOT set, the output CCIR656 streams will be truncated into the valid range (see above).
4	Video enable. When this is clear, the video generator outputs synchronization pulses, but the display is blanked.
3	Reserved, write zero.
2	Active low Horizontal Sync. When this bit is set HSYNC will be active low.
1-0	Synchronization mode. 00 free running (VDG generates a CCIR656 stream) 01 reserved 10 gen-lock to sync inputs 11 reserved

vidCtrl (0) Display Control (1)

```
mode 2: $01
Read / Write
```

Display control register.

Mode 2

Bit	Description
23	Sub-Picture Display Enable.
19-16	Offset in 4bpp overlay. These four bits will be used as upper address bits when accessing the CLUT in 4 bits per pixel mode.
15	Overlay display enable.

12-8	Overlay display mode.
7	Main display enable.
4-0	Display mode. See the list below.

pixHscale (0) Main Video Channel Horizontal scaling control (0)

```
mode 1: $0014
mode 2: $02
Read / Write
```

This register controls horizontal scaling. After each pixel is generated the horizontal increment is added to the horizontal counter, and if the integer field advances then pixel pointer is advanced by one pixel. The horizontal scale counter itself is not in this register, it is concealed and is initialized from the initial value given here at the start of each display line. Its fractional bits control the re-sizing filter.

Mode 1

Bit	Description
31	No Filter bit – Only sub-sampling will be performed, no filter will be applied
30	Expand bit – The output will not be clipped
29-16	Horizontal scale increment (3.11 bits)
15	Linear Filter – The linear filter coefficients will be used.
14	Buffer Underflow.
12-0	Horizontal scale counter initial value (2.11 bits)

Mode 2

Bit	Description
13-0	Horizontal scale increment (3.11 bits)

pixHscale (1) Main Video Channel Horizontal scaling control (1)

```
mode 2: $03
Read / Write
```

Mode 2

Bit	Description
17	No Filter bit – Only sub-sampling will be performed, no filter will be applied
16	Expand bit – The output will not be clipped
15	Linear Filter – The linear filter coefficients will be used.
14	Buffer Underflow.
12-0	Horizontal scale counter initial value (2.11 bits)

pixFifo Main Video Channel FIFO Control

```
mode 1: $0020
mode 2: $04
Read / Write
```

Controls the video FIFO for the Main Video Channel display data.

Bit	Description
23	Overflow of main video FIFO, for debug only.
21-17	Number of entries in the FIFO below which the FIFO become a high bus priority.

15-11	5-bit address specifying the last word of the FIFO within the 32x128 RAM cell.
5-1	5-bit address specifying where the FIFO starts within the 32x128 RAM cell.

ovlFifo Overlay Video Channel FIFO Control

```
mode 1: $0024
mode 2: $05
Read / Write
```

Controls the video FIFO for the Overlay Video Channel display data.

Bit	Description
23	Overflow of overlay video FIFO, for debug only.
21-17	Number of entries in the FIFO below which the FIFO become a high bus priority.
15-11	5-bit address specifying the last word of the FIFO within the 32x128 RAM cell.
5-1	5-bit address specifying where the FIFO starts within the 32x128 RAM cell.

subDFifo Sub-Picture FIFO Control

```
mode 1: $0028
mode 2: $06
Read / Write
```

Controls the video FIFO for the sub-picture display data.

Bit	Description
23	Overflow of sub-picture data FIFO, for debug only.
21-17	Number of entries in the FIFO below which the FIFO become a high bus priority.
15-11	5-bit address specifying the last word of the FIFO within the 32x128 RAM cell.
5-1	5-bit address specifying where the FIFO starts within the 32x128 RAM cell.

subCFifo Sub-Picture control code FIFO Control

```
mode 1: $002C
mode 2: $07
Read / Write
```

Controls the video FIFO for the sub-picture display control code.

Bit	Description
23	Overflow of sub-picture control FIFO, for debug only.
21-17	Number of entries in the FIFO below which the FIFO become a high bus priority.
15-11	5-bit address specifying the last word of the FIFO within the 32x128 RAM cell.
5-1	5-bit address specifying where the FIFO starts within the 32x128 RAM cell.

mainFifoS Status of Main FIFO Control

```
mode 1: $0030
Read
```

Status register that provides current FIFO pointers information.

Bit	Description
21-16	6-bit specifying the count of number of entries for this FIFO
12-8	5-bit specifying the write pointer current location.
4-0	5-bit specifying the read pointer current location.

ovlFifoS Status of Overlay FIFO Control

```
mode 1: $0034
Read
```

Status register that provides current FIFO pointers information.

Bit	Description
21-16	6-bit specifying the count of number of entries for this FIFO
12-8	5-bit specifying the write pointer current location.
4-0	5-bit specifying the read pointer current location.

subDFifoS Status of Subpicture Data FIFO Control

```
mode 1: $0038
Read
```

Status register that provides current FIFO pointers information.

Bit	Description
22-16	7-bit specifying the count of number of entries for this FIFO
12-8	5-bit specifying the write pointer current location.
5-0	6-bit specifying the read pointer current location.

subCFifoS Status of Subpicture Control FIFO Control

```
mode 1: $003C
Read
```

Status register that provides current FIFO pointers information.

Bit	Description
Bit	**Description**
22-16	7-bit specifying the count of number of entries for this FIFO
12-8	5-bit specifying the write pointer current location.
5-0	6-bit specifying the read pointer current location.

vidBord Border Color

```
mode 1: $0080
mode 2: $08
Read / Write
```

Border color. This 24-bit color value is displayed at any non-blanked pixel position that is not part of the Main Video Channel or Overlay Video Channel display areas.

vidHcnt Video horizontal counter

```
mode 1: $0084
mode 2: $09
Read / Write
```

This may be read to determine the display refresh position, and may be written to for debug purposes. This counter counts in 54MHz cycles, and resets every vidHlen clock cycles.

vidHlen Video horizontal count length

```
mode 1: $0088
mode 2: $0A
Read / Write
```

This defines the length of a video line in system clock cycles. See the table below for a list of recommended settings for PAL and NTSC display.

vidVcnt Video vertical counter

```
mode 1: $008C
mode 2: $0B
Read / Write
```

This may be read to determine the display refresh position, and may be written to for debug purposes. This counts video lines, and resets every vidVlen lines.

vidVlen Video Vertical Count Length

```
mode 1: $0090
mode 2: $0C
Read / Write
```

This defines the height of a video field in video lines. See the table below for a list of recommended settings for PAL and NTSC display.

pixHstart Main Video Channel Horizontal Start

```
mode 1: $0094
mode 2: $0D
Read / Write
```

Horizontal start position of the Main Video Channel bit-map pixel data display. This is value given by a horizontal count value. See the table below for a list of recommended settings for PAL and NTSC display.

pixHend Main Video Channel Horizontal End

```
mode 1: $0098
mode 2: $0E
Read / Write
```

Horizontal end position of the Main Video Channel bit-map pixel data display, given in the same manner as the start. (pixHend - pixHstart) / 4 gives the number of active pixels per line. See the table below for a list of recommended settings for PAL and NTSC display.

ovlHstart Overlay Video Channel Horizontal Start

```
mode 1: $009C
mode 2: $0F
Read / Write
```

Horizontal start position of the Overlay Video Channel bit-map pixel data display. This is value given by a horizontal count value. To align with the main channels this should be set to pixHstart + 1 + n * 4 with n = integer value. Higher values of n will offset it by n pixels to the right of the main video channel. See the table below for a list of recommended settings for PAL and NTSC display.

ovlHend Overlay Video Channel Horizontal End

```
mode 1: $00A0
mode 2: $10
Read / Write
```

Horizontal end position of the Overlay Video Channel bit-map pixel data display, given in the same manner as the start. See the table below for a list of recommended settings for PAL and NTSC display.

brdHstart Border Color Horizontal Start

```
mode 1: $00A4
mode 2: $11
Read / Write
```

Horizontal start position of the non-blanked display. Any non-blanked pixel that is not within the Main Video Channel or Overlay Video Channel bit-map areas is displayed in the border color. This value is given by a horizontal count value. See the table below for a list of recommended settings for PAL and NTSC display. NOTE: This register MUST be setup correctly for the display to work.

brdHend Border Color Horizontal End

```
mode 1: $00A8
mode 2: $12
Read / Write
```

Horizontal end position of the non-blanked display, given in the same manner as the start. See the table below for a list of recommended settings for PAL and NTSC display. NOTE: This register MUST be setup correctly for the display to work.

pixVstart Pixel Display Vertical Start

```
mode 1: $00AC
mode 2: $13
Read / Write
```

Vertical start position of pixel data display. This is given by a vertical count value. vidAoff will be added to this value to generate the correct value for the second field. See the table below for a list of recommended settings for PAL and NTSC display.

pixVend Pixel Display Vertical End

```
mode 1: $00B0
mode 2: $14
Read / Write
```

Vertical end position of the pixel data display, given in the same manner as the start. pixVend minus pixVstart gives the number of active main channel lines in a field. vidAoff will be added to this value to generate the correct value for the second field. See the table below for a list of recommended settings for PAL and NTSC display.

ovlVstart Overlay Video Channel Vertical Start

```
mode 1: $00B4
mode 2: $15
Read / Write
```

Vertical start position of the Overlay Video Channel bit-map pixel data display. This is value given by a vertical count value. vidAoff will be added to this value to generate the correct value for the second field. See the table below for a list of recommended settings for PAL and NTSC display.

ovlVend Overlay Video Channel Vertical End

```
mode 1: $00B8
mode 2: $16
Read / Write
```

Vertical end position of the Overlay Video Channel bit-map pixel data display, given in the same manner as the start. ovlVend minus ovlVstart gives the number of active overlay video lines in a field. vidAoff will be added to this value to generate the correct value for the second field. See the table below for a list of recommended settings for PAL and NTSC display.

brdVstart Border Color Vertical Start

```
mode 1: $00BC
mode 2: $17
Read / Write
```

Vertical start position of the non-blanked display. Any non-blanked pixel, which is not within the Main Video Channel or Overlay Video Channel bit-map areas, is displayed in the border color. This value is given by a vertical count value. vidAoff will be added to this value to generate the correct value for the second field. See the table below for a list of recommended settings for PAL and NTSC display. NOTE: This register MUST be setup correctly for the display to work.

brdVend Border Color Vertical End

```
mode 1: $00C0
mode 2: $18
Read / Write
```

Vertical end position of the non-blanked display, given in the same manner as the start. vidAoff will be added to this value to generate the correct value for the second field. See the table below for a list of recommended settings for PAL and NTSC display. NOTE: This register MUST be setup correctly for the display to work.

vidBlen Video Blanking length

```
mode 1: $0100
mode 2: $19
Read / Write
```

This defines the length of Blanking pixels in each line. See the table below for a list of recommended settings for PAL and NTSC display, and follow them. Use other values at your own risk.

pixPlen Main Channel Video Active Pixel length

```
mode 1: $0104
mode 2: $1A
Read / Write
```

This defines the length of active pixels in each line, and is programmed with four times that number, so is set to 2880 for 720 pixels. This is the number of pixels going into the output channel, and if horizontal scaling is performed it will not be the same as the number of output pixels. In other words, this number defines the amount of DMA fetches per line

synHstart Start Position Of Horizontal Pulse

```
mode 1: $0108
mode 2: $1B
Read / Write
```

This is a value given by a horizontal count value. The actual active edge will start at count + 1 See the table below for a list of recommended settings for PAL and NTSC display.

synHend End Position Of Horizontal Pulse

```
mode 1: $010C
mode 2: $1C
Read / Write
```

This is a value given by a horizontal count value. The actual assert edge will start at count + 1. See the table below for a list of recommended settings for PAL and NTSC display.

vidAline First Active Line Register

```
mode 1: $0114
mode 2: $1E
Read / Write
```

This value defines the first active line in the FIRST field. See the table below for a list of recommended settings for PAL and NTSC display.

vidAlen Length Of Active Lines for first field

```
mode 1: $0118
mode 2: $1F
Read / Write
```

This value defines the number of active lines within the first (top) field. See the table below for a list of recommended settings for PAL and NTSC display.

vidAoff Next Field Offset

```
mode 1: $011C
mode 2: $20
Read / Write
```

Offset from vidAline where the next field will start. See the table below for a list of recommended settings for PAL and NTSC display.

vidFline Odd Field First Line

```
mode 1: $0120
mode 2: $21
Read / Write
```

Line value for the first line of the odd field F=0. See the table below for a list of recommended settings for PAL and NTSC display.

vidFlen Odd Field Last Line

```
mode 1: $0124
mode 2: $22
Read / Write
```

Line value for the last line of the odd field F=0. See the table below for a list of recommended settings for PAL and NTSC display.

vidHint Horizontal Counter Value For Interrupt Position

```
mode 1: $0128
mode 2: $23
Read / Write
```

This is the Horizontal interrupt register. The value written here should be a valid horizontal count (1 to vidHlen). Once a value is written it remains valid until a system reset or power up.

vidVint Vertical Counter Value For Interrupt Position

```
mode 1: $012C
mode 2: $24
Read / Write
```

This is the Vertical interrupt register. The value written here should be a valid vertical count (1 to vidVlen). Once a value is written it remains valid until a system reset or power up. The hardware will generate one interrupt per video field, but by suitably re-programming these registers on the fly, multiple video interrupts can occur in each field.

vidInit Video Initialization Pulse

```
mode 1: $0130
mode 2: $25
Read / Write
```

This register will generate a one 54MHz-tick pulse when written to. This pulse is used to reset the pointers for all the holding registers within each channel based upon the type of pixels. Hence, it **must be** written to whenever the bits per pixel is re-defined.

vidFlush Video FIFO Flushing Pulse

```
mode 1: $0134
mode 2: $26
Read / Write
```

Writing to individual bits will either clear or keep the content of the FIFO for that channel:

Bit	Description
4	Flush DMA interface holding registers. 0 – Flush, 1 - Keep
3	Flush Sub-Picture Control Channel FIFO. 0 – Flush, 1 - Keep
2	Flush Sub-Picture Data Channel FIFO. 0 – Flush, 1 - Keep
1	Flush Overlay Channel FIFO. 0 – Flush, 1 - Keep
0	Flush Main Channel FIFO. 0 – Flush, 1 - Keep

vidVrst Video Reset Pulse

```
mode 1: $0138
mode 2: $27
Read / Write
```

Writing to this location will reset the horizontal and vertical counters to their respective start values.

vidFst Video Frame Start Line Count

```
mode 1: $0140
mode 2: $28
Write only
```

During genlock mode, when the start of an odd field is detected, the vertical counter will be set to this value.

vidAlen2 Length Of Active Lines for second field

```
mode 1: $0144
mode 2: $29
Read / Write
```

This value defines the number of active lines within the SECOND (bottom) field. See the table below for a list of recommended settings for PAL and NTSC display.

vidLeftEdge Left Edge starting count

```
mode 1: $0148
mode 2: $2A
Read / Write
```

When the horizontal counter matches this value, the pix_pos will start counting to define left edge of the screen. This is to inform the Sub-Picture module where the left edge of the screen is, since the sub-picture co-ordinates reference to the active screen rather than the sub-picture window. It is **highly recommended** that one should follow the values given in the table.

ovlCtrl Overlay Video Channel Control

```
mode 1: $0150
mode 2: $2B
Read / Write
```

This register sets up various control factors in the Overlay Video Channel.

Bit	Description
9	Buffer Underflow.
8	Pixel doubling. Setting this bit will allow the overlay video channel to double every pixel.

Bit	Description
7-0	Graphics Alpha. This 8-bit value will be used as the alpha blending index when displaying in 16-bit per pixel mode. Zero is opaque.

ovlPlen Overlay Video Channel Active Pixel Length

```
mode 1: $0160
mode 2: $2C
Read / Write
```

This defines the number of pixels per line to be fetched from memory for the overlay video channel. Similar to vidPlen for the main channel.

vidHedge Ending edge of Horizontal blanking

```
mode 1: $0170
mode 2: $2D
Read / Write
```

When Hcount reaches this value, a one-tick pulse will be generated in the next tick to instruct the Sub-Picture Control Unit to clear the holding register and to re-cycle the FIFO. This is set at 500 by default because we need to provide enough time for the software to reload the FIFO if needed. Yet, at the same time, enough time is also given to the Sub-Picture Control to pre-fetch for the next command. See the table below for a list of recommended settings for PAL and NTSC display.

ovlClut Overlay Video Channel CLUT

```
mode 1: $0200 - $02FF
Read / Write
```

The corresponding register location represents the CLUT location. Each location is a 32-bit entry with following organization:

Bit	Name	Description
31-24	**Y**	Luminance value
23-16	**Cr**	Chrominance value
15-8	**Cb**	Chrominance value
7-0	**Alpha**	Transparency attribute . Zero is opaque.

subCtrl Sub-Picture Control

```
mode 1: $0300
mode 2: $2E
Read / Write
```

Sub-Picture control register.

Bit	Description
31-15	Reserved
14	Use Pixel Alpha. When this bit is set, the alpha value for a given pixel will be extracted from the CLUT. Each entry in the CLUT consists of 32-bit of Y Cr Cb Alpha. If this bit is not set, the alpha will be expanded from the associating contrast value. The Contrast and the Alpha has opposite meanings, Hence to expand it to the proper value, it needs to be subtracted from $F first.
13-9	Number of Pixel Control Bytes to ignore at the beginning of the line.

	This assumes that the "garbage control" will only be present at the FIFO at the beginning of every **line**. Unless a new value is written, the same value from the last write will be used.
8-4	Number of Pixel Data Bytes to ignore at the beginning of the field. This assumes that the "garbage data" will only be present at the FIFO at the beginning of every **field**. Unless a new value is written, the same value from the last write will be used.
3-0	CLUT Select. This nibble will be used as the top 4-address lines that index the CLUT. Since the CLUT has 256 entries, this will allow a selection of 16 different color set.

subHstart Sub-Picture Horizontal Start

```
mode 1: $0304
mode 2: $2F
Read / Write
```

Horizontal start position of the Sub-Picture bit-map pixel data display. The sub-Picture Unit will start generating CCIR656 outputs 2 ticks after the horizontal counter reaches this value. This value is given by a horizontal count value. This should be set to pixHstart + n * 4 with n being the pixel offset from the main video channel. Note that the Horizontal Counter runs on the system clock, which is assumed to be 54MHz = 2 times pixel clock. See the table below for a list of recommended settings for PAL and NTSC display.

subHend Sub-Picture Horizontal End

```
mode 1: $0308
mode 2: $30
Read / Write
```

Horizontal end position of the Sub-Picture bit-map pixel data display, given in the same manner as the start. This is calculated by subHstart + (number of pixels * 4) – 3. See the table below for a list of recommended settings for PAL and NTSC display.

subVstart Sub-Picture Vertical Start

```
mode 1: $030C
mode 2: $31
Read / Write
```

Vertical start position of Sub-Picture pixel data display. This is given by a vertical count value. See the table below for a list of recommended settings for PAL and NTSC display.

subVend Sub-Picture Vertical End

```
mode 1: $0310
mode 2: $32
Read / Write
```

Vertical end position of Sub-Picture pixel data display, given in the same manner as the start. subVend minus subVstart gives the number of active lines in a field See the table below for a list of recommended settings for PAL and NTSC display.

subPcount Sub-Picture PXCTLI Count

```
mode 1: $0314
mode 2: $33
Read / Write
```

This 4-bit count gives the number of PXCTLI package to be processed, when this count is reached, it will start from the beginning again. The default is zero.

subCVstart Sub-Picture Change Line Vertical Start

```
mode 1: $0328
mode 2: $34
Read / Write
```

Vertical start position of Sub-Picture control channel. This is given by a vertical count value. See the table below for a list of recommended settings for PAL and NTSC display.

subCVend Sub-Picture Change Line Vertical End

```
mode 1: $032C
mode 2: $35
Read / Write
```

Vertical end position of Sub-Picture control channel, given in the same manner as the start. subCVend minus subCVstart gives the number of active lines in a field See the table below for a list of recommended settings for PAL and NTSC display.

subColor Sub-Picture Color Register

```
mode 1: $0330
mode 2: $36
Read / Write
```

This register allows the sub-picture color values to be accessed.

Bit	Name	Description
31-24	**Color3**	
23-16	**Color2**	
15-8	**Color1**	
7-0	**Color0**	

subContrast Sub-Picture Contrast Register

```
mode 1: $0334
mode 2: $37
Read / Write
```

This register allows the sub-picture contrast values to be accessed.

Bit	Name	Description
31-24	**Contrast3**	
23-16	**Contrast2**	
15-8	**Contrast1**	
7-0	**Contrast0**	

subPlen Sub-Picture Data Channel Active Pixel Length

```
mode 1: $0338
mode 2: $38
Read / Write
```

This defines the number of pixels (in ticks) per line to be fetched from memory for the Sub-Picture Data channel.

subInit Sub-Picture Initialization Pulse

```
mode 1: $0340
mode 2: $39
Read / Write
```

This is the Sub-Picture Initialization pulse. It should be written to after the DMA is set up, along with all the sub-picture settings and before the beginning of the active region of a field.

subDebug_1 Sub-Picture Debug Register

```
mode 1: $0344
mode 2: $3A
Read only
```

This is the first Sub-Picture debug register.

subDebug_2 Sub-Picture Debug Register

```
mode 1: $0348
mode 2: $3B
Read / Write
```

This is the second Sub-Picture debug register. The **subSoftReset** bit of this register **must** be written to bring Sub-Picture out of reset, as it is active on system reset. Writing a zero (0) to the bottom bit will take the Sub-Picture out of reset.

Bit	Name	Description
31-1		Reserved
0	**subSoftReset**	Soft reset for the Sub-Picture Channel

pixReset Main Channel Reset Register

```
mode 1: $0018
mode 2: $3C
Read / Write: bit0 Only
```

This is the Soft Reset register for the Main Channel. On System reset, it is inactive. You should set the **pixSoftReset** bit and then clear it again to put the Main Channel in reset state.

This should only be used in special circumstances; it should not be touched in normal operation.

Bit	Name	Description
31-1		Reserved
0	**pixSoftReset**	Soft reset for the Main Channel

ovlReset Overlay Channel Reset Register

```
mode 1: $0154
mode 2: $3D
Read / Write
```

This is the Soft Reset register for the Overlay Channel. On System reset, it is inactive. You should set the **ovlSoftReset** bit and then clear it again to put the Overlay Channel into reset state.

This should only be used in special circumstances; it should not be touched in normal operation.

Bit	Name	Description
31-1		Reserved
0	**ovlSoftReset**	Soft reset for the Overlay Channel

NTSC/PAL settings

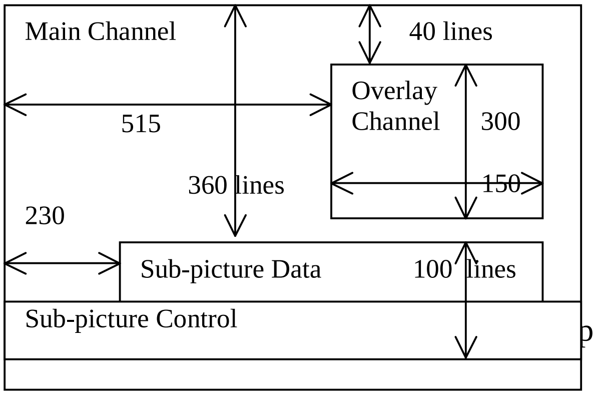
The display engine is built to support various formats by making the timing generator programmable. The most likely candidates for this would be either NTSC or PAL. Once the timing generator is setup, the associating registers should not need any further programming throughout the frames. Hence, for the ease of programming, the following list of registers and it associating values for each format is given.

Register Name	Address	NTSC	PAL
vidHlen	$0088/$0A	3432	3456
vidVlen	$0090/$0C	525	625
brdHstart	$00A4/$11	543	567
brdHend	$00A8/$12	3423	3447
brdVstart	$00BC/$17	10	23
brdVend	$00C0/$18	264	311
vidBlen	$0100/$19	536	560
synHstart	$0108/$1B	3432	3456
synHend	$010C/$1C	544	568
vidAline	$0114/$1E	10	23
vidAlen	$0118/$1F	253	287
vidAlen2	$0144/$29	252	287
vidAoff	$011C/$20	263	313
vidFline	$0120/$21	4	1
vidFlen	$0124/$22	261	311
vidLeftEdge	$0148/$2A	539	563
vidHedge	$0170/$2D	500	524

For the normal case of displaying 720 pixels horizontally and either 480 lines vertically for NTSC or 576 lines vertically for PAL, the channels should be programmed to start and end as follows, this will produce an image with all three channel filling up the entire screen:

Register Name	**Address**	**NTSC**	**PAL**
pixHstart	`$0094/$0D`	543	567
pixHend	`$0098/$0E`	3423	3447
ovlHstart	`$009C/$0F`	544	568
ovlHend	`$00A0/$10`	3424	3448
subHstart	`$0304/$2F`	543	567
subHend	`$0308/$30`	3420	3444
pixVstart	`$00AC/$13`	21	23
pixVend	`$00B0/$14`	261	311
ovlVstart	`$00B4/$15`	21	23
ovlVend	`$00B8/$16`	261	311
subVstart	`$030C/$31`	21	23
subVend	`$0310/$32`	261	311
subCVstart	`$0320/$34`	21	23
subCVend	`$032C/$35`	261	311

Let us consider another example, the following diagram shows an NTSC 720x480 screen:



play,

We should use these values for the main channel:

pixVstart = 21

pixVend = 21 + 240 (half a frame = one field) = 261

pixHstart = 543 (left edge of display for main channel)

pixHend = 543 + (720 x 4) = 3423

Now, for the Overlay channel,

ovlVstart = pixVstart + 20 = 41

ovlVend = ovlVstart + 150 (half a frame = one field) = 191 ovlHstart = 544 (left edge of display for Overlay channel) + (515 x 4) = 2604

ovlHend = ovlHstart + (150 * 4) = 3204

We will need to program both the data channel and the control channel for the sub-picture stream:

For Data channel:

subVstart = pixVstart + 180 = 201

subVend = subVstart + 50 = 251

subHstart = 543 (left edge of display for sub-picture channel) + (230 x 4) = 920

subHend = subHstart + (500 x 4) = 2920

For Control channel:

subCVstart = subVstart + 20 = 221

subCVend = subCVstart + 30 (half a frame = one field) = 251

Display Data Formats

The display modes that follow correspond to the value set in the video control register. The set at the start of the list are intended for synthesized displays, such as those created by 3D rendering; the set at the end of the list are for MPEG-2 modes.

Mode 0 – 4 bits per pixel (pixel map type 1)

This display mode is made up of pixel map type 1 data. This gives 4 bits per pixel, which are expanded to 24 bits and an alpha value by using them as an index in a color look-up table.

Mode 1 – 16 bits per pixel (pixel map type 2)

This display mode is made up of pixel map type 2 data. This gives 16 bits per pixel, which are expanded to 24 bits by adding zeroes in the least significant bit positions.

Mode 2 – 24-bits per pixel (pixel map type 4)

This display mode is made up of pixel map type 4 data. This gives 24 bits per pixel, plus an 8-bit control value that is normally ignored, unless the data is used as an overlay, in which case it controls the degree of transparency. A value of $00 corresponds to opaque, a value of $FF corresponds to transparent.

Mode 3 – 24-bits per pixel (pixel map type 6)

This display mode is made up of pixel map type 6 data. This gives 24 bits per pixel. The remaining bits in the phrase are ignored.

Mode 4 – 16-bits per pixel (pixel map type 7)

This display mode is made up of pixel map type 7 data, i.e. the display generator uses the data for display buffer A, as modes 7-9 represent different views of the same triple buffer packed data. The other data is ignored.

Mode 5 – 16-bits per pixel (pixel map type 8)

This display mode is made up of pixel map type 8 data, i.e. the display generator uses the data for display buffer B. The other data is ignored.

Mode 6 – 16-bits per pixel (pixel map type 9)

This display mode is made up of pixel map type 9 data, and the display generator uses the data for display buffer C. The other data is ignored.

Mode 7 – 16-bits per pixel (pixel map type A)

This display mode is made up of pixel map type A data, i.e. the display generator uses the data for display buffer A, as modes A-B represent different views of the same double buffer packed data. The other data is ignored.

Mode 8 – 16-bits per pixel (pixel map type B)

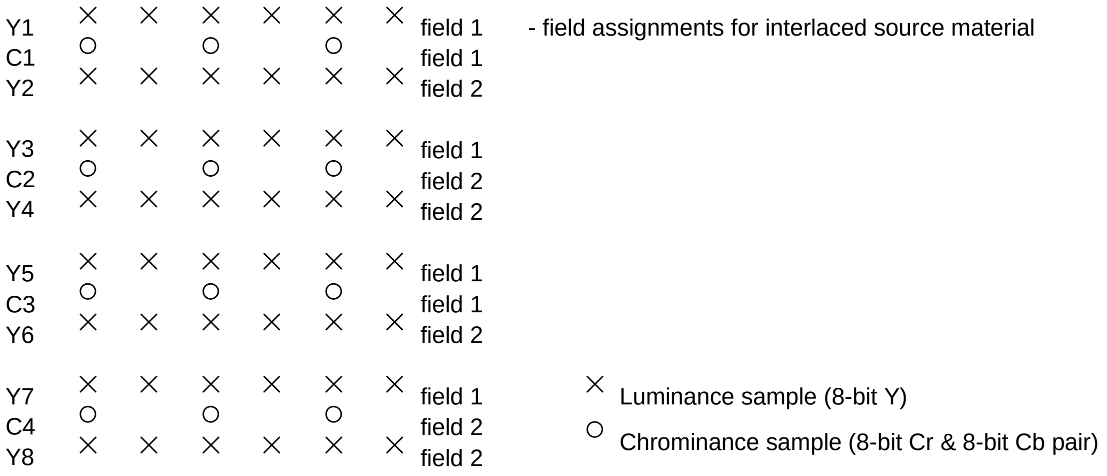
This display mode is made up of pixel map type B data, i.e. the display generator uses the data for display buffer B, as modes A-B represent different views of the same double buffer packed data. The other data is ignored.

Mode 9 – 8 bits per pixel (pixel map type 3)

This display mode is made up of pixel map type 3 data. This gives 8 bits per pixel, which are expanded to 24 bits and an alpha value by using them as an index in a color look-up table.

MPEG Display Modes

The source data is stored in memory in separate Luminance and Chrominance maps in raster scan order. The (logical) position of these samples on the display is like this:


Y1
C1
Y2
field 1
field 1
field 2
- field assignments for interlaced source material
Y3
C2
Y4
field 1
field 2
field 2
Y5
C3
Y6
field 1
field 1
field 2
Y7
C4
Y8
field 1
field 2
field 2
Luminance sample (8-bit Y)
Chrominance sample (8-bit Cr & 8-bit Cb pair)

The field assignments shown are for interlaced source material; for a frame coded picture there is no implied frame assignment. Note that either field 1 or field 2 may be first in temporal order. This storage format is known as 4:2:0, which seems to be short for 4:2:2 / 4:0:0.

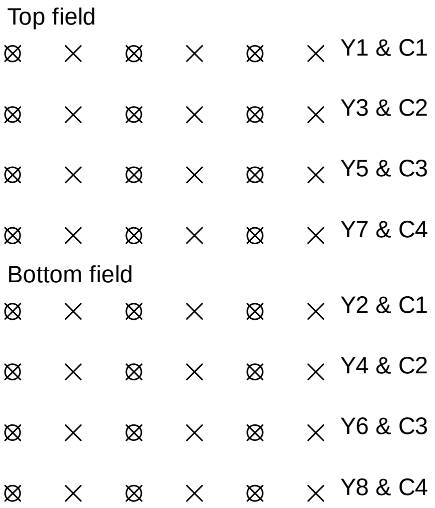
If the source material was progressively scanned (such as a movie), then the relationship between chrominance samples and luminance samples is straightforward, and is as shown in the picture above.

However, if the source material was interlaced (such as the output from a video camera), then the two fields that make up this frame are temporally separate, and care should be taken when reconstructing the chroma to only use those chroma samples appropriate to the field currently being displayed.

In a frame picture, with frame DCT encoding, each block is composed of lines from the two fields alternately. In a frame picture with field DCT encoding, or a field picture, each block is composed of lines from one of the two fields. Chroma blocks are always encoded in the frame structure, that is, lines from the two fields alternately. However, this really only affects how they are stored in memory, the logical numbering remains as shown above. It does mean that the display generator has to understand both organizations.

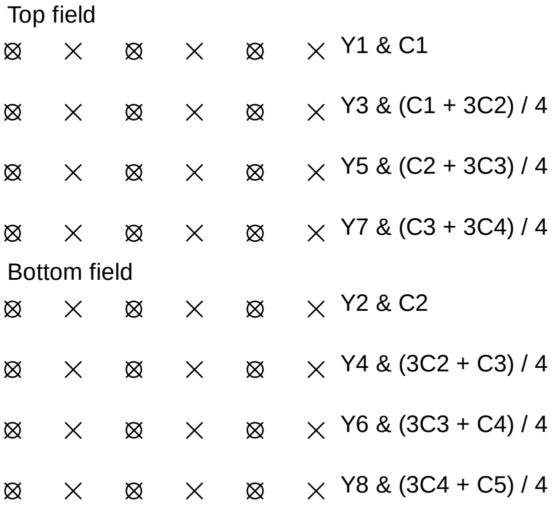
When no vertical scaling is performed, there are four possible display options, given by one of the two modes outlined above, and depending on whether or not interpolation is required. These modes are:

Mode 16 – Progressive Scan Source, No Chroma Interpolation



In this mode, the conversion from the 4:2:0 format in memory to the 4:2:2 format used for display is carried out by repeating the chrominance data in both fields.

Mode 17– Progressive Scan Source, Chroma Interpolation



In this mode, the conversion from the 4:2:0 format in memory to the 4:2:2 format used for display is carried out by interpolating the chroma values. Reference to the source data map above for progressive scan will explain the 1:3 weighting of the chroma values. The first line of both fields is handled specially.

Mode 18 – Interlaced Scan Source, No Chroma Interpolation

Top field	
⊠ × ⊠ × ⊠ ×	Y1 & C1
⊠ × ⊠ × ⊠ ×	Y3 & C1
⊠ × ⊠ × ⊠ ×	Y5 & C3
⊠ × ⊠ × ⊠ ×	Y7 & C3
Bottom field	
⊠ × ⊠ × ⊠ ×	Y2 & C2
⊠ × ⊠ × ⊠ ×	Y4 & C2
⊠ × ⊠ × ⊠ ×	Y6 & C4
⊠ × ⊠ × ⊠ ×	Y8 & C4

In this mode, the conversion from the 4:2:0 format in memory to the 4:2:2 format used for display is carried out by using the nearest chrominance data in the same field as the pixel being displayed.

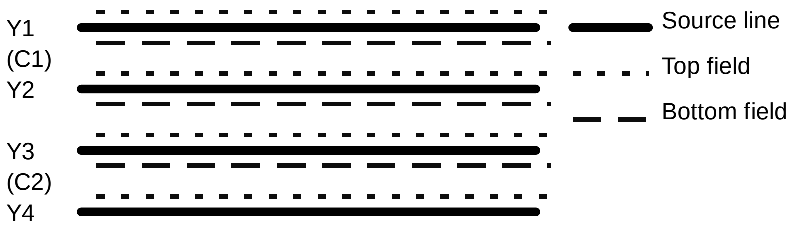
Mode 19 – Interlaced Scan Source, Chroma Interpolation

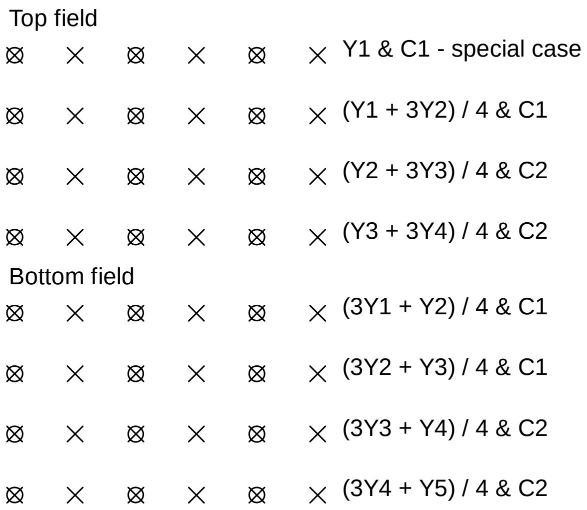
	Accurate interpolation	Approximate interpolation
Top field		
⊠ × ⊠ × ⊠ ×	Y1 & C1	Y1 & C1
⊠ × ⊠ × ⊠ ×	Y3 & (5C1 + 3C3) / 8	Y3 & (C1 + C3) / 2
⊠ × ⊠ × ⊠ ×	Y5 & (C1 + 7C3) / 8	Y5 & C3
⊠ × ⊠ × ⊠ ×	Y7 & (5C3 + 3C5) / 8	Y7 & (C3 + C5) / 2
Bottom field		
⊠ × ⊠ × ⊠ ×	Y2 & C2	Y2 & C2
⊠ × ⊠ × ⊠ ×	Y4 & (7C2 + C4) / 8	Y4 & (3C2 + C4) / 4
⊠ × ⊠ × ⊠ ×	Y6 & (3C2 + 5C4) / 8	Y6 & (C2 + 3C4) / 4
⊠ × ⊠ × ⊠ ×	Y8 & (7C4 + C6) / 8	Y8 & (3C4 + C6) / 4

In this mode, things start getting hairy. The correct interpolation values can be approximated to, and this approximation is worth carrying out to save complexity in the vertical interpolation. It also gives equal weighting to all chroma lines, which the “accurate” figures do not. Note that in both cases, the rounding in the approximation is in the same vertical direction.

Mode 20 – Display of MPEG-1 Material

This procedure involves doubling both the horizontal and the vertical / temporal resolution of the source material. MPEG-1 data is always progressive scan, and luminance interpolation is performed as shown below. Chroma interpolation is not performed.



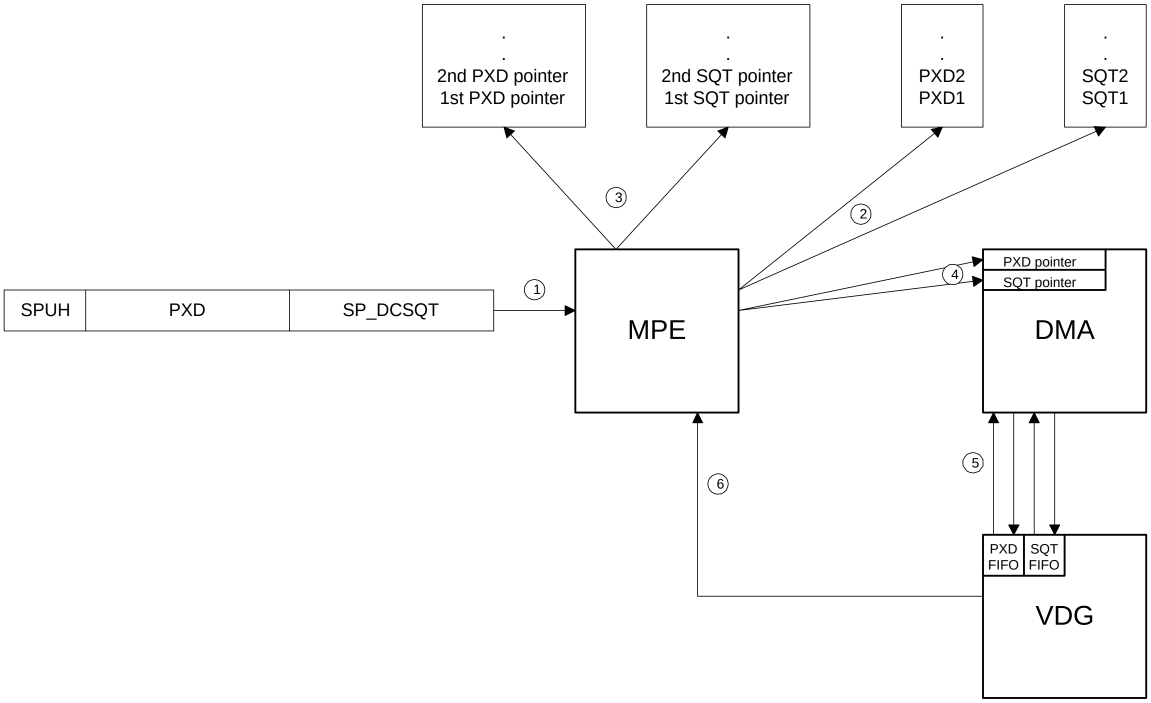


Sub-Picture Video Channel

The Sub-Picture channel supports the display of Sub-Picture stream as described in the “DVD Specifications for Read-Only Disc Version 1.0”. You should read and understand the appropriate sections of that document before reading this section.

The Sub-Picture Video Channel actually consists of two physical DMA channels. One is the data channel (referred to as the SubD channel) and the other is the control code channel (referred to as the SubC channel). A software process is required to set up these channels and keep track of the decoding process. The following description of decoding sequence should help in understanding the overall operation of this channel:

Sub-picture decoding and display sequence



Software will perform the following steps:

First the software will parse the pack SPU to obtain the SPU structure.

1. The packet headers and the SPU headers will provide the necessary information for the software to store the PXD data and SQT commands in memory.
2. Based on the various SPU time stamps and the sequence, a display list is build and placed in memory.
3. The software will then write to the DMA pointers and initiate the DMA transfers.

Hardware will perform the following steps:

4. VDG will issues request whenever the FIFO is low. Once the DMA is enable, PXD data will be provided to the SubD FIFO and the SQT commands will be provided to the SubC FIFO.
5. Once the SPU is done, an interrupt will be sent to the MPE. The MPE will then go down the display list for the next pointers and start from step 4 again.

The DMA will supply both the SubD and SubC channel with linear data. The SubD channel unpacks the data according to the following Run-length compression scheme:

1. If there is a strip of 1 to 3 pixels with the same value, enter the number of the pixel in the first 2 bits and the pixel data in the following 2 bits. The 4 bits are considered to be one unit.

d3 d2	d1 d0
Number of pixel(s)	Pixel Data

2. If there is a strip of 4 to 15 pixels with the same value, specify '0' in the first 2 bits, and enter the number of the pixels in the following 4 bits and the pixel data in the next 2 bits. The 8 bits are considered to be one unit.

d7	d6	d5 d4 d3 d2	d1 d0

0	0	Number of pixels	Pixel Data

3. If there is a strip of 16 to 63 pixels with the same value, specify '0' in the first 4 bits, and enter the number of the pixels in the following 6 bits and the pixel data in the next 2 bits. The 12 bits are considered to be one unit.

d11	d10	d9	d8	d7 d6 d5 d4 d3 d2	d1 d0
0	0	0	0	Number of pixels	Pixel Data

4. If there is a strip of 64 to 255 pixels with the same value, specify '0' in the first 6 bits, and enter the number of pixels in the following 8 bits and the pixel data in the next 2 bits. The 16 bits are considered to be one unit.

d15	d14	d13	d12	d11	d10	d9 d8 d7 d6 d5 d4 d3 d2	d1 d0
0	0	0	0	0	0	Number of pixels	Pixel Data

5. However, if the same pixels follow to the end of the line, specify '0' in the first 14 bits and describe the pixel data in the following 2 bits. The 16 bits are considered to be one unit.

d15 d14 d13 d12 d11 d10 d9 d8 d7 d6 d5 d4 d3 d2	d1 d0
00000000000000	Pixel Data

Data coming from the SubD FIFO (part of the Video FIFO) will be unpacked upon entering the pipe. Hence, once again, the Sub-Picture should only be used to display Sub-Picture type pixels and nothing else.

As for the SubC channel, multiple control commands (48-bits per command) are concatenated together to form a package. The Sub-Picture control registers need to be written to instruct the hardware about the size and numbers of command.

The subPcount (Sub-Picture PXCTLI Count) gives the number of 48-bit command per line. The hardware will cycle through these commands for each line. Up to 8 commands may coexist on a line.

The PXCTLI control code operates on the pixels in the SubD channel with the following format:

b47	b46	b45	b44	b43	b42	b41	b40
reserved						Change Start Pixel number (upper bits)	

b39	b38	b37	b36	b35	b34	b33	b32
Change start pixel number (lower bits)							

b31	b30	b29	b28	b27	b26	b25	b24
New emphasis pixel-2 color code				New emphasis pixel-1 color code			

b23	b22	b21	b20	b19	b18	b17	b16
New pattern pixel color code				New background pixel color code			

b15 b14 b13 b12	b11 b10 b9 b8
New emphasis pixel-2 contrast	New emphasis pixel-1 contrast

b7 b6 b5 b4	b3 b2 b1 b0
New pattern pixel contrast	New background pixel contrast

It is **very important** to remember that even though the DMA treat the SubD and SubC as two separate channels, VDG will display pixels in the SubD channel based on the control codes in the SubC channel. Hence, they function as one unit in the VDG and any attempt to use them as separate channels is highly discourage.

Display Fetch Mechanism

The display fetch mechanism uses a 512 byte RAM as a FIFO, in the display generator. This is a simple pixel FIFO unless the display overlay is enabled, in which case the RAM is split into two equal FIFO blocks. During active video, the display generator will request DMA transfers whenever either FIFO has any space in it. This means that video will consume any spare bus bandwidth. When either FIFO runs dangerously low, the video takes priority on the bus, and will start fetching at the end of the current slot.

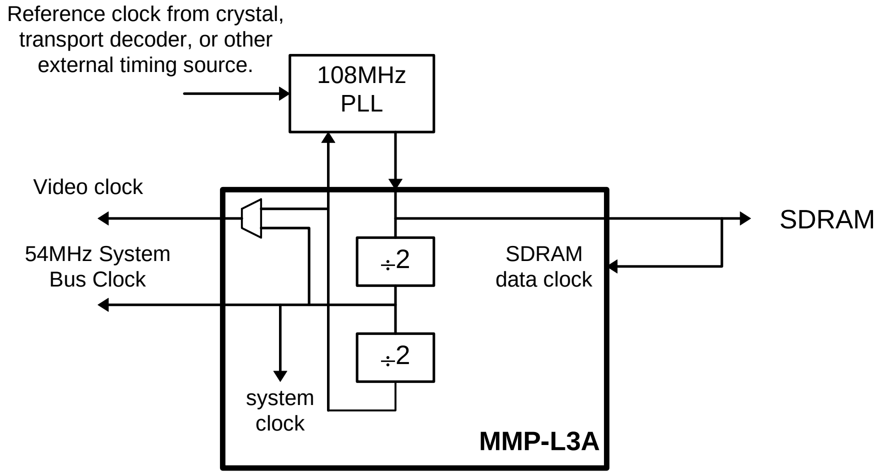
Video Interface Timing

NUON includes an internal video-timing generator. This timing generator can be free running as a master, or can be synchronized to an external Sync source.

In broadcast applications with MPEG2 transport streams, or when it is necessary to mix NUON graphics with analog video, external synchronization is necessary. This can be achieved by locking the master 108MHz NUON clock with the 27 MHz video master (extracted from the transport stream or the composite video signal), and bringing in external HSYNC and FIELD signals.

Video Clock Architecture and synchronization

The following block diagram shows the recommended main and video clock structure.



In this clock architecture, the 108MHz PLL can be locked:

- from a reference crystal, if no external timing reference is required.
- to the recovered 27MHz MPEG2 clock from an external transport stream decoder.
- to the recovered video clock timing from an analog composite video stream.

The MPEG2 clock is used if MPEG video is being decoded and displayed using an external transport stream decoder. Composite video can be used as a clock source if analog video is being displayed, with a requirement for a graphics overlay. The video syncs can be used as inputs to reset the internal video time-base, to achieve coarse locking.

The 108MHz PLL locks the timing reference signal with the VCLK signal, which is used as a master video clock by NUON, any external processors and decoders, and the digital video encoder and mixer.

NUON divides the 108MHz clock by 2, to obtain its internal master clock, and provides this to the System Bus as BCLK.

VIDEO INPUT

NUON supports a CCIR 656 video input stream, which is transferred over the Communication Bus for internal processing.

Video Capture

The incoming video stream is captured and transmitted over the Communication Bus. Software then transfers it into main memory via an ISR. There are two ways this can be performed:

1. Raw form, where the incoming pixel data is directly transferred over the Communication Bus. This is the direct CCIR 656 interleaved data stream including blanking, and the SAV and EAV data, and requires further processing to be used as display or texture data.
2. Pixel form, where the incoming 4:2:2 pixels may be transferred as-is, as a 32-bit pixel (of which 24 are valid), or as 16-bit data by truncation. When performing truncation, dithering may be enabled to prevent ‘banding’ artifacts.

Video capture is not required to be synchronous with the main NUON clock, and can therefore run at any desired clock rate, subject only to the limit of how fast the data can be passed over the Communication Bus.

To reduce bandwidth, the video input channel data may be reduced in resolution before sending over the Communication Bus. Each field can be transmitted at half (2:1) or quarter (4:1) resolution in addition to full resolution. This reduction can be controlled independently both horizontally and vertically and is performed by merely discarding pixels/lines. Filtering is possible in the horizontal direction if desired; adjacent pixels are averaged before sending. Vertically, the resolution may also be reduced by discarding alternate fields instead of lines.

Note that the worst-case throughput requirements are considerable and will tax both the Communication Bus and the receiving processor. The worst case condition, which is when the input stream is converted directly to 32-bit pixels, giving an active video transfer rate requirements of 13.5 x 4 = 54 Mbytes / sec. This figure only applies during active video, so the averaged transfer requirement is 41 Mbytes / sec. These numbers quantify the bandwidth occupied by the actual data. In addition, each Comm Bus packet also requires a fifth 32-bit long which holds the sender and receiver IDs and other status information. Thus the total worst-case Comm Bus bandwidth occupied by video input is 13.5 x 5 = 67.5 Mbytes / sec (55 Mbytes / sec average). Transfers at this rate imply receiving a Communication Bus packet on average every 16 clock cycles, well within the capability of the Communication Bus and an MPE, as long as no other significant use is being made of either at the same time.

Status Bits

In addition to the 16 bytes of data, each Communication Bus packet also returns several status bits. They are described in the table below.

An overflow occurs when a video input packet becomes ready for transmission while the current packet has yet to be sent. In this case the old packet is overwritten by the new one and is lost. Bits 0 and 1 should be ORed to determine overflow.

Bit	**Description**
7	If this is set, the FVH values below are valid
6	F 0 = odd field 1 = even field

5	V 0 = active line 1 = inactive line (vertical blanking)
4	H 0 = active video 1 = inactive video (horizontal blanking)
3	Set to 1 if the data in this packet is a result of a read from vinCtrl or vinRand
2	Unused (always 0)
1	Indicates an overflow occurred sometime since the last video input reset
0	Indicates an overflow just occurred during the transmission of this packet (it is possible for bit 1 to be set without bit 0 ever being set)

Bit Ordering

The table below shows how pixel data is ordered in the Communication Bus packets for the various data types. All of these examples assume an input stream as follows:

Time →

Cb_0 Y_0 Cr_0 Y_1 Cb_1 Y_2 Cr_1 Y_3 Cb_2 Y_4 Cr_2 Y_5 Cb_3 Y_6 Cr_3 Y_7 . . .

	Long 0				Long 1				Long 2				Long 3			
	31:24	23:16	15:8	7:0	31:24	23:16	15:8	7:0	31:24	23:16	15:8	7:0	31:24	23:16	15:8	7:0
RAW	Cb_0	Y_0	Cr_0	Y_1	Cb_1	Y_2	Cr_1	Y_3	Cb_2	Y_4	Cr_2	Y_5	Cb_3	Y_6	Cr_3	Y_7
4:2:2																
HScale 1:1	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Cr_0	Cr_1	Cr_2	Cr_3	Cb_0	Cb_1	Cb_2	Cb_3
HScale 2:1	Y_0	Y_2	Y_4	Y_6	Y_8	Y_{10}	Y_{12}	Y_{14}	Cr_0	Cr_2	Cr_4	Cr_6	Cb_0	Cb_2	Cb_4	Cb_6
HScale 4:1	Y_0	Y_4	Y_8	Y_{12}	Y_{16}	Y_{20}	Y_{24}	Y_{28}	Cr_0	Cr_4	Cr_8	Cr_{12}	Cb_0	Cb_4	Cb_8	Cb_{12}
32-bit																
HScale 1:1	Y_0	Cr_0	Cb_0	00	Y_1	Cr_0	Cb_0	00	Y_2	Cr_1	Cb_1	00	Y_3	Cr_1	Cb_1	00
HScale 2:1	Y_0	Cr_0	Cb_0	00	Y_2	Cr_1	Cb_1	00	Y_4	Cr_2	Cb_2	00	Y_6	Cr_3	Cb_3	00
HScale 4:1	Y_0	Cr_0	Cb_0	00	Y_4	Cr_2	Cb_2	00	Y_8	Cr_4	Cb_4	00	Y_{12}	Cr_6	Cb_6	00
16-bit*																
HScale 1:1	$Y_0Cr_0Cb_0$		$Y_1Cr_0Cb_0$		$Y_2Cr_1Cb_1$		$Y_3Cr_1Cb_1$		$Y_4Cr_2Cb_2$		$Y_5Cr_2Cb_2$		$Y_6Cr_3Cb_3$		$Y_7Cr_3Cb_3$	
HScale 2:1	$Y_0Cr_0Cb_0$		$Y_2Cr_1Cb_1$		$Y_4Cr_2Cb_2$		$Y_6Cr_3Cb_3$		$Y_8Cr_4Cb_4$		$Y_{10}Cr_5Cb_5$		$Y_{12}Cr_6Cb_6$		$Y_{14}Cr_7Cb_7$	
HScale 4:1	$Y_0Cr_0Cb_0$		$Y_4Cr_2Cb_2$		$Y_8Cr_4Cb_4$		$Y_{12}Cr_6Cb_6$		$Y_{16}Cr_8Cb_8$		$Y_{20}Cr_{10}Cb_{10}$		$Y_{24}Cr_{12}Cb_{12}$		$Y_{28}Cr_{14}Cb_{14}$	

* 16-bit data is packed with Y in bits 15-10, Cr in bits 9-5, and Cb in bits 4-0.

Video Input Control Register

The register described below controls the video input channel. This register is programmed over the Communication Bus by a single packet. The Communication Bus identification numbers are defined in the Communication Bus section of this document. A value of h00 in bits 31-24 of the packet's first long of data indicates a write into vinCtrl. The value to write is taken from bits 23-0 of that same long.

A command packet will initiate pixel transfer as specified in bits 18-16. It will wait for the start of a specified field before transmitting, except b100, which causes transfer to start immediately. However, this mode only makes sense when sending raw data (bits 2-0 are b000) as it has no context for discerning EAV, SAV, or other video attributes.

When video capture begins, the Communication Bus ID of the last sender that programmed vinCtrl determines the video input stream's destination.

Before changing vinCtrl, the Video Input module must first be reset. This is accomplished by writing a 1 into bit 20. This clears all bits in vinCtrl and places the module in a known reset state. A subsequent write to vinCtrl will initiate video capture as desired.

The current state of vinCtrl may also be read. Sending a Communication Bus packet with a value of h80 in bits 31-24 of the packet's first long of data indicates a read (all other bits are ignored). Care must be taken when reading vinCtrl during video capture. If the module is trying to send a video packet at the same time as vinCtrl is being read, one or the other may be lost.

vinCtrl Video Input Channel Control

```
Long word 0
Read / Write
```

This register controls the operation of the Video Input module.

Bit	Description
20	Reset video in
18-16	Data transfer mode: 000 Do nothing (video input off) 001 Send the next odd field 010 Send the next even field 011 Send the next field whatever it is 100 Continuously stream data starting immediately 101 Continuously stream data from the start of the next odd field 110 Continuously stream data from the start of the next even field 111 Continuously stream data from the start of the next field
15	Block sending of odd fields (F=0)
14	Block sending of even fields (F=1)
13-12	Data to capture: 00 Send all incoming data 01 Send the active portion of all lines (H=0) 11 Send the active portion of active lines (H=0 and V=0)
10-8	Horizontal scale mode: 000 Transfer 1:1 (send every pixel) 001 Transfer 2:1 (send every 2nd pixel) 010 Transfer 2:1 w/filtering (send average of 2 adjacent pixels) 011 Transfer 4:1 (send every 4th pixel) 100 Transfer 4:1 w/filtering (send average of 4 adjacent pixels)
6-4	Vertical scale mode: 000 Transfer 1:1 (send every line) 001 Transfer 2:1 (send every 2nd line) 011 Transfer 4:1 (send every 4th line)
2-0	Pixel transfer mode: 000 Send raw data 001 Send packed 4:2:2 data 010 Send 32-bit pixel data 011 Send 16-bit pixel data 100 Send 16-bit pixel data (dithered)

Unused bits in the command long word must be written as zero

vinRand Video Input Random Number Register

```
Read only
```

This register is used when in 16-bit dithering mode (bits 2-0 of vinCtrl are b100). It may be read by sending a Communication Bus packet to the Video Input module. A value of h81 in bits 31-24 of the packet's first long word initiates a read of vinRand (all other bits are ignored). The same caution applies here as with reading vinCtrl; namely that the returning data may clash with a video input packet such that one or the other is lost.

Bit	Description
31-0	Random number

AUDIO OUTPUT

The NUON audio output system supports up to ten discrete output channels. Eight of these are carried over four synchronous serial stereo data channels, which can be compatible with I^{2}S. The other pair is carried over an IEC 958 (S/P DIF) channel.

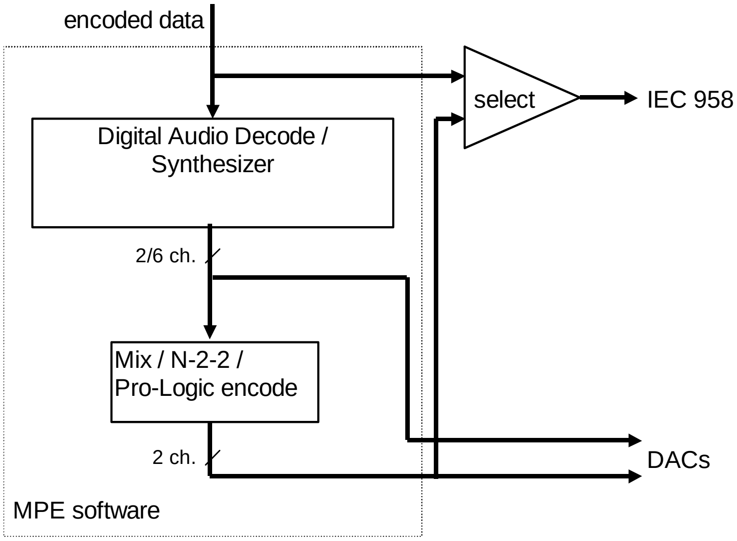
This capability provides support for Dolby Digital (AC-3), DTS, and some MPEG audio decoding, which can require as many as six output channels. These channels are left, center, right, left surround, right surround, and low frequency effects. This 6 channel arrangement is usually referred to as 5.1 channels as the low frequency effects channel is encoded with a lower bandwidth.

The additional channels allow simultaneous output of a mixed-down stereo pair, and a SPDIF output channel carrying either stereo audio or the encoded bitstream.

IEC 958 data is intended for connection to either an external digital audio decoder or a system with its own DACs. It can either output a stereo PCM audio pair with 16, 20 or 24 data bits; or the encoded digital audio data.

The synchronous serial port is intended for connection to on-board DACs, providing analog audio outputs from the system containing NUON. It can output stereo PCM, and six-channel surround PCM. Data may be either 16-bit or 32-bit, the latter containing any desired number of significant bits, limited only by the capability of the DACs.

The diagram below summarizes the intended audio data flow output capability:


encoded data
select
IEC 958
Digital Audio Decode / Synthesizer
2/6 ch.
Mix / N-2-2 / Pro-Logic encode
2 ch.
DACs
MPE software

The rectangular blocks shown in this diagram represent software functions.

Audio data may be written to these output devices either by writing directly to the audio output device using the Communication Bus, or by setting up audio output DMA.

Buffering and Real Time Requirements

The audio output channels are all shift-register-based channels. When they are being fed by audio DMA, a 320-byte FIFO RAM is used to overcome the effects of DMA latency. This RAM may be separated into two logical FIFOs for S/PDIF and DAC data if required (see below).

Data may also be fed to the audio output channels directly under program control using the Communication Bus. A buffer empty interrupt is generated whenever the shift registers are emptied. The program must respond to this interrupt, write to the audio data registers using the Communication Bus, and be ready to receive another interrupt within one sample time.

Summary of audio changes for Aries 3

The changes to the audio output for Aries 3 are significant.

- Add a DMA FIFO RAM so that audio DMA is able to stand longer DMA latencies without underflow. The FIFO RAM is 80 scalars. This should allow 8-channel, 32-bit output running at greater than 96 KHz.
- Add a fourth I2S stereo pair available on GPIO(1), supporting a total of eight I2S output channels, plus 2 S/PDIF channels. This allows a player to simultaneously output six channel audio and mixed down stereo from software, or eight decoded channels if required.
- Allow S/PDIF to run at a slower sample rate than I2S, with a separate DMA channel. This supports applications where the I2S sample rate is higher than the maximum supported S/PDIF sample rate.
- Allow odd clock divide ratios in the clock pre-scaler, so that we can support 384*fs* master clock frequencies. This is controlled by the **clkPrescaleOdd** bit in the **extCtrl** register described below.
- Support double buffering of S/PDIF channel status, reloading from the registers written to by software on a block boundary. This function also double buffers **raw32** and **rare32**.
- Allow the source of the S/PDIF valid flag, user data, and channel status to be individually selected as coming from either the stream or registers. The three **rare32** bits in **extCtrl** are used instead of the **raw32** bit for this.
- The S/PDIF block counter can be reset by a bit in the data stream. See the **blockAlignEna** bit in the **extCtrl** register.

Audio DMA

Audio DMA copies data stored in a DRAM buffer to the audio output devices. The DRAM buffer is a circular buffer of programmable size that contains interleaved audio output data.

The interleaved audio output data has the restriction that the total amount of data for each sample must be a power of two in bytes. This means the useful combinations it can transfer are:

- Two 16-bit output streams to both the IEC 958 and the synchronous serial stereo outputs.
- Four 16-bit output streams to support separate data streams to the IEC 958 and the synchronous serial stereo outputs.
- Eight 16-bit output streams to support separate data streams to the IEC 958 and the synchronous serial six-channel surround outputs.
- Two 32-bit output streams to both the IEC 958 outputs and the synchronous serial stereo outputs (the less significant bits of the data are truncated as appropriate).
- Four 32-bit output streams to support separate data streams to the IEC 958 and the synchronous serial stereo outputs.

- Eight 32-bit output streams to support separate data streams to the IEC 958 and the synchronous serial six-channel surround outputs, or can be used for eight synchronous serial stereo outputs.

If other output combinations are required then you can either transfer the data under program control using the Communication Bus, or you can mute unused output channels with dummy data interleaved into the output DMA buffer.

The circular buffer may be between 1K and 64K bytes. The 64K buffer size will hold enough data for two PAL video frames of audio at 48KHz, with eight 32-bit streams, i.e. a double buffer which is refilled once per frame as required. This is considered to be the largest possible requirement. The buffer pointer may be read, and interrupts generated when it wraps and when it crosses the midpoint, so that it can be treated as a double buffer.

You can send the audio interface a command to force the DMA to repeat a fetch address, or to skip a fetch address. This may be useful for re-synchronizing audio to video by skipping or repeating samples, although several samples will be re-used in some modes.

You can also freeze the buffer pointer, so that the same address is repeatedly read. This lets you output a constant value.

The DMA base address and fetch pointers should only be written when DMA is disabled. The DMA fetch pointer may be polled at any time to determine from where the next data will be read.

DMA Data Organization

DMA data is organized in one of the following repeating structures, depending on the output mode. Note that for the 32-bit forms the data should be aligned against the MSB, i.e. the 12 LSBs are not used for 20-bit samples, and the 8 LSBs are not used for 24-bit samples.

Two 16-bit channels

Bits	Word	Function
31-16	0	Left channel data / IEC958 channel one / left mixed-down channel data
15-0	1	Right channel data / IEC958 channel two / right mixed-down channel data

Four 16-bit channels

Bits	Word	Function
63-48	0	Left channel data / left mixed-down channel data
47-32	1	Right channel data / right mixed-down channel data
31-16	2	IEC958 channel one
15-0	3	IEC958 channel two

Eight 16-bit channels

Bits	Word	Function
127-112	0	Left main channel data
111-96	1	Right main channel data
95-80	2	Left surround channel data
79-64	3	Right surround channel data
63-48	4	Center channel data
47-32	5	Low frequency effects channel data
31-16	6	IEC958 channel one / left mixed-down channel data
15-0	7	IEC958 channel two / right mixed-down channel data

Two 32-bit channels

Bits	Long	Function
63-32	0	Left channel data / IEC958 channel one / left mixed-down channel data
31-0	1	Right channel data / IEC958 channel two / right mixed-down channel data

Four 32-bit channels

Bits	Long	Function
127-96	0	Left channel data / left mixed-down channel data
95-64	1	Right channel data / right mixed-down channel data
63-32	2	IEC958 channel one
31-0	3	IEC958 channel two

Eight 32-bit channels

Bits	Long	Function
255-224	0	Left main channel data
223-192	1	Right main channel data
191-160	2	Left surround channel data
159-128	3	Right surround channel data
127-96	4	Center channel data
95-64	5	Low frequency effects channel data
63-32	6	IEC958 channel one / left mixed-down channel data
31-0	7	IEC958 channel two / right mixed-down channel data

Data Organization for separate IEC958 DMA

Note that if separate IEC958 DMA is enabled, then IEC958 data is not extracted from the main DMA channel as shown above. In this case the channel is always 32-bit, thus:

Two 32-bit channels

Bits	Long	Function
63-32	0	IEC958 channel one
31-0	1	IEC958 channel two

Using Interrupts with a separate IEC958 DMA channel

When using a single DMA channel, the audio buffer in Main Bus DRAM may be maintained using the interrupts that occur when the DMA read pointer passes the halfway and end points in the buffer. The same mechanism can be used with a separate IEC958 DMA buffer, but certain rules must be followed.

1. No extra interrupts are generated, so you must program the sizes of the two buffers so that each loops at the same rate. For example, if you set up the main buffer for eight 32-bit channels, and the IEC 958 channel for two 32-bit channels, then you must set the main buffer size to four times the size of the IEC 958 buffer.
2. You must set the **laterCountWait** bit in the **dmaCtrl**. This ensures that the interrupt does not actually occur until both the buffer pointers have passed the halfway or end points.

Audio DMA FIFO Control

The FIFO sits between the Main Bus DMA interface logic and the data output logic.

Normally you should program the FIFO to be as large as possible. When separate IEC 958 DMA is enabled, the FIFO should be split in a ratio corresponding to the rates at which each channel is consuming data.

Underflow on start-up

If the FIFO underflows immediately on start-up, it may be necessary to enable audio DMA *before* starting up the output stage. Normally when **dmaEnable** is set this starts both the DMA read process which writes into the FIFO, and the transfer process which reads from the FIFO and writes into the

ssData registers. However, at higher sample rates, the FIFO may underflow before the first data has been read.

In these circumstances, the **dmaSepEna** bit in **extCtrl** should be set. The **dmaEnable** (and **dma958** if required) should be set, followed by a pause to allow the FIFO to fill up. About 2,000 clock cycles would be a conservative delay. After this time, or later, **dmaFifoRd** in **extCtrl** should be set to start up the reading of samples from the FIFO.

Synchronous Serial Audio Output Channel (I^2S)

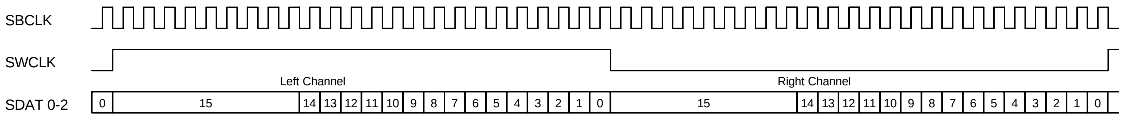
The synchronous serial bit clock, SBCLK, which is derived from an external audio clock as described below, controls audio output timing. One audio sample is then output every 16, 24 or 32 SBCLK cycles, as shown below. The word clock, SWCLK, whose polarity is programmable, gives the framing of the data.

Data is output on three data pins:

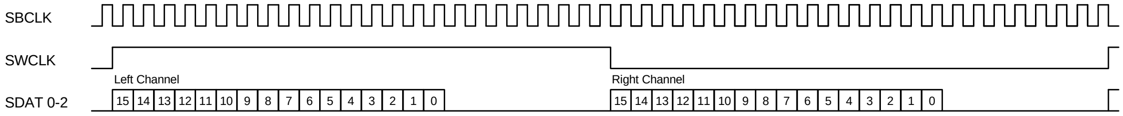
Pin	Left function	Right function
SDAT 0	Left	Right
SDAT 1	Left surround	Right surround
SDAT 2	Center	Low frequency effects

Possible data output modes for 16-bit data are as follows:

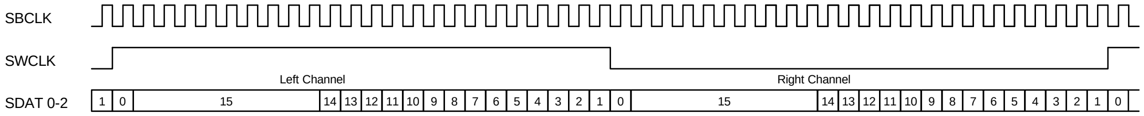
24-bit or 32-bit sample period, left = high SWCLK, right data alignment (24-bit shown):



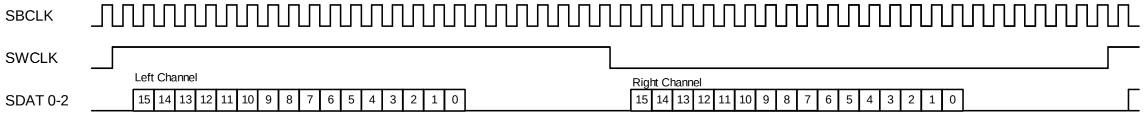
24-bit or 32-bit sample period, left = high SWCLK, left data alignment (24-bit shown):



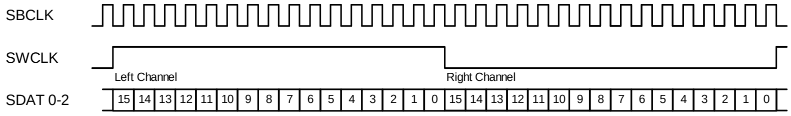
24-bit or 32-bit sample period, left = high SWCLK, right data alignment, delayed data (24-bit shown):



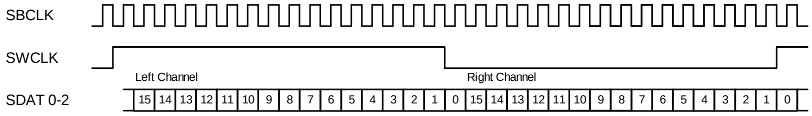
24-bit or 32-bit sample period, left = high SWCLK, left data alignment, delayed data (24-bit shown):



16-bit sample period, left = high SWCLK, either data alignment:

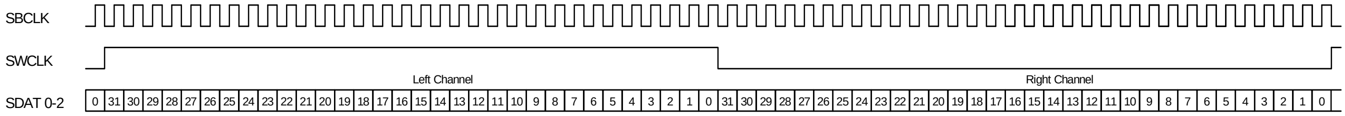


16-bit sample period, left = high SWCLK, either data alignment, delayed data:

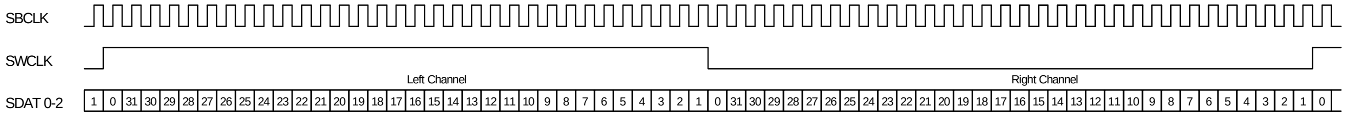


Possible data output modes for 32-bit data are as follows.

32-bit sample period, left = high SWCLK:



32-bit sample period, left = high SWCLK, delayed data:



Should the data ordering need to be reversed, this operation can be performed in software. The MPE mirror instruction is suited to this task.

IEC 958 Audio Output Channel

The IEC 958 audio output channel is a single wire serial, output-only, self-clocking interface. Refer to the IEC 958 standard documentation for further details. This interface is sometimes also known as S/P DIF (Sony/Philips Digital Interface Format).

Unless the interface is operated in the ‘raw’ mode, described below, it is limited to operating at consumer level, and not at professional level, because the channel status word is restricted to the first 32-bits.

This output channel can operate in two modes:

- 16-bit mode, where the interface is provided with 16-bit audio values. The validity flag is programmable, and the user data field is fixed at zero, for every sub-frame; and the first 32 bits of channel status for both channels of each block are programmable with the remaining bits being zero.
- 24-bit mode, where the interface is provided with 32-bit audio values whose eight LSBs are ignored. The validity flag is programmable, and the user data field is fixed at zero, for every sub-frame; and the first 32 bits of channel status for both channels of each block are programmable with the remaining bits being zero.
- 32-bit ‘raw’ mode, where the interface is provided with 32-bit values corresponding to complete sub-frames. The bits corresponding to the Sync Preamble and Parity are ignored as they are generated by the hardware, but all other fields are programmable.

The output channel hardware formats the data according to the IEC 958 standard, and contains a block counter so that it can correctly generate preambles. This counter can be reset.

When the IEC958 channel is enabled, the **`preScale`** value in the **`ssCtrl`** register should be set to give a clock 128 times the sample rate. The SBCLK for the synchronous serial output should be set to a half or a quarter of this rate with the **`bitScale`** value in the **`ssCtrl`** register. This rules out the use of 24-bit long sample period on the synchronous serial interface at the same time as IEC 958 output, although 24-bit sample can still be output in 32-bit mode.

Audio Clocking

The clock rate, and therefore the sample rate, is derived from an audio clock input pin. This is divided to produce:

- SBCLK, the synchronous serial bit clock. This is 64, 48 or 32 times the sample rate.
- The IEC 958 output clock. This requires two edge positions per bit, and contains 64 bits per sample; therefore it requires a clock that is 128 times faster than the sample rate.

The interface usually has to support audio sample rates of at least 48 KHz and 44.1 KHz, with some applications requiring other related rates. Some applications may not require a 44.1 KHz sample rate, which will simplify the clocking requirements.

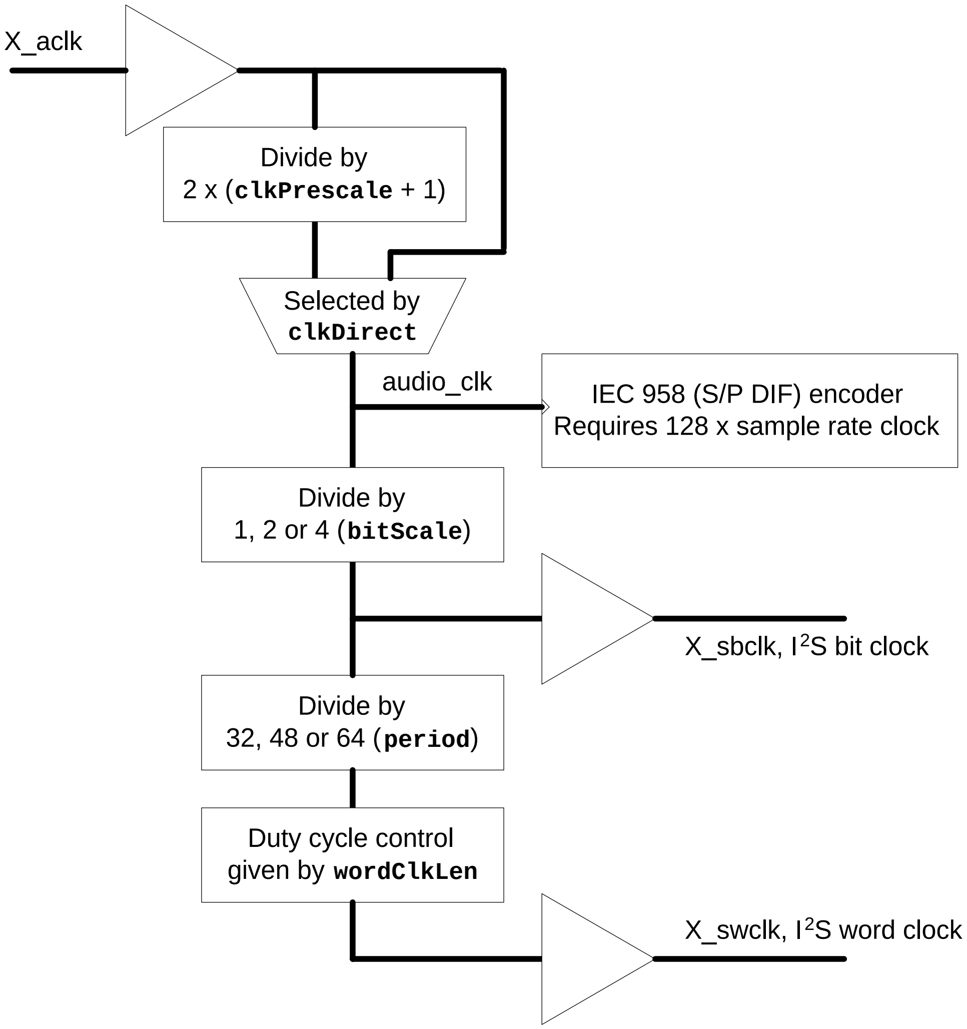
Commonly an external DAC or ADC has a fast clock requirement (often 256 times the sample rate), so the oscillator should be run at that speed, and the pre-scaler used to give the required internal clocks.

As an example, from a 256 times sample-rate clock we can support an IEC958 output, and SBCLK at 64 or 32 times the sample rate, so the oscillator requirements are:

256 x 48K = 12.288 MHz or
256 x 44.1K = 11.2896 MHz

Other applications may require still faster master clocks.

This diagram shows how the audio output clocks are generated:



Note the flexibility this gives, but also the constraints imposed by the IEC 958 output. The control values shown are programmed in the “Synchronous serial control register” shown below.

Audio Output Control Registers

The registers described below control the audio output channels. These registers are programmed over the Communication Bus. The audio output channel is Communication Bus ID 67 ($43). The communication protocol is as follows for the command packet:

Long word	Description
0	0-15 register address 31 set for write, clear for read
1	0-31 write data if a write command
2	unused
3	unused

The response packet is returned if the operation was a read. Its format is:

Long word	Description
0	Bits Function 0-1 0 for a read response 1 for audio input ch. 1 data 2 for audio input ch. 2 data 16-31 register address
1	0-31 read data
2	unused
3	unused

Note that audio input data packets can also be sent out, and if this function is enabled and targeted to the same MPE that is reading a register, then the controlling software will have to differentiate the two received packet types. See below under “Audio Inputs” for further details of the audio input data packet format.

The control registers are:

ssCtrl Synchronous serial control register

```
$0000
Read / Write
```

Bit	Name	Description
31-30	**period**	Sample period in SBCLK cycles 0 64 bit 1 48 bit 2 32 bit 3 unused
29	**wordPolarity**	Set for left = SWCLK high, clear for right = SWCLK high
28	**dataAlign**	Set for left aligned data, clear for right aligned data
27	**dataDelay**	Set to delay data output one SBCLK clock cycle relative to SWCLK, clear for no delay
26		Unused, write zero.
23	**lrMute**	Mute left and right data (force data output to zero). This defaults to set on reset.
22	**suMute**	Mute surround data. This defaults to set on reset.
21	**clMute**	Mute center and low frequency data. This defaults to set on reset.
20-14	**clkPrescale**	This value should be one less than the half clock period of the IEC958 clock (128x the sample rate) in ACLK cycles, i.e. the divide ratio is 2*(**clkPrescale**+1). Therefore, a value of 0 gives a divide by 2, and a value of 127 ($7F) gives a divide by 256. If the IEC958 output is not used, this may be programmed to give SBCLK directly. See **bitScale** below. It can also be by-passed, see **clkDirect** below,
13	**clkPolarity**	Set for output data and word clock changing on a falling edge of SBCLK, clear to have them changing on a rising edge.
12	**clkDirect**	Set for the ACLK input to be used directly as the audio output clock. **clkPrescale** is ignored if this is set.
11-10	**wordClkLen**	Allows some alternative word clock duty cycles, with the high period controlled as follows: 0 the first half of the sample period (normal) 1 the first bit clock of the sample period 2 the first two bit clocks of the sample period

Bit	Name	Description
		3 the first three bit clocks of the sample period Note that if wordPolarity is clear, then this controls the length of the low period of the signal.
9-8	**bitScale**	This gives the SBCLK rate from the output of the **clkPrescale** divider above. When the IEC958 output is enabled this will have to be set to divide by 2 if period is 64 bits, and divide by 4 if period is 32 bits. 0 divide by 1 (no scaling) 1 divide by 2 2 divide by 4
7	**aCntReset**	When this bit is set, the word framing counters are all set to zero and held until this bit is released. This is for special circumstances only.
6	**pCntReset**	When this bit is set, the audio clock pre-scale counter is set to zero and held until this bit is released. This is for special circumstances only.
5	**sCntReset**	When this bit is set, the IEC958 counters are all set to zero and held until this bit is released. This is for special circumstances only.
4-0		Unused, write zero.

extCtrl Extended synchronous serial control register

```
$0001
Write Only
```

Bit	Name	Description
31	**iecMute**	Mute IEC958 data channel to all zeroes. This defaults to zero (not muted) on reset for compatibility with Aries 1 & 2. *This function is available in Aries 3 or higher only.*
30	**mdMute**	Mute the mixed-down stereo I2S data channel. This defaults to one (muted) on reset. *This function is available in Aries 3 or higher only.*
29	**clkPrescaleOdd**	This modifies the behavior of the clock pre-scaler controlled by the **clkPrescale** value. When this bit is set, the clock high time is extended by one clock over the clock low time, so that the high time is clkPrescale+2 clock cycles, and the low time is clkPrescale+1 clock cycles. *This function is available in Aries 3 or higher only.*
28	**chstPipeline**	This makes the S/PDIF channel status double-buffered, so the output values are only re-loaded from the values written to by software on a block boundary. If this bit is clear the values written to by software are used immediately on a change, as Aries2 and below. This function also double buffers **raw32** and **rare32**. *This function is available in Aries 3 or higher only.*
27-25	**rare32**	These three bits allow a finer level of control over the **raw32** function in **iecCtrl**. They allow the source of the valid bit, the user data, and the channel status to be individually controlled. The **data32** flag must be set, and the **raw32** flag must be clear for these to operate, in which case they works as follows: 25 when set, the valid flag comes from the sample data stream, when clear it comes from the **valid** flag in **iecCtrl**. 26 when set, the user data field comes from the sample data stream, when clear it is set to zero. 27 when set, the channel status comes from the sample data stream, when clear it comes from the software programmed channel status registers.

Bit	Name	Description
24	**dmaSepEna**	This allows the read channel from the FIFO to be separately enabled from the write half. When clear, both DMA itself, and FIFO reads are enabled by **dmaEnable**. When set, **dmaFifoRd** needs to be set before FIFO reads will start. This allows DMA to be started up to fill the FIFO before output starts. This mechanism should not be necessary, use the **preventFifoUnderflow** flag in **fifoCtrl5** instead.
23	**dmaFifoRd**	When DMA and FIFO reads are separately enabled by the **dmaSepEna**, then this has to be set before data will be read from the FIFO into the **ssData** registers. **dmaEnable** (and **dma958** if required) have to be set before this.
22	**blockAlignEna**	When this bit is set, and the IEC 958 output channel is running in 32-bit mode (**data32** set), then a one in bit position zero of the data for sub-frame zero (channel 1) will reset the block counter aligned to this sample. This function is not compatible with either **leftAlign32** or **rightAlign32**.

iecCtrl IEC 958 control register

```
$0008
Write only
```

This register controls the IEC 958 interface.

Bit	Name	Description
31	**bCntRst**	Block counter reset. When a one is written to this bit, the block counter will be set to zero when the next value is written to IEC_DATA2 or when the next DMA transfer occurs, so the next pair of values written will be the start of a block. Writing a zero has no effect.
30	**enable958**	Must be set for data transfer to occur. When this bit is clear zero is output on the data channel.
29	**data32**	Set for 32-bit mode – this allows sample values of up to 24 bits to be output, and in conjunction with **raw32** allows the valid flag, user data channel, and channel status bits to be set by applications. Valid bits in the long-word correspond to their positions in the sub-frames, i.e. bits 4-27. and the other data is ignored. The leftAlign32 and rightAlign32 bits below allow other data positions. When this is clear, the sixteen bit samples have 8 zeroes added to the LSBs to give the 24-bit output words.
28	**raw32**	Should be set only when **data32** is set. Allows user programming of the control bits. When this is clear, the user data channel outputs zeroes, and the channel status control word is output every block followed by zeroes.
27	**valid**	Validity flag when **raw32** is clear. Defaults to set (not valid) on reset. This will take effect from the next sub-frame.
26	**leftAlign32**	If this bit is set, in conjunction with data32 but not raw32, then the 24 data bits are considered left aligned in the long-word in memory, i.e. the MSB is in bit 31. *This function is available in Aries 2 or higher only.*
25	**rightAlign32**	If this bit is set, in conjunction with data32 but not raw32, then the 24 data bits are considered right aligned in the long-word in memory, i.e. the LSB is in bit 0. *This function is available in Aries 2 or higher only.*
24-23	**slow958**	These bits allow the IEC 958 output channel to be operated more slowly than its default rate of one sample every 128 **audio_clk** cycles. This is normally used in conjunction with the **dma958** function.

		00 128 clocks per frame, as Aries 1-2 01 256 clocks per frame 10 384 clocks per frame 11 512 clocks per frame *This function is available in Aries 3 or higher only.*
2-0	**subdIntVal**	This is a mechanism for generating interrupts 2^n times slower than the buffer refill interrupt (the one enabled by dmaSampInt), where n is between 1 and 7. A value of zero disables it. A value of 1 causes an interrupt every second buffer refill, 2 every fourth, 3 every eighth up to 7 every 128 buffer refills. Non zero in these three bits is enough to enable the interrupt.

dmaCtrl DMA control

```
$002C
Read / Write
```

This register controls the audio output DMA channel.

Bit	Name	Description
20	**laterCountWait**	When this bit is set, the **dmaWrapInt** and **dmaHalfInt** interrupts will be delayed until both the buffer pointers have met the condition. If this bit is clear, then the interrupts are generated off only the main DMA pointer.
19	**dma958**	Enables separate DMA channel for S/PDIF data.
18-16	**dmaSBuffer**	Circular buffer size for optional separate S/PDIF audio DMA 0 Do not advance from base address 1 1K / $400 bytes 2 2K / $800 bytes 3 4K / $1000 bytes 4 8K / $2000 bytes 5 16K / $4000 bytes 6 32K / $8000 bytes 7 64K / $10000 bytes.
13	**dmaMode**	MSB of **dmaMode**, see below for encoding.
12	**dmaStall**	Sample stall. When this bit is set, the fetch pointer will not advance after the next fetch. This bit is cleared when the stall has occurred by the hardware, and another stall command can be issued. This may not be set at the same time as the skip bit below.
11	**dmaSkip**	Sample skip. When this bit is set, the next fetch pointer advance is two samples. When skip occurs the hardware will clear this bit and another skip command can be issued. This may not be set at the same time as the stall bit above.
10	**dmaSampInt**	Enable DMA buffer empty interrupt. This may be used if audio out is being driven by Communication Bus data instead of DMA. It can be treated as a request to write new data to the audio output data registers.
9	**dmaHalfInt**	Enable audio interrupt when buffer pointer passes half the buffer size.
8	**dmaWrapInt**	Enable audio interrupt when buffer pointer wraps. Note that if you enable this interrupt then an ‘extra’ one will occur whenever the audio buffer base address register is written (if DMA is not enabled it will occur when DMA is first enabled after a buffer base address register write).
7-5	**dmaMBuffer**	Circular buffer size for main audio DMA 0 Do not advance from base address 1 1K / $400 bytes 2 2K / $800 bytes

		3 4K / $1000 bytes 4 8K / $2000 bytes 5 16K / $4000 bytes 6 32K / $8000 bytes 7 64K / $10000 bytes.
13,4-3	**dmaMode**	DMA mode, as follows (see above for details): 0 two 16-bit output streams 1 four 16-bit output streams 2 two 32-bit output streams 3 eight 16-bit output streams 4 *reserved for future use* 5 *reserved for future use* 6 eight 32-bit output streams. 7 four 32-bit output streams
2-1	**dmaPriority**	Audio DMA bus priority. Values of 0-3 are valid, 0 implies priority 4 and should only be used in extreme circumstances.
0	**dmaEnable**	Enable audio output DMA.

iecChst0 IEC 958 Channel status control word 0

```
$000C
Write only
```

IEC 958 Channel status control word for normal (not raw) mode. This is output as the first 32 bits of the channel status on sub frame 0 every block, with the least significant bit first. The remaining bits are forced to zero.

iecChst1 IEC 958 Channel status control word 1

```
$0010
Write only
```

IEC 958 Channel status control word for normal (not raw) mode. This is output as the first 32 bits of the channel status on sub frame 1 every block, with the least significant bit first. The remaining bits are forced to zero.

ssData0 Audio output data long-word 0

```
$0018
Write Only
```

This register allows audio output data to be written using the Communication Bus instead of DMA. The entire audio buffer must always be written whatever the output mode.

ssData1 Audio output data long-word 1

```
$001C
Write Only
```

This register allows audio output data to be written using the Communication Bus instead of DMA. The entire audio buffer must always be written whatever the output mode.

ssData2 Audio output data long-word 2

```
$0020
Write Only
```

This register allows audio output data to be written using the Communication Bus instead of DMA. The entire audio buffer must always be written whatever the output mode.

ssData3 Audio output data long-word 3

```
$0024
Write Only
```

This register allows audio output data to be written using the Communication Bus instead of DMA. The entire audio buffer must always be written whatever the output mode.

ssData4 Audio output data long-word 4

```
$0030
Write Only
```

This register allows audio output data to be written using the Communication Bus instead of DMA. The entire audio buffer must always be written whatever the output mode.

ssData5 Audio output data long-word 5

```
$0034
Write Only
```

This register allows audio output data to be written using the Communication Bus instead of DMA. The entire audio buffer must always be written whatever the output mode.

ssData6 Audio output data long-word 6

```
$0038
Write Only
```

This register allows audio output data to be written using the Communication Bus instead of DMA. The entire audio buffer must always be written whatever the output mode.

ssData7 Audio output data long-word 7

```
$003C
Write Only
```

This register allows audio output data to be written using the Communication Bus instead of DMA. The entire audio buffer must always be written whatever the output mode.

dmaMBase DMA buffer base address for main audio DMA

```
$0028
Write Only
```

DMA base address for main audio DMA transfers. This is always used when audio DMA is enabled, and either points at the buffer for all audio data; or if separate SPDIF DMA is enabled, then this points at the I2S data.

This pointer must lie on a boundary aligned to the current buffer size, i.e. a 4K buffer should lie on a 4K boundary. Note that buffer size 0 must lie on a 1K ($400) byte boundary.

dmaSBase DMA buffer base address for SPDIF audio DMA

```
$0029
Write Only
```

DMA base address for optional separate SPDIF audio DMA transfers. This must lie on a boundary corresponding to the current buffer size. Buffer size 0 must lie on a 1K ($400) byte boundary.

This pointer must lie on a boundary aligned to the current buffer size, i.e. a 4K buffer should lie on a 4K boundary. Note that buffer size 0 must lie on a 1K ($400) byte boundary.

dmaMPointer DMA main channel fetch pointer

```
$0030
Read Only
```

DMA main channel fetch pointer. This is the offset address from the base pointer where the next DMA data will be read.

dmaSPointer DMA SPDIF channel fetch pointer

```
$0031
Read Only
```

DMA SPDIF channel fetch pointer. This is the offset address from the base pointer where the next DMA data will be read.

ssData8 Audio output data long-word 8

```
$0040
Write Only
```

This register allows audio output data to be written using the Communication Bus instead of DMA. This register is only used when dma958 is set.

ssData9 Audio output data long-word 9

```
$0044
Write Only
```

This register allows audio output data to be written using the Communication Bus instead of DMA. This register is only used when dma958 is set.

fifoCtrl0 Audio DMA FIFO Control Register 0

```
$0080
Read / Write
```

This register selects the FIFO size for the main and S/PDIF FIFO channels, and the maximum DMA size.

Bit	Name	Description
22-16	**fifoMaxDMA**	Defines the largest DMA transfer that can be requested by the FIFO, in

		scalars. The default value is eight.
6-0	**fifoBreak**	Defines the first FIFO location used for the S/PDIF FIFO. All scalar addresses starting from location zero to one less than this are used for the main FIFO, all scalars from this address to the top. The FIFO is $50 scalars long, and the default value for this register is $08.

fifoCtrl1 Audio DMA FIFO Control Register 1

```
$0081
Read / Write
```

This register selects the minimum DMA size for the main FIFO channels, and level below which the main FIFO is considered dangerously low and takes priority for DMA.

Bit	Name	Description
31-23	**unused**	Write zeroes.
22-16	**fifoMMinDMA**	Defines the smallest DMA transfer that can be requested by the FIFO for the main DMA channel, in scalars. The default value is eight scalars.
15-7	**unused**	Write zeroes.
6-0	**fifoMLow**	Defines the level in scalars at which the main FIFO is considered dangerously low. Normally the two DMA channels will alternate for memory access, when one is below its low threshold and the other is not, then the low one will take priority. The default value is eight scalars.

fifoCtrl2 Audio DMA FIFO Control Register 2

```
$0082
Read / Write
```

This register selects the minimum DMA size for the S/PDIF FIFO channels, and level below which the S/PDIF FIFO is considered dangerously low and takes priority for DMA.

Bit	Name	Description
31-23	**unused**	Write zeroes.
22-16	**fifoSMinDMA**	Defines the smallest DMA transfer that can be requested by the FIFO for the S/PDIF DMA channel, in scalars. The default value is eight scalars.
15-7	**unused**	Write zeroes.
6-0	**fifoSLow**	Defines the level in scalars at which the S/PDIF FIFO is considered dangerously low. Normally the two DMA channels will alternate for memory access, when one is below its low threshold and the other is not, then the low one will take priority. The default value is eight scalars.

fifoCtrl3 Audio DMA FIFO Control Register 3

```
$0083
Read / Write
```

This register allows the FIFO pointers for the main DMA channel FIFO to be read from and written to. If audio DMA is stopped and restarted then these should be cleared to zero. If these pointers are equal then the FIFO is empty.

Bit	Name	Description
31-23	**unused**	Write zeroes.
22-16	**fifoMRptr**	This pointer points to the first unread scalar location in the main FIFO.
15-7	**unused**	Write zeroes.
6-0	**fifoMWptr**	This pointer points to the next scalar location to be written in the FIFO.

fifoCtrl4 Audio DMA FIFO Control Register 4

```
$0084
Read / Write
```

This register allows the FIFO pointers for the S/PDIF DMA channel FIFO to be read from and written to. If S/PDIF audio DMA is stopped and restarted then these should be set the the value written into **fifoBreak**. If these pointers are equal then the FIFO is empty.

Bit	Name	Description
31-23	**unused**	Write zeroes.
22-16	**fifoSRptr**	This pointer points to the first unread scalar location in the S/PDIF FIFO.
15-7	**unused**	Write zeroes.
6-0	**fifoSWptr**	This pointer points to the next scalar location to be written in the S/PDIF FIFO.

fifoCtrl5 Audio DMA FIFO Control Register 5

```
$0085
Read / Write
```

This register allows the detection or prevention of FIFO underflow. It is not expected that the FIFO will underflow in normal operation if programmed correctly, but should this occur underflow will be flagged here.

It is conceivable that the FIFO could underflow on start-up, as it is initially empty and has less then one sample time to fetch the correct data. If this occurs then the **preventFifoUnderflow** flag should be set to avoid this causing problems such as clicks in the output channel.

Bit	Name	Description
31	**Munderflow**	Flags that the main DMA channel FIFO has underflowed.
30	**Sunderflow**	Flags that the S/PDIF DMA channel FIFO has underflowed.
29	**underflowClr**	Writing a 1 to this bit will clear the underflow flags. Writing a zero has no effect, and this bit is always read as zero.
28	**preventFifoUnderflow**	When this bit is set, the FIFO will not underflow. Instead, transfer from the FIFO to the **ssData** holding registers is disabled if there is less then eight scalars in the main FIFO at the point data is requested. Similarly, SPDIF DMA FIFO reads are disabled if less then 2 scalars exist in its FIFO. When this occurs the previous sample(s) are repeated.
28-0	**unused**	Write zeroes.

fifoRAM Audio DMA FIFO Direct Access

```
$1000 - $104F
Read / Write
```

This register block allows the FIFO to be directly written to and read from. This is provided for diagnostic purposes.

AUDIO INPUTS

NUON supports two stereo audio input channels (referred to as channels 1 and 2) over two independent synchronous serial interfaces, which can be run in a mode compatible with I^{2}S. These interfaces are not connected to the audio output channel, so all three can be run at different sample rates if required.

Pins associated with each channel are:

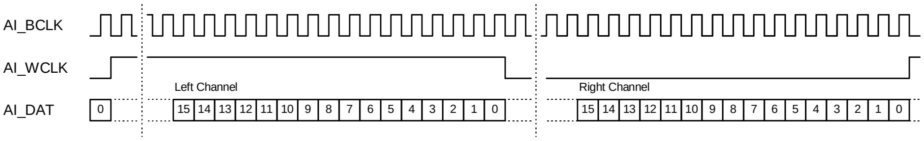
1. The bit clock: data and word clock are sampled on either a rising or falling edge of this.
2. The word clock: this is used to frame the data in a variety of programmable ways.
3. The data input: the serial input stream, always MSB first, may contain pad bits.
4. Over-sample clock: where a higher clock rate is need from the internal timing (channel 1 only).

There is no DMA associated with this channel, as it operates entirely over the Communication Bus. When an audio input channel is enabled it is programmed with the Communication Bus address of the processor which will receive the audio data. The audio input controller will then transfer a pair of samples every time they are received. The receiving processor must be able to receive sample pairs over the Communication Bus at sample rate, which should be straightforward.

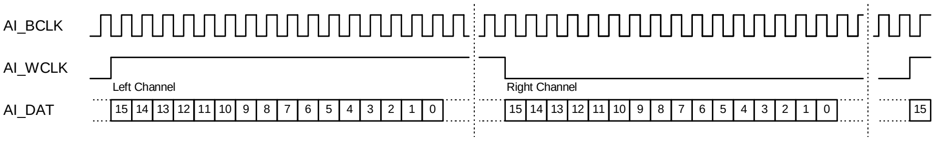
Each receiver port supports much the same set of serial data protocols as the transmitter, with some greater flexibility. Left data alignment means that the receiver uses the first 16 bits that follow an edge on word clock; right data alignment means that the receiver uses the 16 bits that precede an edge on word clock. This means that the receiver does not care how long the period is between edges on word clock.

The receiver can also be programmed to accept data framing where the start or end of the data word is some number of AI_BCLK cycles after the edge on AI_WCLK. This is referred to as delayed data mode.

Left = high AI_WCLK, right data alignment:



Left = high AI_WCLK, left data alignment:



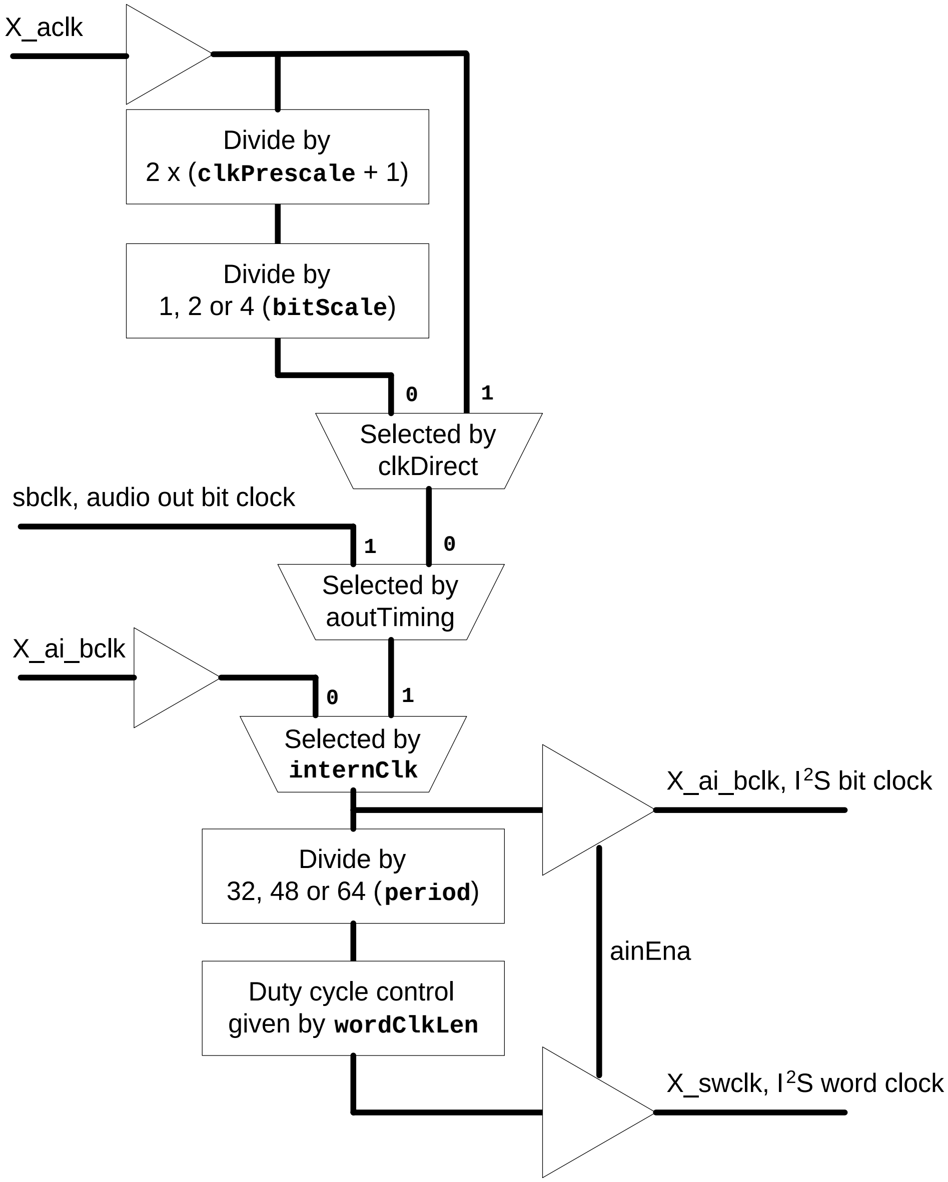
Delayed data relative to AI_WCLK cases are not illustrated here.

The over-sample clock is used when timing is being generated from ACLK, and it gives the output the clock pre-scaler. It is only available from Audio Input Channel 1, so these channels should be set to the same pre-scale value if both require this function. Its phase relationship to the bit clock of channel 2 is not defined.

Audio Input Timing

An audio input channel can be configured as a timing master or slave. It defaults to being a slave, where its timing is derived from an external source, however, it may also be a timing master. As a master, it generates the bit and word clocks, and can also generate a higher frequency over-sample clock, used by some ADCs and Codecs. This is available on one of the GPIO pins.

The audio input clocking is generated thus:



Data Capture

Data capture is independent of the clocking scheme outlined above, and is triggered by edges on the word clock. There are two operating modes, controlled by the **ssyncMode** bit:

0 Data is captured **`dataDelay`** clock cycles after a transition on the word clock. Capture means that the contents of a free running shift register are transferred to the capture register for left or right data, depending on the polarity of the edge. So a delay of zero means that the 16-bits previous to the edge are captured.

1 Left data is captured **`dataDelay`** after the rising edge (or falling edge if **`wordPolarity`** is clear) of the word clock, and then the right data is captured (**`dataDelay2 + 2)`** cycles after that. This allows data to be captured from a stream where the word clock is a pulse, as opposed to an even duty cycle.

Audio Input Channel 2 Pins

The second audio input channel is available as a secondary function on some of the GPIO pins, as shown in the table below. Refer to the “Miscellaneous IO Controller” section of this document for information on how to enable this function onto these GPIO pins.

GPIO	Description
4	Second audio input channel data
5	Second audio input channel bit clock
6	Second audio input channel word flag

Audio Input Control Registers

You have to send Communication Bus packets to control the audio input hardware. This is part of the audio output command register mechanism described above, and the addresses given here are in that space.

ain1Clock Audio input channel 1 clocking

```
$0100
Write only
```

This register controls how the audio input channel timing is derived.

The internal timing generator, if used, is quite separate to that in the audio output channel (which can also be used here). Timing is derived from ACLK, the audio master clock input pin, and is divided to generate AICLK, which may be output as an over-sample clock or used directly as the bit clock, AIBCLK. AIBCLK can be a simple integer sub-multiple of AICLK, if required.

Bit	Name	Description
31	**internClk**	Enable internal timing generators. This derives the bit and word clocks as from the internal timing generators. Power on default is off. When this bit is clear, audio timing is determined externally, whatever the other settings in this register are.
30	**aoutTiming**	When internal clock timing is set, when this bit is set the timing is derived directly from the audio output channel, when clear the audio input timing generation is used.
26-20	**clkPrescale**	This is the clock pre-scale value for the audio input channel timing generator. It runs from ACLK, the audio clock. This value should be one less than the half clock period of either the over-sample clock (if used), or if the over-sample clock function is not used, this may be programmed to give AI_BCLK directly. See **bitScale** below. It can also be by-passed, see

Bit	Name	Description
		clkDirect below,
19	**clkDirect**	Set for the ACLK input to be used directly as the audio input clock. **clkPrescale** is ignored if this is set.
17-16	**bitScale**	This gives the AI_BCLK rate from the output of the **clkPrescale** divider above. 0 divide by 1 (no scaling) 1 divide by 2 2 divide by 4
14-13	**period**	Sample period in AI_BCLK cycles 0 64 bit 1 48 bit 2 32 bit 3 unused

ain1Data Audio input channel 1 data format

```
$0104
Write only
```

Bit	Name	Description
31	**ainEna**	Enable audio input bit clock and word strobe pins as outputs. This should be set if using the internal timing generators to act as a master, and should be clear if an external timing source is used that the interface is a slave.
30	**wordPolarity**	Set for left data latched on or sometime after an AIWCLK rising edge, clear for this to be the right data. Left data is always the word received first, and so in some circumstances the channels may require reversing.
29-24	**dataDelay**	The input data is moved into a 16-bit shift register. This value gives the number of bit clocks after an edge on word clock that this shift register data is captured into the left or right data registers. It should be set to zero for “right” data alignment, and to 16 for “left” data alignment. One should be added to the values if the data framing is delayed one bit cycle relative to the word clock. Normally values between 0 and 31 are used. Valid values are 0-62. A value of 63 may be used here to disable audio input.
23	**clkPolarity**	Set for input data and word clock captured on a falling edge of AI_BCLK, clear to have them captured on a rising edge. This also inverts the clock output on X_ai_bclk when enabled.
22-16	**ainTarget**	Communication Bus address to return data to. Audio data, if enabled, will then be sent to the sender of this command.
15	**ainFlush**	This acts as a soft reset for the audio input capture logic. If it is set to one the capture logic is reset to its power-on state until this bit is released.
14	**ssyncMode**	When this bit is set the word clock is assumed to be a single pulse instead of an even duty-cycle signal. Left data is captured **dataDelay** after the rising edge (or falling edge if **wordPolarity** is clear) of the word clock, then right data is captured (**dataDelay2** + 2) cycles after that. If this bit is clear, then **dataDelay2** is ignored.
13-8	**dataDelay2**	Offset from the **dataDelay** capture point for left data to the capture point of right data, minus two. Normally this will be 14 or 30. Valid values are 0-62.
7	**clkIntPol**	This allows the polarity of the capture bit clock to be inverted independently of the output clock when the audio input channel is the timing master. This signal is exclusive-ORed with the **clkPolarity** control

		before it is applied to the capture clock, i.e. if this bit and **clkPolarity** are the same, data is captured on a rising edge; if they are different data is captured on a falling edge. *This function is available in Aries 2 only.*

ain2Clock Audio input channel 2 clocking

```
$0200
Write only
```

This register controls how the audio input channel timing is derived.

The internal timing generator, if used, is quite separate to that in the audio output channel (which can also be used here). Timing is derived from ACLK, the audio master clock input pin, and is divided to generate AICLK, which may be output as an over-sample clock or used directly as the bit clock, AIBCLK. AIBCLK can be a simple integer sub-multiple of AICLK, if required.

Bit	Name	Description
31	**internClk**	Enable internal timing generators. This derives the bit and word clocks as from the internal timing generators. Power on default is off. When this bit is clear, audio timing is determined externally, whatever the other settings in this register are.
30	**aoutTiming**	When internal clock timing is set, when this bit is set the timing is derived directly from the audio output channel, when clear the audio input timing generation is used.
26-20	**clkPrescale**	This is the clock pre-scale value for the audio input channel timing generator. It runs from ACLK, the audio clock. This value should be one less than the half clock period of either the over-sample clock (if used), or if the over-sample clock function is not used, this may be programmed to give AI_BCLK directly. See **bitScale** below. It can also be by-passed, see **clkDirect** below,
19	**clkDirect**	Set for the ACLK input to be used directly as the audio input clock. **clkPrescale** is ignored if this is set.
17-16	**bitScale**	This gives the AI_BCLK rate from the output of the **clkPrescale** divider above. 0 divide by 1 (no scaling) 1 divide by 2 2 divide by 4
14-13	**period**	Sample period in AI_BCLK cycles 0 64 bit 1 48 bit 2 32 bit 3 unused

ain2Data Audio input channel data format

```
$0204
Write only
```

Bit	Name	Description
31	**ainEna**	Enable audio input bit clock and word strobe pins as outputs. This should be set if using the internal timing generators to act as a master, and should be clear if an external timing source is used that the interface is a slave.
30	**wordPolarity**	Set for left data latched on or sometime after an AIWCLK rising edge, clear for this to be the right data. Left data is always the word received first, and

		so in some circumstances the channels may require reversing.
29-24	**dataDelay**	The input data is moved into a 16-bit shift register. This value gives the number of bit clocks after an edge on word clock that this shift register data is captured into the left or right data registers. It should be set to zero for “right” data alignment, and to 16 for “left” data alignment. One should be added to the values if the data framing is delayed one bit cycle relative to the word clock. Normally values between 0 and 31 are used. Valid values are 0-62. A value of 63 may be used here to disable audio input.
23	**clkPolarity**	Set for input data and word clock captured on a falling edge of AI_BCLK, clear to have them captured on a rising edge. This also inverts the clock output on X_ai_bclk when enabled.
22-16	**ainTarget**	Communication Bus address to return data to. Audio data, if enabled, will then be sent to the sender of this command.
15	**ainFlush**	This acts as a soft reset for the audio input capture logic. If it is set to one the capture logic is reset to its power-on state until this bit is released.
14	**ssyncMode**	When this bit is set the word clock is assumed to be a single pulse instead of an even duty-cycle signal. Left data is captured **dataDelay** after the rising edge (or falling edge if **wordPolarity** is clear) of the word clock, then right data is captured (**dataDelay2** + 2) cycles after that. If this bit is clear, then **dataDelay2** is ignored.
13-8	**dataDelay2**	Offset from the **dataDelay** capture point for left data to the capture point of right data, minus two. Normally this will be 14 or 30. Valid values are 0-62.
7	**clkIntPol**	This allows the polarity of the capture bit clock to be inverted independently of the output clock when the audio input channel is the timing master. This signal is exclusive-ORed with the **clkPolarity** control before it is applied to the capture clock, i.e. if this bit and **clkPolarity** are the same, data is captured on a rising edge; if they are different data is captured on a falling edge. *This function is available in Aries 2 only.*

Using Audio Input Channel 2 to connect to a CD or DVD drive

The second audio input channel may also be used as an interface to a CD or DVD drive. This may be either the primary interface to the drive, or as a secondary interface to the CDI. Where it is an additional I^{2}S interface, it will normally operate in standard audio input mode. When it is the primary interface, it acts as a front-end to the CDI, reformatting the I2S data into an 8-bit parallel stream suitable for the CDI unit.

Data may therefore be passed either directly over the Communication Bus to an MPE, or via the CDI unit. It is possible to lock it in the mode where data is passed through the CDI.

If this channel is the primary interface to the drive, certain CDI pins may be used by this interface, as follows:

X_casdata	Data error flag
X_careq	Sector sync flag
X_cvenab	CD sub-code data

Driving the CDI Interface from Audio Input Channel 2

Audio input channel 2 can also be used to send data to the CDI. The register below controls this function. This mechanism can optionally detect two special long words so that a block structure can be super-imposed on the data-stream. If the *“header”* long-word is encountered in the data-stream then the first byte of it is flagged to the CDI as being the top byte of the block. If an *“escape”* long-word is encountered in the data stream then the next long-word is treated as regular data whatever its contents. Therefore *“header”* in the normal data stream should be replaced by *“escape – header”*, and *“escape”* in the data stream should be replaced by *“escape – escape”*.

Code	Hex Value
Escape	`7E95D5B6`
Header	`A1400796`

ain2Special Audio input channel 2 special function control

```
$0208
Write only
```

This register controls how audio input channel 2 delivers data to the CDI.

Bit	Name	Description
31	**lock**	When this bit is set, audio input channel 2 will not send data over the Communication Bus. This bit may be set by software, but can only be cleared by a power-on reset.
7	**v4CapPol**	This bit control on which phase of WS the captured V4 codes are passed on to the Communication Bus interface.
6	**cdiErrExt**	When this bit is set, the error flag passed to the CDI is captured from the CDI casdata pin. When this bit is clear, no errors are flagged.
5	**cdiHdrExt**	When this bit is set, the header mechanism described above is over-ridden, and the header flag passed to the CDI is instead captured from the CDI careq pin.
4	**cdiHdrPass**	When this bit is set the *“header”* data pattern is passed through to the CDI. Start of block is then flagged on this long-word as opposed to the one that follows it.
3	**cdiEscPass**	When this bit is set the *“escape”* data pattern is passed through to the CDI.
2	**cdiEnable**	When this bit is set, data transfer is enabled to the CDI. Data will no longer be sent over the Communication Bus.
1-0	**cdiEndian**	These control the byte ordering of the data sent to the CDI. Bit 0 swaps bytes within words, bit 1 swaps words.

Audio Input Communication Bus Packet Format

Communication Bus packets are used to pass audio input data to the MPEs. Communication Bus packets can also be sent in response to register reads, and these should be separated in software.

The packet format is:

Long word	**Bits**	**Description**
0	0-1	0 for a read response 1 for audio input ch. 1 data 2 for audio input ch. 2 data
	2-5	unused
	6-15	Extended status data for audio input channel 2, zero for other packets 6-7 right sync data 8-9 right flag data 10-11 left sync data 12-13 left flag data 14-15 V4 data
	16-31	Read address for read response packets.
1	0-31	Register read data or audio input data, left in bits 31-16, right in bits 15-0.
2		unused
3		unused

MISCELLANEOUS IO CONTROLLER

The miscellaneous IO controller is a Communication Bus interface that provides system access to the interface logic for the following:

- 16 general-purpose pins which may be used for communicating with and configuring a variety of external hardware. These pins may be individually configured as inputs or outputs, and can be used as interrupt sources. Some of them may be assigned to special functions.
- The system timers which can generate three interrupts at programmable rates. Each of these interrupts is available to all the MPEs.
- The setup registers for the ROM interface, and for the Communication Bus controller.
- The external controller device (joystick) interface registers.
- The PWM output control register.
- The power-on configuration bits.

Miscellaneous IO Communication Bus Protocol

The registers described below are programmed over the Communication Bus. The miscellaneous IO controller has Communication Bus ID 69 (hex 45). The communication protocol is as follows for the command packet:

Long word	Description
0	0-15 register address 31 set for write, clear for read
1	0-31 write data if a write command
2	unused
3	unused

The response packet is returned if the operation was a read. Its format is:

Long word	Description
0	0-1 read status 16-31 register address of an IO read
1	0-31 read data
2	unused
3	unused

The read status bits are used to indicate if the response packet is an IO read response, or controller data sent automatically because the **commSend** bit is set, as follows:

Status	Function
0	IO Read data
1	Controller 1 data
2	Controller 2 data

System Timers

The system timers provide three interrupt signals, which are available to all the MPEs. These timers may be independently programmed for a variety of system functions.

There are actually four timer-counters, one of which acts as a rate pre-scale counter for the other three. Each timer consists of a holding register and a down counter. A write will change the value in the holding register, but the value is not actually transferred to the counter until the next time it reaches zero. A read returns the current contents of the down counter. Timers 0, 1 and 2 will count down by one whenever the pre-scale counter contains zero.

The timers generate an interrupt pulse every time the count value reaches one. When the count values reaches zero, the counter is reloaded from the holding register. This means that a value of zero disables the counter.

timerPre System Timer Pre-Scale

```
$0000
Read / Write
```

This register defines a pre-scale value from the system clock which controls a pre-scale timer. This is used to give the count rate of the main timers, which are described below. This allows the timers to be programmed by applications without them being aware of the system clock speed. Normally a pre-scale of 54 is set (for NUON), so that the timers count at 1 MHz, and will continue to do so in future faster versions of the architecture,

The current value of the counter may be read, although this is primarily for debug purposes as it counts so fast.

Bit	Description
31-8	Reserved, write zeroes.
7-0	Timer pre-scale value. The system clock is divided by (1+n), where n is the value written to these bits. Zero means that the timers count at the full clock rate.

timer0 System timer 0

```
$0001
Read / Write
```

This register defines the count rate of system timer 0, in units defined by the timer pre-scale value above. Timer 0 is available to all the MPEs, and is used to provide interrupts, as required, at a defined rate. All processors which have this interrupt enabled will be interrupted together.

The current value of the counter may be read.

Bit	Description
31-20	Reserved, write zeroes.
19-0	Timer count value. This timer counts down at a rate defined by the pre-scale value, and will generate loop at a rate given by the output of the pre-scale counter divided by (1+n), where n is the value written to these bits. Zero will disable the timer.

timer1 System timer 1

```
$0002
Read / Write
```

This register defines the count rate of system timer 1, in units defined by the timer pre-scale value above. Timer 1 is available to all the MPEs, and is used to provide interrupts, as required, at a defined rate. All processors which have this interrupt enabled will be interrupted together.

The current value of the counter may be read.

Bit	Description
31-20	Reserved, write zeroes.
19-0	Timer count value. This timer counts down at a rate defined by the pre-scale value, and will generate loop at a rate given by the output of the pre-scale counter divided by (1+n), where n is the value written to these bits. Zero will disable the timer.

timer2 System timer 2

```
$0003
Read / Write
```

This register defines the count rate of system timer 2, in units defined by the timer pre-scale value above. Timer 2 is available to all the MPEs, and is used to provide interrupts, as required, at a defined rate. All processors which have this interrupt enabled will be interrupted together.

The current value of the counter may be read.

Bit	Description
31-20	Reserved, write zeroes.
19-0	Timer count value. This timer counts down at a rate defined by the pre-scale value, and will generate loop at a rate given by the output of the pre-scale counter divided by (1+n), where n is the value written to these bits. Zero will disable the timer.

ROM Interface Control

The ROM interface itself is described in the ROM Bus section of this documentation

romCtrl ROM Interface Control

```
$0010
Read / Write
```

Bit	Name	Description
31	**testerMode**	Sets the ROM interface in tester mode. This forces an idle state between ROM accesses and is only required for diagnostic test mode. This defaults to on.
30-28	**romIdlePause**	ROM idle pause time. If this is non-zero, the tester mode function below is overridden. The ROM interface will go idle for 2+n clock cycles between each transfer, where n is the value written here. This is necessary for ROMs that have a long tri-state disable time (most of them).
27-5	**unused**	Write zero.
4	**oneByte**	Overrides the normal length of Other Bus transfers so that they may be one byte long. When this bit is set Other Bus transfers should always have length 1. One byte will be written or read from bits 31-24 of the data. The address, as usual, can be on any byte boundary. This is useful for programming flash memory.
0-3	**cycleTim**	ROM cycle time in clock cycles. Must be in the range 1-15 as appropriate for the speed of the external memory. See the ROM Bus description.

flashTiming ROM Interface Control

```
$0011
Read / Write
```

This register is available on Aries 3 and upwards only.

Bit	Name	Description
31	**flashMode**	When set, the interface supports NAND flash memory, an 8-bit latched interface as used by SmartMedia. When clear the interface supports regular bus style flash memory, ROM and static RAM. The default state of this bit is also controlled by the configuration resistor on X_vdata_1. This should be pulled high for set, low for clear.
30	**sysBusMode**	When set, the interface is multiplexed onto System Bus pins. When clear, it uses its own private bus. The default state of this bit is also controlled by the configuration resistor on X_vdata_3. This should be pulled high for set, low for clear.
29	**readExtra16**	When set, the "extra" 16 bytes in each 512-byte sector may be read. Normal reads are not possible, and only the bottom 16 bytes of each 512-byte area are valid. When clear reads occur normally.
28	**readExtra16onA23**	When set, the "extra" 16 bytes in each 512-byte sector may be read. These appear on addresses where A23 is high, i.e. from $F0800000 upwards. This is only useful when the flash memory is 8 Mbytes or smaller, where the memory itself ignores A23. When clear reads occur normally.
27-26	**unused**	Write zero.
25-20	**flashTwb**	WE high to busy delay.
19-16	**flashTwp**	Write pulse width.
15-12	**flashTwh**	WE high hold time.
11-8	**flashTrr**	Ready to RE falling edge.
7-4	**flashTrp**	Read pulse width.
3-0	**flashTrh**	Worst of RE high hold time, and RE high to output high impedance.

flashSpecTim ROM Interface Control

```
$0012
Read / Write
```

This register is available on Aries 3 and upwards only.

Bit	Name	Description
31-12	**unused**	Write zero.
11-8	**flashTsu**	Defines the setup time, in main (54 MHz) clock cycles, of a special cycle. In this phase, **CE** is low, **CLE** and **ALE** are as programmed below, and write data is set up if the data is enabled. **RE** and **WE** are both 1 in this phase.
7-4	**flashTsa**	Defines the strobe active time, in main (54 MHz) clock cycles, of a special cycle. In this phase, **RE** goes low if **specialRead** is high, otherwise **WE** goes low.
3-0	**flashTsh**	Defines the hold time, in main (54 MHz) clock cycles, of a special cycle. In this phase conditions are the same as the setup phase.

flashSpecCtl ROM Interface Control

```
$0013
Read / Write
```

This register allows special cycles to be performed. These can be **RE** or **WE** strobed, reads or writes, and **CLE** and **ALE** set. This allows pretty much anything to be performed by software. Controlling software should make sure no reads or writes are active.

During a special cycle, **CE** goes low for the duration of the cycle. The cycle is divided into three phases, with the timing controlled by the **flashSpecTim** register. These are setup, active, and hold. **RE,** or **WE,** is strobed only during the active phase.

This register is available on Aries 3 and upwards only.

Bit	Name	Description
31	**specialActive**	This is set to one to initiate a special cycle. It must be polled until it returns to zero before performing either another special cycle or a normal read or write operation.
30	**specialCle**	Defines the polarity of the **CLE** signal during the special cycle.
29	**specialAle**	Defines the polarity of the **ALE** signal during the special cycle.
28	**specialRead**	Defines whether the cycle is a read, where **RE** is strobed, or a write, where **WE** is strobed. Set for a read, clear for a write.
27	**specialDataDir**	Defines the direction of the data bus during the special cycle. This should be set to one for a read, and zero for a write, in which case **specialWData** is enabled onto the data bus for the duration of the cycle.
26	**specialBusyWait**	Makes the special cycle hold off until **BUSY** is inactive (high). The special cycle will not start its setup phase until **BUSY** goes high.
25	**specialHold**	This signal should be set in order to hold CE low between special cycles. Set this to one for all but the last cycle of a group that must occur during one CE low time. If the flash is on the System Bus, then the System Bus is held by the flash memory throughout this period. Therefore the maximum time this may be set for must be limited to (??) microseconds.
24-16	**unused**	Write zero.
15-8	**specialWData**	Write data for write cycles.
7-0	**specialRData**	Read data for read cycles. This is captured at the end of the active phase of the special cycle. It will not be valid for read until **specialActive** has returned to zero. This is read only (write zeroes).

General Purpose IO Pins

Aries 1 and 2 have 16 general-purpose IO pins, which can be individually configured as inputs or outputs, and may also be used as interrupt sources (when configured as inputs).

Aries 3 adds an additional 20 GPIO pins, some of which overlay existing, but rarely used, functions. Other pins are in unused ball positions. None of these are available when the 208-pin QFP package is selected. GPIO pins 16-28 occupy BGA ball positions which are unused in Aries 1-2. GPIO pins 29-35 allow System Bus address lines 24-18 to be used for alternate functions in the BGA package.

Many of the GPIO pins have special functions, used by modules outside the GPIO pin control logic. The special control bits below switch them over to these special functions, which are:

GPIO	First Special Function	Second Special Function	Third Special Function
0	System Bus interrupt output		
1	Audio input high rate clock output	Audio output I2S data line sdat[3]	
2	Serial Peripheral Bus interface SCL		
3	Serial Peripheral Bus interface SDA		
4	Second audio input channel data		
5	Second audio input channel bit clock		
6	Second audio input channel word flag		
7	System Bus external mode TSIZ(0)	SIO A clock	
8	System Bus external mode TSIZ(1)	SIO A request	
9	System Bus A(25)	SIO A transmit data	
10	System Bus A(26)	SIO A request/acknowledge	Secondary Serial Peripheral Bus interface SCL
11	System Bus A(27)	SIO A receive data	Secondary Serial Peripheral Bus interface SDA
12	System Bus A(28)	System Bus NAND flash busy.	
13	System Bus A(29)	PWM0	
14	System Bus A(30)	PWM1	
15	System Bus A(31)	UAE	
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			
26			
27			
28			
29	System Bus A(18)	SIO B clock	
30	System Bus A(19)	SIO B request	
31	System Bus A(20)	SIO B transmit data	
32	System Bus A(21)	SIO B request/acknowledge	
33	System Bus A(22)	SIO B receive data	
34	System Bus A(23)		
35	System Bus A(24)		

The control registers are:

gpioCtrl0 General Purpose IO Pin Control 0

```
$0020
Read / Write
```

This register controls GPIO pins 0-7 when they are operating normally (special functions disabled). The outputs can also be controlled in an atomic manner by the gpioAt0 register described below. This is useful if multiple processors want to control pins in this group.

Bit	Name	Description
30	**gp7in**	When read, this gives the external state of the GPIO(7) pin.
29	**gp7out**	When the GPIO(7) pin is an output, this is the data driven on to it.
28	**gp7enable**	When set, the GPIO(7) pin is enabled as an output.
26	**gp6in**	When read, this gives the external state of the GPIO(6) pin.
25	**gp6out**	When the GPIO(6) pin is an output, this is the data driven on to it.
24	**gp6enable**	When set, the GPIO(6) pin is enabled as an output.
22	**gp5in**	When read, this gives the external state of the GPIO(5) pin.
21	**gp5out**	When the GPIO(5) pin is an output, this is the data driven on to it.
20	**gp5enable**	When set, the GPIO(5) pin is enabled as an output.
18	**gp4in**	When read, this gives the external state of the GPIO(4) pin.
17	**gp4out**	When the GPIO(4) pin is an output, this is the data driven on to it.
16	**gp4enable**	When set, the GPIO(4) pin is enabled as an output.
14	**gp3in**	When read, this gives the external state of the GPIO(3) pin.
13	**gp3out**	When the GPIO(3) pin is an output, this is the data driven on to it.
12	**gp3enable**	When set, the GPIO(3) pin is enabled as an output.
10	**gp2in**	When read, this gives the external state of the GPIO(2) pin.
9	**gp2out**	When the GPIO(2) pin is an output, this is the data driven on to it.
8	**gp2enable**	When set, the GPIO(2) pin is enabled as an output.
6	**gp1in**	When read, this gives the external state of the GPIO(1) pin.
5	**gp1out**	When the GPIO(1) pin is an output, this is the data driven on to it.
4	**gp1enable**	When set, the GPIO(1) pin is enabled as an output.
2	**gp0in**	When read, this gives the external state of the GPIO(0) pin.
1	**gp0out**	When the GPIO(0) pin is an output, this is the data driven on to it.
0	**gp0enable**	When set, the GPIO(0) pin is enabled as an output.

gpioAt0 General Purpose IO Pin Atomic Control 0

```
$0028
Write Only
```

This register allows GPIO pins 0-7 to be controlled independently by multiple processes, by allowing them to be modified atomically. If the bits corresponding to a particular GPIO pin are all set to zero, it is not modified. If a bit is set, then it is used to either set or clear a GPIO control bit.

For all bits, set has priority over clear, if both commands are given.

Bit	Name	Description
31	**gp7Dset**	The GPIO(7) data output bit (**gp7out**) is set by this bit.
30	**gp7Dclr**	The GPIO(7) data output bit (**gp7out**) is cleared by this bit.
29	**gp7Eset**	The GPIO(7) output enable bit (**gp7enable**) is set by this bit.
28	**gp7Eclr**	The GPIO(7) output enable bit (**gp7enable**) is cleared by this bit.

Bit	Name	Description
27	**gp6Dset**	The GPIO(6) data output bit (**gp6out**) is set by this bit.
26	**gp6Dclr**	The GPIO(6) data output bit (**gp6out**) is cleared by this bit.
25	**gp6Eset**	The GPIO(6) output enable bit (**gp6enable**) is set by this bit.
24	**gp6Eclr**	The GPIO(6) output enable bit (**gp6enable**) is cleared by this bit.
23	**gp5Dset**	The GPIO(5) data output bit (**gp5out**) is set by this bit.
22	**gp5Dclr**	The GPIO(5) data output bit (**gp5out**) is cleared by this bit.
21	**gp5Eset**	The GPIO(5) output enable bit (**gp5enable**) is set by this bit.
20	**gp5Eclr**	The GPIO(5) output enable bit (**gp5enable**) is cleared by this bit.
19	**gp4Dset**	The GPIO(4) data output bit (**gp4out**) is set by this bit.
18	**gp4Dclr**	The GPIO(4) data output bit (**gp4out**) is cleared by this bit.
17	**gp4Eset**	The GPIO(4) output enable bit (**gp4enable**) is set by this bit.
16	**gp4Eclr**	The GPIO(4) output enable bit (**gp4enable**) is cleared by this bit.
15	**gp3Dset**	The GPIO(3) data output bit (**gp3out**) is set by this bit.
14	**gp3Dclr**	The GPIO(3) data output bit (**gp3out**) is cleared by this bit.
13	**gp3Eset**	The GPIO(3) output enable bit (**gp3enable**) is set by this bit.
12	**gp3Eclr**	The GPIO(3) output enable bit (**gp3enable**) is cleared by this bit.
11	**gp2Dset**	The GPIO(2) data output bit (**gp2out**) is set by this bit.
10	**gp2Dclr**	The GPIO(2) data output bit (**gp2out**) is cleared by this bit.
9	**gp2Eset**	The GPIO(2) output enable bit (**gp2enable**) is set by this bit.
8	**gp2Eclr**	The GPIO(2) output enable bit (**gp2enable**) is cleared by this bit.
7	**gp1Dset**	The GPIO(1) data output bit (**gp1out**) is set by this bit.
6	**gp1Dclr**	The GPIO(1) data output bit (**gp1out**) is cleared by this bit.
5	**gp1Eset**	The GPIO(1) output enable bit (**gp1enable**) is set by this bit.
4	**gp1Eclr**	The GPIO(1) output enable bit (**gp1enable**) is cleared by this bit.
3	**gp0Dset**	The GPIO(0) data output bit (**gp0out**) is set by this bit.
2	**gp0Dclr**	The GPIO(0) data output bit (**gp0out**) is cleared by this bit.
1	**gp0Eset**	The GPIO(0) output enable bit (**gp0enable**) is set by this bit.
0	**gp0Eclr**	The GPIO(0) output enable bit (**gp0enable**) is cleared by this bit.

gpioCtrl1 General Purpose IO Pin Control 1

```
$0021
Read / Write
```

This register controls GPIO pins 8-15 when they are operating normally (special functions disabled). The outputs can also be controlled in an atomic manner by the gpioAt1 register described below. This is useful if multiple processors want to control pins in this group.

Bit	Name	Description
30	**gp15in**	When read, this gives the external state of the GPIO(15) pin.
29	**gp15out**	When the GPIO(15) pin is an output, this is the data driven on to it.
28	**gp15enable**	When set, the GPIO(15) pin is enabled as an output.
26	**gp14in**	When read, this gives the external state of the GPIO(14) pin.
25	**gp14out**	When the GPIO(14) pin is an output, this is the data driven on to it.
24	**gp14enable**	When set, the GPIO(14) pin is enabled as an output.
22	**gp13in**	When read, this gives the external state of the GPIO(13) pin.
21	**gp13out**	When the GPIO(13) pin is an output, this is the data driven on to it.
20	**gp13enable**	When set, the GPIO(13) pin is enabled as an output.
18	**gp12in**	When read, this gives the external state of the GPIO(12) pin.
17	**gp12out**	When the GPIO(12) pin is an output, this is the data driven on to it.
16	**gp12enable**	When set, the GPIO(12) pin is enabled as an output.

Bit	Name	Description
14	**gp11in**	When read, this gives the external state of the GPIO(11) pin.
13	**gp11out**	When the GPIO(11) pin is an output, this is the data driven on to it.
12	**gp11enable**	When set, the GPIO(11) pin is enabled as an output.
10	**gp10in**	When read, this gives the external state of the GPIO(10) pin.
9	**gp10out**	When the GPIO(10) pin is an output, this is the data driven on to it.
8	**gp10enable**	When set, the GPIO(10) pin is enabled as an output.
6	**gp9in**	When read, this gives the external state of the GPIO(9) pin.
5	**gp9out**	When the GPIO(9) pin is an output, this is the data driven on to it.
4	**gp9enable**	When set, the GPIO(9) pin is enabled as an output.
2	**gp8in**	When read, this gives the external state of the GPIO(8) pin.
1	**gp8out**	When the GPIO(8) pin is an output, this is the data driven on to it.
0	**gp8enable**	When set, the GPIO(8) pin is enabled as an output.

gpioAt1 General Purpose IO Pin Atomic Control 1

```
$0029
Write Only
```

This register allows GPIO pins 8-15 to be controlled independently by multiple processes, by allowing them to be modified atomically. If the bits corresponding to a particular GPIO pin are all set to zero, it is not modified. If a bit is set, then it is used to either set or clear a GPIO control bit.

For all bits, set has priority over clear, if both commands are given.

Bit	Name	Description
31	**gp15Dset**	The GPIO(15) data output bit (**gp15out**) is set by this bit.
30	**gp15Dclr**	The GPIO(15) data output bit (**gp15out**) is cleared by this bit.
29	**gp15Eset**	The GPIO(15) output enable bit (**gp15enable**) is set by this bit.
28	**gp15Eclr**	The GPIO(15) output enable bit (**gp15enable**) is cleared by this bit.
27	**gp14Dset**	The GPIO(14) data output bit (**gp14out**) is set by this bit.
26	**gp14Dclr**	The GPIO(14) data output bit (**gp14out**) is cleared by this bit.
25	**gp14Eset**	The GPIO(14) output enable bit (**gp14enable**) is set by this bit.
24	**gp14Eclr**	The GPIO(14) output enable bit (**gp14enable**) is cleared by this bit.
23	**gp13Dset**	The GPIO(13) data output bit (**gp13out**) is set by this bit.
22	**gp13Dclr**	The GPIO(13) data output bit (**gp13out**) is cleared by this bit.
21	**gp13Eset**	The GPIO(13) output enable bit (**gp13enable**) is set by this bit.
20	**gp13Eclr**	The GPIO(13) output enable bit (**gp13enable**) is cleared by this bit.
19	**gp12Dset**	The GPIO(12) data output bit (**gp12out**) is set by this bit.
18	**gp12Dclr**	The GPIO(12) data output bit (**gp12out**) is cleared by this bit.
17	**gp12Eset**	The GPIO(12) output enable bit (**gp12enable**) is set by this bit.
16	**gp12Eclr**	The GPIO(12) output enable bit (**gp12enable**) is cleared by this bit.
15	**gp11Dset**	The GPIO(11) data output bit (**gp11out**) is set by this bit.
14	**gp11Dclr**	The GPIO(11) data output bit (**gp11out**) is cleared by this bit.
13	**gp11Eset**	The GPIO(11) output enable bit (**gp11enable**) is set by this bit.
12	**gp11Eclr**	The GPIO(11) output enable bit (**gp11enable**) is cleared by this bit.
11	**gp10Dset**	The GPIO(10) data output bit (**gp10out**) is set by this bit.
10	**gp10Dclr**	The GPIO(10) data output bit (**gp10out**) is cleared by this bit.
9	**gp10Eset**	The GPIO(10) output enable bit (**gp10enable**) is set by this bit.
8	**gp10Eclr**	The GPIO(10) output enable bit (**gp10enable**) is cleared by this bit.
7	**gp9Dset**	The GPIO(9) data output bit (**gp9out**) is set by this bit.

Bit	Name	Description
6	**gp9Dclr**	The GPIO(9) data output bit (**gp9out**) is cleared by this bit.
5	**gp9Eset**	The GPIO(9) output enable bit (**gp9enable**) is set by this bit.
4	**gp9Eclr**	The GPIO(9) output enable bit (**gp9enable**) is cleared by this bit.
3	**gp8Dset**	The GPIO(8) data output bit (**gp8out**) is set by this bit.
2	**gp8Dclr**	The GPIO(8) data output bit (**gp8out**) is cleared by this bit.
1	**gp8Eset**	The GPIO(8) output enable bit (**gp8enable**) is set by this bit.
0	**gp8Eclr**	The GPIO(8) output enable bit (**gp8enable**) is cleared by this bit.

gpioCtrl2 General Purpose IO Pin Control 2

```
$002B
Read / Write
```

This register controls GPIO pins 16-23 when they are operating as GPIO pins. The outputs can also be controlled in an atomic manner by the gpioAt2 register described below. This is useful if multiple processors want to control pins in this group.

Bit	Name	Description
30	**gp23in**	When read, this gives the external state of the GPIO(23) pin.
29	**gp23out**	When the GPIO(23) pin is an output, this is the data driven on to it.
28	**gp23enable**	When set, the GPIO(23) pin is enabled as an output.
26	**gp22in**	When read, this gives the external state of the GPIO(22) pin.
25	**gp22out**	When the GPIO(22) pin is an output, this is the data driven on to it.
24	**gp22enable**	When set, the GPIO(22) pin is enabled as an output.
22	**gp21in**	When read, this gives the external state of the GPIO(21) pin.
21	**gp21out**	When the GPIO(21) pin is an output, this is the data driven on to it.
20	**gp21enable**	When set, the GPIO(21) pin is enabled as an output.
18	**gp20in**	When read, this gives the external state of the GPIO(20) pin.
17	**gp20out**	When the GPIO(20) pin is an output, this is the data driven on to it.
16	**gp20enable**	When set, the GPIO(20) pin is enabled as an output.
14	**gp19in**	When read, this gives the external state of the GPIO(19) pin.
13	**gp19out**	When the GPIO(19) pin is an output, this is the data driven on to it.
12	**gp19enable**	When set, the GPIO(19) pin is enabled as an output.
10	**gp18in**	When read, this gives the external state of the GPIO(18) pin.
9	**gp18out**	When the GPIO(18) pin is an output, this is the data driven on to it.
8	**gp18enable**	When set, the GPIO(18) pin is enabled as an output.
6	**gp17in**	When read, this gives the external state of the GPIO(17) pin.
5	**gp17out**	When the GPIO(17) pin is an output, this is the data driven on to it.
4	**gp17enable**	When set, the GPIO(17) pin is enabled as an output.
2	**gp16in**	When read, this gives the external state of the GPIO(16) pin.
1	**gp16out**	When the GPIO(16) pin is an output, this is the data driven on to it.
0	**gp16enable**	When set, the GPIO(16) pin is enabled as an output.

gpioAt2 General Purpose IO Pin Atomic Control 2

```
$002E
Write Only
```

This register allows GPIO pins 16-23 to be controlled independently by multiple processes, by allowing them to be modified atomically. If the bits corresponding to a particular GPIO pin are all set to zero, it is not modified. If a bit is set, then it is used to either set or clear a GPIO control bit.

For all bits, set has priority over clear, if both commands are given.

Bit	Name	Description
31	**gp23Dset**	The GPIO(23) data output bit (**gp23out**) is set by this bit.
30	**gp23Dclr**	The GPIO(23) data output bit (**gp23out**) is cleared by this bit.
29	**gp23Eset**	The GPIO(23) output enable bit (**gp23enable**) is set by this bit.
28	**gp23Eclr**	The GPIO(23) output enable bit (**gp23enable**) is cleared by this bit.
27	**gp22Dset**	The GPIO(22) data output bit (**gp22out**) is set by this bit.
26	**gp22Dclr**	The GPIO(22) data output bit (**gp22out**) is cleared by this bit.
25	**gp22Eset**	The GPIO(22) output enable bit (**gp22enable**) is set by this bit.
24	**gp22Eclr**	The GPIO(22) output enable bit (**gp22enable**) is cleared by this bit.
23	**gp21Dset**	The GPIO(21) data output bit (**gp21out**) is set by this bit.
22	**gp21Dclr**	The GPIO(21) data output bit (**gp21out**) is cleared by this bit.
21	**gp21Eset**	The GPIO(21) output enable bit (**gp21enable**) is set by this bit.
20	**gp21Eclr**	The GPIO(21) output enable bit (**gp21enable**) is cleared by this bit.
19	**gp20Dset**	The GPIO(20) data output bit (**gp20out**) is set by this bit.
18	**gp20Dclr**	The GPIO(20) data output bit (**gp20out**) is cleared by this bit.
17	**gp20Eset**	The GPIO(20) output enable bit (**gp20enable**) is set by this bit.
16	**gp20Eclr**	The GPIO(20) output enable bit (**gp20enable**) is cleared by this bit.
15	**gp19Dset**	The GPIO(19) data output bit (**gp19out**) is set by this bit.
14	**gp19Dclr**	The GPIO(19) data output bit (**gp19out**) is cleared by this bit.
13	**gp19Eset**	The GPIO(19) output enable bit (**gp19enable**) is set by this bit.
12	**gp19Eclr**	The GPIO(19) output enable bit (**gp19enable**) is cleared by this bit.
11	**gp18Dset**	The GPIO(18) data output bit (**gp18out**) is set by this bit.
10	**gp18Dclr**	The GPIO(18) data output bit (**gp18out**) is cleared by this bit.
9	**gp18Eset**	The GPIO(18) output enable bit (**gp18enable**) is set by this bit.
8	**gp18Eclr**	The GPIO(18) output enable bit (**gp18enable**) is cleared by this bit.
7	**gp17Dset**	The GPIO(17) data output bit (**gp17out**) is set by this bit.
6	**gp17Dclr**	The GPIO(17) data output bit (**gp17out**) is cleared by this bit.
5	**gp17Eset**	The GPIO(17) output enable bit (**gp17enable**) is set by this bit.
4	**gp17Eclr**	The GPIO(17) output enable bit (**gp17enable**) is cleared by this bit.
3	**gp16Dset**	The GPIO(16) data output bit (**gp16out**) is set by this bit.
2	**gp16Dclr**	The GPIO(16) data output bit (**gp16out**) is cleared by this bit.
1	**gp16Eset**	The GPIO(16) output enable bit (**gp16enable**) is set by this bit.
0	**gp16Eclr**	The GPIO(16) output enable bit (**gp16enable**) is cleared by this bit.

gpioCtrl3 General Purpose IO Pin Control 3

```
$002C
Read / Write
```

This register controls GPIO pins 24-31 when they are operating as GPIO pins. The outputs can also be controlled in an atomic manner by the gpioAt3 register described below. This is useful if multiple processors want to control pins in this group.

Bit	Name	Description
30	**gp31in**	When read, this gives the external state of the GPIO(31) pin.
29	**gp31out**	When the GPIO(31) pin is an output, this is the data driven on to it.
28	**gp31enable**	When set, the GPIO(31) pin is enabled as an output.
26	**gp30in**	When read, this gives the external state of the GPIO(30) pin.
25	**gp30out**	When the GPIO(30) pin is an output, this is the data driven on to it.
24	**gp30enable**	When set, the GPIO(30) pin is enabled as an output.

Bit	Name	Description
22	**gp29in**	When read, this gives the external state of the GPIO(29) pin.
21	**gp29out**	When the GPIO(29) pin is an output, this is the data driven on to it.
20	**gp29enable**	When set, the GPIO(29) pin is enabled as an output.
18	**gp28in**	When read, this gives the external state of the GPIO(28) pin.
17	**gp28out**	When the GPIO(28) pin is an output, this is the data driven on to it.
16	**gp28enable**	When set, the GPIO(28) pin is enabled as an output.
14	**gp27in**	When read, this gives the external state of the GPIO(27) pin.
13	**gp27out**	When the GPIO(27) pin is an output, this is the data driven on to it.
12	**gp27enable**	When set, the GPIO(27) pin is enabled as an output.
10	**gp26in**	When read, this gives the external state of the GPIO(26) pin.
9	**gp26out**	When the GPIO(26) pin is an output, this is the data driven on to it.
8	**gp26enable**	When set, the GPIO(26) pin is enabled as an output.
6	**gp25in**	When read, this gives the external state of the GPIO(25) pin.
5	**gp25out**	When the GPIO(25) pin is an output, this is the data driven on to it.
4	**gp25enable**	When set, the GPIO(25) pin is enabled as an output.
2	**gp24in**	When read, this gives the external state of the GPIO(24) pin.
1	**gp24out**	When the GPIO(24) pin is an output, this is the data driven on to it.
0	**gp24enable**	When set, the GPIO(24) pin is enabled as an output.

gpioAt3 General Purpose IO Pin Atomic Control 3

```
$002F
Write Only
```

This register allows GPIO pins 24-31 to be controlled independently by multiple processes, by allowing them to be modified atomically. If the bits corresponding to a particular GPIO pin are all set to zero, it is not modified. If a bit is set, then it is used to either set or clear a GPIO control bit.

For all bits, set has priority over clear, if both commands are given.

Bit	Name	Description
31	**gp31Dset**	The GPIO(31) data output bit (**gp31out**) is set by this bit.
30	**gp31Dclr**	The GPIO(31) data output bit (**gp31out**) is cleared by this bit.
29	**gp31Eset**	The GPIO(31) output enable bit (**gp31enable**) is set by this bit.
28	**gp31Eclr**	The GPIO(31) output enable bit (**gp31enable**) is cleared by this bit.
27	**gp30Dset**	The GPIO(30) data output bit (**gp30out**) is set by this bit.
26	**gp30Dclr**	The GPIO(30) data output bit (**gp30out**) is cleared by this bit.
25	**gp30Eset**	The GPIO(30) output enable bit (**gp30enable**) is set by this bit.
24	**gp30Eclr**	The GPIO(30) output enable bit (**gp30enable**) is cleared by this bit.
23	**gp29Dset**	The GPIO(29) data output bit (**gp29out**) is set by this bit.
22	**gp29Dclr**	The GPIO(29) data output bit (**gp29out**) is cleared by this bit.
21	**gp29Eset**	The GPIO(29) output enable bit (**gp29enable**) is set by this bit.
20	**gp29Eclr**	The GPIO(29) output enable bit (**gp29enable**) is cleared by this bit.
19	**gp28Dset**	The GPIO(28) data output bit (**gp28out**) is set by this bit.
18	**gp28Dclr**	The GPIO(28) data output bit (**gp28out**) is cleared by this bit.
17	**gp28Eset**	The GPIO(28) output enable bit (**gp28enable**) is set by this bit.
16	**gp28Eclr**	The GPIO(28) output enable bit (**gp28enable**) is cleared by this bit.
15	**gp27Dset**	The GPIO(27) data output bit (**gp27out**) is set by this bit.
14	**gp27Dclr**	The GPIO(27) data output bit (**gp27out**) is cleared by this bit.
13	**gp27Eset**	The GPIO(27) output enable bit (**gp27enable**) is set by this bit.

Bit	Name	Description
12	**gp27Eclr**	The GPIO(27) output enable bit (**gp27enable**) is cleared by this bit.
11	**gp26Dset**	The GPIO(26) data output bit (**gp26out**) is set by this bit.
10	**gp26Dclr**	The GPIO(26) data output bit (**gp26out**) is cleared by this bit.
9	**gp26Eset**	The GPIO(26) output enable bit (**gp26enable**) is set by this bit.
8	**gp26Eclr**	The GPIO(26) output enable bit (**gp26enable**) is cleared by this bit.
7	**gp25Dset**	The GPIO(25) data output bit (**gp25out**) is set by this bit.
6	**gp25Dclr**	The GPIO(25) data output bit (**gp25out**) is cleared by this bit.
5	**gp25Eset**	The GPIO(25) output enable bit (**gp25enable**) is set by this bit.
4	**gp25Eclr**	The GPIO(25) output enable bit (**gp25enable**) is cleared by this bit.
3	**gp24Dset**	The GPIO(24) data output bit (**gp24out**) is set by this bit.
2	**gp24Dclr**	The GPIO(24) data output bit (**gp24out**) is cleared by this bit.
1	**gp24Eset**	The GPIO(24) output enable bit (**gp24enable**) is set by this bit.
0	**gp24Eclr**	The GPIO(24) output enable bit (**gp24enable**) is cleared by this bit.

gpioCtrl4 General Purpose IO Pin Control 4

```
$0038
Read / Write
```

This register controls GPIO pins 32-39 when they are operating as GPIO pins. The outputs can also be controlled in an atomic manner by the gpioAt4 register described below. This is useful if multiple processors want to control pins in this group.

Bit	Name	Description
30	**gp39in**	When read, this gives the external state of the GPIO(39) pin.
29	**gp39out**	When the GPIO(39) pin is an output, this is the data driven on to it.
28	**gp39enable**	When set, the GPIO(39) pin is enabled as an output.
26	**gp38in**	When read, this gives the external state of the GPIO(38) pin.
25	**gp38out**	When the GPIO(38) pin is an output, this is the data driven on to it.
24	**gp38enable**	When set, the GPIO(38) pin is enabled as an output.
22	**gp37in**	When read, this gives the external state of the GPIO(37) pin.
21	**gp37out**	When the GPIO(37) pin is an output, this is the data driven on to it.
20	**gp37enable**	When set, the GPIO(37) pin is enabled as an output.
18	**gp36in**	When read, this gives the external state of the GPIO(36) pin.
17	**gp36out**	When the GPIO(36) pin is an output, this is the data driven on to it.
16	**gp36enable**	When set, the GPIO(36) pin is enabled as an output.
14	**gp35in**	When read, this gives the external state of the GPIO(35) pin.
13	**gp35out**	When the GPIO(35) pin is an output, this is the data driven on to it.
12	**gp35enable**	When set, the GPIO(35) pin is enabled as an output.
10	**gp34in**	When read, this gives the external state of the GPIO(34) pin.
9	**gp34out**	When the GPIO(34) pin is an output, this is the data driven on to it.
8	**gp34enable**	When set, the GPIO(34) pin is enabled as an output.
6	**gp33in**	When read, this gives the external state of the GPIO(33) pin.
5	**gp33out**	When the GPIO(33) pin is an output, this is the data driven on to it.
4	**gp33enable**	When set, the GPIO(33) pin is enabled as an output.
2	**gp32in**	When read, this gives the external state of the GPIO(32) pin.
1	**gp32out**	When the GPIO(32) pin is an output, this is the data driven on to it.
0	**gp32enable**	When set, the GPIO(32) pin is enabled as an output.

gpioAt4 General Purpose IO Pin Atomic Control 4

```
$0039
Write Only
```

This register allows GPIO pins 32-39 to be controlled independently by multiple processes, by allowing them to be modified atomically. If the bits corresponding to a particular GPIO pin are all set to zero, it is not modified. If a bit is set, then it is used to either set or clear a GPIO control bit.

For all bits, set has priority over clear, if both commands are given.

Bit	Name	Description
31	**gp39Dset**	The GPIO(39) data output bit (**gp39out**) is set by this bit.
30	**gp39Dclr**	The GPIO(39) data output bit (**gp39out**) is cleared by this bit.
29	**gp39Eset**	The GPIO(39) output enable bit (**gp39enable**) is set by this bit.
28	**gp39Eclr**	The GPIO(39) output enable bit (**gp39enable**) is cleared by this bit.
27	**gp38Dset**	The GPIO(38) data output bit (**gp38out**) is set by this bit.
26	**gp38Dclr**	The GPIO(38) data output bit (**gp38out**) is cleared by this bit.
25	**gp38Eset**	The GPIO(38) output enable bit (**gp38enable**) is set by this bit.
24	**gp38Eclr**	The GPIO(38) output enable bit (**gp38enable**) is cleared by this bit.
23	**gp37Dset**	The GPIO(37) data output bit (**gp37out**) is set by this bit.
22	**gp37Dclr**	The GPIO(37) data output bit (**gp37out**) is cleared by this bit.
21	**gp37Eset**	The GPIO(37) output enable bit (**gp37enable**) is set by this bit.
20	**gp37Eclr**	The GPIO(37) output enable bit (**gp37enable**) is cleared by this bit.
19	**gp36Dset**	The GPIO(36) data output bit (**gp36out**) is set by this bit.
18	**gp36Dclr**	The GPIO(36) data output bit (**gp36out**) is cleared by this bit.
17	**gp36Eset**	The GPIO(36) output enable bit (**gp36enable**) is set by this bit.
16	**gp36Eclr**	The GPIO(36) output enable bit (**gp36enable**) is cleared by this bit.
15	**gp35Dset**	The GPIO(35) data output bit (**gp35out**) is set by this bit.
14	**gp35Dclr**	The GPIO(35) data output bit (**gp35out**) is cleared by this bit.
13	**gp35Eset**	The GPIO(35) output enable bit (**gp35enable**) is set by this bit.
12	**gp35Eclr**	The GPIO(35) output enable bit (**gp35enable**) is cleared by this bit.
11	**gp34Dset**	The GPIO(34) data output bit (**gp34out**) is set by this bit.
10	**gp34Dclr**	The GPIO(34) data output bit (**gp34out**) is cleared by this bit.
9	**gp34Eset**	The GPIO(34) output enable bit (**gp34enable**) is set by this bit.
8	**gp34Eclr**	The GPIO(34) output enable bit (**gp34enable**) is cleared by this bit.
7	**gp33Dset**	The GPIO(33) data output bit (**gp33out**) is set by this bit.
6	**gp33Dclr**	The GPIO(33) data output bit (**gp33out**) is cleared by this bit.
5	**gp33Eset**	The GPIO(33) output enable bit (**gp33enable**) is set by this bit.
4	**gp33Eclr**	The GPIO(33) output enable bit (**gp33enable**) is cleared by this bit.
3	**gp32Dset**	The GPIO(32) data output bit (**gp32out**) is set by this bit.
2	**gp32Dclr**	The GPIO(32) data output bit (**gp32out**) is cleared by this bit.
1	**gp32Eset**	The GPIO(32) output enable bit (**gp32enable**) is set by this bit.
0	**gp32Eclr**	The GPIO(32) output enable bit (**gp32enable**) is cleared by this bit.

gpioSpec General Purpose IO Pin Special Function Control 1

```
$0023
Read / Write
```

This register re-assigns GPIO pins to special functions, as follow:

Bit	Name	Description
31	**gp15spec**	When set, the GPIO(15) function is overridden, and this pin becomes the function selected by **gp15mode**.
30	**gp15mode**	When gp15spec is set, then if this bit is set the GPIO(15) pin becomes UAE, if clear it is SYSA(31)
29	**gp14spec**	When set, the GPIO(14) function is overridden, and this pin becomes the function selected by **gp14mode**.
28	**gp13spec**	When set, the GPIO(13) function is overridden, and this pin becomes the function selected by **gp13mode**.
27	**gp12spec**	When set, the GPIO(12) function is overridden, and this pin becomes SYSA(28). This bit is ignored if **gp12isBusy** is set.
26	**gp11spec**	When set, the GPIO(11) function is overridden, and this pin becomes SYSA(27). This bit is ignored if **gp7to11sio** or **gp11mode** is set.
25	**gp10spec**	When set, the GPIO(10) function is overridden, and this pin becomes SYSA(26). This bit is ignored if **gp7to11sio** or **gp10mode** is set.
24	**gp9spec**	When set, the GPIO(9) function is overridden, and this pin becomes SYSA(25). This bit is ignored if **gp7to11sio** is set.
23	**gp8spec**	When set, the GPIO(8) function is overridden, and this pin becomes TSIZ(1). This bit is ignored if **gp7to11sio** is set.
22	**gp7spec**	When set, the GPIO(7) function is overridden, and this pin becomes TSIZ(0). This bit is ignored if **gp7to11sio** is set.
21	**gp14mode**	When **gp14spec** is set, then if this bit is set the GPIO(14) pin becomes PWM(1), if clear it is SYSA(30)
20	**gp13mode**	When **gp13spec** is set, then if this bit is set the GPIO(13) pin becomes PWM(0), if clear it is SYSA(29)
19	**gp7to11sio**	When **gp7to11sio** is set, GPIO pins 7 to 11 become SIO channel A. This will over-ride all other settings for these bits: **gp11spec**, **gp10spec**, **gp9spec**, **gp8spec**, **gp7spec**, **gp11mode** and **gp10mode**. *This function is available on Aries 3 and later versions only.*
18-11	**unused**	Write zeroes.
10	**gp12isBusy**	When this bit is set GPIO(12) is used as the BUSY input for NAND flash attached to the System Bus. This over-rides **gp12spec**. *This function is available on Aries 3 and later versions only.*
9	**gp1isSdat3**	When **gp1isSdat3** is set, along with **gp1spec**, GPIO(1) becomes the fourth I2S serial audio data channel, SDAT[3]. *This function is available on Aries 3 and later versions only.*
8	**gp11mode**	When set, the GPIO(11) function is overridden, and this pin becomes the secondary Serial Peripheral Bus data pin. This over-rides **gp11spec**. This bit is ignored if **gp7to11sio** is set. *This function is available on Aries 2 and later versions only.*
7	**gp10mode**	When set, the GPIO(10) function is overridden, and this pin becomes the secondary Serial Peripheral Bus clock pin. This over-rides **gp10spec**. This bit is ignored if **gp7to11sio** is set. *This function is available on Aries 2 and later versions only.*
6	**gp6spec**	When set, the GPIO(6) function is overridden, and this pin becomes the audio input channel 2 word flag.
5	**gp5spec**	When set, the GPIO(5) function is overridden, and this pin becomes the audio input channel 2 bit clock.
4	**gp4spec**	When set, the GPIO(4) function is overridden, and this pin becomes the audio input channel 2 data.
3	**gp3spec**	When set, the GPIO(3) function is overridden, and this pin becomes the Serial Peripheral Bus data pin.

Bit	Name	Description
2	**gp2spec**	When set, the GPIO(2) function is overridden, and this pin becomes the Serial Peripheral Bus clock pin.
1	**gp1spec**	When set, the GPIO(1) function is overridden, and this pin becomes the audio input high rate clock output.
0	**gp0spec**	When set, the GPIO(0) function is overridden, and this pin becomes the external host processor interrupt output.

gpioSpec2 General Purpose IO Pin Special Function Control 2

```
$002D
Read / Write
```

This register re-assigns GPIO pins to special functions, as follows. *This register is available on Aries 3 and later versions only.*

Bit	Name	Description
31	**gp35spec**	When set, the SYSA[24] function is overridden, and this pin becomes GPIO[35].
30	**gp34spec**	When set, the SYSA[23] function is overridden, and this pin becomes GPIO[34].
29	**gp33spec**	When set, the SYSA[22] function is overridden, and this pin becomes GPIO[33]. This bit is ignored if **gp29to33sio** is set.
28	**gp32spec**	When set, the SYSA[21] function is overridden, and this pin becomes GPIO[32]. This bit is ignored if **gp29to33sio** is set.
27	**gp31spec**	When set, the SYSA[20] function is overridden, and this pin becomes GPIO[31]. This bit is ignored if **gp29to33sio** is set.
26	**gp30spec**	When set, the SYSA[19] function is overridden, and this pin becomes GPIO[30]. This bit is ignored if **gp29to33sio** is set.
25	**gp29spec**	When set, the SYSA[18] function is overridden, and this pin becomes GPIO[29]. This bit is ignored if **gp29to33sio** is set.
24	**gp29to33sio**	When **gp29to33sio** is set, GPIO pins 29 to 33 become SIO channel B. This will over-ride all other settings for these bits: **gp33spec**, **gp32spec**, **gp31spec**, **gp30mode** and **gp29mode**.
23-0	**unused**	Reserved, set to zero.

gpioInt1 General Purpose IO Pin Interrupt Control

```
$0024
Read / Write
```

These two registers control the GPIO interrupt. Any of GPIO 0-15 pins may be designated as an interrupt source. Interrupts can be independently generated by a low level or falling edge, or by a high level or rising edge.

Each GPIO pin has the following control bits:

- an enable, which allows that pin to generate interrupts
- a polarity flag which indicates that the interrupts are either active high (rising edge) or active low (falling edge)
- an edge flag which makes them edge sensitive instead of level sensitive

- an interrupt flag that indicates that the corresponding GPIO pin was the source of the interrupt. If you write a one to the interrupt latch it clears, so that it is ready to accept another interrupt condition.

Note that you should not perform the clear operation until the external interrupt source has been cleared if the interrupt is level sensitive.

Note also that when performing a read modify write operation on this register you should service all the interrupt sources, because reading then writing back a set interrupt latch will clear it.

Bit	Name	Description
31	**gp15iena**	Enables interrupts from the GPIO(15) pin
30	**gp15pol**	When clear, the interrupt from GPIO(15) is triggered by a rising edge or high level; when set it is triggered by a falling edge or low level.
29	**gp15edge**	When set, the interrupt from GPIO(15) is edge sensitive, when clear it is level sensitive.
28	**gp15int**	Reading a one on this bit indicates that the source of the interrupt was GPIO(15). Writing back a one clears the interrupt, writing a zero has no effect.
27	**gp14iena**	Enables interrupts from the GPIO(14) pin
26	**gp14pol**	When clear, the interrupt from GPIO(14) is triggered by a rising edge or high level; when set it is triggered by a falling edge or low level.
25	**gp14edge**	When set, the interrupt from GPIO(14) is edge sensitive, when clear it is level sensitive.
24	**gp14int**	Reading a one on this bit indicates that the source of the interrupt was GPIO(14). Writing back a one clears the interrupt, writing a zero has no effect.
23	**gp13iena**	Enables interrupts from the GPIO(13) pin
22	**gp13pol**	When clear, the interrupt from GPIO(13) is triggered by a rising edge or high level; when set it is triggered by a falling edge or low level.
21	**gp13edge**	When set, the interrupt from GPIO(13) is edge sensitive, when clear it is level sensitive.
20	**gp13int**	Reading a one on this bit indicates that the source of the interrupt was GPIO(13). Writing back a one clears the interrupt, writing a zero has no effect.
19	**gp12iena**	Enables interrupts from the GPIO(12) pin
18	**gp12pol**	When clear, the interrupt from GPIO(12) is triggered by a rising edge or high level; when set it is triggered by a falling edge or low level.
17	**gp12edge**	When set, the interrupt from GPIO(12) is edge sensitive, when clear it is level sensitive.
16	**gp12int**	Reading a one on this bit indicates that the source of the interrupt was GPIO(12). Writing back a one clears the interrupt, writing a zero has no effect.
15	**gp11iena**	Enables interrupts from the GPIO(11) pin
14	**gp11pol**	When clear, the interrupt from GPIO(11) is triggered by a rising edge or high level; when set it is triggered by a falling edge or low level.
13	**gp11edge**	When set, the interrupt from GPIO(11) is edge sensitive, when clear it is level sensitive.
12	**gp11int**	Reading a one on this bit indicates that the source of the interrupt was GPIO(11). Writing back a one clears the interrupt, writing a zero has no effect.
11	**gp10iena**	Enables interrupts from the GPIO(10) pin
10	**gp10pol**	When clear, the interrupt from GPIO(10) is triggered by a rising edge or high level; when set it is triggered by a falling edge or low level.
9	**gp10edge**	When set, the interrupt from GPIO(10) is edge sensitive, when clear it is level sensitive.
8	**gp10int**	Reading a one on this bit indicates that the source of the interrupt was

Bit	Name	Description
		GPIO(10). Writing back a one clears the interrupt, writing a zero has no effect.
7	**gp9iena**	Enables interrupts from the GPIO(9) pin
6	**gp9pol**	When clear, the interrupt from GPIO(9) is triggered by a rising edge or high level; when set it is triggered by a falling edge or low level.
5	**gp9edge**	When set, the interrupt from GPIO(9) is edge sensitive, when clear it is level sensitive.
4	**gp9int**	Reading a one on this bit indicates that the source of the interrupt was GPIO(9). Writing back a one clears the interrupt, writing a zero has no effect.
3	**gp8iena**	Enables interrupts from the GPIO(8) pin
2	**gp8pol**	When clear, the interrupt from GPIO(8) is triggered by a rising edge or high level; when set it is triggered by a falling edge or low level.
1	**gp8edge**	When set, the interrupt from GPIO(8) is edge sensitive, when clear it is level sensitive.
0	**gp8int**	Reading a one on this bit indicates that the source of the interrupt was GPIO(15). Writing back a one clears the interrupt, writing a zero has no effect.

gpioInt2 General Purpose IO Pin Interrupt control

```
$0025
Read / Write
```

See above for a description of this register.

Bit	Name	Description
31	**gp7iena**	Enables interrupts from the GPIO(7) pin
30	**gp7pol**	When clear, the interrupt from GPIO(7) is triggered by a rising edge or high level; when set it is triggered by a falling edge or low level.
29	**gp7edge**	When set, the interrupt from GPIO(7) is edge sensitive, when clear it is level sensitive.
28	**gp7int**	Reading a one on this bit indicates that the source of the interrupt was GPIO(7). Writing back a one clears the interrupt, writing a zero has no effect.
27	**gp6iena**	Enables interrupts from the GPIO(6) pin
26	**gp6pol**	When clear, the interrupt from GPIO(6) is triggered by a rising edge or high level; when set it is triggered by a falling edge or low level.
25	**gp6edge**	When set, the interrupt from GPIO(6) is edge sensitive, when clear it is level sensitive.
24	**gp6int**	Reading a one on this bit indicates that the source of the interrupt was GPIO(6). Writing back a one clears the interrupt, writing a zero has no effect.
23	**gp5iena**	Enables interrupts from the GPIO(5) pin
22	**gp5pol**	When clear, the interrupt from GPIO(5) is triggered by a rising edge or high level; when set it is triggered by a falling edge or low level.
21	**gp5edge**	When set, the interrupt from GPIO(5) is edge sensitive, when clear it is level sensitive.
20	**gp5int**	Reading a one on this bit indicates that the source of the interrupt was GPIO(5). Writing back a one clears the interrupt, writing a zero has no effect.
19	**gp4iena**	Enables interrupts from the GPIO(4) pin
18	**gp4pol**	When clear, the interrupt from GPIO(4) is triggered by a rising edge or high level; when set it is triggered by a falling edge or low level.
17	**gp4edge**	When set, the interrupt from GPIO(4) is edge sensitive, when clear it is level sensitive.
16	**gp4int**	Reading a one on this bit indicates that the source of the interrupt was GPIO(4). Writing back a one clears the interrupt, writing a zero has no effect.

Bit	Name	Description
15	**gp3iena**	Enables interrupts from the GPIO(3) pin
14	**gp3pol**	When clear, the interrupt from GPIO(3) is triggered by a rising edge or high level; when set it is triggered by a falling edge or low level.
13	**gp3edge**	When set, the interrupt from GPIO(3) is edge sensitive, when clear it is level sensitive.
12	**gp3int**	Reading a one on this bit indicates that the source of the interrupt was GPIO(3). Writing back a one clears the interrupt, writing a zero has no effect.
11	**gp2iena**	Enables interrupts from the GPIO(2) pin
10	**gp2pol**	When clear, the interrupt from GPIO(2) is triggered by a rising edge or high level; when set it is triggered by a falling edge or low level.
9	**gp2edge**	When set, the interrupt from GPIO(2) is edge sensitive, when clear it is level sensitive.
8	**gp2int**	Reading a one on this bit indicates that the source of the interrupt was GPIO(2). Writing back a one clears the interrupt, writing a zero has no effect.
7	**gp1iena**	Enables interrupts from the GPIO(1) pin
6	**gp1pol**	When clear, the interrupt from GPIO(1) is triggered by a rising edge or high level; when set it is triggered by a falling edge or low level.
5	**gp1edge**	When set, the interrupt from GPIO(1) is edge sensitive, when clear it is level sensitive.
4	**gp1int**	Reading a one on this bit indicates that the source of the interrupt was GPIO(1). Writing back a one clears the interrupt, writing a zero has no effect.
3	**gp0iena**	Enables interrupts from the GPIO(0) pin
2	**gp0pol**	When clear, the interrupt from GPIO(0) is triggered by a rising edge or high level; when set it is triggered by a falling edge or low level.
1	**gp0edge**	When set, the interrupt from GPIO(0) is edge sensitive, when clear it is level sensitive.
0	**gp0int**	Reading a one on this bit indicates that the source of the interrupt was GPIO(0). Writing back a one clears the interrupt, writing a zero has no effect.

gpioVin General Purpose Inputs from the Video Input Pins

```
$002A
Read Only
```

This location allows the video-input port to be used instead as a set of general-purpose inputs. Its sole function is to allow the state of those inputs to be read.

Bit	Name	Description
31-24	**vid**	The video input data bus pins.
23	**viclk**	The video input clock pin.

System Bus Control

The System Bus interface itself is described in the System Bus section of this documentation

sysCtrl System Bus Control

```
$0030
Read / Write
```

Refer to the System Bus section of this document for details.

sysMemctl System Bus Memory Control

```
$0031
Read / Write
```

Refer to the System Bus section of this document for details.

sysSdramCtrl System Bus SDRAM Control

```
$0032
Read / Write
```

Refer to the System Bus section of this document for details.

PWM Output Control

Two Pulse Width Modulated (PWM) outputs may be independently enabled on GPIO pins 13 and 14, if required. These registers control that functionality. There are two independent PWM output channels, which behave like each other.

pwm0 PWM Channel 0 Control

```
$0040
Read / Write
```

This register allows the low and high time to be separately programmed. This allows both the duty cycle and the frequency to be controlled.

To set the output to be fixed high, you should set zero in the low time and a non-zero in the high time. To set the output to be fixed low, then the high time should be zero, and the low time non-zero. If both values are set to zero, then the output will stay at its previous value.

Bit	Description
31-27	Reserved, write zero
16-16	Output high time in 54 MHz clock cycles. The value written here should be one less than the desired high period.
15-11	Reserved, write zero
10-0	Output low time in 54 MHz clock cycles. The value written here should be one less than the desired low period.

pwm1 PWM Channel 1 Control

```
$0041
Read / Write
```

This register allows the low and high time to be separately programmed. This allows both the duty cycle and the frequency to be controlled.

To set the output to be fixed high, you should set zero in the low time and a non-zero in the high time. To set the output to be fixed low, then the high time should be zero, and the low time non-zero. If both values are set to zero, then the output will stay at its previous value.

Bit	Description
31-27	Reserved, write zero
16-16	Output high time in 54 MHz clock cycles. The value written here should be one less than the

	desired high period.
15-11	Reserved, write zero
10-0	Output low time in 54 MHz clock cycles. The value written here should be one less than the desired low period.

Power On Configuration

config Power On Configuration

```
$0060
Read Only
```

Reflects the eight power-on configuration bits.

Bit	Description
31-8	Reserved, ignore
7-0	Power on configuration bits.

Communication Bus Configuration

commMpe0 MPE0 Communication Bus Address

```
$0100
Write Only
```

Communication Bus address for MPE0, the valid range is 0-63 and all the MPEs must be unique. At reset, this defaults to zero.

Bit	Description
31-6	Reserved, write zero
5-0	Communication Bus address of MPE0

commMpe1 MPE1 Communication Bus Address

```
$0101
Write Only
```

Communication Bus address for MPE1, the valid range is 0-63 and all the MPEs must be unique. At reset, this defaults to one.

Bit	Description
31-6	Reserved, write zero
5-0	Communication Bus address of MPE1

commMpe2 MPE2 Communication Bus Address

```
$0102
Write Only
```

Communication Bus address for MPE2, the valid range is 0-63 and all the MPEs must be unique. At reset, this defaults to two.

Bit	Description
31-6	Reserved, write zero
5-0	Communication Bus address of MPE2

commMpe3 MPE3 Communication Bus Address

```
$0103
Write Only
```

Communication Bus address for MPE3, the valid range is 0-63 and all the MPEs must be unique. At reset, this defaults to three.

Bit	Description
31-6	Reserved, write zero
5-0	Communication Bus address of MPE3

commMpe4-31 MPE4-31 Communication Bus Addresses

```
$0104 - $011F
Write Only
```

Reserved for MPEs 4-31. One day life will be this good.

ctrlXXX Controller Interface registers

```
$0200 - $02FF
Write Only
```

See the controller interface section below.

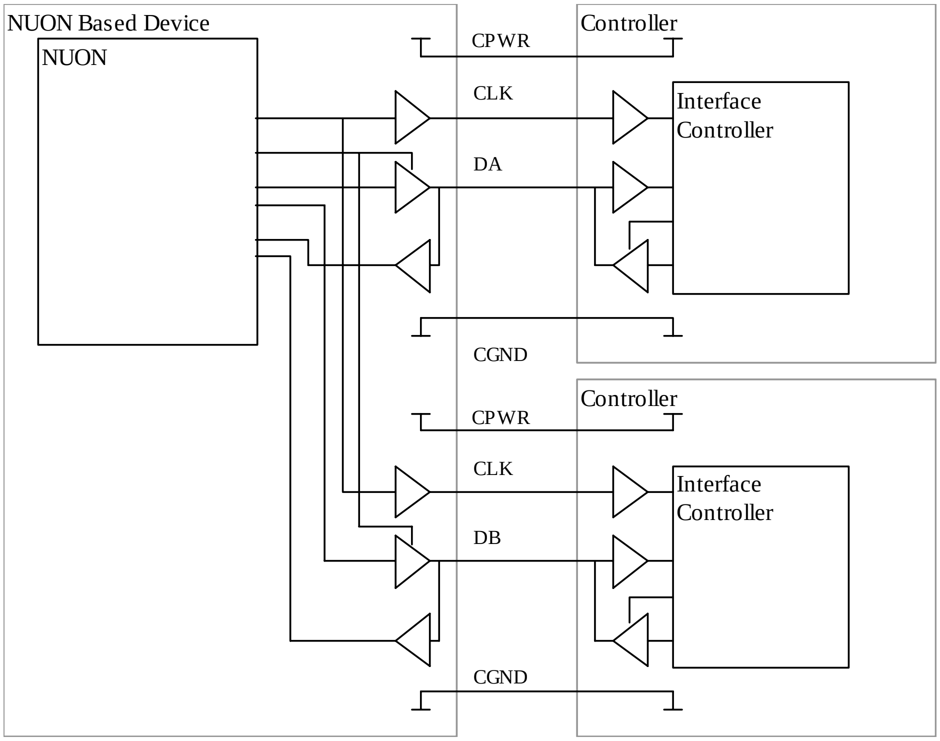
CONTROLLER INTERFACE

Controllers are devices for user interaction with NUON. The controller interface can support a range of these, and will also support storage devices such as game-save and high-score memory cards, and communications devices. Possible ‘controller’ devices on these ports are:

- Game-pad controllers
- Joysticks
- Keyboards
- Mice
- Memory cards
- 3D Glasses (e.g. LCD shutters)
- Modems
- Simple network interfaces

The controller interface itself is a four-pin connector. These pins include power, ground, clock output, and bi-directional data. The protocol over this interface is a synchronous serial bit-stream. The NUON system is the master device that generates the clock and issues commands to individual slave devices, which may then respond. The data transfer rate over this interface is programmable to match the electrical capabilities of the external hardware.

The NUON system has two such ports, and a number of devices may be connected in parallel on each of them.



Protocol

The interface transmits thirty-four bit bursts of data, and expects responses on up to thirty-four bits. The first bit is always set to one, and acts as a start bit to signal the framing of the following bits. The second bit of transmitted data is intended to signal command or data, and the remaining thirty-two bits are programmable.

All receive data packets contain a payload of up to thirty-two data bits, and like transmit packets are preceded by a start bit and a status bit. If the status bit is set, that is intended to indicate that the response is null, and this may be used by the controller interface to signal that it is unable to respond to a command.

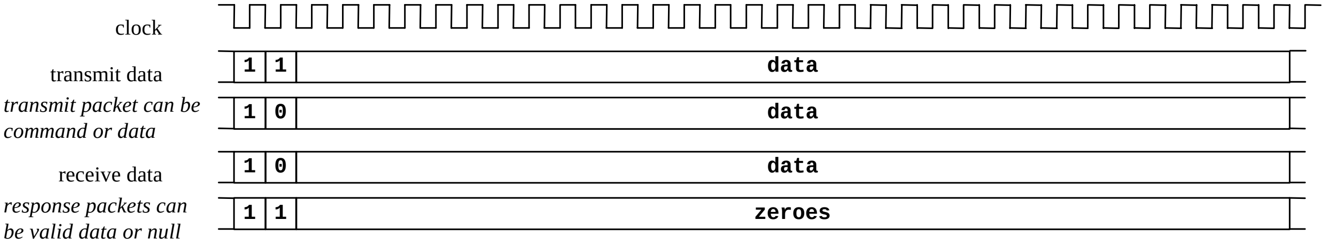

clock
transmit data
1 1 data
transmit packet can be command or data
1 0 data
receive data
1 0 data
response packets can be valid data or null
1 1 zeroes

Figure 8 – Controller Packet Formats

Note that transmitted data changes on the falling edge of clock and should be sampled on the rising edge, because otherwise clock and data can race. The rising edge of receive data is used to recognize the framing of receive data, which is synchronous to the transmit clock but has no defined phase relationship with it.

Because the tri-state enable is shared between port 1 and port 2, port 2 is slaved to port 1. This means that data is transmitted on **both** ports whenever a command or data word is sent on port 1. Therefore, whenever you want to transmit on either port, you should always set up port 2 to transmit, then set up port 1. This implies that you may sometimes have to send some form of null data on one of the ports.

Controller Interface Control Registers

The controller interface registers are part of the miscellaneous IO controller module, which is programmed over the Communication Bus. This interface will normally receive Communication Bus packets at any time, unless it is waiting to transmit a response packet to a read command. The controller interface can also send unsolicited packets if the appropriate enable bits are set. See the control and status register descriptions below.

Long word 0 of a Communication Bus packet from the miscellaneous IO controller module normally contains zero. However, if the packet is received data from a controller interface being automatically forwarded, long word 0 contains the controller number. Long word 1, as usual, will contain the read data.

Sending it to one of the controller command registers, along any associated data transmits a command.

If electrical interference is generated it may corrupt a data transfer in progress, so any data that is sensitive to errors should be protected with a parity flag or a better error detection method. Allowing the interface to go idle for thirty-four cycles will recover from a framing error. If this does not recover an erroneous condition, the reset sequence may have to be performed.

Controller interface registers are in the general IO section described below. The control registers are:

ctrlCmd1 Send a Command on Controller Port 1

```
$0200
Write only
```

Writing to this register sends a command long word to controller port 1. This also causes a transmit on port 2.

Bit	Description
31-0	Controller command.

ctrlSData1 Send Data on Controller Port 1

```
$0201
Write only
```

Writing to this register sends a data long word to controller port 1. This also causes a transmit on port 2.

Bit	Description
31-0	Controller data.

ctrlStat1 Control and read the status of Controller Port 1

```
$0202
Read/Write
```

A read from this register gives the port status. Response format:

Bit	Name	Description
31	**txFull**	Transmit buffer full (read only).
30	**rxFull**	Receive buffer full (read only).
29	**rxNull**	A flag that the last data received was a null response.
28	**reFrame**	Writing a one to this bit sends at least 34 bits of zero. This will allow the system to recover framing, and should always be performed after an error occurs. While this is happening, the transmit buffer will appear full.
27	**reset**	Writing to this register sends at least 34 bits of one. This will reset all the controllers. While this is happening, the transmit buffer will appear full.
24	**commSend**	When set, received data is transmitted on the Communication Bus to the target identified below. When clear, the data will await polling. This bit is shared with controller 2.
22-16	**commID**	Target Communication Bus ID for the function above. These bits are shared with controller 2.
15	**rxOverflow**	Receiver overflow error. A long word was received before the previous one had been read or transmitted over the Communication Bus. This bit is 'sticky' and will stay set until a one is written to it.
11-0	**preScale**	This register controls the clock pre-scale counter that determines the controller bus clock rate. This is a twelve-bit value, where the half clock period of the output clock is given by the number of system clock cycles programmed here. These bits are shared with controller 2.

ctrlRData1 Read Data from Controller Port 1

```
$0203
Read only
```

Reading from this register reads a data long word from controller port 1.

Bit	Description
31-0	Controller data.

ctrlCmd2 Send a Command on Controller Port 2

```
$0210
Write only
```

Writing to this register sends a command long word to controller port 2. Note that this will not be sent until a command or data is written to port 1, as port 2 is the slave.

Bit	Description
31-0	Controller command.

ctrlSData2 Send Data on Controller Port 2

```
$0211
Write only
```

Writing to this register sends a data long word to controller port 2. Note that this will not be sent until a command or data is written to port 1, as port 2 is the slave.

Bit	Description
31-0	Controller data.

ctrlStat2 Control and read the status of Controller Port 2

```
$0212
Read/Write
```

A read from this register gives the port status. Response format:

Bit	Name	Description
31	**txFull**	Transmit buffer full (read only).
30	**rxFull**	Receive buffer full (read only).
29	**rxNull**	A flag that the last data received was a null response.
28	**reFrame**	Writing a one to this bit sends at least 34 bits of zero. This will allow the system to recover framing, and should always be performed after an error occurs. While this is happening, the transmit buffer will appear full.
27	**reset**	Writing to this register sends at least 34 bits of one. This will allow guarantee to reset all the controllers. While this is happening, the transmit buffer will appear full.
24	**commSend**	When set, received data is transmitted on the Communication Bus to the target identified below. When clear, the data will await polling. This bit is shared with controller 1.
22-16	**commID**	Target Communication Bus ID for the function above. These bits are shared with controller 1.
15	**rxOverflow**	Receiver overflow error. A long word was received before the previous one had been read or transmitted over the Communication Bus. This bit is 'sticky' and will stay set until a one is written to it.
11-0	**preScale**	This register controls the clock pre-scale counter, that determines the controller bus clock rate. This is a twelve-bit value, where the half clock period of the output clock is given by the number of system clock cycles programmed here. These bits are shared with controller 1.

ctrlRData2 Read Data from Controller Port 2

```
$0213
Read only
```

Reading from this register reads a data long word from controller port 2.

Bit	Description
31-0	Controller data.

SERIAL PERIPHERAL BUS INTERFACE

The NUON system has a hardware implementation of a standard serial peripheral device bus. It is a multi-master synchronous serial bus running at up to 400 Kbits/second. It is used for communicating with various peripheral devices.

The Serial Peripheral Bus is able to communicate with I^2C bus devices if that bus protocol is followed. The specification for the I^2C bus is published by Philips and is available on their Web site.

There are two quite distinct parts to the NUON Serial Peripheral Bus interface: the master device and the slave device. The master device generates a clock and addresses slave devices, and may read or write data within a transfer. The slave part allows data to be read or written by some other master.

The master is programmed by setting up a series of byte transfers, either reads or writes, which may be optionally qualified by a preceding start condition or by a succeeding stop condition.

The Serial Peripheral Bus can interrupt the MPEs to flag either that a master transfer has completed or that a slave transfer has occurred.

Glitches on either the clock or data lines are rejected by the hardware up to a maximum duration of three clock cycles, i.e. 54ns.

Serial Peripheral Bus Master

The bus master can transmit start codes, transmit or receive bytes, and transmit stop codes. The formatting of a bus master transfer is up to software, and should conform to the protocol being expected by the addressed slave. Up to an 8 byte transfer can be programmed with a single command, and you can perform longer transfers by issuing consecutive commands.

The interface can run at a maximum rate of up to either 100 Kbits/sec or 400 Kbits/sec, according to its mode setting. It will respect clock stretching by slave devices if applied, and can detect bus arbitration failure in a multi-master environment. It also detects start and stop codes issued by other masters, and will not attempt to use the bus while it is owned by another master.

If a transfer fails because a byte is not acknowledged correctly, it will properly terminate the transfer, by issuing a stop command, and flag the error condition.

The bus master will generate an interrupt to the MPEs when it either completes the programmed transfer, or encounters an error condition (this interrupt is combined with the slave interrupt).

Serial Peripheral Bus Slave

The slave interface can recognize its own 7-bit address, and can perform read or write transfers. These will normally be 4 bytes long, to and from holding registers within the interface. The slave interface can also be programmed to perform longer transfers by setting its hold control bit. In this mode the clock is held low if either the receive-buffer fills or the transmit-buffer empties during a slave transfer. This clock hold condition is cleared when software intervenes either to supply more transmit data or read the received data, or to terminate the sequence.

When the transmitter empties or the receiver fills an interrupt is generated to the NUON software. You can also poll to test for his condition.

Slave behavior for address mismatch

This diagram shows the behavior of the slave when the address does not match its internally programmed address. The slave will take no part in the bus transaction, so the master will not see an acknowledge, unless another slave acknowledges the transfer.

Master	Slave	Clocks	Description
Start code			
Slave address		7	must match that programmed into the slaveAddress field of the spbSlaveStatus register described below
Read/not write		1	Must be a 1 for a read
	No action (Nack)	1	Slave will not acknowledge if the address does not match.
Stop code			

Slave behavior for a read of a long-word (4 bytes)

This table shows the behavior of the slave for a read transfer. The 7-bit address must match that programmed into the slaveAddress field of the spbSlaveStatus register described below. The master should read 4 bytes then terminate the transfer.

Master	Slave	Clocks	Description
Start code			
Slave address		7	must match that programmed into the slaveAddress field of the spbSlaveStatus register described below
Read/not write		1	Must be a 1 for a read
	Ack	1	Slave will acknowledge if the address matches
	Data bits 31-24	8	First byte of long-word
Ack		1	Master must acknowledge byte
	Data bits 23-16	8	Second byte of long-word
Ack		1	Master must acknowledge byte
	Data bits 15-8	8	Third byte of long-word
Ack		1	Master must acknowledge byte
	Data bits 7-0	8	Fourth byte of long-word
Nack		1	Master need not acknowledge last byte
Stop code			

Slave behavior for a read of two long-words (8 bytes)

This table shows the behavior of the slave for an extended read transfer. The slaveHold bit must be set, and the 7-bit address must match that programmed into the slaveAddress field of the spbSlaveStatus register described below. The master should read 8 bytes then terminate the transfer. This can be extended for longer transfers, in multiples of 4 bytes.

Note that the clock will be held low by the slave on the first data bit of each long-word (i.e. bit 31 of the first byte of the second long-word in this example), until NUON software supplies some more read data.

Master	Slave	Clocks	Description
Start code			
Slave address		7	must match that programmed into the slaveAddress field of the spbSlaveStatus register described below
Read/not write		1	Must be a 1 for a read
	Ack	1	Slave will acknowledge if the address matches

	Data bits 31-24	8	First byte of first long-word
Ack		1	Master must acknowledge byte
	Data bits 23-16	8	Second byte of first long-word
Ack		1	Master must acknowledge byte
	Data bits 15-8	8	Third byte of first long-word
Ack		1	Master must acknowledge byte
	Data bits 7-0	8	Fourth byte of first long-word
Ack		1	Master must acknowledge byte
	Data bits 31-24	8	First byte of second long-word
Ack		1	Master must acknowledge byte
	Data bits 23-16	8	Second byte of second long-word
Ack		1	Master must acknowledge byte
	Data bits 15-8	8	Third byte of second long-word
Ack		1	Master must acknowledge byte
	Data bits 7-0	8	Fourth byte of second long-word
Nack		1	Master need not acknowledge last byte
Stop code			

Slave behavior for a write of a long-word (4 bytes)

This table shows the behavior of the slave for a write transfer. The 7-bit address must match that programmed into the slaveAddress field of the spbSlaveStatus register described below. The master should write 4 bytes then terminate the transfer.

Master	**Slave**	**Clocks**	**Description**
Start code			
Slave address		7	must match that programmed into the slaveAddress field of the spbSlaveStatus register described below
Read/not write		1	Must be a 0 for a write
	Ack	1	Slave will acknowledge if the address matches
Data bits 31-24		8	First byte of long-word
	Ack	1	Slave will acknowledge byte
Data bits 23-16		8	Second byte of long-word
	Ack	1	Slave will acknowledge byte
Data bits 15-8		8	Third byte of long-word
	Ack	1	Slave will acknowledge byte
Data bits 7-0		8	Fourth byte of long-word
	Ack	1	Slave will acknowledge byte
Stop code			

Slave behavior for a write of two long-words (8 bytes)

This table shows the behavior of the slave for an extended write transfer. The slaveHold bit must be set, and the 7-bit address must match that programmed into the slaveAddress field of the spbSlaveStatus register described below. The master should write 8 bytes then terminate the transfer. This can be extended for longer transfers, in multiples of 4 bytes.

Note that the clock will be held low by the slave on the first data bit of each long-word (i.e. bit 31 of the first byte of the second long-word in this example), until NUON software clears the write data.

Master	**Slave**	**Clocks**	**Description**
Start code			
Slave address		7	must match that programmed into the slaveAddress

			field of the spbSlaveStatus register described below
Read/not write		1	Must be a 0 for a write
	Ack	1	Slave will acknowledge if the address matches
Data bits 31-24		8	First byte of first long-word
	Ack	1	Slave will acknowledge byte
Data bits 23-16		8	Second byte of first long-word
	Ack	1	Slave will acknowledge byte
Data bits 15-8		8	Third byte of first long-word
	Ack	1	Slave will acknowledge byte
Data bits 7-0		8	Fourth byte of first long-word
	Ack	1	Slave will acknowledge byte
Data bits 31-24		8	First byte of second long-word
	Ack	1	Slave will acknowledge byte
Data bits 23-16		8	Second byte of second long-word
	Ack	1	Slave will acknowledge byte
Data bits 15-8		8	Third byte of second long-word
	Ack	1	Slave will acknowledge byte
Data bits 7-0		8	Fourth byte of second long-word
	Ack	1	Slave will acknowledge byte
Stop code			

Serial Peripheral Bus Control Registers

The Serial Peripheral Bus interface is controlled over the Communication Bus. It has its own packet format so that a bus master transfer up to 8 bytes long can be programmed with a single Communication Bus packet.

The Communication Bus status bits, set in the MPE **comminfo** register are used to control the action of each Comm Bus packet. The top bit is a read write flag, i.e. $80 for reads and $00 for writes should be added to the addresses given below, and placed in the transmit status bits.

spbMasterCommand

```
$00
Read/Write
```

This command packet initiates a Serial Peripheral Bus master transaction of up to eight bytes. At the end of the transfer this may be read back again, and any read commands will have the read data in the corresponding field.

Bits	Description
122-120	Type for byte 0 transfer
119-112	Data for byte 0 transfer
106-104	Type for byte 1 transfer
103- 96	Data for byte 1 transfer
90- 88	Type for byte 2 transfer
87- 80	Data for byte 2 transfer
74- 72	Type for byte 3 transfer
71- 64	Data for byte 3 transfer
58- 56	Type for byte 4 transfer
55- 48	Data for byte 4 transfer
42- 40	Type for byte 5 transfer
39- 32	Data for byte 5 transfer

`26- 24`	Type for byte 6 transfer
`23- 16`	Data for byte 6 transfer
`10-  8`	Type for byte 7 transfer
`7-  0`	Data for byte 7 transfer

This corresponds to a 16-bit structure repeated 8 times for each byte. The first byte sent over the bus is byte 0, held in the top 16 bits of the packet, i.e. the top 16 bits of scalar 0 of the vector. The details of this structure are:

Bits	**Description**
`10-8`	Type field. This tells the Serial Peripheral Bus master how to deal with this byte in the transfer. Available types are: 0 **nop** do nothing - this will terminate any current master transfer 1 **xmitByte** transmit the corresponding byte 2 **recvByte** receive a byte and place it in the command buffer 3 **xmitStart** transmit a byte preceded by a start code 4 **xmitStop** transmit a byte followed by a stop code 5 **recvStop** receive a byte and buffer it, then transmit a stop code 6 **xmitStartR** transmit a byte preceded by a repeated start code
`7-0`	Data field. For transmit types this hold the data to be transmitted. For receive types the data received is placed in this field, and is available when the command packet is read back.

Note that the **xmitStartR** type is only implemented in Aries 3 and upwards. It allows the start code to be repeated in the middle of a master sequence. A master transaction that requires a repeated start code must use **xmitStart** at the beginning of the sequence, **xmitStartR** in the middle of it, and one of the two stop code forms at the end.

spbMasterStatus

```
$01
Read/Write
```

This register allows various aspects of the master’s operation to be controlled and observed. This is a 32-bit register, and is read and written in the low 32 bits of the Comm Bus packet, i.e. scalar 3 of the vector. This register must not be written to during a master transfer unless it is to set the abort command.

Bits	**Name**	**Description**
`31`	**masterABort**	Aborts the master’s current operation immediately. This may be set to one then zero to recover, should the bus hang for any reason. It may be necessary to transmit a stop code after this to clear the bus. This bit can also be set to completely disable the master.
`30`	**masterFastMode**	Sets the bus to run at up to 400 Kbits/sec. When this bit is clear the bus will run at up to 100 Kbits/sec.
`29`	**masterNoArb**	When this bit is set, multi-master arbitration is disabled. If the NUON is the only device on the bus, this might prevent false arbitration failures. If the bus is operating properly, this bit should never need to be set.
`28`	**masterNotEmpty**	This error flag indicates that a command packet was written while the previous command was still being processed. The newer packet is ignored, and this flag will be set. This flag should be cleared by software.

27	**masterNack**	This error flag indicates that acknowledge failure occurred during the master transfer. If a byte is not acknowledged by the slave, then the master will transmit a stop bit, end the current transfer, and set this bit. The controlling software should then take appropriate action, i.e. retry or give up. This flag should be cleared by software.
26	**masterArbf**	This error flag indicates that an arbitration failure occurred during a master transfer. An arbitration failure is detected if the data being written during a transmit byte is high, but the bus remains low on a rising clock edge. When this occurs the master transfer aborts immediately. This flag should be cleared by software.
3-0	**bytesToSend**	This read only counter shows where the master is in the command, 0 meaning that it is idle, and 8-1 being a down count as it works through the bytes of the command packet. You can write anything to these bits, it will not affect the counter.

spbSlaveTxData

$10
Read/Write

This 32-bit register holds the slave transmit data. Data is transmitted starting from bit 31. This register is read and written in the low 32 bits of the Comm Bus packet, i.e. scalar 3 of the vector.

spbSlaveRxData

$11
Read Only

This 32-bit register holds the slave receive data. Received data is stored here starting from bit 31. This register is read in the low 32 bits of the Comm Bus packet, i.e. scalar 3 of the vector.

spbSlaveStatus

$12
Read/Write

This register controls the slave's operation, and allows its status to be observed. This register is read in the low 32 bits of the Comm Bus packet, i.e. scalar 3 of the vector.

Bits	Name	Description
31	**slaveAbort**	Aborts the slave's current operation immediately. This may set to one then zero to recover should the bus hang for any reason. This bit can also be set to completely disable the slave.
30-24	**slaveAddress**	The seven bits give the Serial Peripheral Bus address that the slave responds to. The default is $42, being the answer to life, the universe and everything in hex.
23	**slaveRxFull**	This read-only flag is set when the receiver has received four bytes. Reading the slave receive data register will implicitly clear this bit.
22	**slaveHold**	This bit is used for slave transfers longer than 4 bytes. When either receive or transmit completes 4 bytes then the Serial Peripheral Bus clock is held low until software intervenes, either to transfer more data or to clear this bit which allows the bus to clear.
21	**slaveAlone**	This bit is used to separate the master and slave busses, so that the internal slave is connected to GPIO 2-3 and the internal master to GPIO 10-11. When this bit is clear the master and slave are

		combined and connected to GPIO 2-3. *Aries 2 and up only*.
`10-8`	**slaveTxCount**	This counter indicates how many bytes the slave transmitter has sent. It is set to zero when the transmit data register is written to.
`3-0`	**slaveRxCount**	This read-only counter indicates how many valid bytes have been received.

spbHysteresis

```
$20
Write Only
```

This register controls the rejection of noise on the I2C interface. The value is used for both clock and data inputs for both the master and slave. The hysteresis controls the length of time, in system (54 MHz) clock cycles, that a change on the input has to be stable for before it will be recognized.

Bits	Name	Description
`31-20`	**hysteresis**	This 12-bit value is the stable time required of an input change before it is passed on to the master and slave logic. The default value is 3. *Aries 2 and up only*.

DUAL SIO CHANNEL INTERFACE

Introduction

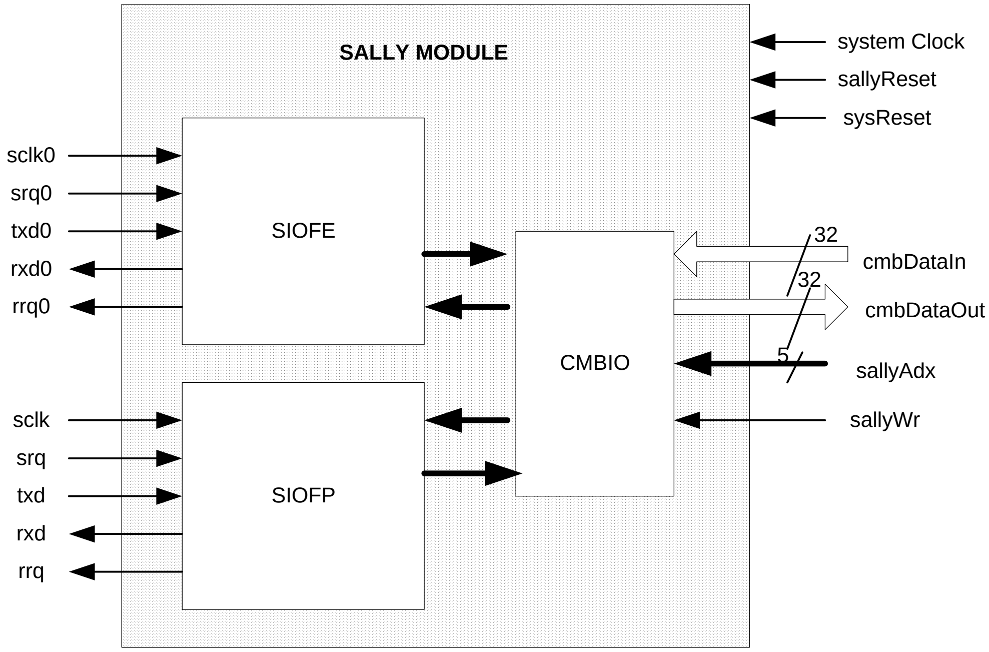
The dual SIO channel interface implements two serial IO channels for communication with external devices, usually micro-controllers. This feature is available in Aries 3 and upwards. Prior to Aries 3 it was implemented in an external ASIC code-named “Sally”.

The interface resides in the miscellaneous IO controller of the NUON chip. It communicates to the NUON chip through the Communications Bus to read and write commands to and from the external devices.

Note that it does not provide two discreet channels, as this interface is defined to fit a specific function.

Block Diagram

The following block diagram shows internal blocks of the interface:



The CMBIO module contains all the logic that communicates with the System Bus interface. It contains all the registers that are shown in the programmer’s model below. The address bus consists of 5 bits of word address (32-bits). See programmers model below for more detail.

The SIOFP and SIOFE modules perform the communication, and are identical.

Programmers Model

The registers described below are programmed over the Communication Bus. The miscellaneous IO controller has Communication Bus ID 69 (hex 45). The communication protocol is as follows for the command packet:

Long word	Description
0	0-15 register address 31 set for write, clear for read
1	0-31 write data if a write command
2	unused
3	unused

The response packet is returned if the operation was a read. Its format is:

Long word	Description
0	0-2 read status 16-31 register address of an IO read
1	0-31 read data
2	unused
3	unused

The following tables show the write and read IO registers

cmdArgs

```
$0080
Write Only
```

cmdArgs is a 32-bit register to send the argument to a command to the front end or font panel. The most significant byte is the first byte to be sent onto the bus, and then, depending on the total number of parameters, the rest of the arguments are also sent. **cmdId** will contain the actual command id. The parameters should be written first since the command will be sent out as soon as **cmdId** is written with the command id. There is a separate document that describes commands and encoding.

Bits	Name	Description
31-0	**cmdArgs**	4 bytes of command arguments

cmdId

```
$0081
Write Only
```

See the comments on **cmdArgs**.

Bits	Name	Description
31-0	**cmdId**	Command ID Register. Data sent when this is written.

cmdCount

```
$0082
Write Only
```

CmdCount is a 4-bit register used to indicate how many commands have to be sent out. If sending to the front-end micom then set bit 3 to zero and bits 2-0 to the exact number of bytes (including command id). If sending to the front panel micom then set bit 3 to ‘1’ and bits 2-0 to 000 if sending 1-byte or to 011 if sending 4 bytes.

Bits	Name	Description
3	**destination**	0: for Tosh: count means exact # of bytes 1: for Sanyo: count of 0 means 1 byte count of 3 means 4 bytes
2-0	**count**	Number of bytes of command to be sent

feRcvCnt

```
$0083
Write Only
```

FERcvCnt is a 1-bit register used to specify how many bytes to expect from the front-end micom. For a status command we expect 5 bytes, for a TOC command 10 bytes are returned.

Bits	Name	Description
0	**FERcvCnt**	0: 5 bytes 1: 10 bytes

fpMode

```
$0084
Write Only
```

Fpmode is a 4-bit register holding the mode bits to be sent to the front panel.

Bits	Name	Description
3-0	**Fpmode**	4 mode bits for front panel

ctrlReg Control and Status register

```
$0085
Read / Write
```

The CtrlReg is a 6-bit register with the following control bits:

Bits	Name	Description
5	**hShakeMode**	Setting this bit to 0 forces (power up state) forces Sally into handshake mode with the front panel. This means that Sally will not accept a new message from the front panel until the last message has been read out of Sally, i.e. the FPReceived bit has been cleared. This is used primarily at boot up since the front panel micom will send a series of unsolicited messages to NUON that need to be read and verified. In no handshake mode, the next incoming front panel message will overwrite the current one if not read from NUON.
4	**softReset**	Writing a ‘1’ to this location will reset the Sally chip to a power up state. This is for diagnostic purposes only.
3	**FPReceived**	This bit will be set when a message for NUON has been received by Sally from the front panel. Writing a 1 to this location will reset this bit.
2	**cmdSentFP**	This bit will be set when the latest command to the front panel micom has been sent out of Sally. Writing a 1 to this location will reset this bit
1	**cmdSentFE**	This bit will be set when the latest command to the front-end micom has been sent out of Sally. Writing a 1 to this location will reset this bit
0	**statReady**	On a read status or a TOC type command this bit will be set when the data has been received from the front end controller. Writing a 1

		to this location will reset this bit

statusA

$0080
Read Only

The status registers are used to obtain the data after a command. For a 5-byte status command, registers **statusA** and **statusB** are used.

This register holds the first four bytes received.

Bits	Name	Description
31-0	**statusA**	First four bytes received

statusB

$0081
Read Only

Next four significant bytes received

Bits	Name	Description
31-0	**statusB**	Next four significant bytes received

statusC

$0082
Read Only

Next two significant bytes received (left aligned)

Bits	Name	Description
31-16	**statusC**	Next two significant bytes received

fpReceive

$0083
Read Only

Front Panel Receive Info:

Bits	Name	Description
31-24	**ReceiveData**	receive data
10-8	**fpMode**	
5-0	**Status**	Same as the status register (see above)

debug1

$0086
Read Only

For debug purposes only. The Debug registers contain no useful information during normal operation

debug2

$0087
Read Only

For debug purposes only.

DEBUG CONTROL MODULE

The NUON system supports breakpointing of the system in various ways in order to allow real-time debugging of operation.

A central debug control module, accessible over the Communication Bus, has a set of control registers for debug and exception functions. Ideally, an external processor runs the debugger software and interfaces with this module, although one of the internal processors may also perform this function. The debug module generates a debug exception interrupt, which is handled by whichever processor is running the debug software.

Debug conditions are:

- MPE debug condition. The MPEs support a variety of debug exceptions, and these are described more fully in the MPE section.
- DMA data and address breakpoint comparator. This unit can compare for either an address, or an address and data condition appearing during a DMA transfer. This may be independently enabled for both read and write. The address may be in linear or XY form. It may also specify a bus master which is either excluded from the condition, or is the sole bus master for the condition. When this occurs all the MPEs can be frozen and a debug interrupt is generated. The current bus owner can be determined. It can also breakpoint on address 0 to catch un-initialized pointers.
- DMA exception. The DMA controller will cause an exception on a range of illegal operations. These can be handled either as a debug interrupt, or as a catastrophic system event, which requires an external processor to debug and restart the system.
- DMA warning. The DMA controller will flag a range of dumb operations, such as zero length transfers. These warning can be ignored, or can be handled in the same manner as DMA exceptions so that the cause of the warning may be determined.
- Other Bus bad address exception. This means that an Other Bus command was issued with an address that was not a valid Other Bus memory space. The transfer will complete harmlessly.
- System Bus bad address exception. A System Bus transfer occurred that, while within the overall valid System Bus address, is not allowable within the current System Bus configuration.

MPE Breakpoints

A breakpoint or halt instruction may be present in the MPE program, which when it is executed causes operation to be suspended, and a debug interrupt to be generated.

The breakpoint instruction may be inserted into source code, or used to over-write another instruction in MPE program memory. Where it is used to over-write another instruction, if that instruction is longer than 16 bits, then multiple breakpoint instructions must be inserted to over-write all of it. This means that the instruction that follows the breakpoint is always valid.

A match between the Data Address Compare Breakpoint register and an MPE data transfer gives a data breakpoint. This may be a scalar, pixel, small vector or vector transfer that includes this address. The data breakpoint can be disabled, and can be restricted to only occurring on write cycles.

Exceptions

MPEs support internal exceptions, which normally result from instruction error conditions. Exceptions can sometimes be handled within the MPE, but are usually considered debug conditions, and so they stop the MPE by putting it into single-step mode, and interrupt the debug processor.

DMA Breakpoints and Exceptions

The DMA unit can cause breakpoints on certain transfer conditions, and can also cause warnings and exceptions on certain conditions. Normally these are handled by interrupts to the debug processor which can take appropriate debug actions.

DMA exceptions, and warnings, may therefore be configured to cause a catastrophic event. When a catastrophic event occurs, all processors and activity in the NUON system shut down, and an external host must be used to find the cause of the error. The external host will use either the System Bus to query the system.

Debug Module DMA

The debug module can be used to perform read or write DMA transfers on the Other Bus or Main Bus of one or two long words. This allows all of memory and the MPE state to be examined and modified directly. It can also perform remote DMA transfers of any length between two other parts of system memory, within the limitations of the corresponding bus.

Debug module DMA transfers have the highest priority on either bus. They are fired off by a write to the base address register, and the transfer pending bit can be polled to determine when the operation has completed.

This channel is intended solely as a back door for debug operations, and will be disabled on production machines.

Watchdog Controller

The debug module contains a watchdog controller whose purpose is to ensure that the system will reset itself should it stop executing its normal software. It is a count down timer of programmable length, which should be set back to its initial value before the count reaches zero. Should the count reach zero, then the NUON is reset to its power on state.

The watchdog control registers are described below.

Debug Module Control Registers

The registers described below are programmed over the Communication Bus. The debug controller has Communication Bus ID 68 (hex 44). The communication protocol is as follows for the command packet:

Long word	Description
0	0-15 register address 31 set for write, clear for read
1	0-31 write data if a write command
2	unused
3	unused

The response packet is returned if the operation was a read. Its format is:

Long word	Description
0	unused
1	0-31 read data
2	unused
3	unused

debugCtrl Debug control

```
$0000
Read / Write
```

This register gives some debug control bits.

Bit	Description
7	Soft reset to the MCU. While this bit is set, the MCU is held in reset.
6	Lock to disable the debug controller. This bit is cleared by a power-on reset or an external host reset, and may be set, but not cleared, by software. Once set, no reads or writes can be performed to the debug controller.
5	Lock to make the debug controller accessible only to the MPEs. This prevents a malicious external host from using the debug controller to compromise security. This flag is set by a power-on reset or an external host reset, but may be modified by the MPEs.
4	Enables setting of the system stop flag by Main Bus DMA exceptions. Defaults to enabled.
3	System stop flag. This bit is normally set by a DMA exception, and optionally by warnings and breakpoints, and causes the Main Bus DMA controller to freeze. This can be set and cleared by software.
2	Enables setting of the system stop flag by Main Bus DMA warnings. Defaults to disabled.
1	Enables setting of the system stop flag by Main Bus DMA breakpoints. Defaults to disabled.
0	System reset flag. All system resources outside the debug module are restored to their power-on state. This bit is cleared by the hardware.

intEna Exception Interrupt Enables

```
$0001
Read / Write
```

This register allows the individual exceptions to cause debug interrupts.

Bit	Description
31	System Bus address exception interrupt enable
30	DMA breakpoint interrupt enable.
29	DMA warning debug interrupt enable.
28	DMA exception debug interrupt enable.
27	Other Bus bad address exception interrupt enable.
3	Enable a debug interrupt on an MPE 3 exception.
2	Enable a debug interrupt on an MPE 2 exception.
1	Enable a debug interrupt on an MPE 1 exception.
0	Enable a debug interrupt on an MPE 0 exception.

mpeReset MPE Reset Control

```
$0002
Read / Write
```

This register allows individual MPE units to be reset. These should be set then cleared again by software

Bit	Description
3	Reset MPE3.
2	Reset MPE2.
1	Reset MPE1.
0	Reset MPE0.

intFlags Debug Interrupt Flags

```
$0003
Read only
```

This register flags the source of a debug interrupt.

Bit	Description
31	System Bus bad address interrupt
30	DMA breakpoint interrupt
29	DMA warning interrupt
28	DMA exception interrupt
27	Other bus Bus bad address exception interrupt.
3	MPE 3 exception interrupt
2	MPE 2 exception interrupt
1	MPE 1 exception interrupt
0	MPE 0 exception interrupt

intClear Clear Debug Interrupts

```
$0003
Write only
```

This register allows interrupt latches to be cleared. Writing a one clears the corresponding latch, writing a zero has no effect, and there is no need to write a zero after a one.

Bit	Description
31	System bus Bus bad address interrupt
30	DMA breakpoint interrupt
29	DMA warning interrupt
28	DMA exception interrupt
27	Other bus Bus bad address exception interrupt.
3	MPE 3 exception interrupt
2	MPE 2 exception interrupt
1	MPE 1 exception interrupt
0	MPE 0 exception interrupt

debugDmaCtrl Control Bits for Debug DMA

```
$0010
Read / Write
```

This register controls the operation of Debug DMA debug transfers.

Bit	Description
4	Debug DMA abort. Setting this bit clears the debug DMA state machines, allowing DMA warning or exception conditions to be cleared.
3	Main Bus transfer pending flag. This will be set for a Main Bus Transfer from when the base address register is written to when the data transfer has completed. You should ensure this bit is clear before reading from the transfer data or writing to any of these Debug DMA registers.

2	Target mainMain bus Bus instead of otherOther bus Bus. This defaults to zero (Other Bus).
1	Other Bus Bus transfer pending flag. This will be set for an Other Bus Transfer from when the base address register is written to when the data transfer has completed. You should ensure this bit is clear before reading from the transfer data or writing to any of these Debug DMA registers.
0	Read transfer flag. This should be set for read transfers and clear for write transfers. Note that bit 13 of the Command Word 1 register also controls this bit.

debugDmaCmd1 Command Word 1 for Debug DMA

```
$0017
Read / Write
```

This register holds the first long-word of the command for the Debug Controller DMA transfer. These are the flags that control the DMA transfer for all types. As all these bits are programmable, other forms of DMA may be performed, such as a transfer from System Bus space to internal memory for faster downloads. After reset, this register defaults to $10010000.

Note that the read bit of the control register above is duplicated in bit 13 here.

Bit	Description
31-0	DMA command long-word 1

debugDmaCmd2 Command Word 2 for Debug DMA

```
$0011
Read / Write
```

This register holds the second long-word of the command for the Debug Controller DMA transfer. This is the base address of the transfer for most DMA operations. After reset this register is not defined.

A write to this register triggers the DMA operation.

Bit	Description
31-0	DMA command long-word 2

debugDmaCmd3 Command Word 3 for Debug DMA

```
$0016
Read / Write
```

This register holds the third long-word of the command for the Debug Controller DMA transfer. This is the internal address for Other Bus transfers, and has various functions for Main Bus transfers. After reset this register defaults to $FFF00000.

Bit	Description
31-0	DMA command long-word 3

debugDmaCmd4 Command Word 4 for Debug DMA

```
$0018
Read / Write
```

This register holds the fourth long-word of the command for the Debug Controller DMA transfer. This is not used for Other Bus transfers, bit is used for pixel mode Main Bus transfers. After reset this register is not defined.

Bit	Description
31-0	DMA command long-word 4

debugDmaData0 Transfer Data Long-word 0 for Debug DMA

```
$0012
Read / Write
```

This register holds both the read and write data for Debug DMA transfer. It should be written to before the command is written for write operations, and can be read once the transfer pending flag has cleared for read operations.

If a single long word is transferred, this register is always used. If two long words are transferred, this is the lower address of the pair.

Bit	Description
31-0	Debug DMA transfer data long word 0

debugDmaData1 Transfer Data Long-word 1 for Debug DMA

```
$0013
Read / Write
```

This register holds both the read and write data for Debug DMA transfer. It should be written to before the command is written for write operations, and can be read once the transfer pending flag has cleared for read operations.

If a single long word is transferred, this register is not used. If two long words are transferred, this is the higher address of the pair.

Bit	Description
31-0	Debug DMA transfer data long word 1

debugRegLock Locks for individual register within the debug controller

```
$0021
Read / Write
```

This register allows individual register within the debug controller to be locked. As the set includes this register, these lock bits can themselves be locked. The default state for bits in this register is set, i.e. writes to all registers are enabled.

This function is available from Aries 2 onwards.

Bit	Name	Description
31	**debugCtrlWrEna**	When set, the **debugDebugCtrl** register may be written to.
30	**intEnaWrEna**	When set, the **debugIntEna** register may be written to.
29	**mpeResetWrEna**	When set, the **debugMpeReset** register may be written to.
28	**intClearWrEna**	When set, the **debugIntClear** register may be written to.
27	**dmaCtrlWrEna**	When set, the **debugDmaCtrl** register may be written to.
26	**dmaCmd2WrEna**	When set, the **debugDmaCmd2** register may be written to.
25	**dmaData0WrEna**	When set, the **debugDmaData0** register may be written to.
24	**dmaData1WrEna**	When set, the **debugDmaData1** register may be written to.
23	**dmaCmd3WrEna**	When set, the **debugDmaCmd3** register may be written to.
22	**dmaCmd1WrEna**	When set, the **debugDmaCmd1** register may be written to.
21	**dmaCmd4WrEna**	When set, the **debugDmaCmd4** register may be written to.
20	**regLockWrEna**	When set, the **debugRegLock** register may be written to.

Watchdog Timer

NUON contains a watchdog timer that may be used if it is necessary to ensure that the system can recover in some manner from a crash. When this function is enabled, software must regularly re-trigger the timer, or the system will be reset. This function is part of the debug controller.

wDogCtrl Control Bits for the Watchdog

```
$0020
Write only
```

This register controls the operation of the watchdog circuit. This is a down counter, which will reset NUON unless it is re-triggered before it reaches zero. This is generally used as an emergency recovery mechanism for systems that have crashed.

Bit	**Description**
31	Re-trigger the watchdog. The counter is reloaded with its start value and commences counting down again.
30	Watchdog lock. If a one is written to this bit then the watchdog is locked. The only action that you can perform after you have set this bit is set is a re-trigger. You cannot unlock it, disable it, or change the watchdog count value.
29	Watchdog enable. Until this bit is set the watchdog is idle.
28-0	Watchdog count value. This is the interval, in main (54 MHz) clock cycles, that is required between re-trigger actions to prevent a system reset. *Aries 2 and earlier – this is a 24-bit count value in bits 23-0.*

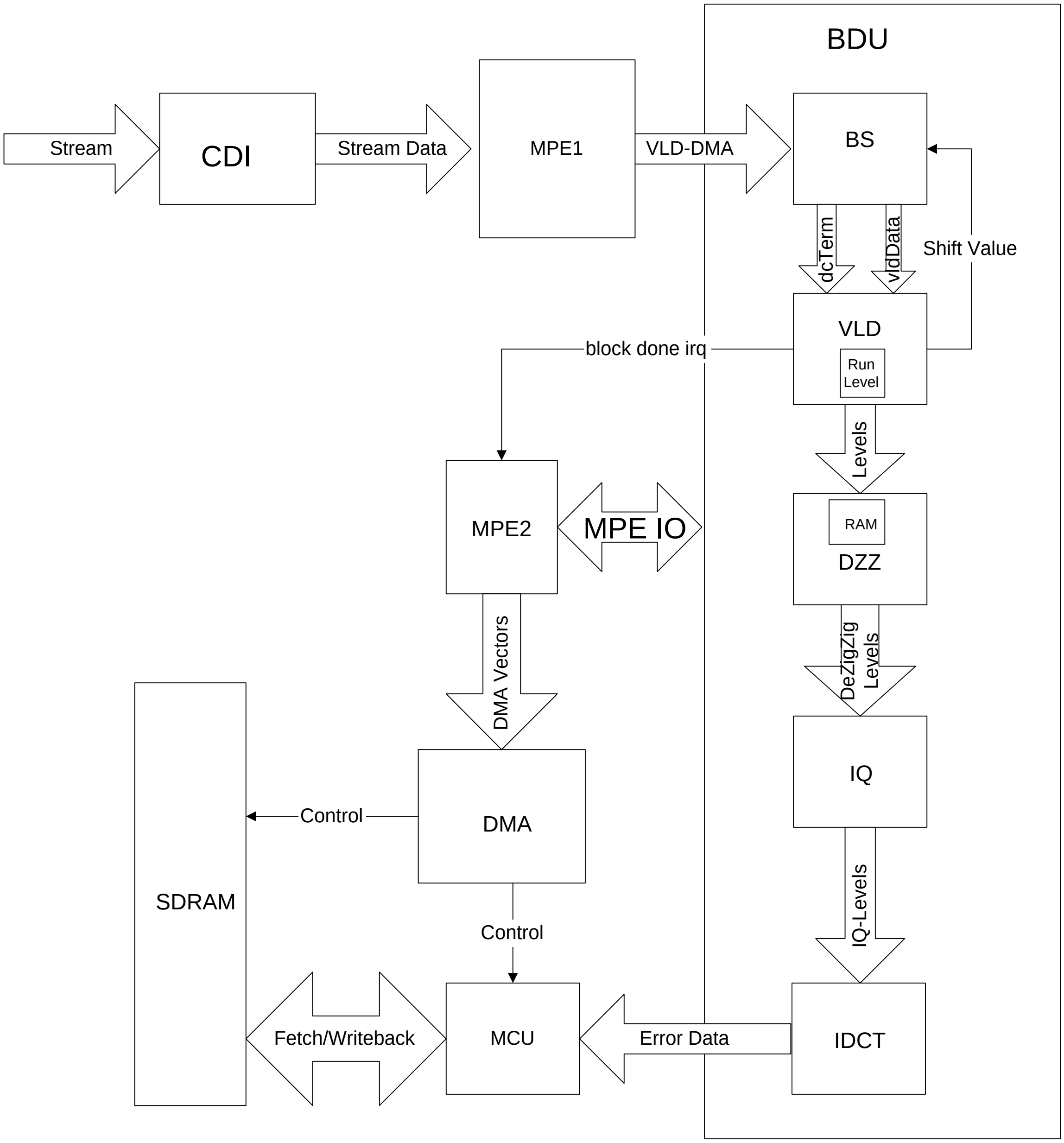
MACROBLOCK DECODING UNIT (BDU)

This section describes the macroblock decoding hardware unit. This unit performs the decoding of one complete macroblock at a time upon receiving a command from MPE2. The main components of the MPEG hardware decode path are:

- **CDI:** The coded data interface talks to the external world, receiving a PES stream or a program stream and transferring the data to MPE1 via the Communication Bus.
- **MPE1:** Media Processor Element number 1: The main function of this processor is to decode and transfer data from the CDI to the BDU shifter through the VBV Buffer and vldDma interface.
- **MPE2:** Media Processor Element number 2: The main function of this processor is to decode the higher stream layers and the motion vectors and control the BDU hardware
- **BS:** The bit shifter requests bit stream binary data from the vlddma (from MPE1) and provides the data to a consumer. The consumer could be MPE2 to perform high level parsing or the VLC (variable length code) decoder unit in the BDU. Another consumer is the dc code decoding logic for intra blocks.
- **VLD :** This is the Huffman code (VLC) decoding unit. It reads valid data from the bit shifter, decodes it and informs the bit shifter how many bits have been consumed. The VLD unit provides 64 levels per block to the DZZ, de-zig-zag unit.
- **DZZ:** The de-zig-zag unit contains a 64x12 single port RAM that is written with the data coming from the VLD unit in zig-zag order and read by the IQ unit in normal order (determined by the IDCT logic).
- **IQ:** The inverse quantization unit reads the data from the zigzag memory in the order required by the IDCT unit and performs the inverse quantization of this data. It has a 32x32 RAM that will hold the two current inverse quantization matrices.
- **IDCT:** The IDCT unit performs the inverse discrete cosine transform algorithm over all the inverse quantized blocks in coming from the IQ unit. Each block is fed into the motion compensation unit (MCU) for further processing (this unit is not part of the BDU, please see a separate specification of it).
- **DMA:** Direct Memory Access engine (subject of a separate document)
- **MCU:** Motion Compensation Unit (subject of a separate document)

The BDU ‘s components are the BS, VLD, DZZ, IQ, and IDCT units. The other units are the subject of separate chapters.

The following block diagram shows the major data and control path in the MPEG hardware decoding unit,



BDU IO Interface

Bitstream Data Interface

The bitstream data interface allows stream binary data to be fed into the bit shifter for further processing. This is achieved by requesting data from the vld dma (see MPE specification) module attached to MPE1. From the BDU point of view, the bit shifter control logic issues a request to MPE1. MPE1 will reply when a new set of 16 bits are available. The next new serial bit will be in the most significant location.

MPE2 Command Interface

The BDU internal control and command registers can be read and written by MPE2 as any other coprocessor space registers. The BDU has a 5-bit (long-word aligned) address space between MPE2 IO address $20501200 and address $205013F0.

Two bits in the interface are used as two additional condition codes into the execution unit of MPE2. This allows and easy way of waiting and acting on status information coming from the BDU. These bits are coprCC[1], which if asserted indicates a start code has been found after a getStartCode command and coprCC[0] which is used to indicate to the driver code if the data read by MPE2 from the bit shifter was valid or not.

The following are the write only addresses:

vldcmd VLD Command Register

```
vldcmd = $2050 1200
Write only
MPE 2 only
```

This is the BDU command register (called VLD because of historical reasons). It holds a three bit command for the BDU:

Code	Command	Description
000	**NOP**	No Operation
001	**getStartCode**	Bit shifter will consume bit stream data until a start code is found. The start code id bits will be the most significant bits sitting in the vldData register. CoprCC[1] condition code will be set upon successful completion of this command
010	**resetDcPredictor**	MPE2 indicates to the BDU to reset its dc predictors
011	**resetBitShifter**	MPE2 indicates to the BDU to reset the bit shifter
100	**resetBdu**	Soft reset for the whole BDU unit
101	**decodeMBlock**	MPE2 indicates to the BDU to decode a macroblock starting with the current contents of the vldData register
110	**clrErrorBits**	MPE2 indicates to the BDU to clear error bits in error register

The getStartCode command will clear the coprCC[1] bit indicating start code detected and will set it again when a start code is found. The following piece of code makes efficient use of this logic:

```
    mv_s #get_start_code_cmd, temp  ; get start code command
    st_s temp, (vldcmd)             ; issue get_start_code command
    nop                             ; make sure that cf1lo bit is
                                    ; cleared in hw
wait_for_start_code :
    jmp cf1lo, wait_for_start_code, nop
                                    ; wait in this loop until
                                    ; start code is detected
```

vldmode VLD Mode Register

```
vldmode = $2050 1210
Write only
MPE 2 only
```

This register, sets a few important mode bits critical for the BDU operation. These bits are chosen so that they will not change too often (not on a per macroblock basis). The mode bits are:

Bit	Name	Description
0	**intra_vlc_format**	Follows the status of this bit in the bitstream. Used to decode which AC VLC table to use
1	**reserved**	Unused
2	**scanOrder**	Indicates if normal (0) or alternate scan order (1)
6-3	**intra_dc_mult**	0010: 2 0100: 4 1000: 8
7	**streamType**	0: mpeg2 stream 1: mpeg1 stream

vldmblock VLD Mblock Register

```
vldmblock = $2050_1220
Write only
MPE 2 only
```

This register holds a few configuration bits important for macroblock decoding that are likely to change on macroblock per macroblock basis.

Bit	Name	Description
6:0	**quantScaleValue**	quantizer scale value (from bits stream)
7	**intraMBlock**	indicates if intra(1) or non-intra (0) mblock
8	**dct_type**	indicates if field (1) or frame (0) dct_type

vlcbp CBP register

```
$20501230
Write only
MPE 2 only
```

This register holds 6 bits with the coded block pattern for the macroblock being decoded.

Bit	Description
0	if set : block 5 is coded
1	if set : block 4 is coded
2	if set : block 3 is coded
3	if set : block 2 is coded
4	if set : block 1 is coded
5	if set : block 0 is coded

vldbits VLD Bits

```
$2050 1240
Write only
MPE 2 only
```

This register is used when the MPE has control over the bit shifter. When this 5-bit register is written, the bit shifter will shift the current vldData contents by the amount indicated in vldBits (0 to 28). If this register is written while BDU has control over the bit shifter (after a decode macroblock command and before a end of macroblock interrupt), strange things can happen. Also the driver must ensure that there is enough valid data to be shifted out. It will take at least 2 ticks until the new shifted data is ready to be read. When the coprCC[0] bit is set, data in vldData is valid. The following code shows an efficient way of using this command:

```
      st_s   #17, (vldbits)          ; store number of bits to consume
                                     ; (17 in the example)
      ld_s   (vlddata), temp_input0  ; load new data (2-tick operation)
      jsr    cf0lo, vlddata_retry, nop
                                     ; jump on coprCC[0] if vlddata
                                     ; not valid to retry

vlddata_retry :
      push   r0, cc,rzi1,rz          ; push stuff (if needed)
vlddata_retry_loop :
      ld_s   (vlddata), temp_input0  ; retry loading from vlddata
      jmp    cf0lo, vlddata_retry_loop, nop
                                     ; if vlddata not valid keep retrying
      rts                            ; return if done
      pop    r0, cc,rzi1,rz          ; pop stuff if needed
      nop
```

vldctrl BDU Ctrl

```
vldctrl = $2050 1250
Write only
MPE 2 only
```

These two bits perform BDU control tasks as follows:

Bit	Name	Description
0	**startCodeEnable**	Enable start code detection logic
1	**bitShifterEnable**	Enable bit shifter logic

vldresetiqvalid Reset IQ Valid

```
vldresetiqvalid  = $2050 1260
Write only
MPE 2 only
```

Writing to this address will reset the IQValid status bit. This command should always be executed before issuing a Communication BusCommunication Bus transfer command, to transfer IQ data from the DZZ memory to the IQ tables memory

The following are the read only addresses. The values are right justified into the 32 bit read data bus.

vldstatus VLD Status Register

```
vldstatus = $2050 1300
Read only
MPE 2 only
```

This register contains three status bits:

Bit	Name	Description
7	**iqValid**	Indicates that the data in IQ memory is valid after a Communication Bus transfer command.
1	**bsDcVlcErr**	bad VLC found in DC term decoding process
0	**realVlcErr**	bad VLC found in decoding of AC elements

The two error bits can be cleared with the clrErrorBits command. The iqValid bit can be reset with the Reset IQ Valid command.

vlddata VLD Data Register

```
vlddata = $2050 1310
Read only
MPE 2 only
```

This 28-bit register contains the current data coming out of the bit shifter. If this data is valid or not is indicated by the coprCC[0] as indicated above. This register is used to enable the MPE to make use of the bit shifter to decode the upper layers of a video bit stream. Up to 28 bits can be shifted out in a single operation.

vldzeros VLD Zeros Register

```
vldzeros = $2050 1320
Read only
MPE 2 only
```

This 4-bit register indicates the current number of leading zeros in the VLD Data register (from 0 to 15 leading zeros). It is up to the driver to ensure that the bits are valid (coprCC[0]).

vlddebug1 Debug 1

```
vlddebug1 = $2050 1330
Read only
MPE 2 only
```

This and the next two registers are mostly used for debugging purposes to enable reading of internal status of important registers and state machines. It can also be used for driver development

Bits	Name	Description
20:13	**vldMode**	Contents of vldMode register
12:7	**cbp**	Contents of CBP register
6:2	**vldBits**	Contents of vldBits register
1:0	**bduCtrl**	Contents of bduCtrl register

vlddebug2 Debug 2

```
vlddebug2 = $2050 1340
Read only
MPE 2 only
```

See comment on Debug1 register

Bits	Name	Description
29:28	**cntOut**	bs input fifo counter
27:12	**dbgOutReg**	outputReg in bs. Data that is not visible in vldData
11:9	**vldCmd**	Contents of vldCmd register
8:0	**vldMblock**	Contents of vldMblock register

vlddebug3 Debug 3

```
vlddebug3= $2050 1350
Read only
MPE 2 only
```

See comment on Debug1 register

Bits	Name	Description
31	**swSection**	state bit from bs state machine
30:25	**vldCount**	number of valid bits in vldData
24:20	**dzzIqState**	state of dzz Iq control state machine
19:17	**vldState**	state of vld control state machine
16:14	**mainBsState**	state of main bit shifter control state machine in bs module
13:11	**scState**	state of start code control state machine in bs module
10:8	**bsState**	state of bit shifter control state machine in bs module
7:5	**dcState**	state of dc term control state machine in bs module
4:0	**idctState**	state of idct control state machine in idct module

BDU Interrupts

There are two pulse interrupts from the BDU unit to MPE2. These are:

- endOfMbIrq : End Of Macroblock Interrupt. This interrupt is asserted when the last block of the current macroblock has finished using the bit shifter. The latter is available now to be used by MPE2 to decode further downstream until the next macroblock
- bduErrorInt : Bdu Error Interrupt: This interrupt is currently asserted in two scenarios: If the vld decoding logic detects a dc component vlc error or a ac component vlc error.

Communication Bus Interface (Communication Bus)

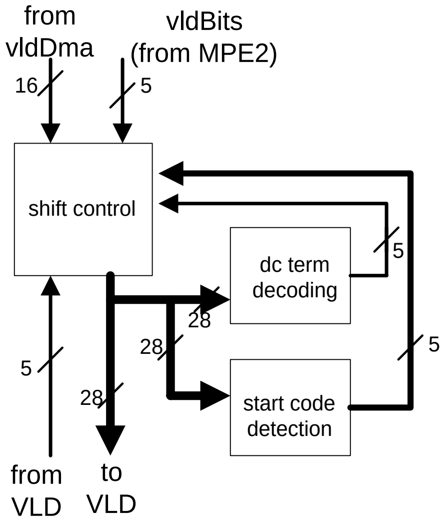
The BDU unit is connected also to the Communication Bus to allow access to internal structures such as the de-zigzagger memory and the IQ table memory. The idea is to provide some flexibility to be able to use the BDU with other applications and also to increase the visibility for debugging purposes. The Communication Bus ID for the BDU is 73 (decimal). Please see the DZZ and IQ section for details on Communication Bus transaction with the BDU.

BDU Units

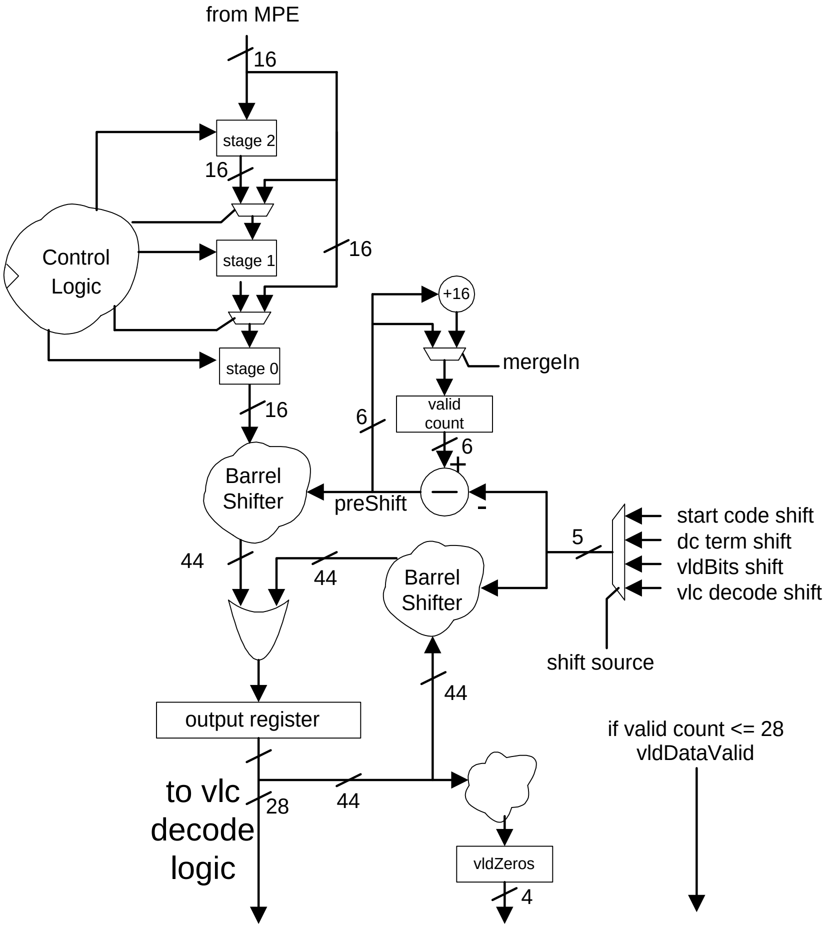
Bit Shifter

The Bit Shifter units talks to MPE2 through the vld dma interface. For details on this interface, see the “Bitstream Data Interface” paragraph above. It’s main mission is to maintain at least 28 valid bitstream bits available at all times to be used by the vlc decoding unit or by MPE2. The valid data is IO mapped on MPE2. Once data is consumed by one of the clients (MPE2, vlc decoder, start code detection logic or dc term decoding logic), they can request a left shift from 0 to 28 positions. From MPE2 this is done through the vldBits register and from all the other units this is accomplished through a control bus.

The following block diagram show the main units, data paths, and shift control sources for the Bit Shifter unit



The following figure shows a block diagram of the shift control logic of the bit shifter unit.



On top of the shifting logic is a three stage 16 bit wide FIFO that reduces the probability of stalling when data coming from the vld dma logic in MPE1 has to be delayed for any reason. The unused data in the output register is fed back, reshifted, and merged in with new data coming from the vld dma logic. The amount of consumed data is determined by the mux on the right that selects between the four sources for shift data MPE2 through vldBits register, start code detection logic, vlc code decoding logic, and dc term decoding logic).

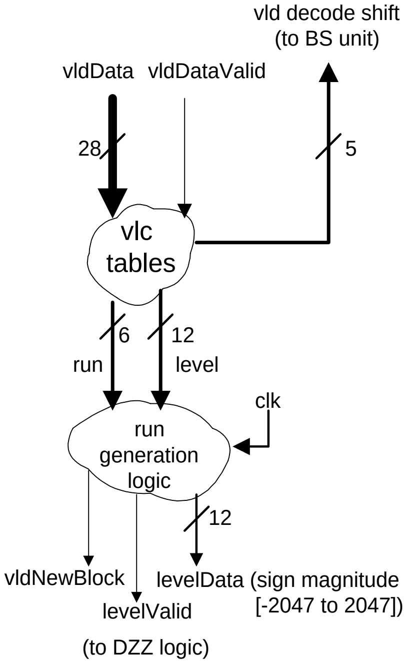
The most significant 28 bits out of the output registers are passed to the data clients (it can be read by MPE2 through the vldData register). The data is valid when those 28 bits are valid as indicated by the vldDataValid signal. This signal is also used by the BDU logic to generate the coprCC[0] condition code bit.

MPE2 is able to use the bit shifter through the vldBits coprCC[0] mechanism as shown earlier, to parse and decode all the higher stages of a MPEG2 stream. Once it has decoded a macroblock header and loaded the control registers with the appropriate (cbp, vldMode, vldMBlock) values, MPE2 will issue a decode macroblock command to the BDU and will relinquish control of the bit shifter until the BDU unit is done parsing the data for the current macroblock (indicated through the endOfMbIrq interrupt generated in the vld unit).

When the bit shifter unit gets a ‘decode macroblock’ command and using the data in the control registers, the BS logic will first decide if a dc term decoding is necessary (for intra macroblocks). If so required, the dc term decoding state machine will take control over the shift control logic and decode the dc term for the next block and store it in the vldDc register, as well as update the predictor registers. Upon completion of the dc term, the VLD module will take control of the bit shifter and start decoding all 64 (or 63) ac terms of each block. When a new shift value is provided to the shift control logic, the shifted data will appear in the vldData register and bus after the next clock rising edge.

VLD

The VLC Decoding unit takes valid bitstream data from the bit shifter, does a table lookup to extract run/level pairs, provides the runs of zeroes and the levels to DZZ unit and indicates to the Bit Shifter unit how many bits have been consumed. The following is a block diagram of the VLD unit.

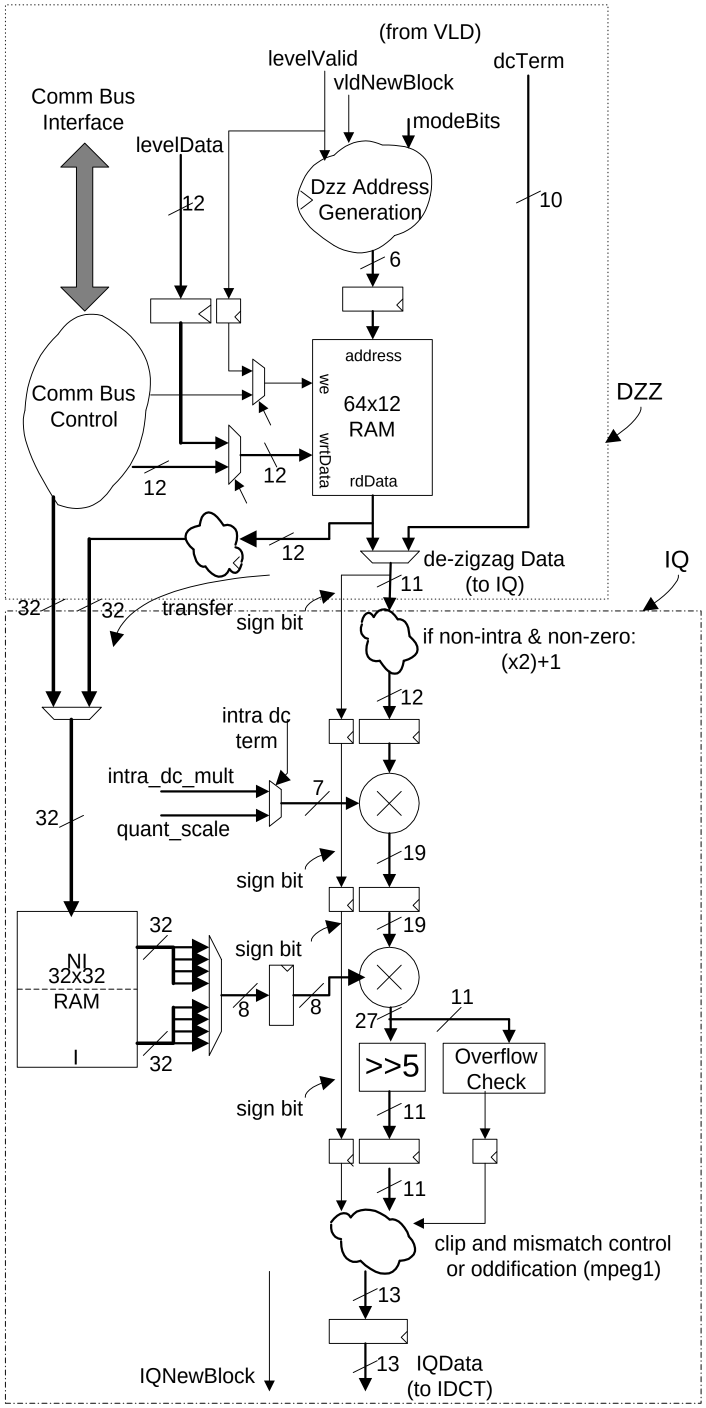


Providing no stalls from the upper pipe stages, it is possible for the vlc decoding logic to decode 1 symbol on every clock tick for any legal sequence, with exception of escape code variable length codes which could take up to 2 symbols in MPEG1. The largest single symbol in MPEG1 or MPEG2 is 28 bits (MPEG1 escape level) and the bit shifter can handle that in 1 clock tick.

The first element of a new block (could be element 0 or 1 depending on the macroblock type) is accompanied by a set vldNewBlock signal, to indicate this fact to the DZZ logic. The levelData (validated by levelValid) is represented in sign-magnitude format (from -2047 to +2047].

DZZ

The de-zigzagger unit accepts the incoming data from the vlc decoding unit, writes it into a 64x12 static ram in one of the two zigzag orders allowed in the MPEG2 specification. It then allows the lower pipe stages from the BDU to read the data in raster scan order. The read order is actually driven by the order required by the IDCT (see chapter below), but the data in the DZZ RAM is in Raster scan order for the current block. Since the DZZ and the IQ units are so closely related, the following block diagram show both units.



The Communication Bus Control logic for the DZZ/IQ unit accepts a few commands as shown in the following table.

Command	**Id**	**Comment**
Fill IQ Intra	1	Write to IQ ram intra section
Fill IQ Non-Intra	2	Write to IQ ram non-intra section
DZZ read	3	Read the DZZ Ram
DZZ fill	4	Write to DZZ Ram
IntraTransfer	5	Transfer DZZ Ram to IQ Ram Intra section
non-intraTransfer	6	Transfer DZZ Ram to IQ Ram Non-Intra section

Read IQ	7	Read IQ memory

The Communication Bus Control logic for the DZZ/IQ unit on the three least significant bits of the first long word of the message as follows:

127	98 96	63	32	0
0	C M D	0	0	0

The Fill IQ Intra command is used to write to the Intra section of the IQ RAM (32x32 Ram, addresses 16 to 31). First the command is sent as above and then four Communication Bus packets with 16 bytes each are sent with the 64 data bytes. The first byte of the first long will go into the first address (16) and so on.

The Fill IQ Non-Intra command is used to write to the non-intra section of the IQ RAM (addresses 0 to 15). The procedure is similar to the above

The DZZ read command, allows any Communication Bus master to read the DZZ RAM contents. The reply will be sixteen Communication Bus packets directed at the source of the request with four data elements each with the following format:

127	107 96		75 64		43 32		11 0
0	Data0	0	Data1	0	Data2	0	Data3

Data0 of the first message is the DC term and Data1 is the first AC term and so on.

The DZZ fill command, allows any Communication Bus master to load the DZZ RAM with random data. The command is sent as seen above. Then, DZZ expects 16 more messages with 4 dzz ram entries each with the same formatting as the read DZZ command.

Data 0 is the left most entry of the four in raster scan order (lower frequency element).

The intraTransfer and non-intraTransfer command are used to transfer data from the DZZ memory to the IQ memory holding the IQ weights. The commands are different for intra and non-intra macroblocks since there are two current matrices stored at different base addresses in the IQ logic. This is useful to use the DZZ ram to de-zigzag the IQ weights coming in the bitstream. Before issuing a transfer command, the driver should issue a resetIqValid command, so that the corresponding status bit gets reset. This bit will get set when the weights matrix has been completely written into the corresponding IQ ram. Both these commands use the format show above.

The Read IQ command is sent as shown above and it is used to read the contents of the IQ memory. The data is packed onto 8 consecutive Communication Bus packets with 16 bytes each. The first byte of the first long of the first message has address 0 and so on. This command was added only after Aries 2.1 (i.e. should be only after Aries 3.0)

IQ

The IQ unit will read the values out of the DZZ memory in the order requested by the IDCT unit, and perform the inverse quantization of each element. The block diagram is shown above in the DZZ section, since both units share a considerable amount of logic.

The 12-bit sign-magnitude data is read from the DZZ ram. The sign bit is pretty much propagated untouched down the pipe to the IDCT. The magnitude is multiplied by two and incremented if it is a non-intra block and a non-zero element. The resultant 12-bit value is multiplied by the intra_dc_mult value, if this is an intra macoblock, or by the quantizer scale value, if this is an ac coefficient. The full precision result is 19 bits wide, and it is multiplied in the next pipe stage by the IQ weight read from the weight's RAM.

The resulting 27 bits magnitude is divided by 32 and the upper 11 bits are OR'd together to see if overflow has happened. In this case the next stage will clip the result to the IDCT input range ([-2048, 2047]). The final logic will perform mismatch control or oddification (for mpeg1 streams) and the 13-bit final sign-magnitude data is passed to the IDCT block.

For the DZZ to MCU (through IDCT and IQ) pipe, there is no stalling mechanism. When the last block element is written into the DZZ RAM, this pipe is triggered and will only stop when the whole block has been written to the MCU ram. Therefore, all the IDCT needs to know is when a new block (IQNewBlock) is available to start processing the whole block without interruption. It is up to the scheduling of events done by the driver running on MPE2 to make sure that there is room in the MCU buffers for the current macroblock in the BDU pipe.

IDCT

The IDCT unit performs the inverse discrete cosine transform operation on a block-by-block basis as defined in the MPEG2 specifications

IDCT PRECISION ANALYSIS

This section describes the considerations used for the idct precision analysis model and hardware implementation. This implementation is with two 1-D transforms using identical hardware. The original 8x8 block in the frequency domain is transformed by the hardware row by row (or column by column) and the intermediate result is stored in RAM. The same hardware performs a second pass reading the intermediate block from the RAM processing it and producing the samples in space domain. The IDCT operation has to meet the precision numbers specified in the IEEE Std 1180-1990 document.

The basic IDCT transform equation is

$$f(y,x) = \sum_{0}^{7} \frac{C(v)}{2} \sum_{0}^{7} \frac{C(u)}{2} * F(v,u) * \cos[(2x+1) * \frac{u\pi}{16}] * \cos[(2y+1) * \frac{v\pi}{16}]$$

where

$$C(u) = \tfrac{1}{\sqrt{2}} \quad u = 0 \qquad\qquad C(v) = \tfrac{1}{\sqrt{2}} \quad v = 0$$

$$C(u) = 1 \quad u > 0 \qquad\qquad C(v) = 1 \quad v > 0$$

Due to separability, this can be performed with two 1-D operations according to the following equation:

$$f(x) = \sum_{0}^{7} \frac{C(u)}{2} * F(u) * \cos[(2x+1) * \frac{u\pi}{16}]$$

which in turn translates into the following matrix operation for each 8x8 row or column of the original frequency domain block

$$\begin{bmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \end{bmatrix} = \begin{bmatrix} C0 & C1 & C2 & C3 & C4 & C5 & C6 & C7 \\ C0 & C3 & C6 & -C7 & -C4 & -C1 & -C2 & -C5 \\ C0 & C5 & -C6 & -C1 & -C4 & C7 & C2 & C3 \\ C0 & C7 & -C2 & -C5 & C4 & C3 & -C6 & -C1 \\ C0 & -C7 & -C2 & C5 & C4 & -C3 & -C6 & C1 \\ C0 & -C5 & -C6 & C1 & -C4 & -C7 & C2 & -C3 \\ C0 & -C3 & C6 & C7 & -C4 & C1 & -C2 & C5 \\ C0 & -C1 & C2 & -C3 & C4 & -C5 & C6 & -C7 \end{bmatrix} \begin{bmatrix} F(0) \\ F(1) \\ F(2) \\ F(3) \\ F(4) \\ F(5) \\ F(6) \\ F(7) \end{bmatrix}$$

where the constants are defined as $Ck = \frac{C(k)}{2} * \cos(\frac{k\pi}{16})$. There is clear mirror type symmetry along the horizontal centerline which is used in the hardware implementation. Also, we observe that $C0 = C4 = \frac{1}{2\sqrt{2}}$. Two symmetrical terms can be calculated at the same time. For example:

$$\begin{gathered} PE = C0*F(0) + C2*F(2) + C4*F(4) + C6*F(6) \\ PO = C1*F(1) + C3*F(3) + C5*F(5) + C7*F(7) \\ f(0) = PE + PO \\ f(7) = PE - PO \end{gathered}$$

The two precision parameters to determine are how many bits to be used for the constants and how many bits are required for the intermediate value. The input range is [-2048,2047] (12 bits) and the output needs to be rounded and clipped to [-256,255] (9 bits).

Let C_i be the #C bits wide i-th coefficient width and #I be the intermediate RAM width.

Since the same data path has to be used for the 1st and 2nd passes the original 12 bit inputs needs to be left shifted to match the width of the intermediate value (#I). The coefficient is shifted left by #C to avoid the floating point. To obtain one of the f(x) (1 pass) we calculate:

$$\sum_{0}^{7} F_i * 2^{\#I-12} * C_i * 2^{\#C}$$

Each product term is #I+#C bits wide. If #M is the right shift after each product term multiplication (reduction in fracbits), the width of each term is #I+#C-#M. Considering the maximum ranges for F and C, it is easy to verify that the previous equations output cannot exceed 15 integer bits for the first pass. In other words, we will have #I-15 fractional bits in the intermediate RAM module and the summation add 3 bits to the 12 integer input bits. The final sum, will also have #I-12 zeroes to the right which can be left out of the intermediate value. Therefore the intermediate value can be rewritten as the total summation width minus all the right-most zeros:

$$\#I+\#C-\#M+3-(\#I-12)=\#I$$

from where we obtain that the multiply shift value need be:

$$\#M=\#C-\#I+15$$

The second pass is the following summation

$$\sum_{0}^{7} J_i * C_i * 2^{\#C}$$

where J_i is the intermediate value from the RAM. Each product term is of width #I+#C and after the multiply shift #I+#C-#M = 2x#I-15. Since we expect integer outputs, we have to adjust for the #I-15 fractional bits of the J_i (the summation doesn't add fracbits) and the #C left shifting of the coefficient. Since we already shifted right by #M, we still have to shift right by

$$\#C+(\#I-15)-\#M=2x\#I-30$$

Finally we have to adjust for the $\frac{1}{2}$ factor of the coefficient that we didn't consider in each pass. The final right shift will be 2x#I-28.

A C model of the hardware was written and verified to comply with IEEE 1180-1990 for values of #I > 17 and #C > 12 which are the values used in the hardware implementation.

CODED DATA INTERFACE

The Coded Data Interface is a programmable interface, responsible for conveying compressed data streams to a designated MPE in the MMP system. Compressed data could be in the form of either audio and/or video elementary streams or transport / program streams. The CDI presents a glue-less interface to a number of commercial transport stream de-multiplexers, channel decoders and CD-DSPs. Application layer video elementary stream data is carried over a byte-wide interface (most significant byte first) that can be programmed as synchronous or asynchronous. Audio elementary stream data is either multiplexed with video over the byte-wide interface, or carried over an independent bit-serial interface (most significant byte/bit first). The bit-serial interface is also programmable as synchronous or asynchronous. System level compressed data (transport streams and program streams) is carried over the byte-wide interface.

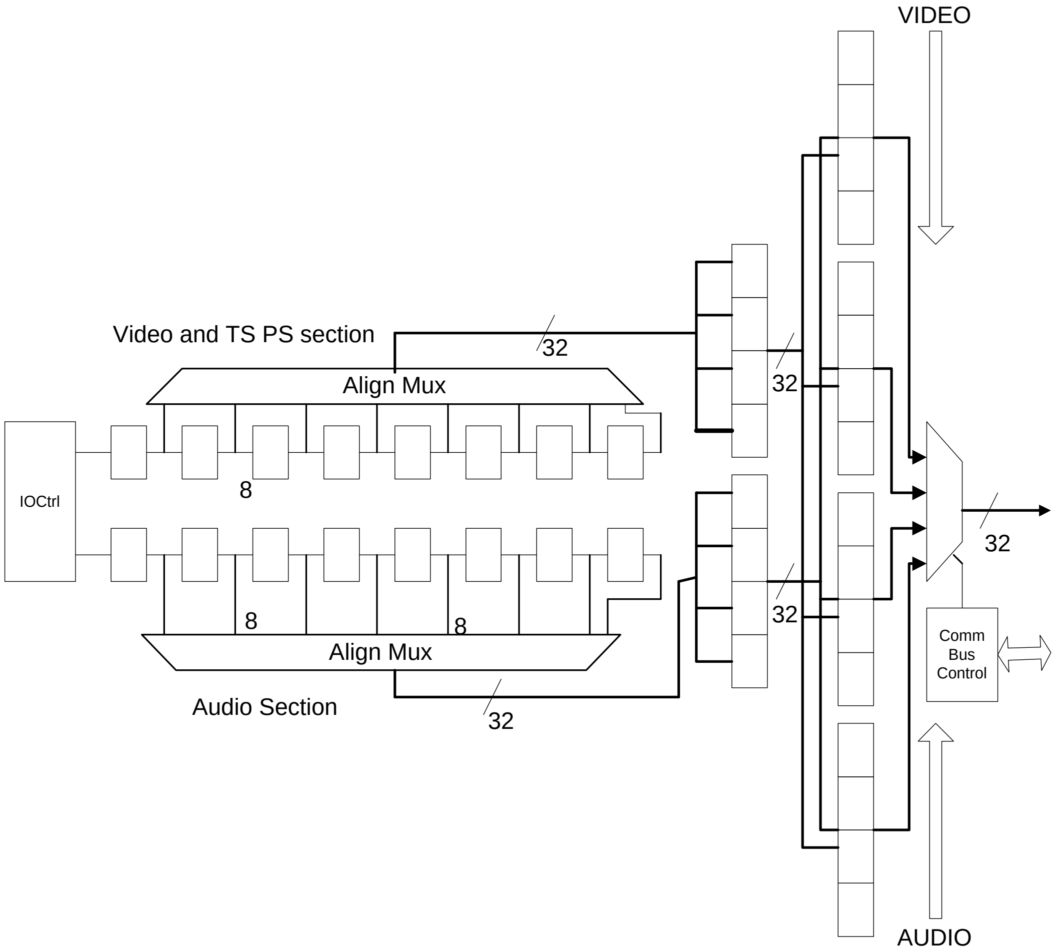
The CDI IO is multiplexed to make better use of the Chip IO pins. The following table shows the CDI interface pins to the external world and the different modes where they are used.

Pin Names	**I/O**	**Program Elem. Stream Video**	**Program Elem. Stream Audio**	**Transport Stream**	**Program Stream**
CVDATA0/CAPDATA0	I	√	√(Par. Aud)	√	√
CVDATA1/CAPDATA1	I	√	√(Par. Aud)	√	√
CVDATA2/CAPDATA2	I	√	√(Par. Aud)	√	√
CVDATA3/CAPDATA3	I	√	√(Par. Aud)	√	√
CVDATA4/CAPDATA4	I	√	√(Par. Aud)	√	√
CVDATA5/CAPDATA5	I	√	√(Par. Aud)	√	√
CVDATA6/CAPDATA6	I	√	√(Par. Aud)	√	√
CVDATA7/CAPDATA7	I	√	√(Par. Aud)	√	√
CVREQ*	O	√			√
CVENAB/CVSTROBE*	I	√		√	√
CVCLK*	I	√		√	√
CAENAB/CASTROBE*	I		√		
CASDATA/CVERRFLG*	I		√(Ser. Aud)	√	√
CAREQ/CVTOP*	I/O		√(O)	√(I)	√(I)
CACLK*	I		√		

*: Signal polarities (active edges in case of clocks) are programmable

Table 1: Coded Data Interface Pin Configuration

The following block diagram shows the main sections of the coded data interface



The IOCtrl section of the CDI takes care of all the asynchronous clock domains and serial/parallel input formats and provides an 8-bit synchronous external interface to the rest of the logic. The top section in the above figure takes care of parallel video, and transport and program streams. The bottom section is used for serial/parallel audio and top and error bits for PS and TS. The series of 8 flip-flops is used to take care of the alignment commands coming from the MPE. The data is then arranged in 32-bit words and load into the Communication Bus data holding registers for transmission to MPE1. The Communication Bus message can contain a mixture of audio and video data on 32 bit boundaries. This is achieved by loading the video data from the top down and the audio data from the bottom up. 8 special bits in the Communication Bus message will contain the necessary information for the receiver to parse the incoming data correctly (see below).

8 128

Coded Data Interface Operation

The command and status information is exchanged between MPEs and CD Interface by the following means:

- Communication Bus Transmit status field (8 bits): From an MPE to convey a command and from the CD Interface to convey status. The transmit status bits from an MPE are received in the Communication Bus Receive Status field of the CD Interface. Likewise, status information encoded by the CD Interface in its Communication Bus Transmit Status field is received by the target MPE in its Communication Bus Receive Status field.

- Communication Bus Transmit register: Certain Communication Bus Transmit field commands and status bits are supplemented by information encoded in the most significant long word (MSLW) portion of the Communication Bus Transmit register. In this implementation, this field is used to transmit PS mode error information only**.**

Three types of commands are sent by the MPE to the CD Interface. First is the configuration command which is invoked during initialization to configure the CD Interface (by writing the CDI_config register). Associated with this command is a status query command that allows the MPE to read the current configuration as programmed in the CDI_config register. The second type of command is used by the MPE to force alignment of incoming data. In all modes of operation, alignment is carried out by offsetting the initial position of the shift register write counter. The offset can take values 1, 2 or 3 to force an effective right shift on incoming data, of 1, 2 or 3 bytes (Note: A value of ‘0’ in the align field implies no alignment for the target stream). This allows incoming byte aligned data to be aligned on long word boundaries. In this implementation, an additional restriction is placed on alignment in the TS/PS modes of operation. This restriction allows alignment to be carried out only at the second position of Communication Bus Transmit registers. The CD Interface responds to the alignment command by carrying out the alignment, and returning a status code marking the first Communication Bus data packet which contains aligned data. All future data transmitted by the CD Interface is considered “normal” data, until the next MPE alignment command is processed. The third command is used by the MPE to force a dump of the residual incoming bytes in a burst mode TS/PS stream. This command is used in cases where PS packs arrive in bursts, followed by a few clocks of silence. The receiving MPE has the option of sending a “flush” command during the silent period. This command forces a Communication Bus cycle that carries the residual bytes left in the CD Interface shift and holding registers.

The information exchanged between MPEs and CD Interface in different operating modes is described below:

CD Interface General Configuration

Following commands and status information are exchanged between MPEs and CD Interface during CD Interface configuration:

Source	**Description**	**TX Status**	**MSLW**	**Dest.**
MPE	Command to query CDI Configuration status	1000 0000	<NONE>	CDI
CDI	Response to requesting MPE <1000 0000> command	1000 0000	CDI_config reg.	MPE
MPE	Command to program CDI_config register	1001 0000	CDI_config info.	CDI
MPE	Command to flush CDI data (PS/TS only)	1110 0000	<NONE>	CDI

cdiConfig CDI Configuration

This register may be read from and written to as shown above.

Bit	**Name**	**Description**
30	**toshCDMode**	select logic to fix weird Toshiba CD handling of error stuff
29	**i2sModeEn**	enable i2s mode
28	**cdiNoAReq**	squash the audio request line
27	**cdiNoVReq**	squash the video request line
26	**stopCmbBus**	prevents comm bus from sending unsolicited data

25	**polAReqClk**	polarity control signal for output synchronous audio request line
24	**polVReqClk**	polarity control signal for output synchronous video request line
23	**parallVideo**	indicates parallel/serial video
22	**parallAudio**	indicates parallel/serial audio
21	**syncAudio**	indicates sync/async audio
20	**syncVideo**	indicates sync/async video
19	**polAClk**	audio clock polarity control (use rising edge if asserted)
18	**polTop**	polarity for the top bit
17	**polSDataErr**	error polarity control signal
16	**polAEnaStrb**	polarity control for caEna_Strb
15	**polAReq**	audio request polarity control
14	**polVClk**	video clock polarity control (use rising edge if asserted)
13	**polVEnaStrb**	polarity control for cvEna_Strb
12	**polVReq**	video request polarity control
11-5	**cmbDest**	6 bit comm bus destination
4-2	**streamType**	indicates stream type as follows: 000, 011, 100: reserved 001: PS program stream 010: TS transport stream 101: PES video only enabled 110: PES audio only enabled 111: PES audio and video enabled
1	**cdiEnable**	enable cdi after reset (resettable config bit)
0	**softReset**	soft reset signal

PES operating mode

Following command and status information is exchanged in the PES mode of operation:

Source	**Description**	**TX Status**	**MSLW**	**Dest.**
CDI	Normal mixed audio/video data transmission: 3 video long words, 1 audio long word	`0100 0000`	Data	MPE
CDI	Normal mixed audio/video data transmission: 1 video long word, 3 audio long words	`0110 0000`	Data	MPE
CDI	Normal mixed audio/video data transmission: 2 video long words, 2 audio long words	`0010 0000`	Data	MPE
CDI	Normal video only transmission	`0001 0000`	Data	MPE
CDI	Normal audio only transmission	`0000 0000`	Data	MPE
MPE	Command to long word align incoming video data. `vv` takes values 0, 1, 2, or 3 `aa` takes values 0, 1, 2, 3 Note: `00` is no align	`110a a0vv`	<NONE>	CDI

CDI	Response to long word align command from MPE: Used to mark the first transmission of aligned data with the data in 3 video and 1 audio long word format (whether alignment is of audio, video or both is indicated by bits[3..2])
	`10vv` is video only aligned
		`1100 10vv`
		`1100 0100`
		`1100 11vv`
			Data	MPE

Source	Description	TX Status	MSLW	Dest.
	0100 is audio only aligned 11vv is audio/video aligned Note: vv takes values 0, 1 or 2 (top down)			
CDI	Response to long word align command from MPE: Used to mark the first transmission of aligned data with the data in 1 video and 3 audio long word format (whether alignment is of audio, video or both is indicated by bits[3..2]) 1000 is video only aligned 01aa is audio only aligned 11aa is audio/video aligned Note: aa takes values 0, 1 or 2 (bottom up)	1110 1000 1110 01aa 1110 11aa	Data	MPE
CDI	Response to long word align command from MPE: Used to mark the first transmission of aligned data with the data in 2 video and 2 audio long word format (whether alignment is of audio, video or both is indicated by bits[3..2]) 10vx is video only align 01xa is audio only align 11av is audio/video align vv takes values 0, 1; aa takes values 0, 1; v takes values 0, 1; a takes values 0 {for MSLW3} or 1 {for MSLW2})	1010 10vx 1010 01xa 1010 11va	Data	MPE
CDI	Response to long word align command from MPE: Used to mark the first transmission of aligned data with the data in video only format vv takes values 0, 1, 2 or 3	1001 10vv	Data	MPE
CDI	Response to long word align command from MPE: Used to mark the first transmission of aligned data with the data in audio only format aa takes values 0, 1, 2 or 3	1000 01aa	Data	MPE

PS operating mode

Following command and status information is exchanged in the PS mode of operation:

Source	Description	TX Status	MSLW	Dest.
CDI	Normal data	0000 XXXX	Data	MPE
CDI	Data with byte error(s). Next transmission identifies the error byte positions	0001 XXXX	Data	MPE
CDI	Data with top byte: Indicates start of a new pack in the current transmission. This fact is established by the assertion of the top byte signal by the device upstream nnnn is the location of top byte	0010 nnnn	Data	MPE
CDI	Data with top byte and error(s): Indicates start of new pack with error(s) in current and/or new pack. Next transmission identifies error byte positions	0011 nnnn	Data	MPE

Source	Description	TX Status	MSLW	Dest.
	nnnn is the location of top byte			
MPE	Command to long word align incoming data xx takes values 00,01,10,11	1100 00xx	<NONE>	CDI
MPE	“Flush” command from MPE to force a Communication Bus transaction containing the residual bytes at the end of a PS pack	1110 0000	<NONE>	CDI
CDI	Aligned data: Response to long word align command from MPE. Used to mark the first transmission of aligned data. Data always aligned starting at MSLW position	0100 XXXX	Data	MPE
CDI	Aligned data with error(s): Response to long word align command from MPE. Used to mark the first transmission of aligned data. Also indicates that data has error(s). Next transmission identifies error byte positions. Data always aligned starting at MSLW position	0101 XXXX	Data	MPE
CDI	Aligned data with top byte: Response to long word align command from MPE. Used to mark the first transmission of aligned data. Indicates alignment was implemented at the tail end of a pack. Data always aligned starting at MSLW position nnnn is the location of top byte	0110 nnnn	Data	MPE
CDI	Aligned data with top byte and error(s): Response to long word align command from MPE. Used to mark the first transmission of aligned data. Indicates alignment was implemented at the tail end of a pack. Next transmission identifies error byte positions. Data always aligned starting at MSLW position nnnn is the location of top byte	0111 nnnn	Data	MPE
CDI	Sequel to 0001 0000 status indicating position(s) of byte error(s)	1001 0000	CDI_status (16 bit field identifies error byte positions)	MPE
CDI	Sequel to 0011 nnnn status indicating error byte positions	1011 0000	CDI_status (16 bit field identifies error byte positions)	MPE
CDI	Sequel to 0101 0000 status indicating position(s) of byte error(s)	1101 0000	CDI_status (16 bit field identifies error byte positions)	MPE
CDI	Sequel to 0111 nnnn status indicating error byte positions	1111 0000	CDI_status (16 bit field identifies error byte positions)	MPE
CDI	Response to “Flush” command from MPE. Communication Bus registers carry residual bytes at the end of a PS pack. Command and response executed during silence between pack bursts	1110 0000	Data	MPE

TS operating mode

This mode of operation is very similar to the PS mode except in the following ways. Instead of isolated error bytes, the entire transport packet is marked as bad. Consequently, error related status transmission needs only be made at the transition points between good and trashed transport packets. Also, a top byte signal may not always be present to mark the start of a new transport packet.

Following command and status information is exchanged in the TS mode of operation:

Source	Description	TX Status	MSLW	Dest

CDI	Normal data: A previous error packet code <0001 0000> is terminated at the first shift-in of good data. In this case, auxiliary field (bits[3..0]) points to start of good packet
	`0000` good pkt -> good pkt
	`nnnn` location of error pkt -> good pkt
		`0000 0000`
		`0000 nnnn`
			Data	MPE

CDI	Error packet data: A previous good packet code <0000 0000> is terminated at the first shift-in of error packet data. In this case, auxiliary field (bits[3..0]) points to start of error packet
	`0000` error pkt -> error pkt
	`nnnn` location of good pkt -> error pkt
		`0001 0000`
		`0001 nnnn`
			Data	MPE

CDI	Data with top byte: Indicates start of a new packet in the current transmission. This fact is established by the assertion of the top byte signal by the device upstream. A previous error packet code <0001 0000> is terminated at the first shift-in of good data. Auxiliary field (bits[3..0]) points to start of new packet `nnnn` location of new packet	`0010 nnnn`	Data	MPE
CDI	Data with top byte and error: Indicates start of an error packet. Auxiliary field (bits[3..0]) points to start of new error packet `nnnn` location of new packet	`0011 nnnn`	Data	MPE
MPE	Command to long word align incoming data xx takes values 00,01,10,11	`1100 00xx`	<NONE>	CDI
MPE	“Flush” command from MPE to force a Communication Bus transaction containing the residual bytes at the end of a TS packet	`1110 0000`	<NONE>	CDI

CDI	Aligned data: Response to long word align command from MPE. Used to mark the first transmission of aligned data. If previous code was <0000 0000> then this code signifies alignment of a good data packet. If previous code was <0001 0000> then this code identifies alignment of the tail end of the previous error packet, followed by the current good packet. In this case, auxiliary field (bits[3..0]) points to start of good packet. Data always aligned starting at MSLW position	`0100 0000`
		`0100 nnnn`
			Data	MPE

Source	Description	TX Status	MSLW	Dest
	0000 good pkt aligned nnnn location of error pkt -> good pkt			
CDI	Aligned data with error: Response to long word align command from MPE. Used to mark the first transmission of aligned data. If previous code was <0001 0000> then this code signifies alignment of error packet. If previous code was <0000 0000> then this code identifies alignment of the tail end of the previous packet with the current packet being trashed. In this case, auxiliary field (bits[3..0]) points to start of error packet. Data always aligned starting at MSLW position 0000 error pkt aligned nnnn location of good pkt -> error pkt	0101 0000 0101 nnnn	Data	MPE
CDI	Aligned data with top byte: Response to long word align command from MPE. Used to mark the first transmission of aligned data. This code identifies alignment of the tail end of the previous packet. Data always aligned starting at MSLW position nnnn top byte position	0110 nnnn	Data	MPE
CDI	Aligned data with top byte and error: Response to long word align command from MPE. Used to mark the first transmission of aligned data. This code identifies alignment of the tail end of the previous packet with the current packet being trashed. Data always aligned starting at MSLW position nnnn top byte position	0111 nnnn	Data	MPE
CDI	Response to "Flush" command from MPE. Communication Bus registers carry residual bytes at the end of a TS packet. Command and response executed during silence between packet bursts	1110 0000	Data	MPE

Coded Data Interface timing

The Coded Data Interface is designed to satisfy transport stream interface timing requirements outlined in the DAVIC 1.1 A0 Specification. This allows the interface to handle sustained bit rates of up to 72 Mbits/s. For program streams and video/audio elementary streams, request-enable type of handshaking allows data to be input in bursts of up to 16 bytes at a time. The interface is capable of handling CACLK and CVCLK rates of up to 25 MHz. The relationships between different signals in different operating modes are shown in Figure 9-Figure 15.

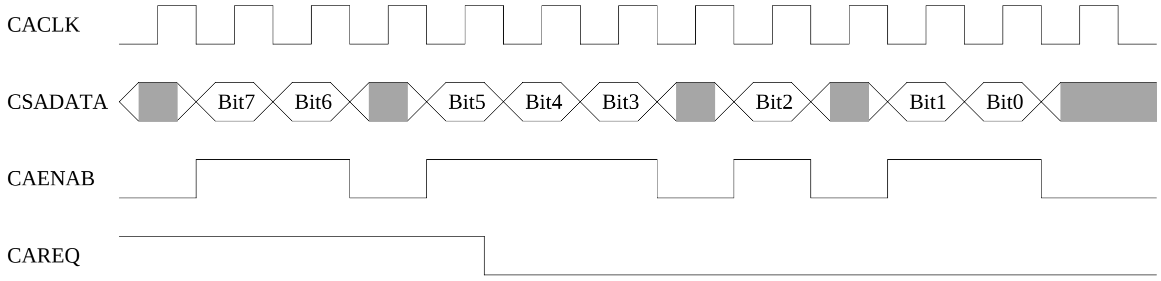


Figure 9: Synchronous Serial Audio mode of Coded Data Interface

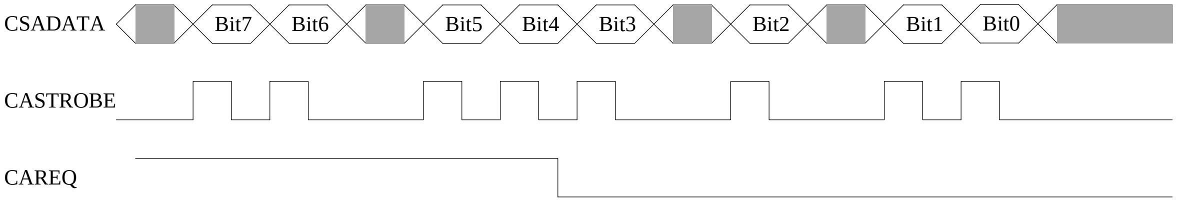


Figure 10: Asynchronous Serial Audio mode of Coded Data Interface

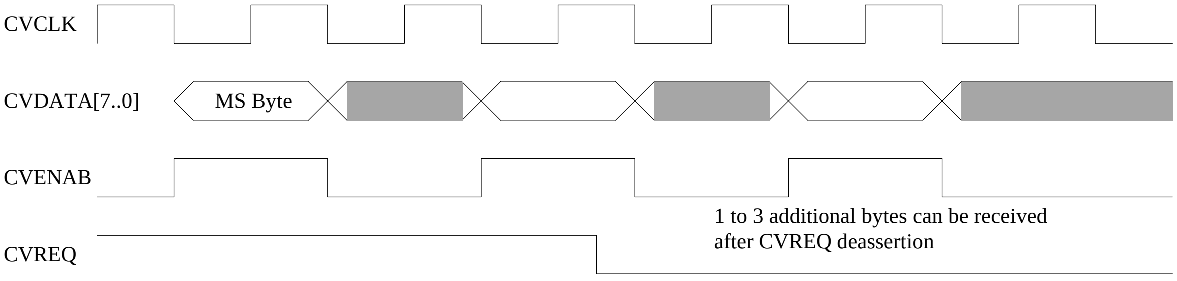


Figure 11: Synchronous Video mode of Coded Data Interface

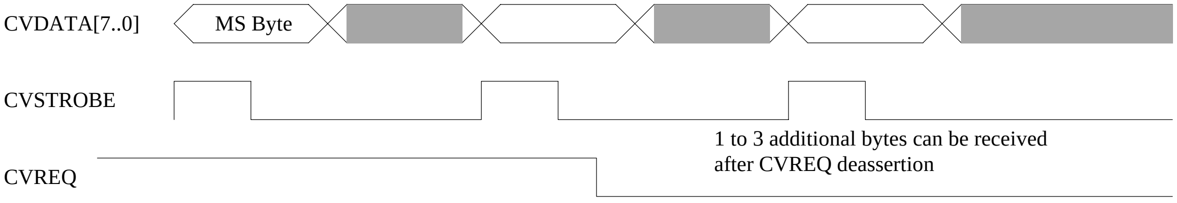


Figure 12: Asynchronous Video mode of Coded Data Interface

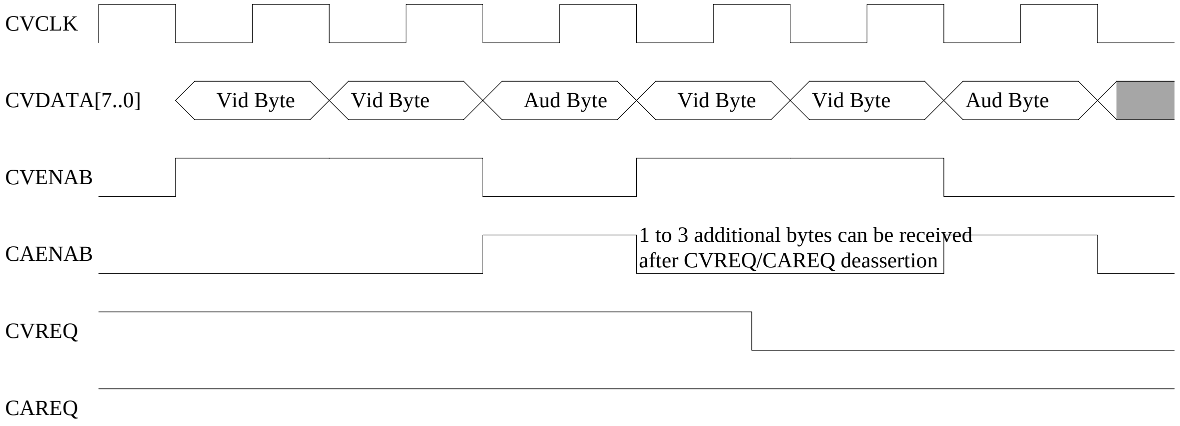


Figure 13: Synchronous multiplexed Parallel Video/Audio mode of Coded Data Interface

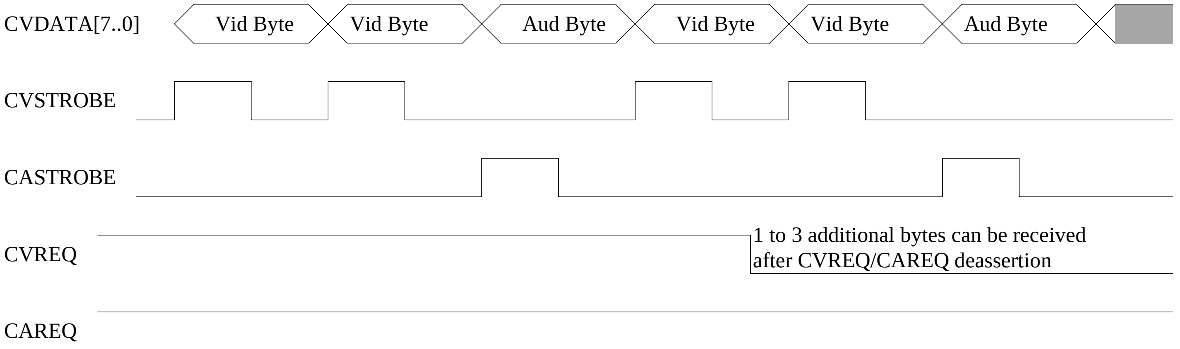


Figure 14: Asynchronous multiplexed Parallel Video/Audio mode of Coded Data Interface

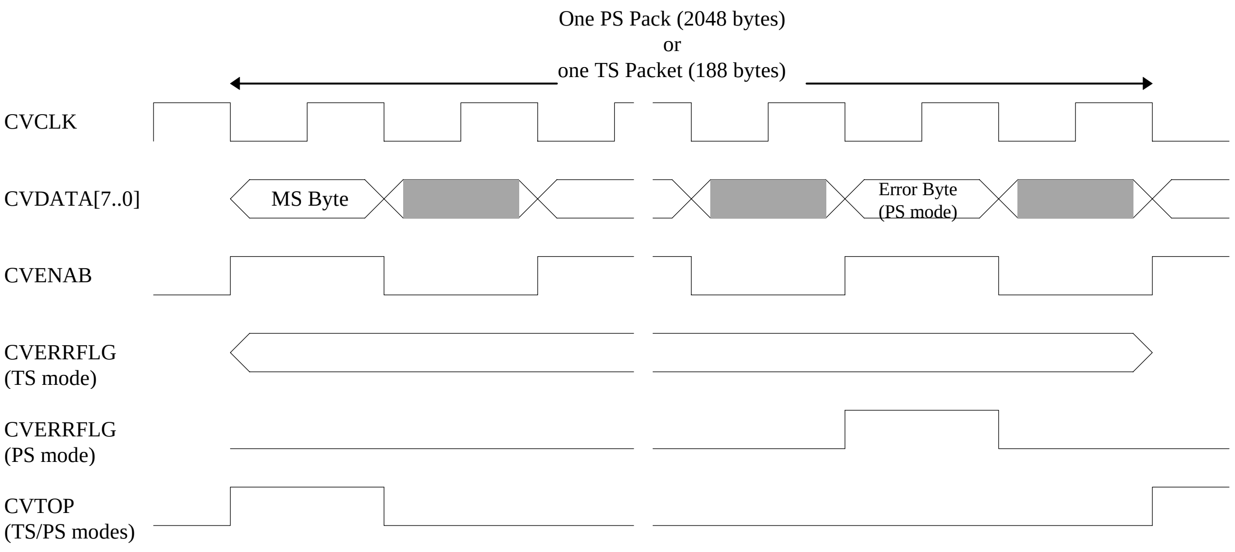


Figure 15: Transport Stream/Program Stream modes of Coded Data Interface

CDI Configuration register

Bit	Name	Description
0	**SoftReset**	Soft reset CDI
1	**CdiEnable**	Enable cdi for operation
4:2	**StreamType**	001: PS, 010: TS, 101: PES video, 110: PES audio, 111: PES audio an video
11:5	**CmbDest**	Communication Busdestination ID
12	**PolVReq**	Polarity video request
13	**PolVEnaStrb**	Polarity video enable
14	**PolVClk**	Polarity video clock
15	**PolAReq**	Polarity audio request
16	**PolAEnaStrb**	Polarity audio enable
17	**PolSDataErr**	Polarity serial data/error
18	**PolTop**	Polarity top bit
19	**PolAClk**	Polarity audio clock
20	**SyncVideo**	Synchronous video
21	**SyncAudio**	Synchronous audio
22	**ParallAudio**	Parallel audio
23	**ParallVideo**	Parallel video
24	**PolVReqClk**	Polarity video request edge
25	**PolAReqClk**	Polarity audio request edge
26	**StopCmbBus**	Disable Communication Bus
27	**CdiNoVReq**	Squash cvReq
28	**CdiNoAReq**	Squash caReq
29	**i2sModeEn**	Enable I^{2}S mode
30	**toshCDMode**	Special stuff for tosh. Subcode/error in CD mode

Notes:

- In I^{2}S mode, the interface is parallel synchronous, with top and error bits, ALL active high.
- A ‘1’ in polarity control bit means rising edge or active high on clocks. On strobes, a ‘1’ means falling edge is the capture edge.
- Error register: This 3-bit register is read together with the cdi configuration information :

  - 001 video overflow (and PS/TS)

  - 010 audio overflow

  - 100 bad receive Communication Bus info
- When using an asynchronous setup, make sure that the clock pins cvclk and/or caclk are tied high, to avoid spurious clocks when configuring

CSS Section:

Startup procedure:

Operation	**SW**	**HW**
1. Disc Insertion		
2. Get Lead-in Sector Information	Read single sector sector # 02FD02 (**rdSector**)	Wait for specified sector send to cdi
3. Extract disk reference key and vmlabs disk key	Process Data From Lead-In Sector	
4 Calculate intermediate master key A	Select master key A and issue **gIMK** command	Run LFSR with MKA and write IMKA 40 bits to register
5. Calculate disk key A and write back to hardware	Read register command (IMKA) Write keyInReg register command with disk key A	
6. Calculate intermediate disk key A	Get intermediate master disk key command (**gIMDK**)	Run LFSR with DKA and write IDKA 40 bits to register
7. Calculate disk key A' and compare to DKA if match then skip to 12	Read register command (IDKA)	
8 Calculate intermediate master key B	Select master key B and issue **gIMK** command	Run LFSR with MKB and write IMKB 40 bits to register
9. Calculate disk key B and write back to hardware	Read register command (IMKB) Write keyInReg register command with disk key B.	
10. Calculate intermediate disk key B	Get intermediate master disk key command (**gIMDK**)	Run LFSR with DKB and write IDKB 40 bits to register
11. Calculate disk key B' and compare to DKB. If no match => error	Read register command (IDKB)	
12. Get encrypted title key from disk	Select header bit and read sector command (**rdSector**)	Get sector containing encrypted title key,
13. Extract encrypted title key	Extract encrypted title key	
14. Calculate intermediate disk key	Write keyInReg register command with disk key	
15. Calculate intermediate disk key	Get intermediate disk key command (**gIDK**)	Run LFSR with DK and write IDK 40 bits register
16. Calculate title key	Read register command (IDK) Write keyInReg register command with title key	Store title key in safe place
17. Read Data Sectors…	Read Sector (**rdSector)**	+ Wait for incoming sector data to match sector number + Check scramble bit + Strip 12 byte headers off + Send descrambled data to FIFO or CDI

CDI IO Registers

IO registers are written through the Communication Bus setting Communication Bus info to: 1010_0000. The first 32 bits contain the right aligned address of the register with bit 31 indicating read(0) or write(1). The second 32 bits contain the right aligned write data (24 bits max). In case of a read the cdi will return data to the requester through the Communication Bus on the first 32 bits (right aligned).

Address	Register	Description	Default
0000	**SctID**	24 bit sector id	XXX
0001	**KeyInRegL *1**	24 LSB of key register	XXX
0010	**KeyInRegH *1**	16 MSB of key register	XX
0011	**CmdReg**	Command register (see below)	HALT
0100	**CtrlReg**	Control register (see below)	5’b11001
0101	**Status**	Status register (read only, see below)	2’b00
0110	**KeyOutRegL**	24 LSB of key register (read only)	XXX
0111	**KeyOutRegH**	16 MSB of key register (read only)	XX
1000	**PostFilter *1**	bytes after payload (0-7) (2060-2067)	000
1001	**FifoFullMargin**	When to indicate full fifo (0-4)	100

*1: No writes allowed in VMMode (softReset allowed, it resets to 0)

The **FifoFullMargin** register indicates how many empty bytes must be left in the FIFO before indicating FIFO full and deasserting the request line. This is important because it affects the input bandwidth available. In most cases 2 would be plenty since the source of data will send at the most 1 extra byte when the request line is deasserted. It must also be considered that there is at least 1 tick from the detection of the full condition and the deassertion of the request line, so another byte could sneak in. ‘0’ means there are no empty slots when full is indicated, and ‘1’ means there is 1 empty slot when full is indicated, etc.

Commands

The Command register is a 4-bit register at address 0x8 that holds the current command. The current commands are:

Code	Name	Description
0000	**NOP**	no operation
0001	**HALT**	stop gathering of data (state machines back to IDLE)
0010	**rdSector**	read sector (options: continuous, header)
0011	**reserved**	
0100	**gITDK**	get intermediate title disk key
0101	**gIMDK**	get intermediate master disk key
0110	**gIMK**	get intermediate master key (options: A or B)
0111	**reserved**	

The non-continuous rdSector command (read 1 sector) has the problem of leaving data in the pipe because nobody is ‘pushing’ behind it. The Comm-Bus will send only multiples of 16 packets. Therefore, to avoid this problem, the read 1 sector command will require a 4 byte postfilter to get complete data (no align commands allowed of course)

The 40 bits of reply to some of the above commands will come in bits {(47:32),(23:0)}

Control Register:

Bit	Name	Description	Default
0	**CssSoftReset**	soft reset for CSS only	1
1	**CssBypass *1**	bypass css logic	1
2	**Continuous**	continuous read	0
3	**SectFilterOff**	turn sector filter off	1
4	**MasterA**	choose masterkey A	1
5	**DescrOff *1**	force no descrambling of scrambled sectors	0
6	**VMMode *2**	vmlabs descramble mode	0
7	**Kill12 *1**	kill first 12 bytes of each sector.	0

*1: No writes allowed in VMMode, no softReset

*2: can be written to, only once, no softReset

Status register

Bit	Name	Description
0	**fifoOvfl**	Fifo overflow
1	**fifoUnfl**	Fifo underflow
7-2	**reserved**	Available bits

Communication Bus info byte, for writes and reads : 1010_0000

reply will be: 1000_0000

Performance

Through CSS with LFSR involvement, the maximum data rate possible is 54 Mbits/second or 6.75 MBytes/second. Otherwise it is 36MBytes/second.

When data is passed from the audio in I2S inputs, it is parallel sync at 27 MHz in bursts of four, therefore it can at the most be 4 bytes every 13 bytes (4 bytes + 9 idle cycles).

MOTION COMPENSATION UNIT

The Motion Compensation Unit (MCU) performs the motion compensation operation for the MPEG decode specific hardware and is closely linked with the DMA controller. The design is implemented around four 48x40 banks of memory. In normal MPEG operations, these memory banks are written directly with the IDCT data coming out of the BDU. They can also be read and written through Communication Bus operations to allow calculated IDCT data to be written directly into the MCU or to enable the use of the memory banks as scratch space (32-bit wide only).

Another function of the MCU is for hardware semaphores. When not decoding MPEG, the 8 LSBs of every location in the internal banks (192) can be used as a hardware semaphore (is that enough for you?). When decoding MPEG, there are additional 8 bits that can be accessed as hardware semaphores as shown below.

Finally, the MCU memory can also be used as scratch memory space.

The MCU is controlled by an interface available through the DMA controller’s Communication Bus interface, and by some MCU specific DMA commands. MCU commands are 32 bits wide and are sent with a normal Communication Bus packet through the DMA unit. The DMA unit will identify the command for the MCU and forward it. Up to four MCU commands can be sent in a single Communication Bus message. If the MCU replies with some data, the DMA will send a Communication Bus packet to the original requester with the 32 reply bits and the 3 32-bit words of garbage.

MCU Communication Bus Interface

This interface is available through the DMA controller’s Communication Bus interface. Refer to that section for further details.

mcuTS Test and set semaphore (memory)

Tests the state of one or more semaphore bits, returns their state, and sets them, as determined by the mask. The return Communication Bus packet has the 8 status bits, in the 8 LSB positions of the first 32-bit word of the Communication Bus packet.

Bit	Name	Description
31-25	**Header**	1 111 000
23-16	**Adx**	Memory address
7-0	**mask**	Select which of the 8 semaphores to operate on

mcuTC Test and clear semaphore (memory)

Tests the state of one or more semaphore bits, returns their state, and clears them, as determined by the mask. The return Communication Bus packet has the 8 status bits, in the 8 LSB positions of the first 32-bit word of the Communication Bus packet.

Bit	Name	Description
31-25	**Header**	1 111 001
23-16	**Adx**	Memory address
7-0	**mask**	Select which of the 8 semaphores to operate on

wrtDCT Write a DCT value

Writes a DCT value sequentially into MCU RAM. The MCU Ram will be filled in raster scan order, i.e. the dataL of the first dct write will be the DC term of the block and the dataH will be the AC term to the right of it. The next write dct MUST contain the next two dct values to the right, since four values are written to the RAM at a time.

When the last dct values are written for that block the MCU will generate a dct done interrupt.

Bit	Name	Description
31-25	**Header**	0 111 100
24-16	**dataL**	DCT value (low)
15	**dctType**	0 = frame, 1 = field
14	**newBlock**	Indicates new block start. Set with first dct pair of each block only
13-11	**blockNum**	Indicates which block number (0 to 5)
10	**newMBlock**	Indicates new Mblock. Set only with the first two samples of the macro-block.
9	**reserved**	write 0
8-0	**dataH**	DCT value (high)

wrtAdxDh Write to MCU RAM

Writes a value into MCU RAM. This forms a pair with the next command to write an entire 32-bit location. This command **must** precede wrAdxDl since the actual write operation occurs during wrAdxDl. These 16 bits are the high bits of the 32-bit word.

If the testMode bit is set (see mcuTSreg), this command doesn’t do much, but it still needs to be executed right before wrAdxDl to set the address in the hardware. In normal mode, bits 39, 30,29 20,19, 10, 9, 0 of each entry of each RAM are not tested. When test mode is set, these bits are treated as a byte and are accessible from Communication Bus.

Bit	Name	Description
31-25	**Header**	0 111 101
23-16	**Adx**	Memory address (0-191 decimal)
15-0	**dataH**	16 bits of data

wrAdxDl Write to MCU RAM

Writes a value into MCU RAM. This forms a pair with the previous command to write an entire 32-bit location. This command **must** follow wrtAdxDh since the actual write operation happens during this command These 16 bits are the low bits of the 32-bit word.

If the testMode bit is set (see mcuTSreg), this command will write the 8 bits (of the MCU RAM) that are not accessed in non-test mode.

Bit	Name	Description
31-25	**Header**	0 111 110
23-16	**Adx**	Memory address (0-191 decimal)
15-0	**dataL**	16 bits of data

rdAdx Read from MCU RAM

Reads a 32-bit value from MCU RAM. The data will be returned as the first 32 data bits in the reply Communication Bus message.

Bit	Name	Description
31-25	**Header**	1 111 111
23-16	**Adx**	Memory address (0-191 decimal)
15-0	**reserved**	write 0

mcuTSreg Test and set semaphore register

Tests the state of one or more semaphore register bits, returns their state, and sets them, as determined by the mask. This register contains the 8 semaphores always available (even if the MPEG engine is running). The return Communication Bus packet has the 8 status bits, in the 8 LSB positions of the first 32-bit word of the Communication Bus message.

If Bit 8 is set, it will put the MCU Ram in a mode (testMode) where the bits that are not accessible through the above MCU read and write commands are accessible. There will be a return packet in this case too.

Bit	Name	Description
31-25	**Header**	1 111 010
8	**Test**	Enables extra RAM data in test mode if set, should be zero for normal operation
7-0	**mask**	Select which of the 8 semaphores to operate on

mcuTCreg Test and clear semaphore register

Tests the state of one or more semaphore register bits, returns their state, and clears them, as determined by the mask. This register contains the 8 semaphores always available (even if the MPEG engine is running). The return Communication Bus packet has the 8 status bits, in the 8 LSB positions of the first 32-bit word of the Communication Bus message.

If Bit 8 is set, it will take the MCU Ram back into normal operation (default). There will be a return packet in this case too.

Bit	Name	Description
31-25	**Header**	1 111 011
8	**Test**	Disables extra RAM data in test mode (if set, 0 for normal op)
7-0	**mask**	Select which of the 8 semaphores to operate on

ARIES 3 ELECTRICAL SPECIFICATIONS

Aries 3 Package Options

Aries 3 is available in the following package options:

ARIES30-BA3	256 pin BGA	For new Aries 3 specific board designs.
ARIES30-BA2	256 pin BGA	For Aries 2 drop-in replacement.
ARIES30-QA2	208 pin PQFP	For new Aries 3 specific board designs.

The sections below discuss the details of these.

Qualification levels

Aries 3 parts will be available in three qualification levels, according to the qualification status when they are manufactured:

-ES	Engineering Samples
-XC	Conditional Qualification
-VC	Fully qualified.

Part Numbering

The part numbering is a combination of the package option, followed by the qualification level. For example: ARIES30-BA2-ES is an Aries 2 drop-in replacement part qualified as an engineering sample.

Aries 3 Pinout - QFP-208 Package

QFP-208 Package

Detailed package drawings are available from VM Labs.

Pin pitch: 0.50mm.

Package dimension: 28mm X 28mm.

	208	207	206	.	.	.	159	158	157	
1										156
2										155
3										154
.										.
.										.
.										.
.										.
.										.
.										.
.										.
.										.
.										.
.										.
50										107
51										106
52										105
	53	54	55	.	.	.	102	103	104	

QFP-208 Pinout Ordered by Pin Number

Pin	Signal Name	Pin	Signal Name	Pin	Signal Name	Pin	Signal Name
1	X_pll1_avss	53	GND_18	105	X_sysa_19	157	GND_18
2	X_pll1_avdd	54	VDD_18	106	X_sysa_20	158	VDD_18
3	X_pll1_dvss	55	X_hsync	107	X_sysa_21	159	X_sd_a_3
4	X_pll1_dvdd	56	X_sysd_0	108	X_sysa_22	160	X_sd_a_4
5	GND_33	57	X_sysd_1	109	X_sysdramcs	161	X_sd_a_2
6	X_pll_clki	58	X_sysd_2	110	X_sysrw	162	X_sd_a_5
7	VDD_33	59	X_sysd_3	111	X_syscas	163	X_sd_a_1
8	X_rom_cs_b	60	X_sysd_4	112	X_sysbg	164	X_sd_a_6
9	X_gpio_0	61	X_sysd_5	113	GND_33	165	X_sd_a_0
10	X_gpio_1	62	GND_33	114	VDD_33	166	GND_33
11	X_gpio_2	63	VDD_33	115	X_sysbb	167	VDD_33
12	X_gpio_3	64	X_sysd_6	116	X_sysgpcs0	168	X_sd_a_7
13	GND_18	65	X_sysd_7	117	X_syscs	169	X_sd_a_10
14	VDD_18	66	X_sysd_8	118	GND_18	170	X_sd_a_8
15	X_gpio_4	67	X_sysd_9	119	X_sysbclk	171	X_sd_a_12
16	X_gpio_5	68	X_sysd_10	120	VDD_18	172	X_sd_a_9
17	X_gpio_6	69	X_sysd_11	121	X_sysoe	173	X_sd_a_13

Pin	Signal Name
18	X_gpio_7
19	X_gpio_8
20	X_gpio_9
21	X_gpio_10
22	X_gpio_11
23	X_gpio_12
24	X_gpio_13
25	X_gpio_14
26	X_gpio_15
27	GND_18
28	X_reseti_b
29	VDD_18
30	X_cp_ena_b
31	X_cp_clk
32	GND_33
33	VDD_33
34	X_cp_dout1
35	X_cp_dout2
36	X_cp_din1
37	X_cp_din2
38	X_test
39	GND_18
40	VDD_18
41	X_vdata_0
42	X_vdata_1
43	X_vdata_2
44	X_vdata_3
45	X_vdata_4
46	X_vdata_5
47	X_vdata_6
48	X_vdata_7
49	GND_33
50	X_vclk
51	VDD_33
52	X_field
70	X_sysd_12
71	GND_33
72	VDD_33
73	X_sysd_13
74	X_sysd_14
75	X_sysd_15
76	X_sysgpcs1
77	X_syswe
78	GND_18
79	VDD_18
80	X_sysa_2
81	X_sysa_3
82	GND_33
83	VDD_33
84	X_sysa_4
85	X_sysa_5
86	X_sysa_6
87	X_sysa_7
88	X_sysa_8
89	X_sysa_9
90	X_sysa_10
91	GND_33
92	VDD_33
93	X_sysa_11
94	X_sysa_12
95	X_sysa_13
96	X_sysa_14
97	X_sysa_15
98	X_sysa_16
99	X_sysa_17
100	VDD_33
101	GND_33
102	X_sysa_18
103	VDD_18
104	GND_18
122	X_ai_data
123	X_ai_bclk
124	X_ai_wclk
125	X_cvdata_0
126	X_cvdata_1
127	X_cvdata_2
128	X_cvdata_3
129	X_cvdata_4
130	GND_18
131	VDD_18
132	X_cvdata_5
133	X_cvdata_6
134	X_cvdata_7
135	X_cvreq
136	X_cvenab
137	X_cvclk
138	X_caenab
139	X_casdata
140	X_careq
141	X_caclk
142	GND_18
143	X_aclk
144	VDD_18
145	X_sbclk
146	X_sdat_0
147	X_swclk
148	X_sdat_2
149	X_sdat_1
150	X_spdif
151	VDD_33
152	GND_33
153	X_pll2_dvdd
154	X_pll2_dvss
155	X_pll2_avdd
156	X_pll2_avss
174	X_sd_a_11
175	X_sd_cs_b_0
176	GND_33
177	VDD_33
178	X_sd_a_14
179	X_sd_ras_b
180	X_sd_cas_b
181	GND_18
182	X_sd_clk
183	VDD_18
184	X_sd_we_b
185	X_sd_dqm_0
186	X_sd_dqm_1
187	X_sd_dq_7
188	GND_33
189	VDD_33
190	X_sd_dq_8
191	X_sd_dq_6
192	X_sd_dq_9
193	X_sd_dq_5
194	X_sd_dq_10
195	X_sd_dq_4
196	X_sd_dq_11
197	X_sd_dq_3
198	GND_33
199	VDD_33
200	X_sd_dq_12
201	X_sd_dq_2
202	X_sd_dq_13
203	X_sd_dq_1
204	X_sd_dq_14
205	X_sd_dq_0
206	X_sd_dq_15
207	VDD_18
208	GND_18

QFP-208 Pinout Ordered by Signal Name

Pin	Signal Name
13	GND_18
27	GND_18
39	GND_18
53	GND_18
78	GND_18
104	GND_18
118	GND_18
130	GND_18
142	GND_18
143	X_aclk
123	X_ai_bclk
122	X_ai_data
124	X_ai_wclk
141	X_caclk
138	X_caenab
140	X_careq
139	X_casdata
31	X_cp_clk
28	X_reseti_b
8	X_rom_cs_b
145	X_sbclk
165	X_sd_a_0
163	X_sd_a_1
161	X_sd_a_2
159	X_sd_a_3
160	X_sd_a_4
162	X_sd_a_5
88	X_sysa_8
89	X_sysa_9
90	X_sysa_10
93	X_sysa_11
94	X_sysa_12
95	X_sysa_13
96	X_sysa_14
97	X_sysa_15
98	X_sysa_16

Pin	Signal Name
157	GND_18
181	GND_18
208	GND_18
5	GND_33
32	GND_33
49	GND_33
62	GND_33
71	GND_33
82	GND_33
91	GND_33
101	GND_33
113	GND_33
152	GND_33
166	GND_33
176	GND_33
188	GND_33
198	GND_33
14	VDD_18
29	VDD_18
40	VDD_18
54	VDD_18
79	VDD_18
103	VDD_18
120	VDD_18
131	VDD_18
144	VDD_18
158	VDD_18
183	VDD_18
207	VDD_18
7	VDD_33
33	VDD_33
51	VDD_33
63	VDD_33
72	VDD_33
83	VDD_33
92	VDD_33
100	VDD_33
114	VDD_33
151	VDD_33
167	VDD_33
177	VDD_33
189	VDD_33
199	VDD_33

Pin	Signal Name
36	X_cp_din1
37	X_cp_din2
34	X_cp_dout1
35	X_cp_dout2
30	X_cp_ena_b
137	X_cvclk
125	X_cvdata_0
126	X_cvdata_1
127	X_cvdata_2
128	X_cvdata_3
129	X_cvdata_4
132	X_cvdata_5
133	X_cvdata_6
134	X_cvdata_7
136	X_cvenab
135	X_cvreq
52	X_field
9	X_gpio_0
10	X_gpio_1
11	X_gpio_2
12	X_gpio_3
15	X_gpio_4
16	X_gpio_5
17	X_gpio_6
18	X_gpio_7
19	X_gpio_8
20	X_gpio_9
21	X_gpio_10
22	X_gpio_11
23	X_gpio_12
24	X_gpio_13
25	X_gpio_14
26	X_gpio_15
55	X_hsync
6	X_pll_clki
2	X_pll1_avdd
1	X_pll1_avss
4	X_pll1_dvdd
3	X_pll1_dvss
155	X_pll2_avdd
156	X_pll2_avss
153	X_pll2_dvdd
154	X_pll2_dvss

Pin	Signal Name
164	X_sd_a_6
168	X_sd_a_7
170	X_sd_a_8
172	X_sd_a_9
169	X_sd_a_10
174	X_sd_a_11
171	X_sd_a_12
173	X_sd_a_13
178	X_sd_a_14
180	X_sd_cas_b
182	X_sd_clk
175	X_sd_cs_b_0
205	X_sd_dq_0
203	X_sd_dq_1
201	X_sd_dq_2
197	X_sd_dq_3
195	X_sd_dq_4
193	X_sd_dq_5
191	X_sd_dq_6
187	X_sd_dq_7
190	X_sd_dq_8
192	X_sd_dq_9
194	X_sd_dq_10
196	X_sd_dq_11
200	X_sd_dq_12
202	X_sd_dq_13
204	X_sd_dq_14
206	X_sd_dq_15
185	X_sd_dqm_0
186	X_sd_dqm_1
179	X_sd_ras_b
184	X_sd_we_b
146	X_sdat_0
149	X_sdat_1
148	X_sdat_2
150	X_spdif
147	X_swclk
80	X_sysa_2
81	X_sysa_3
84	X_sysa_4
85	X_sysa_5
86	X_sysa_6
87	X_sysa_7

Pin	Signal Name
99	X_sysa_17
102	X_sysa_18
105	X_sysa_19
106	X_sysa_20
107	X_sysa_21
108	X_sysa_22
115	X_sysbb
119	X_sysbclk
112	X_sysbg
111	X_syscas
117	X_syscs
56	X_sysd_0
57	X_sysd_1
58	X_sysd_2
59	X_sysd_3
60	X_sysd_4
61	X_sysd_5
64	X_sysd_6
65	X_sysd_7
66	X_sysd_8
67	X_sysd_9
68	X_sysd_10
69	X_sysd_11
70	X_sysd_12
73	X_sysd_13
74	X_sysd_14
75	X_sysd_15
109	X_sysdramcs
116	X_sysgpcs0
76	X_sysgpcs1
121	X_sysoe
110	X_sysrw
77	X_syswe
38	X_test
50	X_vclk
41	X_vdata_0
42	X_vdata_1
43	X_vdata_2
44	X_vdata_3
45	X_vdata_4
46	X_vdata_5
47	X_vdata_6
48	X_vdata_7

Aries 3 Pinout - BGA-256 Package

BGA-256 Package

Detailed package drawings are available from VM Labs.

Ball pitch: 1.27mm.

Package dimension: 27mm X 27mm.

Variation between ARIES30-BA2 and ARIES30-BA3 Packages

Aries 3 is available in two BGA package options. ARIES30-BA2 is intended to be a drop-in replacement for Aries 2, and ARIES30-BA3 is intended for new designs. The pinout tables below describe the ARIES30-BA3 variant. The following balls are all no connects on the ARIES30-BA2 variant:

Ball	Signal Name	Comment
J3	X_pll1_avss	Bonded to internal ground ring.
H1	X_pll1_avdd	Bonded to internal 1.8V power ring.
L2	X_pll1_dvss	Bonded to internal ground ring.
J1	X_pll1_dvdd	Bonded to internal 1.8V power ring.
H18	X_pll2_dvdd	Bonded to internal 1.8V power ring.
D19	X_pll2_dvss	Bonded to internal ground ring.
J17	X_pll2_avdd	Bonded to internal 1.8V power ring.
H20	X_pll2_avss	Bonded to internal ground ring.
B12	X_sd_a_14	Not used – this is only required for SDRAM larger than 64 Mbit. On Aries 2 this is used for X_sd_clk_in, which is no longer required.

BGA-256 Pinout Ordered by Ball Designator

Ball	Signal Name
A1	GND
A2	X_sd_dq_0
A3	X_sd_dq_1
A4	X_sd_dq_2
A5	X_sd_dq_3
A6	X_sd_dq_4
A7	X_sd_dq_5
A8	X_sd_dq_6
A9	X_sd_dq_7
A10	X_sd_dqm_0
A11	X_sd_we_b
A12	X_sd_cas_b
A13	X_sd_ras_b
A14	X_sd_cs_b_0
A15	X_sd_a_11
A16	X_sd_a_10
A17	X_sd_a_0
A18	X_sd_a_1
A19	X_sd_a_2
D5	VDD_33
D6	VDD_18
D7	VDD_33
D8	GND
D9	X_gpio_24
D10	VDD_33
D11	VDD_18
D12	X_gpio_18
D13	GND
D14	X_sdat_2
D15	VDD_18
D16	X_gpio_17
D17	GND
D18	VDD_33
D19	X_pll2_dvss
D20	X_careq
E1	X_rom_lat_0
E2	X_rom_d_7
E3	X_rom_d_6
L1	X_cp_ena_b
L2	X_pll1_dvss
L3	X_pll_ref
L4	X_gpio_16
L17	VDD_18
L18	X_sysoe
L19	X_sysbclk
L20	X_sys_rdy_b
M1	X_vid_6
M2	X_vid_7
M3	X_gpio_0
M4	X_gpio_1
M17	X_sysbg
M18	X_sysbb
M19	X_sysgpcs0
M20	X_syscs
N1	X_gpio_2
N2	X_gpio_3
N3	X_gpio_4
U17	GND
U18	X_sysa_14
U19	X_sysa_13
U20	X_sysa_16
V1	X_reseti_b
V2	X_cp_dout2
V3	X_cp_din2
V4	X_vdata_3
V5	X_vdata_7
V6	X_hsync
V7	X_sysd_2
V8	X_sysd_5
V9	X_sysd_9
V10	X_sysd_12
V11	X_sysd_16
V12	X_sysd_20
V13	VDD_33
V14	X_sysd_27
V15	X_sysd_31

Ball	Signal Name
A20	X_sd_a_3
B1	X_sd_dq_15
B2	X_sd_dq_14
B3	X_sd_dq_13
B4	X_sd_dq_12
B5	X_sd_dq_11
B6	X_sd_dq_10
B7	X_sd_dq_9
B8	X_sd_dq_8
B9	X_gpio_22
B10	X_sd_dqm_1
B11	X_sd_clk
B12	X_sd_a_14
B13	X_sd_a_9
B14	X_sd_cs_b_1
B15	X_sd_a_8
B16	X_sd_a_7
B17	X_sd_a_6
B18	X_sd_a_5
B19	X_sd_a_4
B20	X_sd_a_12
C1	X_rom_d_4
C2	X_rom_d_0
C3	VDD_33
C4	X_gpio_25
C5	VDD_33
C6	X_spdif
C7	VDD_33
C8	VDD_33
C9	X_gpio_23
C10	X_gpio_21
C11	X_gpio_20
C12	X_gpio_19
C13	VDD_33
C14	X_sdat_1
C15	VDD_33
C16	X_swclk
C17	VDD_33
C18	X_sdat_0
C19	X_sbclk
C20	X_sd_a_13
D1	X_rom_d_5
D2	X_rom_d_1
D3	X_rom_d_2
D4	GND
E4	X_rom_d_3
E17	X_caclk
E18	VDD_33
E19	X_casdata
E20	X_cvenab
F1	VDD_33
F2	X_rom_cs_b
F3	X_rom_lat_1
F4	VDD_18
F17	VDD_18
F18	X_caenab
F19	X_cvreq
F20	X_cvdata_6
G1	X_viclk
G2	X_rom_oe_b
G3	X_rom_we_b
G4	X_rom_lat_2
G17	X_cvclk
G18	X_cvdata_7
G19	X_cvdata_5
G20	X_cvdata_4
H1	X_pll1_avdd
H2	X_vid_1
H3	X_vid_0
H4	GND
H17	GND
H18	X_pll2_dvdd
H19	X_aclk
H20	X_pll2_avss
J1	X_pll1_dvdd
J2	X_pll_clki
J3	X_pll1_avss
J4	X_vid_2
J17	X_pll2_avdd
J18	X_cvdata_3
J19	X_cvdata_2
J20	X_cvdata_1
K1	X_vid_5
K2	X_vid_3
K3	X_vid_4
K4	VDD_18
K17	X_cvdata_0
K18	X_ai_wclk
K19	X_ai_bclk
K20	X_ai_data
N4	GND
N17	GND
N18	X_syscas
N19	VDD_33
N20	X_sysbr
P1	X_gpio_5
P2	X_gpio_6
P3	X_gpio_8
P4	X_gpio_10
P17	X_sysa_21
P18	X_sysa_24
P19	X_sysdramcs
P20	X_sysrw
R1	X_gpio_7
R2	VDD_33
R3	X_gpio_11
R4	VDD_18
R17	VDD_18
R18	X_sysa_20
R19	X_sysa_22
R20	X_sysa_23
T1	X_gpio_9
T2	X_gpio_12
T3	X_gpio_14
T4	X_cp_clk
T17	X_sysa_15
T18	X_sysa_17
T19	X_sysa_18
T20	X_sysa_19
U1	X_gpio_13
U2	X_gpio_15
U3	X_cp_dout1
U4	GND
U5	X_vdata_4
U6	VDD_18
U7	VDD_33
U8	GND
U9	X_sysd_8
U10	VDD_18
U11	X_sysd_17
U12	X_sysd_21
U13	GND
U14	X_sysd_30
U15	VDD_18
U16	X_sysa_4
V16	X_sysa_2
V17	X_sysa_5
V18	X_sysa_8
V19	X_sysa_12
V20	VDD_33
W1	X_cp_din1
W2	X_test
W3	X_vdata_1
W4	X_vdata_2
W5	X_vclk
W6	X_sysd_0
W7	X_sysd_3
W8	X_sysd_6
W9	X_sysd_10
W10	VDD_33
W11	X_sysd_15
W12	X_sysd_19
W13	X_sysd_23
W14	X_sysd_25
W15	X_sysd_28
W16	X_sysgpcs1
W17	X_sysa_3
W18	X_sysa_6
W19	X_sysa_9
W20	X_sysa_11
Y1	X_vdata_0
Y2	VDD_33
Y3	X_vdata_5
Y4	X_vdata_6
Y5	X_field
Y6	X_sysd_1
Y7	X_sysd_4
Y8	X_sysd_7
Y9	X_sysd_11
Y10	X_sysd_13
Y11	X_sysd_14
Y12	X_sysd_18
Y13	X_sysd_22
Y14	X_sysd_24
Y15	X_sysd_26
Y16	X_sysd_29
Y17	X_syswe
Y18	VDD_33
Y19	X_sysa_7
Y20	X_sysa_10

BGA-256 Pinout Ordered by Signal Name

Ball	Signal Name
A1	GND
D4	GND
D8	GND
D13	GND
D17	GND
H4	GND
H17	GND
N4	GND
N17	GND
U4	GND
U8	GND
U13	GND
U17	GND
D6	VDD_18
D11	VDD_18
D15	VDD_18
F4	VDD_18
F17	VDD_18
K4	VDD_18
L17	VDD_18
R4	VDD_18
R17	VDD_18
U6	VDD_18
U10	VDD_18
U15	VDD_18
C3	VDD_33
C5	VDD_33
C7	VDD_33
C8	VDD_33
C13	VDD_33
C15	VDD_33
C17	VDD_33
D5	VDD_33
D7	VDD_33
D10	VDD_33
D18	VDD_33
E18	VDD_33
F1	VDD_33
N19	VDD_33
R2	VDD_33
U7	VDD_33
V13	VDD_33
V20	VDD_33
W10	VDD_33
Y2	VDD_33
Y18	VDD_33
H19	X_aclk
J18	X_cvdata_3
G20	X_cvdata_4
G19	X_cvdata_5
F20	X_cvdata_6
G18	X_cvdata_7
E20	X_cvenab
F19	X_cvreq
Y5	X_field
M3	X_gpio_0
M4	X_gpio_1
N1	X_gpio_2
N2	X_gpio_3
N3	X_gpio_4
P1	X_gpio_5
P2	X_gpio_6
R1	X_gpio_7
P3	X_gpio_8
T1	X_gpio_9
P4	X_gpio_10
R3	X_gpio_11
T2	X_gpio_12
U1	X_gpio_13
T3	X_gpio_14
U2	X_gpio_15
L4	X_gpio_16
D16	X_gpio_17
D12	X_gpio_18
C12	X_gpio_19
C11	X_gpio_20
C10	X_gpio_21
B9	X_gpio_22
C9	X_gpio_23
D9	X_gpio_24
C4	X_gpio_25
V6	X_hsync
J2	X_pll_clki
L3	X_pll_ref
H1	X_pll1_avdd
J3	X_pll1_avss
J1	X_pll1_dvdd
L2	X_pll1_dvss
J17	X_pll2_avdd
H20	X_pll2_avss
H18	X_pll2_dvdd
D19	X_pll2_dvss
V1	X_reseti_b
F2	X_rom_cs_b
A20	X_sd_a_3
B19	X_sd_a_4
B18	X_sd_a_5
B17	X_sd_a_6
B16	X_sd_a_7
B15	X_sd_a_8
B13	X_sd_a_9
A16	X_sd_a_10
A15	X_sd_a_11
B20	X_sd_a_12
C20	X_sd_a_13
B12	X_sd_a_14
A12	X_sd_cas_b
B11	X_sd_clk
A14	X_sd_cs_b_0
B14	X_sd_cs_b_1
A2	X_sd_dq_0
A3	X_sd_dq_1
A4	X_sd_dq_2
A5	X_sd_dq_3
A6	X_sd_dq_4
A7	X_sd_dq_5
A8	X_sd_dq_6
A9	X_sd_dq_7
B8	X_sd_dq_8
B7	X_sd_dq_9
B6	X_sd_dq_10
B5	X_sd_dq_11
B4	X_sd_dq_12
B3	X_sd_dq_13
B2	X_sd_dq_14
B1	X_sd_dq_15
A10	X_sd_dqm_0
B10	X_sd_dqm_1
A13	X_sd_ras_b
A11	X_sd_we_b
C18	X_sdat_0
C14	X_sdat_1
D14	X_sdat_2
C6	X_spdif
C16	X_swclk
L20	X_sys_rdy_b
V16	X_sysa_2
W17	X_sysa_3
U16	X_sysa_4
V17	X_sysa_5
W18	X_sysa_6
P18	X_sysa_24
M18	X_sysbb
L19	X_sysbclk
M17	X_sysbg
N20	X_sysbr
N18	X_syscas
M20	X_syscs
W6	X_sysd_0
Y6	X_sysd_1
V7	X_sysd_2
W7	X_sysd_3
Y7	X_sysd_4
V8	X_sysd_5
W8	X_sysd_6
Y8	X_sysd_7
U9	X_sysd_8
V9	X_sysd_9
W9	X_sysd_10
Y9	X_sysd_11
V10	X_sysd_12
Y10	X_sysd_13
Y11	X_sysd_14
W11	X_sysd_15
V11	X_sysd_16
U11	X_sysd_17
Y12	X_sysd_18
W12	X_sysd_19
V12	X_sysd_20
U12	X_sysd_21
Y13	X_sysd_22
W13	X_sysd_23
Y14	X_sysd_24
W14	X_sysd_25
Y15	X_sysd_26
V14	X_sysd_27
W15	X_sysd_28
Y16	X_sysd_29
U14	X_sysd_30
V15	X_sysd_31
P19	X_sysdramcs
M19	X_sysgpcs0
W16	X_sysgpcs1
L18	X_sysoe
P20	X_sysrw
Y17	X_syswe
W2	X_test
W5	X_vclk

Ball	Signal Name
K19	X_ai_bclk
K20	X_ai_data
K18	X_ai_wclk
E17	X_caclk
F18	X_caenab
D20	X_careq
E19	X_casdata
T4	X_cp_clk
W1	X_cp_din1
V3	X_cp_din2
U3	X_cp_dout1
V2	X_cp_dout2
L1	X_cp_ena_b
G17	X_cvclk
K17	X_cvdata_0
J20	X_cvdata_1
J19	X_cvdata_2

Ball	Signal Name
C2	X_rom_d_0
D2	X_rom_d_1
D3	X_rom_d_2
E4	X_rom_d_3
C1	X_rom_d_4
D1	X_rom_d_5
E3	X_rom_d_6
E2	X_rom_d_7
E1	X_rom_lat_0
F3	X_rom_lat_1
G4	X_rom_lat_2
G2	X_rom_oe_b
G3	X_rom_we_b
C19	X_sbclk
A17	X_sd_a_0
A18	X_sd_a_1
A19	X_sd_a_2

Ball	Signal Name
Y19	X_sysa_7
V18	X_sysa_8
W19	X_sysa_9
Y20	X_sysa_10
W20	X_sysa_11
V19	X_sysa_12
U19	X_sysa_13
U18	X_sysa_14
T17	X_sysa_15
U20	X_sysa_16
T18	X_sysa_17
T19	X_sysa_18
T20	X_sysa_19
R18	X_sysa_20
P17	X_sysa_21
R19	X_sysa_22
R20	X_sysa_23

Ball	Signal Name
Y1	X_vdata_0
W3	X_vdata_1
W4	X_vdata_2
V4	X_vdata_3
U5	X_vdata_4
Y3	X_vdata_5
Y4	X_vdata_6
V5	X_vdata_7
G1	X_viclk
H3	X_vid_0
H2	X_vid_1
J4	X_vid_2
K2	X_vid_3
K3	X_vid_4
K1	X_vid_5
M1	X_vid_6
M2	X_vid_7

Signal Description

This section describes the functionality of each signal, and also the characteristics of the signal – output drive, input specification, etc.

System Bus

The system bus is used as one of the SDRAM memory busses in Aries products. It is also used to connect a debug card in development systems, and can be used as an expansion bus for additional IO and memory devices.

Signal Name	Type	Signal Description
`X_sysd_(15-0)`	Bidirectional 8mA output CMOS input	**System Bus Data Low Word:** This is the data bus over which the Aries Media Processors communicate with the System Bus memory and any external host controller that may be present.
`X_sysd_(31-16)`	Bidirectional 8mA output CMOS input BGA only	**System Bus Data High Word:** This is the data bus over which the Aries Media Processors communicate with the System Bus memory and any external host controller that may be present. When using SDRAM on the system bus interface, only data lines 15-0 are used to interface with the 16-bit SDRAM. Data pins 31-16 are left no connect on the BGA package (they should however be either pulled up or down).
`X_sysa(24-23)`	Bidirectional 8mA output CMOS input BGA only	**System Bus Address bits 24-23:** When Aries is controlling a memory device, it outputs address bits 24-23 on these lines. Bit 24 is used as bit 1 of the external host address for slave access when **xhostAddrLo** is clear. These pins may also be configured as GPIO pins.
`X_sysa_(22-18)`	Bidirectional 8mA output CMOS input	**System Bus Address bits 22-18:** These lines are driven by Aries when it 'owns' the bus. Bit 20 is used as bit 1 of the external host address for slave access when **xhostAddrLo** is clear. These pins may also be configured either as GPIO pins or as the SIO B channel.
`X_sysa_17`	Bidirectional 8mA output CMOS input	**System Bus Address bit 17/SDRAM Row Address Strobe:** Normally this pin functions as address bit 17. When using SDRAM on the system bus this pin functions as the SDRAM row address strobe. **Note:** When being used as RAS for SDRAM, this line should be pulled up using a 10K resistor
`X_sysa_(16-15)`	Bidirectional 8mA output CMOS input	**System Bus Address bits[16:15]/SDRAM Bank Address [1:0]:** Normally these pins function as address bits 16-15. When using SDRAM on the system bus these pins function as the SDRAM bank select bits. Bits 14-13 are used as bits 1-0 of the external host address for slave access when **xhostAddrLo** is set. **Note:** When using a 2-bank SDRAM on the system bus, BA[1] is a no connect
`X_sysa_(14-2)`	Bidirectional 8mA output CMOS input	**System Bus Address bits[14:2]:** Normally these pins function as address bits 14-2. When accessing SDRAM on the system bus these pins drive address lines 12-0 on the SDRAM chips. **Note:** The least significant address bits need to be used based on the type of SDRAM

Signal Name	Type	Signal Description
X_syswe	8mA output Active low	**System Bus Write Enable:** When Aries is accessing memory on the System Bus, this pin functions as System Bus Write Enable. It is used as a memory write enable for DRAM and other memory types. **Note:** It is a recommended that a 10K pull up be used on this line
X_sysgpcs1	8mA output Active low	**General Purpose Chip Select 1:** In this mode, this pin functions as General Purpose Chip Select 1. It can used to access SRAM, ROM, or FLASH memory. It is active low.
X_sysoe	8mA output Active low	**DRAM Output Enable Control/SDRAM Data Mask:** When accessing SDRAM on the system bus, this pin functions as the data mask enable/disable. For 16-bit devices this signal functions as both the UDQM as well as the LDQM because byte-wide operations are not supported on the System bus SDRAM interface. **Note:** A 10K pullup is required on this line.
X_sysbr	4mA output Active low BGA only	**Bus Request:** This signal is asserted by Aries, to indicate to the external host controller (or external bus arbiter) that it needs to gain control of the System bus.
X_sysbg	CMOS input Active low	**Bus Grant:** This signal is driven by the external host controller (or external bus arbiter), in response to the bus request assertion by Aries, indicating to Aries that it has been granted the System bus.
X_sysbb	Bidirectional 8mA output CMOS input Active low	**Bus Busy:** This line is asserted by Aries to indicate that it owns the bus. This line can be asserted only after a Bus Grant has been received, and there is no other device that is driving the Bus Busy signal. This output may be driven low by other bus masters, so is driven low on assertion, and is then briefly driven high before being set to tri-state. This signal needs to have an external pull-up.
X_sysdramcs	8mA output CMOS input	**SDRAM Chip Select 0:** When accessing SDRAM on the system bus, this line functions as the logical bank 0 select pin. **Note:** A 10K pullup is required on this line
X_syscas	Bidirectional 8mA output CMOS input Active low	**SDRAM Column Address Strobe:** When accessing SDRAM on the system bus, this line functions as the SDRAM column address strobe. **Note:** It is a recommended that a 10K pullup be used on this line
X_sysrw	Bidirectional 4mA output CMOS input	**System Bus Read/Write:** Indicates the type of the transfer for both master and slave cycles.
X_sysgpcs0	8mA output Active low	**General Purpose Chip Select 0/SDRAM Chip Select 1:** This pin functions as General Purpose Chip Select 0 to access SRAM, ROM, or FLASH memory. When accessing SDRAM on the System Bus, this pin can also used as the Logical bank 1 select if enabled. Software should be set appropriately to select the required mode. **Note:** A 10K pullup is required on this line
X_syscs	CMOS input Active low	**Chip Select:** When this signal is asserted, Aries will qualify X_sysa_(24-2) and X_sysrw to allow an external bus master to access its internal registers.
X_sysbclk	8mA output	**System bus Clock:** This clock is driven by Aries onto the System Bus and is derived from the internal 108Mhz master clock divided by two to give a 54 MHz output.

Signal Name	Type	Signal Description
`X_sys_rdy_b`	CMOS input Active low	**System Ready:** This indicates the completion of a cycle. When enabled to do so, a falling edge (assertion) on this signal will end a ROM/SRAM access. **Note:** It is a recommended that a 10K pull up be used on this line
`X_gpio_15`	Bidirectional 4mA output CMOS input	**Upper Address Enable/System Address Bit 31:** GPIO[15] can be optionally assigned to the System bus under software control. When assigned to the System bus, it functions as a time multiplexed Upper Address Enable and System Address bit 31. *Upper Address Enable:* This signal is driven low at the same time as SYSBB, to indicate that Aries 'owns' the bus. UAE may be used to externally force an address on the upper pins of the bus *System Address Bit 31:* Driven by Aries when it owns the bus
`X_gpio_(14-9)`	Bidirectional 4mA output CMOS input	**System Address Bits 30-25:** GPIO[14:9] can be assigned to the System bus under software control. When assigned to the System bus, they function as System Address bits 30-25. They are driven by Aries when it owns the bus
`X_gpio_0`	Bidirectional 4mA output CMOS input	**System Bus Interrupt Output:** GPIO0 can be assigned to the System Bus under software control. When assigned to the System bus it functions as the system bus interrupt output.

Table 3 - System Bus Interface

Main Bus

The Main Bus is the primary SDRAM memory bus in Aries based products. It is configurable to support one or two 16-bit SDRAM devices operating at 108 MHz.

Signal Name	Type	Signal Description
`X_sd_dq_(15-0)`	Bidirectional 8mA output CMOS input	**Main Memory Data Bus:** This is the conduit for transferring data between Aries and the SDRAM on the Main Memory bus.
`X_sd_a_(11-0)`	8mA output	**Main Memory Address Bus:** Aries sends out the multiplexed row/column address, to the main memory, on this bus.
`X_sd_a_12`	8mA output	**Main Memory Address Bus Bit 12/Data Byte Mask for Bank 2:** When using a 64Mbit density SDRAM, this pin should be used to convey address information to the SDRAM. (When using 2 banks of 16Mbit density SDRAMs, this pin should be used as the data mask for the lower DRAM byte for the secondary bank)
`X_sd_a_13`	8mA output	**Main Memory Address Bus Bit 13/Data Byte Mask for Bank 2:** When using a 64Mbit density SDRAM, this pin should be used to convey address information to the SDRAM. (When using 2 banks of 16Mbit density SDRAMs, this pin should be used as the data mask for the upper DRAM byte for the secondary bank.)
`X_sd_a_14`	8mA output	**Main Memory Address Bus Bit 14:** This additional address line is required for 128 Mbit SDRAM only. **Note:** This pin is not available in some Aries 3 packaging options.
`X_sd_cas_b`	8mA output Active low	**SDRAM Column Address Strobe**
`X_sd_ras_b`	8mA output Active low	**SDRAM Row Address Strobe**
`X_sd_cs_b_0`	8mA output Active low	**SDRAM Chip Select 0:** This chip select is used to select the primary SDRAM chip.

Signal Name	Type	Signal Description
X_sd_cs_b_1	8mA output Active low BGA only	**SDRAM Chip Select 1:** This chip select is used to select the secondary SDRAM chip. This function is only available in the BGA package.
X_sd_dqm_(1-0)	8mA output	**SDRAM Data Byte Masks 1-0:** Byte masks 1 and 0, enable selection/de-selection of the upper and lower byte, respectively, of the primary SDRAM bank.
X_sd_we_b	8mA output Active low	**SDRAM Write Enable**
X_sd_clk	8mA output	**SDRAM Clock Output:** Aries drives the 108Mhz SDRAM clock to the SDRAM banks.

Table 4 - SDRAM Interface

ROM Bus

The ROM bus is used to attach a ROM, EPROM or Flash memory to boot Aries. Refer to the ROM interface section below for more information on how to attach the boot ROM. The pin description here is for Mode 0 operation only.

Signal Name	Type	Signal Description
X_rom_d_(7-0)	Bidirectional 4mA output CMOS input BGA only	**ROM Multiplexed Address/Data Bus:** This is the multiplexed ROM Address and Data bus. Aries will send out the 24-bit ROM address on this byte-wide bus. Three external octal-latches have to be utilized to generate the full-blown 24-bit address.
X_rom_lat_2	Bidirectional 4mA output CMOS input BGA only	**ROM Address Latch Enable 2:** This is the latch enable for the octal flip-flop that generate bits 16-23 of the ROM address. The latch will capture the data on the rising edge of this signal. Bidirectional to support other modes, see ROM interface section.
X_rom_lat_1	4mA output BGA only	**ROM Address Latch Enable 1:** This is the latch enable for the octal flip-flop that generate bits 8-15 of the ROM address. The latch will capture the data on the rising edge of this signal.
X_rom_lat_0	4mA output BGA only	**ROM Address Latch Enable 0:** This is the latch enable for the octal flip-flop that generate bits 0-7 of the ROM address. The latch will capture the data on the rising edge of this signal.
X_rom_cs_b	4mA output Active low	**ROM Chip Select**
X_rom_we_b	4mA output Active low BGA only	**ROM Write Enable**
X_rom_oe_b	4mA output Active low BGA only	**ROM Output Enable**

Table 5 - ROM Interface

Video Interface

Aries supports CCIR656 compliant video output and input.

Signal Name	Type	Signal Description
X_vdata_(7-0)	Bidirectional 4mA output CMOS input	**Video data Output/Mode Select:** Video data conforming to the CCIR 656 standard is output on these pins. During Reset these pins are used to configure Aries. Please refer to the section below on Power Up Mode Selection for details.
X_vclk	12mA output	**Video Clock Output:** The CCIR 656 27 MHz Video Clock output, that is sent along with the Video Data output , is driven on this pin. This signal is derived internally
X_field	Bidirectional 4mA output CMOS input	**Field Synchronization:** Derived internally or externally. This signal is LOW during the first or even field, and HIGH during the second or odd field. As an input, an edge resets the vertical counter. This signal and X_hsync do not normally need to be connected to the video encoder as the timing information is embedded in the CCIR 656 stream (SAV and EAV fields).
X_hsync	Bidirectional 4mA output CMOS input Active low	**Horizontal Synchronization:** Derived internally or externally. As an input, the start of horizontal sync resets the horizontal counter
X_viclk	Schmitt input BGA only	**Video Input Clock:** Clock for the CCIR656 video input channel.
X_vid_(7-0)	CMOS input BGA only	**Video Input Data:** This input data is synchronous to the input clock

Table 6 - Video Interface

Audio Interface

Aries supports up to four stereo output pairs and two stereo input pairs, in addition to a IEC 958 (S/P DIF) output port.

The second stereo input channel is optionally available on three of the general purpose IO pins. This second channel can also be used to capture data in I^{2}S format from a drive.

Signal Name	Type	Signal Description
X_aclk	Bidirectional 8mA output CMOS input	**Audio Master Clock:** This pin carries the clock used to define the timing for the Aries audio output. It can be either an output from the internal Audio PLL, or an input from an external PLL. Aries normally operates with this clockat 256 times the sample rate, but can be programmed to work at other multiples.
X_sbclk	4mA output	**Audio Output Bit Clock:** The synchronous serial bit clock. It is derived from the external audio clock and is 64, 48, or 32 times the sample rate.
X_swclk	X_spdif	**Audio Output Word Clock:** This word clock gives the framing of the audio serial bit stream. The polarity of this signal, and its alignment to the data, are programmable
X_sdat_(2-0)	4mA output	**Audio Output Serial Data 0-2:** These are the serial PCM audio data outputs. Each pin supports one stereo pair, for a total of up to six channels of audio data.
X_gpio_1	Bidirectional 4mA output CMOS input	**Audio Output Serial Data 3:** This pin may optionally be configured to carry a serial PCM audio data output. This pin supports one additional stereo pair, for a grand total of up to eight channels of audio data.
X_spdif	4mA output	**IEC 958 Output:** Self Clocking IEC 958 Data.

Signal Name	Type	Signal Description
X_ai_bclk	Bidirectional 4mA outputCMOS input	**Audio Input Channel 1 Bit Clock:** Provides the synchronous serial bit clock for the serial input data. This does not have to operate at the same sample rate as the serial bit clock output. It can be either internally generated and output here, or externally generated and therefore input here.
X_ai_wclk	Bidirectional 4mA output CMOS input	**Audio Input Channel 1 Word Clock:** Provides the framing information for the serial data input stream. The relationship of this to the input data is programmable. It can be either internally generated and output here, or externally generated and therefore input here.
X_ai_data	CMOS input	**Audio Input Channel 1 Serial Data:** This is a serial stereo audio input channel that conforms to the serial (I^2S) interface
X_gpio_5	Bidirectional 4mA output CMOS input	**Audio Input Channel 2 Bit Clock:** Provides the synchronous serial bit clock for the serial input data. This does not have to operate at the same sample rate as either the serial bit clock output or input channel 1. It can be either internally generated and output here, or externally generated and therefore input here.
X_gpio_6	Bidirectional 4mA output CMOS input	**Audio Input Channel 2 Word Clock:** Provides the framing information for the serial data input stream. The relationship of this to the input data is programmable. It can be either internally generated and output here, or externally generated and therefore input here.
X_gpio_4	Bidirectional 4mA output CMOS input	**Audio Input Channel 2 Serial Data:** This is a serial stereo audio input channel that conforms to the serial (I^2S) interface

Table 7 - Audio Interface

Coded Data Interface

The Coded Data Interface is used to connect to the read data channel of DVD Drives or other media devices. This interface is highly configurable by software.

Signal Name	Type	Signal Description
X_cvdata_(7-0)	CMOS input	**Coded Video Data Bus:** Can be used for parallel or serial Program Elementary Stream Video/Audio, Program Stream or Transport Stream Data.
X_cvenab	CMOS input	**Coded Video Enable:** Can be used for Data Enable (CVENAB) or Data Strobe (CVSTROBE) for the Coded Video/Program Stream/Transport Stream. The signal polarity is programmable.
X_cvclk	Schmitt input	**Coded Video Clock:** Can be used for Coded Video/Program Stream/Transport Stream Data Clock. The signal polarity (active edge) is programmable.
X_cvreq	4mA output	**Coded Video Request:** Can be used for Coded Video/Program Stream Request when this interface is hand-shaked. The signal polarity is programmable.
X_caenab	CMOS input	**Coded Audio Enable:** Can be used for Coded Audio Data Enable (CAENAB) or Coded Audio Data Strobe (CASTROBE). The signal polarity is programmable.
X_casdata	CMOS input	**Coded Audio Serial Data:** This pin can be used either for Coded Serial Audio Data (CASDATA), or Transport Stream/Program Stream Error Flag (CVERRFLG)
X_caclk	Schmitt input	**Coded Audio Clock:** This is the Coded Audio Data Clock. The signal polarity (active edge) is programmable.

Signal Name	Type	Signal Description
`X_careq`	Bidirectional 4mA output CMOS input	**Coded Audio Request:** This pin can be used as either the Coded Audio Request (CAREQ), or Transport Stream/Program Stream Top of Packet Signal (CVTOP). The signal polarity is programmable.

Table 8 - Coded Data Interface

NUON Serial Bus Interface

The NUON serial bus interface is used to connect to the two controller ports on NUON compatible products.

Signal Name	Type	Signal Description
`X_cp_clk`	4mA output	**Controller Clock Output:** Normally operates at 1 MHz,.
`X_cp_dout(2-1)`	4mA output	**Controller Data Outputs 2-1**
`X_cp_din(2-1)`	CMOS input	**Controller Data Inputs 2-1**
`X_cp_ena_b`	4mA output Active low	**Controller Data Tri-state Enable:** Used to control an external tri-state buffer.

Table 9 - Controllers Interface

Clocks and Reset

Signal Name	Type	Signal Description
`X_reseti_b`	Schmitt input	**System Reset:** This is the system power-on reset input. An assertion on this pin will bring Aries back to its initial state.
`X_pll_clki`	CMOS input	**System Clock Input:** This is the main clock input to Aries. This clock input is either a frequency to reference the mail PLL (27 MHz is normally used), or 108Mhz to be used directly as the system clock.
`X_pll_ref`	4mA output BGA only	**PLL Reference Clock Output:** This pin outputs a 27 MHz reference clock to an external PLL. This is normally only required when Aries is being used with an external Transport Stream decoder.

Table 10 - Clocks and Reset

General IO Interface, Multi-Purpose Pins

Aries 3 can have as many as 36 pins allocated as general purpose inputs and outputs, under software control. Many of these can have alternate dedicated functions, and not all are available when using the QFP package.

Signal Name	Type	Signal Description
`X_sysa_24`	Bidirectional 8mA output CMOS input BGA only	Configurable as: • System Bus address bit 24, and bit 1 of the external host address for slave access when **xhostAddrLo** is clear. • GPIO[35]
`X_sysa_23`	Bidirectional 8mA output CMOS input BGA only	Configurable as: • System Bus address bit 23 • GPIO[34]

Signal Name	Type	Signal Description
X_sysa_22	Bidirectional 8mA output CMOS input	Configurable as: • System Bus address bit 22 • GPIO[33] • SIO B receive data
X_sysa_21	Bidirectional 8mA output CMOS input	Configurable as: • System Bus address bit 21 • GPIO[32] • SIO B request/acknowledge
X_sysa_20	Bidirectional 8mA output CMOS input	Configurable as: • System Bus address bit 20, and bit 0 of the external host address for slave access when **xhostAddrLo** is clear. • GPIO[31] • SIO B transmit data
X_sysa_19	Bidirectional 8mA output CMOS input	Configurable as: • System Bus address bit 19 • GPIO[30] • SIO B request
X_sysa_18	Bidirectional 8mA output CMOS input	Configurable as: • System Bus address bit 18 • GPIO[29] • SIO B clock
X_gpio_(25-16)	Bidirectional 4mA output CMOS input BGA only	General Purpose IO Pins for the BGA package.
X_gpio_15	Bidirectional 4mA output CMOS input	Configurable as: • GPIO(15) • System Bus address bit 31 • System Bus upper address enable
X_gpio_14	Bidirectional 4mA output CMOS input	Configurable as: • GPIO(14) • System Bus address bit 30 • PWM output 1
X_gpio_13	Bidirectional 4mA output CMOS input	Configurable as: • GPIO(13) • System Bus address bit 29 • PWM output 0
X_gpio_12	Bidirectional 4mA output CMOS input	Configurable as: • GPIO(12) • System Bus address bit 28 • System Bus NAND flash busy.
X_gpio_11	Bidirectional 4mA output CMOS input	Configurable as: • GPIO(11) • System Bus address bit 27 • Secondary Serial Peripheral Bus interface SDA • SIO A receive data

Signal Name	Type	Signal Description
`X_gpio_10`	Bidirectional 4mA output CMOS input	Configurable as: • GPIO(10) • System Bus address bit 26 • Secondary Serial Peripheral Bus interface SCL • SIO A request/acknowledge
`X_gpio_9`	Bidirectional 4mA output CMOS input	Configurable as: • GPIO(9) • System Bus address bit 25 • SIO A transmit data
`X_gpio_8`	Bidirectional 4mA output CMOS input	Configurable as: • GPIO(8) • SIO A request
`X_gpio_7`	Bidirectional 4mA output CMOS input	Configurable as: • GPIO(7) • SIO A clock
`X_gpio_6`	Bidirectional 4mA output CMOS input	Configurable as: • GPIO(6) • Second audio input channel word flag
`X_gpio_5`	Bidirectional 4mA output CMOS input	Configurable as: • GPIO(5) • Second audio input channel bit clock
`X_gpio_4`	Bidirectional 4mA output CMOS input	Configurable as: • GPIO(4) • Second audio input channel data
`X_gpio_3`	Bidirectional 4mA output CMOS input	Configurable as: • GPIO(3) • Serial Peripheral Bus interface SDA
`X_gpio_2`	Bidirectional 4mA output CMOS input	Configurable as: • GPIO(2) • Serial Peripheral Bus interface SCL
`X_gpio_1`	Bidirectional 4mA output CMOS input	Configurable as: • GPIO(1) • Audio input high rate clock output • Audio output I2S data line sdat[3]
`X_gpio_0`	Bidirectional 4mA output CMOS input	Configurable as: • GPIO(0) • System Bus interrupt output

Table 13 - General IO Interface

System Bus Overview

This interface has two modes of operation, external and internal. Internal mode allows Aries to directly interface to devices on the bus by generating the necessary strobes for each device. It can handle multiple bus master. Internal mode is used in all current applications.

In the external mode it operates in conformance with the PowerPC MPC860 type bus interface which uses the 860 bus arbiter to accommodate multiple bus masters. **Note:** External mode is considered obsolete and is no longer supported. It is not documented here.

Internal Mode

In this mode of operation, Aries is responsible for reaching the various devices on the bus. While there is still provision for multiple masters on the bus, Aries is the memory controller and will generate the needed strobes. In this mode, Aries also takes responsibility for DRAM refresh. External masters cannot access memory using the Aries internal memory controller.

Internal mode SDRAM Control

This table summarizes the System Bus SDRAM requirements:

Frequency	54 MHz
Memory width	16-bit
Memory Size	2 MB min, 256 MB max
Banks	1 or 2 logical banks (chip selects), each 16-bit wide
Interface	LVTTL, 3.3V operation

SDRAM types supported

Technology	Internal banks	I/O	# Devices/ Chip Sel	Row/Col Addr	Bank Addr	Memory Size / Chip Select
16 Mbit	2	16	1	X_sysa[12:2]	X_sysa[15]	2 MB
	2	8	2	X_sysa[12:2]	X_sysa[15]	4 MB
	2	4	4	X_sysa[12:2]	X_sysa[15]	8 MB
64 Mbit	2	16	1	X_sysa[14:2]	X_sysa[15]	8 MB
	2	8	2	X_sysa[14:2]	X_sysa[15]	16 MB
	2	4	4	X_sysa[14:2]	X_sysa[15]	32 MB
	4	16	1	X_sysa[13:2]	X_sysa[16:15]	8 MB
	4	8	2	X_sysa[13:2]	X_sysa[16:15]	16 MB
	4	4	4	X_sysa[13:2]	X_sysa[16:15]	32 MB
128 Mbit	4	16	1	X_sysa[13:2]	X_sysa[16:15]	16 MB
	4	8	2	X_sysa[13:2]	X_sysa[16:15]	32 MB
	4	4	4	X_sysa[13:2]	X_sysa[16:15]	64 MB
256 Mbit	4	16	1	X_sysa[14:2]	X_sysa[16:15]	32 MB
	4	8	2	X_sysa[14:2]	X_sysa[16:15]	64 MB
	4	4	4	X_sysa[14:2]	X_sysa[16:15]	128 MB

Programmable options

- Either EDO or SDRAM can be enabled. However, both types cannot co-exist. Use of EDO DRAM is now considered obsolete and not supported.

- Each logical bank can have any of the supported SDRAM types (depends on being able to meet timing). This means that the minimum memory size available is 2 MB (2MB + 0 MB), and the maximum memory size is 256 MB (128 MB + 128 MB).
- Each logical bank can be individually enabled or disabled.
- An option to make both the logical banks appear as either contiguous or non-contiguous memory is provided.
- CAS Latencies of 1/2/3 are supported.
- Burst lengths of 2/4/8 are supported.
- Programmable tWR (a.k.a tDPL), tRAS, tRP, and tRC timings are supported. tRCD is always 2 clocks.
- Programmable refresh rates are supported.
- An option is provided to tri-state the sdram address, control, and data signals during an external master access to systembus sdram.

Pinout

Pin Name	**SDRAM signal**	**Remarks**
`X_sysdramcs`	CS0#	10 K pull-up required. Logical bank0 chip select.
`X_sysgpcs0`	CS1#	10 K pull-up required. Logical bank1 chip select if enabled, else behaves as a chip select signal to access SRAMS/ROMS, etc.
`X_sysoe`	DQM	10 K pull-up required. For 16-bit devices, this signal functions as UDQM as well as LDQM because byte-wide operations are not supported.
`X_sysa_17`	RAS#	10K pull-up recommended.
`X_syscas`	CAS#	10K pull-up recommended.
`X_sysa_1`	WE#	10K pull-up recommended.
`X_sysa_(14-2)`	A[12:0]	Use the least significant address bits, depending on the type of SDRAM.
`X_sysa_(16-15)`	BA[1:0]	X_sysa[16] is a no-connect for 2 Bank parts.
`X_sysd[15:0]`	DATA[15:0]	
`X_sysbclk`	CLK	
	CKE	10 K pull-up required.

SDRAM initialization

The following flowchart may be used for sdram initialization, assuming that sysCtrl, SysMemctl, and sdramCtrl all have default values.

1. Disable refresh by setting sdramCtrl[30] = 0.
2. Set internal mode of operation: sysCtrl[18] = 0.
3. Program refresh rate in sysMemctl[14:4].
4. Set up SDRAM logical bank0 and bank1 options in sdramCtrl:
   dram0Enable, dram0Banks, dram0Width, dram0Tech,
   dram1Enable, dram1Banks, dram1Width, dram1Tech,
   contiguousSdram,
   Set **trc** to a minimum value of 2
   Set **tras** to a minimum value of 2

If Micron SDRAM, set **twr** value = 1, else set **twr** value = 0
trp value may be set to 0 for most SDRAMs.
Set valid values for **casLatency**, **burstLength**
Clear **refreshCmd**, **prechCmd**, and **mrsCmd** bits.

5. Set **prechReq**. Wait until it is cleared by the SDRAM controller.
6. Wait for a few clocks (about 4-5 clocks)
7. Set **mrsReq**. Wait until it is cleared by the SDRAM controller.
8. Wait for a few clocks (about 4-5 clocks)
9. Set **refreshReq**. Wait until it is cleared by the SDRAM controller.
10. Wait for a few clocks (about 4-5 clocks)
11. Set **refreshReq**. Wait until it is cleared by the SDRAM controller.
12. Wait for a few clocks (about 4-5 clocks)
13. Enable refresh by setting sdramCtrl[30] = 1.

Address Space

Dram0 _enable	**Dram1 _enable**	**Contiguous _sdram**	**DRAM0 Address Space**	**DRAM1 Address Space**
0	0	X	Disabled	Disabled
1	0	X	8XXXXXXXh	Disabled
0	1	0	Disabled	9XXXXXXXh
0	1	1	Disabled	8XXXXXXXh
1	1	0	8XXXXXXXh	9XXXXXXXh
1	1	1	8XXXXXXXh	contiguous to DRAM0

Operation

The Systembus controller can operate in external or internal mode. The following description applies to the SDRAM controller (SDRAMC) within the Systembus controller which is active only in internal mode.

In internal mode, the Systembus controller allows the MPEs to access Systembus SDRAM as well as other devices like SRAM, ROM, and other chip-select controlled peripheral devices via the internal Other Bus DMA (ODMA) interface. The Aries 3.0 is the default owner of the Systembus. External masters on the Systembus can access Aries 3.0 internal registers by de-asserting the grant signal and asserting the chip-select signal, or they can access memory devices on the bus, by not asserting chip-select to the Aries 3.0.

The SDRAMC can be programmed to support two logical banks DRAM0 and DRAM1. Each logical bank is 16-bits wide, and is controlled with a separate chip select signal.

Summary of Internal Mode Cycles

The various types of cycles in internal mode are described below:

1. MPEs access the systembus SDRAM memory through the Other Bus interface.
2. SDRAM Controller issues software initiated mode register set, pre-charge, and refresh commands to the SDRAMs.
3. SDRAM Controller issues refresh cycles to the SDRAMs.

4. External Systembus masters access Aries2.0 registers by de-asserting the grant and by asserting chip-select, or access the systembus SDRAM memory by de-asserting the chip-select and by taking away the grant.

MPE data accesses via the ODMA interface

MPEs access the SDRAM via the ODMA interface. The MPEs communicate the start address, the length of the DMA transfer in long-words, and the type of cycle (read/write) to the ODMA controller. The ODMA controller sends the start address and the cycle type information to the SDRAMC. The ODMA controller then sends as many data requests as specified by the MPE. Since the SDRAMC does not know the length of the transfer, the ODMA controller sends a special command indicating the last transfer. One long-word is transferred between the ODMA controller and the SDRAMC for each data request from the ODMA controller.

In response to an ODMA address pointer transfer command, the SDRAMC will first get ownership of the bus, and will activate the same row in all the internal banks of the logical bank being accessed. Since the width of the SDRAM interface is 16-bits, each data request from the ODMA controller is serviced as a read or write transfer on the SDRAM bus as a burst cycle of length 2, regardless of the burst-length programmed into the SDRAM. After the last transfer in a DMA cycle, all the rows in both logical banks are closed. The order in which the internal banks are activated depends on the bank in which the start address falls.

Start Address falls in Bank:	Activation order (4 Banks)	Activation order (2 Banks)
0	0 → 1 → 2 → 3	0 → 1
1	1 → 2 → 3 → 0	1 → 0
2	2 → 3 → 0 → 1	0 → 1
3	3 → 0 → 1 → 2	1 → 0

During SDRAM read cycles, the SDRAMC will fetch data from incremental address locations and store them in a 16 bit x 4 deep FIFO. ODMA read requests are serviced from the FIFO. If the ODMA controller holds off the other bus, the FIFO will become full, and, further read commands are not issued until the ODMA controller reads from the FIFO again. If the SDRAM is programmed with a burst length greater than 2, and if the FIFO becomes full in the middle of a burst, the extra data from the SDRAM is ignored. A new read command from the correct address is issued once there is space in the FIFO. When the SDRAMC receives the last request command from the ODMA controller, it will stop issuing read commands on the SDRAM bus. It will then wait until the FIFO becomes empty and will issue a pre-charge-all banks command to the SDRAM. The FIFO pointers are cleared at this time because the FIFO could have data from anticipatory reads that were actually not required.

During write cycles, data from the ODMA controller is written to the FIFO. The SDRAMC will read from the FIFO and will write the data to the SDRAM. If the ODMA controller holds off the other bus, the FIFO will become empty, and the SDRAMC will not issue write commands to the SDRAM until the ODMA controller writes more data into the FIFO. If the SDRAM is programmed with a burst length greater than 2, and if the FIFO becomes empty in the middle of a burst write, the DQM signal is de-asserted and the SDRAM will ignore the data on the bus. When the SDRAMC receives the last request command from the ODMA controller, it will write out all the data from the FIFO, and will issue a pre-charge-all banks command to the SDRAM.

If a page break is detected, the SDRAMC will issue a pre-charge-all banks command to the SDRAM which will cause all open rows to be closed. The new row will then be activated in all the internal banks of the logical bank being accessed, and the SDRAMC will continue with read or write commands. When the new row is activated in all the banks because of a page miss, the activation order is always 0 → 1 → 2 → 3.

If the two logical banks are programmed as non-contiguous, accesses within each logical bank will wrap to address 0 once the maximum address is reached. If the two logical banks are programmed as contiguous, if the maximum address of DRAM0 is reached, it is treated as a page-break, and new accesses will begin from Address 0 of DRAM1. Once the maximum address of DRAM1 is reached, accesses will wrap to Address 0 of DRAM0.

In the following timing diagrams, the signal “other” is an internal signal that indicates that a DMA cycle is in progress on the Systembus. The signal “ref_req” is an internal signal that indicates that refresh is pending. The signal “refresh” indicates that a refresh operation is in progress. The signal “sa_enable_b” is the output enable control for the dram memory address bus, the bank address bits, dram_ras_b, and dram_we_b

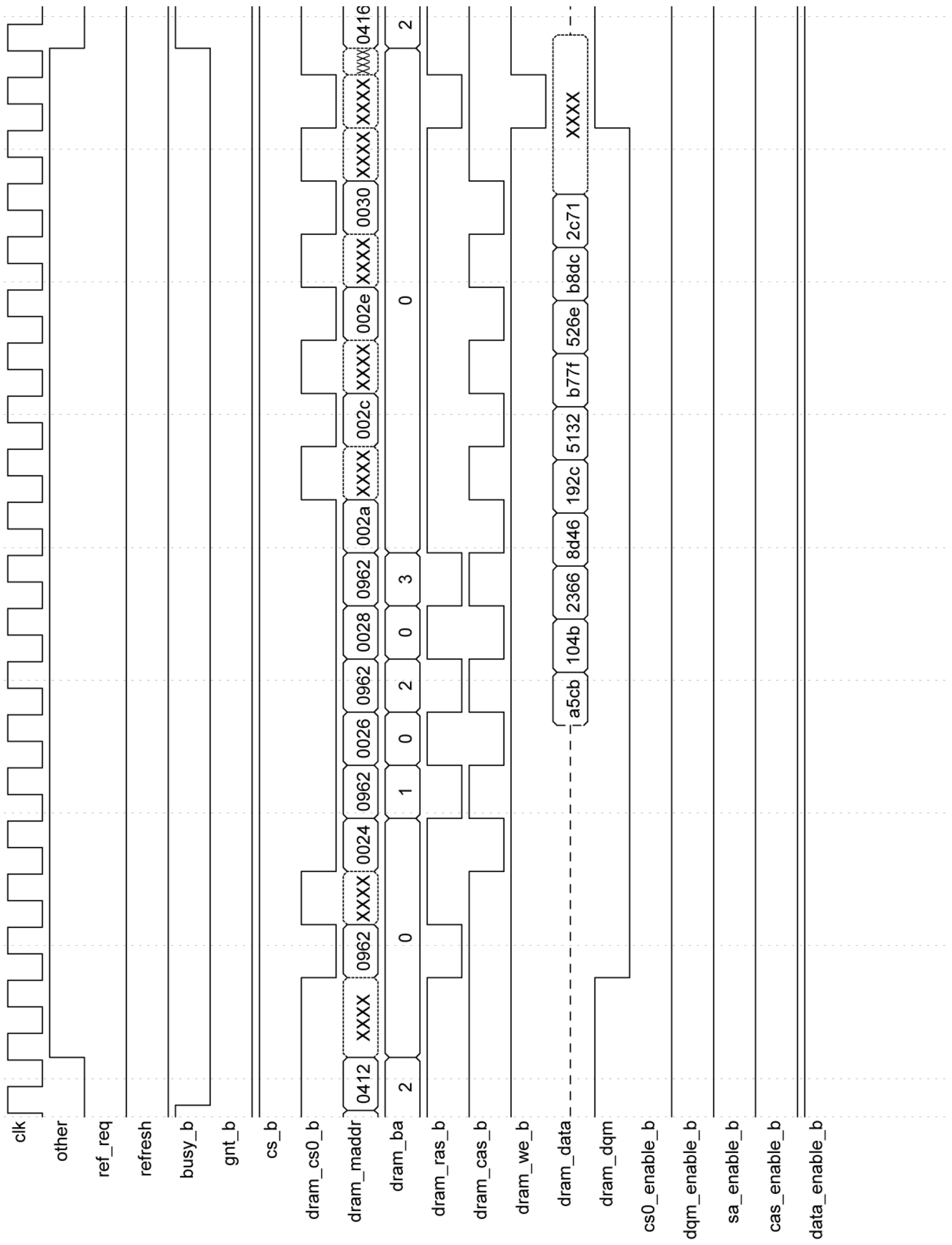


Figure 46: Basic SDRAM read cycle (64 Mbit, 4B x 16, CASLAT = 3)

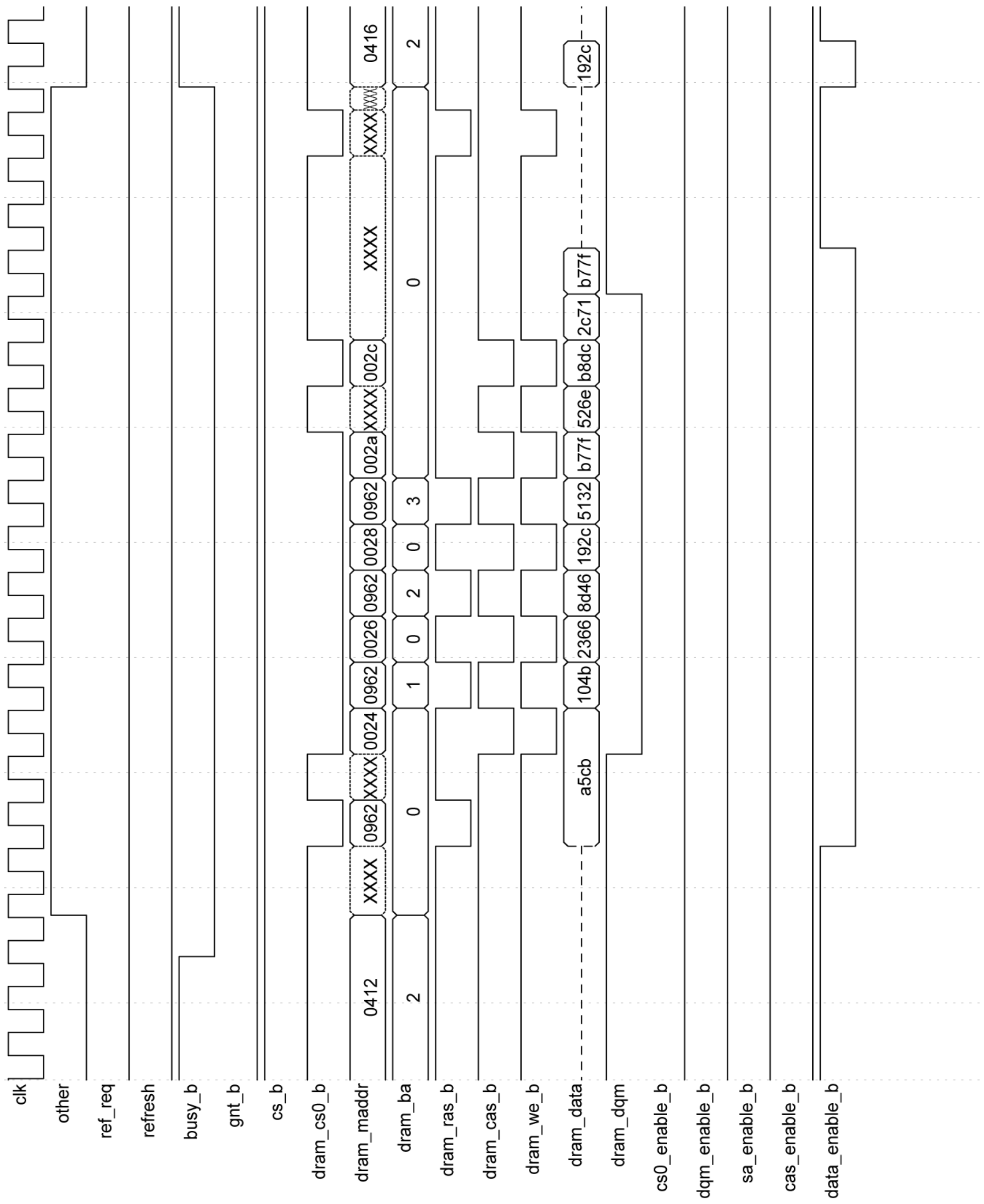


Figure 47: Basic SDRAM write cycle (64 Mbit, 4B x 16)

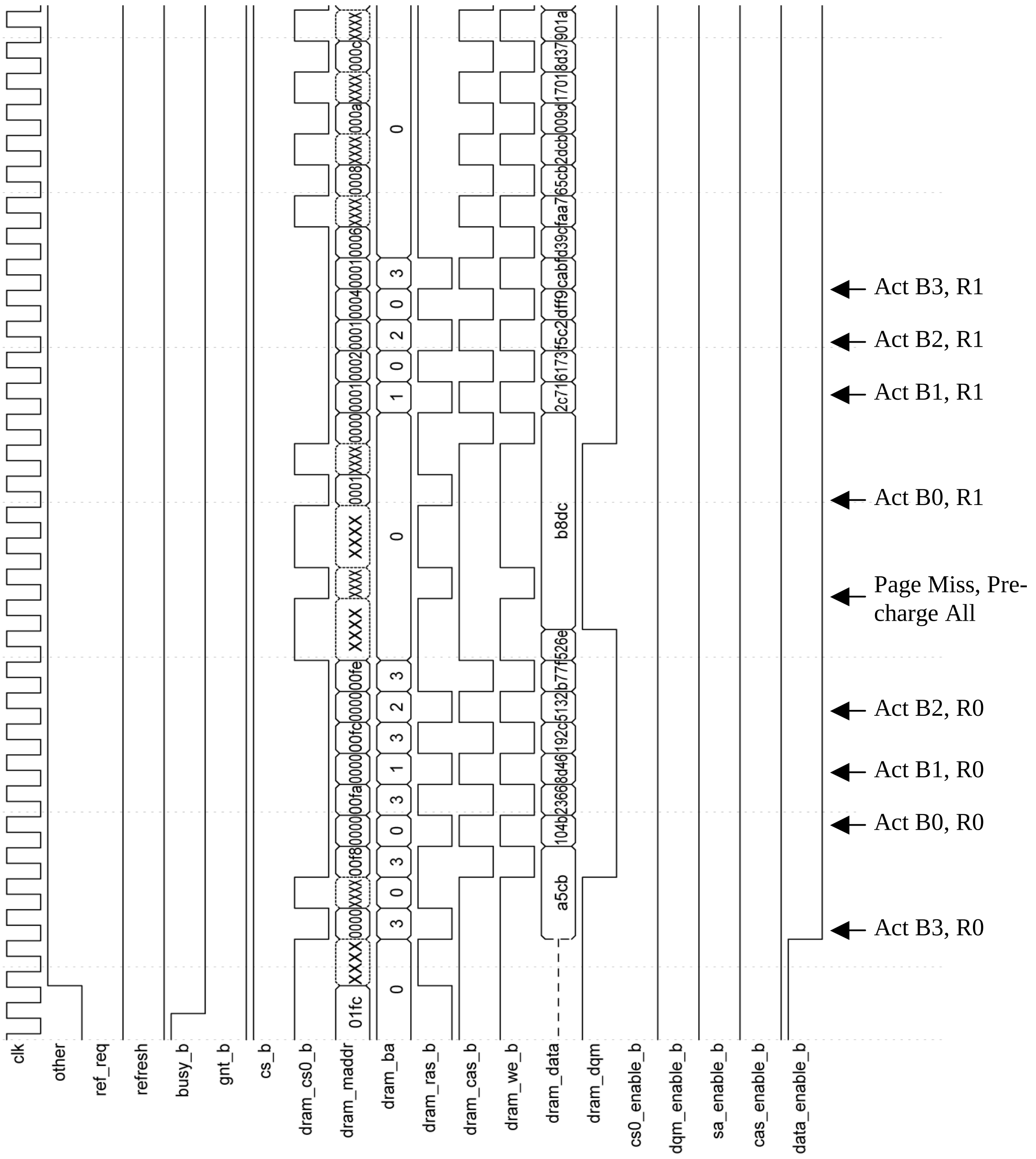
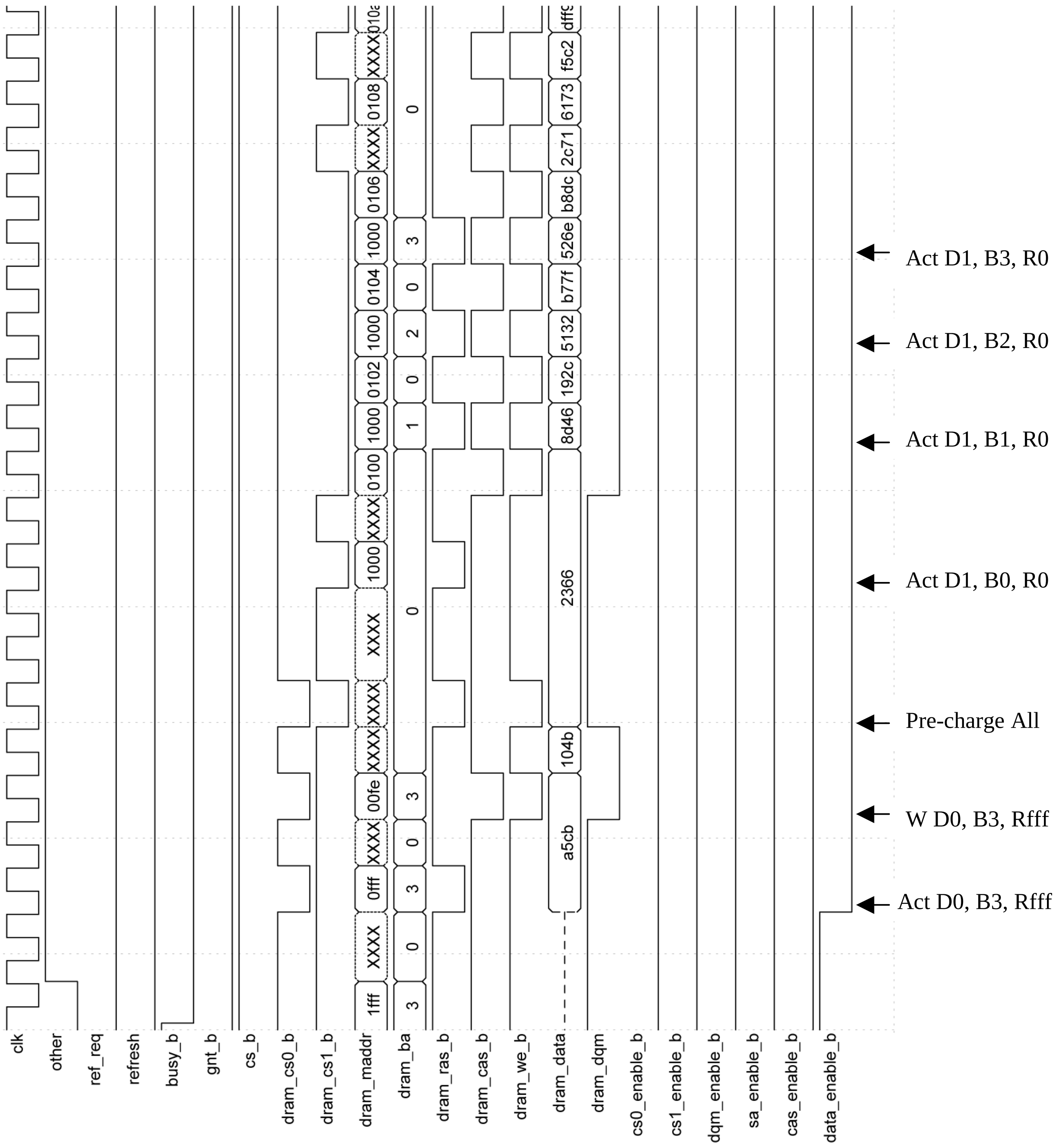


Figure 48: Page Break (64Mbit, 4B x 16). Access switches from Bank3, Row0 to Bank0, Row1.



Act D1, B3, R0

Act D1, B2, R0

Act D1, B1, R0

Act D1, B0, R0

Pre-charge All

W D0, B3, Rfff

Act D0, B3, Rfff

Figure 49: Seamless switch from DRAM0 to DRAM1 (contiguous, Dx = DRAM0 or DRAM1)

Software initiated cycles

Initializing SDRAM requires a pre-charge-all command, a mode register set command, and two refresh commands, after which the SDRAM is ready. These cycles are issued when the initialization code sets the **mrs_cmd**, **prech_cmd**, or the **refresh_cmd** bits in SDRAM_CTRL. In response, Aries arbitrates for the bus, and will issue the command to the SDRAMs. The bits are cleared by the SDRAMC after execution. Note that these commands are issued to both the logical banks at once (CS0# and CS1#), so all DRAMs are initialized at once. Software must insert delays between writing these register bits.

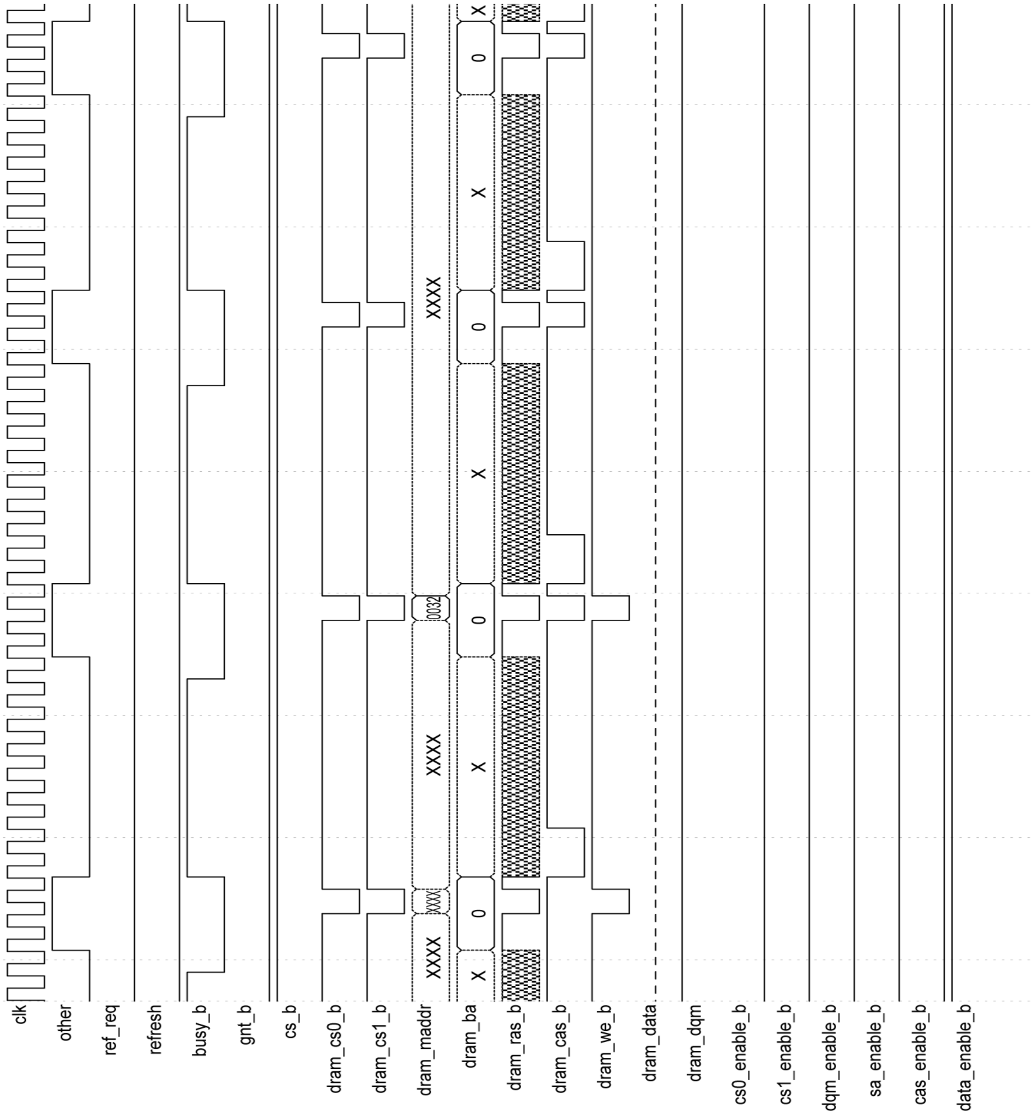
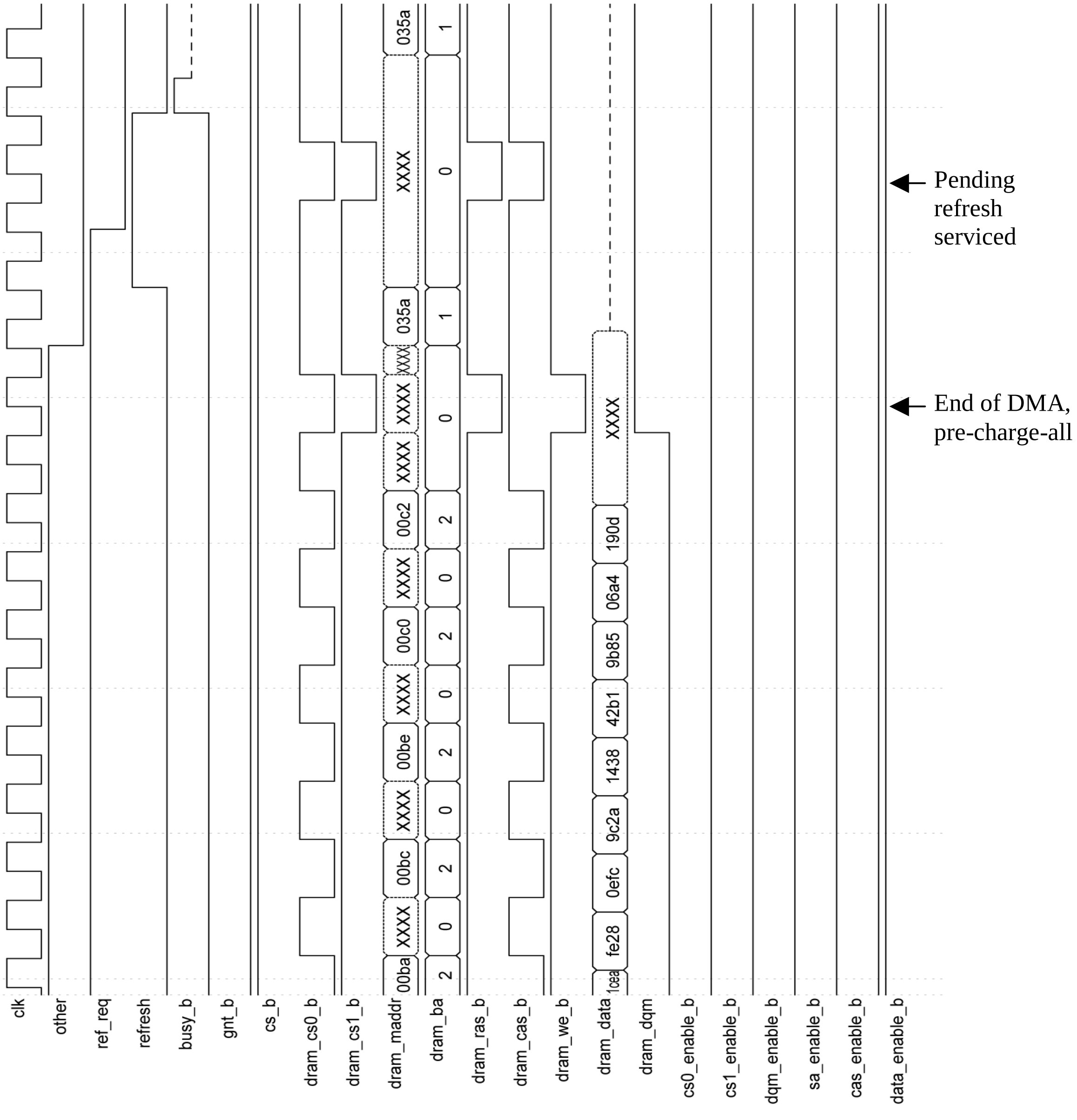


Figure 50: SDRAM initialization

SDRAM Refresh

The refresh rate is controlled through the refLength bits in the sysMemctl register. When the refresh request goes active, the SDRAMC will issue an auto-refresh command to the logical banks that are enabled, if Aries owns the Systembus. If the bus is not free, the refresh will be internally pending until the bus becomes free. If a refresh request goes active in the middle of a DMA cycle, the refresh cycle is performed after the DMA cycle completes and all the rows are closed.



Pending refresh serviced

End of DMA, pre-charge-all

Figure 51: Refresh cycle after SDRAM read

Address Mapping

The following table shows the memory map from other-bus address to SDRAM addresses.

16 Mbit	512Kx16 x2		1Mx8x2		2Mx4x2																					
64 Mbit							2Mx16x2		4Mx8x2		8Mx4x2		1Mx16x4		2Mx8x4		4Mx4x4									
128 Mbit															2Mx16x4		4Mx8x4		8Mx4x4							
256 Mbit																					4Mx16x4		8Mx8x4		16Mx4x4	
	11 x 8		11 x 9		11 x 10		13 x 8		13 x 9		13 x 10		12 x 8		12 x 9		12 x 10		12 x 11		13 x 9		13 x 10		13 x 11	
	Row	Col	Row	Col	Row	Col	Row	Col	Row	Col	Row	Col	Row	Col	Row	Col	Row	Col	Row	Col	Row	Col	Row	Col	Row	Col
MA 0	10	GND	11	GND	11	GND	10	GND	11	GND	11	GND	11	GND	11	GND	11	GND	11	GND	11	GND	11	GND	11	GND
MA 1	11	2	12	2	12	2	11	2	12	2	12	2	12	2	12	2	12	2	12	2	12	2	12	2	12	2
MA 2	12	3	13	3	13	3	12	3	13	3	13	3	13	3	13	3	13	3	13	3	13	3	13	3	13	3
MA 3	13	4	14	4	14	4	13	4	14	4	14	4	14	4	14	4	14	4	14	4	14	4	14	4	14	4
MA 4	14	5	15	5	15	5	14	5	15	5	15	5	15	5	15	5	15	5	15	5	15	5	15	5	15	5
MA 5	15	6	16	6	16	6	15	6	16	6	16	6	16	6	16	6	16	6	16	6	16	6	16	6	16	6
MA 6	16	7	17	7	17	7	16	7	17	7	17	7	17	7	17	7	17	7	17	7	17	7	17	7	17	7
MA 7	17	8	18	8	18	8	17	8	18	8	18	8	18	8	18	8	18	8	18	8	18	8	18	8	18	8
MA 8	18	10	19	10	19	10	18	10	19	10	19	10	19	23	19	23	19	23	19	23	19	24	19	24	19	24
MA 9	19	22	20	22	20	22	19	24	20	24	20	24	20	24	20	24	20	24	20	24	20	25	20	25	20	25
MA 10	20	GND	21	GND	21	GND	20	GND	21	GND	21	GND	21	GND	21	GND	21	GND	21	GND	21	GND	21	GND	21	GND
MA 11	21	23	22	23	22	23	21	25	22	25	22	25	22	25	22	25	22	25	22	25	22	26	22	26	22	26
MA 12	22	24	23	24	23	24	22	26	23	26	23	26	23	26	23	26	23	26	23	26	23	27	23	27	23	27
BA1	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
BA0	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9

Table 19: Other bus to SDRAM memory map (Array Size x Width x #Banks)

Note: MA0 is forced to 0 during read/write commands because accesses to the SDRAM are always on long-word boundaries. MA10 is forced to 0 during read/write commands because the auto-pre-charge feature of the SDRAM is not used. Shaded regions indicate address bits that are not seen by the SDRAMs.

Internal Mode Master Operation

When Aries takes control of the bus it can access any of the devices on the bus by means of the various strobes implemented in this mode. The main thing to access on the bus will be the DRAM. This will be done by the DRAM control signals, while other devices such as SRAM, ROM or micro-controllers can be addressed using the general-purpose chip selects. Memory accesses can be done anytime the grant signal is active. Once the bus has been obtained it can be locked by keeping the busy line active. This puts the other bus master on hold until Aries decides to deactivate the busy line.

Arbitration

There can be two types of cycles done on the System Bus, internal Other Bus DMA and refresh cycles. A third type may or may not be needed depending if there is an external processor that wants to communicate with Aries. For those that come from within Aries, each has its own request line which indicates that cycle time is needed on the bus. If both types of cycles are being requested then the refresh request has priority since it can take less time to complete and occurs less often than OB cycles. The extra variable in the arbitration comes about when an external CPU may be competing for the bus. Therefore, the process of obtaining bus has two parts for refresh and OB cycles originating from within Aries. The first involves competing between refresh and OB, and the second has the winner competing with the external host.

The figure below shows the waveforms of refresh and internal Other Bus DMA wanting to get on the bus at the same time. The bus is idle to begin with so the only competition is between the two requests. As soon as one starts, the bus-busy signal is asserted to notify a would-be external host that the bus is being used. After the refresh cycle has finished the OB cycle will start immediately since the request has already been set for some time. Once done, and since no more requests are present for the bus, the bus will go idle and the bus-busy signal is de-asserted. At the time the second cycle started, an external host could have very well removed the bus-grant signal in order to get the bus once it has become idle.

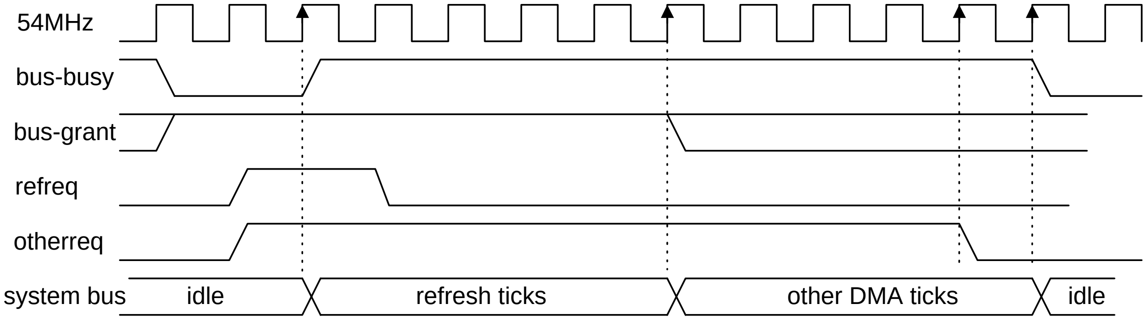


Figure 55: Other DMA and refresh arbitration

So instead of going idle the bus could have been taken over by the external host, but not until Aries was completely done with its current cycles. Another variation to this timing waveform picture would occur if the requests came in while the external processor controlled the bus. In that case, the two requests will both wait until the grant line tells Aries that the bus is available for use.

Aries can lock the bus in the same manner it did in the other mode. This is done by setting the bus-busy line active for as long as it wants. Of course, first it must obtain the bus which sets the busy line active, after the initial cycle the busy line will remain active so that subsequent Aries requests can take place immediately. The bus-busy line tells the external host that the bus is occupied.

Master accesses

External masters can access Aries registers by asserting Aries chip-select, driving a valid address on the System Bus address, driving a valid value on the RW signal, and either reading or writing the data bus.

If the **tristate_sdram_control** bit is set, external masters can also access the SDRAM directly by making Aries tri-state all the SDRAM interface signals. In this case, the external master is responsible for generating all the SDRAM address, control, and data signals. When the external master gains control of the bus, it is guaranteed that all the SDRAM rows will be in a pre-charged state. Once the external master asserts **X_sysbb**, the earliest clock edge on which it can issue commands to the SDRAM is determined by the SDRAM’s tRC1 (refresh to activate) timing parameter, and also the time it takes for Aries to tri-state all the SDRAM signals. It is recommended that the external master have a programmable wait state (minimum 1, maximum 4) between asserting **X_sysbb** and starting SDRAM accesses. When the external master finishes its access, it must pre-charge all open rows in all the internal banks of all logical banks before handing over control of the bus back to Aries. Once the master de-asserts **X_sysbb** signaling the completion of its cycle, if Aries has an internal cycle pending, it will assert the enable signals for the CAS# and DQM signals along with **X_sysbb**, and the enables for the CS#, WE#, RAS#, SDRAM address, bank address, one clock later. The external master is not allowed to issue a Mode Register Set command to the SDRAM.

Note that in internal mode, Aries is the default owner of the bus. External masters can issue cycles by negating the **X_sysbg** to Aries. If the external masters are expected to access SDRAM, for the SDRAM signals to be tri-stated, the grant must remain de-asserted for the duration of the external master cycle. Aries will not issue SDRAM refresh cycles when the external master owns the bus. If the master is expected to hold the bus for more than one refresh period, it is responsible for generating auto-refresh commands to the SDRAM.

For master cycles, the value of the register bit **tristate_sdram_bus** determines the signals that will be tri-stated by Aries. The following table describes the conditions under which the System Bus signals are tri-stated in internal mode. See the following diagrams for tri-state timing information.

Table 20: Tri-state control

Signal Name	**Tristate control**	**Remarks**

`X_sysdramcs,`
`X_sysgpcs0,`
`X_sysoe,`
`X_syscas`
	`0`	Always driven. These are the SDRAM CS0#, CS1#, DQM, and CAS# signals respectively.

	`1`	Driven one clock after Aries 3 gets bus grant, or if Aries 3 is executing a Systembus cycle and has asserted X_sysbb. In other words, these signals are tri-stated when an external master drives the bus (the X_sysbg is inactive, and X_sysbb cannot be asserted by Aries 3 anyway until it has the grant for at least one clock).
`X_sysa_`	`X`	This bus carries the SDRAM address, bank address, RAS# and WE# signals. This bus is driven by Aries 3 only when it is executing a Systembus cycle. It is tri-stated at all other times.
`X_sysd`	`X`	This bus carries the SDRAM data signals. It is driven by the Aries 3 when it is executing Systembus write cycles, or when an external master is accessing the Aries 3 registers by asserting the Aries 3 chip-select input. It is tri-stated at all other times.

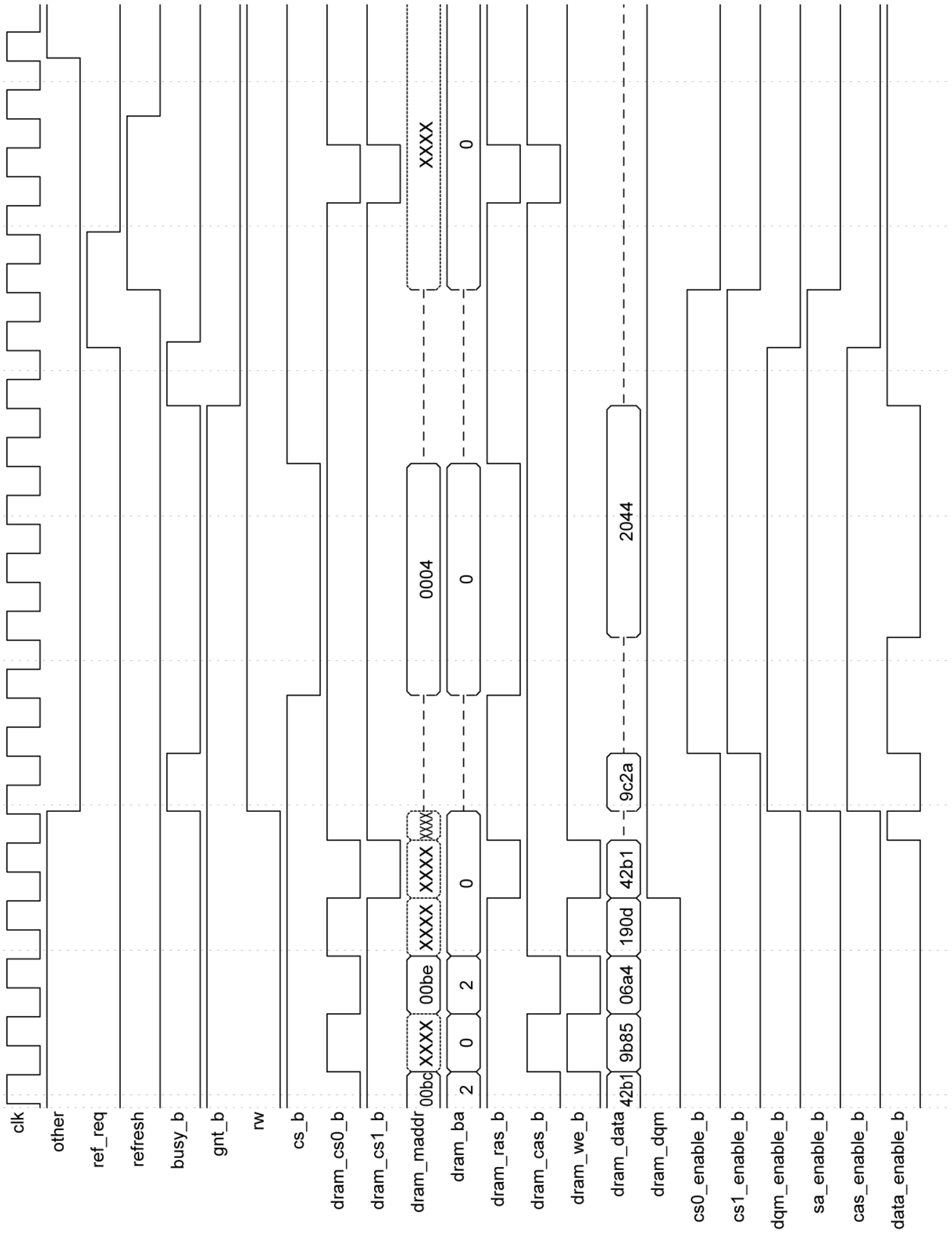


Figure 56: External Master access to Aries register.

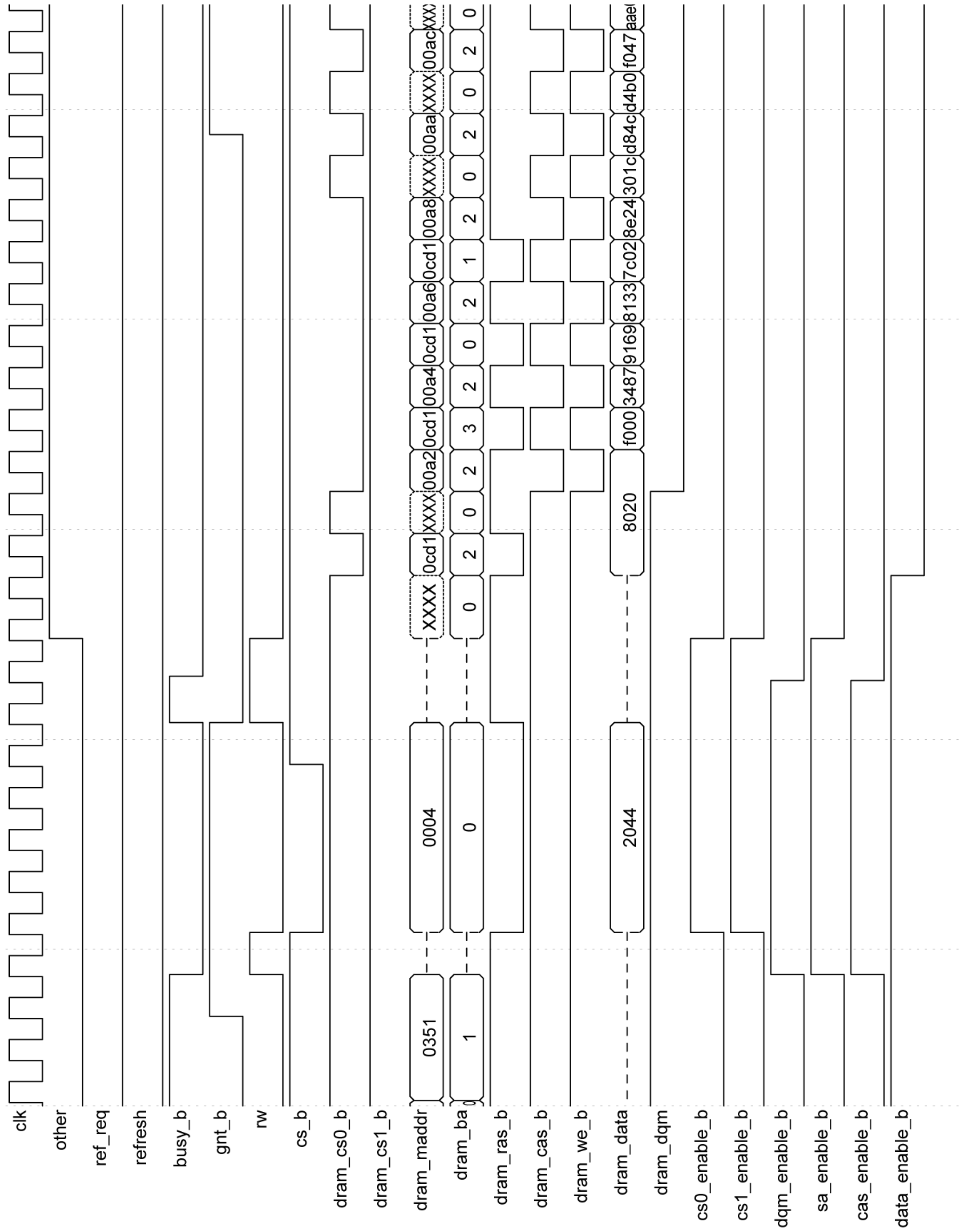


Figure 57: Master Handoff to Aries

System Bus Internal Mode Signal Timing

Aries Bus Master

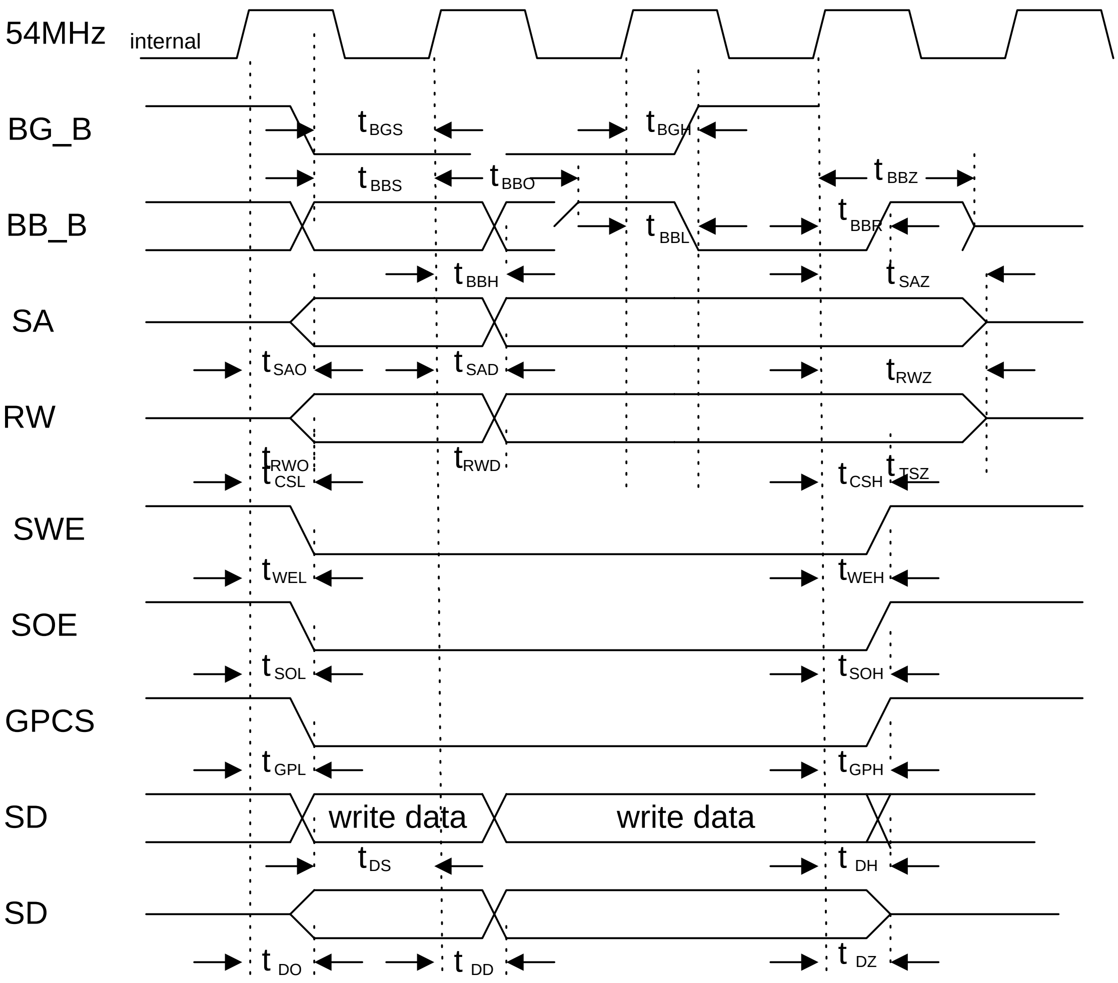


Figure 58

Internal Mode timing

Name	Min (ns)	Max (ns)	Description
t_{WEL}	2	7	Write enable falling edge referenced to System Bus clock
t_{WEH}	2	7	Write enable rising edge referenced to System Bus clock
t_{SOL}	2	7	Output enable falling edge referenced to System Bus clock
t_{SOH}	2	7	Output enable rising edge referenced to System Bus clock
t_{GPL}	1.3	6	General chip select falling edge referenced to System Bus clock
t_{GPH}	1.3	6	General chip select rising edge referenced to System Bus clock

All timings are provisional.

Internal Mode SDRAM timing

Control outputs: RAS, CAS, WE, CS0, CS1, DQM

Data outputs: D15-D0

Addr outputs: A0-A12, BA0-BA1

Data inputs: D0-D15

Name	Min (ns)	Max (ns)	Description
T_{SCO}	10	15	Control output valid from System Bus clock
T_{SDO}	9	15.2	Data output valid from System Bus clock
T_{SAO}	9	16.5	Addr output valid from System Bus clock
T_{SDS}	7		Data input setup to System Bus clock
T_{SDH}	2		Data input hold to System Bus clock

All timings are provisional.

Main Bus SDRAM Interface

Aries supports either one or two 16 bit wide SDRAMs, or either two or four 8 bit wide SDRAMs, for a total of 2 to 16 Megabytes of DRAM. Either 16 or 64 Mbit parts may be used.

The DRAM specification requirements are summarized below:

RAM Specification (Minimum)	
I/O	LVTTL (using 3.3V CMOS IO on Aries)
Frequency	>108MHz
Burst Length	4
Wrap	Sequential
Latency	3
Mode Register	Programmable
Init Sequence	Programmable
TCAS	Programmable (pipelined or pre-fetch)
TRCD	Programmable RAS to CAS delay (min) (2..5)
TRAS	Programmable RAS to pre-charge delay (min) (1..16)
TRP	Programmable Pre-charge to RAS delay (min) (1..8)
TRRD	Programmable RAS to RAS delay (min) (1..8)
TDPL	Programmable Write to Pre-charge delay (min) (1..4)
Refresh Type	RAS only refresh
Refresh Cycle	Programmable

Commands not used	
NOP	No operation
BST	burst stop
REDA	read with auto pre-charge
WRITEA	write with auto pre-charge
PALL	pre-charge all (can be forced during initialization)
MRS	mode register set (can be forced during initialization)

The SDRAM configurations are summarized below:

	A	**B**	**C**	**D**
MBITS	16	16	16	64
BY	16	16	8	16
PARTS	1	2	2	1
MBYTES	2	4	4	8

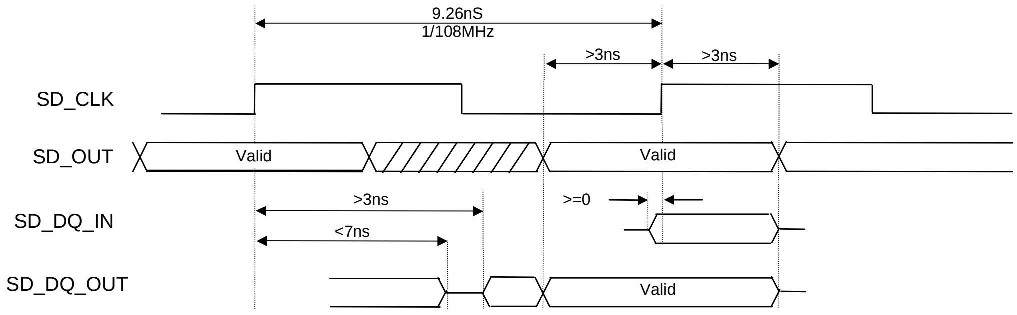


Figure 59: SDRAM bus timing

The following devices are not yet qualified, but appear to meet this criteria, providing speed grades greater than 108MHz are used.

16 bit wide parts:	TOSHIBA TC59S1616AFT NEC uPD4516161 SAMSUNG KM416S1120A
8 bit wide parts:	NEC uPD4516821 TOSHIBA TC59S1608AFT SAMSUNG KM48S1120A

The following is the timing on the Aries Main Bus SDRAM interface:

Name	**Min (ns)**	**Max (ns)**	**Description**
t_{DSU}	0.6		Data input setup time required relative to SD_CLK_IN rising edge
t_{DHD}	0.8		Data input hold time required relative to SD_CLK_IN rising edge
t_{DOD}	4.6	6.7	Data output delay relative to SD_CLK rising edge
t_{AOD}	4.9	7.1	Address output delay relative to SD_CLK rising edge
t_{RASD}	5.0	6.5	SD_RAS output delay relative to SD_CLK rising edge
t_{CASD}	5.2	6.6	SD_CAS output delay relative to SD_CLK rising edge
t_{WED}	5.3	6.9	SD_WE output delay relative to SD_CLK rising edge
t_{DQMD}	4.8	6.0	SD_DQM[0-1] output delay relative to SD_CLK rising edge
t_{CSD}	4.8	5.9	SD_CS[0-1] output delay relative to SD_CLK rising edge

ROM Interface

The Aries ROM interface is used to attach a ROM or Flash type memory for initial bootstrap, and is usually where the operating software is loaded from. The ROM interface may either be on a separate bus, or may be part of the System Bus. When the 208-pin PQFP package option is selected for Aries 3 then the ROM has to be attached to the system bus. This is modes 2 and 3 as described below.

ROM Interface description

The ROM interface has been enhanced for Aries 3 to support NAND flash as well as bus type memory devices, and also to allow the boot ROM memory to be on the System Bus. These options are configured at reset by option resistors on X_vdata[3] and [1] as follows:

X_vdata[3,1]	Mode	Description
00	0	Bus type memory on separate ROM interface pins, as Aries 1 & 2. External octal latches required for address bus.
01	1	NAND flash type memory on ROM interface pins.
10	2	Bus type memory on System Bus interface pins. Full address bus without latches.
11	3	NAND flash type memory on System Bus interface pins.

Pins are used to connect to memory devices for the boot code as follows:

Pin	Mode 00	Mode 01	Mode 10	Mode 11
X_rom_cs_b	CSB	CEB	CSB	CEB
X_rom_oe_b	OEB	REB	*not used*	*not used*
X_rom_we_b	WEB	WEB	*not used*	*not used*
X_rom_lat[0]	latch A0-7	CLE	*not used*	*not used*
X_rom_lat[1]	latch A8-15	ALE	*not used*	*not used*
X_rom_lat[2]	latch A16-23	R/B	*not used*	*not used*
X_rom_d[0-7]	DQ0-7	I/O 1-8	*not used*	*not used*
X_sysd[0-7]			DQ0-7	I/O 1-8
X_systa (syscas / systa)			OEB	REB
X_syswe			WEB	WEB
X_sysd[8]			A0	CLE
X_sysd[9]			A1	ALE
X_sysd[10-15]			A2-7	
X_sysa[2-17]			A8-23	
X_gpio[12]				R/B

Note that this table shows only how the appropriate memory type is connected to Aries. Other functions of the System Bus are not shown here.

Mode 0 Operation

The ROM interface uses a multiplexed address/data bus in order to conserve pins. External to the Aries chip there are three rising-edge triggered octal latches, which hold the 24-bit address. These latches are reloaded at the start of each transfer, if they do not already contain the correct bit pattern. For localized access, usually only one of these will need to be reloaded.

The speed of the ROM interface is programmable by one of the system control registers, and will power up to its slowest possible setting.

A read or write cycle on the ROM interface is preceded by between 0 and 3 ROM address latch update cycles. Internal logic ensures that only the latches that need to be changed are written to, and this improves performance. At the end of reset, all three latches are simultaneously loaded with zeroes.

This mode is the only mode available on Aries 2, but is only supported in the BGA package option for Aries 3. In the QFP package, mode 2 should be used for bus type ROM/Flash memory.

ROM Interface signals

Signal Name	Function	Pins	Active	I/O	Description
`X_rom_d(0-7)`	`DQ0-7`	8	H	IO	Multiplexed address/data
`X_rom_lat(0-2)`		3	L	O	ROM address latch controls
`X_rom_cs_b`	`CE#`	1	L	O	ROM chip select
`X_rom_we_b`	`WE#`	1	L	O	Write enable (for SRAM/FLASH)
`X_rom_oe_b`	`OE#`	1	L	O	Output Enable

Warning: devices on the ROM Interface

In a non-standard Aries implementation with devices on the ROM interface that are not static memory, problems may arise for the device detecting back to back cycles with no intervening address latch update cycle. This situation can only occur from two successive byte transfers to the same address, and may not matter. Suitable programming can avoid this situation. Please note that this will not be of concern when only memory devices are present on this bus.

Mode 1 Operation

Mode 1 supports NAND flash memory, as used in SmartMedia cards, and can actually be used with those cards if a socket is hooked up. Only memory devices of up to 16 Megabytes are supported. The filing system used on SmartMedia cards is not compatible with the Aries boot process as the boot code starts from the base of the memory. There is no hardware support for error detection or correction.

The timing of this interface is programmable as follows, where the control values are programmed in 54 MHz clock cycles. One should be subtracted from the desired value before programming it into the corresponding register bits.

Parameter	Range	Function
flashTwp	1-16	flash write pulse width
flashTwh	1-16	flash hold time for control signals after we
flashTwb	1-64	flash delay from we rising to busy valid
flashTrr	1-64	flash read pre-charge from tRR and tAR2
flashTrp	1-16	flash read pulse width, and data valid time
flashTrh	1-16	flash hold time for control signals after re
flashTsu	1-16	flash setup time for re/we for special cycles
flashTsa	1-16	flash active time for re/we for special cycles
flashTsh	1-16	flash hold time after re/we for special cycles

ROM Interface signals

Signal Name	Function	Pins	Active	I/O	Description
`X_rom_d(0-7)`	`I/O 1-8`	8	H	IO	Multiplexed address/data

X_rom_lat(0)	CLE	1	H	O	Command latch enable
X_rom_lat(1)	ALE	1	H	O	Address latch enable
X_rom_lat(2)	R/B#	1	H	I	Ready / Busy
X_rom_cs_b	CE#	1	L	O	Chip select
X_rom_we_b	WE#	1	L	O	Write enable
X_rom_oe_b	RE#	1	L	O	Read enable

Mode 2 Operation

Mode 2 is functionally the same as mode 0, with the exception that the Flash memory / ROM is connected to the system bus, and external ROM address latches are not required as the full address is available.

In Mode 2 the ROM bus uses the same pins as the system bus, so software that makes use of both busses at the same time will experience reduced performance due to bus sharing.

ROM Interface signals

Signal Name	Function	Pins	Active	I/O	Description
X_sysd(0-7)	DQ0-7	8	H	IO	Data bus
X_sysd(8-15)	A0-A7	8	H	O	Least significant 8 bits of address
X_sysa(2-17)	A8-A23	16	H	O	Most significant 16 bits of address
X_rom_cs_b	CE#	1	L	O	ROM chip select
X_syswe	WE#	1	L	O	Write enable (for SRAM/FLASH)
X_sysoe	OE#	1	L	O	Output Enable

Mode 3 Operation

Mode 3 is functionally the same as mode 1, with the exception that the Flash memory / ROM is connected to the system bus.

In Mode 3 the ROM bus uses the same pins as the system bus, so software that makes use of both busses at the same time will experience reduced performance due to bus sharing.

ROM Interface signals

Signal Name	Function	Pins	Active	I/O	Description
X_sysd(0-7)	I/O 1-8	8	H	IO	Multiplexed address/data
X_sysd(8)	CLE	1	H	O	Command latch enable
X_sysd(9)	ALE	1	H	O	Address latch enable
X_gpio(12)	R/B#	1	H	I	Ready / Busy
X_rom_cs_b	CE#	1	L	O	Chip select
X_syswe	WE#	1	L	O	Write enable
X_syswe	RE#	1	L	O	Read enable

ROM Interface Read and Write Cycle timing

Mode 0 and 2

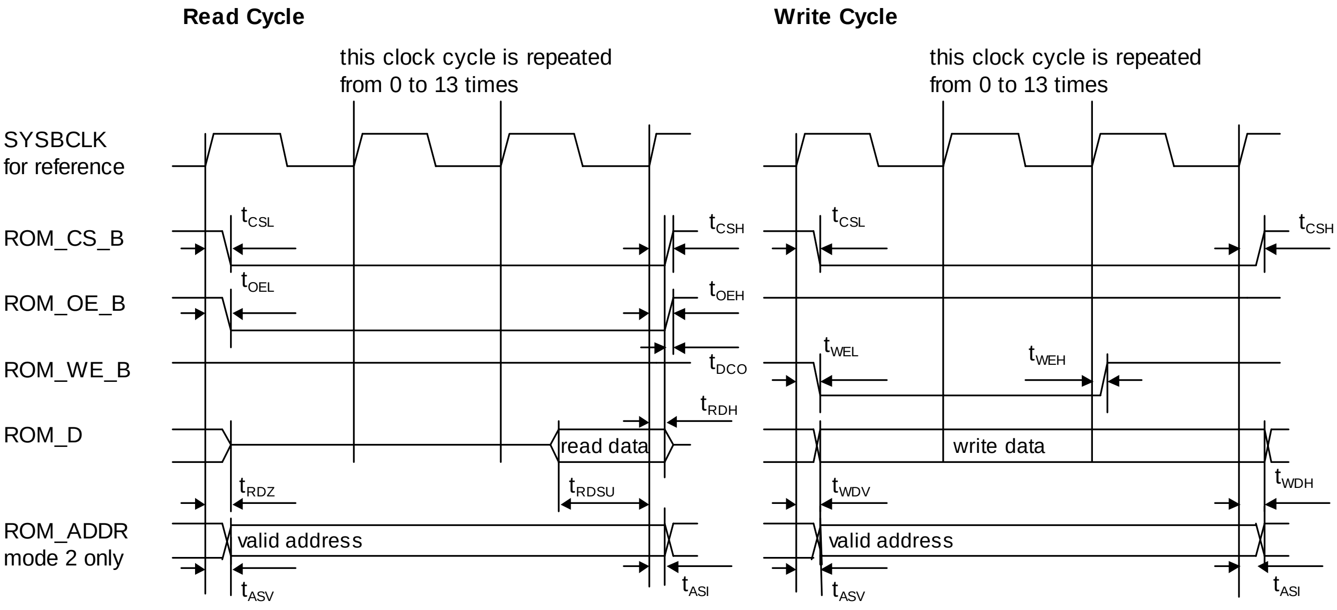


Figure 60

Read and write cycle timing is given relative to SYSBCLK as a point of reference, but note that this interface is intended to be usable as an asynchronous interface, and there should be no need to use a clock to connect a ROM or other static memory device.

Name	Min (ns)	Max (ns)	Description
t_{CSL}	2	6	Chip select falling edge referenced to System Bus clock
t_{CSH}	2	6	Chip select rising edge referenced to System Bus clock
t_{OEL}	2	6	Output enable falling edge referenced to System Bus clock
t_{OEH}	2	6	Output enable rising edge referenced to System Bus clock
t_{WEL}	2	6	Write enable falling edge referenced to System Bus clock
t_{WEH}	2	6	Write enable rising edge referenced to System Bus clock
t_{ASV}	1	6	Address valid time referenced to System Bus clock
t_{ASI}	1	6	Address not valid time referenced to System Bus clock
t_{RDZ}	1	6	Read cycle data bus float delay
t_{RDSU}	3		Read cycle data bus setup time referenced to System Bus clock
t_{RDH}	2		Read cycle data bus hold time referenced to System Bus clock
t_{DCO}	0		Read cycle data hold time referenced to the earlier of either chip select or output enable rising edge.
t_{WDV}	1	6	Write cycle data valid delay referenced to System Bus clock
t_{WDH}	1	6	Write cycle data hold time referenced to System Bus clock

All timings are provisional.

Mode 0 Address Latch timing.

Aries 3 and later

An address latch update cycle is two 54 MHz clock cycles long. Address latch update cycles may have a programmable number of idle cycles before them, so that tri-state disable times for ROM (and other memory devices) may be met before the address byte is presented on the data bus.

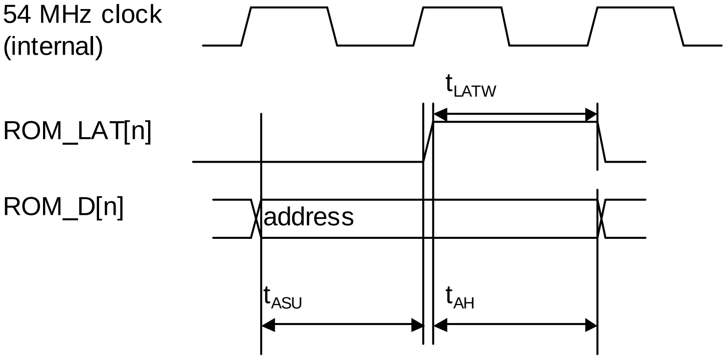


Figure 61

Address latch cycle timing

Name	Min (ns)	Max (ns)	Description
t_{LATW}	t_{CYC} - 2	t_{CYC} + 2	X_rom_lat[n] pulse width
t_{ASU}	t_{CYC} - 2		Address set up time before X_rom_lat rising edge
t_{AH}	t_{CYC} - 2		Address hold time before X_rom_lat rising edge

t_{CYC} is the 54 MHz clock period

All timings are provisional.

Aries 2 and earlier

An address latch update cycle is one 54 MHz clock cycle long. Address latch update cycles may have a programmable number of idle cycles before them, so that tri-state disable times for ROM (and other memory devices) may be met before the address byte is presented on the data bus.

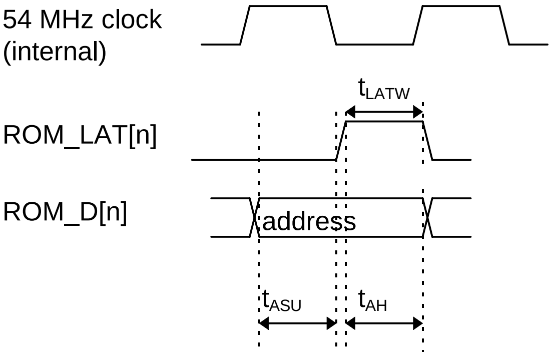


Figure 62

Address latch cycle timing

Name	Min (ns)	Max (ns)	Description
t_{LATW}	(t_{CYC} / 2) - 1	(t_{CYC} / 2) + 1	X_rom_lat[n] pulse width
t_{ASU}	(t_{CYC} / 2) - 2		Address set up time before X_rom_lat rising edge
t_{AH}	(t_{CYC} / 2) - 2		Address hold time before X_rom_lat rising edge

t_{CYC} is the 54 MHz clock period

All timings are provisional.

Mode 1 and 3

NAND Flash timing can be configured to meet the timing requirements of most commercially available memory devices. Please consult with VM Labs for the specific timing values required for a particular manufacturer's part.

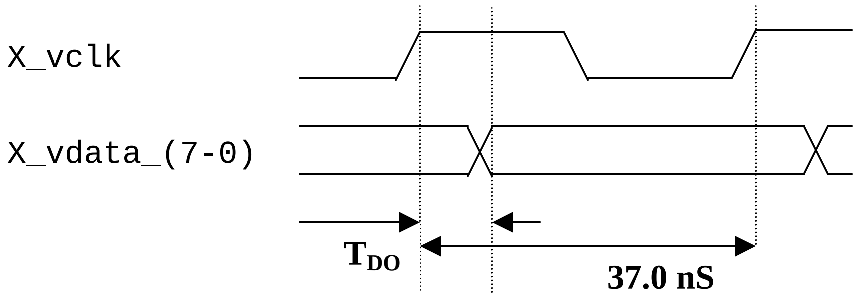
Video Interface

Aries outputs video in an 8-bit synchronous data stream, and follows the CCIR 656 standard. The data is formatted according to CCIR 601.

Aries includes an internal video timing generator. This timing generator can be free running as a master, or can be synchronized to an external Sync source.

In broadcast applications with MPEG2 transport streams, or when it is necessary to mix Aries generated graphics with analog video, external synchronization is necessary. This can be achieved by locking the master Aries clock with the (typically 27 MHz) video master (extracted from the transport stream or the composite video signal), and bringing the video counters into lock either external HSYNC and FIELD signals, or from timing on the video input CCIR 656 stream.

The CCIR 656 output stream includes SAV and EAV codes, and conforms to the CCIR 656 standard.

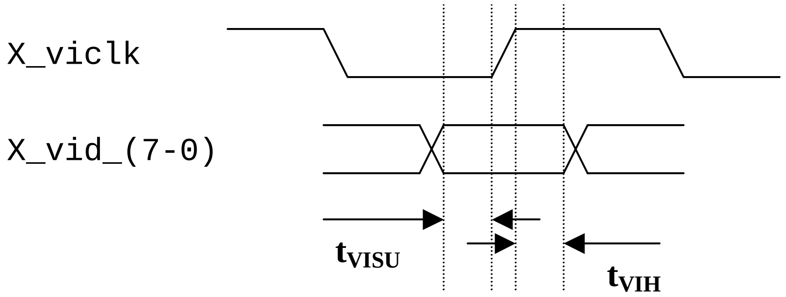


Name	Min (ns)	Max (ns)	Description
t_{DO}	15.0	21.0	VDATA[7:0] output delay referenced to VCLK rising edge

All timings are provisional.

Video Input Interface description

The video input interface conforms to the CCIR 656 interface specification, and should contain CCIR 601 data. Refer to these standards for more information.



Name	Min (ns)	Max (ns)	Description
T_{VISU}	6.0		Data bus setup time referenced to VICLK rising edge
T_{VIH}	6.0		Data bus hold time referenced to VICLK rising edge

All timings are provisional

Audio Interface

The Aries audio interface supports up to eight discrete output channels, plus one IEC 958 (S/PDIF) digital output, and a two stereo audio input interfaces.

Audio Clocking

The audio clock rate, and therefore the sample rate, is usually derived in an Aries 3 application from the internal Audio PLL. It may also be derived from an external clock source. This clock is either output or input on the pin X_aclk. This is divided to produce:

- SBCLK, the synchronous serial bit clock. This is 64, 48 or 32 times the sample rate.
- The IEC 958 output requires two edge positions per bit, and contains 64 bits per sample, therefore requiring a clock 128 times the sample rate.

If an external DAC or ADC has a faster clock requirement (such as 256 times the sample rate), the oscillator should be run at that speed, and the internal Aries pre-scaler used to give the required internal clocks.

The audio clock control output, ACTL, can be configured as either a digital select line to multiplex between two externally derived clocks, or as the output of a phase comparator to control an external VCO. The incoming ACLK and the system master clock are both divided by programmable divide chains to give the input to a phase comparator, whose output appears on the audio clock control output pin ACTL.

PCM Audio Output description

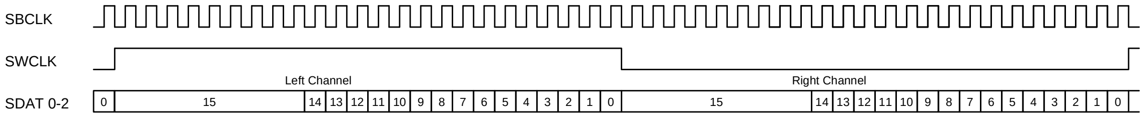
The synchronous serial bit clock, SBCLK, which is derived from an external audio clock as described below, controls audio output timing. One audio sample is then output every 16 or 24 SBCLK cycles, as shown below. The word clock, SWCLK, whose polarity is programmable, gives the framing of the data.

Data is output on four data pins:

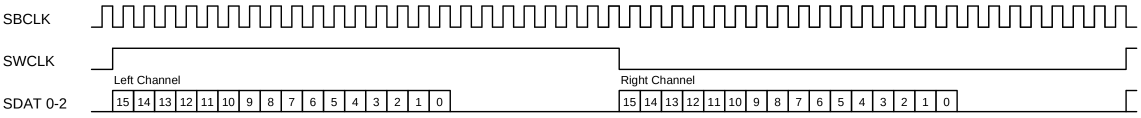
Pin	Left function	Right function
SDAT 0	Left	Right
SDAT 1	Left surround	Right surround
SDAT 2	Center	Low frequency effects
GPIO(1)	Left mix-down	Right mix-down

Possible data output modes are as follows:

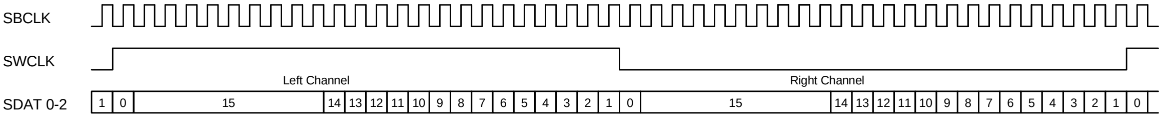
24-bit sample period, left = high SWCLK, right data alignment:



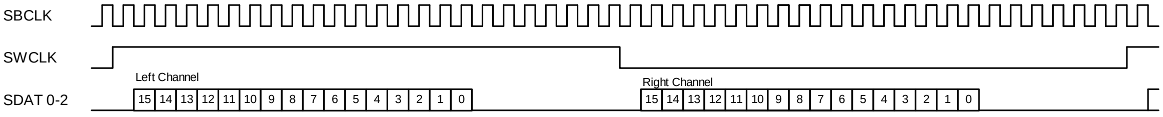
24-bit sample period, left = high SWCLK, left data alignment:



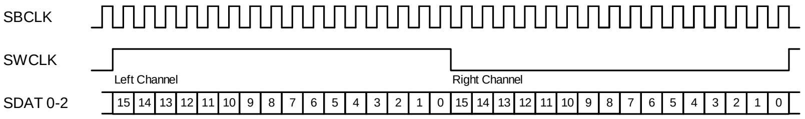
24-bit sample period, left = high SWCLK, right data alignment, delayed data:



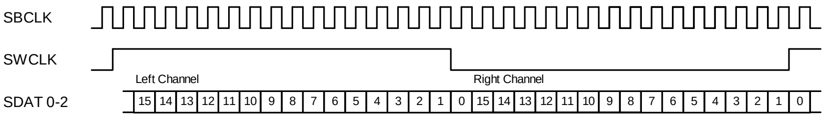
24-bit sample period, left = high SWCLK, left data alignment, delayed data:



16-bit sample period, left = high SWCLK, either data alignment:



16-bit sample period, left = high SWCLK, either data alignment, delayed data:



PCM Audio Output timing

An external device using SBCLK samples the audio outputs. Timing is therefore given with reference to this clock as measured external to Aries.

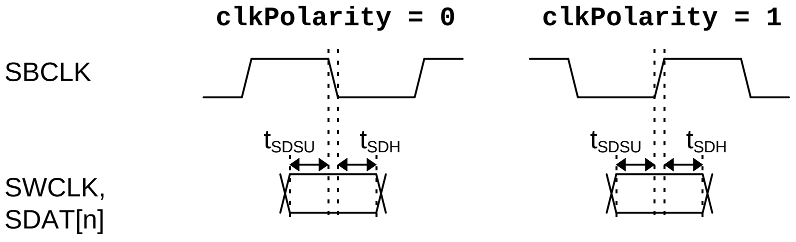


Figure 63

Address latch cycle timing

Name	Min (ns)	Max (ns)	Description
t_{SDSU}	$(t_{ACYC} / 2) - 15$		SWCLK and SDAT setup before SBCLK edge
t_{SDH}	$(t_{ACYC} / 2) - 15$		SWCLK and SDAT hold after SBCLK edge

t_{ACYC} is the audio clock period

All timings are provisional

S/PDIF (IEC 958) Audio Output description

The IEC 958 audio output channel is a serial, output-only, self-clocking interface. Refer to the IEC 958 Standard documentation for further details.

The output channel can operate in two modes:

- 16-bit mode, where the interface is provided with 16-bit audio values. The validity flag is programmable, and the user data field is fixed at zero, for every sub-frame; and the first 32 bits of channel status are programmable with the remaining bits being zero.
- 32-bit mode, where the interface is provided with 32-bit values corresponding to complete sub-frames. The bits corresponding to the Sync Preamble and Parity are ignored as they are generated by the hardware, but all other fields are programmable.

The output channel hardware formats the data according to the IEC 958 standard, and contains a block counter so that it can correctly generate preambles. This counter can be reset.

When the IEC958 channel is enabled, the `preScale` value in the `ssCtrl` register should be set to give a clock 128 times the sample rate. The SBCLK for the synchronous serial output should be set to a half or a quarter of this rate with the `bitScale` value in the `ssCtrl` register. This rules out use of 24 bit samples.

S/PDIF (IEC 958) Audio Output timing

TBD

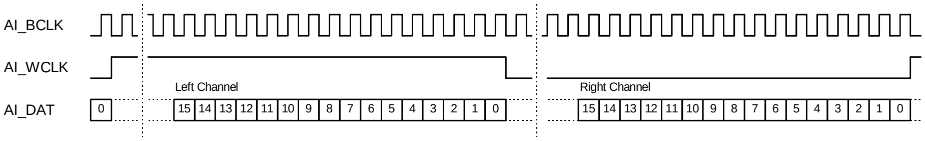
Audio Input Description

Aries supports two stereo audio inputs over synchronous serial (I^2S) interface. This interface is not connected to the audio output channel, and so can be run at a different sample rate if required.

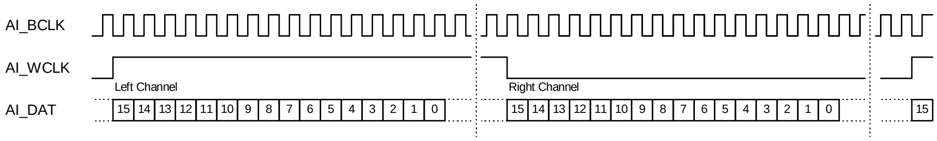
This interface supports the same set of serial data protocols as the transmitter, with some greater flexibility. Left data alignment means that the receiver uses the first 16 bits that follow an edge on word clock; right data alignment means that the receiver uses the 16 bits that precede an edge on word clock. This means that the receiver does not care how long the period is between edges on word clock.

The receiver can also be programmed to accept data framing where the start or end of the data word is some number of AI_BCLK cycles after the edge on AI_WCLK. This is referred to as delayed data mode.

Left = high AI_WCLK, right data alignment:



Left = high AI_WCLK, left data alignment:



Delayed data relative to AI_WCLK cases are also handled, but are not illustrated here.

Audio Input Timing

The audio input channel can be configured as a timing master or slave. It defaults to being a slave, where its timing is derived from an external source. However, it may also be a timing master. As a master, it generates the bit and word clocks, and can also generate a higher frequency over-sample clock, used by some ADCs and Codecs. This is available on one of the GPIO pins.

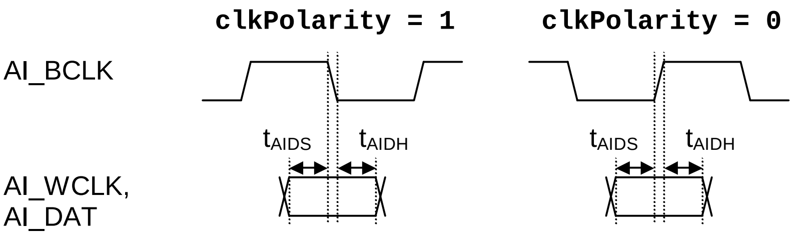


Figure 64

Audio input timing parameters

Name	Min (ns)	Max (ns)	Description
t_{AIDS}	20		AI_WCLK and AI_DAT setup before AI_BCLK edge
t_{AIDH}	20		AI_WCLK and AI_DAT hold after AI_BCLK edge

Notes:

1. Timing applies to both audio input channels.
2. The clock polarity described here is the XOR of the **clkPolarity** and **clkIntPol** bits.

Controller Interface

Controller Interface description

The Controller Interface is used to interface with Controllers and other similar devices, including:

- Game-pad controllers
- Joysticks
- Keyboards
- Mice
- Memory cards
- 3D Glasses (e.g. LCD shutters)
- Modems
- Simple network interfaces

The interface to Controllers themselves consists of eight wires; power, ground, differential clock out, differential data out and differential data in. There are two such interfaces, each with separate data connections to Aries, although the clock is shared.

Controller Interface Timing

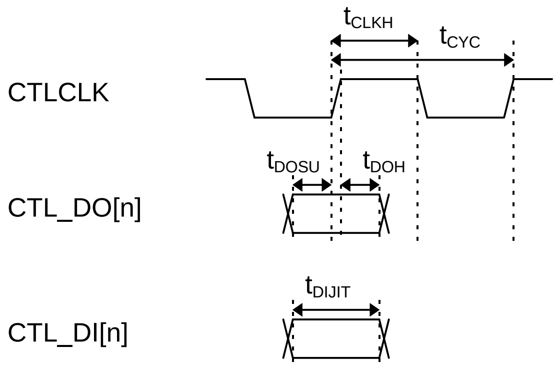


Figure 65

Name	Min (ns)	Max (ns)	Description
t_{CYC}	74	∞	Clock period
t_{CLKH}	$(t_{CYC} / 2) - 10$	$(t_{CYC} / 2) + 10$	Clock high time to show duty cycle variation
t_{DOSU}	$(t_{CYC} / 2) - 15$	-	CTL_DO setup before CTLCLK edge
t_{DOH}	$(t_{CYC} / 2) - 15$	-	CTL_DO hold after CTLCLK edge
t_{DIJIT}	$t_{CYC} - 18$	$t_{CYC} + 18$	Maximum jitter between the position of any data in edge in a burst, expressed for one data value, but the jitter range is actually relative to the start edge of the burst.

All timings are provisional

Coded Data Interface description

The Coded Data Interface is a programmable interface, responsible for conveying compressed data streams to a designated MPE in the MMP system. Compressed data could be in the form of either audio and video elementary streams or transport/program streams. The CDI presents a glueless interface to a number of commercial transport stream demultiplexers, channel decoders and CD-DSPs. Application layer video elementary stream data is carried over a byte-wide interface (most significant byte first) that can be programmed as synchronous or asynchronous. Audio elementary stream data is either multiplexed with video over the byte-wide interface, or carried over an independent bit-serial interface (most significant byte/bit first). The bit-serial interface is also programmable as synchronous or asynchronous. System level compressed data (transport streams and program streams) is carried over the byte-wide interface. The different operating modes of the CDI are listed in the table below..

CDI_ config (24-22)	Config Audio	Config Video	CVDATA/ CAPDATA [7..0]	CVREQ	CVENAB/ CVSTROBE	CVCLK	CASDATA/ CVERRFLG	CAREQ/ CVTOP	CAENAB/ CASTROBE	CACLK
000	Serial Asynch	Asynch	CVDATA	√	STROBE		DATA	CAREQ	STROBE	

001	Serial Asynch	Synch	CVDATA	√	ENABLE	√	DATA	CAREQ	STROBE	
101	Serial Synch	Asynch	CVDATA	√	STROBE		DATA	CAREQ	ENABLE	√
011	Serial Synch	Synch	CVDATA	√	ENABLE	√	DATA	CAREQ	ENABLE	√
100	Parallel Asynch	Asynch	CVDATA/ CAPDATA muxed	√	STROBE			CAREQ	STROBE	
111	Parallel Synch	Synch	CVDATA/ CAPDATA muxed	√	ENABLE	√		CAREQ	ENABLE	(*1)
TS mode Synch			CVDATA		ENABLE	√	ERRFLG	TOP (*2)		
PS mode Synch			CVDATA	√	ENABLE	√	ERRFLG	TOP		

Table 21: Coded Data Interface Operating Modes

Notes:

*1: In the all-synchronous mode, audio and video interfaces share the same clock.

*2: This signal may or may not be present.

Coded Data Interface timing

The Coded Data Interface is designed to satisfy transport stream interface timing requirements outlined in the DAVIC 1.1 A0 Specification. This allows the interface to handle sustained bit rates of up to 72 Mbits/s. For program streams and video/audio elementary streams, request-enable type of handshaking allows data to be input in bursts of up to 16 bytes at a time. The interface is capable of handling CACLK and CVCLK rates of up to 25 MHz. The relationships between different signals in different operating modes are shown in the figures below.

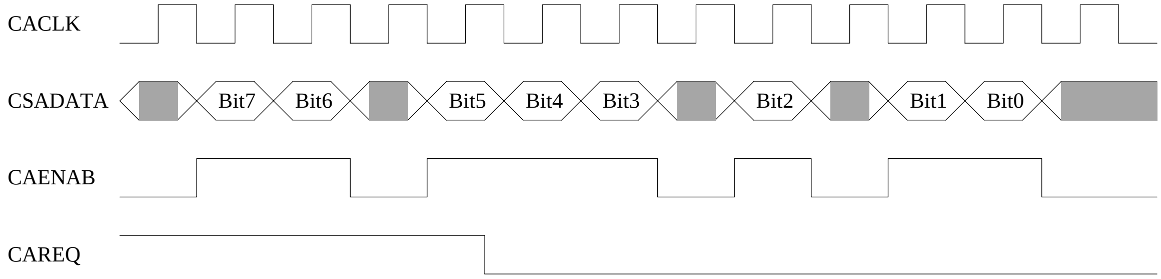


Figure 66: Synchronous Serial Audio mode of Coded Data Interface

Synchronous Serial Audio Mode (rising edge sampled)

Name	Min (ns)	Max (ns)	Description
t_{DSU}	3.5		CASDATA setup time referenced to CACLK rising edge
t_{DH}	2.0		CASDATA hold time referenced to CACLK rising edge
t_{ENSU}	4.5		CAENAB setup time referenced to CACLK rising edge
t_{ENH}	2.0		CAENAB hold time referenced to CACLK rising edge
t_{OUT}	3.0	15.0	CAREQ output delay from CACLK rising edge

All timings are provisional.

Synchronous Serial Audio Mode (falling edge sampled)

Name	Min (ns)	Max (ns)	Description
t_{DSU}	4.5		CASDATA setup time referenced to CACLK falling edge
t_{DH}	2.0		CASDATA hold time referenced to CACLK falling edge
t_{ENSU}	5.0		CAENAB setup time referenced to CACLK falling edge
t_{ENH}	2.0		CAENAB hold time referenced to CACLK falling edge
t_{OUT}	3.0	15.0	CAREQ output delay from CACLK falling edge

All timings are provisional.

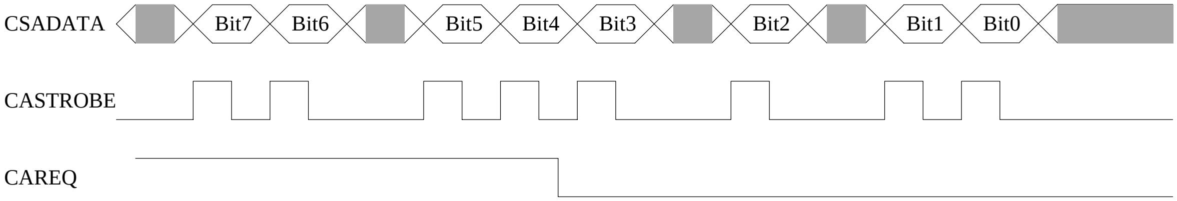


Figure 67: Asynchronous Serial Audio mode of Coded Data Interface

Asynchronous Serial Audio Mode (rising edge sampled)

Name	Min (ns)	Max (ns)	Description
t_{DSU}	3.5		CASDATA setup time referenced to CASTROBE rising edge
t_{DH}	2.0		CASDATA hold time referenced to CASTROBE rising edge
t_{OUT}	3.0	15.0	CAREQ output delay

All timings are provisional.

Asynchronous Serial Audio Mode (falling edge sampled)

Name	Min (ns)	Max (ns)	Description
t_{DSU}	3.5		CASDATA setup time referenced to CASTROBE falling edge
t_{DH}	2.0		CASDATA hold time referenced to CASTROBE falling edge
t_{OUT}	3.0	15.0	CAREQ output delay

All timings are provisional.

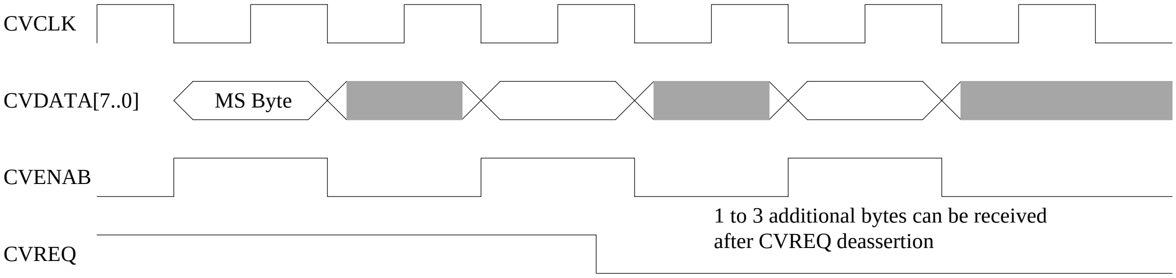

CVCLK
CVDATA[7..0]
MS Byte
CVENAB
CVREQ
1 to 3 additional bytes can be received after CVREQ deassertion

Figure 68: Synchronous Video mode of Coded Data Interface

Synchronous Video Mode (rising edge sampled)

Name	Min (ns)	Max (ns)	Description
t_{DSU}	3.0		CVDATA[7:0] setup time referenced to CVCLK rising edge

t_{DH}	3.0		CVDATA[7:0] hold time referenced to CVCLK rising edge
t_{ENSU}	2.0		CVENAB setup time referenced to CVCLK rising edge
t_{ENH}	2.0		CVENAB hold time referenced to CVCLK rising edge
t_{OUT}	4.0	18.0	CVREQ output delay from CVCLK rising edge

All timings are provisional.

Synchronous Video Mode (falling edge sampled)

Name	**Min (ns)**	**Max (ns)**	**Description**
t_{DSU}	3.0		CVDATA[7:0] setup time referenced to CVCLK falling edge
t_{DH}	3.0		CVDATA[7:0]hold time referenced to CVCLK falling edge
t_{ENSU}	3.0		CVENAB setup time referenced to CVCLK falling edge
t_{ENH}	2.5		CVENAB hold time referenced to CVCLK falling edge
t_{OUT}	4.0	18.0	CVREQ output delay from CVCLK falling edge

All timings are provisional.

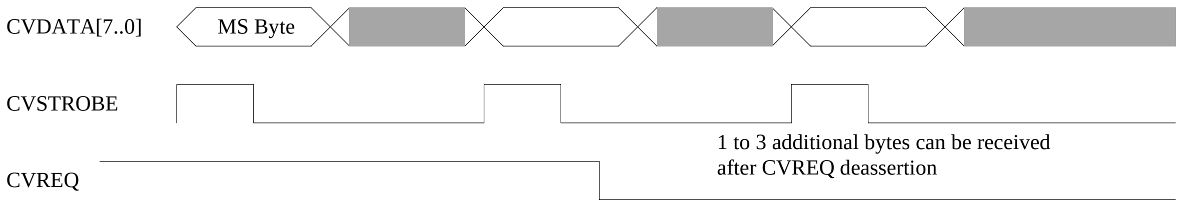


Figure 69: Asynchronous Video mode of Coded Data Interface

Asynchronous Video Mode (rising edge sampled)

Name	**Min (ns)**	**Max (ns)**	**Description**
t_{DSU}	3.0		CVDATA[7:0] setup time referenced to CVSTROBE rising edge
t_{DH}	2.0		CVDATA[7:0] hold time referenced to CVSTROBE rising edge
t_{OUT}	4.0	18.0	CVREQ output delay

All timings are provisional.

Asynchronous Video Mode (falling edge sampled)

Name	**Min (ns)**	**Max (ns)**	**Description**
t_{DSU}	3.0		CVDATA[7:0] setup time referenced to CVSTROBE falling edge
t_{DH}	2.0		CVDATA[7:0] hold time referenced to CVSTROBE falling edge
t_{OUT}	4.0	18.0	CVREQ output delay

All timings are provisional.

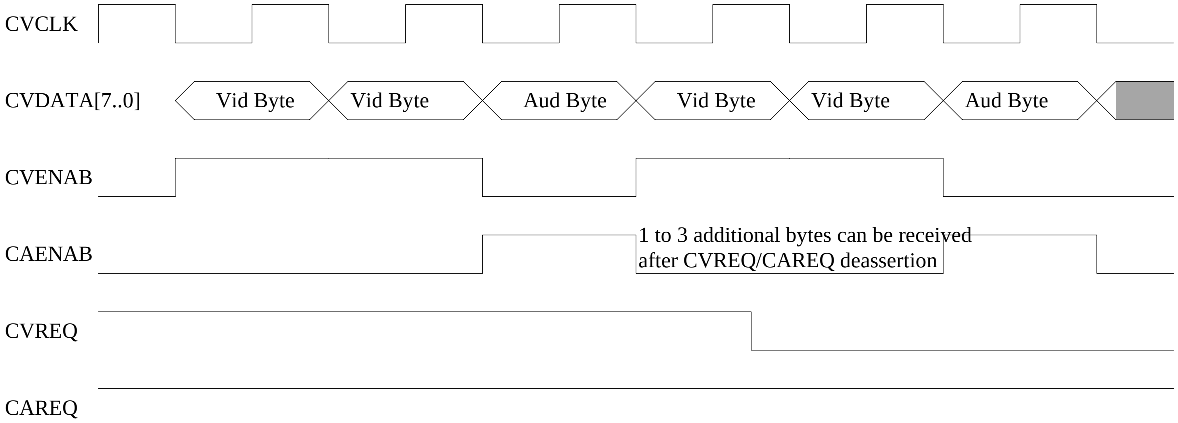


Figure 70: Synchronous multiplexed Parallel Video/Audio mode of Coded Data Interface

Synchronous multiplexed parallel video/audio Mode (rising edge sampled)

Name	Min (ns)	Max (ns)	Description
t_{DSU}	3.0		CVDATA[7:0] setup time referenced to CVCLK rising edge
t_{DH}	3.0		CVDATA[7:0] hold time referenced to CVCLK rising edge
t_{VENSU}	3.0		CVENAB setup time referenced to CVCLK rising edge
t_{VENH}	3.0		CVENAB hold time referenced to CVCLK rising edge
t_{AENSU}	3.0		CAENAB setup time referenced to CVCLK rising edge
t_{AENH}	3.0		CAENAB hold time referenced to CVCLK rising edge
t_{VOUT}	4.0	18.0	CVREQ output delay from CVCLK rising edge
t_{AOUT}	3.0	15.0	CAREQ output delay from CVCLK rising edge

All timings are provisional.

Synchronous multiplexed parallel video/audio Mode (falling edge sampled)

Name	Min (ns)	Max (ns)	Description
t_{DSU}	3.0		CVDATA[7:0] setup time referenced to CVCLK falling edge
t_{DH}	3.0		CVDATA[7:0] hold time referenced to CVCLK falling edge
t_{VENSU}	3.0		CVENAB setup time referenced to CVCLK falling edge
t_{VENH}	3.0		CVENAB hold time referenced to CVCLK falling edge
t_{AENSU}	3.0		CAENAB setup time referenced to CVCLK falling edge
t_{AENH}	3.0		CAENAB hold time referenced to CVCLK falling edge
t_{VOUT}	4.0	18.0	CVREQ output delay from CVCLK falling edge
t_{AOUT}	3.0	15.0	CAREQ output delay from CVCLK falling edge

All timings are provisional.

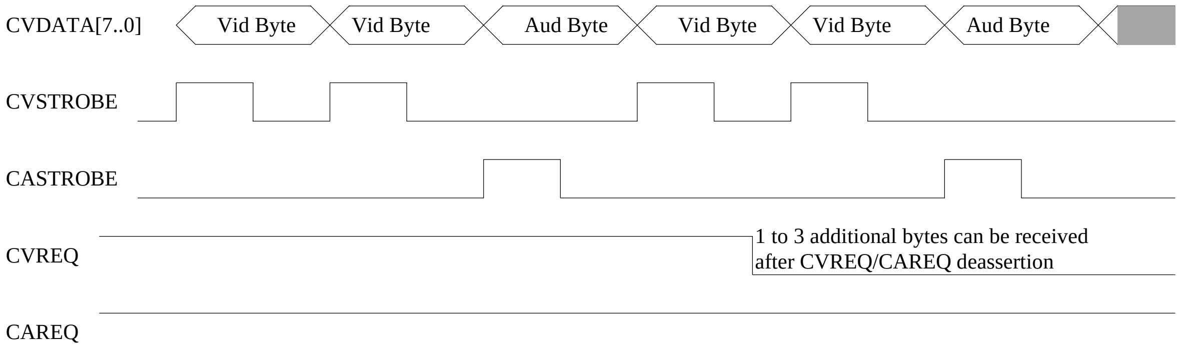


Figure 71: Asynchronous multiplexed Parallel Video/Audio mode of Coded Data Interface

Asynchronous multiplexed parallel video/audio Mode (rising edge sampled)

Name	**Min (ns)**	**Max (ns)**	**Description**
t_{VDSU}	3.0		CVDATA[7:0] setup time referenced to CVSTROBE rising edge
t_{VDH}	3.0		CVDATA[7:0] hold time referenced to CVSTROBE rising edge
t_{ADSU}	3.0		CVDATA[7:0] setup time referenced to CASTROBE rising edge
t_{ADH}	3.0		CVDATA[7:0] hold time referenced to CASTROBE rising edge
t_{VOUT}	4.0	18.0	CVREQ output delay
t_{AOUT}	3.0	15.0	CAREQ output delay

All timings are provisional.

Asynchronous multiplexed parallel video/audio Mode (falling edge sampled)

Name	**Min (ns)**	**Max (ns)**	**Description**
t_{VDSU}	3.0		CVDATA[7:0] setup time referenced to CVSTROBE falling edge
t_{VDH}	3.0		CVDATA[7:0] hold time referenced to CVSTROBE falling edge
t_{ADSU}	3.0		CVDATA[7:0] setup time referenced to CASTROBE falling edge
t_{ADH}	3.0		CVDATA[7:0] hold time referenced to CASTROBE falling edge
t_{VOUT}	4.0	18.0	CVREQ output delay
t_{AOUT}	3.0	15.0	CAREQ output delay

All timings are provisional.

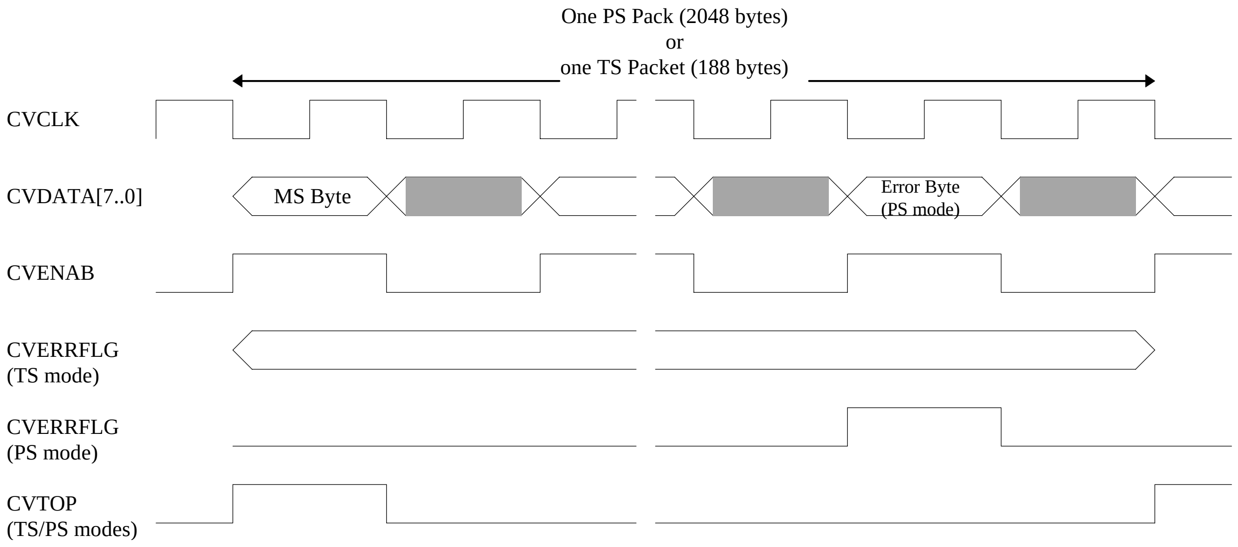


Figure 72: Transport Stream/Program Stream modes of Coded Data Interface

Transport Stream/Program Stream Mode (rising edge sampled)

Name	Min (ns)	Max (ns)	Description
t_{DSU}	3.0		CVDATA[7:0] setup time referenced to CVCLK rising edge
t_{DH}	3.0		CVDATA[7:0] hold time referenced to CVCLK rising edge
t_{VENSU}	3.0		CVENAB setup time referenced to CVCLK rising edge
t_{VENH}	3.0		CVENAB hold time referenced to CVCLK rising edge
t_{ERRSU}	3.0		CVERRFLG setup time referenced to CVCLK rising edge
t_{ERRH}	3.0		CVERRFLG hold time referenced to CVCLK rising edge
t_{TOPSU}	3.0		CVTOP setup time referenced to CVCLK rising edge
t_{TOPH}	3.0		CVTOP hold time referenced to CVCLK rising edge

All timings are provisional.

Transport Stream/Program Stream Mode (falling edge sampled)

Name	Min (ns)	Max (ns)	Description
t_{DSU}	3.0		CVDATA[7:0] setup time referenced to CVCLK falling edge
t_{DH}	3.0		CVDATA[7:0] hold time referenced to CVCLK falling edge
t_{VENSU}	3.0		CVENAB setup time referenced to CVCLK falling edge
t_{VENH}	3.0		CVENAB hold time referenced to CVCLK falling edge
t_{ERRSU}	3.0		CVERRFLG setup time referenced to CVCLK falling edge
t_{ERRH}	3.0		CVERRFLG hold time referenced to CVCLK falling edge
t_{TOPSU}	3.0		CVTOP setup time referenced to CVCLK falling edge
t_{TOPH}	3.0		CVTOP hold time referenced to CVCLK falling edge

All timings are provisional.

Clocking and Reset

Description

The system is designed to be clocked from an internal 108 MHz PLL or external 108 MHz clock source. The 108 MHz clock input is divided by four and output again on the PLL_REF signal, which is used for frequency comparison with an external 27 MHz source. In normal operation the video clock, VCLK, is the same as PLL_REF.

During power on reset, the system clocks run, so that PLL_REF, VCLK and SYSBCLK are all output normally. VCLK is run in its default (27 MHz) speed. Power on reset should be applied with the main clock input running for at least 24 main clock cycles. However, note that the power up mode selection bits must also be settled on the rising edge of reset, and this may dictate longer reset pulses. These are described under “Power Up Mode Selection” below.

The system reset input is active low, and a falling edge on this signal asynchronously resets the Aries chip. When this signal is released, the trailing edge of the internal reset is held until a subsequent rising edge of the internal clock to assure that the system starts cleanly.

Clock Timing

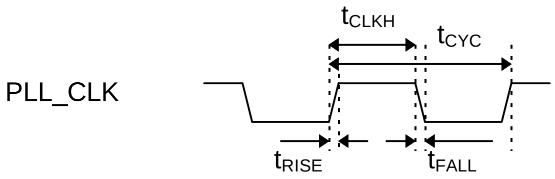


Figure 73

Name	Min (ns)	Max (ns)	Description
t_{CYC}	9.26	∞	Clock period
t_{CLKH}	$(t_{CYC} / 2) - 1$	$(t_{CYC} / 2) + 1$	Clock high time to show duty cycle variation
t_{RISE}	0	1	Clock rise time
t_{FALL}	0	1	Clock fall time

All timings are provisional

Power Up Mode Selection

The power up mode of the Aries device is determined by bias resistors attached to the video output pins VDATA(7:0). During reset, and for two clock cycles after the end of it these pins are not driven. One clock cycle after the end of reset their state is latched within Aries, and used to configure the device with parameters that cannot be set under software control. This allows a significant period for these signal lines to settle, so a fairly large resistor can be used (probably 10 Kohms), which should have no effect when these lines are outputs.

Power up configuration in the VDATA bus is as follows:

Pins	**Aries revision**			**Function**
	1	**2**	**3**	
7-5	✓	✓	✓	Sets the Aries System Bus slave register address offset. The slave registers 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. This allows up to 8 Aries devices to be present on the same System Bus, without requiring them to contain different firmware.
4				unused
3			✓	System Bus boot ROM mode – configures the ROM interface to use the System Bus (see above under ROM interface
2	✓	✓	✓	Test mode – must be pulled low.
1			✓	Flash memory mode – configure the ROM interface to use a NAND flash bus (see above under ROM interface).
0				unused

ARIES 3 BUG LIST

This list contains all the known bugs in the Aries 2 silicon.

Last update 18 June 2001

Bug levels are:

0. A quirk that probably ought to be fixed.
1. This bug can be worked around in software without significant system overhead.
2. This bug can be worked round, but has a significant system impact.
3. This bug cannot be entirely worked around, and could mask further bugs.

MPE

Dcache freeze

Level	1
Description	This bug can cause the MPE to freeze when a "load" access to a cacheable address results in a dcache miss, and the immediately following instruction is a "load" or "store" access to a non-cacheable address (including local MPE memory and registers).
Work-round	The simplest workaround is to make sure that any ld from a cacheable address is not immediately followed by a ld or st access to a non-cacheable address, inserting a nop if necessary. The assembler will probably be updated so that it can warn about or automatically fix this problem. Because it might become wasteful for the assembler to assume that every indirect ld instruction is cacheable, there will likely need to be options for finer control by user directives.
Date added	Unchanged from the MMP-L3A

Icache overlay

Level	1
Description	This bug can cause incorrect results if while running an Icached program, the MPE IRAM is accessed by dma or by a ld/st instruction.
Work-round	(a) The only workaround is to avoid any other access to the IRAM when an icached program is running. This does not affect most normal uses of the MPE icache. I believe it causes only a few esoteric restrictions: an icached program should not perform an explicit dma to bring a code overlay into its own IRAM (a strange concept actually); while an MPE is running an icached program, other processors should not dma into any part of that MPE's IRAM; icached programs should not use dma command blocks in IRAM; and finally, icached programs should not use ld/st to access IRAM (which, if you noticed the MPE's Harvard-like architecture, you probably thought was impossible anyway). (b) For the specific problem of how to move an MPE from icached execution to local execution when all the IRAM is dedicated to the icache, there are a number of solutions. Probably the most elegant is to dma a small program to DTRAM, then jump directly there to execute it; that program can then dma some other code to the base of IRAM and jump directly there, thus converting to local non-icache execution.
Date added	Unchanged from the MMP-L3A

Main Bus DMA and data cache conflict

Level	1
Description	If a data cache miss occurs while a Main Bus DMA is in progress, then the load instruction can return invalid data. The correct data is in fact loaded into the cache line.
Work-round	Do not use the data cache while performing Main Bus DMA.
Date added	4 October 1999

Data address breakpoints fail to trigger on some instruction forms

Level	1
Description	Data-address-write-breakpoints will fail to trigger when the monitored location is written by either of the instruction forms "st_s #nn,<labelC>" or "st_s #nnnn,<labelD>". (Note: this is both the 32-bit and 64-bit forms of st_s with an immediate data value.)
Work-round	
Date added	3 August, 2000

Main Bus DMA and SDRAM Interface

MPE Instruction Tags cannot be accessed using Main Bus DMA

Level	0
Description	Main Bus DMA to the instruction tags causes a DMA exception
Work-round	If you need DMA access, use the Other Bus
Date added	Unchanged from the MMP-L3A

Vertical filtering of MPEG data does not filter chroma

Level	0
Description	If you apply a vertical filter to MPEG data, and scale it up, you will sometimes notice a blockiness in chroma. This is because only luma is filtered in this mode.
Work-round	For static images, apply a software filter.
Date added	January 30, 1999

VDG

Ringing artifacts from horizontal linear filter

Level	2
Description	There is a bug in the VDG's linear horizontal filter, that is introducing ringing-like artifacts in both natural MPEG image and synthetic color bars images. It is also possible that the filter amplifies edge artifacts that occur during the encoding or decoding process and that would normally be invisible. This is what we have been able to confirm and verify: - When a 528-pixel wide source is expanded to 720 pixels on the display, the video firmware engages a linear four-tap filter. In certain areas of low or little variation in luma and chroma (i.e., uniform color), faint vertical strips appear. These correspond to macroblock edges (16 source pixels apart). In some cases, they correspond to block edges (8 source pixels apart).

	- When we display a color bar image (synthetically created, so the luma and chroma in each bar is EXACTLY uniform), expanding a 528-pixel source rectangle to 720 display pixels, the vertical strips are quite evident in the magenta bar, less evident in the red bar, and not evident in the other colors. The stripes do not appear when the filter is turned off or if we use the custom four-tap filter.
Work-round	a) Disable the horizontal filter altogether; here we suspect that the expanded image will not look too pretty b) Use the custom four-tap filter; In this case some images will exhibit real ringing (the custom filter coefficients were designed for showing reduced images in a certain narrow range). c) Experiment with other scaling ratios. E.g. 528 to 720 pixel expansion shows the problem, 528 to 704 expansion makes it far less visible,
Date added	September 5, 2001

Communication Bus

Receive buffer full flag is set too soon

Level	0
Description	The receive buffer full flag is set as soon as the first long word of data is received. This means that the complete receive buffer register is not valid for a further three clock cycles.
Work-round	Care should be taken when optimizing code to ensure that you don’t read the data until at least three clock cycles after you have ascertained that there is data to be read. Interrupt code is not affected as you can't get into an interrupt service routine that fast.
Date added	Unchanged from the MMP-L3A

Other Bus

The Debug Controller is unable to do remote transfers on the Other Bus

Level	1
Description	If the debug controller does an Other Bus transfer where the debugDmaData registers are neither the source nor the destination of the transfer, then the Other Bus Transfer Pending flag is never cleared.
Work-round	Use an MPE to perform the transfer.
Date added	January 21, 1999

GENERAL IO

The debug controller cannot do two long-word DMA transfers on the Other Bus

Level	1
Description	Other Bus read transfers of length two do not work for the debug controller. The second long-word of data is incorrect. Length two transfers work fine for Other Bus writes, and for all Main Bus transfers.
Work-round	Other Bus read transfers should be restricted to one long-word.
Date added	March 3, 1999

Debug Controller

Soft resets do not work unless booting from the ROM bus as Aries 2

Level	2
Description	When a software generated reset happens because a one was written to the debug_reset bit in the debugCtrl register, the bits that control where Aries 3 boots from are reset to zero, which only works if the chip is booting from normal bus ROM/Flash on the ROM bus , as in Aries 2 and below.
Work-round	Add external circuitry to generate a soft reset when a GPIO pin is driven.
Date added	October 19, 2001

I2C Interface

There is no comminfo data on packets sent by the I2C controller

Level	0
Description	Software cannot tell what a packet sent by the I2C controller is in response to. This makes it hard to operate the master and slave independently unless communication bus reads block.
Work-round	Reads have to wait for the response.
Date added	June 19, 2001

Audio Interface

FIFO underflow prevention is broken

Level	1
Description	The preventFifoUnderflow has an incorrect test, and prevents audio output from working at all when set.
Work-round	Do not use this function. Underflow should never occur if DMA is set up properly.
Date added	October 12, 2001

Audio In is turned on at reset

Level	0
Description	As it comes out of reset, the audio in channel may send one packet to MPE 0 shortly after reset.
Work-round	Boot ROMs created after 28 Jan 98 contain appropriate code to fix this. The code that fixes Oz will also work for Aries
Date added	Carried over from the MMP-L3A. Only sends one packet in Aries 3.

Top and Error flags from audio input channel 2 to the CDI are swapped

Level	1
Description	The Top and Error flags captured on audio input channel 2 and passed to the CDI are

	reversed. This means the signal captured on X_casdata is passed to the CDI as top, and the signal captured on X_careq is passed to the CDI as error. This affects systems capturing data on audio input channel 2, and also capturing errors or framing flags on this channel, when the data is passed through to the CDI.
Work-round	These signals should be swapped externally, and if the data is also sometimes received directly from the audio input channel, then the software should be changed to reverse the convention in the documentation, i.e. to: 6-7 right flag (or error) data 8-9 right sync (or top) data 10-11 left flag (or error) data 12-13 left sync (or top) data
Date added	October 11, 1999

MMP-L3C (ARIES 2) BUG LIST

This list contains all the known bugs in the Aries 2 silicon.

Last update 11 October 1999

Bug levels are:

4. A quirk that probably ought to be fixed.
5. This bug can be worked around in software without significant system overhead.
6. This bug can be worked round, but has a significant system impact.
7. This bug cannot be entirely worked around, and could mask further bugs.

MPE

Dcache freeze

Level	1
Description	This bug can cause the MPE to freeze when a "load" access to a cacheable address results in a dcache miss, and the immediately following instruction is a "load" or "store" access to a non-cacheable address (including local MPE memory and registers).
Work-round	The simplest workaround is to make sure that any ld from a cacheable address is not immediately followed by a ld or st access to a non-cacheable address, inserting a nop if necessary. The assembler will probably be updated so that it can warn about or automatically fix this problem. Because it might become wasteful for the assembler to assume that every indirect ld instruction is cacheable, there will likely need to be options for finer control by user directives.
Date added	Unchanged from the MMP-L3A

Icache overlay

Level	1
Description	This bug can cause incorrect results if while running an Icached program, the MPE IRAM is accessed by dma or by a ld/st instruction.
Work-round	(a) The only workaround is to avoid any other access to the IRAM when an icached program is running. This does not affect most normal uses of the MPE icache. I believe it causes only a few esoteric restrictions: an icached program should not perform an explicit dma to bring a code overlay into its own IRAM (a strange concept actually); while an MPE is running an icached program, other processors should not dma into any part of that MPE's IRAM; icached programs should not use dma command blocks in IRAM; and finally, icached programs should not use ld/st to access IRAM (which, if you noticed the MPE's Harvard-like architecture, you probably thought was impossible anyway). (b) For the specific problem of how to move an MPE from icached execution to local execution when all the IRAM is dedicated to the icache, there are a number of solutions. Probably the most elegant is to dma a small program to DTRAM, then jump directly there to execute it; that program can then dma some other code to the base of IRAM and jump directly there, thus converting to local non-icache execution.
Date added	Unchanged from the MMP-L3A

Main Bus DMA and data cache conflict

Level	1
Description	If a data cache miss occurs while a Main Bus DMA is in progress, then the load instruction can return invalid data. The correct data is in fact loaded into the cache line.
Work-round	Do not use the data cache while performing Main Bus DMA.
Date added	4 October 1999

Data address breakpoints fail to trigger on some instruction forms

Level	1
Description	Data-address-write-breakpoints will fail to trigger when the monitored location is written by either of the instruction forms "st_s #nn,<labelC>" or "st_s #nnnn,<labelD>". (Note: this is both the 32-bit and 64-bit forms of st_s with an immediate data value.)
Work-round	
Date added	3 August, 2000

Main Bus DMA and SDRAM Interface

MPE Instruction Tags cannot be accessed using Main Bus DMA

Level	0
Description	Main Bus DMA to the instruction tags causes a DMA exception
Work-round	If you need DMA access, use the Other Bus
Date added	Unchanged from the MMP-L3A

Vertical filtering of MPEG data does not filter chroma

Level	0
Description	If you apply a vertical filter to MPEG data, and scale it up, you will sometimes notice a blockiness in chroma. This is because only luma is filtered in this mode.
Work-round	For static images, apply a software filter.
Date added	January 30, 1999

Communication Bus

Receive buffer full flag is set too soon

Level	0
Description	The receive buffer full flag is set as soon as the first long word of data is received. This means that the complete receive buffer register is not valid for a further three clock cycles.
Work-round	Care should be taken when optimizing code to ensure that you don’t read the data until at least three clock cycles after you have ascertained that there is data to be read. Interrupt code is not affected as you can't get into an interrupt service routine that fast.
Date added	Unchanged from the MMP-L3A

Other Bus

The Debug Controller is unable to do remote transfers on the Other Bus

Level	1
Description	If the debug controller does an Other Bus transfer where the debugDmaData registers are neither the source nor the destination of the transfer, then the Other Bus Transfer Pending flag is never cleared.
Work-round	Use an MPE to perform the transfer.
Date added	January 21, 1999

GENERAL IO

The debug controller cannot do two long-word DMA transfers on the Other Bus

Level	1
Description	Other Bus read transfers of length two do not work for the debug controller. The second long-word of data is incorrect. Length two transfers work fine for Other Bus writes, and for all Main Bus transfers.
Work-round	Other Bus read transfers should be restricted to one long-word.
Date added	March 3, 1999

The I2C controller cannot send a start code in the middle of a transfer

Level	1
Description	If you send a byte with type xmitStart in the middle of a transfer, the controller will hang and nothing will be sent. The only supported sequence is start bit, transmitted and received bytes, then a stop bit.
Work-round	Bit-bang the I2C to achieve other modes.
Date added	September 9, 2000

System Bus Interface

The read addresses for External Host Interrupt Control and Status are swapped

Level	1
Description	The read addresses for the External Host Interrupt Control register and the External Host Interrupt Status register are reversed.
Work-round	Write to the External Host Interrupt Control register at offset $14 and read it at offset $18. Read to the External Host Interrupt Status register at offset $14.
Date added	November 18, 1998

Audio Interface

Audio In is turned on at reset

Level	0
Description	As it comes out of reset, the audio in channel may send one packet to MPE 0 shortly after reset.
Work-round	Boot ROMs created after 28 Jan 98 contain appropriate code to fix this. The code that fixes

	Oz will also work for Aries
Date added	Carried over from the MMP-L3A. Only sends one packet in the MMP-L3B.

Top and Error flags from audio input channel 2 to the CDI are swapped

Level	1
Description	The Top and Error flags captured on audio input channel 2 and passed to the CDI are reversed. This means the signal captured on X_casdata is passed to the CDI as top, and the signal captured on X_careq is passed to the CDI as error. This affects systems capturing data on audio input channel 2, and also capturing errors or framing flags on this channel, when the data is passed through to the CDI.
Work-round	These signals should be swapped externally, and if the data is also sometimes received directly from the audio input channel, then the software should be changed to reverse the convention in the documentation, i.e. to: 6-7 right flag (or error) data 8-9 right sync (or top) data 10-11 left flag (or error) data 12-13 left sync (or top) data
Date added	October 11, 1999

MMP-L3B (ARIES 1) BUG LIST

This list contains all the known bugs in the Aries silicon.

Last update 11 October 1999

MPE

Dcache freeze

Level	1
Description	This bug can cause the MPE to freeze when a "load" access to a cacheable address results in a dcache miss, and the immediately following instruction is a "load" or "store" access to a non-cacheable address (including local MPE memory and registers).
Work-round	The simplest workaround is to make sure that any ld from a cacheable address is not immediately followed by a ld or st access to a non-cacheable address, inserting a nop if necessary. The assembler will probably be updated so that it can warn about or automatically fix this problem. Because it might become wasteful for the assembler to assume that every indirect ld instruction is cacheable, there will likely need to be options for finer control by user directives.
Date added	Unchanged from the MMP-L3A

Icache overlay

Level	1
Description	This bug can cause incorrect results if while running an Icached program, the MPE IRAM is accessed by dma or by a ld/st instruction.
Work-round	(a) The only workaround is to avoid any other access to the IRAM when an icached program is running. This does not affect most normal uses of the MPE icache. I believe it causes only a few esoteric restrictions: an icached program should not perform an explicit dma to bring a code overlay into its own IRAM (a strange concept actually); while an MPE is running an icached program, other processors should not dma into any part of that MPE's IRAM; icached programs should not use dma command blocks in IRAM; and finally, icached programs should not use ld/st to access IRAM (which, if you noticed the MPE's Harvard-like architecture, you probably thought was impossible anyway). (b) For the specific problem of how to move an MPE from icached execution to local execution when all the IRAM is dedicated to the icache, there are a number of solutions. Probably the most elegant is to dma a small program to DTRAM, then jump directly there to execute it; that program can then dma some other code to the base of IRAM and jump directly there, thus converting to local non-icache execution.
Date added	Unchanged from the MMP-L3A

Main Bus DMA and data cache conflict

Level	1
Description	If a data cache miss occurs while a Main Bus DMA is in progress, then the load instruction can return invalid data. The correct data is in fact loaded into the cache line.
Work-round	Do not use the data cache while performing Main Bus DMA.
Date added	4 October 1999

Main Bus DMA and SDRAM Interface

MPE Instruction Tags cannot be accessed using Main Bus DMA

Level	0
Description	Main Bus DMA to the instruction tags causes a DMA exception
Work-round	If you need DMA access, use the Other Bus
Date added	Unchanged from the MMP-L3A

Vertical filtering of MPEG data does not filter chroma

Level	0
Description	If you apply a vertical filter to MPEG data, and scale it up, you will sometimes notice blockiness in chroma. This is because only luma is filtered in this mode.
Work-round	For static images, apply a software filter.
Date added	January 30, 1999

Vertical filtering of progressive MPEG data with down-scaling is wrong

Level	2
Description	When vertically scaling progressive data for interlaced output, and you are scaling down, e.g. letterbox mode, the hardware chooses the wrong lines and wrong filter coefficients.
Work-round	1. Telling the filter the source is interlaced helps a little 2. Reprogramming the filters and DMA on a line-by-line basis.
Date added	July 6, 1999

Video Output can be shifted right by 16 pixels

Level	0
Description	Video can sometimes be starved for data, resulting in a 16-pixel right shift. It is not a real bug, in that it something that is behaving as designed but in pathological cases can result in very long delays for high priority dma data. It is a consequence of an attempt to optimize accesses to memory by prioritizing first requests that use the same RAS and then requests that use the other bank of memory to enable hidden pre-charging.
Work-round	The best fix for video is to issue video DMAs that are 1 line longer than required and then flush the FIFOs and drain the main channel (as we do now). If we don't issue longer DMAs there could be visible bad pixels in the bottom right portion of the image (16 pixels max). All this also applies for overlays.
Date added	July 6, 1999

Communication Bus

Receive buffer full flag is set too soon

Level	0
Description	The receive buffer full flag is set as soon as the first long word of data is received. This means that the complete receive buffer register is not valid for a further three clock cycles.
Work-round	Care should be taken when optimizing code to ensure that you don’t read the data until at least three clock cycles after you have ascertained that there is data to be read. Interrupt code is not affected as you can't get into an interrupt service routine that fast.

Date added	Unchanged from the MMP-L3A

The Comm Bus arbiter can stream I2C packets and drop CDI data

Level	2
Description	There is an error in the Comm Bus arbiter that means that if the CDI requests the Comm Bus while the I2C owns the Comm Bus, the bus is wrongly granted to the I2C instead of the CDI. This has the effect that the I2C will get stuck in a loop streaming Comm Bus packets corresponding to the read command until somebody else requests the Comm Bus. While stuck in this loop the CDI cannot transmit data so CDI data may be lost.
Work-round	There are two possibilities: 1. Immediately follow all I2C reads with an I2C write, with hardware retries enabled. This guarantees that the write will complete after the read, and will pull the arbiter out of the loop. At least one spurious extra read packet may still be transmitted. 2. Detect the error condition arising (un-requested I2C packet) and immediately send any Comm Bus packet, e.g. send yourself one.
Date added	June 3, 1999

Other Bus

The Debug Controller is unable to do remote transfers on the Other Bus

Level	1
Description	If the debug controller does an Other Bus transfer where the debugDmaData registers are neither the source nor the destination of the transfer, then the Other Bus Transfer Pending flag is never cleared.
Work-round	Use an MPE to perform the transfer.
Date added	January 21, 1999

GENERAL IO

General IO register reads may conflict with Serial Device Bus input data

Level	1
Description	If a register within the General IO section is read while serial device is being sent over the Comm Bus, then packets can get lost, and data can be corrupted. The effect is that if the response to a register read occurs at the same time as a Serial Device Bus read packet a collision occurs and only one packet is sent. This packet is a hybrid of the two in that: - it contains the read data of the register read, including address and status - it is sent to the target of the Serial Device Bus read
Work-round	Do not read any General IO registers while Serial Device Bus reads are active. If you cannot avoid doing this, then make sure that only one MPE is responsible for both the register reads and the Serial Device Bus data. Then you only need to write your code so that lost packets from the Serial Device Bus can be handled. You will need to do that anyway.
Date added	October 16, 1998

The debug controller system reset and watchdog functions do not reset the MPEs

Level	1
Description	The System Reset flag, and watchdog function, in the debug controller module do not reset the MPEs. They only reset the rest of the system. This makes them essentially useless.

Work-round	System resets can be performed externally, e.g. by hooking a GPIO pin into the reset circuit. The watchdog function could also be implemented externally.
Date added	February 9, 1999

The debug controller cannot do two long-word DMA transfers on the Other Bus

Level	1
Description	Other Bus read transfers of length two do not work for the debug controller. The second long-word of data is incorrect. Length two transfers work fine for Other Bus writes, and for all Main Bus transfers.
Work-round	Other Bus read transfers should be restricted to one long-word.
Date added	March 3, 1999

The I2C slave cannot flag that it is empty to an external master

Level	1
Description	If the I2C slave is empty, i.e. slaveTxCount is four, a read address will still be acknowledged, but the master will read $FF in all byte positions. Similarly, for writes, the write address will be acknowledged, but in this case the write bytes will not be acknowledged.
Work-round	When the transmit buffer empties, you could change the slave address so that future transfers are not acknowledged.
Date added	April 7, 1999

System Bus Interface

The read addresses for External Host Interrupt Control and Status are swapped

Level	1
Description	The read addresses for the External Host Interrupt Control register and the External Host Interrupt Status register are reversed.
Work-round	Write to the External Host Interrupt Control register at offset $14 and read it at offset $18. Write to the External Host Interrupt Status register at offset $18 and read it at offset $14.
Date added	November 18, 1998

The host reset function does not work

Level	1
Description	The reset bit in bit 1 of the hostIntReq register in the external host register space has no effect.
Work-round	Use external reset hardware, or the reset function in the debug controller if available.
Date added	December 4, 1998

Data transfers can be corrupted in the chip select address spaces

Level	1
Description	During ODMA writes to SRAM/ROM space on the systembus, if the Other Bus controller holds up the transfer for >= cs_length, wrong data will be written to the slave device. The logic will also return wrong data during ODMA reads from SRAM/ROM space if the controller holds off the transfers.

	This is not a problem for single transfer dma cycles (because there is no Hold off).
Work-round	Two options: 1. Program the CS length to at least 5 clock cycles. A hold off longer than that can not occur in MPE / System Bus transfers. 2. Force single long word DMA transfers. This is not really practicable.
Date added	April 14, 1999

Audio Interface

The dataDelay flag for audio out delays by the wrong clock

Level	1 – hardware only
Description	If the dataDelay bit is set in the audio output section, audio output data is delayed by one audio_clk cycle and not by one X_sbclk cycle as it should be (see the clock section in the audio output description). This makes it only useful if bitScale is set to divide by one.
Work-round	Do not use DACs that require this function – all the DACs we have used so far will work fine.
Date added	September 29, 1998

Audio In is turned on at reset

Level	0
Description	As it comes out of reset, the audio in channel may send one packet to MPE 0 shortly after reset.
Work-round	Boot ROMs created after 28 Jan 98 contain appropriate code to fix this. The code that fixes Oz will also work for Aries
Date added	Carried over from the MMP-L3A. Only sends one packet in the MMP-L3B.

Audio register reads may conflict with audio input data

Level	1
Description	If a register within the audio section is read while audio input is being streamed over the Communication Bus, then packets can get lost, and data can be corrupted.
Work-round	Do not read any audio registers while audio input is active.
Date added	October 16, 1998

Audio input clock polarity is not programmable in master mode

Level	1
Description	When the audio input is acting as a timing master, with its bit and word clocks as outputs, the bit clock polarity is always rising edge for capture.
Work-round	Operate the interface in slave mode. It will usually make sense to use the audio output channel bit and word clocks, with both the ADC and the MMP-L3B audio inputs acting as slaves.
Date added	November 2, 1998

Top and Error flags from audio input channel 2 to the CDI are swapped

Level	1
Description	The Top and Error flags captured on audio input channel 2 and passed to the CDI are reversed. This means the signal captured on X_casdata is passed to the CDI as top, and the signal captured on X_careq is passed to the CDI as error. This affects systems capturing data on audio input channel 2, and also capturing errors or framing flags on this channel, when the data is passed through to the CDI.
Work-round	These signals should be swapped externally, and if the data is also sometimes received directly from the audio input channel, then the software should be changed to reverse the convention in the documentation, i.e. to: 6-7 right flag (or error) data 8-9 right sync (or top) data 10-11 left flag (or error) data 12-13 left sync (or top) data
Date added	October 11, 1999

BDU

Last Block Element Truncation "White Dots Bug"

Level	2
Description	This bug occurs when the last element of a block is negative, but small enough that it will be truncated to zero by the inverse quantization, and even parity. In this case the pipeline propagates the value as a 'negative zero' which is taken by the mismatch control as a negative even value as opposed to a positive even value, i.e. it subtracts one instead of adding one to the last block element, which totally messes up the idct.
Work-round	None
Date added	July 6, 1999

VDG

These first two bugs were fixed before production started in Aries 1.1, and are included here only for reference.

Video clock relationship to video data is not defined – FIXED IN ARIES 1.1

Level	*1 – hardware*
Description	*The video output can start up in one of two phases relative to the video output clock, either correctly, or where data and clock rising edge coincide.*
Work-round	*A small delay on video clock (ideally one quarter-cycle) will provide enough setup and hold to work with the current video encoder.*
Date added	*December 1, 1998*

Sub-picture not work at some horizontal alignments – FIXED IN ARIES 1.1

Level	*3*
Description	*The "startsub" signal, which is fixed to be 100 ticks before "SubHstart", stops incrementing "pix_pos". "pix_pos" will then stop incrementing and hold its last value until the next "left edge". Normally this will be OK if the PXD window is aligned to the left of the*

	screen. But if we move PXD to the right, eventually "startsub" will occur after "left_edge", which will stop the pix_pos from counting and hence mess up the change color commands.
Work-round	*None*
Date added	*December 1, 1998*