VICE Design Specification 099-0123-003

Salvatore Arcuri
David Barnett
Michael Fuccio
Bent Hagemark
Steve Klinger
Te-Li Lau
Henry Moreton
Robb Pryor
Doug Quist
Mike Travis
Mark Troeller
Chuck Tuffli

Desktop Systems Division

Version1.0

Preface

The ideas for building a general purpose block of hardware that can be applied as a Unix Workstation resource are not new. That is in fact what a RISC core is, what a Floating Point Unit is and what various I/O blocks are. We wanted to build something that could be used for image processing and that could be shared across multiple applications. Rather than build a hard coded block that was dedicated to a single task such as texture generation or data compression or image processing, we wanted it all.

It is most interesting to see applications that manipulate video and images on the workstation rather than just playing this information through the workstation to emulate a television or a photograph. Bringing an additional billion instructions per second of general purpose pixel manipulation seems like a useful resource in the workstation to apply across a wide range of real world applications. Feeding the incredible high bandwidth of the human eye, creates an insatiable demand for processing power. We hope this creation will serve up some tasty visual meals.

1.0 Introduction....................15
1.1 General Overview....................15
1.2 System Overview....................15
1.3 Features....................17
1.4 VICE Overview....................19
1.4.1 Media Signal Processor....................19
1.4.2 Bitstream Processor....................19
1.4.3 DMA Unit....................19
1.4.4 Host Interface....................20
2.0 Programmer's Interface....................22
2.1 Address Map....................22
2.2 Vice_ID decoding....................22
2.2.1 VICE address protection....................25
2.2.2 VICE address convenience....................25
2.3 Chip Initialization....................26
2.3.1 Unix Processor - VICE reset interaction....................26
2.3.2 Internal VICE initialization....................26
2.3.3 Selective VICE initialization....................26
2.4 Interrupts/Exceptions....................27
2.4.1 MSP Exception Processing....................27
2.4.2 Exception Priority....................29
2.4.3 Handling Multiple Exception....................29
2.5 MSP Code Management....................30
2.5.1 Basic MSP Operation....................30
2.5.2 Code Segment Updates....................30
2.5.3 Debug Operations....................30
2.6 BSP Code Management....................32
2.6.1 Initialization....................32
2.6.2 Code Segment Updates....................32
2.6.3 Debug Operations....................32
2.7 DMA Management....................33
2.7.1 DMA Programming Restrictions....................33
2.7.2 TLB and Address Space....................33
2.7.3 64K Linear Tiles....................35
2.7.4 64K Frame Buffer Tiles....................36
2.7.5 Hits and Misses....................37
2.7.6 TLB Entry Format....................38
2.8 Register Address Map Summary....................39
2.8.1 DMA descriptor registers....................42
2.9 Register Description....................44
2.9.1 VICE_ID - Chip ID and Revision Register Format....................44
2.9.2 VICE_CFG - General Configuration Register....................44
2.9.3 VICE_INT_RESET - Interrupt Reset Register....................45
2.9.4 VICE_INT - Interrupt Status Register....................46
2.9.5 VICE_INT_EN - Interrupt Enable Register....................47
2.9.6 BSP_SW_INT - BSP Software Interrupt Register....................47
2.9.7 MSP_SW_INT - MSP Software Interrupt Register....................47
2.9.8 MSP_D_RAM - Data RAM Arbitration Register....................48
2.9.9 MSP_CTL_STAT - Media Signal Processor Control Register....................49
2.9.10 MSP_PC Media Signal Processor Program Counter Register....................50
2.9.11 MSP_BadAddr Register....................50

2.9.12 MSP_WatchPoint Register 50
2.9.13 MSP_EPC Registers 50
2.9.14 MSP_CAUSE Register 51
2.9.15 MSP_ExcpFlag Registers 52
2.9.16 VICEMSP_COUNT - MSP Free Running Counter 53
2.9.17 BSP_CTL_STAT - Bit Stream Processor Control and Status Register 54
2.9.18 BSP_WatchPoint Register 54
2.9.19 BSP_IN_COUNT - BSP Input bits counter 55
2.9.20 BSP_OUT_COUNT - BSP Output bits counter 55
2.9.21 BSP_IN_BOX - Bit Stream Processor Input Mailbox 56
2.9.22 BSP_OUT_BOX - Bit Stream Processor Output Mailbox 57
2.9.23 HST_BSP_IN_BOX - Host Snoop of BSP Input Mailbox 58
2.9.24 HST_BSP_OUT_BOX - Host Snoop of BSP Output Mailbox 58
2.9.25 BSP_PC Bitstream Processor Program Counter Register 59
2.9.26 BSP_EPC Bitstream Processor Exception Program Counter 59
2.9.27 BSP_HALT_RESET - Bit Stream Processor Halt and Reset Register 60
2.9.28 BSP_CAUSE Register 61
2.9.29 BSP_FIFO_CTL_STAT - Bit Stream Processor Fifo Control and Status Register. 62
2.9.30 BSP_AVALID_BITS - Bit Stream Processor A Fifo Valid Bits Register 63
2.9.31 BSP_FVALID_BITS - Bit Stream Processor F Fifo Valid Bits Register 63
2.9.32 DMA_CTL_CH1 - DMA Control Register 64
2.9.33 DMA_STAT_CH1 - VICE DMA Status Register 65
2.9.34 DMA_DATA_CH1 - VICE DMA Data Fill Register 66
2.9.35 DMA_MEM_PT_CH1 - DMA System Memory Pointer 66
2.9.36 DMA_VICE_PT_CH1 - DMA Internal Vice Memory Pointer 66
2.9.37 DMA_COUNT_CH1 - DMA Internal Vice DMA Counter 67
2.9.38 DMA_CTL_CHX_DY - DMA Descriptor Control Entry 68
2.9.39 DMA_SMEM_HI_CHX_DY - System Memory Upper Address Pointer 71
2.9.40 DMA_SMEM_LO_CHX_DY - System Memory Lower Address Pointer 71
2.9.41 DMA_WIDTH_CHX_DY - DMA Descriptor Width 71
2.9.42 DMA_STRIDE_CHX_DY - DMA Descriptor Stride 71
2.9.43 DMA_LINES_CHX_DY - DMA Descriptor Lines 72
2.9.44 DMA_VMEM_Y_CHX_DY - Vice Address Y 72
2.9.45 DMA_VMEM_C_CHX_DY - Vice Address C 72

3.0 System Interface 73

3.1 VICE <-> CRIME SysAD Protocol 73
3.1.1 Physical Signals 73
3.1.2 Address 74
3.1.3 Bytes, Words, Cycles 75
3.1.4 SysCMD Extensions 77
3.1.5 Data Identifiers 80
3.2 Unix Processor read/write of VICE 83
3.3 VICE read/write of System Memory 84
3.3.1 VICE DMA Read 84
3.3.2 VICE DMA Write 86
3.4 VICE SysAD Protocol Timing Diagrams 89
3.5 Clock Interface 92
3.6 Error Checking 94

4.0 Architectural Description 95

4.1 Host Interface 97
4.1.1 Host Access 97

4.1.2 DMA 97
4.1.3 Host/DMA interaction 98
4.2 Arbiter/Internal Bus Sharing 99
4.2.1 Rules for Access to Internal VICE buses 100
4.2.2 Common Bus Arbiter Protocol 104
4.2.3 DMA Bus Arbiter Protocol 106
4.3 DMA 109
4.3.1 DMA Descriptors 110
4.3.2 DMA Registers 112
4.4 Media Signal Processor Overview 113
4.4.1 Instruction Fetch Mechanism 115
4.4.2 Common Bus Interface 118
4.4.3 Shared Memory 119
4.5 MSP Scalar Unit 120
4.5.1 Scalar unit instructions format 122
4.5.2 Scalar Unit Instruction Set 122
4.5.3 Instructions not supported 124
4.5.4 Pipeline 124
4.5.5 Scalar unit operation 129
4.5.6 Scalar Unit Blocks 131
4.5.7 Registers 132
4.5.8 Load Store Mechanism 132
4.6 MSP Vector Unit 143
4.6.1 Functional Overview 143
4.6.2 VU Features 143
4.6.3 VU Programming Model 144
4.6.4 Binary Fixed-Point Format 144
4.6.5 Instruction Set Overview 144
4.6.6 VU Instruction Pipeline 165
4.7 Bitstream Processor 172
4.7.1 Bitstream Processor Programming Model 176
4.7.2 VLC Decode Table Structure 185
4.7.3 VLC Encode Table Structure 188
4.7.4 Programming Restrictions. 189
4.7.5 Performance of Bitstream Processor 191
4.7.6 Bitstream Processor Hardware Architecture 192
4.7.7 Bitstream Processor / Scalar Unit Synchronization 196
5.0 Operational Description 198
6.0 Performance Analysis 199
6.0.1 Peak Hardware Performance 199
6.1 Baseline JPEG Decode Application 199
6.1.1 Baseline JPEG decode Data Flow 200
6.1.2 Baseline JPEG decode registration Diagram 201
6.2 Baseline JPEG Encode (lossy) Application 201
6.2.1 Baseline JPEG Encode MCU data flow: 202
6.2.2 Baseline JPEG Encode registration diagram: 203
6.3 Lossless JPEG Application 203
6.4 MPEG-2 Decode Application 203
6.4.1 MPEG-2 Decode Application Data Flow: 204
6.4.2 MPEG-2 Decode Application registration Diagram 205
6.5 H.261 Application 205

6.6 Image Vision Library Primitives 205
7.0 Precision Analysis 206
7.1 Scalar Unit 206
7.2 Vector Unit 207
8.0 Device Interface 208
8.1 Signal Descriptions 208
8.2 Pin Assignments 213
8.3 Test Modes 231
8.4 Schematic Icon 231
8.5 Physical Packaging Diagram 232
8.6 Physical Package Markings 233
8.7 Bonding Diagram 233
9.0 Device Characteristics 234
9.1 Absolute Maximum Ratings 234
9.2 Operating Range 234
9.3 DC Characteristics and Capacitance 235
9.4 AC Characteristics 235
9.4.1 PLL Characteristics 235
9.5 Package Thermal Characteristics 236
10.0 Bugs 237
10.1 Software Simulator vs. Silicon Behavior 237
10.1.1 MSP_D_EN Register 237
10.1.2 MSP_CAUSE Register 237
10.2 Silicon Bugs 238
10.2.1 Leading Zero Bug 238
10.2.2 Rocky bad block 239
10.2.3 Low Quant - Low Compression 239
10.2.4 Skier Sparkle 240
10.2.5 Decode 240
10.2.6 MPEG hang 241
10.2.7 VSUM2 241
10.2.8 BSP Halt Ack 242
10.2.9 MSSM Reset 242
10.2.10 MSP PC Pins 243
10.2.11 Vice-Crime Handshake Pins 244
11.0 Revision History 245
Appendix A: 250
Vector Unit Instruction Set Details 250
Appendix B: 392
Vector Unit Block Diagrams 392
Appendix C 413
Bitstream Processor Instruction Set Details: 413
Appendix D: 431
Test Plan 431
Bibliography 468

FIGURE 1. VICE applied to Silicon Graphics entry workstation 16
FIGURE 2. Logical pin diagram of VICE 18
FIGURE 3. VICE Functional Block Diagram...21
FIGURE 4. SMEM Pointer decomposition for 64K Linear Page 35
FIGURE 5. SMEM Pointer decomposition for 64K Linear Page 36
FIGURE 6. 4Meg System Memory using 64K tiles 37
FIGURE 7. Moosehead SysAD Bus processor connections 76
FIGURE 8. System Interface Command Syntax Bit Definition...................................77
FIGURE 9. Read Request SysCmd Bus Bit Definition 77
FIGURE 10. Write Request SysCmd Bus Bit Definition..79
FIGURE 11. Data Identifier SysCmd Bus Bit Definition 80
FIGURE 12. MPEG-2 Field Predictor DMA Read - System Memory to MSP Data RAM85
FIGURE 13. MPEG-2 Frame Picture DMA Write - VICE Data RAM to System Memory87
FIGURE 14. MPEG-2 Field Picture DMA Write - MSP Data RAM to System Memory 88
FIGURE 15. Unix Processor Write to VICE Address Space..89
FIGURE 16. Unix Processor Read from VICE Address Space 89
FIGURE 17. VICE Bus Request 90
FIGURE 18. VICE Bus Release..90
FIGURE 19. VICE DMA read request to CRIME, VICE already owns SysAD bus 91
FIGURE 20. VICE-CRIME DMA Read Response, CRIME already owns SysAD bus 91
FIGURE 21. VICE - CRIME DMA Block Write 91
FIGURE 22. SysAD Clock Distribution 93
FIGURE 23. VICE block diagram 96
FIGURE 24. Internal Address/Control Flow 102
FIGURE 25. Host/DMA Block Diagram 103
FIGURE 26. MSP (Scalar Unit) Access on Common Bus 105
FIGURE 27. Bit Stream Processor Access on Common Bus 106
FIGURE 28. Bit Stream Processor Access on DMA Bus 108
FIGURE 29. Host/DMA Access on DMA Bus...109
FIGURE 30. DMA Descriptor Format..111
FIGURE 31. Media Signal Processor Block Diagram 114
FIGURE 32. Instruction RAM and Instruction Fetch Control 116
FIGURE 33. Fetching 2 Instructions from the Instruction Ram 116
FIGURE 34. Scalar Unit Instruction Format..122
FIGURE 35. Su Instruction Pipeline 125
FIGURE 36. Visualization of the various pipeline stages 125
FIGURE 37. Illustration of Branch Taken 126
FIGURE 38. Illustration of Branch Not Taken 126

FIGURE 39. Illustration of wrap-around access ...133
FIGURE 40. Big Endian Mode ...133
FIGURE 41. Bitstream Processor in the context of the VICE chip ...173
FIGURE 42. Bitstream Processor and Memory ...174
FIGURE 43. BSP Internal Block Diagram ...175
FIGURE 44. BSP Instruction Pipeline ...176
FIGURE 45. BSP Instruction Pipeline - Multi-cycle ...177
FIGURE 46. BSP Jump/Branch Instruction ...177
FIGURE 47. BSP Status and Control Register ...178
FIGURE 48. Generic Table Entry Format ...186
FIGURE 49. Run-Level Table Descriptor Format ...187
FIGURE 50. Run-Length Table Descriptor Format ...188
FIGURE 51. Bitstream Buffer ...194
FIGURE 52. BSP Code Search Block Diagram ...195
FIGURE 53. Logical pin diagram of VICE ...208
FIGURE 54. 380 Lead Tab Ball Grid Array ...232
FIGURE 55. Package Markings ...233

TABLE 1. VICE_ID Address Response....23
TABLE 2. VICE Address Map....24
TABLE 3. Exception Priority Order....29
TABLE 4. VICE DMA view of System Memory....35
TABLE 5. MSP DMA TLB Entry Format....38
TABLE 6. Register Address Map Summary....40
TABLE 7. DMA Descriptor Address Map....42
TABLE 8. VICE_ID Register Format....44
TABLE 9. VICE_CFG Register Format....44
TABLE 10. VICE_INT_RESET Register Format....45
TABLE 11. VICE_INT Register Format....46
TABLE 12. VICE_INT_EN Register Format....47
TABLE 13. MSP_D_RAM Register Format....48
TABLE 14. MSP_CTL_STAT Register Format....49
TABLE 15. MSP_PC Register Format....50
TABLE 16. MSP_BadAddr Register Format....50
TABLE 17. MSP_WatchPoint Register Format....50
TABLE 18. MSP_EPC Register Format....51
TABLE 19. MSP_CAUSE Register Format....51
TABLE 20. Exception Code Field of Cause Register....51
TABLE 21. MSP_ExcpFlag Register Format....52
TABLE 22. MSP_COUNT Register Format....53
TABLE 23. BSP_WatchPoint Register Format....54
TABLE 24. BSP_IN_COUNT Register Format....55
TABLE 25. BSP_OUT_COUNT Register Format....55
TABLE 26. BSP_IN_BOX Register Format....56
TABLE 27. BSP_OUT_BOX Register Format....57
TABLE 28. HST_BSP_IN_BOX Register Format....58
TABLE 29. HST_BSP_OUT_BOX Register Format....58
TABLE 30. BSP_PC Register Format....59
TABLE 31. BSP_EPC Register Format....59
TABLE 32. BSP_HALT_RESET Register Format....60
TABLE 33. BSP_CAUSE Register Format....61
TABLE 34. BSP_FIFO_CTL_STAT Register Format....62
TABLE 35. BSP_AVAILD_BITS Register Format....63
TABLE 36. BSP_FVALID_BITS Register Format....63
TABLE 37. DMA_CTL_CH1 Register Format....64
TABLE 38. DMA_STAT_CH1 Register Format....65
TABLE 39. DMA_DATA_CH1 Register Format....66
TABLE 40. DMA_MEM_PT_CH1 Register Format....66
TABLE 41. DMA_VICE_PT_CH1 Register Format....66

TABLE 42. DMA_COUNT_CH1 Register Format 67
TABLE 43. DMA_CTL_CHX_DY Register Format...68
TABLE 44. DMA_SMEM_HI_CHX_DY Register Format 71
TABLE 45. DMA_SMEM_LO_CHX_DY Register Format 71
TABLE 46. DMA_WIDTH_CHX_DY Register Format 71
TABLE 47. DMA_STRIDE_CHX_DY Register Format 72
TABLE 48. DMA_LINES_CHX_DY Register Format 72
TABLE 49. DMA_VMEM_Y_CHX_DY Register Format 72
TABLE 50. DMA_VMEM_C_CHX_DY Register Format 72
TABLE 51. VICE <-> CRIME unique connections..74
TABLE 52. Data Size Name Convention 75
TABLE 53. Encoding of SysCmd(7:5) for System Interface Commands.....................77
TABLE 54. Encoding of SysCmd(4:3) for Read Requests 78
TABLE 55. Encoding of SysCmd(2:0) for Block Read Requests.................................78
TABLE 56. Doubleword, Word, or Partial-word Read Request Data Size Encoding of SysCmd(2:0)78
TABLE 57. Encoding of SysCmd(4:3) for Write Requests 79
TABLE 58. Encoding of SysCmd(2:0) for Block Write Requests................................79
TABLE 59. Doubleword, Word, or Partial-word Write Request Data Size Encoding of SysCmd(2:0)80
TABLE 60. Unix Processor Data Identifier Encoding of SysCmd(7:3)........................81
TABLE 61. Vice generated External Data Identifier Encoding of SysCmd(7:3)..........81
TABLE 62. VICE as Processor Data Identifier Encoding of SysCmd(7:3) 82
TABLE 63. CRIME generated External Data Identifier Encoding of SysCmd(7:3).....82
TABLE 64. Unix Processor PClock - SClock Relationship..92
TABLE 65. 94
TABLE 66. 94
TABLE 67. VICE Datapath Flow..101
TABLE 68. List of Signals on Common Bus 104
TABLE 69. List of Signals on DMA Bus...107
TABLE 70. Dependency check for 2 cycle load delay slots 127
TABLE 71. Memory byte marks 135
TABLE 72. Element Selection for Computational Instructions 151
TABLE 73. VICE Pin Descriptions..209
TABLE 74. 380 BGA Pin Assignments - Pin Order 213
TABLE 75. Signal Name - Pin Assignment...217
TABLE 76. 231
TABLE 77. Absolute Maximum Ratings - Non Operational 234
TABLE 78. Device Operating Range 234
TABLE 79. DC Characteristics ($T_j = 0$ to 110° C) 235
TABLE 80. AC Characteristics 235

TABLE 81. Package Thermal Resistance Characteristics - Air Flow Consideration..236
TABLE 82. HD diag list....432
TABLE 83. BSP diag list....436
TABLE 84. MSP diag list....447
TABLE 85. Acceptance test....466

If a picture is worth a thousand words, imagine what 30 pictures every second is worth.

.... Unknown 1994

1.0 Introduction

This document itemizes requirements and design implementation for hardware and embedded software support of image processing in the next generation low end Silicon Graphics Workstation. Image processing is defined as any function(s) that apply to two dimensional blocks of pixels. These pixels may be in the format of file system images, fields or frames of video entering the workstation through video ports, mass storage devices such as CD-ROMs, fixed-disk subsystems and Local or Wide Area network ports. This hardware will be able to convert between various image protocols, colorspace and different signal domains such as frequency and time. The demanding performance requirements for these kinds of operations presently exceed the capabilities of our general purpose microprocessors that run the Unix Operating system and its associated application programs. The intent of this hardware will be to allow images to course through the workstation without requiring significant processing resources from the central CPU. The goal is to provide this additional hardware support at a minimum cost that is leveraged over existing and new image processing tasks so as to make its price increase to the base system cost effective AND justifiable.

1.1 General Overview

This document describes a digital ASIC that will reside alongside the Unix Operating System processor and perform image processing tasks. The ASIC is called VICE for Video, Imaging and Compression Engine. It consists of several blocks to achieve the feature set listed below. The major blocks will be DMA engines to move data from and to system RAM, an implementation of the Silicon Graphics Media Signal Processor architecture (MSP) that can perform integer, logical, and mathematical operations necessary for signal processing and Data and Instruction Memory for the MSP to operate upon. A Bitstream Processor complements the MSP architecture and is optimized to perform variable bit length processing common in compression and de-compression algorithms.

1.2 System Overview

VICE is one of the major chips of Moosehead. Moosehead contains a large bandwidth bus, through which devices communicate with main memory. The main memory acts as a system memory, frame buffer, Z buffer and texture memory. Graphics primitives are rendered by the CRIME into main memory.Video and audio streams travel through the I/O chip to and from main memory. Image compression and expansion, and image processing are implemented in VICE.

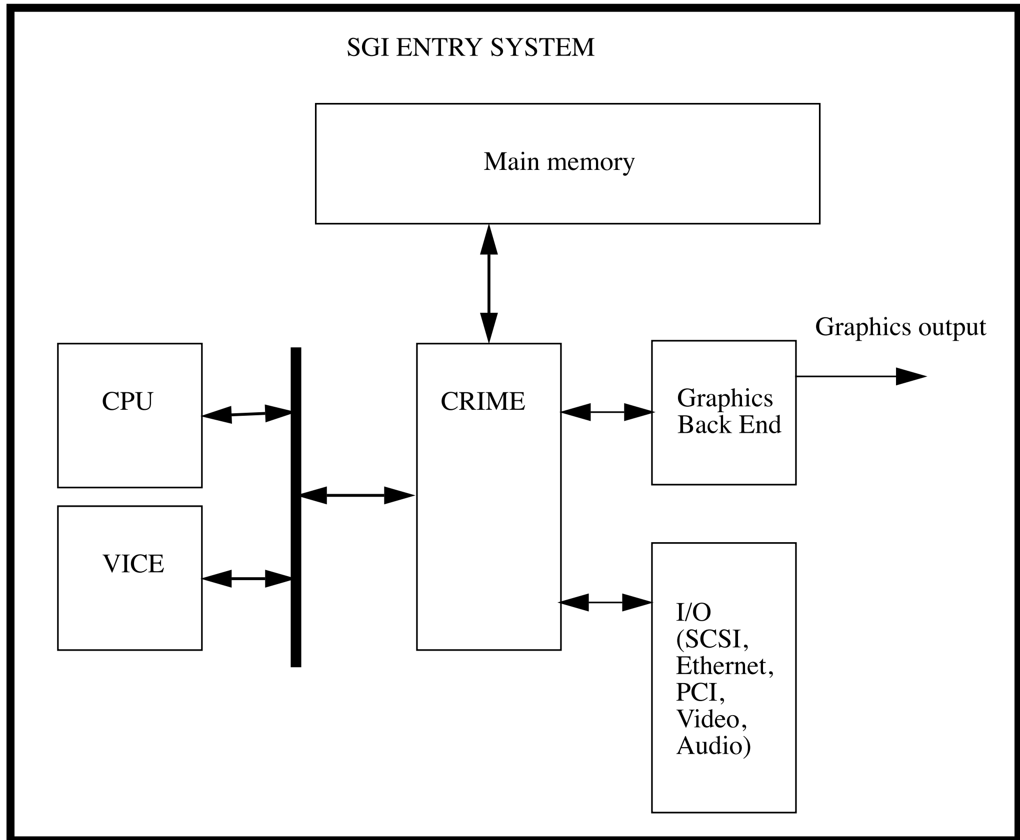


FIGURE 1. VICE applied to Silicon Graphics entry workstation

1.3 Features

Silicon Graphics Media Signal Processor Architecture

- 6K Byte Data RAM
- 4K Byte Instruction RAM
- 32 bit Scalar Unit running at 66 MHz, 1 instruction per clock
- 128 bit SIMD Vector Unit running at 66 MHz, 1 Multiply and 1 Accumulate per clock
- Dual Issue Instruction Dispatch Unit

Bit Processor to accelerate compression standard variable length bit formats

- 16 Bit RISC Core
- Multi-Cycle Instruction Extensions
- Programmable Tables for Multi-Standard Support (JPEG, MPEG, Px64)

Intelligent Direct Memory Access Controller

- Flexible address generation to use Unix System Memory and avoid costly dedicated RAM
- "Virtual" address support for Media Signal Processor access to Unix System Memory
- Special Features to manipulate pixels during DMA (Y/C split & 1/2 Pixel Calculation)

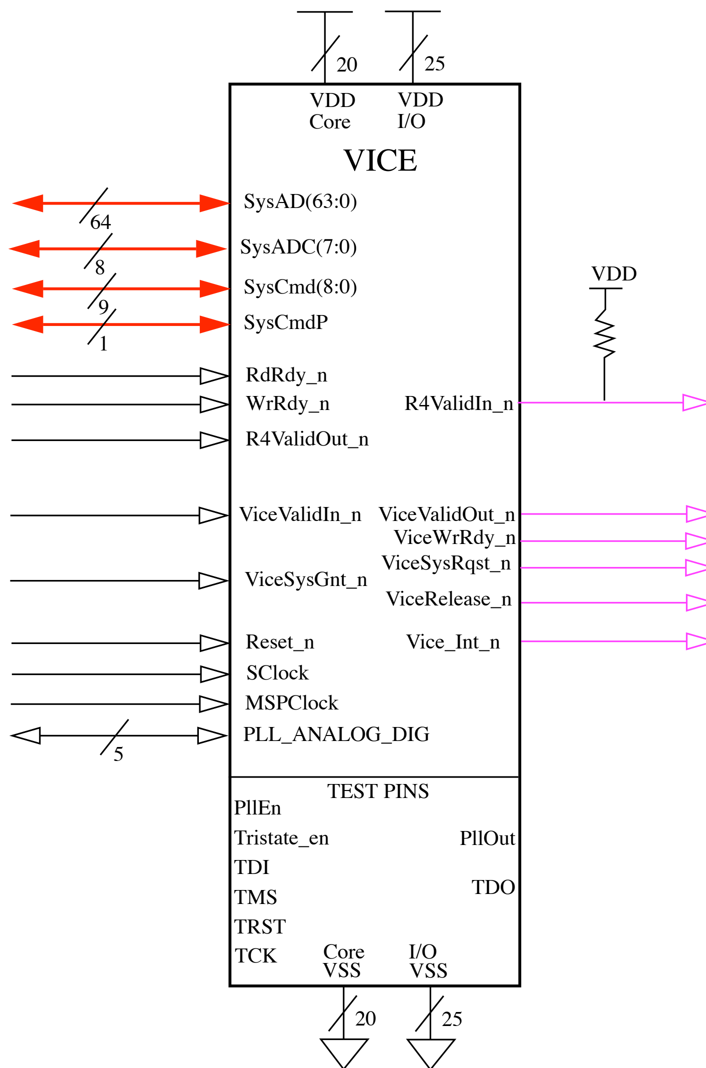


FIGURE 2. Logical pin diagram of VICE

1.4 VICE Overview

VICE contains four major functional blocks:

- Media signal processor.
- Bit stream processor
- DMA unit
- Host Interface

These processing units accelerate industry standard compression and de-compression algorithms. Figure 3, "VICE Functional Block Diagram," on page 21 shows a block level representation of the units within the VICE ASIC.

1.4.1 Media Signal Processor

For more information on the origin of the Media Signal Processor architecture, refer to *MIPS Media Engine Sketch* referenced in the Bibliography.

The implementation of the Media Signal Processor for VICE has been customized in the areas of Memory organization, Pipeline Depth and Interlocks and, when necessary, the actual computational elements in the pipelines have been modified for the target silicon technology chosen for the VICE ASIC.

This Processor is really two processors consisting of one 32 bit instruction unit with 32 bit data path operands (referred to hereafter as the Scalar Unit) and a second 32 bit instruction unit with 128 bit data path operands (referred to hereafter as the Vector Unit). The data path of the Vector Unit can be sliced in 8/16 bit pieces for the purpose of performing integer mathematical operations. No branch instructions are included in the Vector Unit instruction set, as it relies on the Scalar Unit for those functions.

1.4.2 Bitstream Processor

The bitstream processor is a programmable device which is tailored for processing bitstreams of compressed data. The bitstream processor has a 16-bit RISC-like load-store architecture. Hence, it has an instruction set which has familiar register to register operations (such as arithmetic operations), instruction stream control (jumps and branches) and memory to register transfer of data. In addition, the bitstream processor has instructions which are specific to manipulating arbitrarily aligned tokens in a bitstream of data. furthermore, the bitstream processor has instructions which can perform the table lookup operations necessary to decode variable length tokens in a bitstream. The tables provided to these instructions are programmable. They may be programmed to support the MPEG-1, MPEG-2, H.261 and JPEG compression standards, and with restrictions, can be programmed for proprietary algorithms.

1.4.3 DMA Unit

The DMA unit of VICE consists of two DMA channels. Each DMA channel consists of a DMA state machine, control registers and a descriptor memory. The MSP Scalar Unit, Bitstream Processor or the host can program the control registers and descriptor memory of a DMA channel. DMA transactions can occur from Unix System Memory to any memory resource inside the VICE chip. DMA transactions can occur from any memory resource inside of the VICE chip to Unix System Memory. The DMA engine can be used to fill any memory resource inside of the VICE chip using an on-chip data pattern register.

For DMA transaction to/from Unix System Memory, the DMA Unit decomposes the descriptor requests into physical Unix System Memory address and byte count. A Lookup table to convert between the contiguous 4Meg address space of the on-chip VICE processors (MSP & BSP) and the 4Meg physical System Memory

is implemented with 64K page segment entries. The DMA Unit will “batch” these requests to exploit System Memory bandwidth which is maximized when 256 byte transfers are performed.

Because the spans of image data that make up a 256 byte block may not be continuous, the DMA unit must break up the memory requests into address plus byte count so that the CRIME memory controller need not manage row and column “gaps” in the memory requests.

A special mode of the DMA engine will allow data stored as 4:2:2 YCrCb in System Memory to be retrieved and split into separate Y and CrCb blocks of MSP Data RAM. Consideration for further decimating the CrCb channel is covered as well. Similarly, the DMA engine can gather separate Y and CrCb blocks of MSP Data RAM and interleave it when writing the data back into System Memory in 4:2:2 YCrCb format.

Finally, Half-Pixel Calculation is available as part of the DMA transaction. For example, a strip of 17 pixels can be DMA’ed from System Memory resulting in 16 pixels in the VICE Data RAM. Modes for independent Horizontal and Vertical Half-Pel calculation are available. A transaction that interleaves two fields into a frame AND performs half-pel calculation AND separates the target data into Y/C uses most of the features of the DMA engine.

1.4.4 Host Interface

The VICE chip shares the SysAD bus with the Unix processor. Extensions to allow multiple outstanding reads have been added to CRIME and VICE to support efficient transfer of large blocks of data between System Memory and VICE internal data RAM. Arbitration takes place to allow VICE to initiate transactions on the SysAD bus.

The R4K processor family will be able to write/read directly to the VICE chip to a specific address space that is recognized by CRIME and VICE alike to allow efficient transfer of data between VICE and the Unix processor without the overhead of CRIME acting as an intermediary in the data path.

For more information on the SysAD interface refer to Section 3.0 on page 73 of this document and to the *CRIME Design Specification*.

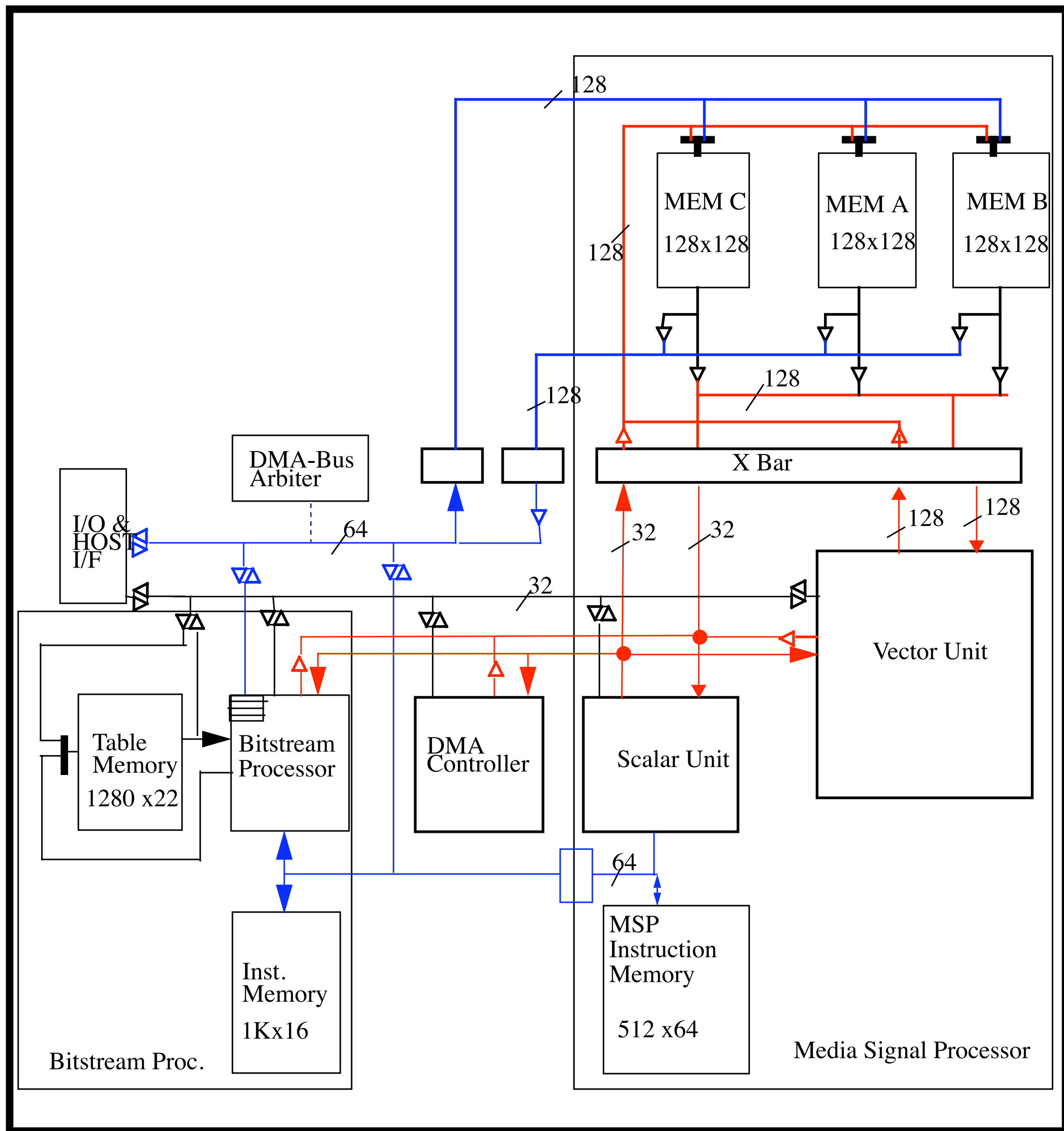


FIGURE 3. VICE Functional Block Diagram

2.0 Programmer's Interface

VICE will communicate with the system in the following ways. All chip Control Registers are mapped to the Unix system address space. All internal RAM is also mapped to the Unix system address space. Only the DMA engines within VICE can access the Unix system address space. Access by the DMA engines will go through a mapping memory that will limit access to a physical 4 MByte region of Unix system memory. This is added as protection to prevent errant code on the VICE chip from corrupting Unix system memory.

The Register Files of the on-chip VICE processors (Media Signal Processor and Bit Stream Processor) are not mapped to Unix system address space.

The VICE device driver will most likely load initialization code into the Instruction RAM of the VICE ASIC and then utilize the MSP_CTL_STAT register, Table 14, "MSP_CTL_STAT Register Format," on page 49 to allow the VICE Media Signal Processor to begin code execution.

Interrupts from the VICE chip are collected within VICE and presented to the system on a signal pin that ultimately finds its way to the Unix system processor.

The VICE Media Signal Processor does not respond to interrupts. It does however halt on exceptions and it can be halted and restarted by using the MSP_CTL_STAT register from the Unix host processor. Additional communication with the Unix System processor is expected to take place in shared system RAM or through internal VICE data RAM. There are no plans to provide register mailboxes within the VICE chip for this inter-processor communication.

The Bitstream Processor and the Media Signal Processor **CAN** communicate with each other through special Mailbox registers.

2.1 Address Map

The chip is mapped to system address space so that the Unix processor can access internal information in the VICE chip. On-Chip registers (excluding BSP and MSP register files) and RAM are also accessible to the on-board Media Signal Processor and Bit Stream Processor. The map below shows the general layout of the VICE chip address space and the address space through which each processor (MSP, BSP and Unix processor) can access that resource.

Unix processor accesses within an address range that exceed the physical range implemented will return 0's in the data field and will NOT return data from a register or memory location. This allows for expansion of physical memory such as Data RAM or Instruction RAM to be implemented in future versions of the chip while remaining obvious and deterministic to the Programmer's Interface.

Note that the MSP does **NOT** have self diagnostic capability for its Instruction RAM and will rely on the System address space ports to verify this memory. The same is true for the Instruction RAM of the BSP. The Address Sequence has been preserved for the System, MSP and BSP address space across the Instruction RAM gaps of each processor, to maintain simplicity of address decode for the entire ASIC.

The MSP and BSP will require diagnostic code to exercise and verify proper operation of their register files.

2.2 Vice_ID decoding

The VICE_ID(1:0) pins on VICE allow for multiple VICE chips to share the SysAD bus. The value of the VICE_ID pins are compared with SysAD Address bits (21:20), to provide the following mapping:

TABLE 1. VICE_ID Address Response

VICE_ID	VICE Address Range
00	0x0 170F FFFC - 0x0 1700 0000
01	0x0 171F FFFC - 0x0 1710 0000
10	0x0 172F FFFC - 0x0 1720 0000
11	0x0 173F FFFC - 0x0 1730 0000

Note that only PIO decoding of the SysAD protocol is covered by use of the VICE_ID pins. The VICE <-> CRIME pins would need to be repeated on CRIME for each VICE implemented on the SysAD bus. So the ViceSysRqst_n pin would be ViceSysRqst0_n and ViceSysRqst1_n on CRIME, ViceSysGnt0_n and ViceSysGnt1_n etc.

The use of multiple VICE chips has NEVER been tested on the SysAD bus. It has not been tested in the simulator either!

TABLE 2. VICE Address Map

System Address	VICE Address	Address Range	Function	Comments	MSP Access	BSP Access
Contained in TLB	0x10BF FFFC 0x1080 0000	4M	Vice Accessible System Memory	64K Frame Buffer Tiles	DMA Engine	DMA Engine
Contained in TLB	0x00BF FFFC 0x0080 0000	4M	Vice Accessible System Memory	64K Linear Page	DMA Engine	DMA Engine
0x0 17FF FFFC 0x0 1701 0000	Not Used		Unused Space		Exception	Exception
0x0 1700 FFFC 0x0 1700 F000	0x0000 FFFC 0x0000 F000	4K	VICE TLB	64 entry map	Exception	Exception
0x0 1700 EFFC 0x0 1700 E000	0x0000 EFFC 0x0000 E000	4K	Kernel Restricted Regs	Protected from debugger/user	Exception	Exception
0x0 1700 DFFC 0x0 1700 C000	0x0000 DFFC 0x0000 C000	8K	Unused	Unused	Exception	Exception
0x0 1700 BFFC 0x0 1700 9800	0x0000 BFFC 0x0000 9800	10K	Data RAM - Unused		Exception	Exception
0x0 1700 97FC 0x0 1700 9000	0x0000 97FC 0x0000 9000	2K	Data RAM - Bank C		Load/ Store	Load/ Store
0x0 1700 8FFC 0x0 1700 8800	0x0000 8FFC 0x0000 8800	2K	Data RAM - Bank B		Load/ Store	Load/ Store
0x0 1700 87FC 0x0 1700 8000	0x0000 87FC 0x0000 8000	2K	Data RAM - Bank A		Load/ Store	Load/ Store
0x0 1700 7FFC 0x0 1700 7000	0x0000 7FFC 0x0000 7000	4K	Bitstream Processor Input/Output buffers	IN 0x7800 OUT 0x7000	Exception	Load/ Store?
0x0 1700 6FFC 0x0 1700 5000	0x0000 6FFC 0x0000 5000	8K	Bitstream Processor Table Memory	1280K x 22	None	Load/ Store
0x0 1700 4FFC 0x0 1700 4000	0x0000 4FFC 0x0000 4000	4K	Bitstream Processor Instruction	1K x 16	None	Fetch Only
0x0 1700 3FFC 0x0 1700 3000	0x0000 3FFC 0x0000 3000	4K	MSP Instruction RAM	Reserved Not Implemented	Exception	None
0x0 1700 2FFC 0x0 1700 2000	0x0000 2FFC 0x0000 2000	4K	MSP Instruction RAM		Fetch Only	None
0x0 1700 1FFC 0x0 1700 1000	0x0000 1FFC 0x0000 1000	4K	Chip DMA Descriptor Set Registers	64 r/w locations accessed as COP3	MTC3/ CTC3	Load/ Store
0x0 1700 0FFC 0x0 1700 0008	0x0000 0FFC 0x0000 0008	4K	Chip Registers	Mode, Status/ Control, Interrupts	MTC1/ MFC1	Load/ Store
0x0 1700 0007 0x0 1700 0000	0x0000 0007 0x0000 0000	8Bytes	Reg Address 0 “Safe” Not Used		Safe Watchpoint	Safe Watchpoint

2.2.1 VICE address protection

To aide in Unix System Memory protection, the VICE chip contains a software managed translation buffer known as a TLB. On-chip VICE processors can modify this TLB through use of the DMA engine. A special permission bit controlled by the Unix Processor must be enabled before the DMA engine can modify the TLB. If any address beyond the 4Meg range is programmed into one of the DMA engine registers or if the DMA engine itself computes an address beyond this “logical” address range, the DMA engine will halt and raise the interrupt line to the Unix Processor.

Note also that Load/Store operations from the MSP/BSP cannot access the 4 Meg range of System Memory. This is as much for protection as it is to maintain hardware simplicity so that the VICE TLB can be dedicated to the VICE DMA engines.

The VICE TLB is expected to have a Unix System Memory page size of 64K. Sequential pages in the VICE TLB do not need to be physically contiguous blocks of memory.

2.2.2 VICE address convenience

For hardware simplicity, the MSP and BSP will not be able to perform load/store operations on their Instruction RAM. Only the Unix Processor and the VICE DMA engine will be able to access Instruction RAM address space. Also, because of the varying datapath arrangements, the Instruction RAM and Data RAM will not be swappable.

The Instruction RAM diagnostics will be a combination of host access and VICE assembly language that performs various branches whose targets perform Data RAM modifications that can be monitored by the host.

2.3 Chip Initialization

2.3.1 Unix Processor - VICE reset interaction

Vice has a dedicated pin named ViceReset_n. It is expected that this be driven by a pin on the CRIME chip. The present plan is to connect the MIPS RESET* pin and ViceReset_n together. CRIME will drive this net.

During the reset condition, VICE will tri-state all of its common I/O shared on the SysAD bus with the Unix host processor. This list:

SysAD(63:00)
SysADC(7:0)
SysCmd(8:0)
SysCmdP
R4ValidIn_n

These VICE <-> CRIME handshake pins are 3-state during ViceReset_n = 0. They must be pullued up on the CPU Board so that CRIME will see valid levels during reset:

ViceValidOut_n = 1
Vice SysRqst_n = 1
ViceRelease_n = 1
ViceInt_n = 1

2.3.2 Internal VICE initialization

VICEwill require 100 milliseconds for its PLL to stabilize. On a Power-On Reset, the Unix Processor RESET* pin will be asserted at least this long (along with the VCCOk signal). The present plan is to connect the Unix Processor RESET* pin and ViceReset_n together. CRIME will drive this net.

If the PLL is stable, then VICE will only require 16 clocks? to fully initialize. ViceReset_n must be held asserted for at least 16 clocks for VICE to fully reset.

ViceReset_n will cause all internal state machines and processors inside of VICE to initialize.

2.3.3 Selective VICE initialization

The MSP has a control register that can be used to reset the MSP independently of the rest of the VICE chip. The DMA engine has the same. The BSP has a control register that can be used to reset the BSP independently of the rest of the VICE chip. The host interface in VICE is not resettable in this manner. If the host interface needs to be reset, the ViceReset_n pin must be asserted.

Any individual register initiated reset will require the reset mode to be active for a minimum of 16 clocks before removing the reset condition.

2.4 Interrupts/Exceptions

Interrupts are generated to the Unix system processor from MSP program control, MSP exceptions and DMA complete or error conditions. The cause of any interrupt is indicated by a bit in the VICE_INT register which is cover in the register section.

2.4.1 MSP Exception Processing

This section describes what conditions generate an MSP exception.

On an exception the MSP immediately halts and posts an interrupt to the host CPU. The host CPU examines MSP registers to determine the cause of the exception and must restart the MSP as appropriate. Once the MSP has halted itself it must be restarted by the host CPU -- i.e., an agent external to the MSP.

MSP exceptions are imprecise. That is, when the exception is detected, no further instructions are issued and instructions already in the SU pipe are completed. The state of the MSP when it has halted will be different from that when the exception was detected.

After an exception has been taken, the MSP PC is non-deterministic and must be set to a known value before the Go bit is set.

2.4.1.1 Load Store Address Error Exception (0,1)

Cause. The Address Error Exception occurs when an attempt is made to:

- MSP SU load or store a word that is not aligned on a word boundary
- MSP SU load or store a halfword that is not aligned on a word boundary
- MSP VU load or store a half that is not aligned on a quad, quad+1 boundary
- MSP VU load or store a fourth that is not aligned on a quad, quad+1, quad+2, quad+3 boundary
- MSP VU load or store a transpose/wrap that is not aligned on a quad boundary
- MSP VU load or store an alterante that is not aligned on a quad, quad+2 boundary
- MSP SU/VU load or store access inst ram space
- MSP SU/VU load or store access address space outside valid MSP Address space
- MSP SU/VU load or store access valid address space but exceed physical range implemented

Handling. When this exception occurs, the *BadAddr* register retains the load store address that caused the exception. On loads, undefined data will be returned and written to the destination register in the SU or VU file. On stores, undefined data will be written to Data Memory. The address where this undefined data will be read from or written can be found in Table 6, “Register Address Map Summary,” on page 40. This is the offset address that must be combined with the base address for registers wich can be found in Table 2, “VICE Address Map,” on page 24.

The EPC register points at the instruction that caused the exception. The AdEL or AdES code in the Cause register is set.

2.4.1.2 Breakpoint Exception (2)

Cause. The Breakpoint exception occurs when a BREAK instruction is reached.

Handling. The BP code in the Cause register is set.

The EPC register points at the BREAK instruction.

Programmer's Note.

If the instruction immediately following the BREAK instruction is not a VU instruction, or if the BREAK instruction was the latter instruction in a 2-instruction issue boundary, the Breakpoint exception is precise. i.e. the SU & VU Reg File after this exception has occurred would be the state before the exception is taken.

The BREAK instruction should not be inserted in a branch or jump delay slot.

2.4.1.3 Watchpoint Exception (3)

Cause. The Watchpoint exception occurs when a MSP load or store instruction references the address specified in the MSP Watchpoint Register. In doing the address comparison, the low 3 bits of the address are ignored.

Handling. The WP code in the Cause register is set. The load/store operation that causes this exception completes.

The EPC register points at the load store instruction that caused this exception. To disable Watchpoint exception, clear the MSP_WatchPoint Register to 0x00000000.

2.4.1.4 Scalar Unit Reserved Instruction Exception (4)

Cause. The Reserved Instruction exception occurs when an attempt is made to execute an instruction whose major opcode (bits 31..26) is undefined, or a SPECIAL instruction whose minor opcode (bits 5..0) is undefined.

Handling. The SuRI code in the Cause register is set.

The EPC register points at the reserved instruction.

Programmer's Note. If an undefined instruction falls outside of the above definition for detecting a reserved instruction exception, the hardware will default to the nearest synthesized valid opcode which is non-deterministic.

2.4.1.5 Vector Unit Reserved Instruction Exception (5)

Cause. The Reserved Instruction exception occurs when an attempt is made to execute a VU instruction that is undefined.

Handling. The VuRI code in the Cause register is set.

The EPC register points at the reserved instruction.

2.4.1.6 Contention Exception (6)

Cause. The Contention exception occurs when the MSP SU load or store to a Data RAM bank deselected by the MSP_Config register.

Handling. The CON code in the Cause register is set.

The EPC register points to the load or store instruction that cause this exception

2.4.1.7 Instruction Fetch Address Exception(7)

Cause. The Instruction Fetch Address Exception occurs when an attempt is made to:

- MSP PC access address space outside Instruction RAM
- MSP PC access valid address space but exceed physical range implemented

In this implementation, the PC is only 14 bits wide with bits [31:16] and bits[1:0] grounded to 0. Hence, range checking is only done on bits[15:3].

Handling. *BadAddr* register is NOT updated with the faulting address. The AdEI code in the Cause register is set.

The EPC register points to the instruction that causes this exception.

2.4.2 Exception Priority

The following table indicates which exception will be posted if more than one exception condition arise simultaneously. To observe if multiple exceptions occurred, look at the Exception Flag Register.

TABLE 3. Exception Priority Order

Instruction fetch addr exception
Breakpoint exception
SU Reserved instruction exception
VU Reserved instruction exception
Contention exception
Address error exception (load)
Address error exception (store)
Watchpoint exception

2.4.3 Handling Multiple Exception

If more than one exception occur during the same cycle, the exception which is stored into the EPC and the Cause register is determined by the above Exception Priority. If however, at the next cycle, should an exception be detected, the EPC or the Cause Register is not updated. That is, the EPC, Cause, BadAddr register is only updated when all bits in the Exception Flag Register are cleared to 0.

Note: *Information regarding subsequent exceptions will only be written into the EPC and the Cause Register when all bits in the Exception Flag Register are cleared.*

2.5 MSP Code Management

This section describes how the host manages the MSP.

2.5.1 Basic MSP Operation

Before starting the MSP the host CPU allocates physical system memory for input and/or output data buffers, application-specific constants and scratch area and loads the VICE TLB with mappings to this memory (see DMA Management). This memory must be pinned down and otherwise allocated for VICE operation.

The host loads the MSP instruction RAM with MSP instruction text and loads Data RAM with arguments. The host then sets the MSP_PC and takes the MSP out of halt (see MSP_CTL_STAT in Register Description).

The MSP now runs to completion with no further interaction with the host CPU. The MSP program reads its input data and/or writes its output data in host memory using DMA. The MSP reads in additional program segments using DMA (see Code Segment Update below).

The MSP program completes on its own by executing the break instruction which causes an exception interrupting the host CPU. The MSP may alternately leave completion notification to the BSP and simply spin -- the BSP then halts and interrupts the host CPU which then halts the spinning MSP using the MSP_CTL_STAT register.

2.5.2 Code Segment Updates

The MSP reads in additional program segments using DMA. The MSP programs DMA as it would for any transfer to VICE, indicating MSP Instruction RAM as the destination.

The MSP Instruction RAM is dual ported. DMA can write to the Instruction RAM at the Same time that the MSP is executing from Instruction RAM. MSP Code can intelligently manage overlays to allow program execution to continue while new Instructions are brought in by the DMA engine.

2.5.3 Debug Operations

The MSP is debugged from the host using a debugger that maps in VICE RAMs and registers. This host-resident debugger uses the break instruction and its own MSP code to save and restore scalar and vector registers to implement single stepping, breakpoints and register set and dump.

VICE Data RAM is directly accessible from the host are the MSP and BSP Instruction RAMs and BSP Table RAMs.

2.5.3.1 Break points

To set a breakpoint the MSP debugger replaces an MSP instruction with the break instruction. The debugger continues execution by replacing the original instruction and then takes the MSP out of halt.

2.5.3.2 Register dump

MSP Vector and Scalar Unit registers are not mapped into host address space. To examine or set vector or scalar registers the MSP debugger first halts the MSP. Data RAM and MSP Instruction RAM are then both

copied from VICE and saved on the host. The debugger loads its own code to save or set registers using load and store instructions. The register values are staged through Data RAM. The debugger continues execution of the MSP by replacing the saved Instruction and Data RAM.

Programmer's Note: When the MSP is halted either by reaching a BREAK or by setting the HALT bit in the MSP_CTL_STAT Register, the programmer must write the PC to a known value since the PC is indeterminate whenever the MSP is halted.

2.6 BSP Code Management

2.6.1 Initialization

To have the software (unix processor) reset the BSP, perform the following sequence:

1) Assert bit 0 (write 1) of the BSP_HALT_RESET register. This will reset the write buffer, encode pipe and decoding state machine.

> Assertion of this bit will HALT THE BSP, where the PC is pointing at that instance. NOPs will be pushed through the instruction pile. The HALT bit (bit 1) will be set and the halt will be acked (bit 2).

2) Assert bit 2 of the BSP_FIFO_CTL_STAT register. (write a 1). This resets the input FIFO.

Note: To start the BSP up again requires the following steps:

1) Put a valid value in the BSP's PC (zero is a good choice).
2) Negate bit 2 of the BSP_FIFO_CTL_STAT register. (write a 0)
3) Negate bit 0 (write a 0) of the BSP_HALT_RESET register.

Now the reset is complete and the BSP is simply halted.

To run the BSP, negate bit 1 of the BSP_HALT_RESET register.

2.6.2 Code Segment Updates

2.6.3 Debug Operations

2.7 DMA Management

This section covers the programmers interface to the various DMA subsystems on the VICE chip. There are two DMA channels that either the MSP Scalar Unit, the Bit Stream Processor or the host can access. The MSP takes advantage of the DMA controller being a virtual device to simplify the overhead involved in accessing Unix system memory buffers.

The TLB is used only for DMA transfers. MSP load/store instructions operate only on MSP Data RAM.

MSP initiated DMA transfers may use TLB-mapped addresses. If the bit to allow the DMA engine to bypass the TLB has been enabled by the host processor in the VICE_CFG register (See "VICE_CFG - General Configuration Register" on page 44.) and that function is enabled in the DMA_CTL register for that dma channel. This is a host owned safety mechanism. This restriction implements system security! MSP TLB entries are writable by the host CPU and by the DMA engine itself. [They could be read by the MSP if the address range to the Common Bus is sufficient].

2.7.1 DMA Programming Restrictions

The DMA_FILL mode does not work in Y/C split mode.

For 4:2:2 -> 4:2:0 and 4:2:2 -> 4:2:2 Y/C split mode, and luma only mode and chroma only mode, the width granularity in System Memory must be modulo quad word (16 byte) and the starting address in system memory must be quad word (16 byte) aligned.

The 4:2:2 -> 4:2:0 Y/C split mode, the algorithm that the DMA engine uses is to overwrite the chroma value of the first line transferred with the value of the second line transferred. This works fine for even line reference pictures that have "synthesized" 4:2:2. That is the second line of chroma always matches the first line since it was copied from the first line. For dma transfers that originate on an odd line, the algorithm implemented in the DMA engine is incorrect. This is because the second line of chroma is from the next even line, while the chroma we want is actually from the previous even line.

Note for VICE respin (version -004), the algorithm to decimate chroma should be to always keep the first line of chroma specified by the DMA engine system memory pointer, drop the second line, keep the third, drop the fourth etc. This is the correct algorithm that will work for descriptors that originate on even or odd scan lines of reference pictures.

2.7.2 TLB and Address Space

MSP DMA engine addresses are translated into physical addresses using the TLB. The TLB is a 64 element array of physical (64Kbyte) page numbers mapping a 4Mbyte virtual address space to System RAM.

If a given (64Kbyte) range of the address space is not accessible for any reason, a TLB entry is marked invalid.

The page size is expected to be fixed to 64K bytes and aligned on 64K byte boundaries in system memory.

Both linear and "tiled" 64K pages in system memory are supported. These modes are selected through the address programmed into the DMA_SMEM_HI_CHX_DY entry in the DMA Descriptors as part of DMA setup. The supported ranges for this pointer are enumerated in Figure 4, "VICE DMA view of System Memory," on page 35.

The contents of the TLB are really unaffected by the two types of pages. The bit fields of the DMA_SMEM_HI_CHX_DY register are interpreted differently by the hardware during the TLB lookup process depending on the value of bit 28 of the DMA_SMEM_HI_CHX_DY.

TABLE 4. VICE DMA view of System Memory

SMEM Address	Address Range	Function	Comments	MSP Access	BSP Access
0xFFFF FFFC 0x2000 0000	3512M	Unused	DMA error int	DMA Engine	DMA Engine
0x1FFF FFFC 0x10C0 0000	244M	Reserved Future Space	64K Frame Buffer Tiles	DMA Engine	DMA Engine
0x10BF FFFC 0x1080 0000	4M	Vice Accessible System Memory	64K Frame Buffer Tiles	DMA Engine	DMA Engine
0x107F FFFC 0x1000 0000	8M	Unused	DMA error int	DMA Engine	DMA Engine
0x0FFF FFFC 0x00C0 0000	244M	Reserved Future Space	64K Linear Page	DMA Engine	DMA Engine
0x00BF FFFC 0x0080 0000	4M	Vice Accessible System Memory	64K Linear Page	DMA Engine	DMA Engine
0x007F FFFC 0x0000 0000	8M	Unused	DMA error int	DMA Engine	DMA Engine

2.7.3 64K Linear Tiles

For a contiguous Linear UNIX page size of 64K, the SMEM pointer is interpreted by the hardware as shown in the figure below. Address bits 21 through 16 form the pointer to select one of 64 addresses of the TLB. The Contents of that TLB address are used to produce the upper 16 bits of system memory address space. These bits are combined with the lower 16 bits of the SMEM pointer to form the complete system memory address. The hardware decomposition of the SMEM pointer is shown graphically in Figure 4, “SMEM Pointer decomposition for 64K Linear Page,” on page 35

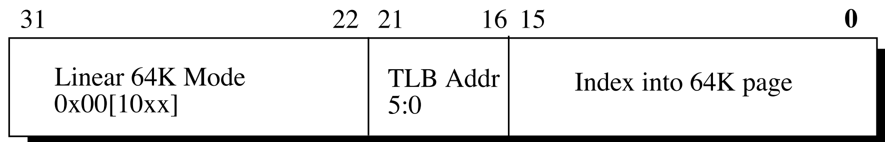


FIGURE 4. SMEM Pointer decomposition for 64K Linear Page

This corresponds to the address range 0x0080 0000 - 0x00BF FFFC in TABLE 1. Vice Address Map.

2.7.4 64K Frame Buffer Tiles

The 4 Meg System Memory can also be configured as a 4K x 1K rectangle. Within this rectangle are placed blocks 512 x 128 of contiguous memory that represent a 64K tiled page. (This can also be thought of as a 1K x 1K rectangle with 128 x 128 blocks with each location 4 bytes deep). For an MSP or BSP program to see this 4K x 1K region as continuously incrementing in raster order, the hardware will decompose the SMEM pointer differently than the 64K linear mode. The indication to the hardware that this mode is selected is through the address space in bit 28 of the SMEM pointer. The TLB is still accessed with a 6 bit pointer made up of bits 21:19 and 11:9 of the SMEM pointer.

Bits 18:12 and 8:0 of the SMEM pointer are appended to the 16 bits from the TLB to form the complete system address.

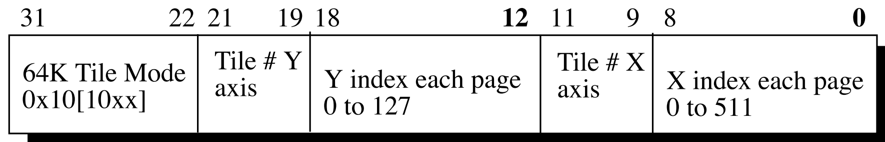


FIGURE 5. SMEM Pointer decomposition for 64K Linear Page

This corresponds to the address range 0x1080 0000 - 0x10BF FFFC in Figure 4, “VICE DMA view of System Memory,” on page 35

This tiled format allows the MSP and BSP to continue to locate pixels in raster order in a linear increasing progression. The fact that memory is crossing page boundaries every 512 locations in the X axis and 128 lines in the Y axis is hidden by the TLB.

This relationship can be seen in Figure 4, “SMEM Pointer decomposition for 64K Linear Page,” on page 35

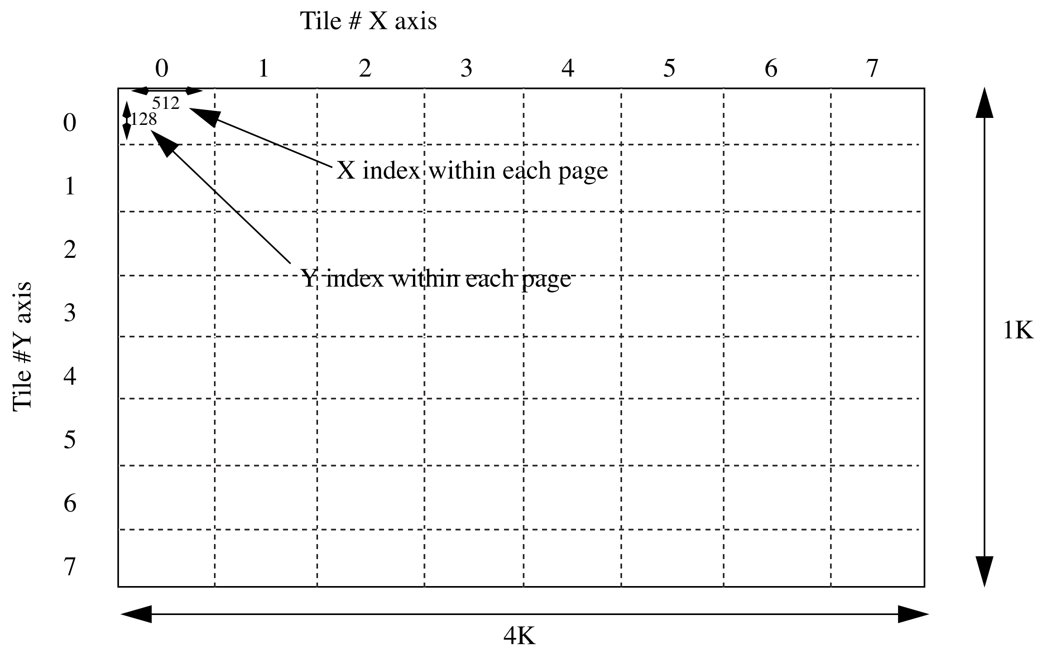


FIGURE 6. 4Meg System Memory using 64K tiles

2.7.5 Hits and Misses

If the virtual address maps to a valid TLB entry the physical page number is extracted from the TLB and concatenated with the offset to form the physical address.

If the virtual address maps to a invalid TLB entry the MSP generates a TLB Miss Exception.

2.7.6 TLB Entry Format

TABLE 5. MSP DMA TLB Entry Format

Bits	Function	Read/Write	Reset
0	Valid	r/w	undefined
1	Writable	r/w	undefined
15:2	Undefined	no access	undefined
31:16	64k Physical Page Number	r/w	undefined

Process to load TLB is as follows:

The Operating System provides a 4k page number for the first 4k page of the 64k physically contiguous range. Since this 4k page number is aligned to 64k its low 4 bits will be 0. That number is shifted right 4 bits and stored into 31:16 of the TLB entry.

The Valid bit is set to 1.

If the Writable bit is set to one (binary on), this allows permission for Vice DMA to write to this range of system memory. If set to zero (binary off) this will prevent Vice DMA from writing to this range of system memory.)

Note that a TLB entry must be written all 32 bits at a time so it must be assembled with the Physical Page Number, Valid and Writeable bits before being written into VICE.

2.8 Register Address Map Summary

The offset within the System and MSP Address space for the Registers is given below. Note that no Address is given to locations of the MSP register file in the MSP address register column. This is because Register File addresses are implicit in the instructions. Table 6, “Register Address Map Summary,” on page 40 shows the accessible registers in the address space. All accesses are on a double-word (64 bit) boundary regardless of the actual data size of the register.

The MSP accesses all control registers and DMA descriptors inside of the VICE chip through co-processor 1 and 3 op codes. This causes the MSP to use the internal VICE Common Address and Data buses during these register accesses. This allows the BSP or DMA engines access to the VICE internal Data RAM at the same time the MSP is accessing a register. Note that any writable register that is accessed with a MTCz or CTCz instruction can be read with it’s complementary MFCz or CFCz instruction.

TABLE 6. Register Address Map Summary

SYS Offset Addr.	MSP Offset Addr.	Register Name	Group	Function	r/w	Reset Value	bits
0008	No Access	VICE_ID	Chip	Rev/ID	r	0xE1	8
0010 0020	No Access	Unused	Chip				
E000	No Access	VICE_CFG	Chip	Vice General Config	r/w	0x00	16
E008	No Access	VICE_INT_RESET	Chip	Reset Interrupt	w	0x00	9
E010	No Access	VICE_INT_EN	Chip	Interrupt Enable Reg	r/w	0x00	9
0028	No Access	HST_BSP_IN_BOX	BSP	Host copy of BSP/ MSP In mailbox	r	0x00	16
0030	No Access	HST_BSP_OUT_BOX	BSP	Host copy of BSP/ MSP Out mailbox	r	0x00	16
0038	Not Access	Unused	Chip				
0040	No Access	MSP_CTL_STAT	MSP	MSP Control/Status Reg	r/w	0x00	32
0048	No Access	MSP_ExcpFlag	MSP	MSP Exception Flag	r/w	0x00	32
0050	No Access	MSP_PC	MSP	MSP Program Counter	r/w	0xXX	32
0058	No Access	MSP_BadAddr	MSP	MSP Bad Address	r	0xXX	32
0060	No Access	MSP_WatchPoint	MSP	MSP WatchPoint	r/w	0x00	32
0068	No Access	MSP_EPC	MSP	MSP Exception PC	r	0xXX	32
0070	No Access	MSP_CAUSE	MSP	MSP Exception Cause	r	0xXX	32
0078	No Access	BSP_RPAGE	BSP	BSP R Page	r/w	0xXX	16
0080	No Access	BSP_SW_INT	Chip	BSP Software Int.	w	0xXX	0
0100	CTC1 $0	MSP_D_RAM	MSP	MSP Data RAM Arbi- tration Register	r/w	0x00	32
0108	CFC1 $1	VICEMSP_COUNT	MSP	MSP Free Running Counter	r	0xXX	32
0110	CTC1 $2	BSP_CTL_STAT	BSP	BSP Control/Status Reg	r/w	0x00	16
0118	CTC1 $3	BSP_WatchPoint	BSP	BSP WatchPoint	r/w	0x00	16
0120	CFC1 $4	BSP_IN_COUNT	BSP	BSP Decoded Bits Counter	r	0xXX	24
0128	CFC1 $5	BSP_OUT_COUNT	BSP	BSP Encoded Bits Counter	r	0xXX	24
0130	CTC1 $6						
0138	CFC1 $7						
0140	CTC1 $8	BSP_PC	BSP	BSP Program Counter	r/w	0x00	16
0148	CTC1 $9	BSP_EPC	BSP	BSP Exception PC	r	0x00	16

TABLE 6. Register Address Map Summary

SYS Offset Addr.	MSP Offset Addr.	Register Name	Group	Function	r/w	Reset Value	bits
0150	CTC1 $10	BSP_HALT_RESET	BSP	BSP Halt and Reset Control Register	r	0x00	2
0158	CTC1 $11	BSP_CAUSE	BSP	BSP Exception Cause	r	0x00	16
0160	CTC1 $12	VICE_INT	Chip	Interrupt and Status	r	0x00	9
0168	CTC1 $13	BSP_FIFO_CTL_STAT	BSP	BSP FIFOs, Control and Status Register	r/w	0x05	6
0170	CTC1 $14	BSP_AVALID_BITS	BSP	BSP A Fifo Valid Bits (Decode Fifo)	r/w	0x00	?
0178	CTC1 $15	BSP_FVALID_BITS	BSP	BSP F Fifo Valid Bits (Encode Fifo)	r/w	0x00	?
0180	CTC1 $16	DMA_CTL_CH1	DMA	Ch 1 DMA Control	r/w	0x10	16
0188	CFC1 $17	DMA_STAT_CH1	DMA	Ch 1 DMA Status	r	0x10	16
0190	CTC1 $18	DMA_DATA_CH1	DMA	Ch 1 DMA Data Fill	r/w	0x00	16
0198	CFC1 $19	DMA_MEM_PT_CH1	DMA	Ch 1 DMA Sys Pointer	r	0x00	32
01A0	CFC1 $20	DMA_VICE_PT_CH1	DMA	Ch 1 DMA Vice Pointer	r	0x00	16
01A8	CFC1 $21	DMA_COUNT_CH1	DMA	Ch 1 DMA Remaining Count	r	0x00	16
01B0	CFC1 $22	Unused					
01B8	CFC1 $23	MSP_SW_INT	Chip	MSP Software Inter-rupt	w	0xXX	0
01C0	CTC1 $24	DMA_CTL_CH2	DMA	Ch 2 DMA Control	r/w	0x10	16
01C8	CFC1 $25	DMA_STAT_CH2	DMA	Ch 2 DMA Status	r	0x10	16
01D0	CTC1 $26	DMA_DATA_CH2	DMA	Ch 2DMA Data Fill	r/w	0x00	16
01D8	CFC1 $27	DMA_MEM_PT_CH2	DMA	Ch 2 DMA Sys Pointer	r	0x00	32
01E0	CFC1 $28	DMA_VICE_PT_CH2	DMA	Ch 2 DMA Vice Pointer	r	0x00	16
01E8	CFC1 $29	DMA_COUNT_CH2	DMA	Ch 2 DMA Remaining Count	r	0x00	16
01F0	CFC1 $30	BSP_IN_BOX	BSP	BSP/MSP In mailbox	r	0x00	16
01F8	CTC1 $31	BSP_OUT_BOX	BSP	BSP/MSP Out mail-box	r/w	0x00	16

2.8.1 DMA descriptor registers

The DMA descriptor registers are grouped below in their own table.

TABLE 7. DMA Descriptor Address Map

SYS Offset Addr.	MSP Offset Addr.	Register Name	Group	Function	r/w	Reset Value	bits
1000	MTC3 $0	DMA_CTL_CH1_D1	DMA	Descriptor Control	r/w	0xXX	16
1008	MTC3 $1	DMA_SMEM_HI_CH1_D1	DMA	Upper Address Pointer	r/w	0xXX	16
1010	MTC3 $2	DMA_SMEM_LO_CH1_D1	DMA	Lower Address Pointer	r/w	0xXX	16
1018	MTC3 $3	DMA_WIDTH_CH1_D1	DMA	Width in Bytes of Line	r/w	0xXX	16
1020	MTC3 $4	DMA_STRIDE_CH1_D1	DMA	# Bytes to Skip	r/w	0xXX	16
1028	MTC3 $5	DMA_LINES_CH1_D1	DMA	# Lines	r/w	0xXX	16
1030	MTC3 $6	DMA_VMEM_Y_CH1_D1	DMA	Vice Pointer Y comp	r/w	0xXX	16
1038	MTC3 $7	DMA_VMEM_C_CH1_D1	DMA	Vice Pointer C comp	r/w	0xXX	16
1040	MTC3 $8	DMA_CTL_CH1_D2	DMA	Descriptor Control	r/w	0xXX	16
1048	MTC3 $9	DMA_SMEM_HI_CH1_D2	DMA	Upper Address Pointer	r/w	0xXX	16
1050	MTC3 $10	DMA_SMEM_LO_CH1_D2	DMA	Lower Address Pointer	r/w	0xXX	16
1058	MTC3 $11	DMA_WIDTH_CH1_D2	DMA	Width in Bytes of Line	r/w	0xXX	16
1060	MTC3 $12	DMA_STRIDE_CH1_D2	DMA	# Bytes to Skip	r/w	0xXX	16
1068	MTC3 $13	DMA_LINES_CH1_D2	DMA	# Lines	r/w	0xXX	16
1070	MTC3 $14	DMA_VMEM_Y_CH1_D2	DMA	Vice Pointer Y comp	r/w	0xXX	16
1078	MTC3 $15	DMA_VMEM_C_CH1_D2	DMA	Vice Pointer C comp	r/w	0xXX	16
1080	MTC3 $16	DMA_CTL_CH1_D3	DMA	Descriptor Control	r/w	0xXX	16
1088	MTC3 $17	DMA_SMEM_HI_CH1_D3	DMA	Upper Address Pointer	r/w	0xXX	16
1090	MTC3 $18	DMA_SMEM_LO_CH1_D3	DMA	Lower Address Pointer	r/w	0xXX	16
1098	MTC3 $19	DMA_WIDTH_CH1_D3	DMA	Width in Bytes of Line	r/w	0xXX	16
10A0	MTC3 $20	DMA_STRIDE_CH1_D3	DMA	# Bytes to Skip	r/w	0xXX	16
10A8	MTC3 $21	DMA_LINES_CH1_D3	DMA	# Lines	r/w	0xXX	16
10B0	MTC3 $22	DMA_VMEM_Y_CH1_D3	DMA	Vice Pointer Y comp	r/w	0xXX	16
10B8	MTC3 $23	DMA_VMEM_C_CH1_D3	DMA	Vice Pointer C comp	r/w	0xXX	16
10C0	MTC3 $24	DMA_CTL_CH1_D4	DMA	Descriptor Control	r/w	0xXX	16
10C8	MTC3 $25	DMA_SMEM_HI_CH1_D4	DMA	Upper Address Pointer	r/w	0xXX	16
10D0	MTC3 $26	DMA_SMEM_LO_CH1_D4	DMA	Lower Address Pointer	r/w	0xXX	16
10D8	MTC3 $27	DMA_WIDTH_CH1_D4	DMA	Width in Bytes of Line	r/w	0xXX	16
10E0	MTC3 $28	DMA_STRIDE_CH1_D4	DMA	# Bytes to Skip	r/w	0xXX	16
10E8	MTC3 $29	DMA_LINES_CH1_D4	DMA	# Lines	r/w	0xXX	16
10F0	MTC3 $30	DMA_VMEM_Y_CH1_D4	DMA	Vice Pointer Y comp	r/w	0xXX	16
10F8	MTC3 $31	DMA_VMEM_C_CH1_D4	DMA	Vice Pointer C comp	r/w	0xXX	16
1100	CTC3 $0	DMA_CTL_CH2_D1	DMA	Descriptor Control	r/w	0xXX	16
1108	CTC3 $1	DMA_SMEM_HI_CH2_D1	DMA	Upper Address Pointer	r/w	0xXX	16

TABLE 7. DMA Descriptor Address Map

SYS Offset Addr.	MSP Offset Addr.	Register Name	Group	Function	r/w	Reset Value	bits
1110	CTC3 $2	DMA_SMEM_LO_CH2_D1	DMA	Lower Address Pointer	r/w	0xXX	16
1118	CTC3 $3	DMA_WIDTH_CH2_D1	DMA	Width in Bytes of Line	r/w	0xXX	16
1120	CTC3 $4	DMA_STRIDE_CH2_D1	DMA	# Bytes to Skip	r/w	0xXX	16
1128	CTC3 $5	DMA_LINES_CH2_D1	DMA	# Lines	r/w	0xXX	16
1130	CTC3 $6	DMA_VMEM_Y_CH2_D1	DMA	Vice Pointer Y comp	r/w	0xXX	16
1138	CTC3 $7	DMA_VMEM_C_CH2_D1	DMA	Vice Pointer C comp	r/w	0xXX	16
1140	CTC3 $8	DMA_CTL_CH2_D2	DMA	Descriptor Control	r/w	0xXX	16
1148	CTC3 $9	DMA_SMEM_HI_CH2_D2	DMA	Upper Address Pointer	r/w	0xXX	16
1150	CTC3 $11	DMA_SMEM_LO_CH2_D2	DMA	Lower Address Pointer	r/w	0xXX	16
1158	CTC3 $11	DMA_WIDTH_CH2_D2	DMA	Width in Bytes of Line	r/w	0xXX	16
1160	CTC3 $12	DMA_STRIDE_CH2_D2	DMA	# Bytes to Skip	r/w	0xXX	16
1168	CTC3 $13	DMA_LINES_CH2_D2	DMA	# Lines	r/w	0xXX	16
1170	CTC3 $14	DMA_VMEM_Y_CH2_D2	DMA	Vice Pointer Y comp	r/w	0xXX	16
1178	CTC3 $15	DMA_VMEM_C_CH2_D2	DMA	Vice Pointer C comp	r/w	0xXX	16
1180	CTC3 $16	DMA_CTL_CH2_D3	DMA	Descriptor Control	r/w	0xXX	16
1188	CTC3 $17	DMA_SMEM_HI_CH2_D3	DMA	Upper Address Pointer	r/w	0xXX	16
1190	CTC3 $18	DMA_SMEM_LO_CH2_D3	DMA	Lower Address Pointer	r/w	0xXX	16
1198	CTC3 $19	DMA_WIDTH_CH2_D3	DMA	Width in Bytes of Line	r/w	0xXX	16
11A0	CTC3 $20	DMA_STRIDE_CH2_D3	DMA	# Bytes to Skip	r/w	0xXX	16
11A8	CTC3 $21	DMA_LINES_CH2_D3	DMA	# Lines	r/w	0xXX	16
11B0	CTC3 $22	DMA_VMEM_Y_CH2_D3	DMA	Vice Pointer Y comp	r/w	0xXX	16
11B8	CTC3 $23	DMA_VMEM_C_CH2_D3	DMA	Vice Pointer C comp	r/w	0xXX	16
11C0	CTC3 $24	DMA_CTL_CH2_D4	DMA	Descriptor Control	r/w	0xXX	16
11C8	CTC3 $25	DMA_SMEM_HI_CH2_D4	DMA	Upper Address Pointer	r/w	0xXX	16
11D0	CTC3 $26	DMA_SMEM_LO_CH2_D4	DMA	Lower Address Pointer	r/w	0xXX	16
11D8	CTC3 $27	DMA_WIDTH_CH2_D4	DMA	Width in Bytes of Line	r/w	0xXX	16
11E0	CTC3 $28	DMA_STRIDE_CH2_D4	DMA	# Bytes to Skip	r/w	0xXX	16
11E8	CTC3 $29	DMA_LINES_CH2_D4	DMA	# Lines	r/w	0xXX	16
11F0	CTC3 $30	DMA_VMEM_Y_CH2_D4	DMA	Vice Pointer Y comp	r/w	0xXX	16
11F8	CTC3 $31	DMA_VMEM_C_CH2_D4	DMA	Vice Pointer C comp	r/w	0xXX	16

2.9 Register Description

In the following 8/16/32-bit register descriptions, all bits not explicitly defined are read back as 0 from the Host port. Note that an approved "Name" has been suggested for each register. This name appears as part of each registers' paragraph title (so that it will show up in the table of contents) and in the Table Title. Where a register is representative of a set of registers (one per channel) all register names are listed in the section paragraph while the first channel is listed in the Paragraph and Table titles. This will help both hardware and software refer to the register throughout the life of this hardware by using the same moniker.

Registers appear on doubleword address boundaries (8bytes) regardless of the size of the register.

The term "write 1" or "write a logical 1" or "write 0" or "write a logical 0" used in the following descriptions implies that those particular bits should be that value during the write. All other bits will take effect during the write, so the write operation needs to consider this as well. This will require a read-modify-write approach to make sure that software only changes the desired bits.

2.9.1 VICE_ID - Chip ID and Revision Register Format

This register will change only when the chip itself goes through a revision.

TABLE 8. VICE_ID Register Format

Bits	Function	Read/Write	Mask Value
3:0	VICE revision number 099-0123-001 Vice-A (VTI # vy06762) 099-0123-002 Vice-B (VTI # vy21314-) "DX" 099-0123-003 Vice-C (VTI # vy21314b) "TRE"	Read Only	 0001 0010 0011
7:4	VICE ID value Vice 'A" -001 vy6167	Read Only	

2.9.2 VICE_CFG - General Configuration Register

General purpose configuration register for the VICE chip. Features which tend to be performed one time at system power up or initialization should be put in this register. If we decide to support little endian (which I would strongly discourage) then this would be the place to put it.

TABLE 9. VICE_CFG Register Format

Bits	Function	Read/ Write	Reset Value
0	check_data_sysad 0=No check data [SysCmd(4) = 1 on data identifiers when VICE is external agent] 1=Check data	r/w	0
1	MSP TLB Bypass 0=Do not allow MSP to bypass TLB even if it sets the bit in the DMA config register 1=Allow MSP to bypass TLB under control of bit in the DMA config register that MSP can access	r/w	0
31:02	Not Defined		0

2.9.3 VICE_INT_RESET - Interrupt Reset Register

The **VICE_INT_RESET** register is write only. Writing a logical 1 to a bit in this register will clear the corresponding bit in the VICE_INT register. A write to the register affects all bits that are being written with a logical 1. The register is cleared (logical 0) on reset. Writing a logical 0 to any of the bits has no effect on the value of the bit.

TABLE 10. VICE_INT_RESET Register Format

Bits	Function	Read/ Write	Reset Value
0	DMA complete interrupt Channel 1	w	0
1	DMA error interrupt Channel 1	w	0
2	MSP Software interrupt to Unix Processor	w	0
3	MSP exception interrupt to Unix Processor	w	0
4	BSP Software Interrupt to Unix Processor	w	0
5	BSP exception Interrupt to Unix Processor	w	0
6	SysAD erroneous data received When Data Identifier SysCmd(5) = 1 then data on SysAD contains error. This interrupt bit is set on that condition.	w	0
7	DMA complete interrupt Channel 2	w	0
8	DMA error interrupt Channel 2	w	0

2.9.4 VICE_INT - Interrupt Status Register

The **VICE_INT** register is read only. All bits are active high (logical 1). The bits are set (logical 1) by hardware and cleared by a software write to the VICE_INT_RESET register, (write logical 1to clear). The register is cleared (logical 0) on reset. Writing to this register has no effect on the value of the bit.

TABLE 11. VICE_INT Register Format

Bits	Function	Read/ Write	Reset Value
0	DMA complete interrupt Channel 1	r	0
1	DMA error interrupt Channel 1	r	0
2	MSP Software interrupt to Unix Processor	r	0
3	MSP exception interrupt to Unix Processor	r	0
4	BSP Software Interrupt to Unix Processor	r	0
5	BSP exception Interrupt to Unix Processor	r	0
6	SysAD erroneous data received When Data Identifier SysCmd(5) = 1 then data on SysAD contains error. This interrupt bit is set on that condition.	r	0
7	DMA complete interrupt Channel 2	r	0
8	DMA error interrupt Channel 2	r	0

2.9.5 VICE_INT_EN - Interrupt Enable Register

The **VICE_INT_EN** register is read/write. Writing a logical 1 to a bit location of the register will allow the interrupt (when it occurs) to affect the interrupt pin of the VICE chip. Regardless of the state of this bit, the VICE_INT register reflects the state of any interrupt condition.

This means that an interrupt can be disabled by writing its appropriate bit to a 0 in this register. When a condition occurs that would normally create an interrupt, the relevant bit in the VICE_INT register will be set to a logical 1.

TABLE 12. VICE_INT_EN Register Format

Bits	Function	Read/ Write	Reset Value
0	DMA complete interrupt Channel 1	r/w	0
1	DMA error interrupt Channel 1	r/w	0
2	MSP Software interrupt to Unix Processor	r/w	0
3	MSP exception interrupt to Unix Processor	r/w	0
4	BSP Software Interrupt to Unix Processor	r/w	0
5	BSP exception Interrupt to Unix Processor	r/w	0
6	SysAD erroneous data received When Data Identifier SysCmd(5) = 1 then data on SysAD contains error. This interrupt bit is set on that condition.	r/w	0
7	DMA complete interrupt Channel 2	r/w	0
8	DMA error interrupt Channel 2	r/w	0

2.9.6 BSP_SW_INT - BSP Software Interrupt Register

The **BSP_SW_INT** register is write only. Writing any value to this register will cause the BSP Software Interrupt to Unix Processor bit to be set. This bit is contained in the VICE_INT Register. A read from this register will deliver meaningless data.

The interrupt bit in the VICE_INT Register will be set and latched regardless of the state of the VICE_INT_EN bit corresponding to this interrupt.

2.9.7 MSP_SW_INT - MSP Software Interrupt Register

The **MSP_SW_INT** register is write only. Writing any value to this register will cause the MSP Software Interrupt to Unix Processor bit to be set. This bit is contained in the VICE_INT Register. A read from this register will deliver meaningless data.

The MSP accesses this register as a CTC1 $31 instruction.

The interrupt bit in the VICE_INT Register will be set and latched regardless of the state of the VICE_INT_EN bit corresponding to this interrupt.

2.9.8 MSP_D_RAM - Data RAM Arbitration Register

Data RAM arbitration register. The MSP can inform the arbiter as to which Data RAM banks it intends to access. Any access to a Data RAM bank that has been configured as “Not Allowed” will cause the Load/Store instruction for that address to be ignored by the Data RAM control logic. The MSP will read unknown data. Data will not be written to Data RAM on Stores that violate the “Not Allowed” address space.

The D_RAM_EN bits are used by the internal arbiter to allow users of the DMA bus access to a particular Data Ram Bank without suspending access by the Scalar Unit. The safe way is to allow the MSP access to all three memories, but then DMA will be stalled whenever the Scalar Unit does a Load/Store.

TABLE 13. MSP_D_RAM Register Format

Bits	Function	Read/ Write	Reset Value
0	D_RAM_A_EN 0=Do Not Allow MSP access to Data RAM A 1=Allow MSP access to Data RAM A	r/w	0
1	D_RAM_B_EN 0=Do Not Allow MSP access to Data RAM B 1=Allow MSP access to Data RAM B	r/w	0
2	D_RAM_C_EN 0=Do Not Allow MSP access to Data RAM C 1=Allow MSP access to Data RAM C	r/w	0

2.9.9 MSP_CTL_STAT - Media Signal Processor Control Register

The **MSP_CTL_STAT** register is read/write. This is primarily a diagnostic register that is used in conjunction with the BREAK Instruction to set a breakpoint for the Media Signal Processor.

After Power Up Reset this register will be 0x00.

Any time the MSP is running, it can be halted and reset by writing this register as 0x00.

For setting up the MSP in normal running mode Take MSP out of Reset (bit 1 write 1), and write the GO bit to logical 1. Poll the Go bit see if the MSP is still running or halted, because of reaching a BREAK instruction. For example sequences, See "Debug Operations" on page 30.

TABLE 14. MSP_CTL_STAT Register Format

Bits	Function	Read/ Write	Reset Value
0	MSP GO/HALT 1 - Writing a 1; Causes MSP to start. 0 - Writing a 0; If MSP Stopped No effect, If MSP Running, MSP will be halted. 1 - Reading a 1; MSP is running 0 - Reading a 0; MSP is halted	r/w	0
1	Reset MSP 0 - MSP is held in reset. 1- MSP taken out of reset. Wait for Go bit.	r/w	0

2.9.10 MSP_PC Media Signal Processor Program Counter Register

The *MSP_P*C register is read/write. The MSP_PC register must always be loaded whenever the MSP is taken out of its halted state in order to direct restart of program execution.

TABLE 15. MSP_PC Register Format

Bits	Function	Read/Write	Reset
31:16	0	Readonly	0
15:2	MSP PC	Read/Write	Unknown
1:0	0	Readonly	0

Programmer's Note: Writing the MSP_PC writes into the Instruction Fetch PC while reading the MSP_PC returns the value of the Decode PC. Hence, when the MSP is halted, writing the MSP_PC and then immediately reading the MSP_PC will return a different value from what was written.

2.9.11 MSP_BadAddr Register

The *MSP_BadAddr* register is a read-only register that displays the load/store address that caused an exception. (AdEL or AdES)

TABLE 16. MSP_BadAddr Register Format

Bits	Function	Read/Write	Reset
31:0	Bad Address	Readonly	Unknown

2.9.12 MSP_WatchPoint Register

The MSP provides a debugging feature to detect references to a selected physical address; MSP load or store operations to the location specified by the *MSP_WatchPoint* register. This is a read-write register.

TABLE 17. MSP_WatchPoint Register Format

Bits	Function	Read/Write	Reset
31:3	WatchPoint Address	Read/Write	0
2:0	0	Readonly	0

2.9.13 MSP_EPC Registers

The *Exception Program Counter* register, EPC, is a 32-bit read-only register that contains the address of the instruction which was the direct cause of the exception.

TABLE 18. MSP_EPC Register Format

Bits	Function	Read/Write	Reset
1:0	0	Readonly	0
15:2	Exception Program Counter	Readonly	Unknown
31:16	0	Readonly	0

2.9.14 MSP_CAUSE Register

The *Cause* register is a 32-bit read-only register. Its contents describe the cause of the last exception. A 5 bit exception code is listed below. The Branch Delay (BD) bit indicates whether the last exception was taken while executing in a branch delay slot. (0 -> normal; 1 -> delay slot)

TABLE 19. MSP_CAUSE Register Format

Bits	Function	Read/Write	Reset
1:0	0	Readonly	0
6:2	Exception Code	Readonly	Unknown
30:7	0	Readonly	0
31	BD	Readonly	Unknown

TABLE 20. Exception Code Field of Cause Register

Exception Code Value	Mnemonic	Description	Source
0	AdEL	Address error exception (load)	SU
1	AdES	Address error exception (store)	SU
2	BP	Breakpoint exception	SU
3	WP	Watchpoint exception	SU
4	SuRI	SU Reserved instruction exception	SU
5	VuRI	VU Reserved instruction exception	VU
6	Con	Contention exception	SU
7	AdEI	Instruction fetch addr exception	SU
8-31	-	Reserved for future use	-

2.9.15 MSP_ExcpFlag Registers

The *Exception Flag* register, EPC, is a 32-bit read-write register that has one bit associated with each exception. Once an exception is detected, this corresponding bit is set. The bits need to be cleared explicitly by software. This register is to be mainly used as a diagnostic register to catch multiple exceptions. If several exceptions should occur within the same cycle, the Cause register would only record the cause of the exception which has the highest priority. This Exception Flag register would allow the programmer to know if multiple exceptions occurred.

Note: *All bits in the MSP_ExcpFlag Register must be cleared before exceptions will be detected.*

TABLE 21. MSP_ExcpFlag Register Format

Bits	Function	Read/Write	Reset
0	AdEL	Read/Write	0
1	AdES	Read/Write	0
2	BP	Read/Write	0
3	WP	Read/Write	0
4	SuRI	Read/Write	0
5	VuRI	Read/Write	0
6	Con	Read/Write	0
7	AdEI	Read/Write	0
31:8	0	Readonly	0

2.9.16 VICEMSP_COUNT - MSP Free Running Counter

32 bit counter that runs off the MSP clock. Used to measure number of clocks that have elapsed for a given operation.

Counts up in binary. 4Gig resolution allows for approximately 60 seconds before counter rolls over. This assumes 66MHz MSP clock.

Counter is writable for diagnostics reasons. If scan logic implemented on this counter in the physical design of the chip, it will be changed to read only.

TABLE 22. MSP_COUNT Register Format

Bits	Function	Read/ Write	Reset Value
31:00	VICEMSP_COUNT	r/w	0

2.9.17 BSP_CTL_STAT - Bit Stream Processor Control and Status Register

The **BSP_CTL_STAT** register is read/write. For a complete description of this register refer to Figure 47, "BSP Status and Control Register," on page 178.

2.9.18 BSP_WatchPoint Register

The BSP provides a debugging feature to detect references to a selected physical address; BSP load or store operations to the location specified by the **BSP_WatchPoint** register. This is a read-write register. Bit 0 is ignored so byte and word accesses to the selected physical address will cause a match to occur.

TABLE 23. BSP_WatchPoint Register Format

Bits	Function	Read/Write	Reset
0	0	Readonly	0
15:01	WatchPoint Address	Read/Write	0

2.9.19 BSP_IN_COUNT - BSP Input bits counter

24 bit counter that updates as bits are extracted by the Bit Stream Processor from the Bit Stream Buffer. Used by the Host to implement a rate buffer control algorithm for de-compression applications. This regiater counts the number of 32-bit transfers from the BSP Input FIFO to the BSP's Beta register. This register along with the fVALIDBITS register to compute the number of bits consumed by the BSP.

TABLE 24. BSP_IN_COUNT Register Format

Bits	Function	Read/ Write	Reset Value
23:00	BSP_IN_COUNT	r/w	0

2.9.20 BSP_OUT_COUNT - BSP Output bits counter

24 bit counter that updates as the bits are inserted by the Bit Stream Processor into the Bit Stream Buffer. Used by the Host to implement a buffer control algorithm for compression applications. This register counts the number of 16-bit ransfers from the BSP Alpha register to the BSP's write buffer. Along with the fALPHAVALIDBITS and FVALIDBITS registers, the number of bits produced by the BSP can be computed.

TABLE 25. BSP_OUT_COUNT Register Format

Bits	Function	Read/ Write	Reset Value
23:00	BSP_OUT_COUNT	r/w	0

2.9.21 BSP_IN_BOX - Bit Stream Processor Input Mailbox

The **BSP_IN_BOX** register is read/write by the Medial Signal Processor. It is read only by the Bit Stream Processor. This register is used to communicate control information between the MSP and the BSP.

Bit 15 of this register is a “magic” bit. When the Media Signal Processor writes this register, Bit 15 will become a logical 1. This will occur regardless of the value that the Media Signal Processor placed in Bit 15 of the word written. When the Bit Stream Processor reads this register, Bit 15 will become a logical 0. The value of Bit 15 will be read by the BSP along with bits 14-00. The value of Bit 15 read by the BSP will be the value prior to the read operation. The MSP can read this register at any time and use the state of Bit 15 to decide if the BSP has read its mail.

After Power Up Reset this register will be 0x00.

TABLE 26. BSP_IN_BOX Register Format

Bits	Function	MSP Read/ Write	BSP Read/ Write	Reset Value
14:00	BSP_IN_MESSAGE	r/w	r	0
15	BSignal Any MSP, BSP or HD write to this register causes the bit to become a logical 1. Any MSP,BSP or HD read from this register causes the bit to become a logical 0. The value read will correspond to the value of the bit prior to it being set to zero. 1 - MSP Read 1, BSP has **NOT** read the message 0 - MSP Read 0, BSP has read the message	r/w	r	0

2.9.22 BSP_OUT_BOX - Bit Stream Processor Output Mailbox

The **BSP_OUT_BOX** register is read/write by the Bit Stream Processor. It is read only by the Media Signal Processor. This register is used to communicate control information between the BSP and the MSP.

Bit 15 of this register is a “magic” bit. When the Bit Stream Processor writes this register, Bit 15 will become a logical 1. This will occur regardless of the value that the Bit Stream Processor placed in Bit 15 of the word written. When the Media Signal Processor reads this register, Bit 15 will become a logical 0. The value of Bit 15 will be read by the MSP along with bits 14-00. The value of Bit 15 read by the MSP will be the value prior to the read operation. The BSP can read this register at any time and use the state of Bit 15 to decide if the MSP has read its mail.

After Power Up Reset this register will be 0x00.

TABLE 27. BSP_OUT_BOX Register Format

Bits	Function	MSP Read/ Write	BSP Read/ Write	Reset Value
14:00	BSP_OUT_MESSAGE	r	r/w	0
15	SUSignal Any BSP, HD or MSP write to this register causes the bit to become a logical 1. Any MSP, HJD or BSP read from this register causes the bit to become a logical 0. The value read will be that value prior to the bit being set to zero. 1 - BSP Read 1, MSP has **NOT** read the message 0 - BSP Read 0, MSP has read the message	r	r/w	0

2.9.23 HST_BSP_IN_BOX - Host Snoop of BSP Input Mailbox

This address is a shadow read address of the BSP_IN_BOX register. It allows the Host (or any processor with access to the common bus address space internal to VICE) to read the BSP_IN_BOX without affecting the “magic” bit (Bit 15). The MSP cannot access this register as it not assigned one of the CTC addresses used by the MSP to access the common bus registers.

TABLE 28. HST_BSP_IN_BOX Register Format

Bits	Function	Read/ Write	Reset Value
14:00	BSP_IN_MESSAGE	r	0
15	BSignal	r	0

2.9.24 HST_BSP_OUT_BOX - Host Snoop of BSP Output Mailbox

This address is a shadow read address of the BSP_OUT_BOX register. It allows the Host (or any processor with access to the common bus address space internal to VICE) to read the BSP_OUT_BOX without affecting the “magic” bit (Bit 15). The MSP cannot access this register as it not assigned one of the CTC addresses used by the MSP to access the common bus registers.

TABLE 29. HST_BSP_OUT_BOX Register Format

Bits	Function	Read/ Write	Reset Value
14:00	BSP_OUT_MESSAGE	r	0
15	SUSignal	r	0

2.9.25 BSP_PC Bitstream Processor Program Counter Register

The **BSP_PC** register is read/write. The BSP_PC register may be loaded whenever the BSP is taken out of its halted state in order to direct restart of program execution. The reset value of this register is zero, so without initialization, the BSP will begin execution at location 0x0000 of it instruction memory.

TABLE 30. BSP_PC Register Format

Bits	Function	Read/Write	Reset
15:0	BSP PC	Read/Write	0x0000

2.9.26 BSP_EPC Bitstream Processor Exception Program Counter

The *Exception Program Counter* register, EPC, is a 16-bit read-only register that contains the address of the instruction which was the direct cause of the exception.

TABLE 31. BSP_EPC Register Format

Bits	Function	Read/Write	Reset
15:0	BSP EPC	Read	Unknown

2.9.27 BSP_HALT_RESET - Bit Stream Processor Halt and Reset Register

The **BSP_HALT_RESET** register is read/write. This is primarily a diagnostic register that is used in conjunction with the BREAK Instruction to set a breakpoint for the Bit Stream Processor.

After Power Up Reset this register will be 0x00.

Any time the BSP is running, it can be halted and reset by writing this register as a logical 0.

For setting up the BSP in normal running mode Take BSP out of Reset (bit 1 write 1) and write the GO bit to logical 1. Poll Go bit to see if the BSP is still running or if it has halted because of reaching a BREAK instruction.

TABLE 32. BSP_HALT_RESET Register Format

Bits	Function	Read/ Write	Reset Value
0	Reset BSP 0 - BSP is held in reset. 1- BSP taken out of reset. Wait for Go bit.	r/w	0
1	BSP HALT/GO 0- Writing a 0; Causes BSP to start. 1 - Writing a 1; If BSP Stopped No effect, If BSP Running, BSP will be halted. 0- Reading a 0; BSP is running 0- Reading a 1; BSP is halted	r/w	1
2	HALT_ACK 0 - BSP has not recognized a HALT request (1 written into bit 0 of this register). 1- BSP has recognized the HALT request (1 written into bit 0 of this register) . When this bit is asserted, all instructions in progress when the HALT request is asserted (1) have come to completion. This includes any instructions using the encode pipeline or the multi-cycle state machine (format D BSP instructions).	r/w	0

Assertion of the HALT/GO bit to 1 will cause the BSP to come to a graceful stop. The Exception Program Counter (EPC) captures the contents of the PC when the HALT/GO bit is set. All subsequent instructions in the BSP execution pipeline are nullified. Any instructions in progress, including instructions in the BSP encode pipe execute to completion before the HALT_ACK bit is asserted in acknowlegement of the halt request. Note: All instructions executing when the HALT/GO is asserted will come to completion, incurring stalls in the process. If the stall condition cannot be satisfied, the BSP may remain in a stall condition indefinitely. This HALT/GO bit does not pre-empt any BSP stall conditions. In order to receive the HALT_ACK, all stall conditions must be resolved. This means that one should be careful to make sure that the BSP is HALTED before the HD processor so that the HD can service potential BSP stall conditions.

2.9.28 BSP_CAUSE Register

The *Cause* register is a 16-bit read-only register. Its contents describe the cause of the last exception.

TABLE 33. BSP_CAUSE Register Format

Bits	Function	Read/Write	Reset
0	Address Error - LOAD	Readonly	0
1	Address Error - STORE	Readonly	0
2	BREAK exception	Readonly	0
3	WATCHPOINT exception	Readonly	0
4	RESERVED INSTRUCTION-exception	Readonly	0
5	INSTRUCTION ADDRESS exception	Readonly	0
6	BITSTREAM ERROR exception	Readonly	0
7	WRITE_BUFFER ADDRESS error exception	Readonly	0

An address error on a load is caused by loading a halfword from an odd-byte address.

An address error on a store is caused by trying to store a halfword to an odd-byte address.

A break exception is caused by the execution of a BREAK instruction.

A watchpoint eception --- NOT implemented.

A reserved-instruction exception is caused by the attempt to execute an undefine instruction.

An instruciton address exception is caused by trying to execute from non-existent instruction memory.

A bitstream error exception is cused by the detection of an error in a bitstream (during bitstream decoding). This exception does not cause the BSP to stop executing. Whne detected, the BSP will take a branch to the error location (0x00c0) in its instruction memory and will continue to execute from there. (Note: a break instruction can be palce there to cause an exception to be taken).

A write-buffer address exception iws cause by an odd-byte address in the BSP write buffer. NB. the write buffer only writes halfwords.

2.9.29 BSP_FIFO_CTL_STAT - Bit Stream Processor Fifo Control and Status Register

The **BSP_FIFO_CTL_STAT** register is read/write. This is primarily a diagnostic register that is used to check the BSP Input Fifo Flags. The BSP Input Fifo pointers can be reset with this register.

The values on the MSP_SIGNAL and BSP_SIGNAL bits are take directly out to the MSP_SGNAL and BSP_SIGANL outputs of the VICE chip.

.

TABLE 34. BSP_FIFO_CTL_STAT Register Format

Bits	Function	Read/ Write	Reset Value
0	BSP INPUT FIFO EMPTY 1 - Reading a 1; BSP input Fifo is Empty 0 - Reading a 0; BSP input Fifo is Not Empty (Contains 1 or more entries)	r	1
1	BSP INPUT FIFO FULL 1 - Reading a 1; BSP input Fifo is Full, Contains 16 entries 0 - Reading a 0; BSP input Fifo is Not Full (Contains 15 or less entries)	r	0
2	Reset BSP INPUT FIFO 1- BSP Input FIFO is reset and held in reset. 0- BSP Input FIFO is taken out of reset. Normal Operation	r/w	0
3	MSP_SIGNAL	r/w	?
4	BSP_SIGNAL	r/w	?

2.9.30 BSP_AVALID_BITS - Bit Stream Processor A Fifo Valid Bits Register

The **BSP_AVALID_BITS** register is read/write.

After Power Up Reset this register will be 0x00.

TABLE 35. BSP_AVAILD_BITS Register Format

Bits	Function	Read/ Write	Reset Value
5:0	BSP A VALID BIT Pointer	r/w	0
15:6	Reserved	r/w	0

2.9.31 BSP_FVALID_BITS - Bit Stream Processor F Fifo Valid Bits Register

The **BSP_FVALID_BITS** register is read/write.

After Power Up Reset this register will be 0x00.

TABLE 36. BSP_FVALID_BITS Register Format

Bits	Function	Read/ Write	Reset Value
5:0	BSP F VALID BIT Pointer	r/w	0
15:6	Reserved	r/w	0

2.9.32 DMA_CTL_CH1 - DMA Control Register

The **DMA_CTL_CH1** and **DMA_CTL_CH2** registers set up control functions for each channel of DMA. The data in this register informs the DMA engine which descriptor set to start with, when to stop, when to go and when to reset. If interrupts are to be allowed there is a bit to enable them.

TABLE 37. DMA_CTL_CH1 Register Format

Bits	Function	Read/ Write	Reset Value
0	DMA_CTL_GO 0=DMA hardware resets to 0 when complete 1=Start DMA Note: Writing a 0 to this bit has no affect on stopping DMA. The DMA_CTL_STOP bit must be asserted to stop DMA that is in progress.	r/w	0
1	DMA_CTL_IE 0=No interrupt when DMA complete 1=Allow Interrupt when DMA complete	r/w	0
2	DMA_CTL_STOP 0=Allow DMA to Run 1=Stop DMA	r/w	0
3	DMA_CTL_RESET 0=Allow DMA to Run 1=Reset DMA State Machine and FIFOs independent of VICE chip reset.	r/w	0
7:4	DMA_CTL_DESCR_PT 0001=Start DMA with First Descriptor Set 0010=Start DMA with Second Descriptor Set 0100=Start DMA with Third Descriptor Set 1000=Start DMA with Fourth Descriptor Set	r/w	0001
8	DMA_TLB_BYP 0=DMA will use TLB for address translation 1=DMA will bypass the TLB and treat addresses in the descriptor set as physical memory addresses.	r/w	0
9	DMA_FLUSH_BUF 0=Normal DMA Operation 1=If DMA w/ Non-fifo source, empty SysAD buffer and indicate DMA Complete. If DMA Write BSP Output Buffer -> System Memory then empty buffer and dump DMA buffer into memory before setting DMA Done bit. This bit is not set at the beginning of a DMA transfer. It is set to terminate a DMA transfer of compressed bits and flush the last of those bits prior to terminating the DMA.		0

2.9.33 DMA_STAT_CH1 - VICE DMA Status Register

The **DMA_STAT_CH1** and **DMA_STAT_CH2** registers allow monitoring of each channel of DMA. The information in this register can allow a processor to determine if DMA is active, if it has completed and if it completed as a result of an error. Internal states of DMA are brought out as Status codes.

TABLE 38. DMA_STAT_CH1 Register Format

Bits	Function	Read/ Write	Reset Value
0	DMA_STAT_DONE 0=DMA Not complete 1=DMA Complete (all descriptors finished)	r	0
1	DMA_STAT_ERROR 0=No DMA error has occurred 1=DMA error has occurred	r	0
2	DMA_STAT_ACTIVE 0=DMA is not running 1=DMA is running	r	0
3	DMA_STAT_RW 0=DMA is performing a Write Descriptor 1=DMA is performing a Read Descriptor	r	0
7:4	DMA_STAT_DESCR_PT 0001=DMA working on or pointing to First Descriptor Set 0010=DMA working on or pointing to Second Descriptor Set 0100=DMA working on or pointing to Third Descriptor Set 1000=DMA working on or pointing to Fourth Descriptor Set	r	0001
11:8	DMA_STAT_CODE 0000=DMA Idle 0001=DMA Halted from DMA Stop bit in Control Reg 0010=DMA Halted from DMA Halt bit in Descriptor 0011=DMA computing addresses 0100=DMA waiting to move addresses to CRIME 0101=DMA waiting for response data from CRIME 0110=DMA moving data on internal VICE bus 0111=DMA Halted from TLB Miss (Address Invalid) 1000=DMA Halted from TLB MOD (Write attempted to read only address) 10001=DMa fetching Descriptors	r	0

2.9.34 DMA_DATA_CH1 - VICE DMA Data Fill Register

The **DMA_DATA_CH1** and **DMA_DATA_CH2** registers are used as the data source when the DMA mode to fill memory with a data pattern is selected. This 16 bit value is replicated 4 times by the DMA engine to build a 64 bit word. The DMA engine then runs at full DMA bus bandwidth (inside the VICE chip) during the DMA data fill operation. Note that this limits the data pattern to a unique 16 bit value that is then replicated across the 64 bit word.

The destination for a DMA data fill operation cannot be Unix System memory. However, this effect can be achieved by first performing a DMA data fill operation on the VICE internal Data RAM and then performing a DMA write operation from VICE internal Data RAM to Unix System memory.

TABLE 39. DMA_DATA_CH1 Register Format

Bits	Function	Read/ Write	Reset Value
15:0	DMA_DATA	r/w	0

2.9.35 DMA_MEM_PT_CH1 - DMA System Memory Pointer

The **DMA_MEM_PT_CH1** and **DMA_MEM_PT_CH2** registers contain internal DMA engine state. The DMA engine calculates Addresses from the descriptor list. These registers contain the internal state machine's last calculated physical address for Unix System Memory. This address is 4 byte aligned in system memory. Useful for diagnostics when figuring out where the DMA engine halted.

System Memory Pointer is updated after data moved from System RAM to internal Vice DMA buffer. Vice Memory Pointer is updated after data is moved from internal Vice DMA buffer to Vice Data RAM. DMA Count Updated after data moved from internal Vice DMA buffer to Vice Data RAM.

TABLE 40. DMA_MEM_PT_CH1 Register Format

Bits	Function	Read/ Write	Reset Value
31:0	DMA_MEM_PT	r	0

2.9.36 DMA_VICE_PT_CH1 - DMA Internal Vice Memory Pointer

The **DMA_VICE_PT_CH1** and**, DMA_VICE_PT_CH2** registers contain internal DMA engine state. The DMA engine calculates Addresses from the descriptor list. These registers contain the internal state machine's last calculated Address for memory inside of VICE. Valid for Y channel only in the Y/C split mode. Useful for diagnostics.

TABLE 41. DMA_VICE_PT_CH1 Register Format

Bits	Function	Read/ Write	Reset Value
31:16	DMA_VICE_PT Contains the C address pointer in Y/C mode	r	0
15:0	DMA_VICE_PT Contains the Y address pointer in Y/C mode	r	0

2.9.37 DMA_COUNT_CH1 - DMA Internal Vice DMA Counter

The **DMA_COUNT_CH1** and **DMA_COUNT_CH2** registers contain the remaining byte count of the current DMA descriptor. Useful for diagnostics to see how many bytes have been transferred when DMA terminated. Useful to check progress within a descriptor to see how much longer the DMA channel will be processing the current descriptor.

TABLE 42. DMA_COUNT_CH1 Register Format

Bits	Function	Read/ Write	Reset Value
15:0	DMA_COUNT	r	0

2.9.38 DMA_CTL_CHX_DY - DMA Descriptor Control Entry

Each DMA channel has four descriptor sets that can be programmed. A set of descriptors consists of 8 register entries:
DMA_CTL_CH
DMA_SMEM_HI
DMA_SMEM_LO
DMA_WIDTH
DMA_STRIDE
DMA_LINES
DMA_VMEM_Y
DMA_VMEM_C

The DMA engine can process these descriptors sequentially without additional input from a processor. Any descriptor in the list can be programmed to cause the DMA engine to halt AFTER completing the DMA associated with that descriptor. This allows all 4 descriptors to be useful.

All the DMA modes are embedded in the descriptor control entry. This allows the descriptors to select different DMA modes while running unattended. For example two descriptors could be DMA reads and the next two could be DMA writes. Or two desciptors could be memory fill operations while the next two descriptors could be Y/C split mode transfers.

It is recommended to load all the descriptors prior to setting the GO bit in the DMA_CTL_CH1 register. Note that the DMA_CTL_CH1 register is only one per DMA engine while this Descriptor Control register appears 4 times for each DMA engine. It is possible to set up one or more descriptors prior to starting DMA. It is important to set the halt bit in the last valid descriptor so that only valid descriptors are processed.

The DMA_CTL_CHX_DY notation is shorthand to describe 4 of these control descriptors per DMA channel. For a complete enumeration of each register's full name refer to Table 7, "DMA Descriptor Address Map," on page 42

TABLE 43. DMA_CTL_CHX_DY Register Format

Bits	Function	Read/ Write	Reset Value
1:0	~~DMA_DCTL_CHF~~ ~~Chroma Half Flag~~s **NOT IMPLEMENTED DO NOT USE** ~~00= Full Pel Vertical Full Pel Horizontal~~ ~~01= Full Pel Vertical Half Pel Horizontal~~ ~~10= Half Pel Vertical Full Pel Horizontal~~ ~~11= Half Pel Vertical Half Pel Horizontal~~	r/w	X
2	DMA_CHROMA_ONLY 0= Follow DMA_DCTL_YC settings 1= Keep Chroma Only, Drop Luma if DMA_DCTL_YC is 00 = Reserved 01 = Reserved 10 = Y/C 4:2:2 -> Y/C 4:2:2 Drop Y, Keep all C 11 = Reserved	r/w	X

TABLE 43. DMA_CTL_CHX_DY Register Format

Bits	Function	Read/ Write	Reset Value
3	DMA_DCTL_HPEN 0= Normal DMA, Ignore HP(9:8) bit settings. 1= Some attempt here to leave the span/stride settings alone whether or not Half-Pel Horiz/Vertical is detected for each axis. Not sure this will work. Better if DMA Descriptors are programmed with correct count!	r/w	X
6:4	DMA_DCTL_LOC DMA location inside of VICE. **NOT IMPLEMENTED but no adverse affect for programming these fields. Address inside of VICE is derived from DMA_VMEM_Y and DMA_VMEM_C descriptor fields.** 000= Data RAM A Width = 64 001= Data RAM B Width = 64 010= Data RAM C Width = 64 011= MSP Instruction RAM Width = 16 100= BSP Instruction RAM Width = 64 101= BSP Table RAM Width = 32 110= BSP Compressed Bits Fifo Width = 16 BSP_OUT_FIFO 0x7000 Width = 32 BSP_IN_FIFO 0x7800 111= DMA TLB RAM Width = 32	r/w	X
7	~~DMA_DCTL_ILV~~ 0= Block Mode **MUST BE SET TO BLOCK MODE** ~~1= Interleave Mode~~	r/w	X
9:8	~~DMA_DCTL_YHF~~ ~~Luma Half Flags~~ **NOT IMPLEMENTED DO NOT USE** ~~00= Full Pel Vertical Full Pel Horizontal~~ ~~01= Full Pel Vertical Half Pel Horizontal~~ ~~10= Half Pel Vertical Full Pel Horizontal~~ ~~11= Half Pel Vertical Half Pel Horizontal~~	r/w	X
11:10	DMA_DCTL_YC 00= Block Mode No Y/C split 01= Y/C 4:2:2 mode to Y/C 4:2:0 split into separate components 10= Y/C 4:2:2 mode to Y/C 4:2:2 split into separate components 11= Y/C 4:2:2 mode to Y only. All Chroma is dropped. Valid in DMA read mode only.	r/w	X
12	DMA_DCTL_FILL 0= Use system memory as one terminal for this transfer 1= Use DMA_DATA register as system memory for this transfer	r/w	X
13	DMA_DCTL_RW 0= DMA Write (Vice Internals -> System Memory) 1= DMA Read (System Memory -> Vice Internals)	r/w	X

TABLE 43. DMA_CTL_CHX_DY Register Format

Bits	Function	Read/ Write	Reset Value
14	DMA_DCTL_SKIP 0= Use this descriptor 1= Do not used this descriptor, go to next descriptor.	r/w	X
15	DMA_DCTL_HALT 0= Do not halt after this descriptor, go to next descriptor when done. 1= Stop DMA after completing this descriptor	r/w	X

2.9.39 DMA_SMEM_HI_CHX_DY - System Memory Upper Address Pointer

Upper 16 bits of the address that the DMA engine uses to point to system memory. This is interpreted as a Virtual Address when the DMA engine has been programmed to use the TLB. It is considered a physical address when the DMA has been programmed to bypass the TLB.I

TABLE 44. DMA_SMEM_HI_CHX_DY Register Format

Bits	Function	Read/ Write	Reset Value
15:0	DMA_SMEM_HI_PT	r/w	XX

2.9.40 DMA_SMEM_LO_CHX_DY - System Memory Lower Address Pointer

Lower 16 bits of the address that the DMA engine uses to point to system memory. This is interpreted as a Virtual Address when the DMA engine has been programmed to use the TLB. It is considered a physical address when the DMA has been programmed to bypass the TLB.

Address is quad word aligned. for 4:2:2 Y/C split modes. See “DMA Programming Restrictions” on page 33.

TABLE 45. DMA_SMEM_LO_CHX_DY Register Format

Bits	Function	Read/ Write	Reset Value
15:0	DMA_SMEM_LO_PT	r/w	XX

2.9.41 DMA_WIDTH_CHX_DY - DMA Descriptor Width

Length of a memory strip in bytes located in system memory. Total number of bytes to transfer per memory strip when Y/C split mode selected is quad-word. See“DMA Programming Restrictions” on page 33.

TABLE 46. DMA_WIDTH_CHX_DY Register Format

Bits	Function	Read/ Write	Reset Value
15-00	Width of strip to be DMA’d to/from system memory. (Bytes)	r/w	XX

2.9.42 DMA_STRIDE_CHX_DY - DMA Descriptor Stride

Number of bytes from start of first memory strip to second memory strip. For most applications this will be the count (in bytes) of the image’s horizontal dimension. For example, a 720 pixel wide image in YCrCb 4:2:2 space would have 2 bytes per pixel so the stride should be set to 1440. Normally 2 byte increments.

Stride must be quad word aligned in Y/C split modes. See “DMA Programming Restrictions” on page 33

TABLE 47. DMA_STRIDE_CHX_DY Register Format

Bits	Function	Read/ Write	Reset Value
15-00	Stride to next strip to be DMA’d to/from system memory. (Bytes)	r/w	XX

2.9.43 DMA_LINES_CHX_DY - DMA Descriptor Lines

Number of lines to DMA

TABLE 48. DMA_LINES_CHX_DY Register Format

Bits	Function	Read/ Write	Reset Value
15-00	Number of lines to DMA	r/w	XX

2.9.44 DMA_VMEM_Y_CHX_DY - Vice Address Y

Starting address (8-byte granularity and 8 byte aligned) of luma channel of DMA. This is an address of a source or destination inside of VICE. In the normal block mode only this address is valid and the C address is ignored.

TABLE 49. DMA_VMEM_Y_CHX_DY Register Format

Bits	Function	Read/ Write	Reset Value
15-00	Starting address for DMA transfer internal to VICE	r/w	XX

2.9.45 DMA_VMEM_C_CHX_DY - Vice Address C

Starting address (8-byte granularity and 8 byte aligned) of chroma channel of DMA. This is an address of a source or destination inside of VICE. In the normal block mode this address is ignored and only the Y address is used.

TABLE 50. DMA_VMEM_C_CHX_DY Register Format

Bits	Function	Read/ Write	Reset Value
15-00	Starting address for DMA transfer internal to VICE	r/w	XX

3.0 System Interface

VICE is implemented on the 64-bit SysAD processor bus. For a complete description of the R4K bus refer to *MIPS Microprocessor R4000 User's Manual*, Chapter 12 (See Bibliography).VICE can respond as a slave to bus transactions initiated by either the Unix processor or the CRIME ASIC. VICE can request mastership of the SysAD bus through handshake signals, with CRIME, which, in turn, requests that the Unix processor release the SysAD bus. Once VICE owns the SysAD bus it can perform block pipelined writes and reads (which is how VICE performs "DMA" transfers). VICE can also perform single read and write transactions as part of DMA. VICE can only access System Memory address space when it is master of the SysAD bus.

Interrupts to the Unix Processor are communicated with a level sensitive signal that is driven by VICE and received by CRIME.

While VICE is implemented on the SysAD bus, special characteristics exist in the CRIME chip's interface with both VICE and the Unix Processor. For application of VICE to other systems which include a SysAD bus, these same features would need to exist in whatever device communicates as the primary SysAD bus interface with the Unix processor. These features are highlighted in the first section below.

The Chip can be reset by the ViceReset_n pin. The MSP, BSP and DMA units internal to VICE can each be independently reset under software control.

The connections between the Unix processor, CRIME and VICE are shown in Figure 7, "Moosehead SysAD Bus processor connections," on page 76

3.1 VICE <-> CRIME SysAD Protocol

Special features in the CRIME ASIC allow the VICE chip to operate on the SysAD bus as both a master and a slave. These features include special signals as well as an address range that allows VICE to respond to the Unix Processor while CRIME assists with handshake. In addition, extensions to the SysCMD block transfer for both reads and writes have been made.

3.1.1 Physical Signals

The signals in Table 51, "VICE <-> CRIME unique connections," on page 74 exist on the VICE chip. There is a set of 3 signals for VICE to request the bus from CRIME, receive the bus from CRIME and then notify CRIME that VICE has released the bus. The VICE ValidIn_n and ViceValidOut_n are signals to indicate valid data cycles between CRIME and VICE. During these type of transfers, VICE will honor the RdRdy_n and WrRdy_n signals that CRIME sends to both the processor and VICE.

ViceWrRdy_n is a special signal that allows VICE to control the flow of write data when the processor is writing VICE address space directly. CRIME uses ViceWrRdy_n to affect the processor WrRdy_n on behalf of VICE for these types of bus transactions.

The signal on VICE called R4ValidIn_n is connected directly to the processor's ValidIn_n pin. As CRIME also drives this same pin on the processor, a protocol for 3-state of the signal allow CRIME and VICE to alternately drive this signal when the processor is performing reads.

A dedicated pin-to-pin connection between VICE and CRIME allows VICE to drive ViceInt_n, a level sensitive interrupt, into CRIME. CRIME passes this signal to the Unix processor through the normal write request protocol. VICE itself does not write registers anywhere in the system and is limited to system memory transactions only, when it is bus master.

Vice has its own reset pin. In the Moosehead system this pin is driven by CRIME to both the VICE chip and the RESET* signal of the MIPS processor to conserve pins on CRIME. Vice is ready 1 clock after the de-assertion of RESET* to accept SysAD transactions. VERIFY THIS in the VHDL

..

TABLE 51. VICE <-> CRIME unique connections

Signal Name	Function
ViceSysRqst_n	SysAD bus request output to CRIME
ViceSysGnt_n	SysAD bus grant input from CRIME
ViceRelease_n	SysAD release output to CRIME
ViceValidIn_n	SysAD valid input from CRIME
ViceValidOut_n	SysAD valid output to CRIME
ViceWrRdy_n	SysAD write ready output to CRIME
R4ValidIn_n	Vice output 3-state. Connected to Unix Processor ValidIn_n. Active when Unix Processor access VICE address space on read/write. Otherwise Unix Processor ValidIn_n is driven by the Crime chip.
Reset_n	Same reset pin that CRIME drives to the Unix Processor. Not really a VICE <-> CRIME unique connection.
ViceInt_n	VICE level sensitive interrupt output to CRIME

3.1.2 Address

No changes from the standard SysAD address bus protocol are implemented by VICE. VICE produces the full 36 bit physical address on the SysAD bus and will respond to 36 bit addresses.

3.1.2.1 VICE address response

VICE responds to a fixed address range to allow the processor direct access to the internal RAM and registers of VICE. This fixed address range is also known to CRIME so that CRIME will not respond to bus transactions to this address space.

The decoding of this address space is kept to 12 bits of address space. This allows the fifos in CRIME and VICE to qualify stores to the command and data fifos with their respective addresses and still meet the 100 MHz (up-to 120MHz?) SysAD bus speed.

The address range that VICE responds to is 0x0 1700 0000 to 0x0 17FF FFF8.

3.1.2.2 Address Conventions

The address conventions of the SysAD bus are upheld by VICE

- Addresses associated with block requests are aligned to doubleword boundaries.
- Doubleword requests set the low-order 3 bits of address to 0.
- Word requests set the low-order 2 bits of address to 0.
- Halfword requests set the low-order bit of address to 0.
- All other requests use the byte address.

3.1.2.3 Data Ordering

VICE always expects data to be returned in sub-block order.

3.1.3 Bytes, Words, Cycles

3.1.3.1 Data Word Size Definition

For clarification, the definition of words and double-words, used throughout this section, is listed in Table 52, "Data Size Name Convention," on page 75.

TABLE 52. Data Size Name Convention

Term	Definition
Byte	8 bits
Halfword	2 bytes
Tribyte	3 bytes
Word	4 bytes
Quintibyte	5 bytes
Sextibyte	6 bytes
Septibyte	7 bytes
Doubleword	8 bytes

3.1.3.2 Words vs. SysAD Block Transfers

A non-block transfer on the SysAD bus can contain 1 valid data cycle. This can contain from 1 to 8 bytes of valid data.

The SysAD bus is 8 bytes wide. A block transfer on the SysAD bus can contain 2, 3 or 4 valid data cycles. This correlates to 4, 6 or 8 words transferred. This correlates to 16, 24 or 32 bytes transferred. These are expected to be the typical cycles for a VICE to CRIME block transfer.

The block transfer protocol allows for 8 valid data cycles. This correlates to 16 words transferred. This correlates to 64 bytes transferred. These cycles may not be implemented in the VICE to CRIME block transfer protocol.

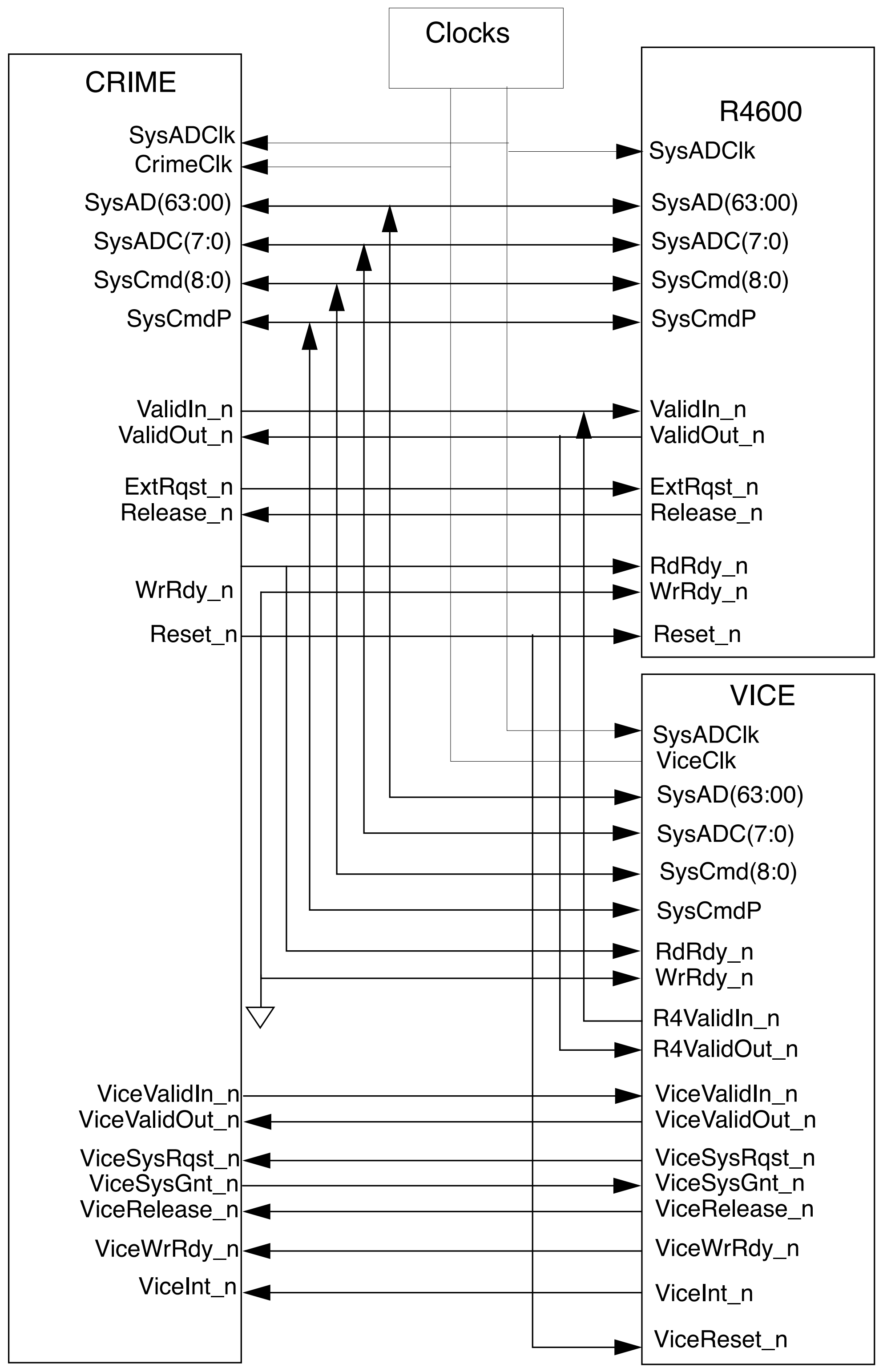


FIGURE 7. Moosehead SysAD Bus processor connections

3.1.4 SysCMD Extensions

Vice follows the command syntax for System Interface Commands and Data Identifiers with one key differences. A block read or write request size of 6 words for Interface Commands is supported in place of the 32 word field used by the R4K family.

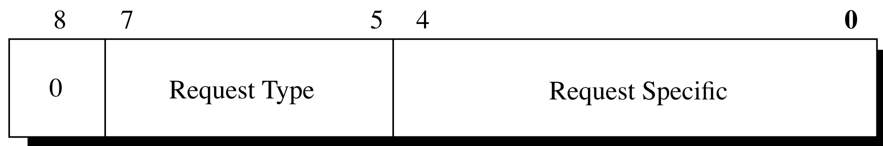


FIGURE 8. System Interface Command Syntax Bit Definition

Of the seven types of System Interface Commands VICE only supports Read and Write Request

TABLE 53. Encoding of SysCmd(7:5) for System Interface Commands

SysCmd(7:5)	R4K, R4400 Command	VICE Command
0	Read Request	Read Request
1	Read-With-Write-Forthcoming Request	Reserved
2	Write Request	Write Request
3	Null Request	Reserved
4	Invalidate Request	Reserved
5	Update Request	Reserved
6	Intervention Request	Reserved
7	Snoop Request	Reserved

SysCmd(4:0) for Read and Write requests are described in the following sections.

3.1.4.1 Read Requests

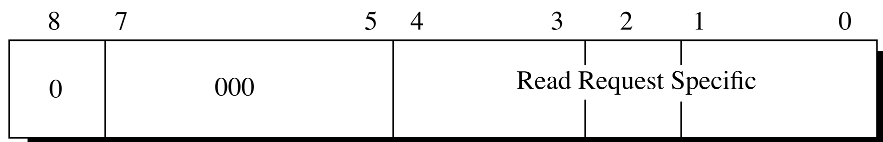


FIGURE 9. Read Request SysCmd Bus Bit Definition

TABLE 54. Encoding of SysCmd(4:3) for Read Requests

SysCmd(4:3)	R4K, R4400 Read Attributes	VICE Read Attributes
0	Coherent block read	Reserved
1	Coherent block read, exclusivity requested	Reserved
2	Noncoherent block read	Noncoherent block read
3	Doubleword, partial doubleword, word, or partial word	Doubleword, partial doubleword, word, or partial word

TABLE 55. Encoding of SysCmd(2:0) for Block Read Requests

SysCmd(2)	R4K, R4400 Link Address Indication	VICE Link Address Ind.
0	Link address not retained	Reserved
1	Link address retained	Reserved
SysCmd(1:0)	**R4K, R4400 Read Block Size**	**VICE Read Block Size**
0	4 words	4 words
1	8 words	8 words
2	16 words	Reserved
3	32 words	6 words * * * Yes 6

TABLE 56. Doubleword, Word, or Partial-word Read Request Data Size Encoding of SysCmd(2:0)

SysCmd(2:0)	R4K, R4400 Read Data Size	VICE Read Data Size
0	1 byte valid (Byte)	1 byte valid (Byte)
1	2 bytes valid (Halfword)	2 bytes valid (Halfword)
2	3 bytes valid (Tribyte)	3 bytes valid (Tribyte)
3	4 bytes valid (Word)	4 bytes valid (Word)
4	5 bytes valid (Quintibyte)	5 bytes valid (Quintibyte)
5	6 bytes valid (Sextibyte)	6 bytes valid (Sextibyte)
6	7 bytes valid (Septibyte)	7 bytes valid (Septibyte)
7	8 bytes valid (Doubleword)	8 bytes valid (Doubleword)

3.1.4.2 Write Requests

Write requests mirror the read requests. See the tables below for details.

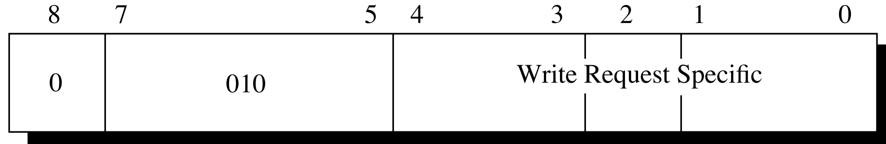


FIGURE 10. Write Request SysCmd Bus Bit Definition

TABLE 57. Encoding of SysCmd(4:3) for Write Requests

SysCmd(4:3)	R4K, R4400 Write Attributes	VICE Write Attributes
0	Reserved	Reserved
1	Reserved	Reserved
2	Block write	Block write
3	Doubleword, partial doubleword, word, or partial word	Doubleword, partial doubleword, word, or partial word

TABLE 58. Encoding of SysCmd(2:0) for Block Write Requests

SysCmd(2)	R4K, R4400 Cache Line Replacement Attr.	VICE (No Cache in VICE)
0	Cache line replaced	Reserved
1	Cache line retained	Reserved
SysCmd(1:0)	**R4K, R4400 Write Block Size**	**VICE Write Block Size**
0	4 words	4 words
1	8 words	8 words
2	16 words	Reserved
3	32 words	6 words * * * Yes 6

TABLE 59. Doubleword, Word, or Partial-word Write Request Data Size Encoding of SysCmd(2:0)

SysCmd(2:0)	R4K, R4400 Write Data Size	VICE Write Data Size
0	1 byte valid (Byte)	1 byte valid (Byte)
1	2 bytes valid (Halfword)	2 bytes valid (Halfword)
2	3 bytes valid (Tribyte)	3 bytes valid (Tribyte)
3	4 bytes valid (Word)	4 bytes valid (Word)
4	5 bytes valid (Quintibyte)	5 bytes valid (Quintibyte)
5	6 bytes valid (Sextibyte)	6 bytes valid (Sextibyte)
6	7 bytes valid (Septibyte)	7 bytes valid (Septibyte)
7	8 bytes valid (Doubleword)	8 bytes valid (Doubleword)

3.1.5 Data Identifiers

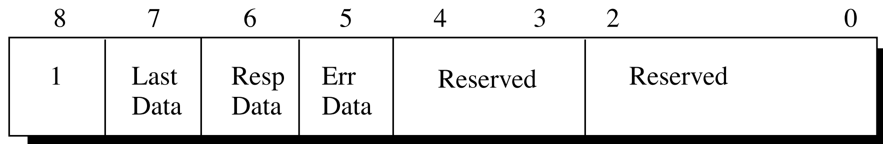


FIGURE 11. Data Identifier SysCmd Bus Bit Definition

SysCmd(8) is set to 1 for all System interface data identifiers. As VICE does not have a cache, it does not utilize coherent data identifiers so SysCmd(2:0) are reserved.

VICE acts as both processor and external agent on the SysAD bus. As such there are four modes of data identifier conditions that involve VICE. The first two modes are when VICE is the external agent and responds to processor read/write sequences. The other two modes are when VICE has acquired the SysAD bus and initiates read/write sequences with the CRIME chip.

3.1.5.1 VICE as External Agent

When the Unix processor initiates a write on the SysAD bus, it will indicate a data identifier encoding of SysCmd(7:3) as shown in Table 60, “Unix Processor Data Identifier Encoding of SysCmd(7:3),” on page 81. If the processor indicates that the data contains a parity error, VICE will indicate the condition with an interrupt, but the write will complete on the SysAD bus (it has to) and VICE will complete the write internal to the VICE chip as well.

When the Unix processor initiates a read request on the SysAD bus, it can handle the combination of data identifiers shown in Table 61, “Vice generated External Data Identifier Encoding of SysCmd(7:3),” on page 81. As VICE does not have on-board parity or ECC for its internal memory, it will always return an

indication that the data is error free so SysCmd(5) is driven to logical 0. For Data Checking Enable, VICE will drive SysCmd(4) according to how the VICE_CFG register is programmed. The power-on default is that VICE will drive SysCmd(4) a logical 1 so that the processor will be asked to not check the data and check bits.

TABLE 60. Unix Processor Data Identifier Encoding of SysCmd(7:3)

SysCmd(7)	Last Data Element Indicator
0	Last Data Element
1	Not the last data element
SysCmd(6)	**Response Data Indication**
0	Data is response data
1	Data is not response data
SysCmd(5)	**Good Data Indication**
0	Data is error free
1	Data is erroneous
SysCmd(4:3)	**Reserved**

TABLE 61. Vice generated External Data Identifier Encoding of SysCmd(7:3)

SysCmd(7)	R4K, R4400 Last Data Element Indicator	VICE Last Data Element Ind.
0	Last Data Element	Last Data Element
1	Not the last data element	Not the last data element
SysCmd(6)	**R4K, R4400 Response Data Indication**	**VICE Response Data Indication**
0	Data is response data	Data is response data
1	Data is not response data	Data is not response data
SysCmd(5)	**R4K, R4400 Good Data Indication**	**VICE Good Data Indication**
0	Data is error free	Vice always indicates error free
1	Data is erroneous	Not Used
SysCmd(4)	**R4K, R4400 Data Checking Enable**	**VICE Data Checking Enable**
0	Check the data and check bits	Controlled with VICE_CFG register
1	Do not check the data and check bits	setting. Default Do Not Check.
SysCmd(3)	**R4K, R4400 Reserved**	**VICE Reserved**

3.1.5.2 VICE as Processor

When VICE is Master of the SysAD bus it behaves as a processor itself. As VICE has no parity or ECC on its internal memory, VICE will always indicate “Good Data” on its DMA write sequences. This is shown inTable 62, “VICE as Processor Data Identifier Encoding of SysCmd(7:3),” on page 82.

With VICE as the processor, the external agent (CRIME) can produce data identifiers identical to those that it intends for the Unix processor. VICE will monitor SysCmd(5) for Good Data Indication and will set the interrupt when erroneous data is indicated. VICE will ignore the Check Data bit from the CRIME chip. See Table 63, “CRIME generated External Data Identifier Encoding of SysCmd(7:3),” on page 82.

TABLE 62. VICE as Processor Data Identifier Encoding of SysCmd(7:3)

SysCmd(7)	**Last Data Element Indicator**
0	Last Data Element
1	Not the last data element
SysCmd(6)	**Response Data Indication**
0	Data is response data
1	Data is not response data
SysCmd(5)	**Good Data Indication**
0	VICE always indicates data is error free
1	Not Used
SysCmd(4:3)	**Reserved**

TABLE 63. CRIME generated External Data Identifier Encoding of SysCmd(7:3)

SysCmd(7)	**R4K, R4400 Last Data Element Indicator**	**VICE Last Data Element Ind.**
0	Last Data Element	Last Data Element
1	Not the last data element	Not the last data element
SysCmd(6)	**R4K, R4400 Response Data Indication**	**VICE Response Data Indication**
0	Data is response data	Data is response data
1	Data is not response data	Data is not response data
SysCmd(5)	**R4K, R4400 Good Data Indication**	**VICE Good Data Indication**
0	Data is error free	Data is error free
1	Data is erroneous	Data is erroneous
SysCmd(4)	**R4K, R4400 Data Checking Enable**	**VICE Data Checking Enable**
0	Check the data and check bits	VICE ignores this bit
1	Do not check the data and check bits	
SysCmd(3)	**R4K, R4400 Reserved**	**VICE Reserved**

3.2 Unix Processor read/write of VICE

VICE (and CRIME) have knowledge of a specific system physical address in which VICE will respond to the Unix Processor and generate the necessary handshake signals. Only non-block read/write access is supported. A block read or write to VICE address space will behave how?

1. Ignore cycle - Unexpected SysAD bus will time out per CRIME?

2. Single transaction - Block Write acts like single write w/ first data word stored, Block-Read returns same data in block? But this requires ValidIn_n and ValidOut_n to work properly, and there must be space in the command and data fifos. On reads, the data should be repeated by VICE, on writes, only the first doubleword on the bus is taken by VICE.

3. Interrupt and Die - Assert ViceInt_n and ignore the bus cycle.

When the Unix Processor issues a write to this VICE address space, it is decoded by both CRIME and VICE to determine that it is a VICE access by the Unix Processor. The Unix Processor obeys the normal WrRdy_n flow control on the pin driven by CRIME. If VICE cannot accept additional writes, then it de-asserts ViceWrRdy_n to the CRIME chip which in turn passes it to the Unix Processor on the WrRdy_n pin with a two flop delay. VICE must have sufficient buffering to accept two potential writes that could occur between the time that VICE de-asserts ViceWrRdy_n and the time the Unix Processor sees WrRdy_n de-asserted from the CRIME chip. VICE must monitor ValidOut_n from the Unix Processor for writes to detect the write cycle.

Reads by the Unix Processor from VICE address space are similar for address decode requirements. VICE however will directly drive its output called R4ValidIn_n which is tied directly to the Unix Processor ValidIn_n input. For this read, CRIME will 3-State its ValidIn_n output and allow VICE to perform the handshake directly with the Unix Processor.VICE must monitor ValidOut_n from the Unix Processor for reads to detect the start of a read cycle.

3.3 VICE read/write of System Memory

VICE must arbitrate for the SysAD bus by asserting ViceSysRqst_n and waiting for assertion of ViceSysGnt_n. VICE will receive ViceSysGnt_n after the CRIME chip has successfully obtained ownership of the bus from the Unix Processor processor. This can take from 4 to 24 Unix Processor PClock cycles.

Once VICE has the bus it may perform a SysAD transaction by using ViceValidIn_n, ViceValidOut_n and ViceRelease_n as communication signals with CRIME. VICE also adheres to the WrRdy_n and RdRdy_n signals from CRIME for these transactions as well.

3.3.1 VICE DMA Read

For a DMA read sequence, VICE will first request the SysAD bus. The handshake for this activity is shown in Figure 17, "VICE Bus Request," on page 90. VICE can then request a sequence of block reads from system memory. These block reads will be aligned on double word (8 byte) boundaries so as to simplify the SysAD bus protocol. Extensions to the Block Read Request Block Size field [SysCMD(1:0)] are detailed in Table 55, "Encoding of SysCmd(2:0) for Block Read Requests," on page 78. This information coupled with the starting address on the SysAD bus will allow for block reads of 8, 16, 24 or 32 bytes. This equates to 2, 4, 6 or 8 words on the SysAD bus. Note that a read of 2 words (8 bytes) is really a single bus access and will follow the non-block protocol format of the SysCMD fields. These requests can be mixed with block requests in order to accomplish efficient DMA transfers.

Each block read request from VICE will reside within a 32 byte, moosehead system, memory word. If the span requested crosses these 32 byte boundaries, VICE will break the request into multiple requests.

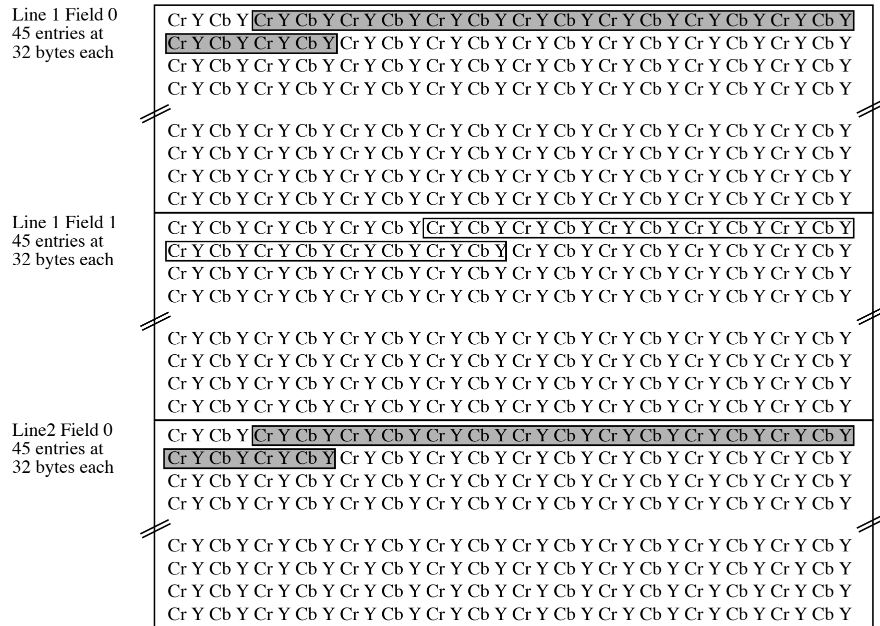
After requesting up to 8 block reads (a maximum of 32 bytes per block read for a total of 256 bytes), VICE will then release the SysAD bus. Once the CRIME memory controller has completed the DMA read from System Memory, it will perform a read response to the VICE chip. Each block read that was requested by VICE will contain its own NEOD marker. Note that each block read response can be from 1 to 4 cycles on the SysAD bus.

Since VICE processors expect to have 4 byte granularity for reads across an 8 byte bus, VICE will be responsible on reads to store a queue of which part of each read response double word is valid (upper, lower or entire). The SysCMD protocol does not include a designation of valid words as part of a read response as only doublewords are defined for block transfers. For Double-word and smaller access, the full SysCMD protocol is followed to allow single byte resolution on both reads and writes. It is not anticipated that VICE will use these sub-word designations as part of the DMA engine.

This protocol is designed to carve rectangular regions from the wide 32 byte System Memory. These rectangular regions can have line skips in them as well. VICE will break spans down into physical System Memory addresses, each requesting no more than 32 bytes. As an example of how the VICE DMA read can read a rectangular region in system memory while skipping lines see Figure 12, "MPEG-2 Field Predictor DMA Read - System Memory to MSP Data RAM," on page 85.

The figure illustrates that a strip of 18 pixels wide data stored as 4:2:2 Y,Cr, Cb data in System Memory can be read Then 17 pixels are selected and these 17 pixels can be horizontally and/or vertically modified to calculate the 16 pixels that represent a half-pixel spacial position in the picture. Finally the DMA engine can separate the pixels into two components of Y and CrCb and place the data into VICE Data RAM.

For more information on the modes supported by the DMA Read operation refer to Section 4.3.1 on page 110.



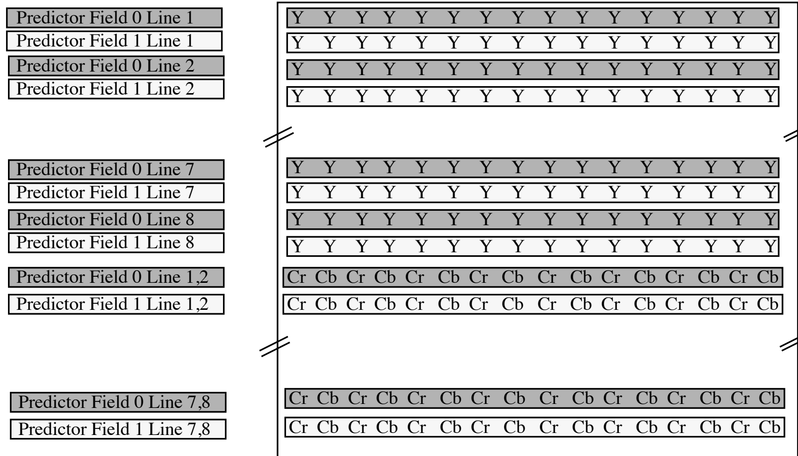


FIGURE 12. MPEG-2 Field Predictor DMA Read - System Memory to MSP Data RAM

3.3.2 VICE DMA Write

For a DMA write sequence, VICE will also arbitrate for the SysAD bus. When owner of the bus, VICE will perform pipelined SysAD block write cycles which will be to the address of the memory locations in system memory. Each address of each block write may be followed by 1 to 4 double words on the SysAD bus.

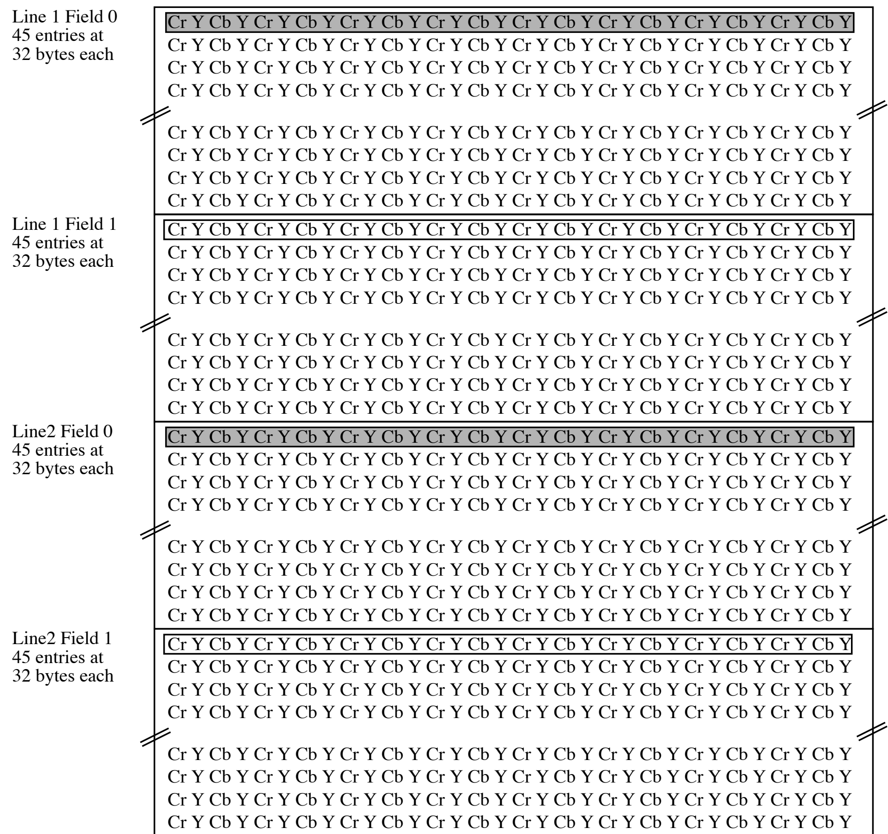
Writes are expected to be packed aligned with system memory so the burden will be on the VICE code to ensure this. The DMA write engine will be able to pull components of luma and color separately and it will be able to deal with 4:2:0 to 4:2:2 interleaving by replication of the Cr and Cb components.

For DMA writes that are not aligned to system memory (not aligned on 8 byte boundaries), the DMA engine will correctly align the data. The performance degradation will be minimized by the DMA engine causing alignment to be re-established at the next 8 byte boundary.

An example of a DMA Write is shown in Figure 12, “MPEG-2 Field Predictor DMA Read - System Memory to MSP Data RAM,” on page 85. The picture information is stored into separate Y and CrCb components in the VICE Data RAM. The DMA engine can merge the Y and CrCb components and interleave them into 4 byte words in the System Memory.

The DMA engine can also perform a DMA write of a packed field in VICE Data RAM and place it within a Frame of data in System Memory. An example of this operation is shown in Figure 12, “MPEG-2 Field Predictor DMA Read - System Memory to MSP Data RAM,” on page 85.

For additional details on the DMA Write modes refer to Section 4.3.1 on page 110.



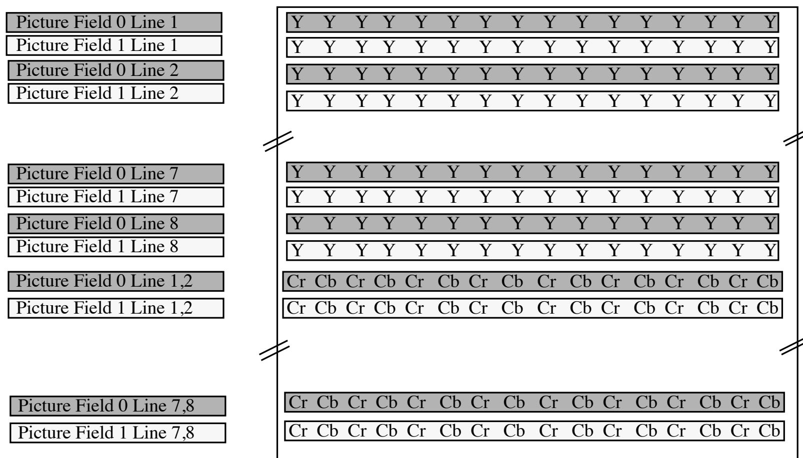
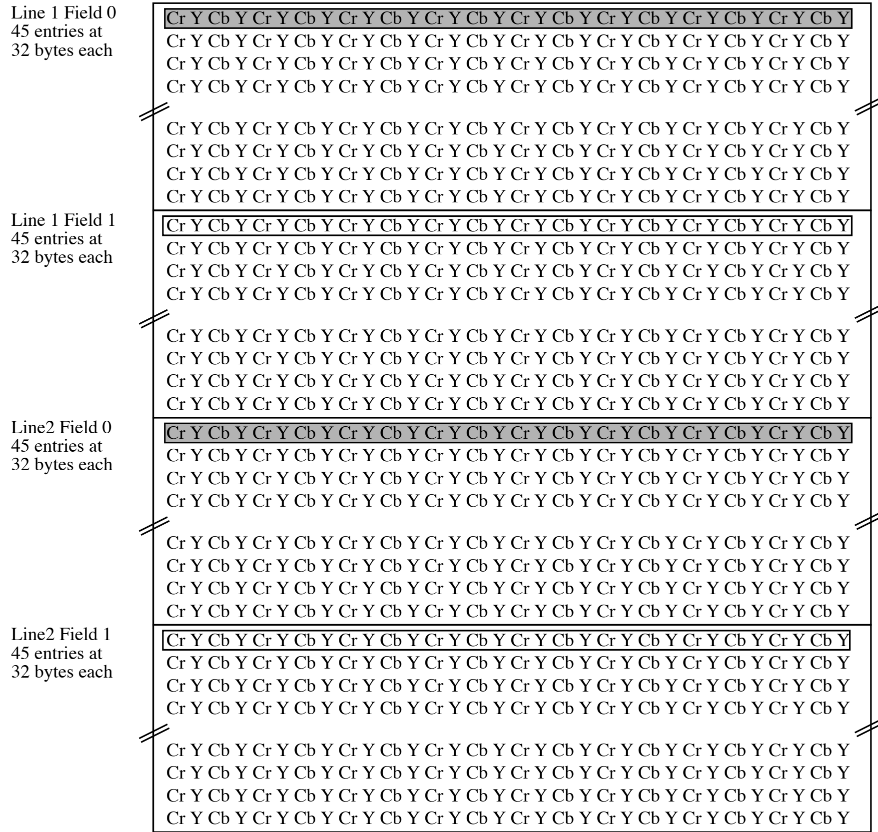


FIGURE 13. MPEG-2 Frame Picture DMA Write - VICE Data RAM to System Memory

Unix System Memory 32 bytes wide



MSP Data RAM Memory16 bytes wide

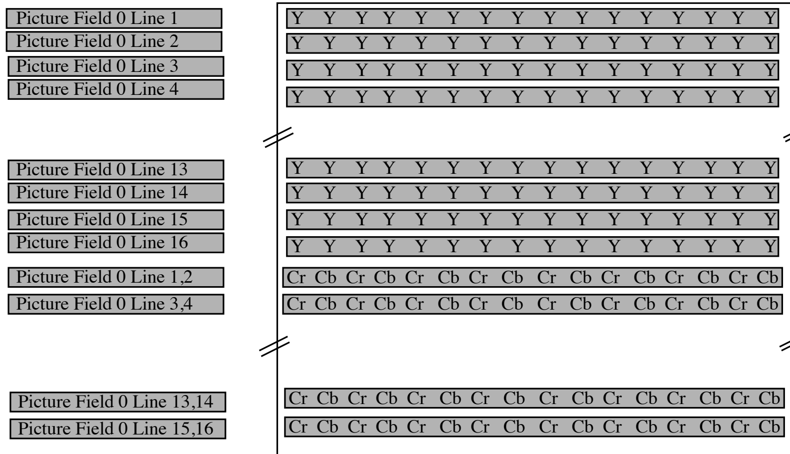


FIGURE 14. MPEG-2 Field Picture DMA Write - MSP Data RAM to System Memory

3.4 VICE SysAD Protocol Timing Diagrams

Protocol sequences for SysAD bus transactions involving VICE are covered in the following pages.

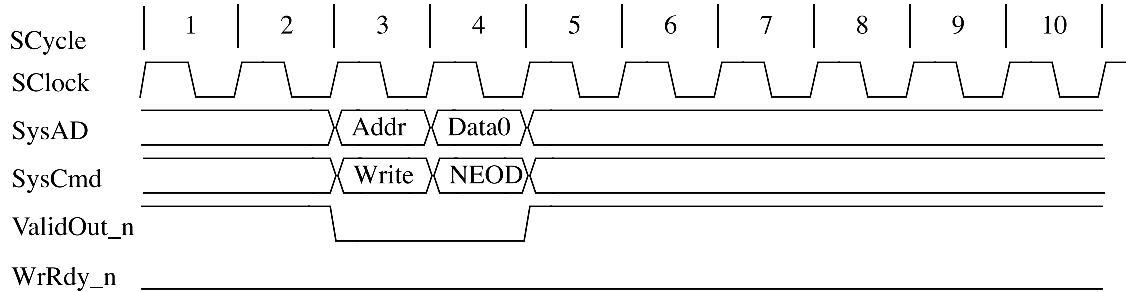


FIGURE 15. Unix Processor Write to VICE Address Space

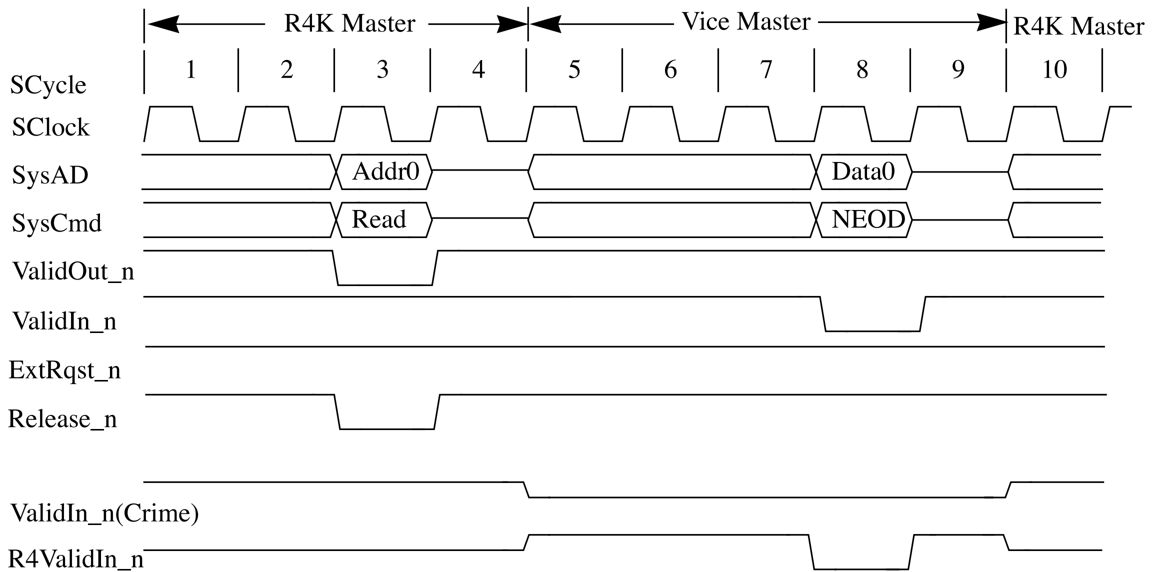


FIGURE 16. Unix Processor Read from VICE Address Space

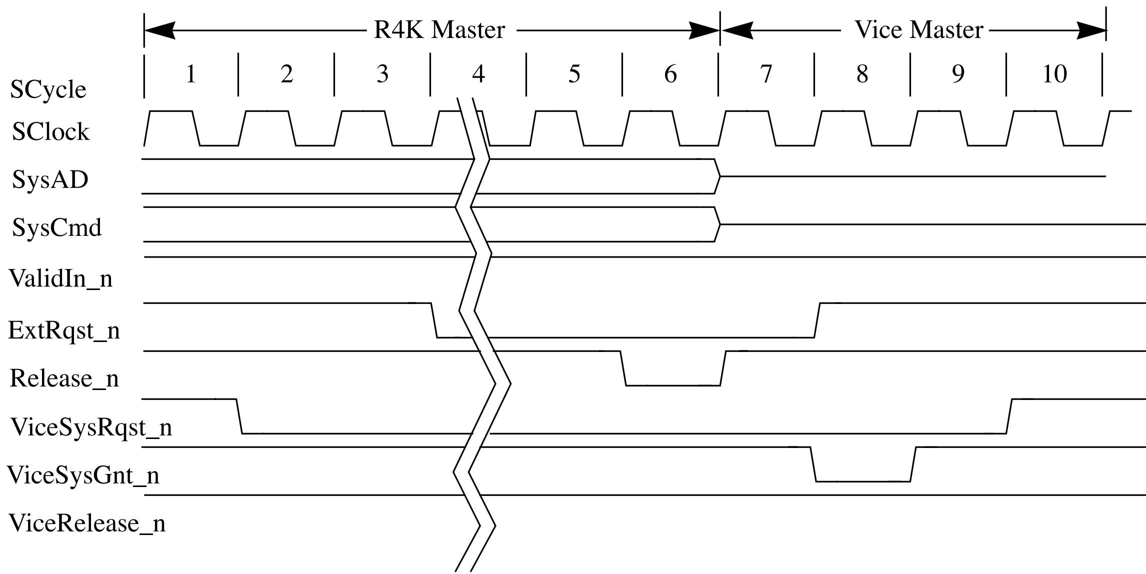


FIGURE 17. VICE Bus Request

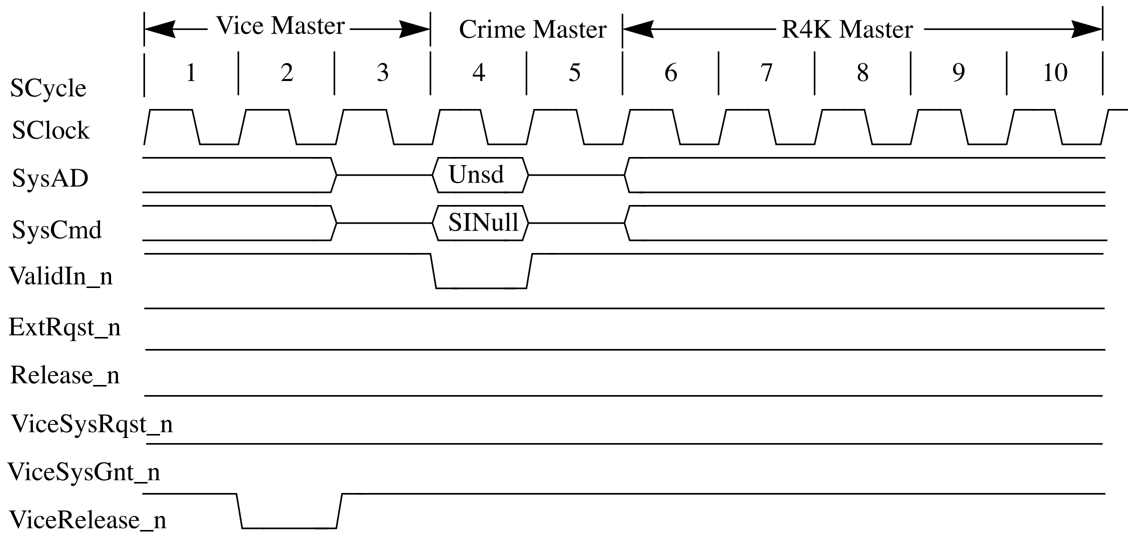


FIGURE 18. VICE Bus Release

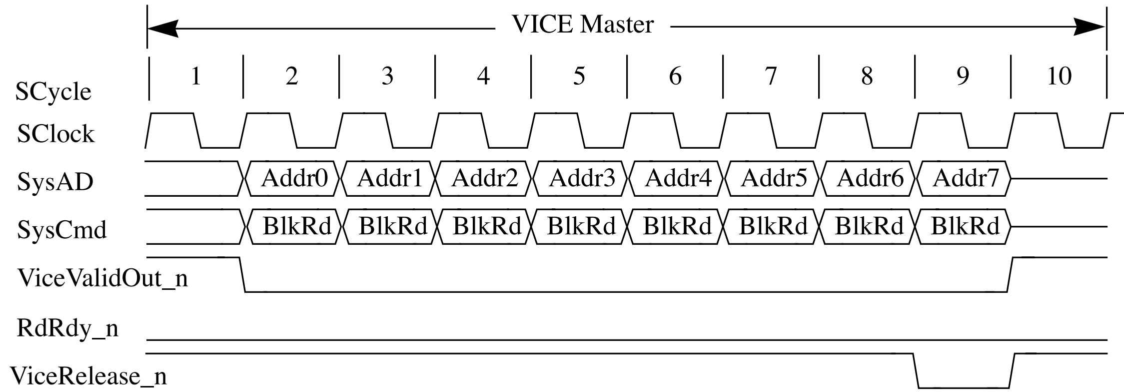


FIGURE 19. VICE DMA read request to CRIME, VICE already owns SysAD bus

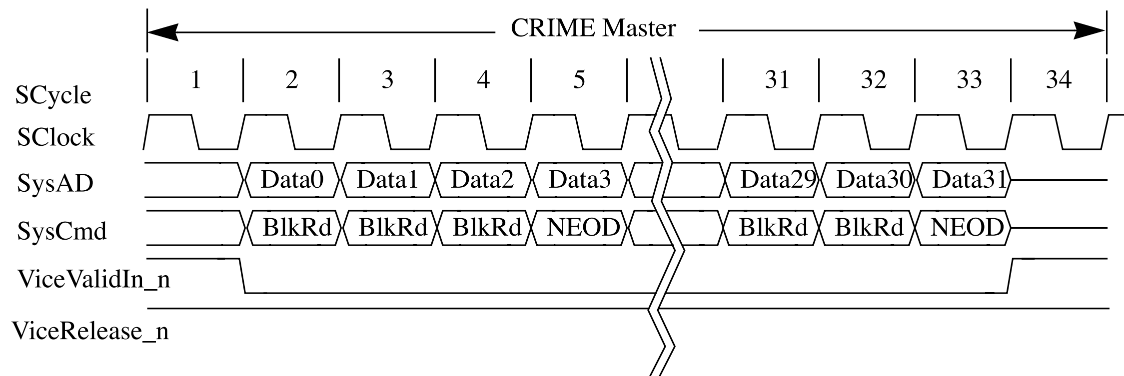


FIGURE 20. VICE-CRIME DMA Read Response, CRIME already owns SysAD bus

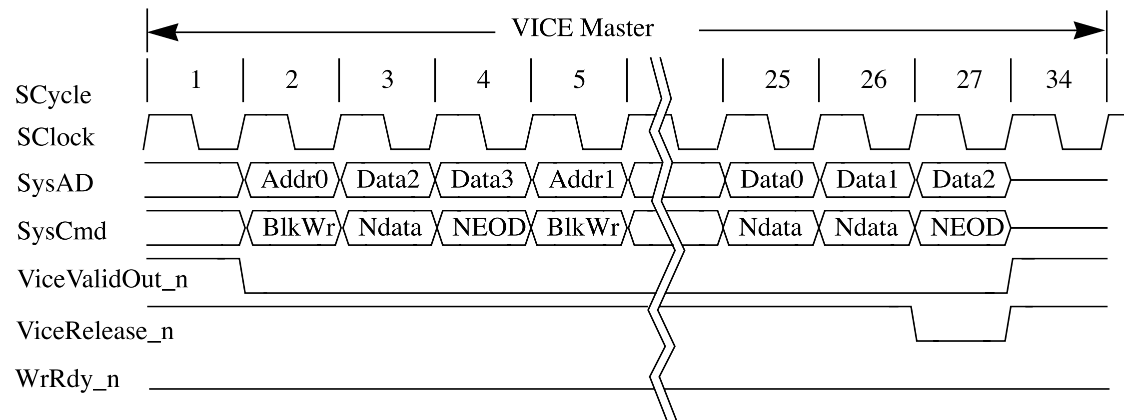


FIGURE 21. VICE - CRIME DMA Block Write

3.5 Clock Interface

VICE will receive SClock. The Unix processor will produce its PClock from SClock in multiples of 2, 3 and 4. This will allow VICE and CRIME to always receive SClock and not have to divide it down. Furthermore, because VICE and CRIME are not generating SClock from MasterClock (R4K and R4600 style) they do not need to phase SClock with the edge of ColdReset_n.

TABLE 64. Unix Processor PClock - SClock Relationship

SClock Provided to Unix Processor	SysAD Transactions	Unix Processor internally synthesized PClock
90 MHz	90 MHz	180 MHz
80 MHz	80 MHz	240 MHz

One backward compatible mode exists in this implementation. For lab bring-up and verification the R4K or R4600 processor can be used as long as that processor's internal PClock to SClock divisor is set at 2. In this mode MasterClock will be fed to the processor and VICE and CRIME. The processor will multiply MasterClock by 2 to produce PClock. PClock will be divided by 2 inside the processor and will match the external MasterClock in phase and frequency. Since VICE and CRIME also receive MasterClock and will use PLLs to create their internal clocks as the same phase and frequency, everything is hunky-dory. Note that any divisor other than 2 for the R4K or R4600 processor will not work in this mode.

Both CRIME and VICE have secondary clocks. This second clock is slower than the SClock and is used to clock most of the processing logic in these two chips. As such, a clock domain is crossed between the SysAD interface Fifos and the remainder of the logic inside both CRIME and VICE.

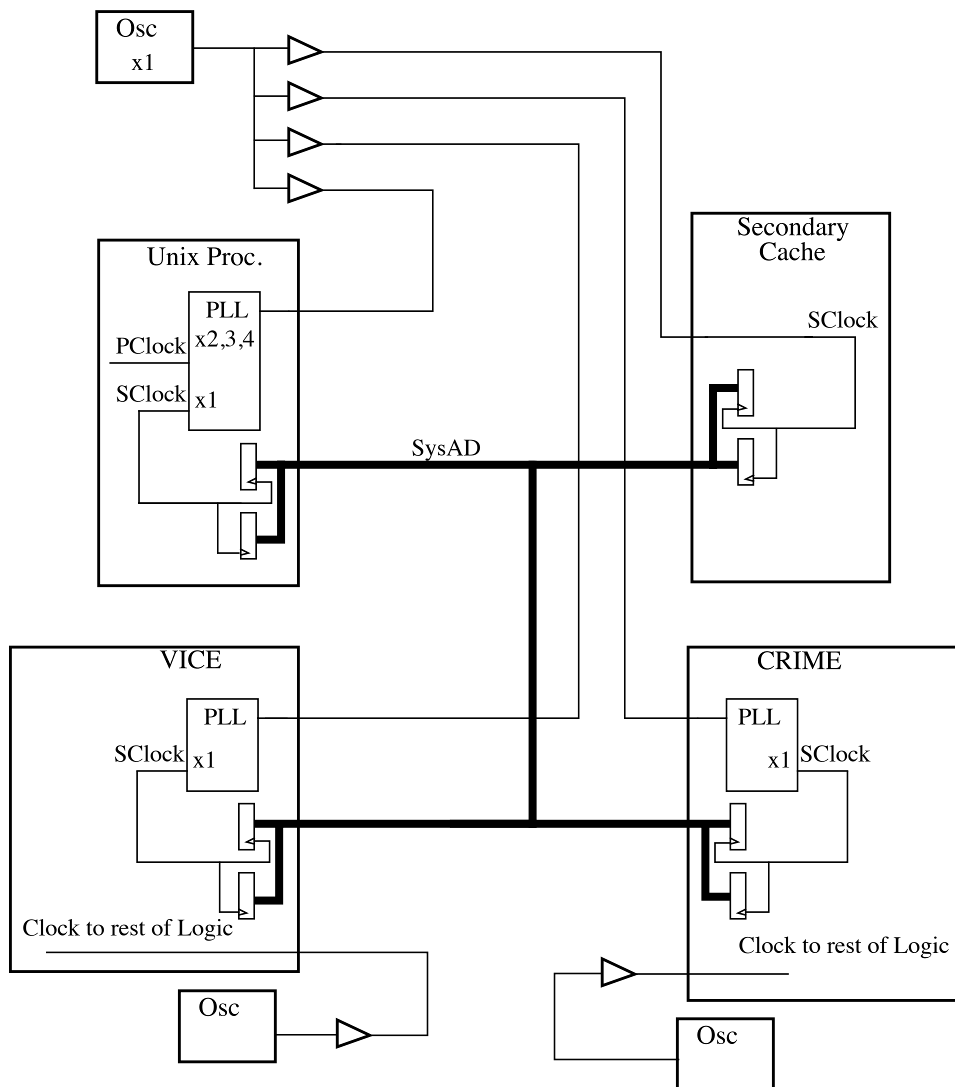


FIGURE 22. SysAD Clock Distribution

3.6 Error Checking

VICE will create even parity for the SysAD and SysCmd bus during all address cycles. Parity will be generated on data values. VICE will set bit4 of the SysCmd field for data identifiers depending on whether data check enable is selected in the VICE_CFG register. The default will be set to no check so that things work OK at power up.

It is not determined at this time if VICE needs to check parity on any received address or data cycles. My inclination is not to support this.

3.7 JTAG Interface

VICE implements the IEEE 1149.1 JTAG Boundary Scan standard with some modifications to handle timing constraints assciated with the SysAD bus. There are 5 pins associated with the JTAG interface which are listed in Table 64.

TABLE 65.

Signal	I/O	Functionality
TDI	input	Test Data In (Serial Input)
TCK	input	Test Clock
TMS	input	Test Mode Select for TAP controller
TRST	input	Asynchronous Reset of TAP controller
TDO	output	Test Data Out (Serial Output)

VICE uses the BS1CONA0 Test Access Port(TAP) library model available from Compass. Refer to Compass Boundary Scan User Guide for details. The VICE JTAG design supports the EXTEST, SAMPLE, BYPASS, and IDCODE instructions. Table 65 lists the supported instructions and the associated opcodes.

TABLE 66.

Instruction	Register	Opcode
EXTEST	BSR	000
SAMPLE	BSR	010
BYPASS	BYPASS	111
IDCODE	ID	001

The VICE BSR register contains 99 register cells. This includes a combination of input, output, bidir, and control cells. The 3 control cells in the BSR chain provide I/O control capability of bidir ports through the JTAG interface. The Boundary Scan Description Language(BSDL) of VICE provides a complete description of the JTAG design including the order of the BSR chain.

4.0 Architectural Description

The VICE chip functionality can be broken into several major blocks. These consist of the Media Signal Processor, the Bitstream Processor, the DMA unit, the Host Interface and the Arbiter.

See Figure 23, “VICE block diagram,” on page 96 for the major blocks of the chip.

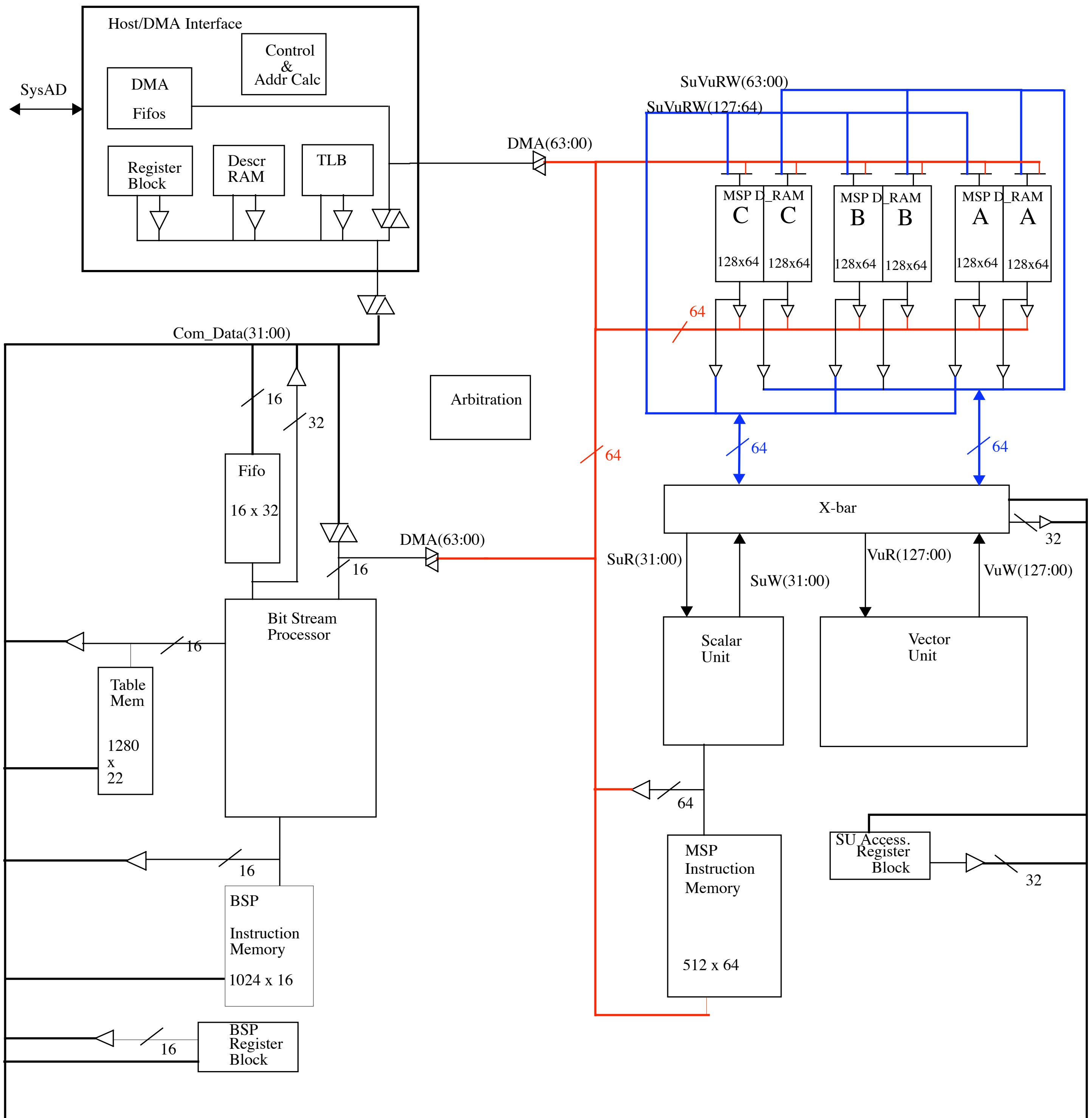


FIGURE 23. VICE block diagram

4.1 Host Interface

The VICE chip communicates with other resources in the workstation through a MIPS R4K like SysAD bus connection. Chapter 3 contains details of this protocol and the deviations from the R4K standard. VICE acts as both a processor AND an external agent.

The Host Interface (the term SysAD Interface is used interchangeably in this document) consists of high speed fifos to capture and produce command and data cycles on the SysAD Interface. Special synchronization techniques are employed to deal with the difference in speed between the SysAD Interface (~100 MHz) and the internal VICE clock (66 MHz) to maintain valid fifo flags.

4.1.1 Host Access

The Unix Processor can perform single write/read accesses to VICE address space. VICE will internally arbitrate between its internal processors (MSP, BSP, DMA engine) accessing registers and the Unix Processor accessing the internal VICE registers.

4.1.1.1 Host Write to VICE internals

The Wrdy_n signal needs to come on when there is still some number of entries in the command/data fifo to allow for the 2 clock delay through VICE and any uncertainty in the on-board VICE flags that deal with the 100 to 66 MHz synchronization. This may be as high as 6 entries.

Writes from the Unix processor can come in while a DMA write is being queued up, while a DMA read is being queued up or while VICE is waiting for a DMA read response from a “launched” DMA.

As the command and write data buffer is being emptied, the SysAD Interface decodes the address and requests a cycle on the Common Bus internal to VICE so that the write will complete. No provisions are made to watch for a Unix processor read to this same internal VICE location so that coherency is maintained. However, this can be controlled because read/write commands use the same fifo and are retired in order.

4.1.1.2 Host Read from VICE internals

Any outstanding reads by the Unix processor will be satisfied before VICE requests the bus for a DMA transaction. This is important for bus ownership control between the processor, CRIME and VICE.

Reads must be serviced to the Unix Processor when VICE is preparing to perform a DMA read or write and when VICE is expecting a DMA read response from CRIME.

4.1.2 DMA

VICE initiates block write/read transfers between internal VICE memory resources and the CRIME chip which in turn accesses workstation system memory. VICE is not allowed to access system I/O address space.

4.1.2.1 DMA Write to System Memory

A sequence of write commands and data are queued up in a fifo inside of the SysAD Interface. A non-empty fifo in the SysAD interface causes a SysAD bus request, and writes are sent to the CRIME chip in sets of up to 10. The address buffer in CRIME is limited to 16 entries. The data buffer is limited to 256 bytes. This should guarantee that Wrdy_n does not have to be asserted by CRIME during these pipelined block writes, however the protocol will be honored regardless. At the end of this set of addresses (up to 8), the VICE chip

will release the SysAD bus. CRIME can then refuse to grant it again until it's buffer has sufficient space to handle the next set of addresses and data.

4.1.2.2 DMA Read from System Memory

DMA reads are similar except that only the command fifo in the SysAD Interface is loaded by the DMA unit. The SysAD interface will only request the bus from CRIME if the SysAD read buffer is ready to accept a full 256 bytes of data that this read may produce. Should the SysAD interface, internal to VICE, wait for the DMA to tell it "go" so that the full 8 to 10 requests are in the fifo and ready to go onto the SysAD bus?

4.1.3 Host/DMA interaction

When a VICE initiated DMA is in progress, the Host is still able to access the internal state of the VICE chip. The Host can perform a read or write of some internal resource of VICE only when the Unix Processor owns the bus.

In the case of DMA writes by VICE, VICE has requested the bus and is writing Address/Data pairs to CRIME. VICE can write as many as 8 block transfer cycles without releasing the SysAD bus. This is a total of 40 SysAD clock cycles. When complete, VICE will release the bus.The Unix processor may subsequently perform single read or write cycles between these atomic block write bursts from the VICE chip. This particular case requires separate buffering between the DMA channel and Unix processor transactions so that read responses produce the correct data to the Unix Processor and do not get mixed with the buffer in VICE that is writing data to the CRIME chip.

In the case of DMA reads by VICE, Response data from CRIME must be differentiated from write data coming from the Unix Processor. This is indicated by the Data Identifier provided by CRIME to VICE during read response cycles.

4.2 Arbiter/Internal Bus Sharing

There are 4 requesting devices internal to VICE that the arbiter handles; the Host interface, the DMA unit, the Bitstream Processor and the Media Signal Processor (Scalar Unit). At any time any of these devices might want to perform a read or write transaction of some resource inside the VICE chip.

The arbiter is of fixed priority as follows:

1. Media Signal Processor (MSP)
2. Host Interface
3. Bit Stream Processor (BSP)
4. DMA engines

This priority is designed to support the Media Signal Processor which is not designed to stall under normal operation. In addition it is designed to satisfy read/writes by the Unix Processor when they collide with the DMA engine's use of the internal bus.

There are three buses internal to VICE that need be arbited. The MSP has a dedicated address and data bus to access the three banks of Data RAM. This is referred to as the SuVuRW data bus. The BSP and Host/DMA blocks share a 64 bit data bus that can also be used to access the three banks of Data RAM. This is referred to as the DMA bus. The arbiter controls access to the Data RAM deciding between the SuVuRW bus and the DMA bus. The arbiter also controls access to the DMA bus by the Bit Stream Processor (BSP) and the Host DMA Interface (H/DMA). The third bus inside of vice is known as the common bus. It is the path to all other memory and registers inside of VICE except the Data RAM and MSP Instruction RAM.

The goal of these three buses is to allow two clients access to the Data RAM simultaneously (if the requested banks are different). A further goal is to allow access by any client to the common bus while still allowing transactions to occur on one or more of the two memory buses.

An overview of the address and control paths inside of VICE is covered in Figure 24, “Internal Address/Control Flow,” on page 102.

A list of datapaths is given in Table 67, “VICE Datapath Flow,” on page 101. In the table is a list of any data bus width mis-matches. In the case of Unix Processor access through the VICE Host Interface, it is possible to force alignment in most cases. However for DMA to/from these same mismatched data paths, alignment must be performed in hardware as the DMA data will be packed.

This table also helps to highlight where stacking and de-stacking logic should be placed. It also helps to show that from a data flow viewpoint, the DMA engine data path is readily extended to reside inside the Host Interface Block. The term “Force Alignment” in the table indicates that the Memory/Register expects the source processor to align the data on it's 16/32/64/128 bit wide bus and provide the correct byte enables to store the data correctly. The host interface will be able to perform 64 bit write/read to the 64 bit internal common bus. For any resource that is 64 bits wide this is an exact match. For any resource smaller than 64 bits, such as registers and various BSP rams, the smaller data width will still take the full 64 bit address range (i.e. least significant address bits will be dropped). This forces the processor to correctly align the data for the register/memory target.

The Bitstream Processor is a special case in that it has a 16 bit wide data path. The transceivers from the BSP will drive all 32 bits of the common bus with the address from the BSP used to control valid byte information.

4.2.1 Rules for Access to Internal VICE buses

MSP will be able to access its resources inside of VICE without waiting. The arbiter will look at the instruction stream from the MSP to detect load/store instructions. Other devices that may be using a particular resource, will be suspended in order to service the MSP. The arbiter qualifies all load/store operations to the Data RAMs (A,B or C). MSP Co-processor Move instructions are used to access the Register Block, the DMA descriptor RAM and the mailboxes to the Bit Stream Processor. The MSP Co-processor instructions use the internal VICE Common Bus which is shared between the MSP, BSP, Host and DMA blocks.

4.2.1.1 MSP Access

The MSP can access the Data RAM, bank A, B or C. It uses it's Scalar Unit bus and the X-bar to do this. If the MSP is accessing a memory that is not being used by the BSP, DMA or Host, then these devices are not required to wait. If the MSP collides with any of these resources, they are forced to suspend their access to allow the MSP to continue with no stalls. Any access by the MSP onto the common bus to access the DMA descriptor RAM or the VICE registers will also preempt the BSP, DMA or Host interfaces on the common bus but should allow them to continue if they are using the DMA memory bus to access Data RAM.

4.2.1.2 Host Access

The Host has second priority behind the MSP. This is to keep the SysAD bus as efficient as possible. The Host can access the Data RAM bank A, B or C at any time. It can access the Register Block at any time. It can write to any of the following RAMs at any time as the write port to that resource is either unused by its respective processor or is shared between the DMA engine and the Host Interface which have knowledge of each others state.

The Host can access the MSP Instruction RAM only after halting the MSP.

4.2.1.3 BSP Access

The BSP owns the read port of its Instruction RAM, its Table RAM and its Decode Fifo. That resource can be revoked with a control bit that also assumes the BSP is stalled. This is anticipated to be a context switch mode or diagnostic.

The BSP can access Data RAM A, B or C through the arbiter and the DMA bus. If the Scalar Unit or Host is not presently using that resource there is no waiting. If DMA is using that resource, DMA will be suspended to allow the BSP to use the DMA bus for its transaction

The BSP can access the DMA Descriptor RAM for writes and the Register Block for reads or writes. If the common bus is utilized the BSP will have to wait to access that resource.

4.2.1.4 DMA Access

The DMA block can access Data RAM A, B or C through the arbiter and the DMA bus. It can access the read port of the DMA Descriptor RAM and the TLB without dynamic arbitration. The DMA can be prevented from accessing the read port of those memories by a mode bit that will allow the host to access them for diagnostic reasons.

The DMA can load any of the memories in the VICE chip. Data RAM A, B or C are loaded using the DMA bus. The MSP Instruction RAM is accessed using the DMA Bus. All other memories are accessed using the common bus.

TABLE 67. VICE Datapath Flow

Data Transfer Path		Data Alignment		Where is steering?
Processor	Memory/Register	Single Transfer	DMA	
Host I/F	BSP Table Memory	Force Alignment	32 to 16	Host I/F
Host I/F	BSP Instruction Memory	Force Alignment	32 to 16	Host I/F
Host I/F	BSP FIFO	Force Alignment	32 to 32	Host I/F
Host I/F	MSP Instruction Memory	Aligned	64 to 64	At MSP Instr Mem-ory
Host I/F	MSP Data Memory - Aligned	Force Alignment	64 to 128	Host I/F
Host I/F	MSP Data Memory - Y/C Mode	Not supported	don't ask	Host I/F
Host I/F	MSP - SU/VU Register File	Not supported	Not supported	
Host I/F	VICE Register Block	Force Alignment	Not supported	Not needed
Host I/F	DMA Descriptor Memory	Force Alignment	Not supported	
Host I/F	DMA TLB	Force Alignment	32 to 32	Host I/F
BSP	MSP Data Memory	Force Alignment	N/A	BSP drivers?
BSP	VICE Register Block	16 to 32	Not supported	BSP or REG?
BSP	DMA Descriptor Memory	16 to 16	Not supported	BSP or DMA Descr Block.
MSP	MSP Data Memory	32 to 128 & 128 to 128 unaligned	N/A	X-Bar
MSP-SU	VICE Register Block	Aligned	Not supported	Not needed
MSP-SU	DMA Descriptor Memory	32 to 16	Not supported	Not needed

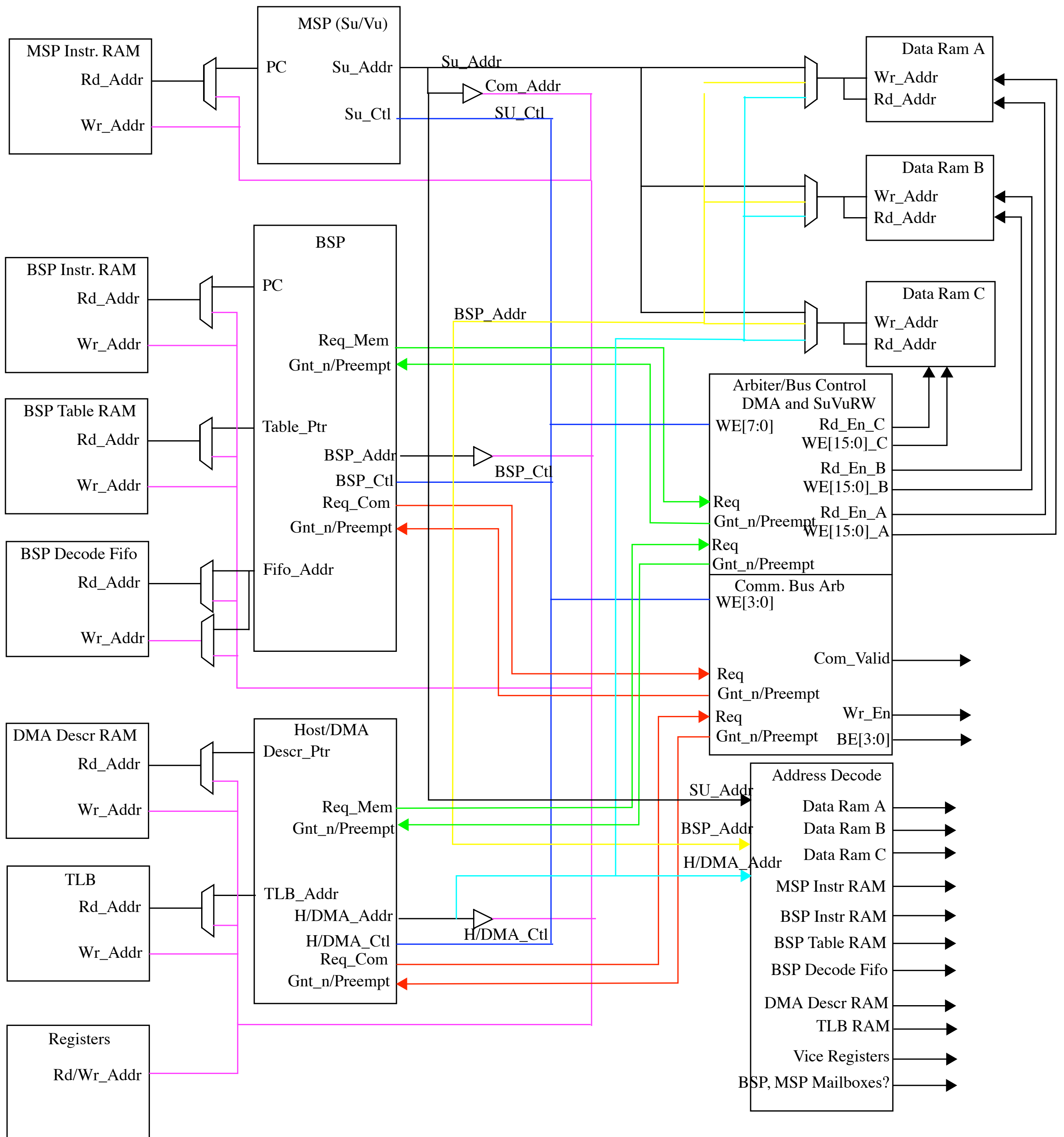


FIGURE 24. Internal Address/Control Flow

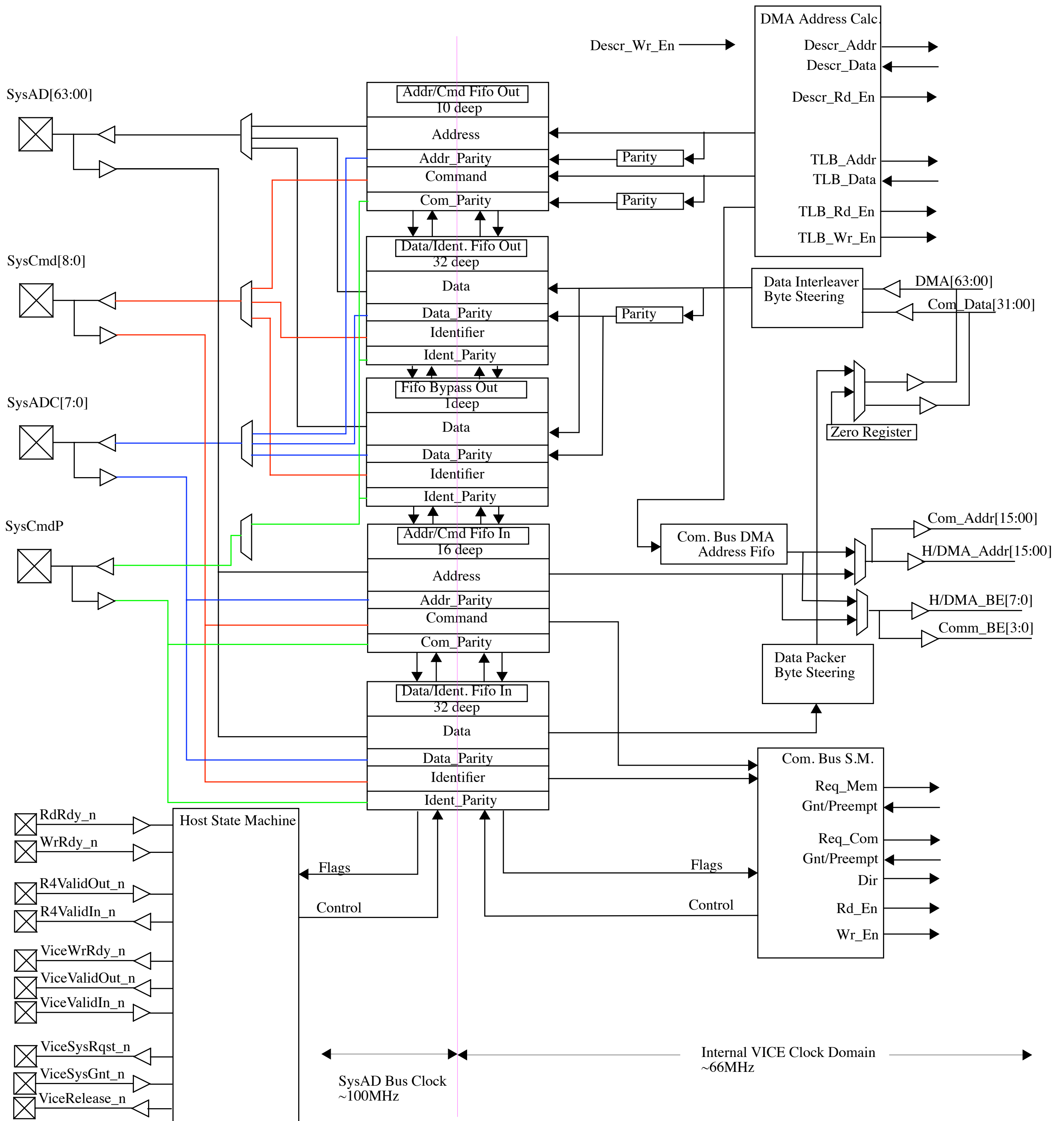


FIGURE 25. Host/DMA Block Diagram

4.2.2 Common Bus Arbiter Protocol

A device wishing to use the internal Common bus of VICE asserts it's Common bus request line to the arbiter. There is a separate request line for the BSP and the H/DMA blocks. The arbiter will grant a device the bus if the Scalar Unit is not using the common bus. The request can be preempted at any time as the Scalar Unit may issue a common bus cycle while the device that owns the bus is attempting to perform a cycle.

The Scalar Unit does not need a grant from the Common bus. The arbiter will ensure the bus is available to the MSP as demanded.

4.2.2.1 Signal List

The MSP, BSP and H/DMA blocks will have a request and a grant signal to arbitrate for the Common Bus. The BSP and H/DMA blocks will have a grant_preempt signal from the arbiter.

TABLE 68. List of Signals on Common Bus

Signal Name	Type for Master	Type for Slave	Signal Description
Com_Addr(15:00)	Output	Input	Common Bus Address, Available 1 Cycle before the Data Bus Information is Valid.
Com_Valid	Input	Input	Common Bus Valid. Address Bus contains valid information. Looks like this should be a logical "or" of all the Common Bus Address OE signals.
Com_Data(31:0)	I/O	I/O	Common Bus Data
Com_WE(3:0)	Output	Input	Common Bus Write Enables, Used to differentiate read/write and serves as byte enables on write cycles.
BSP_COM_RQ	Output	N/A	Bit Stream Processor Common Bus Request
BSP_COM_GNT_PRE	Input	N/A	Bit Stream Processor Common Bus Grant and Pre-empt.
H_DMA_COM_RQ	Output	N/A	Host/DMA Common Bus Request
H_DMA_COM_GNT_PRE	Input	N/A	Host/DMA Common Bus Grant and Pre-empt.
MSP_COM_RQ	Output	N/A	Media Signal Processor Common Bus Request

4.2.2.2 SU - BSP transactions on Common Bus

Each slave device on the common bus is responsible for decoding its address and driving its Data Output Enable signal. Each master device on the common bus is responsible for driving its Address onto the common bus. The WE signals will be muxed onto the common bus by the arbiter to ensure glitchless operation of these control signals.

A transaction by a master on the Common bus is basically a three clock cycle. The first clock requires the master to assert its request and wait for a grant. The cycle after a grant is received, the Master drives its address onto the Common Bus along with the Address_OE signal for its buffers. Slave devices will register the address and use it in the next cycle to select the register or memory location that corresponds to that

address AND to drive the Output Enable (In the case of a bus read). Read is determined by lack of any asserted WE signals.

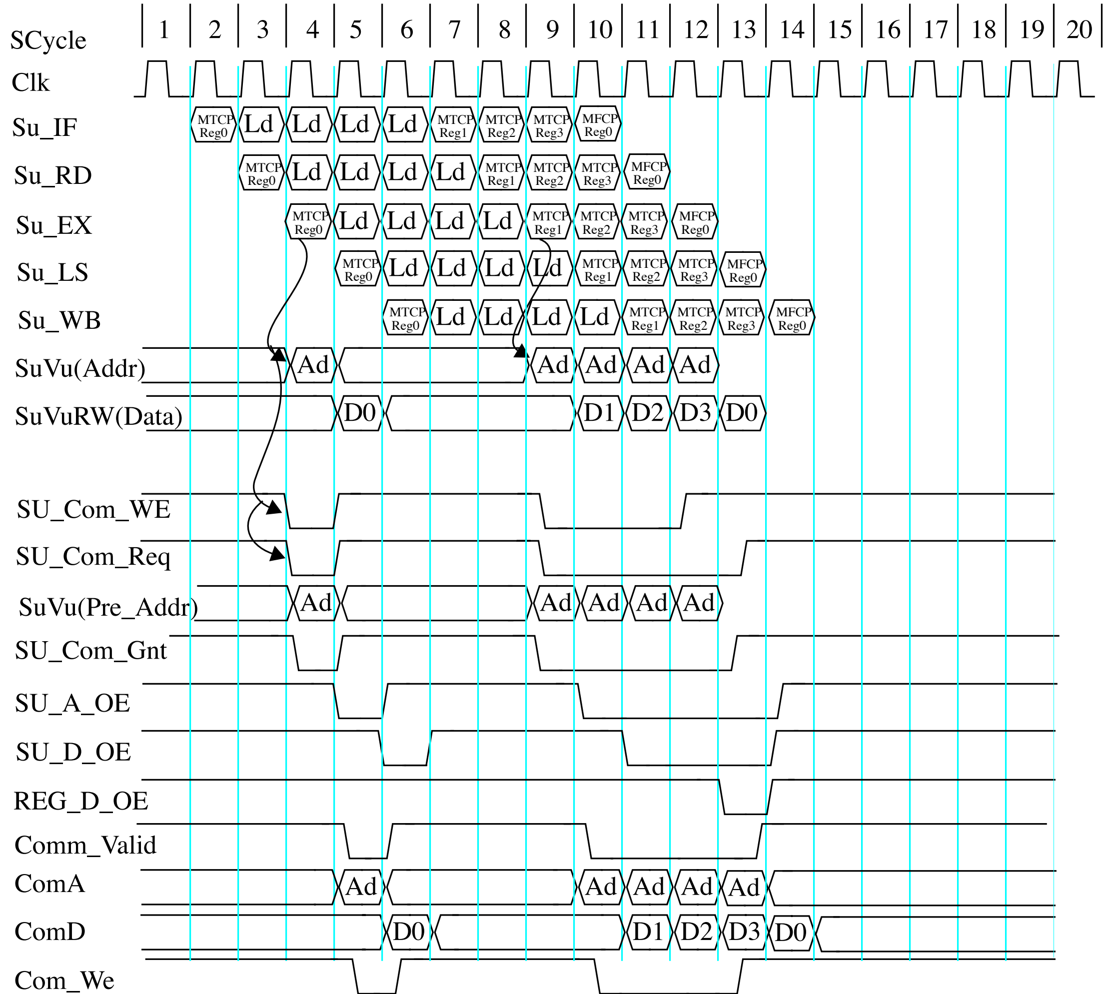


FIGURE 26. MSP (Scalar Unit) Access on Common Bus

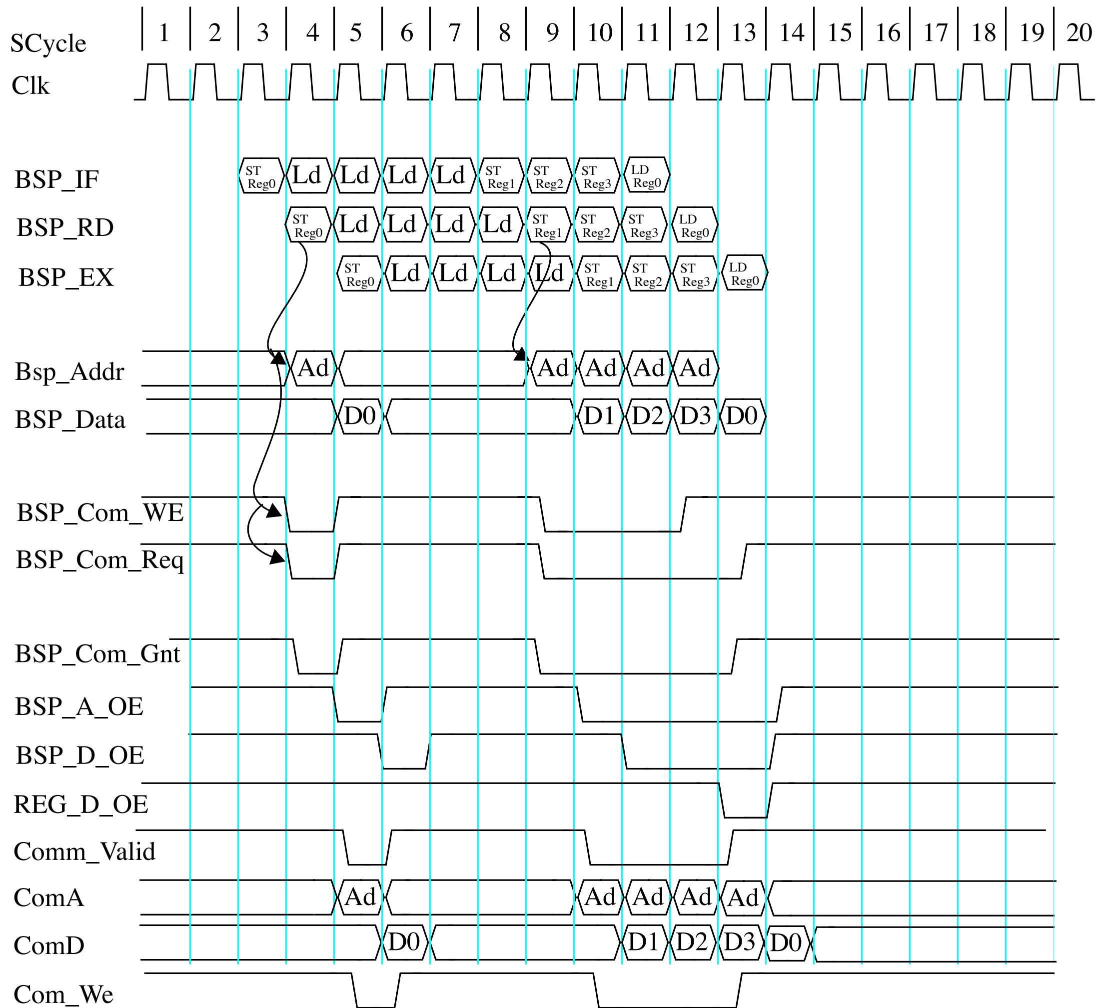


FIGURE 27. Bit Stream Processor Access on Common Bus

4.2.3 DMA Bus Arbiter Protocol

The BSP and H/DMA blocks of VICE share the DMA bus as a pathway to access the internal Data RAM banks of the chip. A device wishing to use this DMA bus of VICE asserts it's DMA bus request line to the arbiter. The arbiter will grant a device the bus if the Scalar Unit is not using the Data RAM bank to be accessed, or if the Scalar Unit has been denied access to the Data RAM bank by use of the MSP_CFG register. The arbiter will determine which bank is to be accessed because of two address bits that will be sent along with the request to indicate which bank of RAM is requested. Once the bus is granted, the request can be preempted at any time as the Scalar Unit may issue a Load/Store to the same RAM bank that is granted to the BSP or H/DMA blocks.

For an MPEG decode application, the Scalar Unit will be assigned memory C permanently and will alternately be assigned memory A or B. Since the DMA will be accessing the other bank of A or B that the Scalar Unit is NOT using, we will be able to get significant memory bandwidth overlap. The Bit-Stream Processor

will be able to access memory A, B or C and the arbiter will delay it based on conflicts with the Scalar Unit. If the Bit-Stream Processor and the DMA are both trying to access a memory not assigned to the Scalar Unit, they will not have to wait for the Scalar Unit, however they will have to wait for each other as they share the DMA bus for data.

4.2.3.1 Signal List

The DMA bus resembles the Common Bus. There are however two additional bits per requestor to allow the arbiter to predict which Data RAM is to be accessed. Because the data bus is 64 bits on the DMA bus, there are 8 WE lines and 64 data lives vs. the 4 WE and 32 data lines for the Common Bus. The address and WE lines on the DMA bus are dedicated lines for each accessing processor (MSP, BSP, H_DMA) that are multiplexed to the address and WE lines of the Data RAM.

The arbiter for the DMA bus chooses between the BSP and the H/DMA units. At the Data RAM itself, there is an arbiter that chooses between the MSP and the DMA bus. This second level of arbitration requires that the MSP provide a Request and a BANK(1:0) ID. These signals are not listed below as part of the DMA bus.

TABLE 69. List of Signals on DMA Bus

Signal Name	Type for Master	Type for Slave	Signal Description
DMA_Addr(15:00)	Output	Input	DMA Bus Address, Available 1 Cycle before the Data Bus Information is Valid.
DMA_Data(63:0)	I/O	I/O	DMA Bus Data
DMA_WE(7:0)	Output	Input	DMA Bus Write Enables, Used to differentiate read/write and serves as byte enables on write cycles.
BSP_DMA_RQ	Output	N/A	Bit Stream Processor DMA Bus Request
BSP_DMA_BANK(1:0)	Output	N/A	BSP Data RAM Bank Request 00- Bank A 01- Bank B 10- Bank C 11- MSP IRAM
BSP_DMA_GNT_PRE	Input	N/A	Bit Stream Processor DMA Bus Grant and Pre-empt.
H_DMA_DMA_RQ	Output	N/A	Host/DMA DMA Bus Request
H_DMA_BANK(1:0)	Output	N/A	Host/DMA Data RAM Bank Request 00- Bank A 01- Bank B 10- Bank C 11- MSP IRAM
H_DMA_DMA_GNT_PRE	Input	N/A	Host/DMA DMA Bus Grant and Pre-empt.

4.2.3.2 BSP Access on DMA Bus

The Bitstream Processor access on the DMA Bus is shown in Figure 28, “Bit Stream Processor Access on DMA Bus,” on page 108.

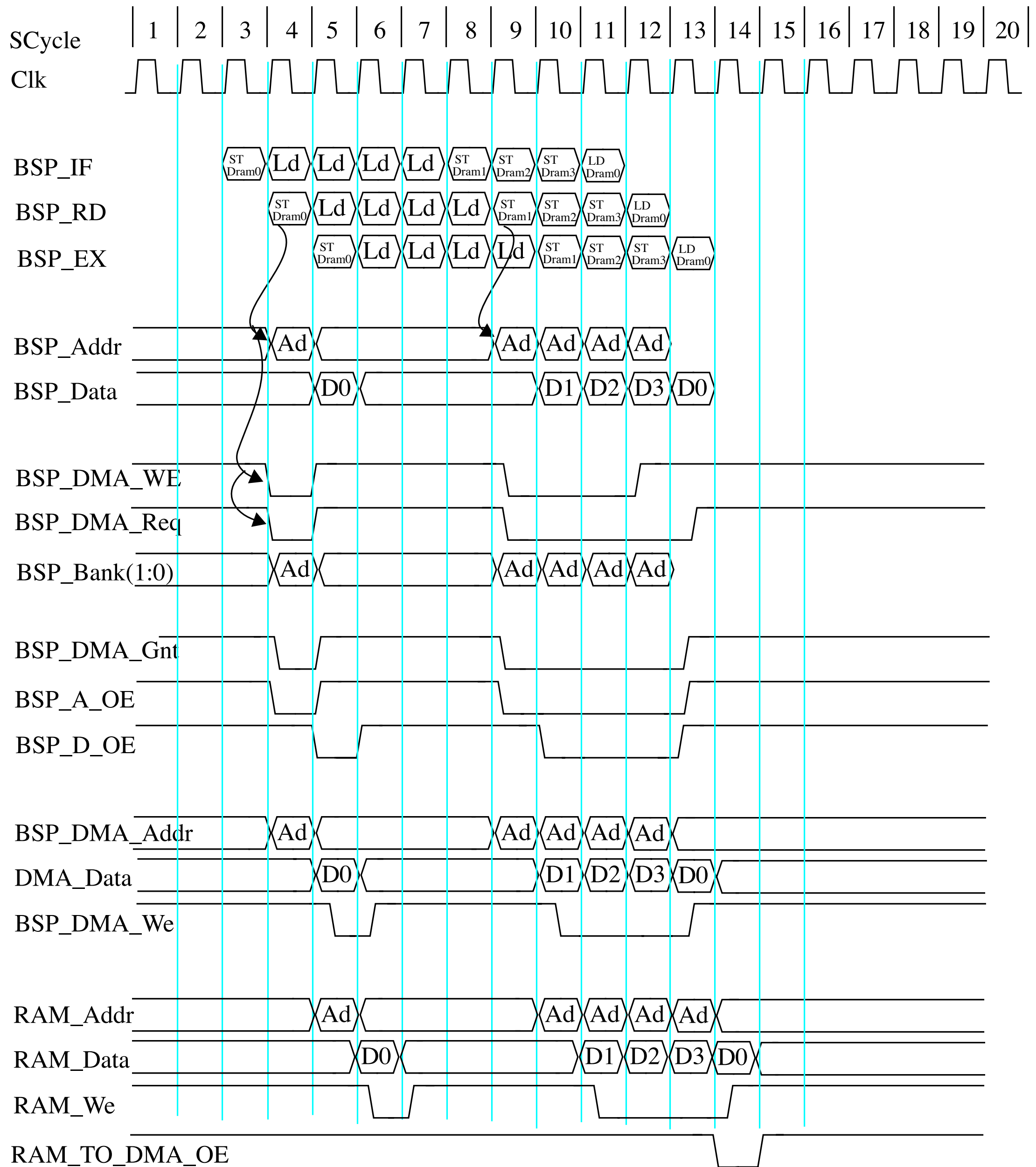


FIGURE 28. Bit Stream Processor Access on DMA Bus

4.2.3.3 H/DMA Access on DMA Bus

Access by the Host/DMA block onto the DMA Bus is shown in Figure 28, “Bit Stream Processor Access on DMA Bus,” on page 108.

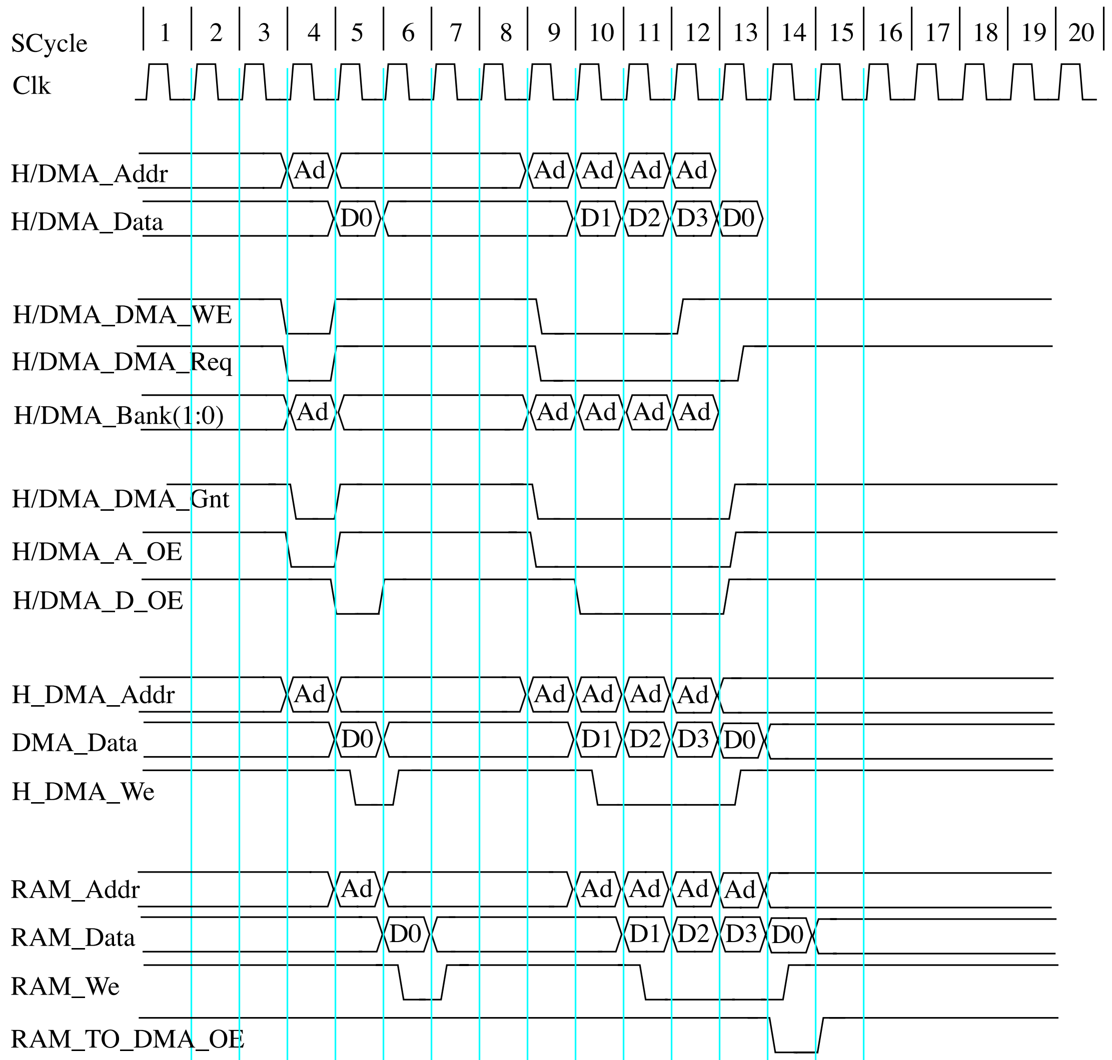


FIGURE 29. Host/DMA Access on DMA Bus

4.3 DMA

The DMA unit consists of the TLB and DMA engine. All VICE initiated Moosehead system memory transactions are handled by the DMA unit. There are two channels to the DMA unit. A channel can be programmed by the Host, the Bitstream Processor or the Media Signal Processor. To initiate a DMA transfer the registers or the memory descriptor list must be programmed. The DMA engine can interrupt the Host on completion of DMA if so desired by the settings in the VICE_INT_EN register and the Interrupt enable bit in each channels control register. The DMA engine can interrupt the Bitstream Processor on completion of a DMA? The DMA engine cannot interrupt the MSP as the MSP does not have interrupts.

Ownership of a DMA channel is not monitored by the DMA subsystem, it assumes others know that they own a DMA channel as a resource. If two users simultaneously utilize a DMA channel by concurrently pro-

gramming its registers or descriptor list, the resulting behavior of the DMA engine will probably produce unexpected results if the registers or memory descriptor list is mixed between the two requestors.

DMA is controlled by writing the DMA Descriptor table which contains four (4) groups of entries that can operate in a ring or as a sequence of DMA transactions that terminate when the last valid Descriptor is processed. There is a Control and Status register per DMA channel (not per descriptor) that further defines the behavior of the DMA.

4.3.1 DMA Descriptors

DMA descriptors can be written by any device that can become master of the Common Bus. This includes the Host Interface, the Bit Stream Processor and the Media Signal Processor (Scalar Unit).

Descriptors are used to define starting addresses of the DMA. They also define the mode (Read/Write, Y/C split) and span and stride settings. Descriptors are 16 bits wide to accommodate the Bit Stream Processor.

There is a set of 8 entries in the descriptor table that make up a DMA transaction. Descriptors will automatically chain to the next descriptor in the list. A descriptor can also be set to halt the dma engine when the DMA for that descriptor has been completed. A descriptor can be skipped over.

The Y/C bit field defines three modes of DMA. For block dma this field is set to 00 and the VICE DMA engine will not separate the YC components and hence the Internal Vice Address C is not needed and is ignored. Only the Y Vice Address is used by the DMA engine as an internal Vice address pointer in this mode. The other two modes separate Y/C when performing a DMA read and re-interleave Y/C when performing a DMA write. For mode 01 the C component is decimated for dma reads and expanded for dma writes. For mode 10 the C component is left at full bandwidth on reads and writes.

For transactions not using the Y/C mode, only the VICE_MEM_Y register need be programmed and not the VICE_MEM_C register. Also for any of the following destinations, the VICE_MEM_C register is ignored by the hardware:

MSP Instruction RAM
BSP Instruction RAM
BSP Table RAM
BSP Decode FIFO
DMA TLB RAM

In the Y/C mode it is legal to have the VICE_MEM_Y register and the VICE_MEM_C register point to different banks of Vice DATA RAM. As a programming convention, the LOC bits (6:4) should reflect the Data RAM bank of the VICE_MEM_Y register.

15	0
Halt(15) Skip(14) RW(13) Fill(12) YC(11:10) HP(9:8) ILV(7) LOC(6:4) HPEN(3) RESV(2:0)	
Virtual System Address (31:16)	
Virtual System Address (15:00)	
Span (15:00) (Line Length)	
Stride (15:00)	
Line Count (15:00)	
Internal Vice Address Y (15:00)	
Internal Vice Address C (15:00)	

RESV(2:0) Reserved, Set to 000 for now.

HPEN(3)
- 0 - Half Pel Mode Disabled, Ignore HP(9:8)
- 1 - Half Pel Mode Enabled, Use HP(9:8) to set mode

LOC(6:4) Vice location for source/destination
- 000 - Data RAM A
- 001 - Data RAM B
- 010 - Data RAM C
- 011 - MSP Instruction RAM
- 100 - BSP Instruction RAM
- 101 - BSP Table RAM
- 110 - BSP Decode Fifo
- 111 - DMA TLB RAM

ILV(7)
- 0 - Process Descriptors Individually.
- 1 - Process Descriptors as Pairs and Interleave them w/Half Pel into VICE memory.

HP(9:8) (Do these modes have intelligence to override descriptor data counts?)
- 00 - Full Pel Vert   Full Pel Horiz
- 01 - Full Pel Vert   Half Pel Horiz
- 10 - Half Pel Vert   Full Pel Horiz
- 11 - Half Pel Vert   Half Pel Horiz

YC(11:10)
- 00 - Block DMA, VICE side treated as continuous block Internal Vice Addr C ignored.
- 01 - Y/C 4:2:2 mode to Y/C 4:2:0 split
- 10 - Y/C 4:2:2 mode to Y/C 4:2:2 split
- 11 - NEW! Y/C 4:2:2 mode to Y only - Works for DMA Read only

Fill(12)
- 0 - DMA is a System Memory <-> VICE transaction.
- 1 - DMA is a Write to VICE from VICEDMA_DATA register.

RW(13)
- 0 - This Descriptor is a read System Memory -> VICE.
- 1 - This Descriptor is a write System Memory <- VICE.

Skip(14)
- 0 - Process this Descriptor and continue to the next when complete.
- 1 - Skip this Descriptor and continue to the next.

Halt(15)
- 0 - Process this Descriptor and continue to the next when complete.
- 1 - Halt DMA After Processing this descriptor.

FIGURE 30. DMA Descriptor Format

4.3.2 DMA Registers

There are six software visible DMA registers per DMA channel.

VICEDMA_CTL_CHX - Control to Set up the DMA.
VICEDMA_STAT_CHX - Status Register of DMA.

VICEDMA_DATA_CHX - Data Register for DMA fill operation (D15-D00).

VICEDMA_MEM_PT_CHX - 32 bit register to read present system address pointed to by DMA address calculation block.
VICEDMA_VICE_PT_CHX - 16 bit register to read present VICE address pointed to by DMA address calculation block.
VICEDMA_COUNT_CHX - 32 bit register to read remaining count of current DMA descriptor.

Only the VICEDMA_CTL_CHX register must be programmed for DMA to occur. The VICEDMA_STAT_CHX register can be used to analyze various halt conditions that the DMA engine may produce. It is also useful for feedback to see which descriptor the DMA engine is presently processing.

The VICEDMA_DATA register must be programmed for the DMA engine to fill an area of on-chip VICE memory with the data contained in this register. Note that the 16 bit value in this register limits the data pattern choices in memory but this should be adequate since most MSP operations are expected to be on 8 or 16 bit integers or fractions.

The registers MEM_PT, VICE_PT and COUNT are intended primarily for diagnostic reasons to see where the DMA address calculators and transaction counts are. This is especially useful if DMA has terminated prematurely.

For a more detailed list of these DMA registers refer to Chapter 2.

4.4 Media Signal Processor Overview

The Media Signal Processor consists of two programmable execution units:

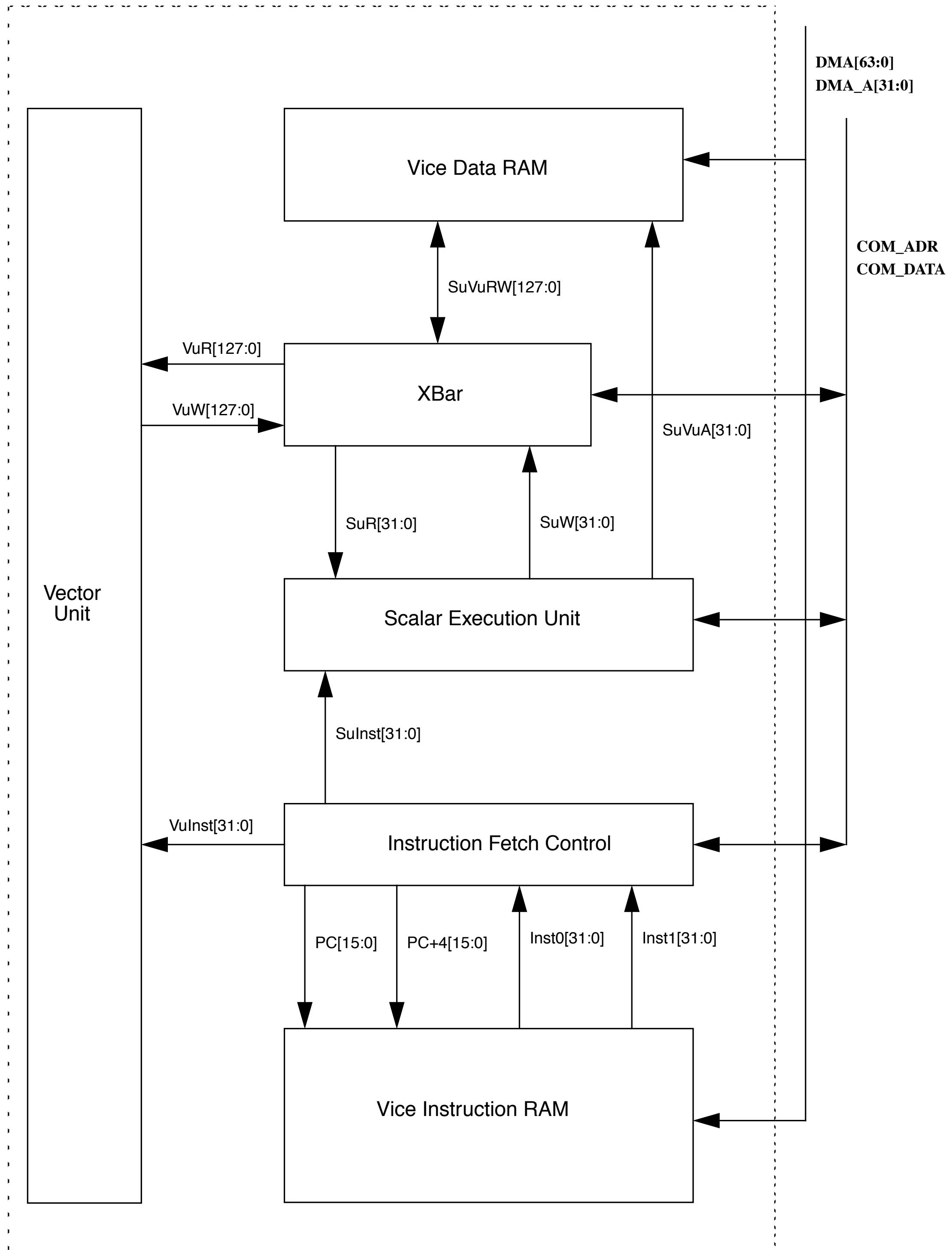
- A scalar execution unit
- A vector execution unit

The two programmable units execute instructions from the MSP Instruction RAM and act on operands contained in the Data RAM. The scalar unit performs control flow operations, scalar integer arithmetic operations and 32 bit logical operations. The vector unit performs SIMD vector integer arithmetic operations and vector logical operations. Programs are stored in the MSP Instruction RAM. The scalar unit and the vector unit are tightly coupled. They execute instructions out of the same Instruction RAM and act on the same Data RAM. A block diagram of the MSP is shown ig Figure 31, "Media Signal Processor Block Diagram," on page 114. The instruction fetch mechanism fetches instructions from the Instruction RAM and channels the appropiration instruction to either the vector or scalar unit. The XBar does data formatting for vector/scalar loads and stores. For further description about the XBar and what the load/store mechanism is, refer to "Load Store Mechanism" on page 132

The MSP exception handling is limited and is different from MIPS 1 exception handling. For a fuller description of this topic, please refer to "MSP Exception Processing" on page 27

The MSP can access VICE registers using Scalar Unit Co-Processor 1 and 3 instructions. These registers control the DMA engine internal to VICE, configuration of the VICE Data RAM and communication and control between the MSP and the Bitstream Processor (BSP).

FIGURE 31. Media Signal Processor Block Diagram



4.4.1 Instruction Fetch Mechanism

4.4.1.1 MSP Instructions

Instructions can be of scalar type or of vector type. The distinction between scalar type and vector type is contained in every instruction. Scalar instructions are a subset of instructions in the MIPS1 ISA. Vector instructions correspond to coprocessor-two instructions of the MIPS1 instruction set. For further description of the instruction set supported, please refer to Section 4.5.2 on page 122 and Section 4.6.5 on page 144

2 instructions are always fetched from the instruction ram. The instruction pair may contain any combination of scalar and vector instructions, i.e two scalar instructions, two vector instructions, the first scalar and the second vector, or the first vector and the second scalar.

4.4.1.2 MSP Instruction RAM

The instruction memory is a dual port read write memory. It needs to be accessible at VICE clock rate. It can be accessed by only one unit at a time: the MSP (scalar unit and vector unit) or the VICE DMA controller. The MSP has only read access to the MSP Instruction RAM. The VICE DMA has both read and write access to the MSP Instruction RAM. DMA or Unix Processor PIO is able to write to the MSP Instruction RAM anytime. DMA or Unix Processor PIO is able to read from the MSP Instruction RAM when the MSP is in HALT state. DMA access to the MSP instruction RAM must be on a 64 bit aligned data transfer. The scalar unit and the vector unit receive instructions out of this memory. During a DMA transfer read, the scalar unit and the vector unit will idle.

The instruction RAM is organized in two banks of 32 bits each, into even and odd words. This gives us the flexibility to fetch any 2 subsequent instructions from the instruction RAM.

The instruction fetch control produces PC[31:0] and PC+4[31:0]. If 2 instructions are issued, the program counter will be incremented by 8. However, if only 1 instruction is issued, the program counter is only incremented by 4.

The 2 instructions (Inst0[31:0], Inst1[31:0]) read from the Instruction RAM are sent to the Instruction Fecth Control where it is sufficiently decoded enough to route them to either the scalar or vector unit as SuInst[31:0] and VuInst[31:0] respectively. An instruction valid signal (SuInstVld, VuInstVld) is also sent along with the instruction to indicate that the data on SuInst[31:0] and VuInst[31:0] is a valid instruction.

Figure 32, “Instruction RAM and Instruction Fetch Control,” on page 116 shows how the Instruction Fetch Control is connected to the Instruction RAM.

Figure 33, “Fetching 2 Instructions from the Instruction Ram,” on page 116 shows the fetching of 2 instructions, when PC is an even word address and the scenario when PC is an odd word address.

FIGURE 32. Instruction RAM and Instruction Fetch Control

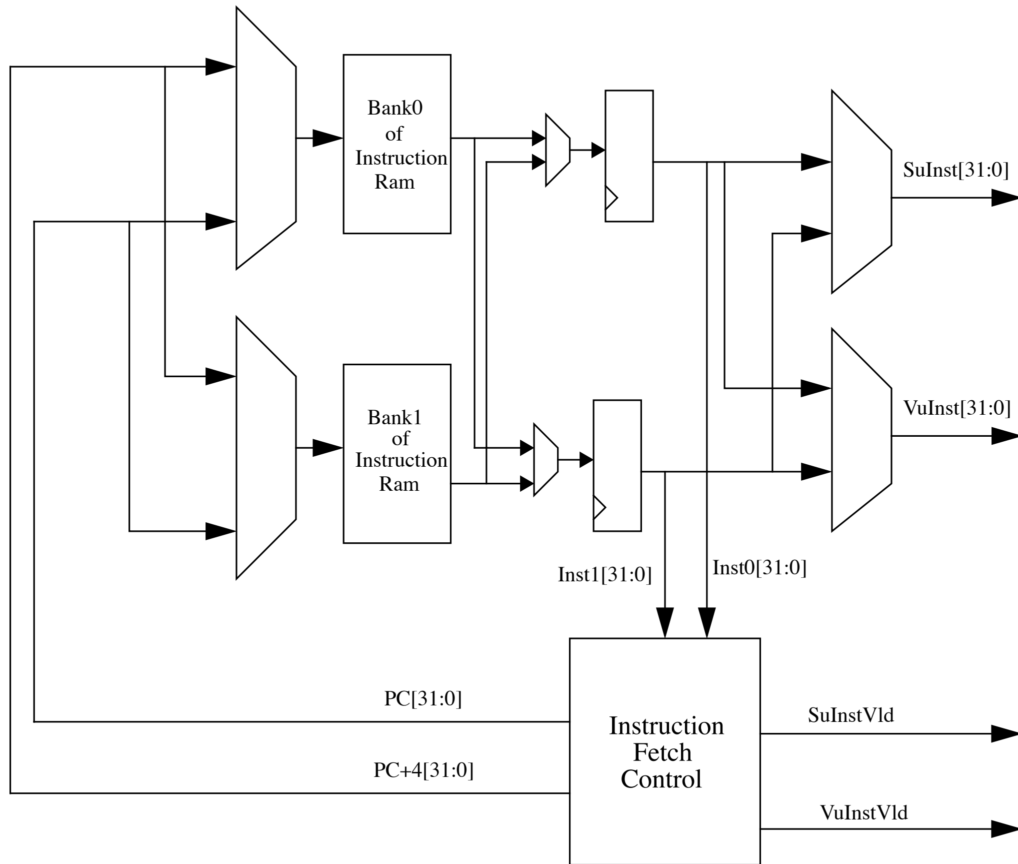
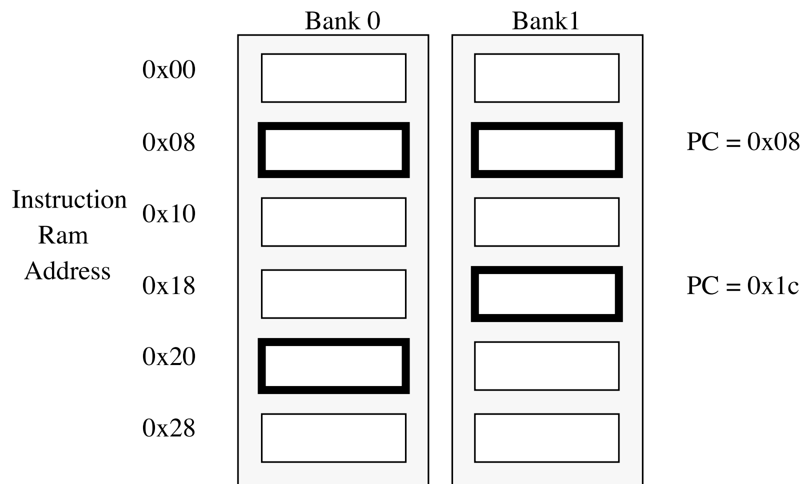


FIGURE 33. Fetching 2 Instructions from the Instruction Ram



4.4.1.3 Instruction Issue

The MSP can execute up to 2 instructions per clock cycle in a 2-instruction group. An instruction group consists of one or no scalar instruction and one or no vector unit instruction in any order. A one instruction group is issued when the next 2 pending instruction fetched from the Instruction RAM are for the same execution unit. LWCz/SWCz, MTCz/MFCz, CTCz/CFCz can be issued in the same instruction group with vector unit computational instruction, i.e., these instruction group with scalar unit instructions.

The branch delay slot instruction is always a one instruction group, consisting of the instruction immediately following the branch. The branch instruction itself may be part of a 2 instruction group if it is immediately preceded by a vector unit computation instruction.

4.4.1.4 Instruction PC

The instruction PC is 32 bits wide with bits [31:16] and [1:0] grounded to 0. This means that the hardware will not be able to detect an exception in the following scenario

- jr $2

If $2 contains an address that is not aligned on a word boundary, this address error won't be detected because $2[1:0] is dropped off before latching into the PC. Also if bits [31:16] of $2 fall outside valid MSP instruction RAM address space, this error will also not be detected.

- beq $0,$0,xffff

Since the 16 bit offset is shifted by 2 giving 18 bits, the 2 high order bits of the shifted offset will not be seen by the hardware. So, if the resulting address should reside fall outside VICE instruction RAM address space, it will not be detected by the hardware.

See "Instruction Fetch Address Exception(7)" on page 29.

4.4.2 Common Bus Interface

The Media Signal Processor communicates with the Bitstream Processor, the DMA engines and the general purpose VICE chip registers, across a bus called the Common bus.

The Scalar Unit Processor of the MSP can access the general purpose registers inside VICE by using CTC1 and CFC1 instructions. The Scalar Unit Processor can access the DMA Descriptors for the channel 1 DMA engine using MTC3 and MFC3 instructions. The Scalar Unit Processor can access DMA Descriptors for the channel 2 DMA engine using CTC1 and CFC1 instructions.

These register and DMA descriptors are accessed with Co-Processor instructions so that the Scalar Unit can determine the address of the transaction early in the pipeline. The Scalar Unit cannot be stalled. If accesses to registers and DMA descriptors was performed with Load/Store operations, the Scalar Unit would halt all activity on all buses inside of VICE with any Load/Store instruction. The separation of register access to the Co-Processor 1 & 2 Op-Codes allows the Scalar Unit to determine which bus is to be accessed.

4.4.3 Shared Memory

The MSP, BSP, DMA and Host Interfaces internal to VICE all have access to the VICE Data RAM. This RAM is the primary communication path between the Unix System Processor, the BSP and the MSP.

4.4.3.1 VICE Data RAM

The Media Signal Processor utilizes the VICE Data RAM as its Data storage area. All MSP Load/Store commands use the VICE Data RAM as either a Source or Target.

The VICE Data RAM is also used by the Bitstream Processor as it's primary Data storage area.

The VICE Data RAM can also be accessed by the DMA engine and by the Unix System Processor.

The Data RAM is organized into 3 separate banks. These Banks are called Bank A, B & C. There are two separate buses that can access each bank. This allows for two devices to access different banks of VICE Data RAM simultaneously.

The Media Signal Processor cannot be stalled. If the MSP has been allowed to access a Bank of Data RAM, it must be allowed to access it without waiting. The programmer may give "clues" to the arbiter of the VICE Data RAM by dis-allowing the MSP access to a particular bank of VICE Data RAM. This will allow the DMA or BSP to access that Data RAM. The arbiter will not stall the DMA or BSP when the Scalar Unit performs a Load/Store operation because the arbiter will know that the MSP is not going to access that Data RAM.

For example; the MSP tells the arbiter that it will not access VICE Data RAM B. The arbiter now knows that all access by the BSP or DMA engine can proceed without regard to MSP Load/Store activity because the MSP will not access Data RAM B.

The MSP has the option of requesting access to all three banks of VICE Data RAM. This will result in lower data throughput for the BSP and DMA engine when those units attempt to access VICE Data RAM.

4.4.3.2 Mail Boxes

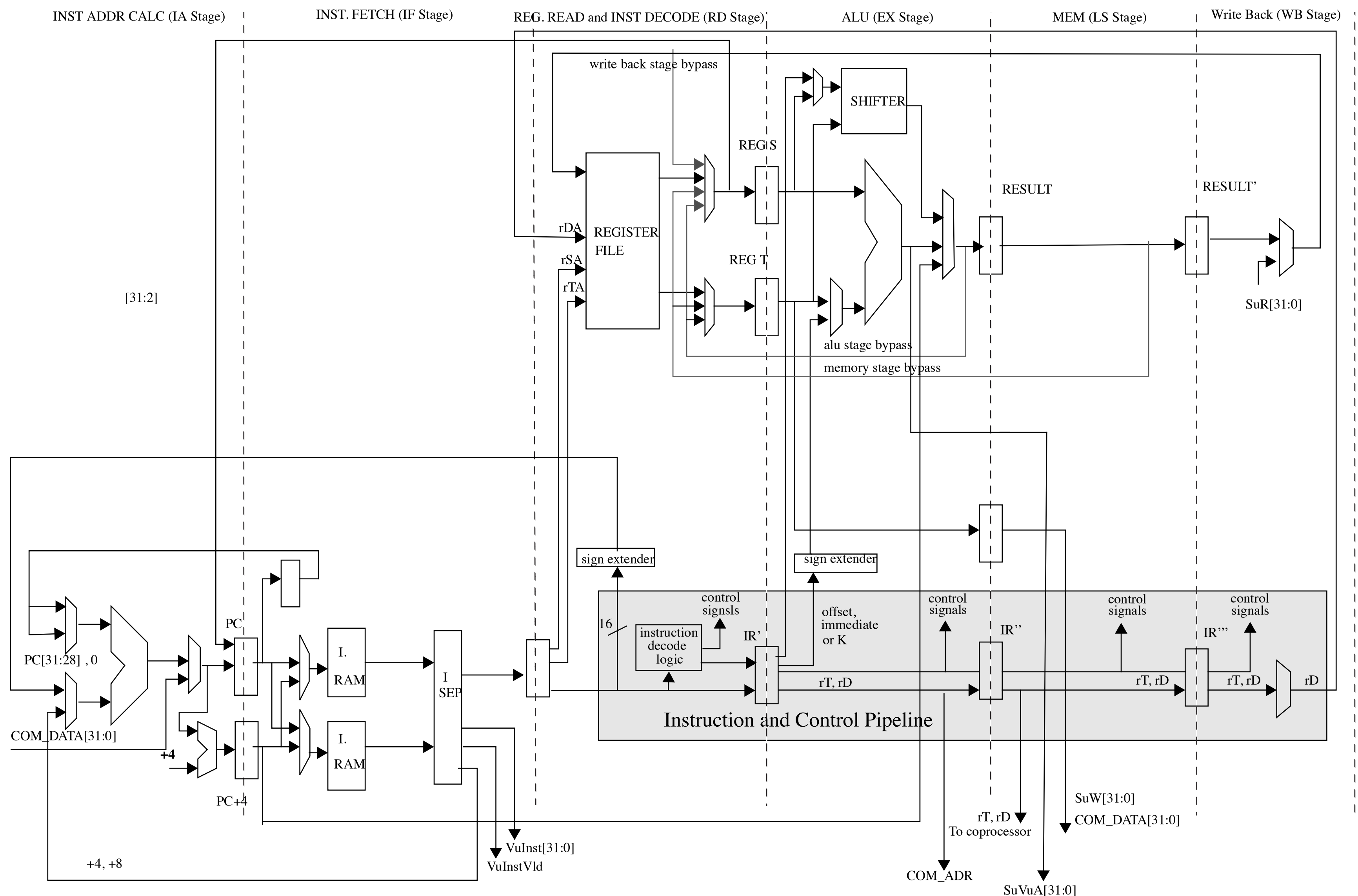
The Media Signal Processor has access to a special set of registers to communicate with the Bitstream Processor. These registers contain a "magic" bit that is reset when a processor "reads" it's mail. These mailboxes are designed to make synchronization tokens between the MSP and BSP be fast and efficient. For more information on these registers, refer to Chapter 2 and Section 4.6.

4.5 MSP Scalar Unit

The scalar unit is an embedded, pipelined, risc processor, designed to implement a subset of the MIPS 1 ISA. It contains:

- Program counter
- Instruction pipeline
- Branch adder
- ALU
- Shifter
- Register file
- Instruction decode logic
- Miscellaneous control logic

The Scalar Unit connects with the MSP Instruction RAM and VICE data RAM. The MSP Instruction RAM and the VICE data RAM can be loaded through the VICE DMA controller. The MSP Instruction RAM is addressed by the Program counter. The program counter can be initialized by the host CPU. The scalar unit can perform load operations (from VICE data RAM to the register file) and store operations (from register file to VICE data RAM). The ALU can receive two operands from the register file or one from the register file and one from an immediate field in the instruction. It performs integer arithmetic or logical operations, and stores the result into the register file. The shifter can be used, instead of the ALU, to perform left and right shift operations by an arbitrary number of bits on an operand. Internal vice registers are accessed through MTCz/MFCz/CTCz/CFCz instructions. See "Register Address Map Summary" on page 39.

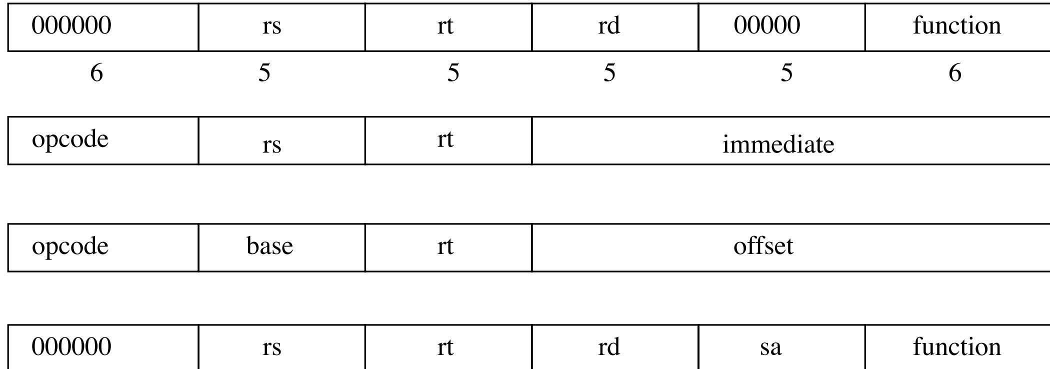


INST ADDR CALC (IA Stage)
INST. FETCH (IF Stage)
REG. READ and INST DECODE (RD Stage)
ALU (EX Stage)
MEM (LS Stage)
Write Back (WB Stage)
write back stage bypass
REGiS
REG T
SHIFTER
RESULT
RESULT'
rDA
rSA
rTA
REGISTER FILE
SuR[31:0]
[31:2]
alu stage bypass
memory stage bypass
sign extender
sign extender
control signsls
control signals
control signals
control signals
offset, immediate or K
16
instruction decode logic
IR'
IR''
IR'''
rT, rD
rT, rD
rT, rD
rD
PC
PC[31:28] , 0
I. RAM
I. RAM
I SEP
COM_DATA[31:0]
+4
PC+4
Instruction and Control Pipeline
SuW[31:0]
COM_DATA[31:0]
rT, rD
To coprocessor
VuInst[31:0]
VuInstVld
+4, +8
COM_ADR
SuVuA[31:0]

4.5.1 Scalar unit instructions format

Instructions of the scalar unit consists of 32 bits which are grouped in several fields. A description of some typical instructions of the MIPS architecture is given in Figure 34, “Scalar Unit Instruction Format,” on page 122. Since the scalar unit implements a subset of the MIPS R300 instruction set, the format of each instruction is not specified in this document. Only a list of the instructions supported and a list of those not supported are given here. .

FIGURE 34. Scalar Unit Instruction Format



4.5.2 Scalar Unit Instruction Set

The scalar unit implements the following instruction set:

ADD	Add
ADDI	Add immediate
ADDIU	Add Immediate Unsigned
ADDU	Add Unsigned
AND	And
ANDI	And Immediate

BEQ	Branch on equal
BNE	Branch on not equal
BGEZ	Branch on greater than or equal to zero
BGEZAL	Branch on greater than equal to zero and link
BGTZ	Branch on greater than zero
BLTZAL	Branch on less than zero and link
BLEZ	Branch on less than or equal to zero
BLTZ	Branch on less than zero

BREAK	Break. See “Breakpoint Exception (2)” on page 27.

CFCz	Move control from coprocessor
CTCz	Move control to coprocessor

MFCz	Move from coprocessor (Vector Unit)
MTCz	Move to coprocessor
J	Jump
JAL	Jump and link
JR	Jump register
JALR	Jump and link register
LB	Load byte
LBU	Load byte unsigned
LH	Load half word
LHU	Load half word unsigned
LUI	Load upper immediate
LW	Load word
NOR	Nor
OR	OR
ORI	OR immediate
XOR	Exclusive or
XORI	Exclusive or immediate
SB	Store Byte
SH	Store halfword
SW	Store word
SLL	Shift left logical
SLLV	Shift left logical variable
SLT	Set on less than
SLTI	Set on less than immediate
SLTIU	Set on less than immediate unsigned
SLTU	Set on less than unsigned
SUB	Subtract
SUBU	Subtract unsigned
SRA	Shift right arithmetic
SRL	Shift right logical
SRAV	Shift right arithmetic Variable
SRLV	Shift right logical variable

Vector unit load stores instructions are overloaded to coprocessor two loads and stores.

LAV	Load alternate to vector unit
LBV	Load byte to vector unit
LSV	Load short word to vector unit
LLV	Load long word to vector unit
LDV	Load double word to vector unit
LQV	Load quad word to vector unit

LRV	Load rest to vector unit
LPV	Load packed signed to vector unit
LUV	Load packed unsigned to vector unit
LHV	Load half words to vector unit
LFV	Load fourths to vector unit
LXV	Load extended to vector unit
LZV	Load zero to vector unit
LTWV	Load transpose and wrap to vector unit

SAV	Store alternate from vector unit
SBV	Store byte from vector unit
SSV	Store short word from vector unit
SLV	Store long word from vector unit
SDV	Store double word from vector unit
SQV	Store quad word from vector unit
SRV	Store rest from vector unit
SPV	Store packed signed from vector unit
SUV	Store packed unsigned from vector unit
SHV	Store half words from vector unit
SFV	Store fourths from vector unit
SXV	Store extended from vector unit
SZV	Store zero from vector unit
STV	Store transpose from vector unit
SWV	Store wrap from vector unit

4.5.3 Instructions not supported

The following MIPS1 instruction are not supported by the MSP scalar unit.

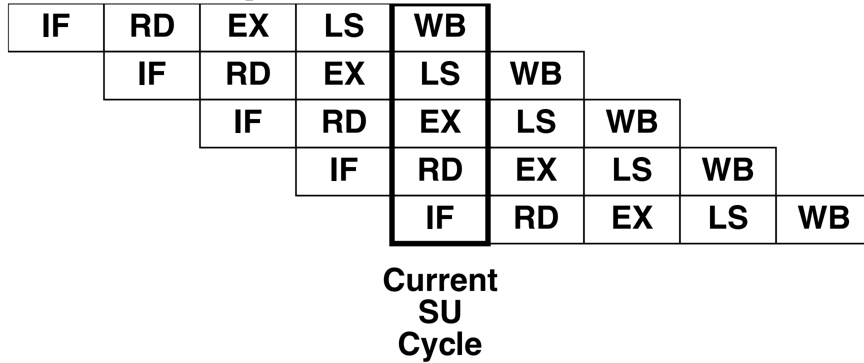
BCzF	Branch On Coprocessor z False
BCzT	Branch on Coprocessor z True
SYSCALL	System call
MULT	All Multiplies must use the vector unit
DIV	No Divides supported on VICE

4.5.4 Pipeline

4.5.4.1 Instruction Execution

The Scalar Unit uses a 5-stage instruction pipeline to process its instructions. Figure 35, “Su Instruction Pipeline,” on page 125 shows how instructions flow through the pipeline.

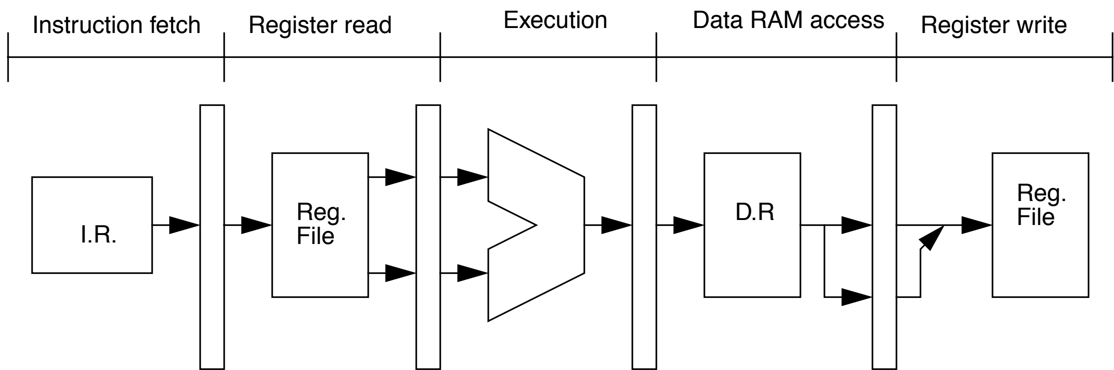
FIGURE 35. Su Instruction Pipeline



SU Instruction Pipeline

- IF : Instruction Fetch
- RD : Register File Read and Instruction Decode
- EX : Execute Stage
- LS : Memory Load/Store stage
- WB : Write Back Stage

FIGURE 36. Visualization of the various pipeline stages

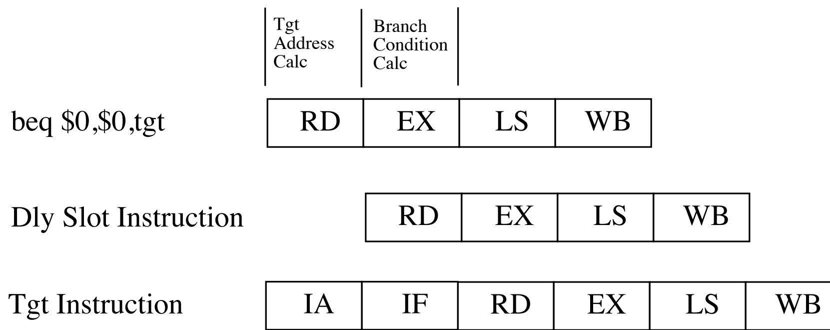


4.5.4.2 Branch Pipeline

4.5.4.2.1 Branch Taken

The target address is computed during the RD stage of the branch instruction. During the EX stage, the target instruction is fetched while the branch condition is determined. If the branch is taken, there is no penalty.

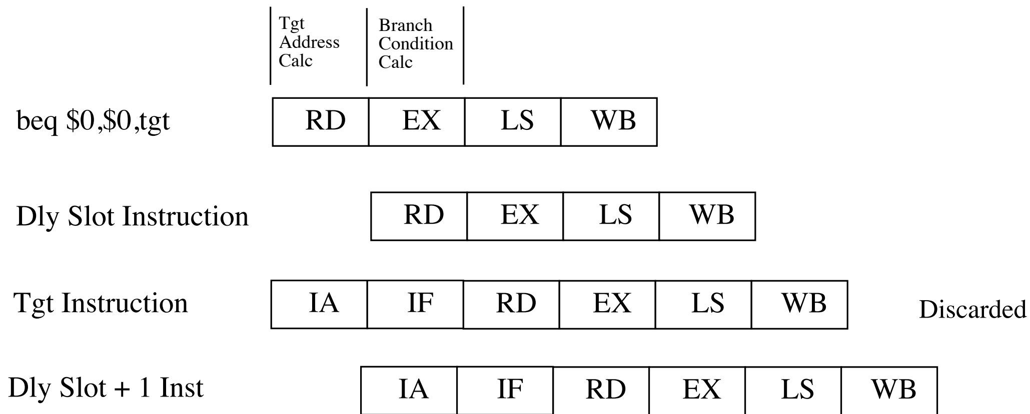
FIGURE 37. Illustration of Branch Taken



4.5.4.2.2 Branch Not Taken

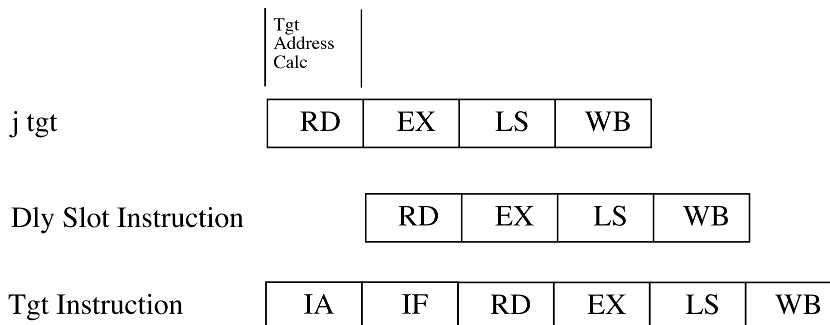
The target address is computed during the RD stage of the branch instruction. During the EX stage, the target instruction is fetched while the branch condition is determined. If the branch is not taken, there is a 1 cycle penalty to refetch the in-line instruction stream.

FIGURE 38. Illustration of Branch Not Taken



4.5.4.2.3 Jumps

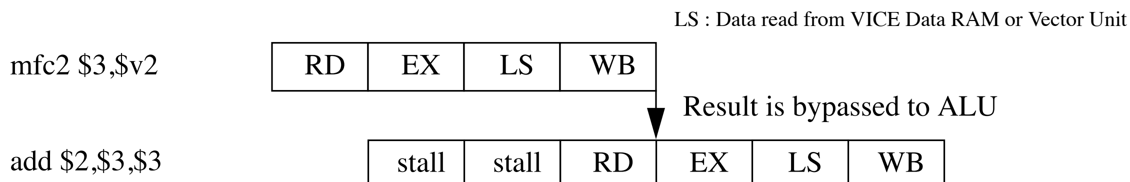
The pipe timing for jumps is similar to branches which are taken.



4.5.4.3 Interlocks

4.5.4.3.1 MFCz/CFCz/LW and SU instruction (RAW Hazard)

MFCz, CFCz, and Scalar Loads have a 2 cycle load delay slot. The hardware interlocks this dependency.



Implementation Details and false interlocks :

To detect this interlock, the harware compares dest of the LW/MFCz/CFCz with the strict RS and RT field of a valid Scalar instruction. i.e. bits [25:21] for RS and [20:16] for RT. Table 70, "Dependency check for 2 cycle load delay slots," on page 127 lists what are the RS and RT fields for different Scalar Instructions. A "*" marks that the corresponding RS or RT field is valid and should be used to do in this interlock detection while a non "*" indicates what other information these bit fields contain.

TABLE 70. Dependency check for 2 cycle load delay slots

RS[25:21]	RT[20:16]	Instruction
*	Dest	LW
*	*	SW
*	Vu Reg	LWC2, SWC2
4	*	MTCz
6	*	CTCz
*	Imm, Tgt	Immediate,JR,JALR
Tgt	Tgt	J, JAL
0	Dest	MFCz
2	Dest	CFCz
*	*	All others

To simplify the detection logic and to not create a critical path in it, the dest is compared with the RS for all valid Scalar Instructions and the RT for all valid Scalar Instructions except LWC2, SWC2, or Immediate Instructions.

This may cause false stalls. A "*" above marks a valid check. A non "*" means that this could cause a false stall. Generally this should not be a problem except for LWC2 & SWC2 because the frequency of LWC2/ SWC2 followed by SU instructions is high. This is why the dest of the LWC2/MFCz/CFCz is compared with the RS for all Scalar Instruciont and also with the RT for all valid Scalar instructions except LWC2, SWC2, or Immediate Instructions.

For e.g. the following code fragment

```
mfc2 $4, $v2
mtc2 $0, $v8
```

will cause a 2 cycle load delay slot even though there is no real dependency. This is because we compare the dest of the mfc2 instruction, which is $4, with bits[25:21] of the mtc2 instruction which is also $4. Since, there is a match, a false interlock occurs.

4.5.5 Scalar unit operation

The scalar unit can be thought as a machine capable of implementing the instructions that have been described. A detailed operation of each pipeline stage is shown in the following tables. The SU generates an address for the VICE instruction RAM. This address can come from two different sources: the DMA controller, to pre-load the VICE Instruction RAM; the program counter, to step through the execution of a program. The program counter increments at every clock cycle and can be initialized by: the system, an operand contained in a jump instruction, the content of a register.

The register file is a three port RAM. It receives three addresses: two read addresses and on write address. The purpose for this organization is to be able to perform operations like (c = a + b) in one cycle. The machine is pipelined, so in effect a and b get read in one cycle and c gets written in a subsequent cycle, but in any given cycle there can be two reads and one write, which don't belong to the same instruction.

4.5.5.1 Load operations

Load operations are transfers of data from the VICE data RAM to the register file. Load instructions contain the address of a "base" register in the register file and an offset address. To implement the load operations first the content of the "base" register is read; then the ALU computes the addresses for the VICE data RAM.

128 bits of data is read from the VICE data RAM at every read. In order to handle LB, LH, LBU, LHU and LW, the 128 bits is rotated and formatted in the XBar to suit the necessary data size. 32 bits is then sent to the Scalar Unit via the SuR[31:0] bus. The XBar also handles sign extension for LB, LH and zero extension for LBU, LHU

Refer to "Load Store Address Error Exception (0,1)" on page 27 for unaligned load/store addresses.

4.5.5.2 Store operations

Store operations are transfers of data from the register file to the VICE data RAM. Store instructions contain the address of a "base" register in the register file and an offset address. One of the read ports of the register file is used to read the "base" address and the other read port is used to read the data that needs to be stored in the VICE data RAM. The ALU computes the destination address for the VICE data RAM, places the destination address on the SuVuA (data RAM address) bus, and the data to be stored on the SuW[31:0] (data bus write).

Store data is sent to the XBar via the SuW[31:0] bus where it is again formatted to suit the 128 bit bus interface of the Vice Data RAM.

See "X-Bar" on page 136. for more information regarding the XBar.

4.5.5.3 ALU operations

The ALU performs operations on the data read from two registers or from one register and an immediate operand supplied in the instruction.

4.5.5.4 Jump operations

Jump operations cause the program counter to continue execution from an arbitrary address in the VICE instruction RAM. There are two kinds of jump instructions: Jump and Jump register.

For the Jump instruction the destination address is provided directly with the instruction;

For the Jump Register instruction the destination address is contained in a register specified by the instruction.

4.5.5.5 Jump and Link

Jump and link operations are similar to jump instructions, with the difference that the content of the program counter before executing the jump is saved in a register that can be specified in the instruction. Later the program can resume execution to where it was before the jump, by executing a jump register instruction, to the register in which the program counter was stored.

4.5.5.6 Branch Operations

Branch operations are similar to jumps with the difference that in the case of a branch instruction the jump occurs conditionally, depending on the result of a logic or arithmetic operation. Branch instructions therefore require two operations: the evaluation of the condition based on which to jump or not to jump, and the calculation of the address to jump to. The ALU evaluates the condition, whereas a second adder, the branch adder, calculates the address to jump to.

4.5.5.7 Coprocessor loads

Coprocessor loads are loads to the vector unit. Coprocessor loads are assisted by the scalar unit. The scalar unit generates the address of the first byte of the quad word to be loaded into one of the 32 registers of the register file of the vector unit.

4.5.5.8 Coprocessor stores

Coprocessor stores are stores to the vector unit. Coprocessor stores are also assisted by the scalar unit. The scalar unit generates the address of the first byte of memory where the content of one of the registers of the vector unit register file needs to be stored.

4.5.5.9 Interrupts

See "Interrupts/Exceptions" on page 27.

4.5.5.10 Debugging capabilities

See "Debug Operations" on page 30.

4.5.5.11 Resolving data hazards

Due to the depth of the pipeline, data hazards may exist. For example when doing
c = a + b;
d = c + f;
the new value of c will not be available from the register file yet, when it is needed for the second instruction. Instead of stalling the pipeline, data hazards are solved by the technique of forwarding. Although c is not available yet from the register file, its value is still in the pipeline, and can be utilized by the ALU. Two multiplexers exist in front of the inputs of the ALU for this purpose.

The forwarding logic can detect the fact that the a required operand is still in the pipeline, by comparing the address of the operands of the current instructions with the address of the destination of previous instructions.

4.5.5.12 Resolving branch hazards

A second kind of hazard that could affect our processor is a branch hazard. This hazard can occur because when we execute a branch instruction, the decision on whether to branch or not to branch depends on the value of condition codes which are available only after the execution stage: two cycles after the instruction fetch. Potentially then, the two instructions following the branch could need data generated by the previous two instructions, which is not in the register file yet. This problem is solved in the following way:

For the instruction immediately following the branch, the “delay slot”, we require the programmer to use an instruction that does not have any dependency on the previous two; if no useful instruction that does that is available, the programmer will insert a NOP. For the other instruction, instead of stalling the pipeline, we continue execution of the program assuming that the branch is taken and we nullify it later if we discover that it should not have been taken. This mechanism allows us not to lose any cycles during the execution of a loop, and it forces us to lose one when the program exits the loop.

4.5.6 Scalar Unit Blocks

4.5.6.1 Instruction and control pipeline

The flow of data through the data path is controlled by multiplexers, which in turn are controlled by the instructions. Each mux gets controlled by the instruction whose level of execution reached the pipeline stage which the mux is in. Instructions are completely decoded as soon as they are received: in the register read stage; and the decoded signals are sent though a control pipeline to match the delays in the data pipeline.

4.5.6.2 Register File

The register file consists of 32 registers of 32 bits each. It contains two read ports and one write port. The two read ports and the write port can be accessed simultaneously.

4.5.6.3 ALU

The ALU performs arithmetic and logic operations. The operands are supplied by the register file. They get registered in a pipeline register before being presented to the inputs of the ALU. One of the operands may alternatively be supplied by the immediate field in instructions. Results of operations get stored in a pipeline register before being written into the register file.

4.5.6.4 Shifter

The shifter performs and left and right shift operations by a number of bits varying from zero to 31. It can shift the bits coming from register t, by a number contained in a 5-bit field in the instruction or in the low-order five bits of register rs referenced by the instruction. A multiplexer in front of the shifter selects the source that indicates how many bits to shift. Left shifts cause insertion of zeros into the low order bits. Right shifts can cause introduction of zeros (logical shifts) or of the sign bit (arithmetic shift) on the high order bits. The shift operation is performed during the ALU cycle. When performing shift instructions, during the ALU cycle the scalar unit utilizes the shifter instead of the ALU. Depending on the type of instruction, a multiplexer selects the output of the shifter or that of the ALU.

4.5.7 Registers

See “Register Address Map Summary” on page 39.

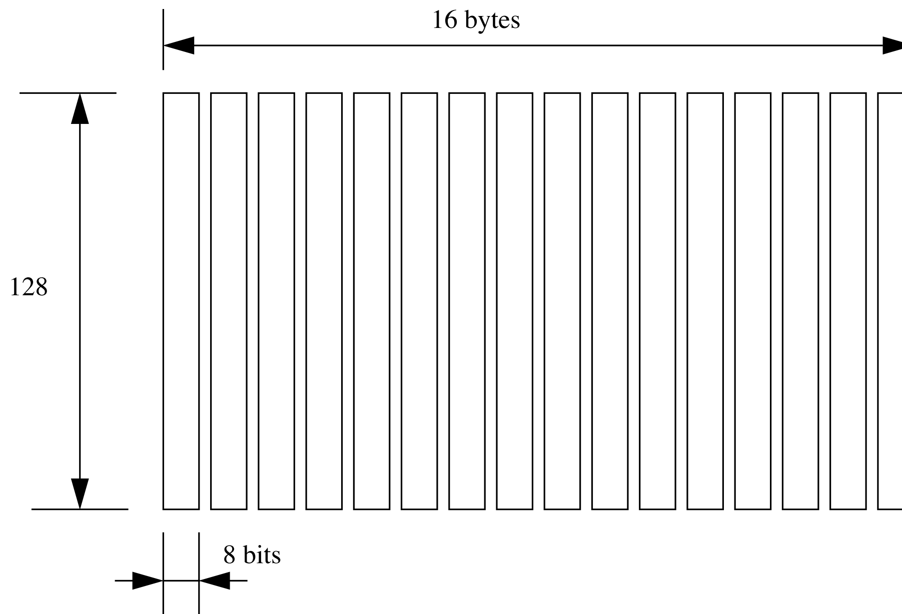
The MSP registers that are in the scalar unit are

MSP_CTL_STAT
MSP_ExcpFlag
MSP_PC
MSP_BadAddr
MSP_EPC
MSP_Cause
MSP_WatchPoint

4.5.8 Load Store Mechanism

4.5.8.1 Data RAM

The are 3 different data memory banks: Banks A, B, C. Each memory bank is comprised of 16 8-byte blocks giving a total width of 128 bits. This was done so that we could have byte write capability of the RAM. In this implementation, they are composed of 16 128x8 RAMs, giving a total memory size of 2 KBytes for each memory bank.



4.5.8.2 Data Formats

In order to understand the memory organization it is convenient to think of the content of the memory as consisting of several “data elements”. Each data element can be one byte or a group of bytes. If it is a group of bytes, the bytes can be: two consecutive bytes, four consecutive bytes, eight consecutive bytes, sixteen consecutive bytes, every other byte, every fourth byte. The memory can be accessed with a byte resolution;

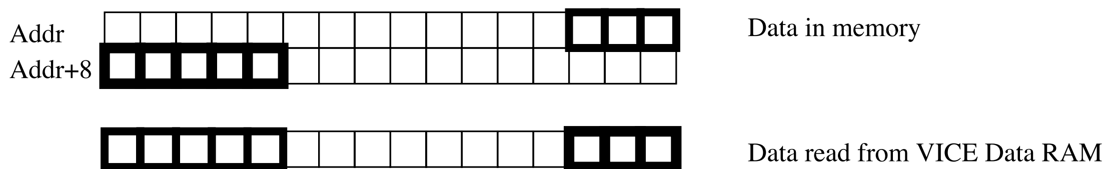
data elements can start at any byte in memory. (Alternates are supported. Please refer to MIPS Media Engine Sketch)

The names of these organizations are:

Byte	One byte only	Example:	00000000 00000001
Short	Two consecutive bytes	Example:	00000000 00000011
Long	Four consecutive bytes	Example:	00000000 00001111
Double	Eight consecutive bytes	Example:	00000000 11111111
Quad	Sixteen consecutive bytes	Example:	11111111 11111111
Packed	Eight consecutive bytes	Example:	00000000 11111111
Halves	Every other byte	Example:	01010101 01010101
Fourths	Every fourth byte	Example:	00010001 00010001

In addition, if the accessed data element (which is smaller than a quad word) crosses a quad word boundary, the VICE data RAM will be supplied an address + 8 so that the complete data is provided in one access. This is not possible for a quad word, therefore the LRV instruction is used,

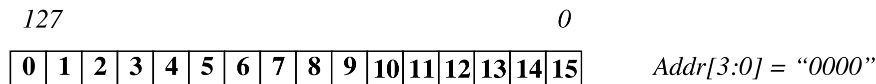
FIGURE 39. Illustration of wrap-around access



4.5.8.3 Big Endian

Only Big Endian is supported on vice. The following diagram illustrates this concept. Hence Addr[3:0]="0000" will access byte 0, and Addr[3:0]="1111" will access byte 15.

FIGURE 40. Big Endian Mode



4.5.8.4 Byte marks

To allow memory to be accessed with a byte granularity and support all the modes described above, the scalar unit produces a double word address and byte marks, for the bytes that need to be accessed. The byte marks are a function of the four least significant bits of the load/store address and of the load/store opcode (such as Half, Fourth, Zero, Xtend, ...) , which is specified by the instruction.

TABLE 71. Memory byte marks

ADR[3:0]	sbv	ssv	slv	sdv	sqv
0	10000000 00000000	11000000 00000000	11110000 00000000	11111111 00000000	11111111 11111111
1	01000000 00000000	01100000 00000000	01111000 00000000	01111111 10000000	01111111 11111111
2	00100000 00000000	00110000 00000000	00111100 00000000	00111111 11000000	00111111 11111111
3	00010000 00000000	00011000 00000000	00011110 00000000	00011111 11100000	00011111 11111111
4	00001000 00000000	00001100 00000000	00001111 00000000	00001111 11110000	00001111 11111111
5	00000100 00000000	00000110 00000000	00000111 10000000	00000111 11111000	00000111 11111111
6	00000010 00000000	00000011 00000000	00000011 11000000	00000011 11111100	00000011 11111111
7	00000001 00000000	00000001 10000000	00000001 11100000	00000001 11111110	00000001 11111111
8	00000000 10000000	00000000 11000000	00000000 11110000	00000000 11111111	00000000 11111111
9	00000000 01000000	00000000 01100000	00000000 01111000	10000000 01111111	00000000 01111111
10	00000000 00100000	00000000 00110000	00000000 00111100	11000000 00111111	00000000 00111111
11	00000000 00010000	00000000 00011000	00000000 00011110	11100000 00011111	00000000 00011111
12	00000000 00001000	00000000 00001100	00000000 00001111	11110000 00001111	00000000 00001111
13	00000000 00000100	00000000 00000110	10000000 00000111	11111000 00000111	00000000 00000111
14	00000000 00000010	00000000 00000011	11000000 00000011	11111100 00000011	00000000 00000011
15	00000000 00000001	10000000 00000001	11100000 00000001	11111110 00000001	00000000 00000001

ADR[3:0]	spv, suv, sxv, szv	shv	sfv	sav
0	11111111 00000000	10101010 10101010	10001000 10001000	00110011 00110011
1	01111111 10000000	01010101 01010101	01000100 01000100	INVALID ADRS
2	00111111 11000000		00100010 00100010	11001100 11001100
3	00011111 11100000		00010001 00010001	INVALID DRS
4	00001111 11110000			00110011 00110000
5	00000111 11111000			INVALID ADRS
6	00000011 11111100			11001100 11000000
7	00000001 11111110			INVALID
8	00000000 11111111			00110011 00000000
9	10000000 01111111			INVALID
10	11000000 00111111			11001100 00000000
11	11100000 00011111			INVALID
12	11110000 00001111			00110000 00000000
13	11111000 00000111			INVALID
14	11111100 00000011			11000000 00000000
14	11111110 00000001			INVALID

4.5.8.5 Vector unit register file byte marks

The bytes of the registers are also controlled by byte marks. By analyzing the tables below, we notice that there are many addresses which are considered invalid. This is because these addresses would cause data elements to be split.

IR[10:7]	lbv	lsv	llv	ldv, lfv	lqv, lpv, luv, lxv, lzv, lhv, ltwv
0	10000000 00000000	11000000 00000000	11110000 00000000	11111111 00000000	11111111 11111111
1	01000000 00000000	INVALID	INVALID	INVALID	INVALID
2	00100000 00000000	00110000 00000000	INVALID	INVALID	INVALID
3	00010000 0000000	INVALID	INVALID	INVALID	INVALID
4	00001000 00000000	00001100 00000000	00001111 00000000	INVALID	INVALID
5	00000100 00000000	INVALID	INVALID	INVALID	INVALID
6	00000010 00000000	0000001100000000	INVALID	INVALID	INVALID
7	00000001 00000000	INVALID	INVALID	INVALID	INVALID
8	00000000 10000000	00000000 11000000	00000000 11110000	00000000 11111111	INVALID
9	00000000 01000000	INVALID	INVALID	INVALID	INVALID
10	00000000 00100000	00000000 00110000	INVALID	INVALID	INVALID
11	00000000 00010000	INVALID	INVALID	INVALID	INVALID
12	00000000 00001000	00000000 00001100	00000000 00001111	INVALID	INVALID
13	00000000 00000100	INVALID	INVALID	INVALID	INVALID
14	00000000 00000010	00000000 00000011	INVALID	INVALID	INVALID
15	00000000 00000001	INVALID	INVALID	INVALID	INVALID

Byte marks for lqv, lrv depends on SuVuA[3:0]

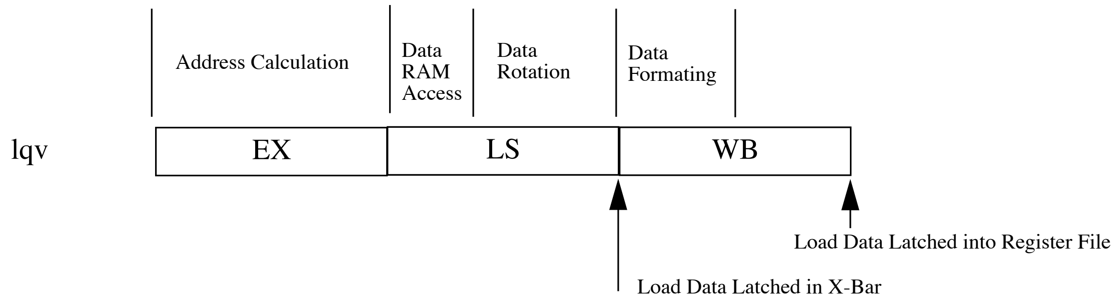
4.5.8.6 X-Bar

The X-Bar is the structure of the load-store mechanism that does the data reformating and rotation to suit the different load store formats and instructions.

All Scalar/Vector loads and stores go through the X-Bar. Thre are 2 main parts of the X-Bar. Data rotation and data formatting.

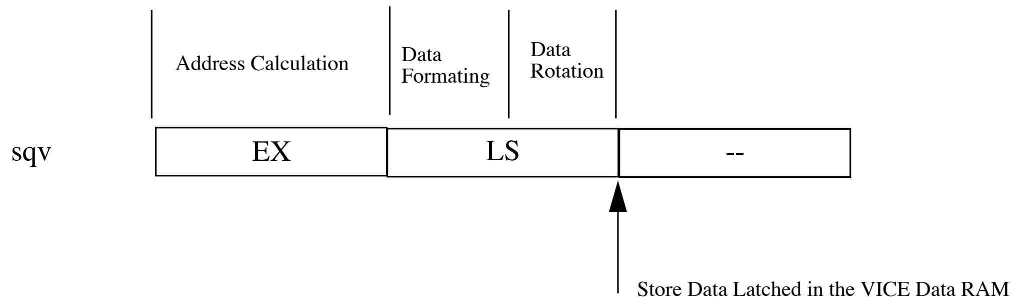
4.5.8.6.1 Loads

Data rotation is done in the LS pipe stage. The result is then clocked into a flip-flop and formatted in the WB pipe stage after which it is then written into the Scalar or Vector Unit Register File.



4.5.8.6.2 Stores

For stores, the store data is 1st formatted and then rotated before it is sent and written into the VICE Data RAM

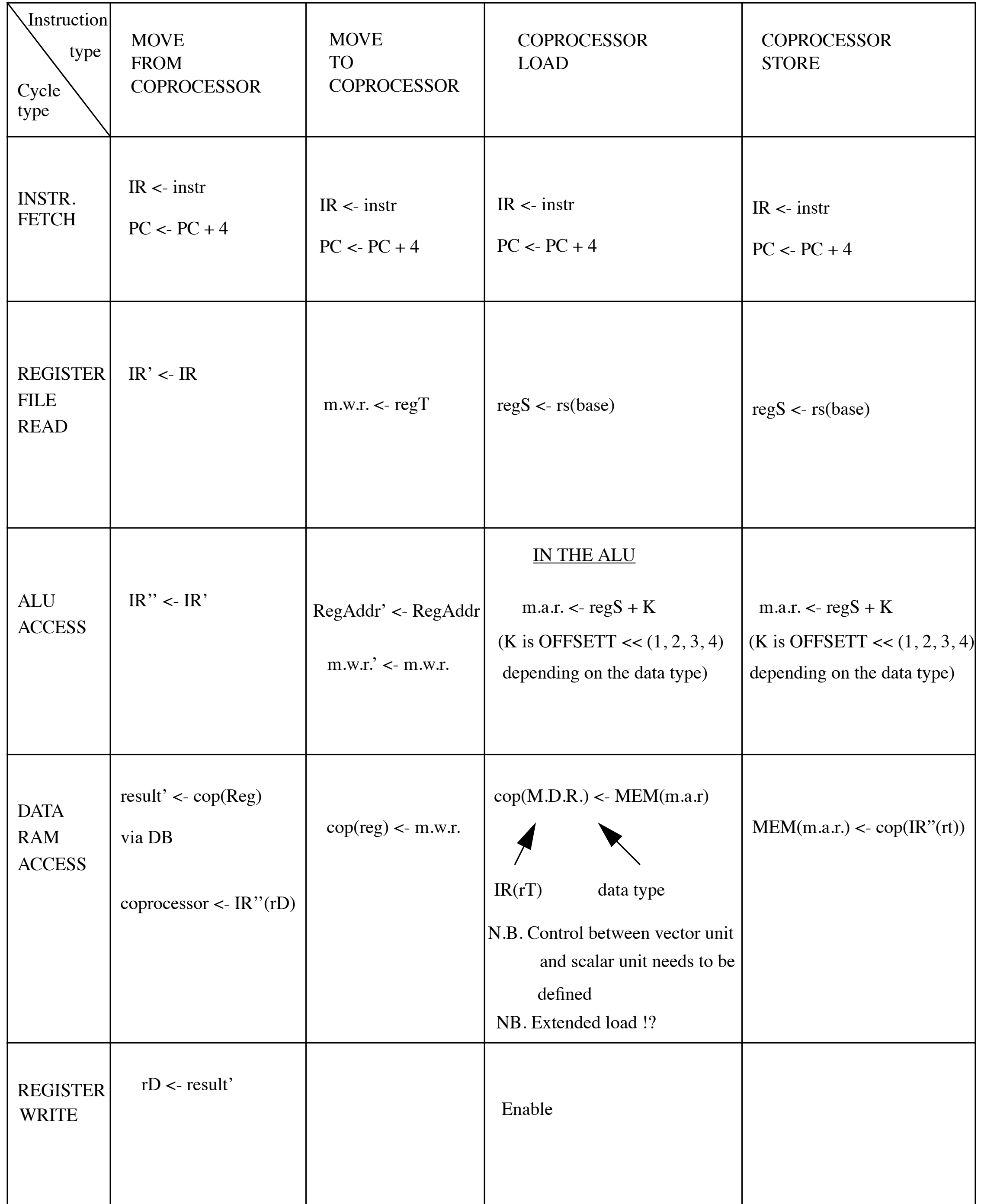


4.5.8.7 Detailed Operation of the Scalar Unit

Cycle type \ Instruction type	ALU	LOAD	STORE
INSTRUCTION FETCH	IR <- instr(ϕ2) PC <- PC + 4	IR <- instr(ϕ2) PC <- PC + 4	IR <- instr(ϕ2) PC <- PC + 4
REGISTER FILE READ	regT <- rT(ϕ2) regS <- rS(ϕ2) imm <- IR[IMM]	regS <- rS(base) regT <- OFFSET (sign extended or zero extended)	regS <- rs(base) regT <- rT T_port_of_ALU <- OFFSET
ALU ACCESS	result <- regT op RegS (immediate)	m.a.r <- regS + OFFSET	m.a.r. <- regS + OFFSET m.w.r <- regT
DATA RAM ACCESS	result' <- result	m.d.r. <- MEM(addr)	MEM(m.a.r.) <- m.w.r.' (deal with data type)
REGISTER WRITE	rD <- result'	rt <- m.d.r (byte h byte l hw h)	NOP

Cycle type \ Instruction type	JUMP J	JUMP AND LINK JAL	JUMP REGISTER JR
INSTRUCTION FETCH	IR <- instr(ϕ2) PC <- PC + 4 PC’ <- PC + 4	IR <- instr(ϕ2) PC <- PC + 4 PC’ <- PC + 4	IR <- instr PC <- PC + 4
REGISTER FILE READ	PC <- $PC_{31:28} \| \text{target} \| O^2$	PC<- $PC_{31:28} \| \text{target} \| O^2$ PC’’ <- PC’	wire, no cycle delay tempPC <- rS PC <- tempPC
ALU ACCESS		<u>BRANCH ADDER</u> link address result <- PC” + 4	
DATA RAM ACCESS		result’ <- result	
REGISTER WRITE		r31 <- result’	

<table>
<tr><th>Instruction type / Cycle type</th><th>JUMP REGISTER AND LINK
JRL</th><th>BRANCH</th><th>BRANCH AND LINK</th></tr>
<tr><td>INSTRUCTION FETCH</td><td>IR <- instr
PC <- PC + 4
PC’ <- PC + 4</td><td>IR <- instr
PC <- PC + 4
PC’ <- PC + 4</td><td>IR <- instr
PC <- PC + 4
PC’ <- PC + 4</td></tr>
<tr><td>REGISTER FILE READ</td><td>wire, no cycle delay
tempPC <- rS
PC <- tempPC
PC’’ <- PC’</td><td>regS <- rS
regT <- rT
branch address
target <- PC’ + OFFSET</td><td>target <- OFFSET”||OFFSET||O^2
- - - - - - - - - - - - - - - -
<u>BRANCH ADDER</u>
PC <- PC + target
PC” <- PC’</td></tr>
<tr><td>ALU ACCESS</td><td>PC ADDER
result <- PC’’ + 4</td><td>ALU:
result <- rS - rT
if(condition) PC <- target</td><td><u>BRANCH ADDER:</u>
result <- PC” + 4 /* link address */
<u>MAIN ALU:</u>
IF ($\overline{\text{CONDITION}}$) NULLIFY
IF STAGE
(stuff NOOP)</td></tr>
<tr><td>DATA RAM ACCESS</td><td>result’ <- result</td><td></td><td>result’ <- result</td></tr>
<tr><td>REGISTER WRITE</td><td>rD <- result’</td><td></td><td>r31 <- result’</td></tr>
</table>



Instruction type / Cycle type	MOVE FROM COPROCESSOR	MOVE TO COPROCESSOR	COPROCESSOR LOAD	COPROCESSOR STORE
INSTR. FETCH	IR <- instr PC <- PC + 4	IR <- instr PC <- PC + 4	IR <- instr PC <- PC + 4	IR <- instr PC <- PC + 4
REGISTER FILE READ	IR’ <- IR	m.w.r. <- regT	regS <- rs(base)	regS <- rs(base)
ALU ACCESS	IR” <- IR’	RegAddr’ <- RegAddr m.w.r.’ <- m.w.r.	IN THE ALU m.a.r. <- regS + K (K is OFFSETT << (1, 2, 3, 4) depending on the data type)	m.a.r. <- regS + K (K is OFFSETT << (1, 2, 3, 4) depending on the data type)
DATA RAM ACCESS	result’ <- cop(Reg) via DB coprocessor <- IR”(rD)	cop(reg) <- m.w.r.	cop(M.D.R.) <- MEM(m.a.r) IR(rT) data type N.B. Control between vector unit and scalar unit needs to be defined NB. Extended load !?	MEM(m.a.r.) <- cop(IR”(rt))
REGISTER WRITE	rD <- result’		Enable	

Instruction type / Cycle type	MOVE CONTROL FROM COPROCESSOR	MOVE CONTROL TO COPROCESSOR	
INSTR. FETCH	IR <- instr PC <- PC + 4	IR <- instr PC <- PC + 4	
REGISTER FILE READ		MWR <- regT	
ALU ACCESS			
DATA RAM ACCESS		cop(reg) <- MWR’	
REGISTER WRITE	rD <- cop(reg)		

4.6 MSP Vector Unit

4.6.1 Functional Overview

The Vector Unit (VU) of the Media Signal Processor (MSP) acts as a coprocessor to the MSP Scalar Unit. The standard MIPS instruction set has been extended with a set of VU instructions to allow it to perform arithmetic and logical operations on individual data elements within a data word using the fixed-point format described below. Data words are treated as vectors of Nx1elements, where N is either 8 or 16. The VU is organized as a Single Instruction, Multiple Data (SIMD) compute engine where each instruction performs the same operation on each element within a vector in parallel. The Vector Unit is organized as 8 identical slices, each 16 bits wide, which are connected in parallel. Figure Appx-B.1 is a RTL block diagram of the hardware contained in a single slice of the VU. Figure Appx-B.2 shows how the individual slices are connected to form a complete VU.

4.6.2 VU Features

- **Very Wide data path**

The Vector Unit can operate on data that is the full width of the local on-chip memories, up to 128-bits. This allows parallel operations on 8 or 16 vector elements in one cycle where each element is 16-bits or 8-bits respectively.

- **32 General Purpose Registers**

The VU has 32 128-bit general purpose vector registers which are visible to the programer and can be used to store intermediate results. Within each register, data may be written, or read, as bytes (8-bits), short-words (16-bits), words (32-bits), double-words (64-bits) or quad-words (128-bits). In this document, data elements containing 16 bits are also referred to as “halfwords”, instead of shortwords. The two terms are interchangeable.

- **Special Purpose Registers**

In addition to the vector registers, the Vector Unit has four control registers which are visible to the programmer as well, the Vector Compare Code Register, the Vector Carry Out Register, the Vector Compare Extension Register, and the Vector CLamp Register. The VCCR, VCOR and VCER save state from the results of previously executed instructions for use with subsequent instructions. The VCLR is set when an instruction generates a saturated result that requires clamping before being stored into the VRegister File. The Vector Unit also features an Accumulator with Vector Unit instructions specifically designed to use it. The Accumulator can be used as both a source and a destination in consecutive cycles without causing pipe stalls for data hazards, as would normally be the case with the VRegister File. It can be thought of as a register with data-forwarding.

- **Load/Store Instruction Set Architecture**

Like the Scalar Unit (SU), the VU features a Load/Store architecture in which memory data being read or stored must first be placed into a Vector Register (VR). Loads to and stores from the VRs take one cycle to execute and have a 2-cycle delay slot associated with them. VRegister Loads and stores may be overlapped with other vector operations.

- **Tightly Coupled Interface**

The Vector Unit resides on the same chip as the rest of the MSP and has tightly coupled connections with the Scalar Unit as well as both of the on-chip memories, the D-RAM and I-RAM.

- **Overlapped Execution of Instructions with Scalar Unit**

For maximum performance, the Vector Unit can execute instructions in parallel with the execution of instructions on the Scalar Unit.

4.6.3 VU Programming Model

The Vector Unit has three groups of registers available to the programmer; 32 general-purpose Vector Registers, organized as a register file, 4 Control Registers, and an Accumulator.

The Vector Registers are read or written only through the use of Vector Unit instructions. Vector Registers are used as sources of operands and the destination of the results of Vector Unit instructions.

The Control Registers, VCCR, VCOR, VCER, and VCLR are written as a result of a VU computational instruction. For each of the 8 halfwords in the VU, there are 2 bits of VCC Register, 2 bits of VCO Register, 1 bit of VCE Register and 2 bits of VCL Register. The lower 8 bits of the VCCR are used to store the results of vector Select instructions for use in a follow-on Vector Merge instruction. The lower 8 bits of the VCOR are used to store a carry-out or borrow-out for use as an carry-in or borrow-in to a follow-on add or subtract instruction. The higher 8 bits of the VCOR are set if the operands are not equal. Once set, the VCOR bits are 'sticky', they will remain set unless a subsequent instruction explicitly resets them.

In addition to the general-purpose Vector Registers and the Control Registers, there is also a 48-bit Accumulator. As the name implies, this register is mostly used to store intermediate add, or subtract, results generated by one instruction with the intermediate add, or subtract, results generated by either previous or successive instructions. Certain instructions specify that the Accumulator is to be used, either as a source or a destination, or both. How the Accumulator is used is defined by the instruction. For those instructions which do not specifically call for the use of the Accumulator, the contents of the Accumulator are unmodified by the instruction. Some instructions only modify a portion of the Accumulator, leaving the rest of the bits unmodified.

4.6.4 Binary Fixed-Point Format

The VU treats all operands as binary fixed-point data, either bytes (8 bits) or halfwords (16 bits). Byte operands are interpreted as unsigned quantities with a range, r, of $0 <= r <= 255$. Halfword operands may be treated as one of three data types; signed integer with a range, r, of $-32768 <= r <= 32767$, unsigned integer with a range, r, of $0 <= r <= 65536$, or signed fraction with a range, r, of $-1.0 <= r <= .99996948$.

Each instruction has an implied data type associated with it. If the results of an instruction are outside of the defined range for that instruction, the results stored back into the Vector Registers are forced to the nearest of the max or min value for the range associated with the instruction. See the section on Clamping and Rounding for more explanation.

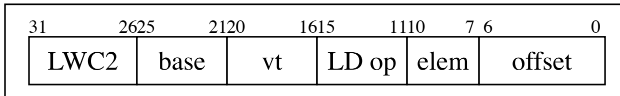
4.6.5 Instruction Set Overview

All the Vector Unit instructions are 32 bits long and are word aligned. They can be classified into the following groups.

- **Load/Store Instructions** for moving data between the Vector Unit register file and on-chip local memory.
- **Move Instructions** for moving data between the Vector Unit register file and the Scalar Unit register file.
- **Move Control Instructions** for moving data between the Vector Unit control registers and the Scalar Unit register file.
- **Computational Instructions** for performing calculations.

Note that there are no VU instructions for controlling the flow of programs. The Scalar Unit performs this function for the Vector Unit. In this sense, the Vector Unit can be thought of as a 'slave' of the Scalar Unit.

4.6.5.1 Load Instructions



Load instructions move data between the on-chip data RAM and the Vector Unit's Register File. All the load Instructions for the Vector Unit are mapped into the MIPS opcode for Load Word Coprocessor 2, lwc2. Although these instructions move data between the VU Register File and memory, they actually execute in the Scalar Unit pipeline. The Scalar Unit has its own read/write ports into the Vector Unit's Register File. This allows for the overlapping of data movement and computations in the VU. There are many types of load instructions, depending on the size, distribution and alignment of the data in memory.

All of the load instructions, except for the Rest and Transpose instructions, operate in the same general manner. The general operation of the load instructions is as follows: The contents of the SU register specified by the *base* field of the instruction are added to the *offset* field to generate an effective byte address in memory. The contents of the base register represent a byte address in memory. *Offset* is an index based on the size of the data item to be moved and is required to be properly weighted before adding to the contents of the base register. If the data item to be moved is not a single byte, *offset* will be shifted left 1, 2, 3 or 4 bit positions to achieve the proper weighting. The data item starting at the effective byte address in memory is loaded into the vector register specified by the *vt* field. A read from the data RAM accesses 2 double-word elements, address and address+8. With this addressing scheme, any data read requiring less than a quad-word of data will always return the requested data. In some cases where the data size is a quad-word, the combination of the byte address and item size can cause some portion of the desired data to be in the next higher double-word in memory than that addressed by address+8. In this case, only the portion of the data in the currently accessed memory double-words is fetched. The remaining data that is not fetched, can be accessed with a second instruction using the Load Rest instruction. The *element* field specifies the alignment for storing the data in the vector register. The *element* field is treated as a byte address within the vector register. For byte loads, the *element* field can be any number between 0 and 15. For larger data items, the *element* field is restricted to values that will result in all the data to be moved being stored into the vector register specified by *vt*.

The operation of the Load/Store Rest instructions is slightly different from the other Load/Store instructions. In memory, all the other Load/Store instructions operate on the bytes between the effective byte address and byte 15 of the accessed quadword (doubleword pair), inclusive. The Rest instructions move the bytes in memory between byte 0 of the accessed quadword and the effective byte address, inclusive. The operation of the Load/Store Rest instruction is similarly different from the other Load/Store instructions with respect to the VRegister File. All the other Load/Store instructions operate on the elements slices left-justified. The Rest instructions operate on the element slices right-justified.

LBV, Load Byte To Vector Unit (coprocessor 2)

LBV vt[element], offset(base)

The byte at the effective address in memory is loaded into the byte of vector register *vt* as specified by *element*.

LSV, Load Shortword To Vector Unit (coprocessor 2)

LSV vt[element], offset(base)

The halfword at the effective address in memory is loaded into the halfword of vector register *vt* as specified by *element*.

LLV, Load Longword To Vector Unit (coprocessor 2)

LLV vt[element], offset(base)

The longword at the effective address in memory is loaded into the longword of vector register *vt* as specified by *element*.

LDV, Load Doubleword To Vector Unit (coprocessor 2)

LDV vt[element], offset(base)

The doubleword at the effective address in memory is loaded into the doubleword of vector register *vt* as specified by *element*.

LQV, Load Quadword To Vector Unit (coprocessor 2)

LQV vt[element], offset(base)

The quadword at the effective address in memory is loaded into vector register *vt*.

LPV, Load Pack Signed To Vector Unit (coprocessor 2)

LPV vt[element], offset(base)

The 8 consecutive bytes starting at the effective address in memory are loaded into the most significant byte of each halfword of vector register *vt*. The least significant byte of each halfword of vector register *vt* is zeroed.

LUV, Load Pack Unsigned To Vector Unit (coprocessor 2)

LUV vt[element], offset(base)

Each byte of the 8 consecutive bytes starting at the effective address in memory is padded with a leading zero. The padded bytes are loaded into the most significant bits of each halfword of vector register *vt*. The least significant bits of each halfword of vector register *vt* are zeroed.

LXV, Load Sign Extended Byte To Vector Unit (coprocessor 2)

LXV vt[element], offset(base)

The 8 consecutive bytes starting at the effective address in memory, with each byte sign-extended to become a full 16-bit shortword element, are loaded into each halfword of vector register *vt*.

LZV, Load Zero Extended Byte To Vector Unit (coprocessor 2)

LZV vt[element], offset(base)

The 8 consecutive bytes starting at the effective address in memory, with each byte zero-extended to become a full 16-bit shortword element, are loaded into each halfword of vector register *vt*.

LHV, Load Alternate Bytes To Vector Unit (coprocessor 2)

LHV vt[element], offset(base)

The 8 alternating bytes, starting at the effective address in memory, are each padded with a leading zero. The padded bytes are loaded into the most significant bits of each halfword of vector register *vt*.

LFV, Load Fourths To Vector Unit (coprocessor 2)

LFV vt[element], offset(base)

Every fourth byte, starting at the effective address in memory is padded with a leading zero. The padded bytes are loaded into the most significant bits of each halfword of vector register *vt* as specified by *element*.

LAV, Load Alternate Shortwords To Vector Unit (coprocessor 2)

LAV vt[element], offset(base)

The 4 alternating halfwords starting at the effective address in memory are loaded into the most significant bits of the halfwords of vector register *vt* specified by *element*.

LTV, Load Transposed To Vector Unit (coprocessor 2)

LTV vt[element], offset(base)

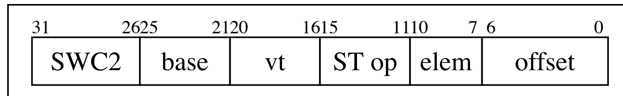
The 8 halfwords starting at the effective address in memory are loaded into 8 consecutive vector registers, starting with the vector register specified by *vt*. The actual register number for each slice is a function of physical slice number and *element*. This instruction takes a single horizontal line of data in memory and loads it into a block of vector registers on a diagonal.

LTWV, Load Transposed Wrapped To Vector Unit (coprocessor 2)

LTWV vt[element], offset(base)

The 8 halfwords starting at the effective address in memory are loaded into 8 consecutive vector registers, starting with the vector register specified by *vt*. The actual register number for each slice is a function of physical slice number and *element*. This instruction takes a single horizontal line of data in memory and loads it into a block of vector registers on a diagonal. The bits of *vt* that are written are also a function of the physical slice number and *element*. In this useage, the *element* field can be thought of as specifying a left-shift of the data in memory by that number of elements.

4.6.5.2 Store Instructions



All the store Instructions for the Vector Unit are mapped into the opcode for Store Word Coprocessor 2, swc2. Like the VU load instructions, these instructions actually execute in the Scalar Unit pipeline. For each load instruction, there is a corresponding store instruction.

The operation of the store instructions is the same as the load instructions except that the flow of data is from the VU Register File to the data RAM. The data item in the vector register specified by *vt* and starting at the element specified by *element* is loaded into memory, starting at the effective byte address. Memory accesses to the data RAM are double-word aligned. A write to the data RAM can access 1 or 2 double-word elements. Any data write that modifies less than a quad-word of data will always write all the data in memory. In some cases where the data size is a quad-word, the combination of the byte address and item size can cause some portion of the data to be written in the next higher double-word in memory than that addressed by address+8. In this case, only the portion of the data that fit in the currently accessed memory double-words is written. The remaining data that is not written, can be stored with a second instruction using the Store Rest instruction. The *element* field specifies the alignment for reading the data from the vector register. For byte stores, the *element* field can be any number between 0 and 15. For larger data items, the *element* field is restricted to values that will result in all the data to be moved being read from the vector register specified by *vt*.

SBV, Store Byte From Vector Unit (coprocessor 2)

SBV vt[element], offset(base)

The byte of vector register *vt*, as specified by *element*, is stored into the byte in memory at the effective byte address.

SSV, Store Shortword From Vector Unit (coprocessor 2)

SSV vt[element], offset(base)

The halfword of vector register *vt*, as specified by *element*, is stored into the halfword in memory at the effective address.

SLV, Store Longword From Vector Unit (coprocessor 2)

SLV vt[element], offset(base)

The longword of vector register *vt*, as specified by *element*, is stored into the longword in memory at the effective address.

SDV, Store Doubleword From Vector Unit (coprocessor 2)

SDV vt[element], offset(base)

The doubleword of vector register *vt*, as specified by *element*, is stored into the doubleword in memory at the effective address.

SQV, Store Quadword From Vector Unit (coprocessor 2)

SQV vt[element], offset(base)

The vector register *vt* is stored into the quadword in memory at the effective address.

SPV, Store Pack Signed From Vector Unit (coprocessor 2)

SPV vt[element], offset(base)

The most significant byte of each halfword of vector register *vt* is stored into 8 consecutive bytes in memory, starting at the effective address.

SUV, Store Pack Unsigned From Vector Unit (coprocessor 2)

SUV vt[element], offset(base)

Bits 14:7 from each halfword of vector register *vt* are stored into 8 consecutive bytes in memory, starting at the effective address.

SXV, Store Sign Extended Byte From Vector Unit (coprocessor 2)

SXV vt[element], offset(base)

The least significant byte of each halfword of vector register *vt* is stored into 8 consecutive bytes starting at the effective address in memory.

SZV, Store Zero Extended Byte From Vector Unit (coprocessor 2)

SZV vt[element], offset(base)

The least significant byte of each halfword of vector register *vt* is stored into 8 consecutive bytes starting at the effective address in memory.

SHV, Store Alternate Bytes From Vector Unit (coprocessor 2)

SHV vt[element], offset(base)

Bits 14:7 from each slice of vector register *vt* are stored into 8 alternating byte in memory, starting at the effective address.

SFV, Store Fourths From Vector Unit (coprocessor 2)

SFV vt[element], offset(base)

Bits 14:7 from each of either the 4 high-order, or the 4 low-order, slices of vector register *vt*, as specified by *element*, are stored into memory at every fourth byte, starting at the effective address in memory.

SAV, Store Alternate Shortwords From Vector Unit (coprocessor 2)

SAV vt[element], offset(base)

Four halfwords from either the 4 high-order, or the 4 low-order, slices of vector register *vt*, as specified by *element*, are stored into 4 alternating halfwords in memory, starting at the effective address in memory.

STV, Store Transposed To Vector Unit (coprocessor 2)

STV vt[element], offset(base)

Single elements from 8 consecutive vector registers, starting with the vector register specified by *vt* are stored into 8 the halfwords in memory starting at the effective address in memory. The actual register number for each slice is a function of physical slice number and *element*. This instruction takes data from a block of vector registers on a diagonal and stores it into a single horizontal line in memory.

SWV, Store Transposed Wrapped To Vector Unit (coprocessor 2)

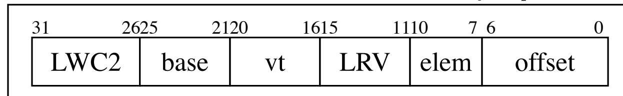
SWV vt[element], offset(base)

Single elements from 8 consecutive vector registers, starting with the vector register specified by *vt* are stored into 8 the halfwords in memory starting at the effective address in memory. The actual register number for each slice is a function of physical slice number and *element*. This instruction takes data from a block of vector registers on a diagonal and stores it into a single horizontal line in memory. The bits of *vt* that are read from are also a function of the physical slice number and *element*. In this useage, the *element* field can be thought of as specifying a left-shift of the data in memory by that number of elements.

4.6.5.3 Load/Store Rest Instructions

The operation of the Load/Store Rest instructions is slightly different from the other Load/Store instructions. In memory, all the other Load/Store instructions operate on the bytes between the effective byte address and byte 15 of the accessed quadword (doubleword pair), inclusive. The Rest instructions move the bytes in memory between byte 0 of the accessed quadword and the effective byte address, inclusive. The operation of the Load/Store Rest instruction is similarly different from the other Load/Store instructions with respect to the VRegister File. All the other Load/Store instructions operate on the elements slices left-justified. The Rest instructions operate on the element slices right-justified.

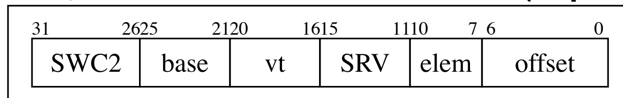
LRV, Load Rest To Vector Unit (coprocessor 2)



LRV vt[element], offset(base)

The contents of the scalar register *base*, added to the *offset* (*offset* is shifted left four bit positions before adding), form the effective byte address in memory. The bytes between the effective address and byte 0, inclusive, of the quadword-aligned address in memory are loaded into vector register *vt*. *Element* is restricted to a value of 0.

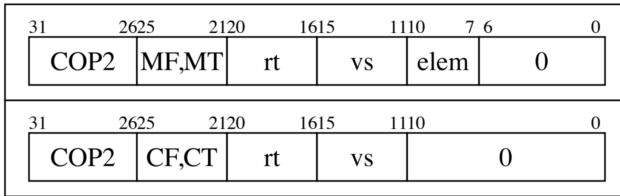
SRV, Store Rest From Vector Unit (coprocessor 2)



SRV vt[element], offset(base)

The contents of the scalar register *base*, added to the *offset* (*offset* is shifted left four bit positions before adding), form the effective byte address in memory. The vector register *vt* is stored into the bytes between the effective address and byte 0, inclusive, of the quadword-aligned address in memory. *Element* is restricted to a value of 0.

4.6.5.4 Move Instructions



The various Move instructions move data between the VRegister File, or the Vector Control Registers, and the Scalar Unit's Register File. For the MFC2 and MTC2 instructions, the *element* field is used to specify which 16-bit element of the vector register, *vs*, is used as the source or destination. For the CFC2 and CTC2 instructions, *vs* is used to specify which one of the four Vector Control Registers is being used and the *element* field is ignored. When moving from the VRegister File to the SU Register File, data is sign extended.

MFC2, Move From Vector Unit (coprocessor 2)

MFC2 rt, vs

Scalar register *rt* is loaded with the contents of vector register *vs*. The actual 16-bit slice of *vs* that is loaded is specified by *element*. The 16-bit value is sign extended to 32 bits as it is loaded into the scalar register.

MTC2, Move To Vector Unit (coprocessor 2)

MTC2 rt, vs

The lower 16 bits of scalar register *rt* are stored into the vector register *vs*. The actual 16-bit slice of *vs* that is stored into is specified by *element*.

CFC2, Move Control Word From Vector Unit (coprocessor 2)

CFC2 rt, VCC

CFC2 rt, VCO

CFC2 rt, VCE

Scalar register *rt* is loaded with the contents of the Vector Control Register specified by *vs*.

CTC2, Move Control Word To Vector Unit (coprocessor 2)

CTC2 rt, VCC

CTC2 rt, VCO

CTC2 rt, VCE

The lower 16 bits of scalar register *rt* are stored into the Vector Control Register specified by *vs*.

4.6.5.5 Computational Instructions

Computational instructions perform arithmetic operations on data stored in the VRegister File. The data format is binary fixed-point. All the Vector Unit computational instructions are mapped into the MIPS opcode for Coprocessor Operation 2. They use a three operand format with two sources, specified by *vt* and *vs*, and a destination, usually specified by *vd*. The contents of the vector register specified by *vs* are always interpreted as vector elements. The contents of the vector register specified by *vt* may used as array of vector elements or they may used as a scalar value, depending on the value of the *element* field. The results of a computational instruction are treated as an array of vector elements.

4.6.5.5.1 Element Selection

The use of the *element* field to select the elements of *vt* to be used as operands is shown in the table below.

TABLE 72. Element Selection for Computational Instructions

Element[3:0]	Vector Elements of vt Selected
0000	Operand is a vector, all 8 elements of vt are used with corresponding 8 elements of vs.
001X	Operand is a scalar, 1 of 2 elements is selected from each quarter-group of elements of vt. Element selected is only applied across quarter of vs from which it is selected.
01XX	Operand is a scalar, 1 of 4 elements is selected from each half-group of elements of vt. Element selected is only applied across half of vs from which it is selected.
1XXX	Operand is a scalar, 1 of 8 elements is selected from vt. Element selected is applied across all 8 elements of vs.

To illustrate with an example, let element[3:0] = 0011. From vector elements 0 and 1 of *vs*, element 1 will be selected and applied across both elements 0 and 1 of *vt*. From vector elements 2 and 3 of *vs*, element 3 will be selected and applied across both elements 2 and 3 of *vt*. From vector elements 4 and 5 of *vs*, element 5 will be selected and applied across both elements 4 and 5 of *v*t. From vector elements 6 and 7 of *vs*, element 7 will be selected and applied across both elements 6 and 7 of *vt*.

4.6.5.5.2 Rounding and Clamping

Depending on the instruction operation, and its implied data type, various rounding modes and saturation clamping ranges are applied to the calculated results of an instruction. In the case of multiplication for quantization, the rounding mode applied is dependent on the sign of the results. Saturation clamping, when applied, clamps to one of several different ranges. When saturation clamping does occur, a bit is set in the VCL Register. For some instructions, rounding is applied to results being stored into the Accumulator as well as to results being stored into the VRegister File. Clamping is only applied to results being stored into the VRegister File.

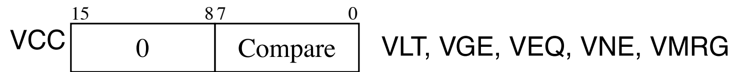
4.6.5.5.3 Vector Control Registers VCO, VCC and VCE

The Vector Control Registers VCO and VCE are used as inputs into certain of the computational intructions to support double-precision operations. Control Registers VCO and VCC are set as part of the output of some of the computational instructions to record some important state about the operation of the instruction than can be examined later on, usually with a MFC2 instruction. The figures below show the bit definitions for each register how they are use by each instruction.

4.6.5.5.3.1 Adds/Subtracts

Each slice of the Vector Unit contributes an Equal/Not_Equal bit(1=Not_Equal) and a Carry/Borrow bit(1=Operation results in Carry/Borrow) to the VCO Register. The 8 Equal bits are grouped into the high-order bits of the register and the 8 Carry bits are grouped into the low-order bits. Slice 0 generates/uses bits [15,7] while slice 7 generates/uses bits [8,0]. VADDC and VSUBC set the bits based on the results of the operation. VADD, VACC, VSUB, VSUC and VSUT use the Carry bits as inputs and clear all the VCO bits on its output. To use this feature for double-precision, first perform the VADDC/VSUBC on the lower half of the operands to set the VCO, then perform the VADD/VSUB on the upper half of the operand. Note that this will clear the VCO after both halves have been processed so that subsequent VADDs on single precision operands will operate correctly.

4.6.5.5.3.2 Compares

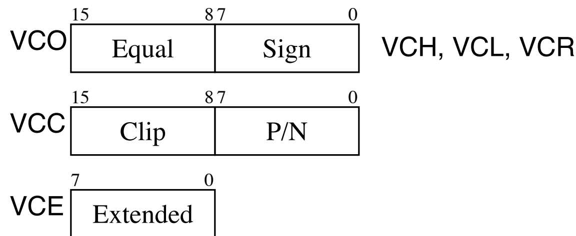


The VCC register is also 16 bits. The compare instructions only use the 8 low-order bits of the VCC..Slice 0 generates bit [7] and slice 7 generates bit [0]. For single precision compares, VLT, VGE, VEQ and VNE set their respective Compare bits for each slice if the comparison is true. The 8 high-order bits of VCC are allways cleared by a compare instruction. VMRG does not set the VCC, it only used the bits as set by a previous compare instruction as input into its selection operation.

For double-precision compares, VLT, VGE, VEQ and VNE also use the VCO register as an input into the decision process. For this to work properly, the VCO must have been previously set by a VSUBC instruction which operated on the upper half of the operands. The comparison instruction completes the double-precision operation.

Whether the comparison is a single precision or a double-precision operation, the compare instructions allways clear the VCO register.

4.6.5.5.3.3 Clips, Clamps, Crimps and Cramps

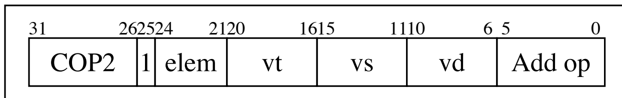


To perform a single-precision clip, use either VCL or VCR. VCL clips to a 2's complement range, while VCR clips to a 1's complement range. For the clip instructions, the VCC have a different meaning. The 8 high-order bits now represent the clip condition. They are set if *vs* is outside of the range of *vt* and cleared if *vs* is inside the range of *vt*. The 8 low-order bits represent the direction of the clip indicated by the high-order bits. This can also be thought of as sign(*vs*). VCL and VCR set VCC on output based on the result of the clip operation.

For double-precision clips, VCH is performed first on the upper-half of the operands. As part of its output, VCH sets the VCO register based on the results of it's operation, but in this case, the 8 low-order bits of VCO have a different meaning. These Sign bits now represent the status of the sign bits of the two operands. They are set if the signs of *vs* and *vt* are opposite and cleared if the signs are equal. A new register is used with VCH, the VCE register. It is only 8 bits. Bits in VCE are set if the signs of *vs* and *vt* are not equal and the clip comparison generates a result of -1. The second half of a double-percision clip is performed with VCL. It uses VCO and VCE as inputs to determine the final result of the clip operation and sets VCC accordingly.

Whether the clip is a single precision or a double-precision operation, VCL and VCR always clear the VCO, and VCE registers on output.

4.6.5.5.4 Add/Subtract Halfwords Instructions



These Add and Subtract Instructions operate on halfword elements. These instructions do not perform rounding. The results, clamped to the range r, -32768 <= r <= 32767, are stored into the VRegister File. Unclamped results are stored into the Accumulator(with and without accumulation on the results currently in the Accumulator). During the accumulation stage, the partial results that were generated during the EX1 stage are aligned with bits [31:16] of the Accumulator before being stored there.

The Accumulators for each Vector Unit slice contain 48 bits. Because the Add/Subtract Halfword Instructions align partial results with ACC[31:16], bits [47:32] of the Accumulator are available for data growth while accumulating data. Up to $2^{16} - 1$ consecutive additions can be performed before results overflow occurs.

VADD, Vector Add

VADD vd, vs, vt[element]

The halfword element(s) of vector register *vt*, as specified by *element*, is added to each halfword element of vector register *vs*. The result is stored into the Accumulator and from there, it is stored into vector register *vd*.

VACC, Vector Add Halfwords and Accumulate

VACC vd, vs, vt[element]

The halfword element(s) of vector register *vt*, as specified by *element*, is added to each halfword element of vector register *vs*. The result is accumulated and the accumulated result is then stored into vector register *vd*.

VSUB, Vector Subtract

VSUB vd, vs, vt[element]

The halfword element(s) of vector register *vt*, as specified by *element*, is subtracted from each halfword element of vector register *vs*. The result is stored into the Accumulator and from there, it is stored into vector register *vd*.

VSUC, Vector Subtract Halfwords and Accumulate

VSUC vd, vs, vt[element]

The halfword element(s) of vector register *vt*, as specified by *element*, is subtracted from each halfword element of vector register *vs*. The result accumulated and the accumulated result is stored into vector register *vd*.

VSUT, Vector Scalar Subtract

VSUT vd, vs, vt[element]

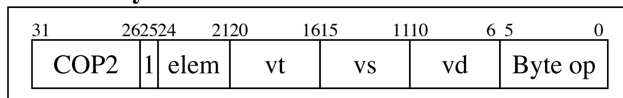
Each halfword element of vector register *vs* is subtracted from the halfword element(s) of vector register *vt*, as specified by *element*. The result is stored into the Accumulator and from there, it is stored into vector register *vd*.

VABS, Absolute Value of Vector

VABS vd, vs, vt[element]

The halfword element(s) of vector register *vt*, as specified by *element*, is conditionally negated, based on the sign of each halfword element of vector register *vs*, and stored into the Accumulator and from there, it is stored into vector register *vd*.

4.6.5.5.5 Byte Instructions



Byte Instructions operate on byte elements from the VRegister File. They store results into the Accumulator (with and without accumulation on the results currently in the Accumulator) during the EX2 stage and from

there, into the vector register specified by *vd* during the WBack stage. Vector element operands are always interpreted as unsigned integers with a range r, 0 <= r <= 255.

The byte Add/Subtract instructions treat the contents of vector register specified by *vt* as vector elements, therefore, the MSB of the *element* field must be 0. The remaining bits of *element*, depending on the instruction, may be used to specify the right-shift amount applied to the results as they are passed from the Accumulator to the VRegister File. The byte Multiply instructions use the *element* field as described in the section on element selection. The programmer should remember that element selection selects a 16-bit item, while the byte multiplies operate on a pair of 8-bit items. If the intent is to multiply 16 vector byte elements by the same scalar value, that value will have to be set in both the high and low bytes of the the register being selected as the scalar.

The byte Accumulators for each Vector Unit slice contain 18 bits. This allows for the growth of accumulated data from 8 bits up to a maximum of 18 bits before results overflow occurs. Up to 1024 consecutive additions can be performed without any loss of accuracy. Bits [17:16] of each of the Byte Accumulators are initialized when one of the following instructions is executed : VADDB, VSUBB, VMULB or VMULBN. Although these two high-order bits are available for calculations, they are not directly readable when byte results are stored into the VRegister File. They may be indirectly observed if the byte operation involves scaling (arithmatic shifting) of the Accumulator results on the way back to the VRegister File, as these bits will be shifted into the high-order bits of each byte during the shift.

After shifting, clamping is performed on the results as they are passed from the Accumulator to the VRegister File. The clamping range, r, is 0 <= r <= 255. Some byte instructions perform rounding on the results as they are stored into the Accumulator. The value added to the results for rounding is 2**(*element* - 1). The VCOR is ignored.

In this implementation of the MSP ISA, the Byte Accumulators are shared with main Accumulator for 16-bits operations. The Vector Unit Programmer should note that the contents of the any and all of the Accumulators will be undefined if 16-bit accumulate instructions are mixed with byte accumulate instructions and that either type of instruction will overwrite the previous contents of the accumulators.

VADDB, Vector Add Bytes

VADDB vd, vs, vt

Each byte element of vector register *vt* is added to each byte element of vector register *vs*. The results are stored into the Accumulator and from there, they are stored into vector register *vd*. *Element* is used both for rounding results stored into the Accumulator as well as for right-shifting results passed from Accumulator to VRegister File. *Element* is restricted to values e <= 7.

VACCB, Vector Add Bytes and Accumulate

VACCB vd, vs, vt, element

Each byte element of vector register *vt* is added to each byte element of vector register *vs*. The results are accumulated and the accumulated results are then stored into vector register *vd*. *Element* is used only for right-shifting results passed from Accumulator to VRegister File. *Element* is restricted to values e <= 7.

VSUBB, Vector Subtract Bytes

VSUBB vd, vs, vt, element

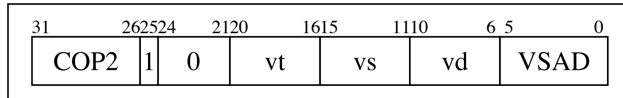
Each byte element of vector register *vt* is subtracted from each byte element of vector register *vs*. The results are stored into the Accumulator and from there, they are stored into vector register *vd*. *Element* is used both for rounding results stored into the Accumulator as well as for right-shifting results passed from Accumulator to VRegister File. *Element* is restricted to values e <= 7.

VSUCB, Vector Subtract Bytes and Accumulate

VSUCB vd, vs, vt, element

Each byte element of vector register *vt* is subtracted from each byte element of vector register *vs*. The results are accumulated and the accumulated results are then stored into vector register *vd*. *Element* is used only for right-shifting results passed from Accumulator to VRegister File. *Element* is restricted to values e <= 7.

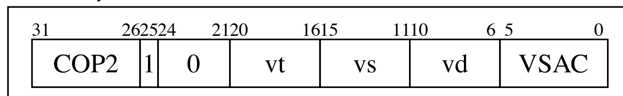
VSAD, Vector Sum of Absolute Differences



VSAD vd, vs, vt

Each byte element of vector register *vt* is subtracted from each byte element of vector register *vs*. For each slice, the absolute values of the high byte difference and the low byte difference are added together. The result of the addition is stored into the Accumulator. The vector register *vs* is written to vector register *vd*. *Element* is not used.

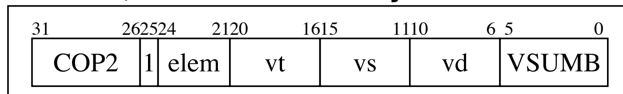
VSAC, Vector Sum of Absolute Differences and Accumulate



VSAC vd, vs, vt

Each byte element of vector register *vt* is subtracted from each byte element of vector register *vs*. For each slice, the absolute values of the high byte difference and the low byte difference are added together. The result of the addition is accumulated into the Accumulator. The vector register *vs* is written to vector register *vd*. *Element* is not used.

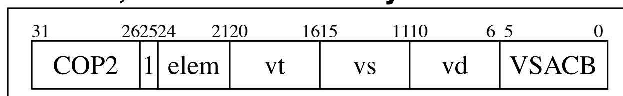
VSUMB, Vector Sum of Bytes



VSUMB vd, vs, vt, element

Each byte element of vector register *vt* is added to each byte element of vector register *vs*. Each pair of addition results, the high and low bytes of each slice, are then added together. The result of the addition is stored into the Accumulator and from there, it is stored into vector register *vd*. *Element* is used only for right-shifting results passed from Accumulator to VRegister File. *Element* is restricted to values e <= 7.

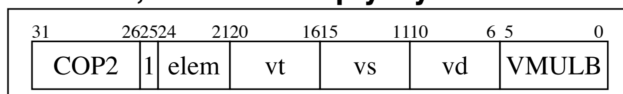
VSACB, Vector Sum of Bytes and Accumulate



VSACB vd, vs, vt, element

Each byte element of vector register *vt* is added to each byte element of vector register *vs*. Each pair of addition results, the high and low bytes of each Vector Unit slice, are then added together. The results of the addition are stored into the Accumulator with accumulation. *Element* is used only for right-shifting results passed from Accumulator to VRegister File. *Element* is restricted to values e <= 7.

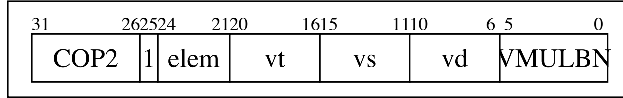
VMULB[1], Vector Multiply Bytes and Load Positive



VMULB vd, vs, vt[element]

The pair of bytes elements of vector register *vt*, as specified by *element*, is multiplied by the corresponding pair of byte elements of vector register *vs*. The result is loaded into the Accumulator without accumulation.

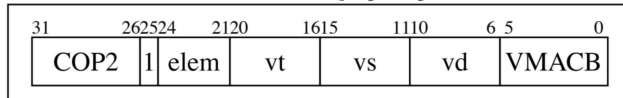
VMULBN[1], Vector Multiply Bytes and Load Negated



VMULBN vd, vs, vt[element]

The pair of bytes elements of vector register *vt*, as specified by *element*, is multiplied by the corresponding pair of byte elements of vector register *vs*. The result is negated (2's complement) and loaded into the Accumulator without accumulation.

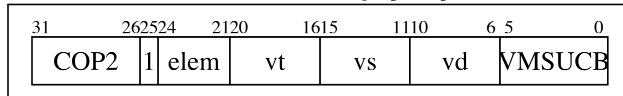
VMACB[2], Vector Multiply Bytes and Add to Accumulator



VMACB vd, vs, vt[element]

The pair of bytes elements of vector register *vt*, as specified by *element*, is multiplied by the corresponding pair of byte elements of vector register *vs*. The result is added to the contents of the Accumulator.

VMSUCB[3], Vector Multiply Bytes and Subtract from Accumulator



VMSUCB vd, vs, vt[element]

The pair of bytes elements of vector register *vt*, as specified by *element*, is multiplied by the corresponding pair of byte elements of vector register *vs*. The result is subtracted from the contents of the Accumulator.

4.6.5.5.6 Accumulator Instructions

These instructions are more specialized and tend to use the Accumulator as an operand source. Unless otherwise stated, there is no clamping or rounding performed by these instructions. The VCOR is ignored.

---

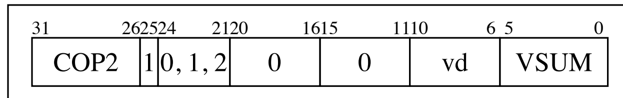
1. This instruction has been conditionally accepted as part of the instruction set. It is thought that it can be implemented with a slight re-organization of the hardware already included for 16x16 multiplies. If it turns out to be the case that implementation of this instruction would require major re-work, or that the resulting multiplier cannot meet timing, this instruction will be deleted from the instruction set. It should be noted that this instruction generates a 16-bit result which is stored into a 16-bit Accumulator. The programmer is cautioned that accumulation of multiplied bytes can easily cause the results to overflow, resulting in lost data.

1. Ibid.

2. This instruction has been conditionally accepted as part of the instruction set. It is thought that it can be implemented with a slight re-organization of the hardware already included for 16x16 multiplies. If it turns out to be the case that implementation of this instruction would require major re-work, or that the resulting multiplier cannot meet timing, this instruction will be deleted from the instruction set. It should be noted that this instruction generates a 16-bit result which is stored into a 16-bit Accumulator. The programmer is cautioned that accumulation of multiplied bytes can easily cause the results to overflow, resulting in lost data.

3. Ibid.

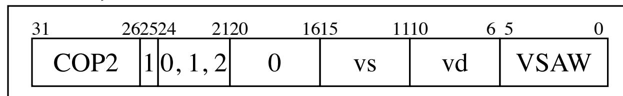
VSUM, Vector Accumulator Sum Reduction



VSUM vd, ACC[element]

Eight Accumulator elements, as specified by *element*[1], are added together to generate a signed 16-bit result(clamped). This result is stored into the low-order 16 bits of vector register *vd*. The *element* field is restricted to values of 0, 1, or 2, representing the high, middle and low thirds of each Accumulator slice. The *vs,* and *vt* fields are ignored.

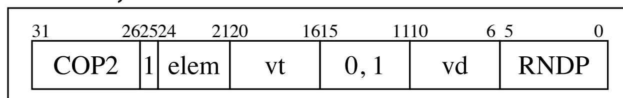
VSAW, Vector Save and Write Accumulator



VSAW vd, vs, ACC[element]

The contents of the Accumulator are stored into vector register *vd* and the contents of vector register *vs* are stored into the Accumulator. *Vt* is ignored. The *element* field, restricted to values of 0, 1, or 2, representing the high, middle and low thirds of each Accumulator slice, is used to select low, middle or high 16 bits of the Accumulator to read and write.

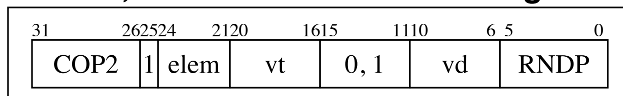
VRNDP, Round Accumulator if Positive



VRNDP vd, vs, vt[element]

The halfword element(s) of vector register *vt* specified by *element* is added to the Accumulator if the Accumulator currently contains a strictly positive number. If the Accumulator contains zero, or a negative number, it is not changed. If the *vs* field has a value of 0x1, then before adding the contents of *vt* to the Accumulator, *vt* is shifted left 16 bit positions. If the *vs* field has a value of 0x0, then no shifting of *vt* occurs before the addition. The resulting sum, clamped to the range r, -32768 <= r <= 32767, is stored into the vector register specified by *vd*. *Vs* is restricted to values of 0 or 1.

VRNDN, Round Accumulator if Negative



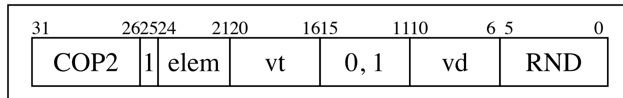
VRNDN vd, vs, vt[element]

The halfword element(s) of vector register *vt* specified by *element* is added to the Accumulator if the Accumulator currently contains a strictly negative number. If the Accumulator contains zero, or a positive number, it is not changed. If the *vs* field has a value of 0x1, then before adding the contents of *vt* to the Accumulator, *vt* is shifted left 16 bit positions. If the *vs* field has a value of 0x0, then no shifting of *vt* occurs before the addition. The resulting sum, clamped to the range r, -32768 <= r <= 32767, is stored into the vector register specified by *vd*. *Vs* is restricted to values of 0 or 1.

---

1. The use of the element field to select which third of the Accumulator is used as the source for the sum reduction has been conditionally accepted as part of the instruction definition. If this feature causes this instruction to exceed timing constraints, it will be removed from the instruction definition.

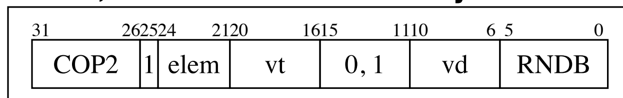
VRND, Round Accumulator



VRND vd, vs, vt[element]

The halfword element(s) of vector register *vt*, as specified by *element*, is added to the Accumulator. If the *vs* field has a value of 0x1, then before adding the contents of *vt* to the Accumulator, *vt* is shifted left 16 bit positions. If the *vs* field has a value of 0x0, then no shifting of *vt* occurs before the addition. The resulting sum, clamped to the range r, -32768 <= r <= 32767, is stored into the vector register specified by *vd*. *Vs* is restricted to values of 0 or 1.

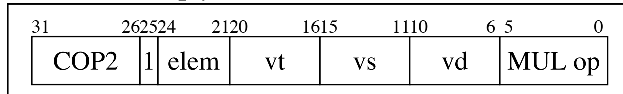
VRND, Round Accumulator Bytes



VRNDB vd, vs, vt, element

The contents of the high-byte and low-byte elements of the each Accumulator are rounded and scaled by *element*. The new Accumulator results are clamped to unsigned 8-bit values, concatenated together to form 16 bits and stored into vector register *vd*. The rounding factor is defined as 2 ** (element - 1). The scaling factor is defined as a shift-right by *element* bit positions, effecting a divide by $2^{element}$. The *vs* and *vt* fields are ignored.

4.6.5.5.7 Multiply Instructions



The multiply instructions operate on halfword operands. The results are stored into the Accumulator (with and without accumulation on the results currently in the Accumulator) during the EX2 stage. From the Accumulator, they are passed to the VRegister File, to be stored into the register specified by *vd,* during the WBack stage. Note that multiplication of two 16-bit operands generates a 32-bit result. All 32-bits are stored into the Accumulator for maximum precision but only 16 bits of the Accumulator get passed on to the VRegister File for write back. The determination of which 16 bits of the Accumulator are stored is based on the instruction being executed.

VMULF, Vector Multiply, Signed Fraction

VMULF vd, vs, vt[element]

The halfword element(s) of vector register *vt* specified by *element* is multiplied by each halfword element of vector register *vs*. Both operands are treated as signed fractions. The resulting product, a fraction, is normalized by shifting right 1 bit position and rounded up by incrementing at bit position 15 before being stored into the Accumulator without accumulation. Bits 31:16 of the Accumulator, clamped to a 16-bit signed value, are passed to the VRegister File and stored in vector register *vd*.

VMACF, Vector Multiply with Accumulation, Signed Fraction

VMACF vd, vs, vt[element]

The halfword element(s) of vector register *vt* specified by *element* is multiplied by each halfword element of vector register *vs*. Both operands are treated as signed fractions. The resulting product, a fraction, is normalized by shifting right 1 bit position and rounded up by incrementing at bit position 15 before being stored into the Accumulator with accumulation. Bits 31:16 of the Accumulator, clamped to a 16-bit signed value, are passed to the VRegister File and stored in vector register *vd*.

VMULU, Vector Multiply, Unsigned Fraction

VMULU vd, vs, vt[element]

The halfword element(s) of vector register *vt* specified by *element* is multiplied by each halfword element of vector register *vs*. Both operands are treated as signed fractions. The resulting product, a fraction, is normalized by shifting right 1 bit position and rounded up by incrementing at bit position 15 before being stored into the Accumulator without accumulation. Bits 31:16 of the Accumulator, clamped to a 16-bit unsigned value, are passed to the VRegister File and stored in vector register *vd*.

VMACU, Vector Multiply with Accumulation, Unsigned Fraction

VMACU vd, vs, vt[element]

The halfword element(s) of vector register *vt* specified by *element* is multiplied by each halfword element of vector register *vs*. Both operands are treated as signed fractions. The resulting product, a fraction, is normalized by shifting right 1 bit position and before being stored into the Accumulator with accumulation. Bits 31:16 of the Accumulator, clamped to a 16-bit unsigned value, are passed to the VRegister File and stored in vector register *vd*.

VMULQ, Vector Multiply, Quantize

VMULQ vd, vs, vt[element]

The halfword element(s) of vector register *vt* specified by *element* is multiplied by each halfword element of vector register *vs*. One operand for this multiply is expected to be a signed fraction, the other a signed integers. The resulting product is shifted left 16 bits and stored into the Accumulator, without accumulation, with the LSB at ACC[16]. Rounding is performed on the results in the Accumulator as they are stored into the VRegister File, depending on the sign of the results in the Accumulator. If and only if the Accumulator results are negative, then round the results toward zero by adding +2031616 (+31, shifted left by 16 bits)[1]. Bits 36:21 of the rounded results, clamped to a 12-bit signed value, are passed to the VRegister File and stored in vector register *vd*.

VMACQ, Vector Accumulator Oddification, Quantize

VMACQ vd, vs, vt[element]

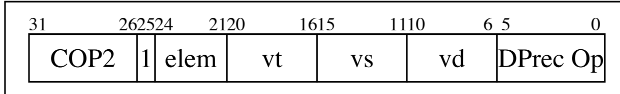
This instruction supports 12-bit MPEG quantization by performing oddification on the Accumulator. *Vt, element* and *vs* are ignored. If ACC[21] is 0, then oddify the Accumulator toward zero by subtracting +2097152 (+32, shifted left by 16 bits) if the Accumulator is positive, and adding +2031616 (+31, shifted left by 16 bits)[2]. If ACC[21] is 1, or if ACC[47:21] = 0, then add zero to the Accumulator. Bits 36:21 of the Accumulator, clamped to a 12-bit signed value, are passed to the VRegister File and stored in vector register *vd*.

---

1. The shifting of the results going into the Accumulator and the round value left by 16 bits, are not required from an architectural perspective. Rather, it is being done to accommodate the clamping hardware. All the other clamps take place on ACC[31:16}. Without the shift left by 16 bits, this instruction would require clamping on ACC[15:0]. With the shift, it clamps on the same ACC bits as all the other instructions.

2. The shifting of the results going into the Accumulator and the round value left by 16 bits, are not required from an architectural perspective. Rather, it is being done to accommodate the clamping hardware. All the other clamps take place on ACC[31:16}. Without the shift left by 16 bits, this instruction would require clamping on ACC[15:0]. With the shift, it clamps on the same ACC bits as all the other instructions.

4.6.5.5.8 Double-precision Instructions



Double-precision operations are supported with instructions that perform addition, subtraction, comparison and multiplication on double-precision operands. Double-precision operands occupy two vector registers. The hi-order 16 bits in one register and the low-order 16-bits in a second register.

4.6.5.5.8.1 Adds, Subtracts and Compares

Support for double-precision for addition, subtraction and comparison is provided through the use of the Vector Carry Out Register (VCOR). VADDC and VSUBC write the carry out from the operation to the VCOR. A subsequent VADD, VSUB or VSELXX instruction will use the VCOR as carry/borrow in and clear the VCOR for use in future operations. Double-precision comparisons are accomplished by performing a VSUBC on the high-order operands followed by a Select instruction performed on the low-order operands. No rounding or clamping is performed for these operations.

VADDC, Vector Add with Carry

VADDC vd, vs, vt[element]

The halfword element(s) of vector register *vt* specified by *element* is added to each halfword element of vector register *vs*. The result is stored in vector register *vd*. The carry out is stored into the VCOR.

VSUBC, Vector Subtract with Carry

VSUBC vd, vs, vt[element]

The halfword element(s) of vector register *vt* specified by *element* is subtracted from each element halfword of vector register *vs*. The result is stored in vector register *vd*. The borrow out is stored into the VCOR.

4.6.5.5.8.2 Multiplies

Several instructions are provided to support double-precision multiplies. They operate on the principle that a 2N-bit multiply can be implemented with four N-bit multiplies, where the partial product of each N-bit multiply can be shifted left or right by N-bits to achieve the proper alignment before being added to the other partial products. A double-precision multiply is a four instruction sequence each instruction taking a single cycle to complete for a total of 4 cycles. During the first instruction, the low-order partial product is computed, shifted right 16 bits and stored into the Accumulator. The next two instructions compute the two middle-order partial products and accumulate them with the low-order partial product. Finally, the hi-order partial product is computed, shifted left 16 bit positions and accumulated with the 3 previous partial products. None of the double-precision multiplies perform rounding on the results and clamping of results is to a 16-bit signed value.

VMUDL, Vector Multiply, Double-Precision Low Partial Product

VMUDL vd, vs, vt[element]

The halfword element(s) of vector register *vt* specified by *element* is multiplied by each halfword element of vector register *vs*. Both operands are treated as unsigned values. The resulting partial product is shifted right 16 bits, to align with ACC[15:0], and stored into the Accumulator without accumulation. The results, ACC[15:0], are then passed from the Accumulator to the VRegister File and stored in vector register *vd*.

VMADL, Vector Multiply with Accumulation, Double-Precision Low Partial Product

VMADL.madl vd, vs, vt[element]

The halfword element(s) of vector register *vt* specified by *element* is multiplied by each halfword element of vector register *vs*. Operands are treated as unsigned values. The resulting partial product is shifted right 16 bits, to align with ACC[15:0], and stored into the Accumulator with accumulation. The results, ACC[15:0], are then passed from the Accumulator to the VRegister File and stored in vector register *vd*.

VMUDM, Vector Multiply, Double-Precision Middle Partial Product

VMUDM vd, vs, vt[element]

The halfword element(s) of vector register *vt* specified by *element* is multiplied by each halfword element of vector register *vs*. *VS* is treated as a signed operand, while *vt* is treated as an unsigned operand. The resulting partial product, aligned with ACC[31:0], is stored into the Accumulator without accumulation. The results, ACC[31:16], are then passed from the Accumulator to the VRegister File and stored in vector register *vd*.

VMADM, Vector Multiply with Accumulation, Double-Precision Middle Partial Product

VMADM vd, vs, vt[element]

The halfword element(s) of vector register *vt* specified by *element* is multiplied by each halfword element of vector register *vs*. *VS* is treated as a signed operand, while *vt* is treated as an unsigned operand. The resulting partial product, aligned with ACC[31:0], is stored into the Accumulator without accumulation. The results, ACC[31:16], are then passed from the Accumulator to the VRegister File and stored in vector register *vd*.

VMUDN, Vector Multiply, Double-Precision Middle Partial Product

VMUDN vd, vs, vt[element]

The halfword element(s) of vector register *vt* specified by *element* is multiplied by each halfword element of vector register *vs*. *VT* is treated as a signed operand, while *vs* is treated as an unsigned operand. The resulting partial product, aligned with ACC[31:0], is stored into the Accumulator without accumulation. The results, ACC[15:0], are then passed from the Accumulator to the VRegister File and stored in vector register *vd*.

VMADN, Vector Multiply with Accumulation, Double-Precision Middle Partial Product

VMADN vd, vs, vt[element]

The halfword element(s) of vector register *vt* specified by *element* is multiplied by each halfword element of vector register *vs*. *VT* is treated as a signed operand, while *vs* is treated as an unsigned operand. The resulting partial product, aligned with ACC[31:0], is stored into the Accumulator without accumulation. The results, ACC[15:0], are then passed from the Accumulator to the VRegister File and stored in vector register *vd*.

VMUDH, Vector Multiply, Double-Precision High Partial Product

VMUL.mudh vd, vs, vt[element]

The halfword element(s) of vector register *vt* specified by *element* is multiplied by each halfword element of vector register *vs*. Both operands are treated as signed values. The resulting partial product is shifted left 16 bits, to align with ACC[47:16], and stored into the Accumulator without accumulation. The results, ACC[31:16], are then passed from the Accumulator to the VRegister File and stored in vector register *vd*.

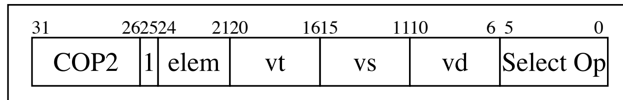
VMADH, Vector Multiply with Accumulation, Double-Precision High Partial Product

VMADH vd, vs, vt[element]

The halfword element(s) of vector register *vt* specified by *element* is multiplied by each halfword element of vector register *vs*. Both operands are treated as signed values. The resulting partial product is shifted left 16

bits, to align with ACC[47:16], and stored into the Accumulator with accumulation. The results, ACC[31:16], are then passed from the Accumulator to the VRegister File and stored in vector register *vd*.

4.6.5.6 Vector Select Instructions



The Vector Select Instructions perform element-by-element comparisons between *vs* and *vt* and set the Vector Compare Code Register (VCCR) based on the results. Each element for which the comparison is true, is stored into *vd*. Vector Select Instructions clear the VCOR. The resulting VCC bit is available for a VMERGE instruction in the following cycle without pipeline delays or stalls. The VCC bits may be written or read by the SU by using a Move Control To/From instruction. The VCC bits are 'sticky'; once set, they will remain set unless a subsequent instruction explicitly resets them.

The Select Instructions can also be used to support comparisons of double-precision data. To do so, the VCO bits are used as part of the comparison. For single-precision comparisons to work properly, the VCO bits for each slice should be preset to VCO[1:0] = 0. See Section 4.6.5.5.8 on page 160 for more information about how double-precision compares use VCO as an input and set VCC on output.

VSELLT, Vector Select, Less Than

VSELLT vd, vs, vt[element]

VLT vd, vs, vt[element]

Each halfword element of vector register *vs* is compared to the halfword element(s) of vector register *vt* specified by *element*. Clear VCC[1] for each slice. If *vs* is less than *vt*, set VCC[0] for the slice and store *vs* into *vd*; else clear VCC[0] for the slice and store *vt* into *vd*.

VSELLE, Vector Select, Less Than or Equal

VSELLE vd, vs, vt[element]

VLE vd, vs, vt[element]

This instruction has been deleted to make room for VSELCH and VSELCL.

VSELEQ, Vector Select, Equal To

VSELEQ vd, vs, vt[element]

VEQ vd, vs, vt[element]

Each halfword element of vector register *vs* is compared to the halfword element(s) of vector register *vt* specified by *element*. Clear VCC[1] for each slice. If *vs* is equal to *vt*, set VCC[0] for the slice and store *vs* into *vd*; else clear VCC[0] for the slice and store *vt* into *vd*.

VSELNE, Vector Select, Not Equal To

VSELNE vd, vs, vt[element]

VNE vd, vs, vt[element]

Each halfword element of vector register *vs* is compared to the halfword element(s) of vector register *vt* specified by *element*. Clear VCC[1] for each slice. If *vs* is not equal to *vt*, set VCC[0] for the slice and store *vs* into *vd*; else clear VCC[0] for the slice and store *vt* into *vd*.

VSELGT, Vector Select, Greater Than

VSELGT vd, vs, vt[element]

VGT vd, vs, vt[element]

This instruction has bee deleted to make room for VSELCH and VSELCL

VSELGE, Vector Select, Greater Than or Equal

VSELGE vd, vs, vt[element]

VGE vd, vs, vt[element]

Each halfword element of vector register *vs* is compared to the halfword element(s) of vector register *vt* specified by *element*. Clear VCC[1] for each slice. If *vs* is greater than or equal to *vt*, set VCC[0] for the slice and store *vs* into *vd*; else clear VCC[0] for the slice and store *vt* into *vd*.

VSELCL, Vector Clip Low

VSELCL vd, vs, vt[element]

VCL vd, vs, vt[element]

Each halfword element of vector register *vs* is compared to the halfword element(s) of vector register *vt* specified by *element*. If *vs* is greater than or equal to *vt*, set VCC[0] for the slice and store *vs* into *vd*. If *vs* is less than *vt* and greater than -*vt* (2's complement), set VCC[1] for the slice and store *vs* into *vd*. If *vs* is less than, or equal, to -*vt*, clear VCC[0] for the slice and store -*vt* into *vd*. This instruction clears the VCE bit.

VSELCR, Vector Crimp

VSELCR vd, vs, vt[element]

VCR vd, vs, vt[element]

Each halfword element of vector register *vs* is compared to the halfword element(s) of vector register *vt* specified by *element*. If *vs* is greater than or equal to *vt*, set VCC[0] for the slice and store *vs* into *vd*. If *vs* is less than *vt* and greater than -*vt* (1's complement), set VCC[1] for the slice and store *vs* into *vd*. If *vs* is less than, or equal, to -*vt*, clear VCC[0] for the slice and store -*vt* into *vd*. This instruction clears the VCE bit.

VMERGE, Vector Merge

VMERGE vd, vs, vt[element]

VMRG vd, vs, vt[element]

For each halfword element of vector register *vs* and the halfword element(s) of vector register *vt* specified by *element*, the corresponding VCC[0] for the slice selects either *vs* or *vt*. If the VCC bit is set, store *vs* into *vd*; else store *vt* into *vd*.

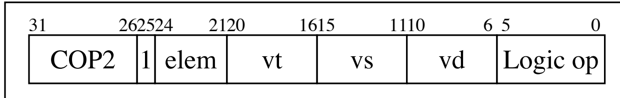
VSELCH, Vector Clip High

VSELCH vd, vs, vt[element]

VCH vd, vs, vt[element]

Each halfword element of vector register *vs* is compared to the halfword element(s) of vector register *vt* specified by *element*. If *vs* is greater than or equal to *vt*, set VCC[0] for the slice and store *vs* into *vd*. If *vs* is less than *vt* and greater than -*vt* (2's complement), set VCC[1] for the slice and store *vs* into *vd*. If the signs of the vs and vt are different and the result of adding the operands is -1 (all ones result), set the VCE bit. If *vs* is less than -*vt*, clear the VCE bit, VCC[0] for the slice and store -*vt* into *vd*.

4.6.5.7 Logical instructions



Logical instructions perform logical bit-wise operations on data stored in the VRegister File.

VAND, Vector And

VAND vd, vs, vt[element]

Each halfword element of vector register *vs* is bit-wise ANDed with the halfword element(s) of vector register *vt*, as specified by *element*. The result is stored into the Accumulator and from there, it is stored into vector register *vd*.

VNAND, Vector Nand

VNAND vd, vs, vt[element]

Each halfword element of vector register *vs* is bit-wise NANDed with the halfword element(s) of vector register *vt*, as specified by *element*. The result is stored into the Accumulator and from there, it is stored into vector register *vd*.

VOR, Vector Or

VOR vd, vs, vt[element]

Each halfword element of vector register *vs* is bit-wise ORed with the halfword element(s) of vector register *vt*, as specified by *element*. The result is stored into the Accumulator and from there, it is stored into vector register *vd*.

VNOR, Vector Nor

VNOR vd, vs, vt[element]

Each halfword element of vector register *vs* is bit-wise NORed with the halfword element(s) of vector register *vt*, as specified by *element*. The result is stored into the Accumulator and from there, it is stored into vector register *vd*.

VXOR, Vector Xor

VXOR vd, vs, vt[element]

Each halfword element of vector register *vs* is bit-wise XORed with the halfword element(s) of vector register *vt*, as specified by *element*. The result is stored into the Accumulator and from there, it is stored into vector register *vd*.

VXNOR, Vector Xnor

VXNOR vd, vs, vt[element]

Each halfword element of vector register *vs* is bit-wise XNORed with the halfword element(s) of vector register *vt*, as specified by *element*. The result is stored into the Accumulator and from there, it is stored into vector register *vd*.

4.6.6 VU Instruction Pipeline

4.6.6.1 Instruction Execution

The Vector Unit uses a 5-stage instruction pipeline to process its instructions. The Figure below shows how instructions flow through the pipeline.

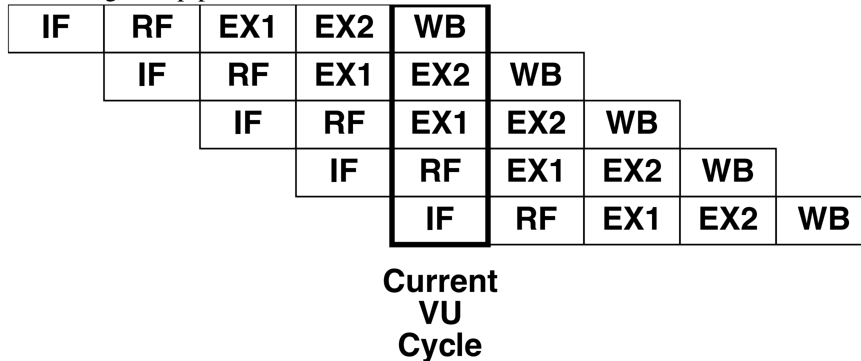


Figure 4-2 VU Instruction Pipeline

During the IF stage, the instruction is fetched from the local on-chip Instruction memory (IRAM). The hardware for this stage is shared with the SU. Only the remaining four stages have dedicated hardware for the VU. During the ID/RF stage, instructions are decoded and register operands are fetched from the VRegister File. Most instructions require two cycles to execute, EX1 stage and EX2 stage. General ALU two-operand operations such adds, subtracts, compares and bit-wise logical ops occur during the EX1 stage. The generation of multiply partial products also takes place during the EX1 stage. The EX2 stage could also be called the ACC(umulator) stage. During this stage, most instructions load the Accumulator with the results from a three operand addition, where one of the operands is often the Accumulator. When the Accumulator is not used as an operand into the addition, a constant of zero is used instead. The other operands in to the three-input adder are; results from the EX1 stage (such as the partial products of a multiply) and constants for rounding. During the WB stage, the results of the instruction are written back to the VRegister File.

4.6.6.2 Instruction Data Flow and Pipeline Stage Sequences

The figures on the next several pages show the data flow of each type of instruction through the pipeline.

4.6.6.3 Instruction Execution Times

Figure4-2 shows VU instructions flowing through the pipeline at the rate of 1 instruction/cycle. This accurate *assuming that there are no data hazards*. Unlike the SU, the VU has no forwarding or by-passing of results in the pipeline. Therefore, all VU computational instructions have a four-cycle latency when followed by a second computational instruction when the second instruction uses the destination of the first instruction as a source. Hardware will detect this condition and stall the second instruction at the ID/RF stage until the results of the first instruction are written back into the VRegister File. The only exception to this general rule is when the conflicting destination and source is the Accumulator, VCOR or VCCR. The results written to any of these registers by a computational instruction are available for use by a second computational instruction in the following cycle.

4.6.6.4 SU/VU Instruction Scheduling (Dependencies and Interlocks)

Below is a list of the various hazard types and a generic description of how they might occur in the MSP on VICE. These cases assume that once an instruction has been issued to the ID/RF stage of either pipeline, there are no subsequent stalls.

There are 3 types of hazards; Read After Write (RAW), also called a true dependency, Write After Write (WAW), also called an output dependency, Write after Read (WAR), also called an anti-dependency. WAW and WAR hazards have been included mainly for the sake of completeness. WAR hazards occur mainly as a result of out-of-order issue to multiple functional units. WAW hazards tend to occur when registers are written by different stages of different pipelines. Hazard detection hardware will detect and prevent these kinds of hazards from actually happening. That is to say hardware will prevent the out-of-order issue of the second instruction when it would lead to these kinds of hazards.

For load transpose and store transpose vector instructions, the lower 3 bits of the reg field should be masked out so that comparison is done on 8 register entries rather than on the base reg.

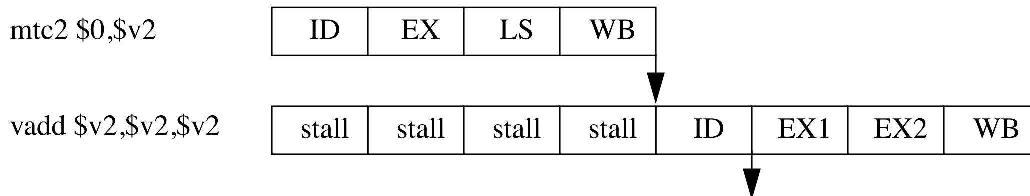
4.6.6.4.1 Stalled Instruction Dependencies

When an instruction gets stalled due to a dependency with a previous instruction, it can cause a WAR hazard to develope which otherwise would not occur. In these situations, any instruction with dependencies following the stalled instruction is stalled as well. The second stalled instruction remains stalled until the initially stalled instruction has been issued and any dependencies with that instruction have been resolved. In other words, instructions may be issued out-of-order, if and only if there are no dependencies between any of the instructions involved.

4.6.6.4.2 Read After Write (RAW) Hazards

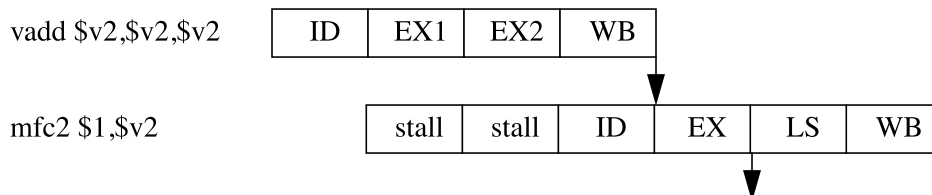
4.6.6.4.2.1 MTC2/LWC2/VU and SAW/VU instruction

MTC2, LWC2 and VU Instructions all write data into the VRegister File during the WB stage. Since there is no bypass mechanism in the VU, a RAW hazard occurs when the VU tries to read the data before the result data is written into the register file. One such scenario of a MTC2 followed by a VU instruction is shown below.



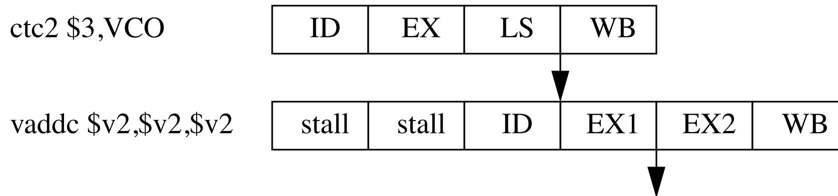
4.6.6.4.2.2 MTC2/LWC2/SAW/VU instruction and SWC2/MFC2

SWC2 and MFC2 operations read data out of the VRegister File during the EX stage and pass it to the SU during the LS stage. VU instructions such as SAW and VADD read the operand from the VRegister File during the ID/RF stage. Hence, the stall conditions for a MTC2/LWC2/VU followed by a SWC2/ MFC2 are less severe than for a MTC2/LWC2/VU followed by a VU Instruction (Refer to 4.6.6.4.1.1); only 3 stall cycles are necessary.



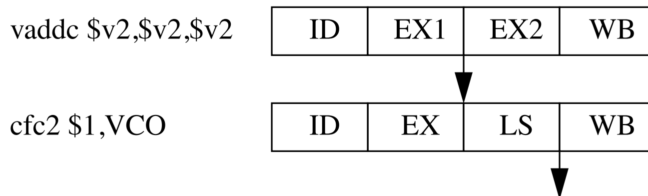
4.6.6.4.2.3 CTC2 and VU Add/Select

The CTC2 instruction writes the VCCR during the LS stage, instead of the WB stage as is the case with most other data transferring instructions in the MSP. VU Add/Subtract and Select instructions read and write the VCCR during EX1 stage. Hence, 2 stall cycles are required when a CTC2 instruction is followed by a VU Add/Subtract or Select Instruction.



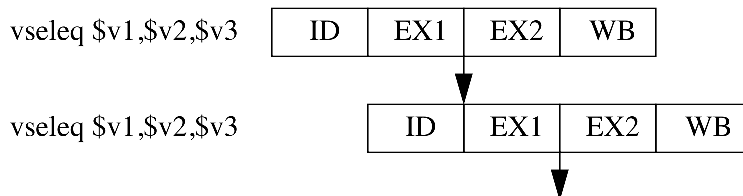
4.6.6.4.2.4 VU Add/Select and CFC2

A VU Add/Subtract or Select instruction followed by a Move Control From Coprocessor 2 does not cause a hazard because the VCOR is written by the VU at the end of EX1 stage and read by the SU at the beginning of the WB stage, which corresponds to the EX2 stage of the VU.



4.6.6.4.3 Accumulator and Vector Register Instructions

The Accumulator, VCOR, VCCR and VCER, are the exceptions to the general statement that the VU has no feedback paths. There is a feedback path for each one of these registers. Therefore, VU instructions which use either the Accumulator, VCOR or VCCR do not cause hazards when followed by a second instruction which also uses one of these registers as a source or a destination.

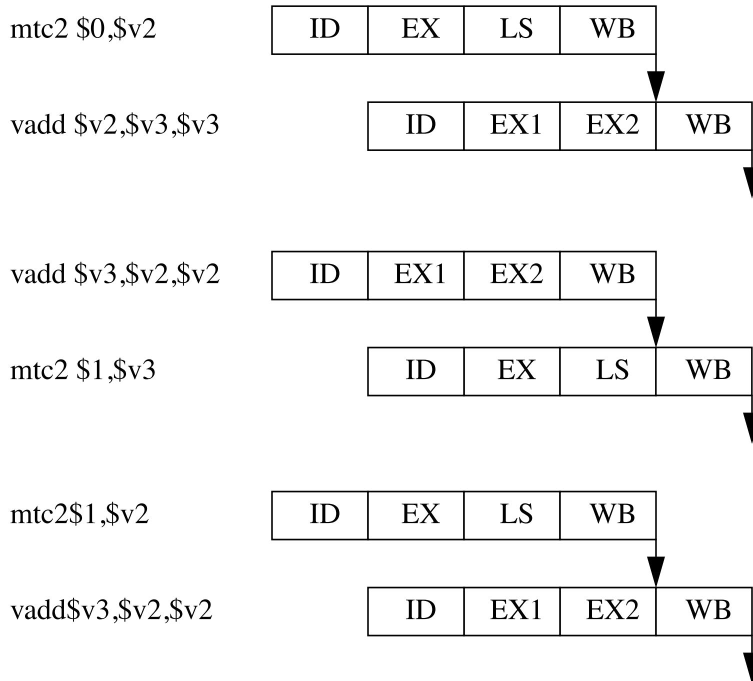


4.6.6.4.4 Write After Write (WAW) Hazards

4.6.6.4.4.1 LWC2/MTC2 and SAW/VU Instruction

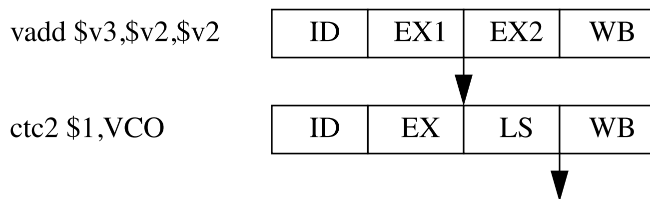
Moves of data to the VRegister File immediately preceded or followed by a VU computational instruction will cause a WAW hazard. Both the SU Move To /Load and the VU computational instruction write results back to the VRegister File at the end of the WB stage. To ensure that the correct results are written to the

VRegister File after one of these instruction pair is executed, the assembler must issue the instructions in the proper order and ensure that these instructions are never issued in the same instruction packet.



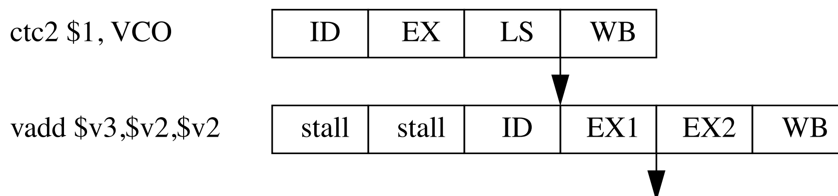
4.6.6.4.5 VU Add/Select and CTC2

A VU Add/Select instruction followed by a Move Control To Coprocessor 2 does not cause a hazard because the VCOR is written by the VU Add/Select instruction at the end of EX1 stage and re-written by the SU at the end of the LS stage, which corresponds to the EX2 stage of the VU.



4.6.6.4.5.1 CTC2 and VU Add/Select

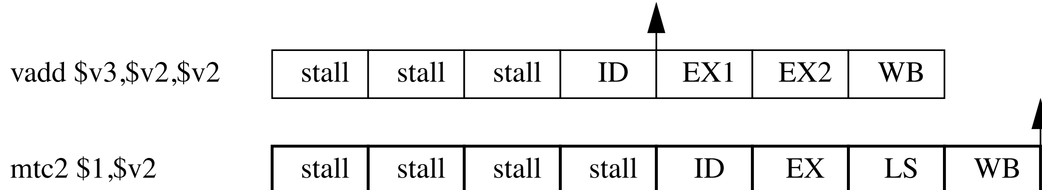
A Move Control To Coprocessor 2 instruction followed by a VU Add/Select instruction causes a hazard because the VCOR is written by the VU Add/Select instruction at the end of EX1 stage and re-written by the SU at the end of the LS stage.



4.6.6.4.6 Write After Read Hazard

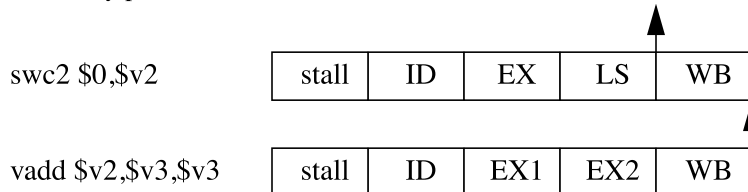
4.6.6.4.6.1 VU Instruction and MTC2

VU instructions read the VRegister File during the RF/ID stage. MTC2 writes the VRegister File during the WB stage. If the VU instruction is stalled do to a dependency with a previous instruction, say another VU instruction, then the MTC2 is stalled as well. The MTC2 instruction is stalled until the VU instruction has complete the RF/ID stage. In this case, the VU instruction can stall at most by 3 cycles (1 stall cycle will have already passed).



4.6.6.4.6.2 SWC2/MFC2 and VU Instruction

SWC2 and MFC2 read data out of the VRegister File during the LS stage to be written into a destination during the WB stage. VU instructions write results back during the WB stage. If the SWC2 instruction is stalled due to a dependency with a previous instruction, say another VU instruction, then a VU instruction following the SWC2 will have to stall a cycle to prevent overwriting the source register before the SWC2 instruction can read it. In this case, the SWC2 instruction will stall at most by 1 cycles (1 stall cycle will have already passed).



4.6.6.5 Illegal Instruction Sequence

The harware interlock logic will detect and automatically stall dependencies between instructions that are issued in different instruction packets. The interlock detection logic will not, however, do any dependency checking between SU and VU instructions which are issued in the same packet. Therefore, the programmer must ensure that hazards do not exist between SU and VU instructions which are issued at the same time. The next several subsections contain illustrative examples of instruction sequences which the interlock logic will not detect as hazards and which not generate the intended results.

4.6.6.5.1 Read After Write (RAW) Hazards

4.6.6.5.1.1 MTC2/LWC2 and VU instruction

```
nop
mtc2 $0, $v2
vadd $vd, $v2, $v2
```

```
nop
lwc2 $v2, 0($0)
vadd $vd, $v2, $v2
```

4.6.6.5.1.2 VU and SWC2/MFC2

```
nop
vadd $vd, $v2, $v2
mfc2 $0, $vd

nop
vadd $vd, $v2, $v2
swc2 $vd, 0($0)
```

4.6.6.5.1.3 CTC2 and VU Add/Select

```
nop
ctc2 $0, VCO
vadd $vd, $vs, $vt
```

4.6.6.5.1.4 VU Add/Select and CFC2

In the scenario below, the VEQ is stalled as it waits for $v2 to be written, however, the CFC2 would be issued before the VEQ is issued, resulting in erroneous sequences.

```
vadd $v2, $v0, $v0
veq $v1, $v2, $v2
cfc2 $1, VCC
```

However, the following code sequence is fine. This is because the VEQ instruction is not stalled and since the VCC is updated during the EX1 stage of the VEQ instruction and the VCC is read during the LS stage of the cfc2 instruction, there are no dependencies.

```
vadd $v2, $v0, $v0
nop
nop
nop
nop
veq $v1, $v2, $v2
cfc2 $1, VCC
```

4.6.6.5.2 Write After Write (WAW) Hazards

4.6.6.5.2.1 MTC2/LWC2 and VU instruction

The following sequences will result in unknown data written into the destination VU reg.

```
nop
mtc2 $0, $v2
vadd $v2, $v0, $v0

nop
lwc2 $v2, 0($0)
vadd $v2, $v0, $v0
```

4.6.6.5.2.2 VU and MTC2/LWC2 instruction

The following sequences will result in unknown data written into the destination VU reg.

```
vadd $v1, $v0, $v0
vadd $v2, $v0, $v0
mtc2 $0, $v2
```

```
vadd $v1, $v0, $v0
vadd $v2, $v0, $v0
lwc2 $v2, 0($0)
```

4.6.6.5.2.3 VU Add/Select and CTC2

In the scenario below, the VEQ is stalled as it waits for $v2 to be written, however, the CTC2 would be issued before the VEQ is issued.

```
vadd $v2, $v0, $v0
veq $v1, $v2, $v2
ctc2 $1, VCC
```

However, the following code sequence is fine. This is because the VEQ instruction is not stalled and since the VCC is updated during the EX1 stage of the VEQ instruction and the VCC is written during the LS stage of the CTC2 instruction, there are no dependencies.

```
vadd $v2, $v0, $v0
nop
nop
nop
nop
veq $v1, $v2, $v2
cfc2 $1, VCC
```

4.6.6.5.2.4 CTC2 and VU Add/Select

```
nop
ctc2 $0, VCO
vadd $vd, $vs, $vt
```

4.7 Bitstream Processor

The processing of high bit-rate bit streams in the JPEG (low compression ration) and MPEG-2 compression standards motivates hardware support for processing these bitstreams since current CPU technology cannot perform the real-time encode and decode functions for the high bit rates in low-compression JPEG and MPEG-2.

The bitstream processor is a programmable device which is tailored for processing bitstreams of compressed data. The bitstream processor has a 16-bit RISC-like load-store architecture. Hence, it has an instruction set which has familiar register to register operations (such as arithmetic operations), instruction stream control (jumps and branches) and memory to register transfer of data. In addition, the bitstream processor has instructions which are specific to manipulating arbitrarily aligned tokens in a bitstream of data. furthermore, the bitstream processor has instructions which can perform the table lookup operations necessary to decode variable length tokens in a bitstream. The tables provided to these instructions are programmable. They may be programmed to support decoding of JPEG, H261 MPEG-1, MPEG-2 bitstreams and encoding of H.261 and JPEG standard bitstreams, and with restrictions, can be programmed for proprietary algorithms.

Refer to Figure 41, “Bitstream Processor in the context of the VICE chip,” on page 173.

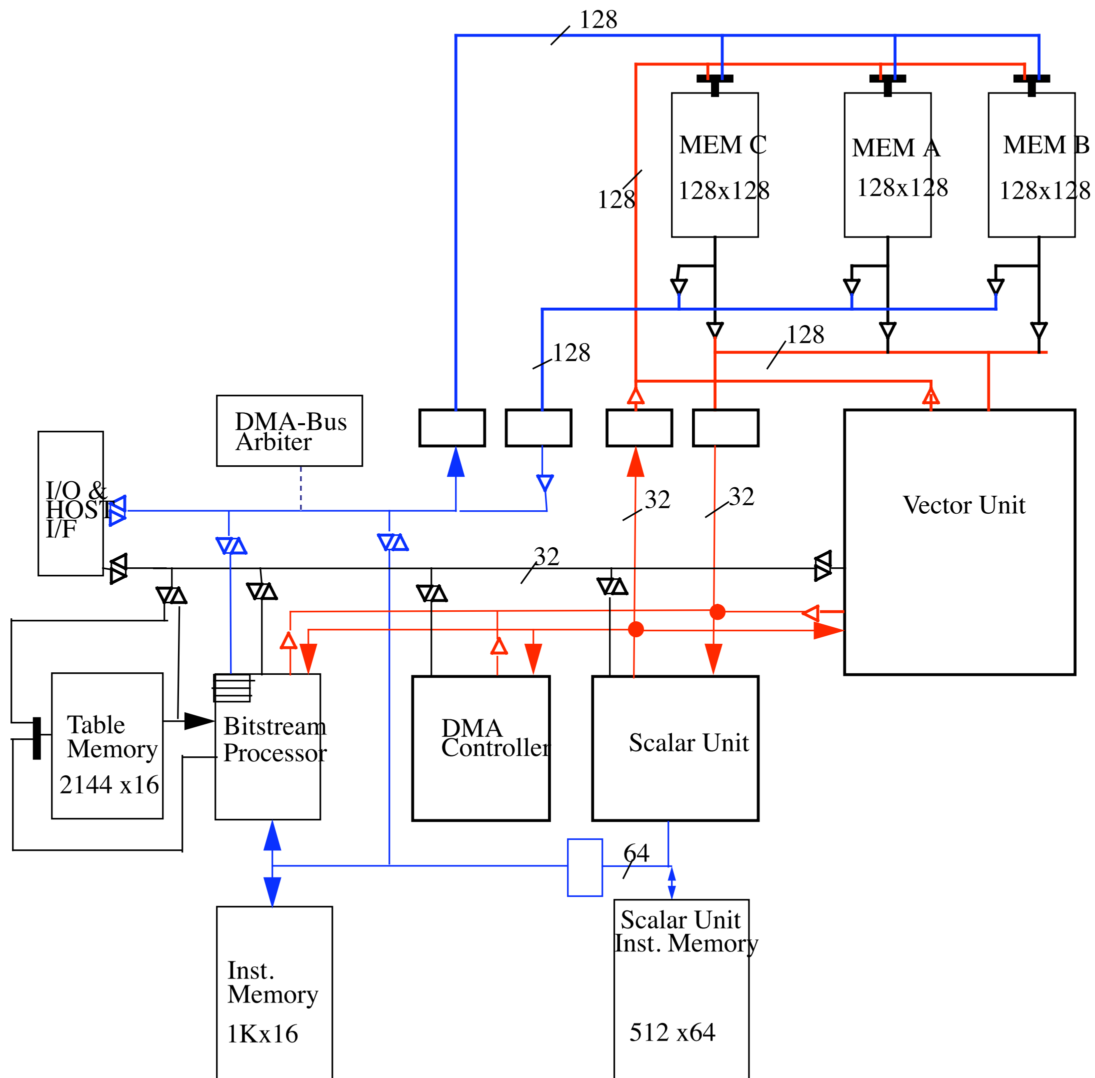


FIGURE 41. Bitstream Processor in the context of the VICE chip

In Figure 42, “Bitstream Processor and Memory,” on page 174, the Bitstream Processor is shown with its instruction memory and table memory. In addition, the Bistream Processor can access data memories (MEM A, B and C) on the VICE chip. There is a 64-byte FIFO memory which is used in decode operations. For decode operations, the bitstream FIFO is the repository from which the bitstream processor fetches bitstream data. In the decode operation, compressed data if put in the bitstream FIFO by the on-chip DMA controller.

In Figure 43, “BSP Internal Block Diagram,” on page 175, the internals of the Bistream Processor are shown.

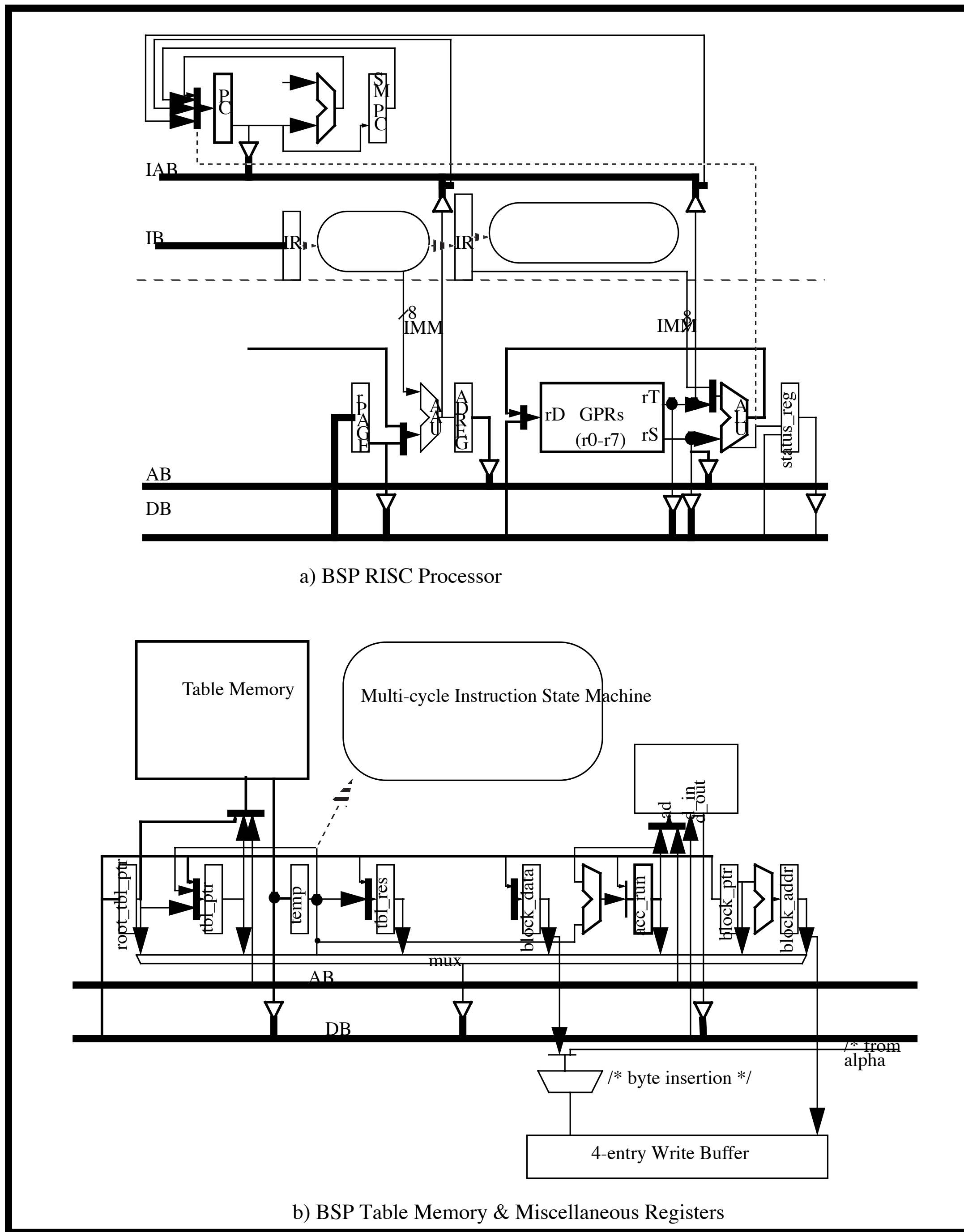


FIGURE 42. Bitstream Processor and Memory

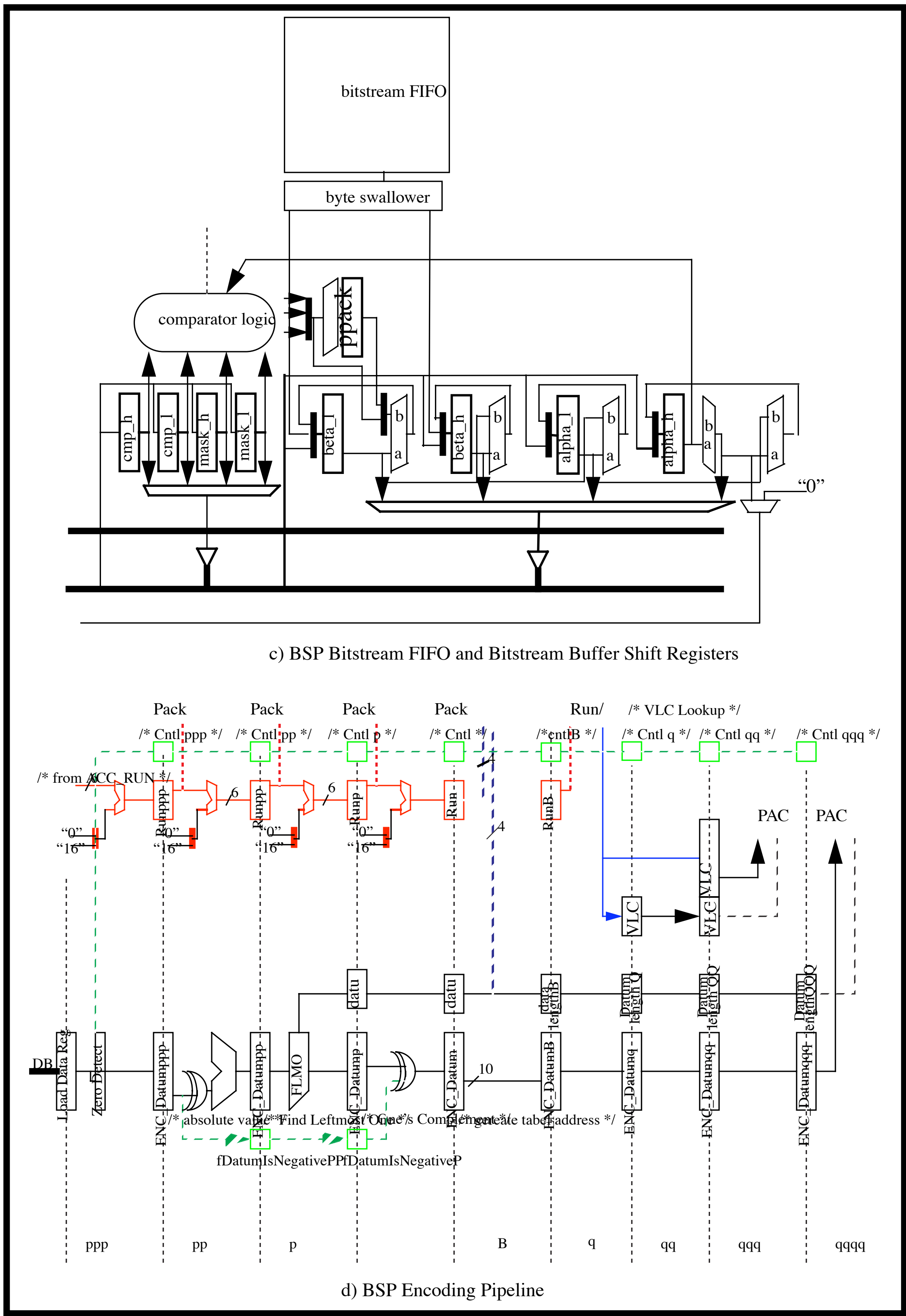


FIGURE 43. BSP Internal Block Diagram

4.7.1 Bitstream Processor Programming Model

The bitstream processor is a 16-bit load store architecture processor which has multi-cycle instructions used for handling bitstreams. Instruction processing is pipelined.

The Bitstream Processor has a three stage instruction execution pipeline. The pipeline stages are the Instruction Fetch (IF)stage, Instruction Decode stage (ID) and instruction execute stage (EX).

Instruction Fetch (IF)

In this pipeline stage, instruction addresses are issued from the program counter and the corresponding instruction is fetched and latched in the instruction register (IR).

Instruction Decode (ID)

In this pipeline stage the instruction in the IR is pre-decoded and the decoded signals are latched in the IR' register. General purpose registers are fetched in this stage.

Instruction Execute (EX)

In this pipeline stage the instruction is executed. Operands are operated upon and then stored back to the register file. For multi cycle instructions it is this pipeline stage which is extended in duration. Condition codes for arithmetic/compare/logical operations are set during this pipeline stage.

Figure 44, "BSP Instruction Pipeline," on page 176, shows the three level instruction pipeline in the bitstream processor.

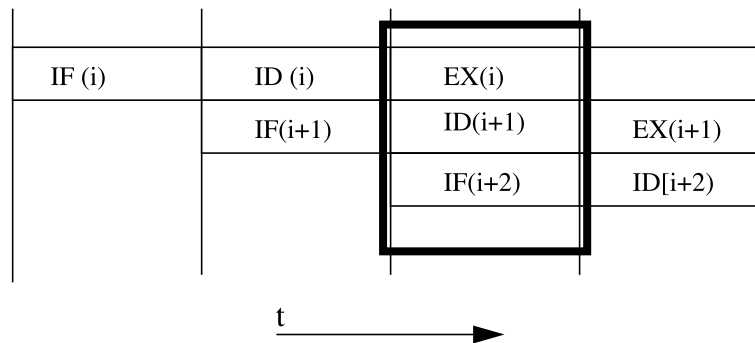


FIGURE 44. BSP Instruction Pipeline

Figure 45, "BSP Instruction Pipeline - Multi-cycle," on page 177, shows the instruction execution pipeline when a multi-cycle instruction is executed. When multi-cycle instructions are executing, no new instructions are issued in the execution pipeline. The execution pipeline is stalled while multi-cycle operations are executing.

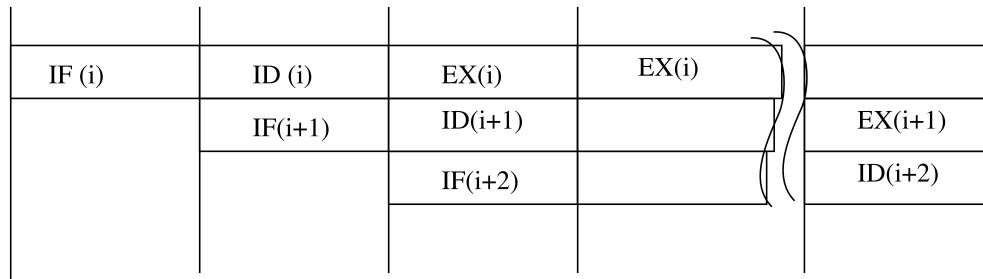


FIGURE 45. BSP Instruction Pipeline - Multi-cycle

Since the instructions have all their execution in the same pipe stage (namely the EX stage), there are no data dependent pipeline hazards. The only pipeline hazard in the bitstream processor is caused by a change to the sequential fetching of instructions. This occurs on jump instructions. Figure 46, "BSP Jump/Branch Instruction," on page 177, shows the execution of a jump instruction.

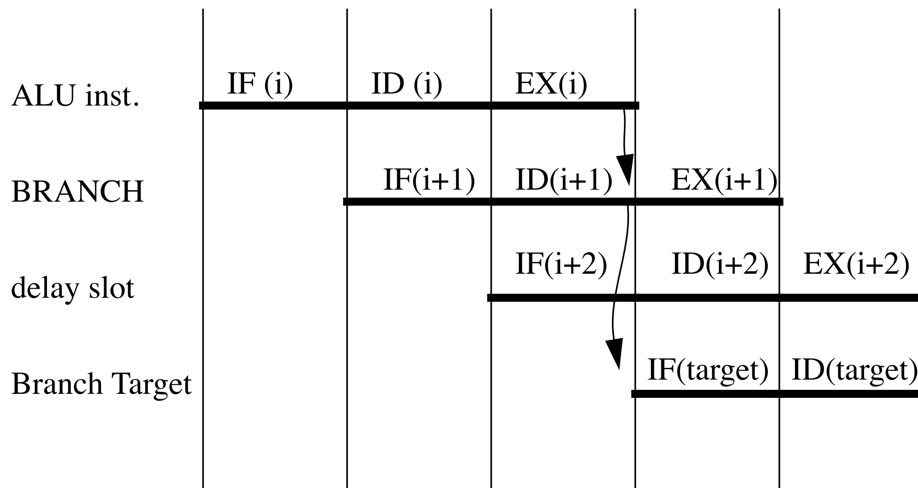


FIGURE 46. BSP Jump/Branch Instruction

Note that by the time the jump instruction can cause a change to the instruction stream by fetching at the jump's target address, the subsequent sequential instruction has entered the instruction execution pipeline. Hence, the instruction following the branch is always executed. This processor has one delay slot following the jump instruction.

4.7.1.1 Bitstream Processor Registers

The bitstream processor has eight general purpose registers (r0,r1,r2,r3,r4,r5,r6,r7). These registers provide source operands for arithmetic and logic instructions. These registers also may hold address information for the load and store instructions. These registers are 16-bits wide.

In addition to the general purpose registers are some function specific registers which are collectively called the ***alternate register set.*** These registers are 32 bit registers which can be accessed 16-bits at a time by the instruction set of the bitstream processor. These registers are enumerated below:

Compare Registers (**CMP_h and CMP_l**). These registers are used to hold a 32-bit compare value for searching the bitstream.

Mask Registers (Mask_h, Mask_l). These registers are used to mask out bits in the compare register so they don't take part in the compare operation while searching the bitstream.

Alpha Registers (Alpha_h, Alpha_l). These registers hold the next 32 bits in a bitstream.

Beta Register. This register holds up to 32 bits in the bitstream following the alpha register.

There are two more registers which have important functionality for performing variable length coding. These are:

Table_Root_Ptr. This register is a pointer to the root table in the decode table tree.

Table_Res. This register gets the data field from the leaf node entry in a table search.

Block_Ptr. This register is used, when decoding a bitstream, as a pointer to the memory location where the decoded block will be stored. When decoding an MPEG, JPEG or H.261 bitstream, the memory region where the 8x8 block is to be put is set to zeros so that the bitstream processor can put the decoded tokens in this zeroed region of memory without also writing the zero tokens.

RPage. This register is used as a base register for loads and stores.

ACC_Run. This register is used when decoding a bitstream, to count the number of zero tokens between non-zero tokens in the bitstream. This value is used when computing the address, in memory, where the non-zero token is to be stored. When encoding a bitstream, this register is similarly used to count the number of zero data values between non-zero data values.

Status and Control. This register holds ALU status flags such as Negative, Zero, Carry and Overflow. In addition it has three bits which may be set by writing to the status-and-control register and subsequently queried as a control flow condition. The format of the BSP status register is shown in Figure 47, "BSP Status and Control Register," on page 178.

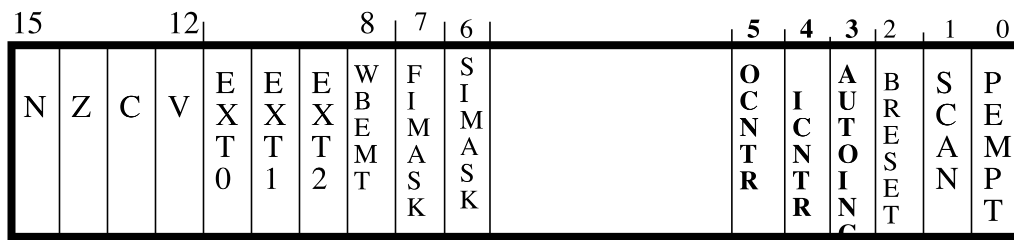


FIGURE 47. BSP Status and Control Register

The status and control register bits 15-9 are status bits which form the condition code upon which instruction

sequence altering instructions (i.e. BRANCH and JR) depend. Bits 15-12 represent status from arithmetic or logical instructions. N means the result of an ALU operation is negative (except for the ABS instruction where it represents the sign of the input operand of the ABS instruction). Z when 1 indicates that the result of an ALU operation is zero. C when 1 indicates that the ALU operation resulted in a carry (or borrow). V when set indicates that an overflow condition has occurred. These bits are zero upon reset. They may not be modified by the bitstream processor except by performing an alu operation.

Bits 11-9 are status bits used for modifying the instruction sequence. These bits may be set or cleared by the bitstream processor (i.e. using a COPYTO instruction) or the VICE scalar unit. These bits are cleared upon reset.

The SCAN bit controls the scan pattern used for MPEG, JPEG and H.261 compression standards. Each of these standards uses an 8x8 array of pixels which is fetched in Zig-Zag order (See relevant standards document). In addition, MPEG-2 has an alternate scan pattern. Instructions in the bitstream processor such as block_run_level_parse, zzxlate, block_run_size_parse, which use these unusual memory access patterns (Zig-Zag and Alternate) require special address translation hardware to convert a “linear” address to the scan pattern address. When the SCAN bit is 0, the Zig-Zag pattern is used. When the SCAN bit is 1, the alternate scan pattern is used. This bit is set to zero on reset.

The FIMASK bit is used to mask out the FIFO-Empty-Too-Long Interrupt. When 0, this interrupt is allowed (i.e. unmasked). When 1, this interrupt is ignored (masked) by the BSP. The reset value of this bit is 1.

The SIMASK bit is used to mask out the Code-Not-Found Interrupt. When 0, this interrupt is allowed (i.e. unmasked). When 1, this interrupt is ignored (masked) by the BSP. The reset value of this bit is 1.

The WBEMT bit is set to 1 when the BSP Write Buffer is empty (i.e. has no data in it). A zero indicates that there is data in the BSP’s write buffer. This bit is set to 1 on reset.

The ICNTR bit is used to reset the INPUT bit counter. A 1 to Zero transition of this bit causes the input bit counter to be reset to zero. The INPUT bit counter is incremented everytime 32-bits are consumed by the bitstream processor.

The OCNTR bit is used to reset the OUTPUT bit counter. A 1 to Zero transition of this bit causes the output bit counter to be reset to zero. The OUTPUT bit counter is incremented whenever 16-bits are produced by the bitstream processor (post byte-insertion for JPEG application.

The BRESET bit is used to reset the valid-bit counter in the bitstream buffer (alpha and beta registers). A 1 to 0 transition of this bit causes the valid bit counter to be reset indicating that there are no bits in the bitstream buffer. The reset value of this bit is 1.

The PEMPT bit is used to determine if the BSP’s encode pipeline has active instructions (PEMPT= 0) or has been completely emptied)PEMPT = 0.

There are two ways to cause the Bitstream processor to be reset. 1)Assertion of the VICE RESET signal will cause the BSP to go into its reset state. All state machines are put in their RESET state when this external signal is asserted. 2) A zero to 1 transition on the RESET BIT in the HALT RESET register will also cause the bitstream processor to be reset. All state machines are put into thier RESET state within 4 clock cycles of the 0 to 1 transition. Note that the reset value of the RESET BIT in the halt-and-reset register is zero. Hence, this bit is cleared to zero after the 0->1 transition is detected. The 0->1 transition is caused by writing a 1 into the RESET bit in the halt-reset register.

4.7.1.2 Bitstream Processor Instruction Summary

In this section, the bitstream processor's instruction set is summarized. Full details are given in Section???

Load/Store Instructions:

LOADH:

Load a 16-bit halfword from memory into a general purpose register.

LOADBl

Load a byte from memory into the least significant half (bits 7:0) a general purpose register.

LOADBh

Load a byte from memory into the most significant half (bits 15:8) a general purpose register.

Load Immediate

Load an unsigned immediate into a byte of a general purpose register

StoreH

Store the contents of a general purpose register to memory

StoreBl

Store the least significant byte of a general purpose register to memory

StoreBh

Store the most significant byte of a general purpose register to memory

Arithmetic/ Logical Instructions

ADD

Triadic addition. Sets condition codes.

ADDC

Triadic addition with carry from previous condition code setting instruction added. Used for double precision arithmetic.

SUB

Triadic Subtract. Sets condition codes

CMP

Compare. Sets condition codes

AND

Bitwise triadic logical AND. Sets Condition codes

OR

Bitwise triadic logical OR. Sets Condition Codes

LSHIFT

Shift a register left by up to eight bits. Fill with zeros

ARSHIFT

Shift a register right by one bit while preserving the sign of the value being shifted.

MULT

Multiply two general purpose registers together. The low half of the product is stored in a general purpose register, the high half of the product is stored in the **cmp_h** *alternate register*.

ZZXLATE

This instruction will map a a 6-bit number into a "Zig-Zag" number or an "Alternate Scan" number. This instruction is useful for MPEG, JPEG, and H.261 compression standards. The mapping performed by this instruction is defined by the SCAN bit in the Status-an-Control register.

XOR

Bitwise exclusive OR. Sets Condition codes

ABS

Absolute value (monadic). Sets Condition codes. The Negative condition code is slightly different from other instructions in that the N-bit is set based upon the sign of the input to the absolute value instruc-

tion. This allows on to simultaneously compute and absolute value and to know whether the input was positive or negative. (This is useful in a number of cases when decoding MPEG bitstreams).

Note about ABS: When taking the absolute value of 0x8000, 0x8000 is returned. This is the anomalous case due to the asymmetry in the 2's compliment number system. The intended use of this instruction is for decoding Motion Vectors in H261 and MPEG bitstreams. These Motion vectors will neve be larger than 12 bits, hence this anomalous case is not a problem.

NEG

Negate (2's complement) a register (monadic). Sets Condition Codes.

Bitstream Instructions:

getBits(q) rD, N

Get N (N =1..16) bits from the bitstream registers (alpha and beta) and put these N bits (right justified) into a general purpose register. Shift the N bits out of the bitstream registers (i.e. advance the bitstream search by N bits). If q= 1 then perform JPEG byte swallowing. (See section????)

probeBits rD, N

Copy N (N = 1..16) bits from the bitstream registers (alpha and beta) and put these N bits (right justified) into the a general purpose register. Do NOT Shift the N bits out of the bitstream registers. Unlike the getBits(N) instruction, this instruction does not advance the bitstream.

ShiftStream(N.q))

Shift N bits off of the bitstream. This instruction can be used to discard (and ignore) bits in the bitstream. User information in the bitstream might be an example of data which may be discarded. Macroblock stuffing bits may also be discarded. If q = 1, then JPEG byte swallowing is performed. (See Section????)

leaf_run_level_parse(q)

This multi-cycle instruction is used to perform the decoding of variable length coded data in the bitstream. In particular, this instruction decodes run-level values pertaining to the H.261 and MPEG-1 and MPEG-2 compression standards. This instruction performs a table searching algorithm to arrive at a run and level value. With constraints, this instruction, by changing the programmable tables, can perform decoding of proprietary codes for the run-level decoding of the pixels in the bitstream. This instruction stops execution when it finds a run-level value (leaf node) in the bitstream. This is determined by a bit in the search tables. If q = 1, the decoded token (level) in the bitstream is pre-modified before the VICE MSP performs inverse quantization. This pre-modification is as follows:

level = 2* level + Sign(level).

This is useful for the H261 inverse quantization operation for inter macroblocks. It is also useful for MPEG inverse quantization operation for non-intra macroblocks. Note that this pre-inverse quantization step is optional based upon the q-bit in the instruction. If q = 0, then the pre-inverse quantization step is not performed and the level is passed to memory unmodified.

block_run_level_parse(q)

This multi-cycle instruction is used to perform the decoding of variable length coded data in the bitstream. In particular, this instruction decodes run-level values pertaining to the H.261 and MPEG-1 and

MPEG-2 compression standards. This instruction performs a table searching algorithm to arrive at a run and level value. With constraints, this instruction, by changing the programmable tables, can perform decoding of proprietary codes for the run-level decoding of the pixels in the bitstream. This instruction stops execution when it has completed all run-level searches for an 8x8 block of pixels. Hence this instruction can generate all run-level values for an 8x8 block of pixels. Each level in the block is written to memory on the VICE chip. If the q-bit in the instruction is set then the level value is pre modified (see description of **leaf-run_level_parse** instruction) before being sent to memory for inverse quantization. If q=0 then the decoded token is passed to memory unmodified. This instruction behaves the same way as the leaf_run_level_parse except that it searches the decode tables until all tokens in an 8x8 block have been decoded and sent to memory.

generic_leaf_parse

This multi-cycle instruction is used to perform the decoding of variable length coded data in the bitstream. Unlike the run_level_parse instruction, no semantic meaning is placed upon the data found at the leaf node of the table search. Hence, this instruction can be used to decode VLC's other than the run-level VLC's in the bitstream. and example is the motion vector VLC's in the MPEG and H.261 compression standards. Note that, with restrictions, by modifying the tables used to perform the VLC decoding, proprietary codes can be supported. Furthermore, this instruction may also be used to walk the run-level tables with additional code used to make semantic meaning out of the table contents.

block_run_size_parse(q)

This instruction supports the JPEG compression standards' technique of encoding the 8x8 block of pixel data. The JPEG bitstream contains a VLC which specifies a run of zeros with a length field which specifies how many bits follow the VLC to represent the data level of interest. Note that with restrictions, the table lookup can be modified to support other VLC encoding of this information. The JPEG standard (unlike the MPEG and H.261 standards) allows modification of these VLC encodings. This instruction enables the JPEG Byte swallowing mode of operation. The Byte-swallowing Mode removes 0x00 bytes from the input bitstream if they follow byte aligned 0xFF tokens. Non-Zero bytes followiing the byte aligned 0xff sre **not** swallowed. See the JPEG standards document for more details.

code_search(q)

This instruction searches the bitstream one bit at a time until it finds a match with the contents of the **CMP** (compare) register). The compare register is 32-bits in length, however, the mask register can be used to eliminate some bits in the **cmp** register from participating in the compare. This instruction is useful for searching for "start" codes up to 32-bits in length. This is useful for passing over "stuffing" bits as well as for error recovery. This operation also allows for skipping the decode of a picture. This can be achieved by searching for the next picture start code. Note that only B-pictures in the MPEG standard may be skipped over.

load_code_pack(q,p) offset

This instruction is used for Huffman encoding of an 8x8 block of data. The instruction fetches data from the effective address (rPage+offset) and places it in the encoding pipeline. If Q = 1, then **JPEG** style byte insertion will be performed. If p = 1 then the instruction performed DC coefficient coding. If p=0, the instruction performs AC coefficient coding. Data passed through the encoding pipe is modified to JPEG specifications. Run of Zeros and datum size are computed and a Huffman Table lookup is performed. The VLC code is then packed (appended) into the bitstream buffer (beta_h). The modified datum is then appended to the bitstream (JPEG-style).

load_code_packH261

This instruction is used for Huffman encoding of an 8x8 block of data. This instruction fetches data from the effective address (rPage + offset) and places it in the encoding pipeline. In this encoding pipeline, H261 Huffman coding is performed.

generic_lookup_pack rT

This instruction is used to perform Huffman encoding of information other than the 8x8 blocks of DCT coefficients. This instruction is used for the encoding of H261 macroblock headers. This instruction gives great flexibility for Huffman coding of many types of data.

pack_bitstream(q) L, rT

This instruction is used for encoding bitstreams. In this instruction, L bits (L<=16) left justified in general purpose register rT are appended to the end of the outgoing bitstream in the alpha and beta registers. If q=1, then JPEG-style byte insertion (See section????) will be performed.

byte_align

This instruction forces byte alignment of the bitstream, filling with bits from the CMP register.

Branch and Compare Instructions:

CMPI

Compare Immediate. This instruction is used for comparing the contents of a general purpose register with an 8-bit immediate value. The comparison is done without altering the general purpose register. This can be very useful for acting on contents of the bitstream. The compare is an 8-bit unsigned compare, with the gpr high byte zeroed.

ANDI:

AND Immediate. This instruction is used for examining bits in a general purpose register. This instruction performs a bitwise logical AND operation with a general purpose register and an 8-bit immediate value. (upper 8 bits of gpr are ignored)

ADDI:

ADD and 8-bit immediate value (zero extended) to another GPR.

BRANCH:

Conditional jump. This instruction is used to alter the instruction execution sequence based upon conditions in the status and control register.

JR.

Jump on condition. This instruction is used to alter the instruction execution sequence. The target address of this jump can be specified in a general purpose register.

RESUME

This instruction is used to resume execution of a multi-cycle instruction which has been "interrupted" by some exceptional condition. (An example os such an exceptional condition is while decoding an MPEG escape code).

BREAK

4.7.1.3 INTERRUPTS

For the purposes of discussion, interrupts in the BSP are a change in execution sequence caused by events not under direct program control. There are four types of interrupts which may occur in the course of BSP execution.

1) Escape Code Detection. This interrupt is taken when indicated by the contents of the VLC decode table (See section 4.7.2). When this interrupt is taken, instruction execution commences at location 0x0010.

2) Bitstream Error Detection. This interrupt is taken in response to an error detection in a decoded bitstream. This interrupt is taken when indicated by the contents of the VLC decode table (See Section 4.7.2). When this interrupt is taken, instruction execution commences at location 0x00c.

3) Bitstream FIFO Empty Too Long. When fetching from the bitstream FIFO, it may be empty. If the bitstream FIFO is empty for 2048 (4096) cycles as configured in the Status & Control Register then this interrupt it taken. When this interrupt is taken, instruction execution commences at location?????.

4) Code Search: CODE NOT FOUND. When fetching a bitstream using the **code_search** instruction it may be the case that the code isn't to be found. When the code search instruction starts executing, a bit counter starts counting the number of bits removed from the bitstream. When 1024K bits (128K bytes) have been removed from the bitstream without the code of interest being found, this interrupt is taken. When this interrupt is taken, instruction execution commences at location????

4.7.2 VLC Decode Table Structure

The tables used for decoding the variable length codes (VLC's) have three implied formats. The first format is the "generic" format. The second format is the "run-level" format, and the third format is the "run-length" format.

4.7.2.1 Generic Format

In the generic format, the decode table entries are shown in Figure 48, "Generic Table Entry Format," on page 186.

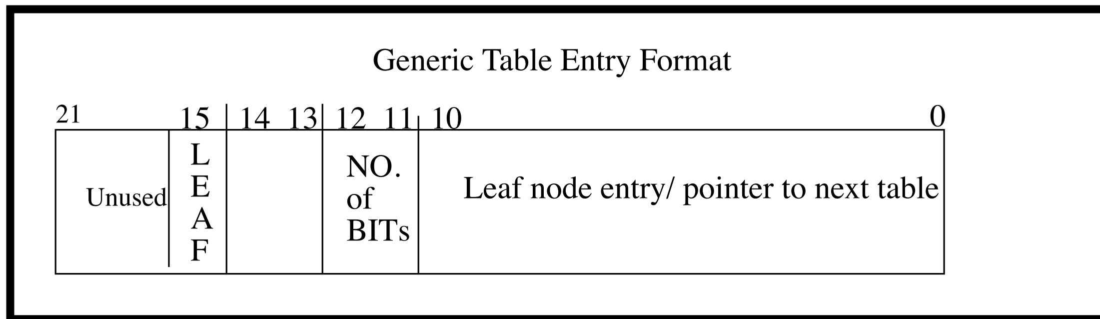


FIGURE 48. Generic Table Entry Format

Each generic lookup table consists of sixteen entries. Four bits at a time are taken from the bitstream and used to perform a table lookup. If the LEAF bit is set then the search stops and the value of the leaf node entry (bits 10:0) is returned. The NO-of-Bits field specifies how many of the four bits used for the table lookup must be discarded. If the LEAF bit is not set then the VLC search must continue. The NO-of-Bits field should specify 4 bits to be discarded and the leaf node entry field will be interpreted as an address of the next lookup table.

When searching for the table entry for a long VLC, many table lookups may be performed.

The “generic” table format is so-called because there is no semantic meaning associated with the leaf node entry (bits 10:0).

4.7.2.2 Run-Level Format

In the Run-Level Format, the decode table entries are shown in Figure 49, “Run-Level Table Descriptor Format,” on page 187.

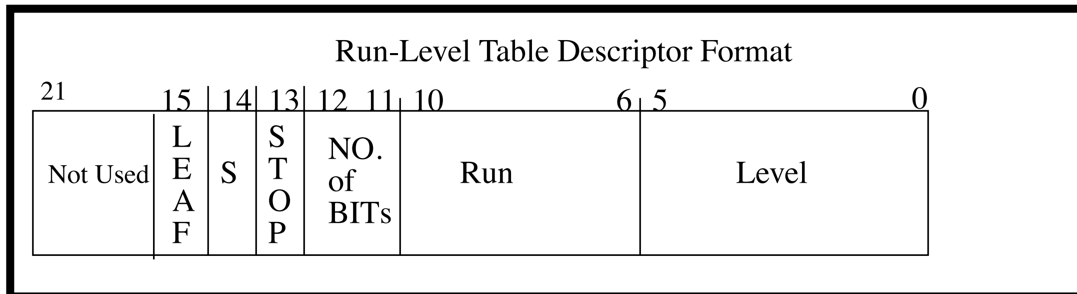


FIGURE 49. Run-Level Table Descriptor Format

The Run-Level table format is used when decoding variable length codes describing run-level encoding of 8x8 blocks of data. In the Run-Level case, the Run represents a a number of zero data values followed by a single data value. This data value is determined by the Level field as well as the S field. Having determined the magnitude of the data value from the Level field, the sign of the data value will be derived from the bit in the bitstream following the VLC. This bit will be queried if the S-field (bit 14) is set.

The Run-Level Format search strategy is the same as the generic format except that rather than stopping the search when the LEAF bit is set, the Run-Level search continues until the STOP bit is set. This STOP bit generally indicates that the end of the 8x8 block has been found. Hence the Run-Level format can be used to acquire a number of leaf nodes in the search until it hits the STOP indication. In this way, multiple searches are performed until the end-of-block (STOP) is indicated.

The Run-Level Format pertains to the MPEG-1, MPEG-2 and H.261 compression standards. Furthermore, within constraints, the tables can be generated to decode proprietary VLC’s.

4.7.2.3 Run-Level Escape Codes:

Note that the Run field is only 5 bits in length. This restricts the longest RUN to be 31. Similarly the Level field is only 6 bits in length. This means that a level magnitude of 63 is the greatest which can be represented. In the MPEG and H.261 compression standards, these field widths are adequate to perform VLC decoding. When an escape code is found in the bitstream this is represented by TBD in the lookup table. When an escape code is detected, the escape code symbol in the bitstream is discarded and the VLC decoding state machine interrupts the bitstream processor. The bitstream processor handles the escape code in an interrupt service routine. Each of the standards, MPEG-1, MPEG-2, JPEG and H.261 have different escape code processing requirements.

4.7.2.4 Run-Level Table Programming Restrictions

When creating the Run-Level Tables, one should be sure that the table lookup graph is a-cyclic. Cycles in the graph can cause the bitstream processor to get into an infinite loop while walking the run-level decode graph.

4.7.2.5 Run-Length Format

The Run-Length format for the VLC lookup tables is shown in Figure 50, “Run-Length Table Descriptor Format,” on page 188.

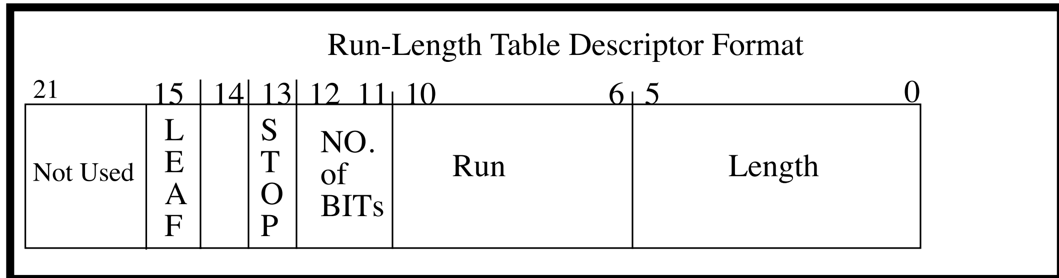


FIGURE 50. Run-Length Table Descriptor Format

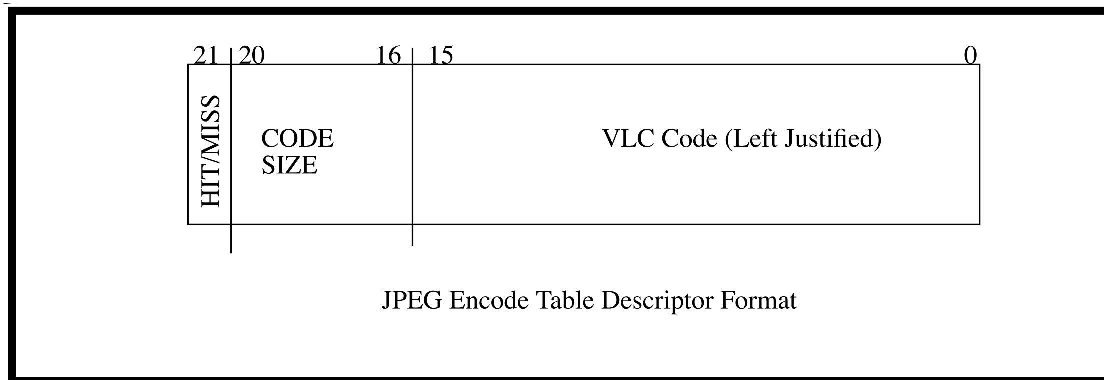
In the Run-Length format the LENGTH field specifies the length, in bits, that must be extracted from the bit-stream to obtain the data value which follows the run of zeros implied in the RUN field.

The Run-Length Format is germane to the JPEG compression standard.

4.7.3 VLC Encode Table Structure

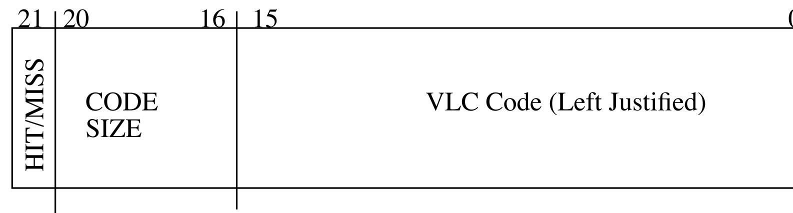
4.7.3.1 JPEG Encode Table Structure

JPEG encode table structure consists of 32-bit table descriptors which are 2-byte aligned. That is, two 16-bit memory locations hold a table descriptor. The following figure shows the format of the JPEG encoding VLC descriptors:



In this descriptor, the HIT/MISS bit indicates a valid table entry. The Code size field indicates how many bits are to be found in the VLC-Code field. In JPEG, VLC codes have a maximum size of 16 bits.

4.7.3.2 H.261 Encode Table Structure



H261 Encode Table Descriptor Format

4.7.4 Programming Restrictions.

In this section we summarize the restrictions placed upon the programmer of the Bitstream Processor:

4.7.4.1 Branch

As previously mentioned, instructions which alter the sequence of instruction fetches have a delayed impact. The instruction immediately following the BRANCH or JR instruction is always executed. Therefore, neither a BRANCH instruction nor a JR instruction may be immediately followed by another BRANCH or JR instruction.

The decision to take a branch is based on the condition codes in the status and control register. These bits are set by most of the format C instructions, including the CMP instruction

The offset for the conditional branch is an 8 bit signed quantity. Therefore, forward branches of more than 127 instructions or backward branches of more than 128 instructions require the use of the JR instruction.

4.7.4.2 Multiply

The multiply instruction takes two cycles to execute. The results of the multiply are not available until two instructions later. Since the low half of the 32-bit product is put into a general purpose register, the instruction following the MULTIPLY instruction cannot store to a general purpose register. The high part of the 32-bit product is put in the CMP_h register.

The preferred method of getting a 32-bit product to the general purpose registers is shown below:

```
MULT rD, rS, rT

NOP

COPYFROM rX, CMP_h
```

The net result is that the multiply takes three cycles to get put into the GPR's

4.7.4.3 Add Immediate

The ADDI instruction is an unsigned compound assignment operator such that *ADDI GPRn, imm* is equivalent to *GPRn = GPRn + imm*. The immediate must be in the range 0-255.

4.7.4.4 Absolute Value

The absolute value instruction (ABS) is included in the instruction set architecture to facilitate decoding of motion vectors, and as such, it has two features programers should note.

The instruction will report the absolute value of -32768 (0x8000) as -32768. Motion vectors in H.261 and MPEG are less than 12 bits and should not be effected by this restriction.

The N bit of the status and control register is set based on the original value, not the computed value of the ABS instruction.

4.7.4.5 BRESET

The BRESET bit of the BSP status and control register is an "edge triggered" event. The bit must be first set and then cleared for the reset to occur. The following is sample code:

```
// get current state of status_cntl
COPYFROM $6, STATUS_CNTL
// load the breset bit
LDI_l $7, BRESET_L
LDI_h $7, BRESET_H
// OR in breset
OR $6, $6, $7
COPYTO STATUS_CNTL, $6
// get current state of status_cntl
COPYFROM $6, STATUS_CNTL
LDI_l $5, BRESET_MASK_L
LDI_h $5, BRESET_MASK_H
// clear breset bit
AND $6, $6, $5
// store it back to status_ctrl
COPYTO STATUS_CNTL, $6
```

The reset does not occur for an additional two cycles.

4.7.4.6 COPYTO Delay Hazard

There is a one cycle delay when copying bits to the status and control register before a branch can be taken based upon these bits..

4.7.4.7 Encode Pipe Latency

The encode pipe is an eight stage pipe. Thus, no changes to any of the bitstream encoding resources (bitstream buffer, root_tbl_ptr, acc_run, alpha and beta registers, block_data, and block_addr) should be made for eight cycles following a load_code_pack or a generic_VLC_lookup_pack.

4.7.4.8 "Interrupted" Instructions.

The following multi-cycle instructions may encounter exception conditions which will cause execution of the instruction to be halted temporarily while a service routine is executed. This may happen, for example, when a bitstream error is detected (indicated in the VLC decode table) or an MPEG (or H.261) ESCAPE Code is encountered. Such instructions must be followed by two NOP instructions!

The instructions which must be followed by **two** NOP instructions are:

- block_run_level_parse
- leaf_run_level_parse

4.7.5 Performance of Bitstream Processor

The performance of the bitstream processor can be characterized in a number of ways depending upon the task it is programmed to perform. The first performance measure is MIPS. Since the bitstream processor executes one instruction per cycle (except for the multi-cycle instructions), the bitstream processor can be thought of as performing approximately 66 MIPS peak rate.

A second performance measure is how many variable length codes can be decoded per second. The current bitstream processor architecture examines 4 bits at a time. It takes two cycles to perform a table lookup and discard of these four bits, hence, the maximum rate at which the VLC tables can be parsed is 2bits/cycle or approximately 133 million bits per second. Now this performance will be de-rated because of a number of factors. First, there is a dependency on the number of bits per symbol in the VLC's. Typically the shortest symbol is two bits. If all symbols were two bits, then it would take two cycles to decode these two bits, hence the decoding bit rate is 1 bit per cycle or approximately 66 million bits per second. Now this number must be further de-rated by the duty cycle of the bitstream processor. Assume that one is decoding the VLC's about 2/3 of the time then the average decoded bit rate is 2/3*66 = 44 million bits per second. This represents approximately a 5:1 compression for 27MHz., 4:2:2 sampled images(CCIR.601). Of course this is statistical in nature and support for 4:1 compression is likely.

MPEG-2 compression is likely in the 8Mbit per second (and lower) range. The current bet estimate is that the duty cycle for the VLC decoding is approximately. 1/3. Hence the bitstream processor has the capability of decoding approximately. 22 million bits per second. This is sufficient for MPEG-2 decoder. In the MPEG-2 decoder, however, larger demand is placed on the bitstream processor to compute motion vector information and addresses for the DMA sub unit. This is why the duty cycle of the VLC processing is smaller than for the JPEG application.

.Include further discussion here on MPEG-2 decode performance requirements????????

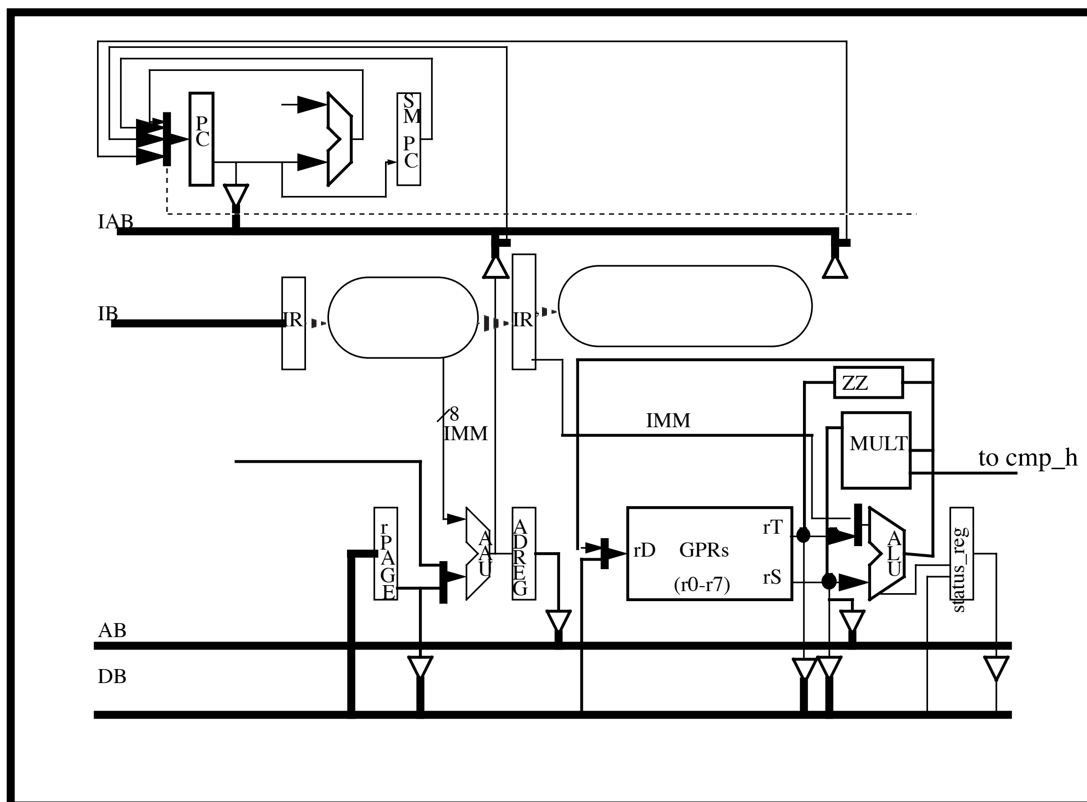
4.7.6 Bitstream Processor Hardware Architecture

4.7.6.1 RISC Processor

The BSP's "RISC" Processor is a 16-bit processor having a load-store architecture like today's RISC processors. There are 8 general purpose registers, one ALU, one Multiplier (pipelined), Shifter as well as some specific function units used in the parsing of bitstreams and the computation of motion-vector information in the bitstreams. These special purpose function units perform negation (2's complement negation), absolute value and Scan pattern translation (Zig-Zag & ALternate Scan). See the MPEG-2 standards document.

The RISC Processor pipeline is discussed in section 4.7.1.

Figure??? Shows the "RISC" processor section of the BSP.



The rPAGE register is notable because it is the base register used by load and store instructions when forming the effective address of the load and store instruction. An 8-bit immediate value is added to the rPage register when performing the effective address calculation, hence the need for the AAU (Address arithmetic Unit). This means that the programmer can load and store from a 128-halfword (16-bits) page of memory. In addition the long-load (LLOAD) and long-store (LSTORE) instructions can be used to access halfwords anywhere within the 16-bit memory space of the BSP, however, the effective address for these operations is absolute, i.e. not formed from base + offset.

The general purpose registers (GPRs) are eight in number and have no special restrictions. Unlike the MSP scalar unit, register 0 is general purpose (ie not fixed to zero),

The ALU is a generic ALU. See Appendix??? on BSP instruction set.

The multiplier is a two cycle multiplier (See programming restrictions) with the lower 16 bits of the product being stored in a GPR and the upper 16 bits of the product being stored in the CMP_h register. The multiplier is intended to speed the computation of addresses for the DMA controller for the MPEG decode application.

The ZZ block performs inverse Scan translation on the lower six bits of the source GPR and puts a zero-extended, translated, six bits back into a GPR. This functionality is used in processing of escape codes in the MPEG and H.261 compression standards.

4.7.6.2 Bitstream Buffer

The bitstream buffer consists of two principle components. The first is a 64 byte (16 x 32) first in first out (FIFO) which is filled by the VICE DMA controller. It is emptied by the bitstream processor. When the bitstream processor requires more bitstream data, the bitstream processor fetches 32-bits at a time from the bitstream FIFO. These 32-bits are fetched into the “Beta register” which is part of the other principle component, the bitstream shifter. The bitstream shifter consists of two 32-bit registers and a 64-bit shifter. Figure 51, “Bitstream Buffer,” on page 194, shows a block diagram of the bitstream buffer.

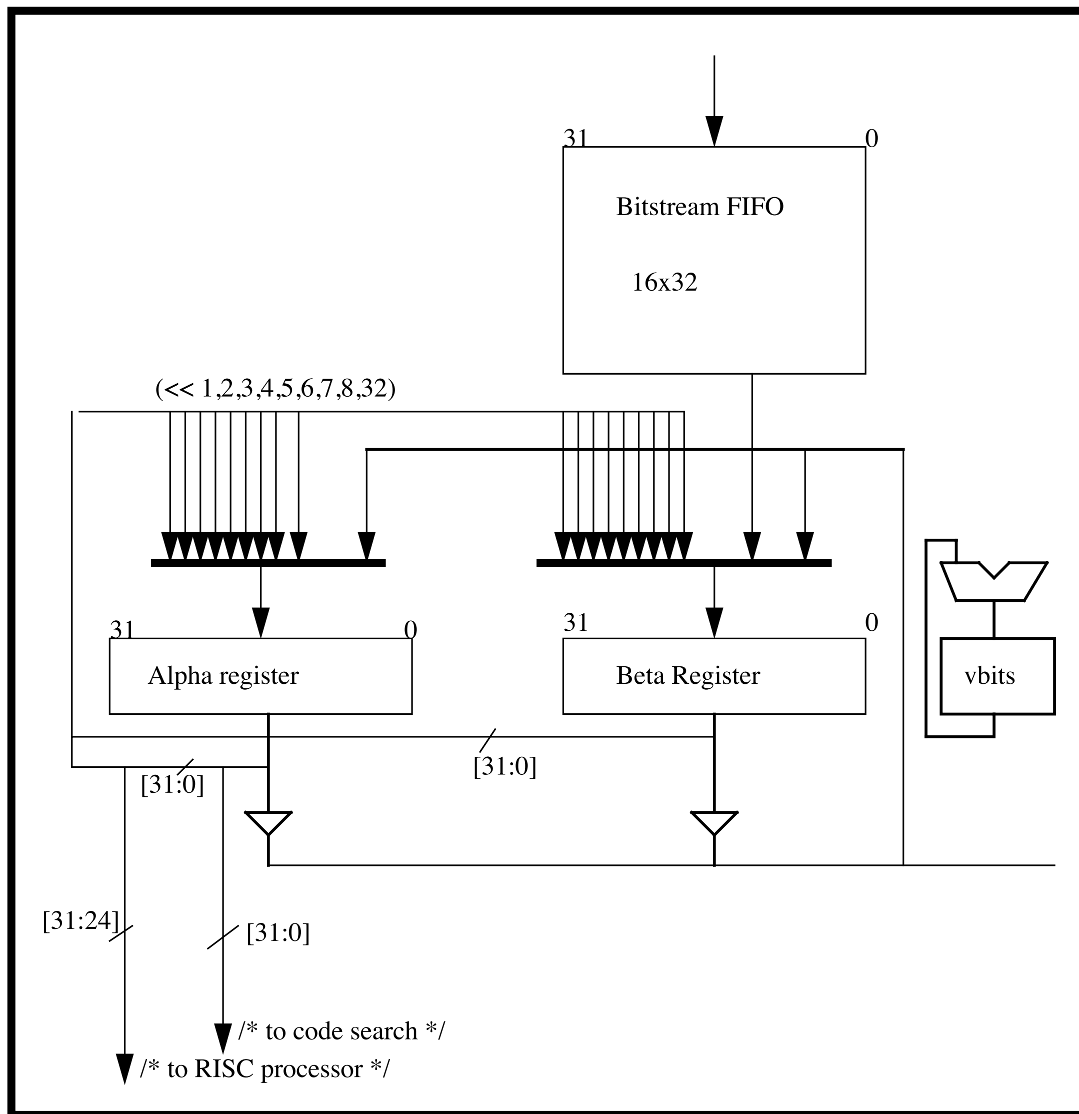


FIGURE 51. Bitstream Buffer

The bitstream buffer keeps the bitstream left justified in the alpha and beta registers. Hence, the bitstream is read left to right. When the bitstream processor gets bits from the bitstream buffer it gets bits from the left-most eight bits of the alpha register (alpha[31:24]). When the bitstream processor gets N bits from the bit-stream (N <=8) the N bits are returned right justified with zero fill to the bitstream processor. These bits are put into a GPR in the RISC processor. When the N bits are read from the head of the alpha register, the alpha and beta registers are shifted left by N bits so that the new head of the bitstream is in the most significant part of the alpha register. Before the shift of the bitstream is performed, the number of valid bits in the beta register is queried. If the number of valid data bits in beta is greater than or equal to N, the shift is performed in one step. In the event that the number of valid bits in beta is less than N, the shift takes place in two steps. The first step shifts the data by the number of valid bits (vbits). Then the next 32-bits in the bitstream FIFO are fetched into the beta register. Having fetched the 32-bits into beta, the bitstream is shifted left by (N-vbits) and the number of valid bits is updated to be 32 - (N - vbits).

This multi-step operation will cause the bitstream processors' RISC processor to stall operation until the multi-cycle bitstream fetch is completed. In the event that no bitstream bits are available in the bitstream FIFO to fill the beta register, the bitstream processor shall----------?????? to be completed.

4.7.6.3 Code Search

The code search section of the bitstream processor allows the bitstream processor to search for codes from one to 32 bits in the bitstream. This is particularly useful for searching for "start" codes in the MPEG bitstream or "markers" in the JPEG bitstream. The codes search section of the bitstream processor contains two 32-bit registers each of which are made up from two 16-bit registers. The compare register (CMP) is used as a reference. It contains the code one is searching for in the bitstream. The mask register (MASK) is a register which is used to specify which bits in the 32-bit compare register are to take part in the compare operation. Figure 52, "BSP Code Search Block Diagram," on page 195 shows the Code search mechanism.

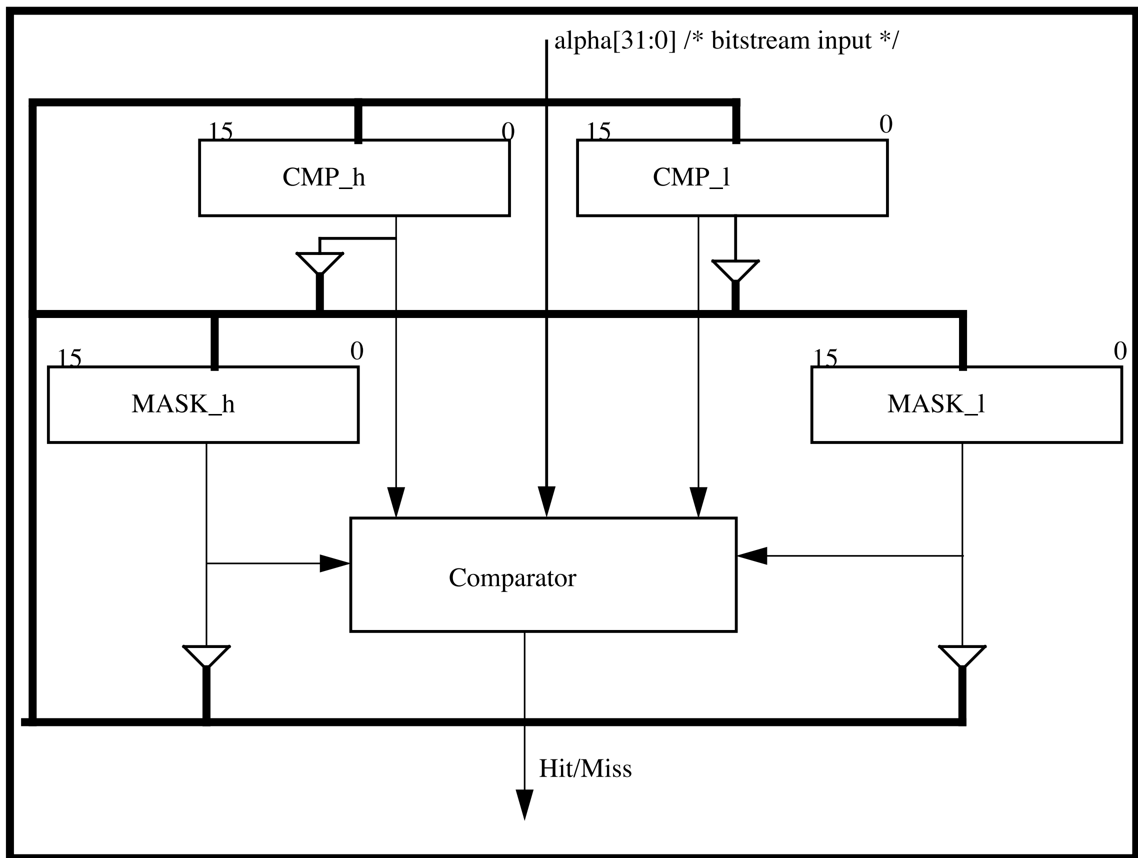


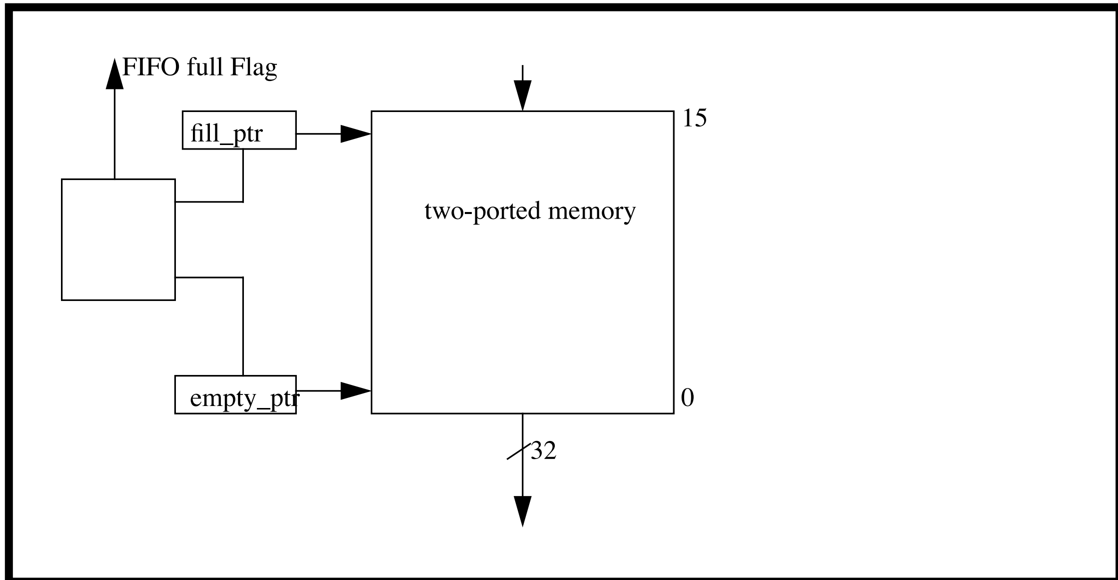
FIGURE 52. BSP Code Search Block Diagram

4.7.6.4 VLC Table Parser

4.7.6.5 Bitstream FIFO

The bitstream FIFO is a 64-byte (16 x 32-bit) first-in-first-out memory buffer. In decoding applications it is filled with bitstream data (VICE DMA) and the BSP empties it 32-bits at a time. In encoding applications, this FIFO is filled with bitstream data from the BSP's write buffer and emptied by VICE DMA.

Figure?? shows the bitstream FIFO.

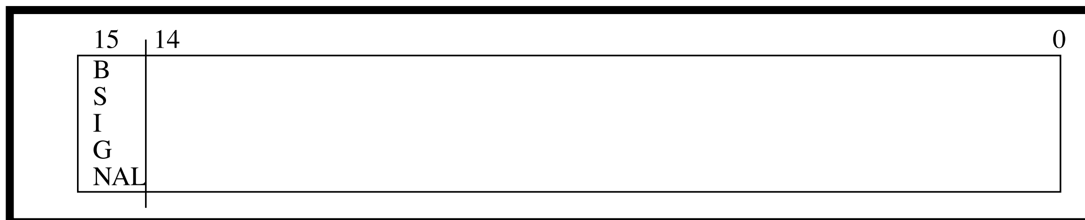


4.7.7 Bitstream Processor / Scalar Unit Synchronization

Program synchronization between the BSP and the Scalar Unit on VICE is done by sending messages between the two sub-units via two unidirectional mailbox registers.

4.7.7.1 BSP_MBOX register.

The BSP_MBOX register is a 16-bit register which is readable by the BSP and both readable and writable by the Scalar Unit. This register is also accessible by the Host CPU. The register format is shown below:

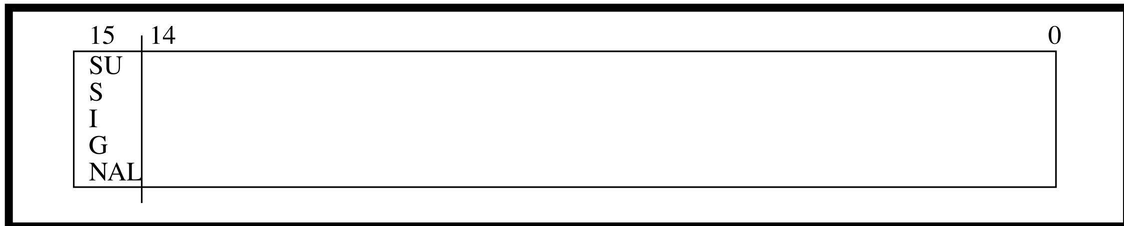


The SU can send a message to the BSP by writing the message in the BSP_MBOX register. This message must fill bits 14-0. When the SU writes into this mailbox register, the BSIGNAL bit (15) is set to one, irrespective of the value which the SU may put on the commensurate bit of its data bus. The SU may read the

contents of this register non-destructively (an SU read of this register will NOT affect its contents). When the BSP reads from this register (LoadH instruction) the BSIGNAL bit (15) is copied into the BSP along with the rest of the register contents. This BSP read, however, clears the BSIGNAL bit for subsequent reads.

4.7.7.2 MSP_MBOX register.

The MSP_MBOX register is a 16 bit register which is readable by the Scalar Unit and both readable and writable by the BSP. This register is also accessible by the host CPU. The register format is shown below:



The BSP can send a message to the SU by writing the message in the MSP_MBOX register. This message must fill bits 14-0. When the BSP writes to this mailbox register, the SUSIGNAL bit (15) is set to one. The BSP may read the contents of this register non-destructively (A BSP read of this register will not affect its contents). When the Scalar Unit reads from this register (MFC instruction). the SUSIGNAL bit is copied to the Scalar Unit along with the rest of the register contents. This SU read, however, clears the SUSIGNAL bit for subsequent reads.

5.0 Operational Description

How various primitives are implemented using this hardware. Could also be used to quantify system bottleneck issues such as data flow rates for various primitives. May be more likely to move the Bent's document called "A Day in the Life of Vice" that talks about the interface between the Unix Processor and Operating System and the onboard MSP activity.

6.0 Performance Analysis

In this section we describe the performance of the VICE chip for a number of application. Each application has different demands on each component ot the VICE architecture. Some applications are very demanding of the bitstream processor while others are very demanding of the computation capability of the vector unit. Most applications place a lot of demand on the DMA processor to provide data in a timely manner.

The VICE chip has a clock Period of 15ns. corresponding to a clock rate of 66 MHz.

Most performace estimates for the applications described are for a level of granularity appropriate for the application. Form MPEG and Px64, this is the Macroblock level. For JPEG it is the MCU level. This granularity is motivated mostly by the available storage on the VICE chip.

Given the macroblock or MCU granularity, it is useful to compute how many clock cycles are available for processing the unit of granularity. For the applications subsequently described, this is principally motivated by picture size and picture rate. ***Hence, for a 720x480 (CCIR-601) picture (1350 Macroblocks) at 30 fps, the time to process a macroblock is approximately 1600 clock cycles.***

For a 720x480 (CCIR 601) we have chosen to have 2700 MCUs in the JPEG still image compression algorithme. ***If one were to try to encode or decode these JPEG pictures at 30 fps there are 800 cycles available to process each JPEG MCU.***

Note that a JPEG MCU is assumed to have 4:2:2 structure, while the MPEG macroblock is assumed 4:2:0.

6.0.1 Peak Hardware Performance

6.1 Baseline JPEG Decode Application

This application is very demanding on the bitstream processor when the compression ration of the bitstream being decoded is small. VICE will be able to decode 4:1 compressed bitstreams (4:2:2 MCU).

The followinglist shows the performance requirements for decoding an MCU from each of the Bitstream Processor, Sclar Unit/Vector Unit and DMA processor:

Scalar Unit/ Vector Unit:

Inverse Quantization:120 clock cycles per MCU

Inverse Discrete Cosine Transform: (4 8x8 blocks) 90 clock cycles per 8x8, giving 360 clocks per 4:2:2 MCU.

Bitstream Proceesor:

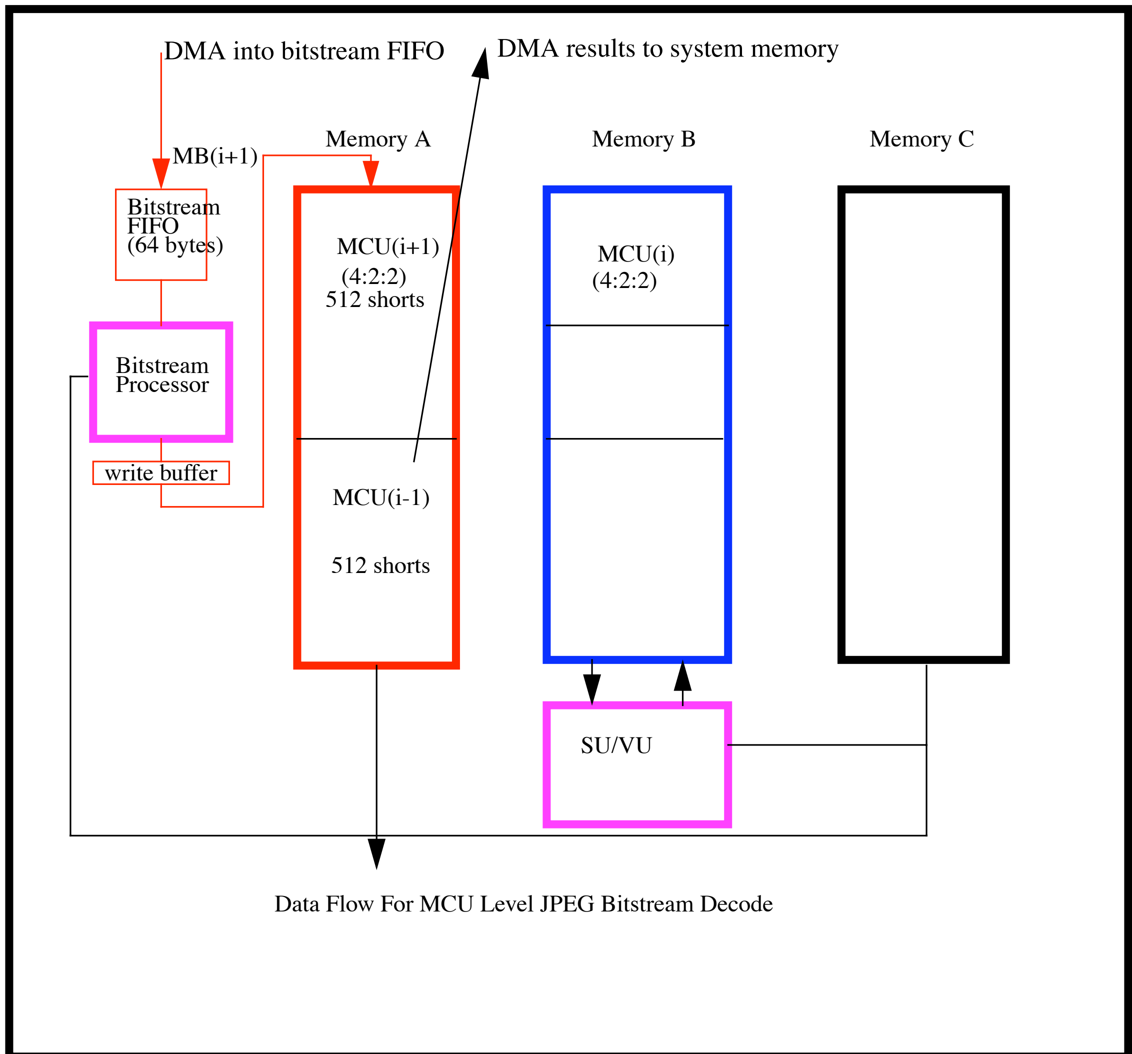
Decode of 8 4:1 compressed 8x8 blocks: The bitstream processor performance is driven by both the number of bits to be decoded as well as the number of tokens to be encoded. From a token based perspective, a 4:1 compression would have 128 tokens (8-bit coded tokens) . It takes the bitstream processor 5 cycles to decode 8 bits. Hence it takes 5*128 or 640 cycles to decode such a bitstream. From a bitrate based perspective, a 4:1 compressed bitstream should have 1024 bits in it. If every token is the shortest token (2 bits) this means decoding 512 tokens. It takes 3 cycles to decode a 2-bit token, hence it will take 1536 cycles to decode such a MCU. Since, the former case (token based perspective) is more realistic (quantization is aimed as reducing the number of tokens) it is expected that decoding of 4:1 compressed bitstrteams is viable. The bitrate based approach being an extreme worst case.

DMA Processor:

Fetch of 4:1 compresses bitstream (~64 bytes for an MCU) (180 clock cycle)

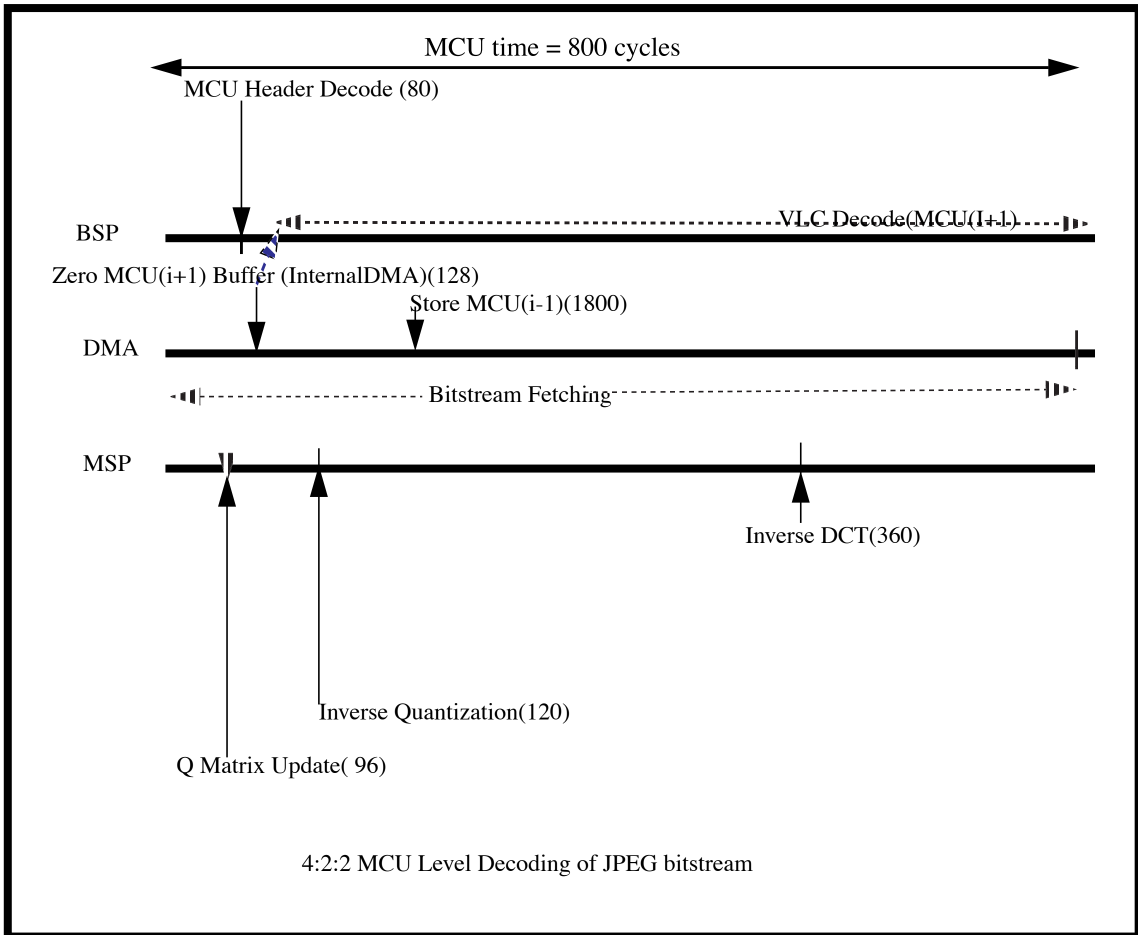
Store of Decompressed MCU:

6.1.1 Baseline JPEG decode Data Flow



Data Flow For MCU Level JPEG Bitstream Decode

6.1.2 Baseline JPEG decode registration Diagram



6.2 Baseline JPEG Encode (lossy) Application

Scalar Unit/ Vector Unit:

Quantization:: 120 cycles.

Discrete Cosine Transform: (4 8x8 blocks): 90 clock cycles per block or 360 cycles per MCU

Bitstream Proceesor:

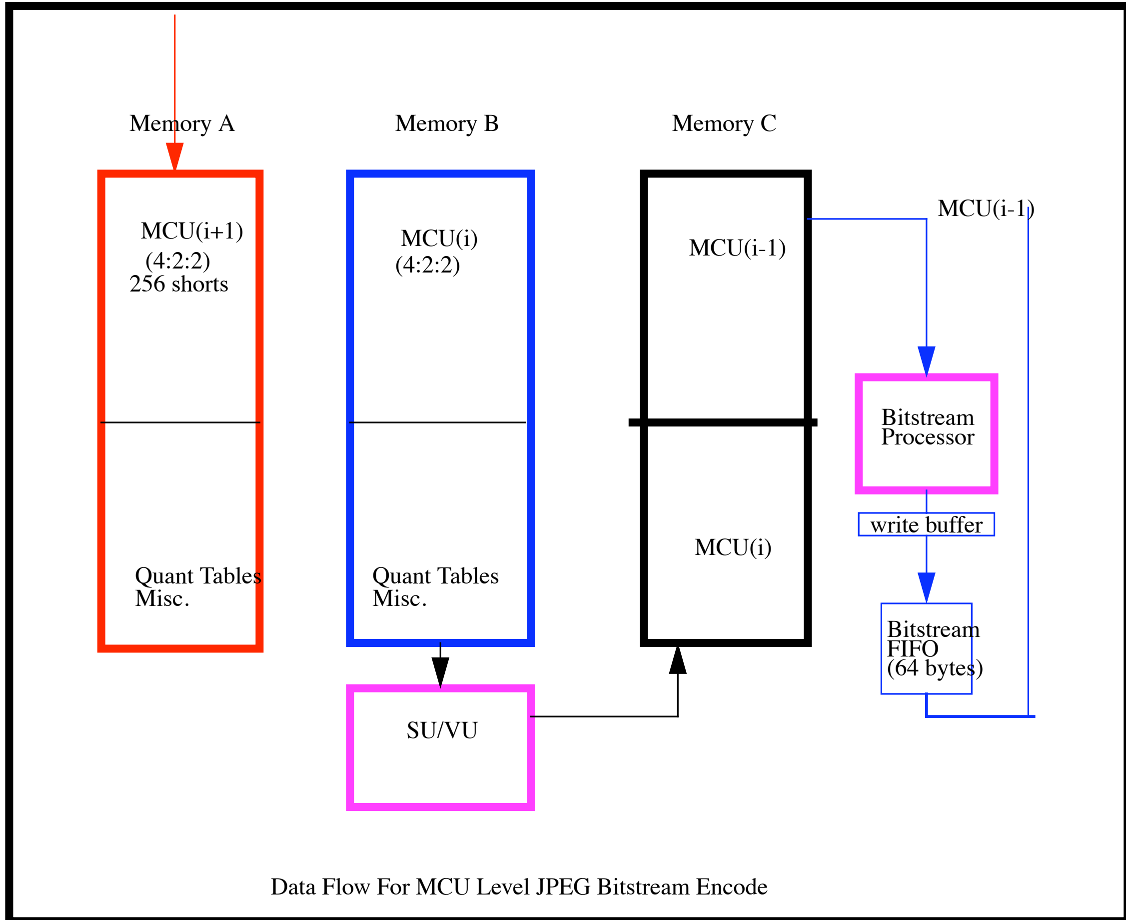
Encode of 8 4:1 compressed 8x8 blocks: The bitstream processor can “Huffman code” an 8x8 block in 128 clock cycles.

DMA Processor:

Fetch of input MCU

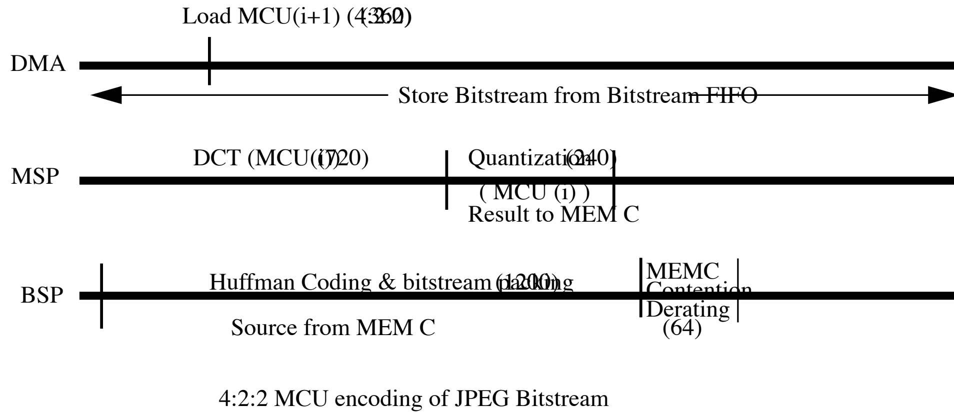
Store of compressed (4:1) MCU (~ 128 bytes for 4:2:2 MCU)

6.2.1 Baseline JPEG Encode MCU data flow:



Data Flow For MCU Level JPEG Bitstream Encode

6.2.2 Baseline JPEG Encode registration diagram:



4:2:2 MCU encoding of JPEG Bitstream

Note, the MSP is putting results into MEM C while the BSP is taking data to be Hufman coded from MEM C.

There are 64 stores to MEM C and 512 reads from MEM C, hence, given the BSP priority for memory, the time to perform Huffman encoding will be lengthened by approx 64 cycles. This accounts for the derating time shown above.

6.3 Lossless JPEG Application

Our studies indicate that the VICE engine can perform lossless JPEG using any of the seven predictors and Huffman coding. See: Tuffli; Lossless JPEG on VICE; internal memo; Sept. 1994

6.4 MPEG-2 Decode Application

Scalar Unit/ Vector Unit:

Update quantization matrix:

Inverse Quantization:

Discrete Cosine Transform:: (540)

Predictor averaging and Motion Compensation:

Bitstream Processor:

Macroblock Header Decode: (120)

Motion Vector Extraction (FWD/BWD, FIELD0/FIELD1, Horizontal/Vertical) (240)

Motion Vector Computation:

Predictor Fetch Set Up.

DMA Processor:

Store of Reconstructed Macroblock

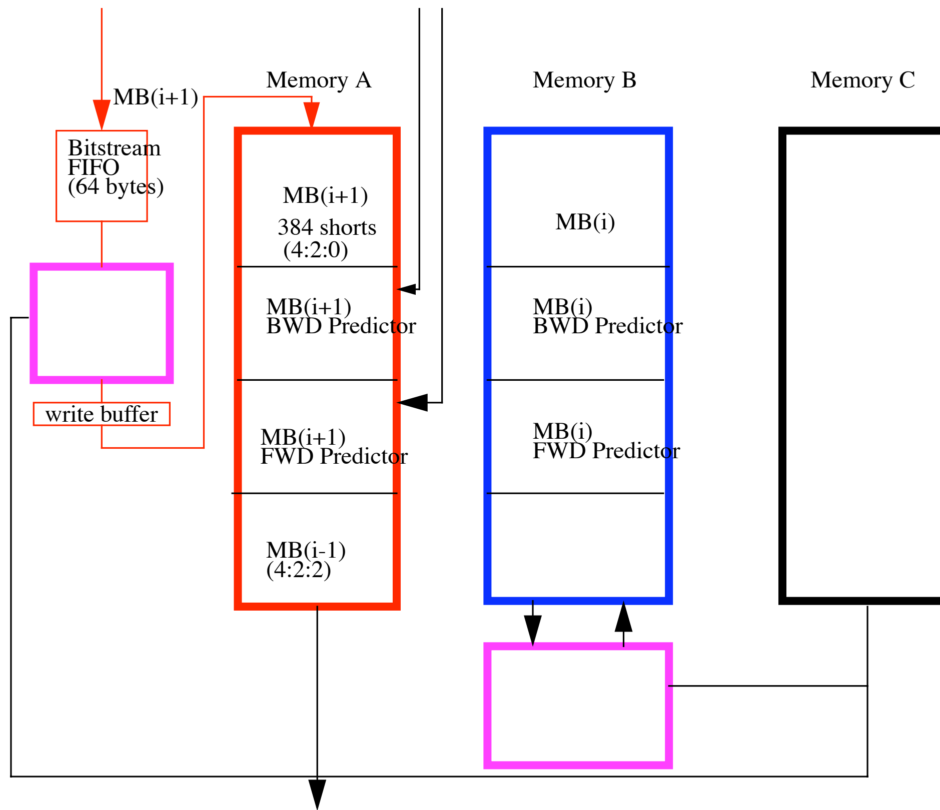
Fetch of FWD Predictor;(Field Prediction) w Half-pel interpolation:

Fetch of BWD predictor (Field Prediction) w Half-pel interpolation:

Fetch of bitstream.

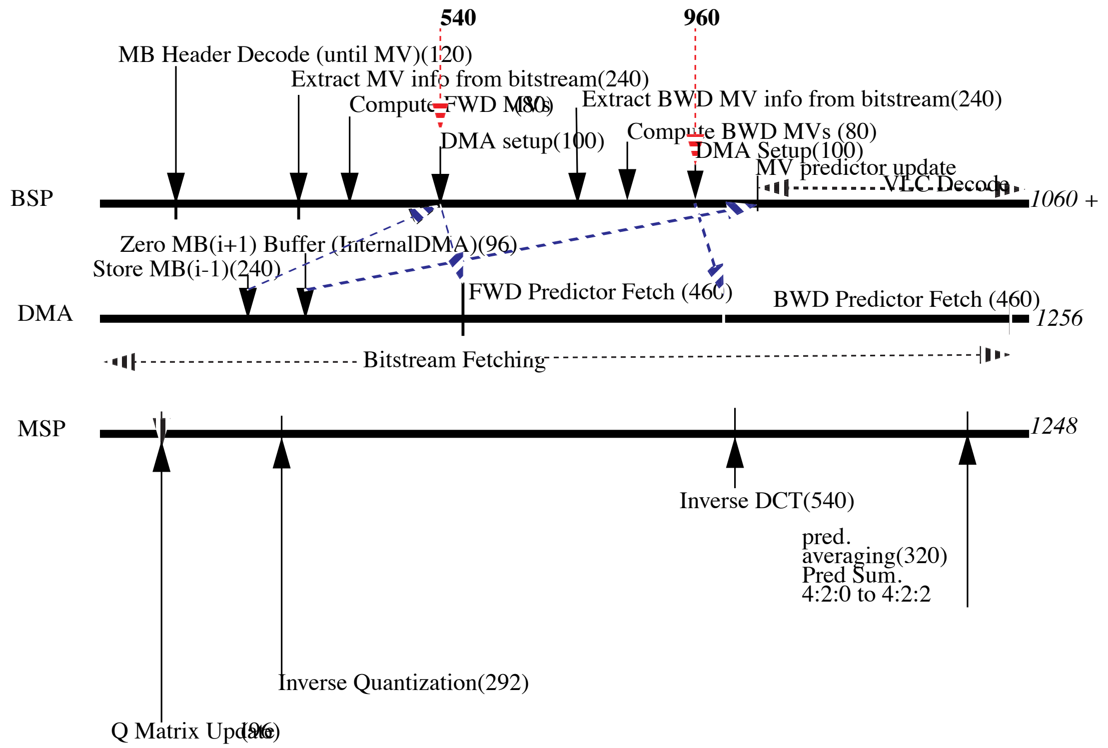
Predictor averaging and Motion Compensation:

6.4.1 MPEG-2 Decode Application Data Flow:



Data Flow For Macroblock Level MPEG-2 Bitstream Decode

6.4.2 MPEG-2 Decode Application registration Diagram



Macroblock Level Decoding of MPEG-2 Bitstream

Bitstream

6.5 H.261 Application

6.6 Image Vision Library Primitives

7.0 Precision Analysis

7.1 Scalar Unit

7.2 Vector Unit

8.0 Device Interface

8.1 Signal Descriptions

Figure 53, "Logical pin diagram of VICE," on page 208 shows the Pins on VICE arranged by functional grouping. Table 73, "VICE Pin Descriptions," on page 209 covers detailed characteristics of each VICE pin (Drive Strength, direction and description).

All I/O signals on VICE and the R4ValidIn_n output are 3-state pins that operate as such in the Moosehead system. These same I/O signals and the R4ValidIn_n output will be put in the high-impedance state with the Reset_n signal when asserted in the Moosehead system. For board test, these same signals and all other chip outputs can be placed in high impedance mode with the Tristate_en pin.

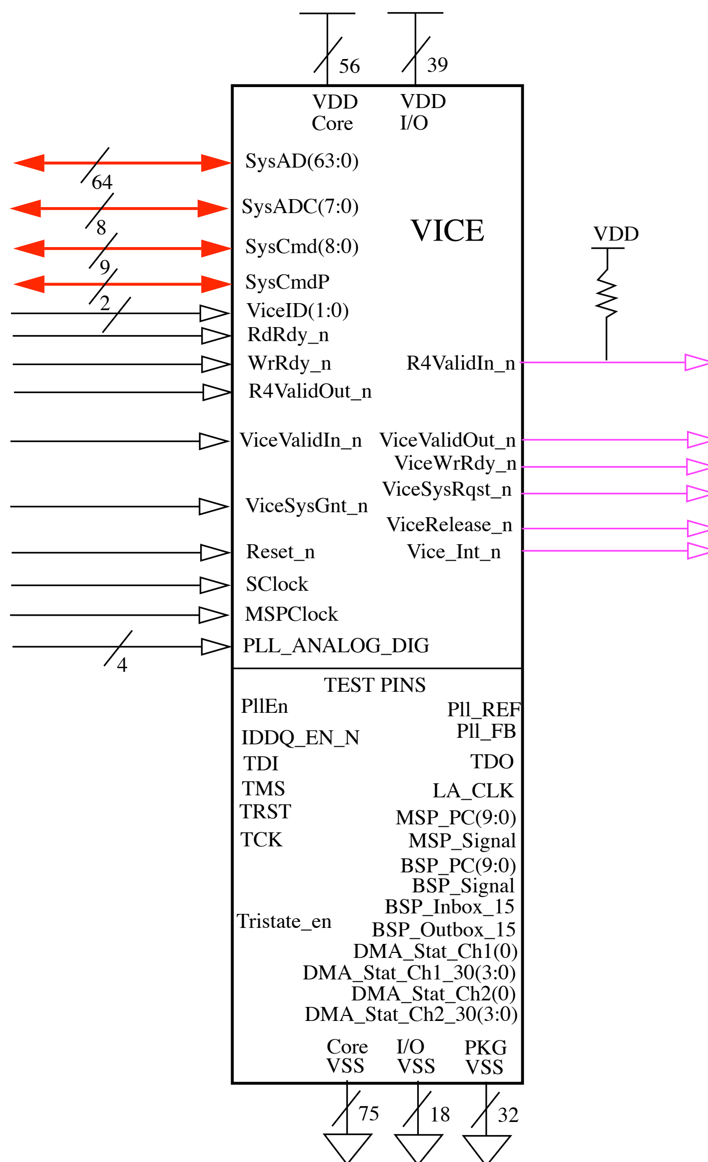


FIGURE 53. Logical pin diagram of VICE

TABLE 73. VICE Pin Descriptions

Pin Name	Type	Description	Output Drive
SysAD(63:0)	I/O	64-bit Multiplexed Address/Data bus. Bidirectional signals used to communicate with the Unix Processor and the CRIME chip. Cycles on the SysAD which contain a valid address are called *address cycle*s. Cycles on the SysAD which contain valid data are called *data cycles*. Validity is identified by the R4ValidOut_n, R4ValidIn_n, ViceValidOut_n and ViceValidIn_n pins.	8 mA
SysADC(7:0)	I/O	8-bit SysAD check bus (even parity). An 8-bit bus containing check bits for the SysAD bus.	8 mA
SysCmd(8:0)	I/O	9-bit command bus. The SysCmd bus identifies the contents of the SysAD bus during any cycle in which it is valid. SysCmd(8) is used to indicate whether the current cycle is an address cycle [SysCmd(8)=0] or a data cycle [SysCmd(8)=1].	8 mA
SysCmdP	I/O	1-bit SysCMD check (even parity) for the SysCmd bus.	8 mA
SClock	I	Input System Clock for SysAD bus Synchronization. The SClock is fed to CRIME, VICE and the Unix Processor which all share the SysAD bus. CRIME and VICE use an internal Phase Lock Loop to align the phase of SClock on the clock tree inside of these chips with the phase of SClock at the input pin. Only the Host Interface block inside of VICE uses the PLL generated SClock.	
RdRdy_n	I	External agent ready to accept a new read. VICE samples the signal RdRdy_n to determine the issue cycle for a processor read request.	
WrRdy_n	I	External agent ready to accept a new write. VICE samples the signal WrRdy_n to determine the issue cycle of a processor write request.	
R4ValidOut_n	I	Data from Unix processor is valid. The Unix processor drives this signal. Both CRIME and VICE monitor this signal. All Unix processor address cycles (read or write) drive this signal active. Unix processor write data cycles drive this signal active.	
R4ValidIn_n	O	Data to Unix processor is valid. If the cycle is a read response data cycle from CRIME to the Unix processor, the CRIME chip drives this signal. If the cycle is a read response data cycle from VICE to the Unix processor, the VICE chip drives this signal. It is a 3-state signal on both CRIME and Vice.	8 mA 3-state shared with CRIME
ViceWrRdy_n	O	To CRIME. This signal is used when the Unix processor writes to the VICE chip. If the write buffer in VICE has 4 entries or less, VICE asserts this signal. The signal is reclocked by CRIME and presented to the Unix processor its WrRdy_n pin. This signal is not used during transactions between VICE and Crime.	8 mA

Pin Name	Type	Description	Output Drive
ViceValidOut_n	O	Data from VICE is valid. VICE drives this signal which is monitored by Crime. All VICE address cycles (read or write) drive this signal active. VICE write data cycles drive this signal active.	8 mA
ViceValidIn_n	I	Data to VICE is valid. CRIME drives this signal which is monitored by Vice. CRIME asserts this signal during read response data cycles.	
ViceSysRqst_n	O	VICE request to use SysAD bus. VICE drives this signal which is monitored by Crime. Any VICE initiated address or data cycle must be preceded by bus ownership. CRIME will request the SysAD bus from the Unix processor when VICE asserts ViceSysRqst_n. VICE will de-assert this signal in the cycle after it receives ViceSysGnt_n.	8 mA
ViceSysGnt_n	I	OK to use SysAD bus. CRIME drives this signal when the Unix processor has Released the SysAD bus to CRIME as the result of a ViceSysRqst_n assertion. CRIME will assert this signal for one SClock cycle when granting the SysAD bus to VICE.	
ViceRelease_n	O	VICE releases SysAD bus. VICE drives this signal which is monitored by CRIME. VICE will assert this signal for one SClock cycle when releasing the SysAD bus. VICE will perform SysAD transactions for up to 8 pipelined block transfers before releasing the SysAD bus.	8 mA
VClock	I	VICE Input Clock 66MHz. Used in the MSP, BSP and DMA blocks of the chip. These blocks use this clock exclusively and are functionally isolated from the clock domain of the SysAD bus (SClock). The Host Interface block inside of VICE also uses this signal when crossing over to the SysAD bus clock domain (SClock).	
VICE_Int_n	O	VICE drives this signal which is monitored by the CRIME chip. This is a level sensitive interrupt signal that is a collection of interrupts from devices internal to the VICE chip. It is also possible that this interrupt may be connected directly to the Unix Processor.	8 mA
Reset_n	I	Reset Signal from CRIME. This signal is driven by CRIME to both VICE and the Unix Processor. It is used by VICE to 3-State SysAD I/O pins and the R4ValidIn_n output pin asynchronous to SClock. Vice re-synchronizes this signal to the SClock and VClock domains for use in resetting all internal state machines.	
Vice_ID(1:0)	I	Vice Id pins. Allows Vice to respond to one of four address ranges. Mapped to bits A21 and A20 of the SysAD bus. 0x0 170X XXXX = Vice_ID 00 0x0 171X XXXX = Vice_ID 01 0x0 172X XXXX = Vice_ID 02 0x0 173X XXXX = Vice_ID 03	
TCK	I	Input for Boundary Scan Clock. Selected per JTAG 1149 controller mode.	

Pin Name	Type	Description	Output Drive
TDI	I	Data Input Pin for Boundary Scan Test.	
TMS	I	Test Mode Select - Boundary Scan Tap Controller	
TRST	I	Test Mode Reset - Boundary Scan Tap Controller	
TDO	O	Test Data Out - Boundary Scan Chain	8 mA
TristateEn	I	1 - Ouputs Active, PLL Enabled and Powered On 0 - Outputs 3-State, PLL Disabled and Powered Off This signal can be asserted asynchronously.	
PLL_AG	GND	Analog Phase Lock Loop Ground	
PLL_AP	PWR	Analog Phase Lock Loop Power	
PLL_BYPASS	I	1- Bypass Internal Phase Lock Loop 0- Use Internal Phase Lock Loop - Normal System Config.	
PLL_DIV20	O	Phase Lock Loop Clock Signal / 2 Output for Test Purposes	2 mA
PLL_FB	O	Phase Lock Loop Clock Signal Output for Test Purposes	2 mA
PLL_REF	O	Clock Signal Input to Phase Comparator of PLL for Test Purposes. (SClock buffered)	2 mA
PLL_TESTE	I	1- Pll_FB and PLL_REF outputs enabled 0- Pll_FB and PLL_REF outputs disabled - Normal System Config.	
MSP_PC(9:0)	O	Media Signal Process Program Counter - Bits 12:3	2 mA
MSP_SIGNAL	O	Bit 4 of BSP_FIFO_STAT register for Hardware Observ.	2 mA
BSP_PC(9:0)	O	Bit Stream Process Program Counter - Bits 10:1	2 mA
BSP_SIGNAL	O	Bit 3 of BSP_FIFO_STAT register for Hardware Observ.	2 mA
BSP_OUTBOX_15	O	Mailbox flag of BSP outbox. Reset by MSP read. Set by BSP write. Available for Hardware Observation.	2 mA
BSP_INBOX_15	O	Mailbox flag of BSP inbox. Set by MSP write. Reset by BSP read. Available for Hardware Observation.	2 mA
DMA_Stat_Ch1(0)	O	Copy of DMA_STAT_CH1 register bit 0. 0=DMA Not complete 1=DMA Complete	2 mA
DMA_Stat_Ch1_30 (3:0)	O	Copy of DMA_STAT_CH1 register bits 11:8 DMA_STAT_CODE	2 mA
DMA_Stat_Ch2(0)	O	Copy of DMA_STAT_CH2 register bit 0. 0=DMA Not complete 1=DMA Complete	2 mA
DMA_Stat_Ch2_30 (3:0)	O	Copy of DMA_STAT_CH2 register bits 11:8 DMA_STAT_CODE	2 mA
VDDI	PWR	56Pins Core +3.3V Power (24 more pad connections from die to pakage internal)	
VDDE	PWR	39Pins I/O +3.3V Power	
VSSI	GND	75Pins Core Ground (28 more pad connections from die to pakage internal)	

Pin Name	**Type**	**Description**	**Output Drive**
VSSE	GND	18Pins I/O Ground (88 more pad connections from die to package internal)	
VSS_PKG	GND	32 Ground connections to TBGA internal Ground Plane.	

8.2 Pin Assignments

The following table provides the physical pin to signal name assignments sorted by the TGBA pin number.

TABLE 74. 380 BGA Pin Assignments - Pin Order

Pin #	Signal	Pin #	Signal	Pin #	Signal
A01	VSS_PKG	A02	VSS	A03	TDI
A04	VSS_PKG	A05	TRST	A06	TCK
A07	VSS_PKG	A08	VDD	A09	VSS
A10	VSS_PKG	A11	VSS	A12	VCLOCK
A13	VSS_PKG	A14	VSS	A15	VSS_PKG
A16	VSS	A17	VSS	A18	VSS_PKG
A19	VSS	A20	VSS	A21	VSS_PKG
A22	VSS	A23	VSS	A24	VSS_PKG
B01	VSS	B02	VSS	B03	VDD
B04	MSP_PC(9)	B05	VSS	B06	VSS
B07	MSP_PC(7)	B08	MSP_PC(6)	B09	VDD
B10	VSS	B11	VSS	B12	VSS
B13	VSS	B14	TEST_OUT_ENB	B15	VSS
B16	VDD	B17	MSP_PC(4)	B18	VDD
B19	VDD	B20	MSP_PC(2)	B21	VDD
B22	VDD	B23	VDD	B24	VSS
C01	SYSAD(32)	C02	VDD	C03	VDD
C04	TMS	C05	TDO	C06	MSP_PC(8)
C07	VSS	C08	VDD	C09	VSS
C10	VDD	C11	LA_CLK	C12	VDD
C13	VSS	C14	VDD	C15	VDD
C16	MSP_PC(3)	C17	VSS	C18	MSP_PC(1)
C19	VDD	C20	VDD	C21	VSS
C22	VDD	C23	BSP_SIGNAL	C24	VSS
D01	VSS_PKG	D02	SYSAD(2)	D03	SYSAD(0)
D04	VSS	D05	VSS	D06	VDD
D07	VSS	D08	VDD	D09	VSS
D10	VSS	D11	VDD	D12	IDDQ_EN_N
D13	VDD	D14	TRISTATEEN	D15	VSS
D16	VSS	D17	MSP_PC(0)	D18	VSS
D19	VSS	D20	VDD	D21	VSS
D22	SYSAD(1)	D23	SYSAD(33)	D24	VSS_PKG
E01	SYSAD(34)	E02	SYSAD(36)	E03	SYSAD(4)
E04	VDD	E05	VDD	E06	VDD
E07	VSS	E08	VDD	E09	VDD
E10	VDD	E11	MSP_PC(5)	E12	VDD
E13	VSS	E14	VDD	E15	VDD
E16	VDD	E17	VDD	E18	VSS

TABLE 74. 380 BGA Pin Assignments - Pin Order

Pin #	Signal	Pin #	Signal	Pin #	Signal
E19	VDD	E20	VDD	E21	VSS
E22	SYSAD(3)	E23	SYSAD(35)	E24	VSS
F01	SYSAD(8)	F02	SYSAD(38)	F03	SYSAD(6)
F04	VDD	F05	VDD		
		F20	VDD	F21	VSS
F22	SYSAD(7)	F23	SYSAD(5)	F24	VDD
G01	VSS_PKG	G02	SYSAD(10)	G03	SYSAD(40)
G04	VDD	G05	VSS		
		G20	VDD	G21	VDD
G22	SYSAD(37)	G23	SYSAD(39)	G24	VSS_PKG
H01	SYSAD(42)	H02	SYSAD(44)	H03	SYSAD(12)
H04	VDD	H05	VSS		
		H20	VSS	H21	VDD
H22	SYSAD(9)	H23	SYSAD(41)	H24	VDD
J01	SYSAD(46)	J02	SYSAD(14)	J03	SYSAD(60)
J04	VSS	J05	VDD		
		J20	VDD	J21	VSS
J22	SYSAD(11)	J23	VDD	J24	VSS
K01	VSS_PKG	K02	SYSAD(30)	K03	SYSAD(62)
K04	VDD	K05	VDD		
		K20	VDD	K21	PLL_FB
K22	PLL_BYPASS	K23	VSS	K24	VSS_PKG
L01	SYSAD(58)	L02	SYSAD(28)	L03	SYSAD(26)
L04	VSS	L05	VDD		
		L20	SCLOCK	L21	PLL_AP
L22	PLL_AG	L23	PLL_DIV20	L24	PLL_TESTE
M01	VSS_PKG	M02	SYSAD(56)	M03	SYSAD(24)
M04	VDD	M05	VSS		
		M20	PLL_REF	M21	VCOK
M22	SYSAD(43)	M23	SYSAD(13)	M24	VDD
N01	SYSAD(54)	N02	SYSAD(22)	N03	SYSAD(50)
N04	VSS	N05	VDD		
		N20	VDD	N21	VSS
N22	SYSAD(45)	N23	SYSAD(63)	N24	VSS_PKG
P01	SYSAD(52)	P02	SYSAD(20)	P03	SYSAD(18)
P04	VDD	P05	VSS		
		P20	VSS	P21	VDD
P22	SYSAD(21)	P23	SYSAD(61)	P24	VSS
R01	VSS_PKG	R02	SYSAD(48)	R03	SYSAD(16)
R04	VSS	R05	VDD		
		R20	VDD	R21	VSS

TABLE 74. 380 BGA Pin Assignments - Pin Order

Pin #	Signal	Pin #	Signal	Pin #	Signal
R22	SYSAD(31)	R23	SYSAD(27)	R24	VSS_PKG
T01	SYSADC(0)	T02	SYSADC(2)	T03	SYSADC(4)
T04	VSS	T05	VDD		
		T20	VDD	T21	VSS
T22	SYSAD(59)	T23	SYSAD(25)	T24	VDD
U01	SYSADC(6)	U02	VDD	U03	VSS
U04	VDD	U05	VDD		
		U20	VSS	U21	VSS
U22	SYSAD(57)	U23	SYSAD(55)	U24	VSS
V01	VSS_PKG	V02	VDD	V03	VSS
V04	VSS	V05	VSS		
		V20	VDD	V21	VDD
V22	SYSAD(53)	V23	SYSAD(23)	V24	VSS_PKG
W01	SYSCMDP	W02	VDD	W03	VSS
W04	VSS	W05	VDD		
		W20	VDD	W21	VSS
W22	SYSAD(19)	W23	SYSAD(51)	W24	VDDE
Y01	SYSCMD(0)	Y02	SYSCMD(2)	Y03	VDD
Y04	VDD	Y05	VDD	Y06	VDD
Y07	VSS	Y08	VDD	Y09	VSS
Y10	VDD	Y11	VSS	Y12	VDD
Y13	VDD	Y14	DMA_STAT_CH2_30(1)	Y15	VDD
Y16	VDD	Y17	VDD	Y18	BSP_PC(9)
Y19	VDD	Y20	VDD	Y21	VDD
Y22	SYSAD(21)	Y23	SYSAD(17)	Y24	VDD
AA01	VSS_PKG	AA02	SYSCMD(1)	AA03	VDD
AA04	VSS	AA05	VDD	AA06	VDD
AA07	VSS	AA08	VDD	AA09	VDD
AA10	VSS	AA11	VICEID(1)	AA12	DMA_STAT_CH1(0)
AA13	VDD	AA14	VDD	AA15	DMA_STAT_CH2_30(3)
AA16	BSP_PC(1)	AA17	BSP_PC(4)	AA18	VSS
AA19	BSP_OUTBOX_15	AA20	VSS	AA21	VSS
AA22	SYSAD(49)	AA23	SYSADC(7)	AA24	VSS_PKG
AB01	SYSCMD(3)	AB02	SYSCMD(4)	AB03	VDD
AB04	VSS	AB05	VSS	AB06	VSS
AB07	VICERELEASE_N	AB08	VICESYSRQST_N	AB09	VICESYSGNT_N
AB10	VDD	AB11	VSS	AB12	DMA_STAT_CH2(0)
AB13	VSS	AB14	VSS	AB15	DMA_STAT_CH2_30(2)
AB16	VSS	AB17	BSP_PC(0)	AB18	VDD
AB19	BSP_PC(6)	AB20	BSP_INBOX_15	AB21	VDD
AB22	SYSADC(5)	AB23	SYSAD(47)	AB24	VDD

TABLE 74. 380 BGA Pin Assignments - Pin Order

Pin #	Signal	Pin #	Signal	Pin #	Signal
AC01	SYSCMD(5)	AC02	SYSCMD(6)	AC03	VDD
AC04	VDD	AC05	VSS	AC06	WRRDY_N
AC07	VICEVALIDIN_N	AC08	VICEVALIDOUT_N	AC09	VICEWRRDY_N
AC10	RESET_N	AC11	VDD	AC12	DMA_STAT_CH1_30(1)
AC13	DMA_STAT_CH1_30(3)	AC14	VSS	AC15	DMA_STAT_CH2_30(0)
AC16	VDD	AC17	VDD	AC18	MSP_SIGNAL
AC19	BSP_PC(3)	AC20	BSP_PC(5)	AC21	BSP_PC(7)
AC22	SYSADC(1)	AC23	SYSAD(15)	AC24	VSS
AD01	VSS_PKG	AD02	SYSCMD(7)	AD03	SYSCMD(8)
AD04	VSS_PKG	AD05	RDRDY_N	AD06	R4VALIDOUT_N
AD07	VSS_PKG	AD08	VICE_INT_N	AD09	VICEID(0)
AD10	VSS_PKG	AD11	DMA_STAT_CH1_30(0)	AD12	VSS_PKG
AD13	DMA_STAT_CH1_30(2)	AD14	R4VALIDIN_N	AD15	VSS_PKG
AD16	VSS	AD17	VSS	AD18	VSS_PKG
AD19	BSP_PC(2)	AD20	VSS	AD21	VSS_PKG
AD22	BSP_PC(8)	AD23	SYSADC(3)	AD24	VSS_PKG

The following table provides the physical pin to signal name assignments sorted by the Signal Name from the VICE VHDL Data Base.

TABLE 75. Signal Name - Pin Assignment

Signal Name	TBGA Pin #	Die Pad #
BSP_INBOX_15	AB20	228
BSP_OUTBOX_15	AA19	234
BSP_PC(0)	AB17	214
BSP_PC(1)	AA16	215
BSP_PC(2)	AD19	212
BSP_PC(3)	AC19	217
BSP_PC(4)	AA17	224
BSP_PC(5)	AC20	223
BSP_PC(6)	AB19	225
BSP_PC(7)	AC21	226
BSP_PC(8)	AD22	229
BSP_PC(9)	Y18	237
BSP_SIGNAL	C23	357
DMA_STAT_CH1(0)	AA12	181
DMA_STAT_CH1_30(0)	AD11	176
DMA_STAT_CH1_30(1)	AC12	183
DMA_STAT_CH1_30(2)	AD13	191
DMA_STAT_CH1_30(3)	AC13	184
DMA_STAT_CH2(0)	AB12	182
DMA_STAT_CH2_30(0)	AC15	195
DMA_STAT_CH2_30(1)	Y14	198
DMA_STAT_CH2_30(2)	AB15	202
DMA_STAT_CH2_30(3)	AA15	205
IDDQ_EN_N	D12	430
LA_CLK	C11	438
MSP_PC(0)	D17	387
MSP_PC(1)	C18	393
MSP_PC(2)	B20	394
MSP_PC(3)	C16	404
MSP_PC(4)	B17	405
MSP_PC(5)	E11	442
MSP_PC(6)	B08	450
MSP_PC(7)	B07	457
MSP_PC(8)	C06	469
MSP_PC(9)	B04	470

TABLE 75. Signal Name - Pin Assignment

Signal Name	TBGA Pin #	Die Pad #
MSP_SIGNAL	AC18	213
PLL_AG	L22	316
PLL_AP	L21	318
PLL_BYPASS	K22	324
PLL_DIV20	L23	315
PLL_FB	K21	327
PLL_REF	M20	310
PLL_TESTE	L24	314
R4VALIDIN_N	AD14	192
R4VALIDOUT_N	AD06	163
RDRDY_N	AD05	155
RESET_N	AC10	172
SCLOCK	L20	320
SYSAD(0)	D03	9
SYSAD(1)	D22	358
SYSAD(2)	D02	19
SYSAD(3)	E22	350
SYSAD(4)	E03	17
SYSAD(5)	F23	339
SYSAD(6)	F03	20
SYSAD(7)	F22	347
SYSAD(8)	F01	41
SYSAD(9)	H22	336
SYSAD(10)	G02	32
SYSAD(11)	J22	329
SYSAD(12)	H03	31
SYSAD(13)	M23	306
SYSAD(14)	J02	42
SYSAD(15)	AC23	249
SYSAD(16)	R03	80
SYSAD(17)	Y23	272
SYSAD(18)	P03	72
SYSAD(19)	W22	264
SYSAD(20)	P02	71
SYSAD(21)	Y22	261
SYSAD(22)	N02	62
SYSAD(23)	V23	276
SYSAD(24)	M03	60
SYSAD(25)	T23	286

TABLE 75. Signal Name - Pin Assignment

Signal Name	TBGA Pin #	Die Pad #
SYSAD(26)	L03	51
SYSAD(27)	R23	294
SYSAD(28)	L02	53
SYSAD(29)	P22	295
SYSAD(30)	K02	50
SYSAD(31)	R22	287
SYSAD(32)	C01	22
SYSAD(33)	D23	348
SYSAD(34)	E01	33
SYSAD(35)	E23	345
SYSAD(36)	E02	28
SYSAD(37)	G22	340
SYSAD(38)	F02	29
SYSAD(39)	G23	335
SYSAD(40)	G03	27
SYSAD(41)	H23	328
SYSAD(42)	H01	44
SYSAD(43)	M22	307
SYSAD(44)	H02	39
SYSAD(45)	N22	304
SYSAD(46)	J01	52
SYSAD(47)	AB23	254
SYSAD(48)	R02	73
SYSAD(49)	AA22	253
SYSAD(50)	N03	63
SYSAD(51)	W23	273
SYSAD(52)	P01	70
SYSAD(53)	V22	271
SYSAD(54)	N01	69
SYSAD(55)	U23	283
SYSAD(56)	M02	61
SYSAD(57)	U22	275
SYSAD(58)	L01	54
SYSAD(59)	T22	282
SYSAD(60)	J03	38
SYSAD(61)	P23	297
SYSAD(62)	K03	43
SYSAD(63)	N23	305
SYSADC(0)	T01	79

TABLE 75. Signal Name - Pin Assignment

Signal Name	TBGA Pin #	Die Pad #
SYSADC(1)	AC22	235
SYSADC(2)	T02	81
SYSADC(3)	AD23	240
SYSADC(4)	T03	85
SYSADC(5)	AB22	241
SYSADC(6)	U01	82
SYSADC(7)	AA23	263
SYSCMD(0)	Y01	94
SYSCMD(1)	AA02	104
SYSCMD(2)	Y02	101
SYSCMD(3)	AB01	107
SYSCMD(4)	AB02	113
SYSCMD(5)	AC01	118
SYSCMD(6)	AC02	127
SYSCMD(7)	AD02	138
SYSCMD(8)	AD03	144
SYSCMDP	W01	90
TCK	A06	456
TDI	A03	473
TDO	C05	472
TEST_OUT_ENB	B14	419
TMS	C04	480
TRISTATEEN	D14	415
TRST	A05	460
VCLOCK	A12	435
VCOK	M21	308
VDDE	C02	10
VDDE	C14	417
VDDE	E05	1
VDDE	E06	487
VDDE	E08	475
VDDE	E09	464
VDDE	E10	453
VDDE	E12	432
VDDE	E15	402
VDDE	E16	391
VDDE	E17	380
VDDE	E19	369
VDDE	E20	367

TABLE 75. Signal Name - Pin Assignment

Signal Name	TBGA Pin #	Die Pad #
VDDE	F20	365
VDDE	F24	334
VDDE	G04	18
VDDE	G21	349
VDDE	H21	346
VDDE	K05	36
VDDE	M04	59
VDDE	M24	313
VDDE	R05	87
VDDE	T05	98
VDDE	V20	252
VDDE	W05	121
VDDE	W24	285
VDDE	Y05	123
VDDE	Y10	158
VDDE	Y12	179
VDDE	Y13	188
VDDE	Y15	209
VDDE	Y16	220
VDDE	Y19	243
VDDE	Y20	245
VDDE	Y24	277
VDDE	AA03	114
VDDE	AA08	143
VDDE	AA09	152
VDDE	AB24	266
VDDI	A08	448
VDDI	B03	479
VDDI	B09	447
VDDI	B16	408
VDDI	B18	398
VDDI	B19	395
VDDI	B21	385
VDDI	B22	376
VDDI	B23	371
VDDI	C08	458
VDDI	C12	429
VDDI	C20	383
VDDI	C22	363

TABLE 75. Signal Name - Pin Assignment

Signal Name	TBGA Pin #	Die Pad #
VDDI	D06	478
VDDI	D11	440
VDDI	E14	413
VDDI	F04	11
VDDI	F05	3
VDDI	H04	21
VDDI	H24	326
VDDI	J05	25
VDDI	J20	342
VDDI	J23	325
VDDI	K04	40
VDDI	K20	331
VDDI	L05	47
VDDI	N05	66
VDDI	N20	301
VDDI	P04	74
VDDI	P21	293
VDDI	R20	280
VDDI	T20	269
VDDI	T24	296
VDDI	U02	84
VDDI	U04	102
VDDI	U05	109
VDDI	V02	91
VDDI	V21	262
VDDI	W02	95
VDDI	W20	247
VDDI	Y03	106
VDDI	Y06	125
VDDI	Y08	136
VDDI	Y17	231
VDDI	Y21	250
VDDI	AA06	133
VDDI	AA13	186
VDDI	AA14	196
VDDI	AB03	119
VDDI	AB10	165
VDDI	AB18	218
VDDI	AC03	132

TABLE 75. Signal Name - Pin Assignment

Signal Name	TBGA Pin #	Die Pad #
VDDI	AC04	141
VDDI	AC11	175
VDDI	AC16	203
VDDI	AC17	206
VDDI	____	7
VDDI	____	26
VDDI	____	48
VDDI	____	65
VDDI	____	86
VDDI	____	108
VDDI	____	129
VDDI	____	148
VDDI	____	170
VDDI	____	187
VDDI	____	208
VDDI	____	230
VDDI	____	251
VDDI	____	270
VDDI	____	292
VDDI	____	309
VDDI	____	330
VDDI	____	352
VDDI	____	373
VDDI	____	392
VDDI	____	414
VDDI	____	431
VDDI	____	452
VDDI	____	474
VDD_SBUF(0)	E04	6
VDD_SBUF(1)	AA05	128
VDD_SBUF(2)	AB21	236
VDD_SBUF(3)	G20	359
VDD_VBUF(0)	D20	372
VDD_VBUF(1)	C15	409
VDD_VBUF(2)	C10	446
VDD_VBUF(3)	C19	386
VDD_VBUF(4)	D13	425
VDD_VBUF(5)	D08	468
VDD_VBUF(6)	C03	485

TABLE 75. Signal Name - Pin Assignment

Signal Name	TBGA Pin #	Die Pad #
VICEID(0)	AD09	174
VICEID(1)	AA11	171
VICERELEASE_N	AB07	149
VICESYSGNT_N	AB09	160
VICESYSRQST_N	AB08	153
VICEVALIDIN_N	AC07	154
VICEVALIDOUT_N	AC08	161
VICEWRRDY_N	AC09	164
VICE_INT_N	AD08	166
VSSE	A11	436
VSSE	B02	5
VSSE	B13	427
VSSE	D07	471
VSSE	D10	449
VSSE	D15	406
VSSE	D18	384
VSSE	D21	370
VSSE	E07	481
VSSE	E18	374
VSSE	E21	361
VSSE	F21	356
VSSE	G05	8
VSSE	K23	317
VSSE	N04	64
VSSE	U21	265
VSSE	AC05	150
VSSE	AC14	193
VSSE	____	2
VSSE	____	12
VSSE	____	13
VSSE	____	23
VSSE	____	24
VSSE	____	34
VSSE	____	35
VSSE	____	45
VSSE	____	46
VSSE	____	55
VSSE	____	56
VSSE	____	67

TABLE 75. Signal Name - Pin Assignment

Signal Name	TBGA Pin #	Die Pad #
VSSE	____	68
VSSE	____	77
VSSE	____	78
VSSE	____	88
VSSE	____	89
VSSE	____	99
VSSE	____	100
VSSE	____	110
VSSE	____	111
VSSE	____	122
VSSE	____	124
VSSE	____	134
VSSE	____	135
VSSE	____	145
VSSE	____	146
VSSE	____	156
VSSE	____	157
VSSE	____	167
VSSE	____	168
VSSE	____	177
VSSE	____	178
VSSE	____	189
VSSE	____	190
VSSE	____	199
VSSE	____	200
VSSE	____	210
VSSE	____	211
VSSE	____	221
VSSE	____	222
VSSE	____	232
VSSE	____	233
VSSE	____	244
VSSE	____	246
VSSE	____	256
VSSE	____	257
VSSE	____	267
VSSE	____	268
VSSE	____	278
VSSE	____	279

TABLE 75. Signal Name - Pin Assignment

Signal Name	TBGA Pin #	Die Pad #
VSSE	____	289
VSSE	____	290
VSSE	____	299
VSSE	____	300
VSSE	____	311
VSSE	____	312
VSSE	____	321
VSSE	____	322
VSSE	____	332
VSSE	____	333
VSSE	____	343
VSSE	____	344
VSSE	____	354
VSSE	____	355
VSSE	____	366
VSSE	____	368
VSSE	____	378
VSSE	____	379
VSSE	____	389
VSSE	____	390
VSSE	____	400
VSSE	____	401
VSSE	____	411
VSSE	____	412
VSSE	____	421
VSSE	____	422
VSSE	____	433
VSSE	____	434
VSSE	____	443
VSSE	____	444
VSSE	____	454
VSSE	____	455
VSSE	____	465
VSSE	____	466
VSSE	____	476
VSSE	____	477
VSSE	____	488
VSSI	A02	484
VSSI	A09	445

TABLE 75. Signal Name - Pin Assignment

Signal Name	TBGA Pin #	Die Pad #
VSSI	A14	420
VSSI	A16	418
VSSI	A17	410
VSSI	A19	407
VSSI	A20	399
VSSI	A22	388
VSSI	A23	382
VSSI	B01	16
VSSI	B05	467
VSSI	B06	461
VSSI	B10	439
VSSI	B11	437
VSSI	B12	428
VSSI	B15	416
VSSI	B24	362
VSSI	C07	462
VSSI	C09	451
VSSI	C13	426
VSSI	C17	397
VSSI	C21	375
VSSI	C24	351
VSSI	D04	4
VSSI	D05	483
VSSI	D09	459
VSSI	D16	396
VSSI	D19	377
VSSI	E13	423
VSSI	E24	338
VSSI	H05	14
VSSI	H20	353
VSSI	J04	30
VSSI	J21	337
VSSI	J24	323
VSSI	L04	49
VSSI	M05	57
VSSI	N21	303
VSSI	P05	76
VSSI	P20	291
VSSI	P24	298

TABLE 75. Signal Name - Pin Assignment

Signal Name	TBGA Pin #	Die Pad #
VSSI	R04	83
VSSI	R21	284
VSSI	T04	93
VSSI	T21	274
VSSI	U03	92
VSSI	U20	258
VSSI	U24	288
VSSI	V03	96
VSSI	V04	105
VSSI	V05	115
VSSI	W03	103
VSSI	W04	112
VSSI	W21	255
VSSI	Y04	117
VSSI	Y07	130
VSSI	Y09	147
VSSI	Y11	169
VSSI	AA04	126
VSSI	AA07	140
VSSI	AA10	162
VSSI	AA18	227
VSSI	AA20	239
VSSI	AA21	248
VSSI	AB04	131
VSSI	AB05	139
VSSI	AB06	142
VSSI	AB11	173
VSSI	AB13	185
VSSI	AB14	194
VSSI	AB16	207
VSSI	AC24	260
VSSI	AD16	201
VSSI	AD17	204
VSSI	AD20	216
VSSI	____	15
VSSI	____	37
VSSI	____	58
VSSI	____	75
VSSI	____	97

TABLE 75. Signal Name - Pin Assignment

Signal Name	TBGA Pin #	Die Pad #
VSSI	____	116
VSSI	____	120
VSSI	____	137
VSSI	____	159
VSSI	____	180
VSSI	____	197
VSSI	____	219
VSSI	____	238
VSSI	____	242
VSSI	____	259
VSSI	____	281
VSSI	____	302
VSSI	____	319
VSSI	____	341
VSSI	____	360
VSSI	____	364
VSSI	____	381
VSSI	____	403
VSSI	____	424
VSSI	____	441
VSSI	____	463
VSSI	____	482
VSSI	____	486
VSS_PKG_VSS	A01	___
VSS_PKG_VSS	A04	___
VSS_PKG_VSS	A07	___
VSS_PKG_VSS	A10	___
VSS_PKG_VSS	A13	___
VSS_PKG_VSS	A15	___
VSS_PKG_VSS	A18	___
VSS_PKG_VSS	A21	___
VSS_PKG_VSS	A24	___
VSS_PKG_VSS	D01	___
VSS_PKG_VSS	D24	___
VSS_PKG_VSS	G01	___
VSS_PKG_VSS	G24	___
VSS_PKG_VSS	K01	___
VSS_PKG_VSS	K24	___
VSS_PKG_VSS	M01	___

TABLE 75. Signal Name - Pin Assignment

Signal Name	TBGA Pin #	Die Pad #
VSS_PKG_VSS	N24	___
VSS_PKG_VSS	R01	___
VSS_PKG_VSS	R24	___
VSS_PKG_VSS	V01	___
VSS_PKG_VSS	V24	___
VSS_PKG_VSS	AA01	___
VSS_PKG_VSS	AA24	___
VSS_PKG_VSS	AD01	___
VSS_PKG_VSS	AD04	___
VSS_PKG_VSS	AD07	___
VSS_PKG_VSS	AD10	___
VSS_PKG_VSS	AD12	___
VSS_PKG_VSS	AD15	___
VSS_PKG_VSS	AD18	___
VSS_PKG_VSS	AD21	___
VSS_PKG_VSS	AD24	___
WRRDY_N	AC06	151

8.3 Test Modes

Describe JTAG mode here..

TABLE 76.

VICE Function				SysAD Flops	All Other Vice Flops

8.4 Schematic Icon

8.5 Physical Packaging Diagram

380 Lead Tab Ball Grid Array package from VLSI. Reference VLSI Drawing Number 25-45000 Rev: **.

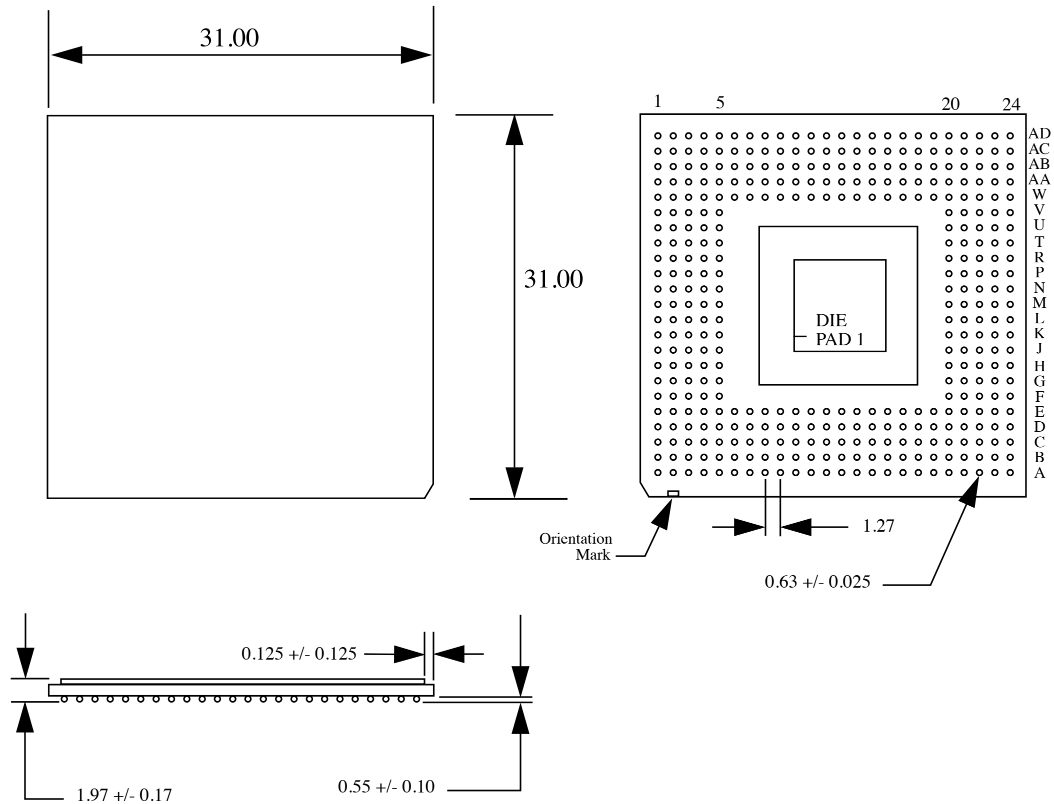


FIGURE 54. 380 Lead Tab Ball Grid Array

8.6 Physical Package Markings

Package Markings for the VICE chip

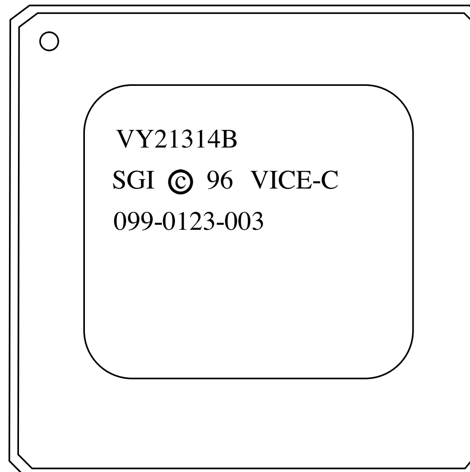


FIGURE 55. Package Markings

Package Markings above shown for Software ID "TRE" using hinv under Unix.

Alternate marking for earlier parts would be:
VY06762 (for VICE-A) 099-0123-001 (None shipped - Not at speed part - proto only)
VY21314- (for VICE-B) 099-0123-002 (Software ID "DX" using hinv under Unix

8.7 Bonding Diagram

Refer to Table 75, "Signal Name - Pin Assignment," on page 217

9.0 Device Characteristics

9.1 Absolute Maximum Ratings

Stresses above those listed may cause permanent damage to the device. These are stress rating only, and functional operation of this device under these or any conditions above those indicated in this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

TABLE 77. Absolute Maximum Ratings - Non Operational

Condition	Allowable Range
Ambient Operating Temperature with voltage applied (junction temperature not to exceed ambient temperature by more than 30 degrees C)	-40 ^{o}C to +85 ^{o}C
Storage Temperature	-40 ^{o}C to +125 ^{o}C
Supply Voltage	-0.5 to +4.0 V
Input Voltage (Any Pin)	-0.5 to +3.9 V
DC Input Current	+/- 20 mA
Lead Temperature	250 ^{o}C

9.2 Operating Range

TABLE 78. Device Operating Range

Condition	Allowable Range
Ambient Operating Temperature with voltage applied (junction temperature not to exceed 110°C which limits internal temperature rise to 60°C)	0 ^{o}C to +50 ^{o}C
Junction Temperature Simulated	110 ^{o}C
VDD Supply Voltage VSS = (0.0) Note: Supply Voltage anticipated to be 3.3V +/1 5% Chip simulated to 3.135V as minimum voltage. For timing and power calculations, chip can run as high as 3.6 V.	3.135 V to 3.6 V

9.3 DC Characteristics and Capacitance

TABLE 79. DC Characteristics (T_j = 0 to 110° C)

Parameter	Description	Conditions	Min	Max	Units
V_{IL}	Low Input Voltage	V_{DD} = 2.7V	-0.33	0.81	V
V_{IL}	Low Input Voltage	V_{DD} = 3.6V	-0.33	1.08	V
V_{IH}	High Input Voltage	V_{DD} = 2.7V	1.89	3.03	V
V_{IH}	High Input Voltage	V_{DD} = 3.6V	2.52	3.93V	V
V_{OL}	Low Output Voltage	V_{DD} = 2.7V I_O =2 to 4 mA	----	0.4	V
V_{OH}	High Output Voltage	V_{DD} = 2.7V I_O = 2 to 4 mA	2.4	----	V
I_{IH}, I_{IL}	Input Leakage Current	V_I = V_{DD}, V_{SS}	-1	+1	uA
I_{OZH}, I_{OZL}	Output Leakage Current	V_O = V_{DD}, V_{SS}	-1	+1	uA
C_{OUT}	Output Pin Capacitance	f = 1 MHz V_{DD} = 0V	----	15	pF
Power	Power Dissipation	VDD = 3.3V	4 -Typical	5	W

9.4 AC Characteristics

AC Characterization performed at worst case low voltage 3.135V, Junction temperature of 110 °C, and load capacitance on SysAD bus interconnect of 80pF.

TABLE 80. AC Characteristics

Parameter	Description	Min	Max	UNITS
SClock-T_{per}	Period Rate		100	MHz
SClock-T_{cy}	Cycle Time	10		ns
SClock-T_{chi}	SClock High Time	4	6	ns
SClock-T_{clo}	SClock Low Time	4	6	ns
VClock-T_{per}	Period Rate		60	MHz
VClock-T_{cy}	Cycle Time	17		ns
VClock-T_{chi}	VICE Clock High Time	7	10	ns
VClock-T_{clo}	VICE Clock Low Time	7	109	ns

9.4.1 PLL Characteristics

Total Jitter < 1 nsec. Capture and Lock 1KHz to 200 MHz.

9.5 Package Thermal Characteristics

The VICE die when does not need a heatsink if the 4.5 W power number is met and an airflow of 200 ft/min can be achieved in the system. It will need a heatsink if the power consumption is higher or if the air flow is lower.

For selecting a package for the VICE die, Table 81, "Package Thermal Resistance Characteristics - Air Flow Consideration," on page 236 can be used. Unitl we have final numbers for a 526 die in a 380 - 31mm package, two different package and die size combinations are listed. Junction Temperature Calculation is as follows:

Tj = Ta + (Theta Ja x Ptotal)

Where
Tj = Junction Temperature measured in ^{o}C
Ta = Ambient Temperature measured in ^{o}C
Theta Ja = Package Thermal Resistance measured in ^{o}C/Watt
Ptotal = Total Typical Power measured in Watts

For the Vice chip then the following values can be used in the formula to find Theta Ja.

Tj = 110 ^{o}C (Chip simulated to operate at this junction temperature)
Ta = 50 ^{o}C (40 ^{o}C temp for the moosehead box w/ 10 ^{o}C rise inside the box)
Ptotal = 4.5 W (Calculated Typical Power Dissipation)

Theta Ja = (Tj - Ta) / Ptotal = (110 - 50) / 4.5 = 13 ^{o}C/W.

For Ptotal = 8 W (Estimate worst case)

Theta Ja = (110 - 50) / 8 = 7.5 ^{o}C/W.

TABLE 81. Package Thermal Resistance Characteristics - Air Flow Consideration

Air Flow (Ft/Min)	**380 Pkg Pads 31mm Body 386mil die Mulit Layer Bd No Heatsink Theta Ja (^{o}C/W)**	**552 Pkg Pads 37mm Body 526 mil die Multi Layer Bd No Heatsink Theta Ja (^{o}C/W)**	**380 Pkg Pads 31mm Body 386mil die Mulit Layer Bd 38mm x 38mm x 7.2mm Heatsink Theta Ja (^{o}C/W)**	**552 Pkg Pads 37mm Body 526 mil die Multi Layer Bd 38mm x 38mm x 7.2mm Heatsink Theta Ja (^{o}C/W)**
Still Air	14.5	11		10
100 Top				7.5
200 Top	12	10.3		6.5
200 Top & Bottom		9		

10.0 Bugs

This section originally was to contain only bugs found in the actual chip. It has been expanded to include differences between the Software Simulation environment and the actual chip.

10.1 Software Simulator vs. Silicon Behavior

10.1.1 MSP_D_EN Register

Must be programmed to allow MSP access to each of the three banks of Data RAM inside of Vice. For performance enhancement, if the MSP is NOT going to access a particular bank of Data RAM, that bit should be set to a logical ‘0’ so as to allow the DMA, the Host or the BSP a higher bandwidth access to that portion of memory.

10.1.2 MSP_CAUSE Register

Because load/store and Contention exceptions occur in the write back stage of the MSP while other exceptions occur during the decode stage, the load/store and Contention exceptions followed immediately by the other type of extensions (for example a Break), will produce an Exception Code of the Breakpoint to appear in the MSP_Cause register rather than the load/store or contention exception.

The MSP_ExcpFlag register will correctly show both exceptions as having occurred.

10.2 Silicon Bugs

Known bugs in the Vice 1 silicon. (VLSI part # vy06762)

10.2.1 Leading Zero Bug

Collection of one bug (A) and one request for feature enhancement (B). Feature enhancement requests that the BSP process be able to reset it's write buffer.

10.2.1.1 Application

JPEG encode

10.2.1.2 Symptom

A- Leading "00" at start of bit stream for a field/frame when the previous field/frame ended on an "FF" code.

B-Encoded bit stream gets "stuck" in the BSP write buffer and does not show up till the next frame is encoded.

10.2.1.3 Cause

A-The byte stuffer in the BSP remembers state of detecting an "FF" code as the last byte of a field/frame when encoding. When the next bits go into the (write buffer? encode pipe?) the byte stuffer puts in the "00" when it should not.

B- DMA only empties the BSP write buffer when it is full. If the encoded bit stream is not a multiple of 8 bytes, the last bytes are not transferred by the DMA engine.

10.2.1.4 Work-around

A- BSP software stuffing extra bytes into the encode pipe to "flush" the state of the byte insertion hardware.

B- BSP software has to monitor write buffer count and pad correctly to make bit stream a multiple of 8 bytes.

10.2.1.5 Fix for Spin

A- Hardware fix to automatically reset the byte stuffer at end of picture?

B- Reset of write buffer accessible to BSP Software?

10.2.2 Rocky bad block

Not fully characterized as of 3-20-96.

10.2.2.1 Application

JPEG encode

10.2.2.2 Symptom

Particular Macro Block with specific bit offset pattern causes a bad block to be encoded.

10.2.2.3 Cause

10.2.2.4 Work-around

10.2.2.5 Fix for Spin

10.2.3 Low Quant - Low Compression

Not fully characterized as of 3-20-96.

10.2.3.1 Application

JPEG encode

10.2.3.2 Symptom

High Bit Rate causes image to be un-recognizable

10.2.3.3 Cause

10.2.3.4 Work-around

10.2.3.5 Fix for Spin

10.2.4 Skier Sparkle

8 Second Movie compressed with Vice one field at a time and played back real-time with Cosmo Compress.

10.2.4.1 Application

JPEG encode

10.2.4.2 Symptom

A “sparkle” observed in the picture during playback that looks like it may be an all white block of some sort.

10.2.4.3 Cause

To: vice@sgi.com

the “sparkle” appears in the original skiing image (frames 208 and 209).

no bugs here.

Chuck Tufflituffli@sgi.com

10.2.4.4 Work-around

10.2.4.5 Fix for Spin

10.2.5 Decode

Not fully characterized as of 3-20-96.

10.2.5.1 Application

JPEG decode

10.2.5.2 Symptom

High Bit Rate causes lockup.

10.2.5.3 Cause

10.2.5.4 Work-around

10.2.5.5 Fix for Spin

10.2.6 MPEG hang

Decode of an MPEG image causes application to hang.

10.2.6.1 Application

MPEG decode

10.2.6.2 Symptom

BSP appears to be in a loop waiting for the DMA engine.

10.2.6.3 Cause

10.2.6.4 Work-around

10.2.6.5 Fix for Spin

10.2.7 VSUM2

Not fully understood as of 3-20-96.

10.2.7.1 Application

vd/regress.csh

10.2.7.2 Symptom

vsum2 test in the regress.csh test fails in the silicon.

vsum2 test in the regress.csh appears to work in the vhdl?

10.2.7.3 Cause

10.2.7.4 Work-around

10.2.7.5 Fix for Spin

10.2.8 BSP Halt Ack

Decode of an MPEG image causes application to hang.

10.2.8.1 Application

Vice Debugger (vd)

10.2.8.2 Symptom

BSP does not raise it's Halt Acknowledge bit when the BSP is reset by its software reset bit.

10.2.8.3 Cause

Not part of hardware defined behavior. This functionality does not exist in the present silicon.

10.2.8.4 Work-around

Vice Debugger does not depend on this signal for now.

10.2.8.5 Fix for Spin

Halt Acknowledge will be set when the BSP is reset by its software reset bit.

10.2.9 MSSM Reset

Multi State State Machine is not reset by the BSP software reset bit.

10.2.9.1 Application

Various hang conditions when running JPEG, MPEG

10.2.9.2 Symptom

Multi-State State Machine is not reset by the BSP software reset bit.

10.2.9.3 Cause

MSSM reset was not included in the BSP software reset in the hardware.

10.2.9.4 Work-around

None. Reset button on workstation is only fix? What about a warm boot as this will reset the Vice chip?

10.2.9.5 Fix for Spin

Yes, MSSM will be reset by the BSP software reset bit.

10.2.10 MSP PC Pins

MSP PC output pins of Vice.

10.2.10.1 Application

Logic Analyzer

10.2.10.2 Symptom

MSP PC does not appear to change on the logic analyzer when it should.

10.2.10.3 Cause

The MSP PC output pins are actually driven by the Exception PC so the pins only update when an MSP exception occurs.

10.2.10.4 Work-around

Use the vice debugger and read back the MSP PC from inside the chip.

10.2.10.5 Fix for Spin

Yes. MSP PC will be connected to the MSP PC output pins. Expect a register delay (or two) from what the actual MSP PC is pointed at when the MSP is executing.

10.2.11 Vice-Crime Handshake Pins.

Vice-Crime Handshake pins are 3-state during reset.

10.2.11.1 Application

Cold Reset/Warm Reset.

10.2.11.2 Symptom

Workstation does not always get through the boot prom after reset.

10.2.11.3 Cause

Petty Crime expects these signals to be at know logic levels (de-asserted) during reset. The proto cpu boards all had pullups on these signals since the Petty Crime and CPU needed to run without Vice installed for some number of months. This is a bug in Vice.

10.2.11.4 Work-around

Leave pullups on all the proto-boards with Vice1 installed.

10.2.11.5 Fix for Spin

Hardware will be modified to provide a logical ‘1’ on the following pins per Section 2.2.1 of this document.ViceValidOut_n, ViceSysRqst_n, ViceRelease_n and ViceInt_n.

An additional point on reset. ViceRelease_n activates for once cycle (logical ‘0’) reset is removed from the Vice chip (chip just coming out of reset) to ensure that the Crime arbiter is reset cleanly. The Vice 1 chip already performs this action.

NOTE: This was not fixed in the spin and still occurs with the latest VICE parts 099-0123-003 vy21314B “VICE-C” Unix hinv report “TRE” (I think that is all the names that it is know by!)

11.0 Revision History

4-17-97 All

Created Revision 1.0

Long list of changes. Just look for the change bars in the margins.

4-5-95 All

Created Revision 0.20

vice.title - D. Barnett, S. Klinger Added
ch1 - System Block Diagram Updated, Logical pin diagram updated pwr/gnd/jtag, spelling fixes
ch2_addr - BSP Table Memory expanded 0x5000 - 0x6FFC and 22 bits wide, BSP FIFO now at 0x7000, TLB moved to 0xF000 on common bus.
ch2_init - ViceReset_n same net as MIPS Reset*. Not independent, not enough pins on CRIME.
ch2_interrupt - No Changes
ch2_msp_manage - Instruction RAM dual ported now. Can DMA update while MSP is executing.
ch2_bsp_manage - No Changes.
ch2_dma_manage - No Changes except change bars removed from last version.
ch2_reg - Look for change bars, lots of little stuff. Each DMA channel now has their own interrupts, Interrupts are now reset by a write only register called VICE_INT_RESET. DMA Flush Buffer Mode added, Chroma only DMA modes specified but may not be required, will test last! Lots of updates to BSP registers since Jan 18 spec.
ch3_sys_intfc - Figure with SysAD Clock Distribution and 66 MHz internal clock source, No ref to DSS proc.
ch4_arch_4.1_4.3 - Clarification to DMA Descriptors. Clarification of SysAD Vice Write bursts. Look at change bars.
ch4_arch_sec4.4.1 - MSP Instruction RAM now dual port
ch4_arch_sec4.4.2 - No Change
ch4_arch_sec4.4.3 - No Change
ch4_arch_4.4.4_su - No changes. Change bars from previous revisions removed.
ch4_arch_4.4.5_vu - Major update of instructions and co-ordination with Appendix A Vector Unit Instruction Details.
ch4_arch_4.6 - BSP programming restrictions updated.
ch5_oper_descr - No Change
ch6_perf - No Changes
ch7 - No Changes
ch8_dev_intfc - JTAG pins added, package changed to 380 TBGA.
ch9_dev_char - Preliminary TBGA thermal characteristics added.
Appendix A - Major updates to Vector Unit Instruction Set Details
Appendix B - Vector Unit Block Diagrams included
Appendix D - Test Plan Added

1-18-95 Te-Li L., Robb P.

Created Revision 0.12

Added Appendix A - Vector Unit Instruction Set Details Release
Updates to Vector Unit Section
- Instruction names have been changed to reflect new naming convenctions.
- Many new instructions have been added.
- Some unused instructions have been deleted.

- All instructions have been updated to reflect new modes of operation.

Updates to Scalar Unit Section

11-23-94 Te-Li L., Michael F., Robb P., Mark T., Bent H.

Created Revision 0.11 ready for limited distribution and review!

Post Review changes for Vector Unit and Bitstream Processor
Added Bitstream Processor Appendix B Instruction Set Details
Added Section to Ch2 for MSP Code Management
Revised Pin Description in Ch3, 4, 8 Vice shares Processor Reset_n now.
Ch4 Common Bus and DMA Bus protocols revised. All cycles now 3 clocks rq/gnt, addr, data

11-13-94 All

Post Spec Review changes marked with change bars as shown at the left (for most changes). Vector Unit and Bitstream Processor sections not yet revised.

10-13-94 Te-Li L., Michael F., Robb P., Mark T. - Results of Detailed Spec Review Ch1 & Ch2
and preparation for Vector Unit and Bitstream Processor reviews.

vice.title
- Revision 0.9, Acknowledgments deleted.

vice.book
- Moved registers in chapter 2 to the end of the chapter.
- Added new chapter for BSP management.

ch1
- Updated feature list, block diagram. Cleaned up errors.

ch2_addr
- All new address map

ch2_reg
- Added MSP_Watchpoint register, MSP_PC is now read/write. MSP_DEBUG now MSP_CTL_STAT
- Added BSP; IN_BOX, OUT_BOX, CTL_STAT, Watchpoint registers
- Added HST_BSO_IN/OUT_BOX shadow registers
- Renumbered register addresses

ch2_init
- Clarified VICE chip and internal block resets.

ch2_interrupt
- DMA and Buss Error exceptions removed from MSP exception processing.
- Vector Unit Reserved Instruction Exception added
- Watchpoint exception ignores 3 lsb address bits.

ch2_bsp_manage
- New chapter added

ch4_arch_sec4.1_4.3
- Added “DMA” bus to ch4 Architecture Section

ch4_arch_sec4.4.5_vu
- Update instruction names and formats
- Additional details for multiply instructions
- Add two new instructions Add With Accumulate and Subtract With Accumulate
- Clean up of interlock section
- Added block diagram
- Added saturation status register - VCLR

- Add Zero and Xtend Load/Store instructions
- General editing for better readability and improved clarity

ch4_arch_sec4.6
- Bit Stream Processor Instruction Set Updated, Logical NOR gone
- Mail Box Registers Defined
- Figures Numbered

9-30-94 All Version 0.8

vice.title
- Revision 0.8

vice.book
- Deleted ch4_arch_sec4.7 which was DMA. Has been in ch4_arch_sec4.1_4.3 all along.

ch1
- Updated DMA function list in overview section

ch2_addr
- New System Address 0x0 1700 0000 was 0xF 0000 0000.
- Also MPS processor registers no longer accessible

ch2_reg
- Deleted MSP_PCTrace Reg Changed MSP_BadVAddr to MSP_BadAddr
- Deleted MSP_Breakpoint Reg Added MSP_EPC Added MSP_ExcpFlag Reg
- Removed Single-Step and Breakpoint in MSP_Debug Register
- Added more documentation for each register Changed MSP_PC to be write only
- DMA registers added.
- BSP registers added.

ch2_interrupt
- Redefined Breakpoint Exception
- Added more documentation about EPC being updated for certain exceptions
- Added definition for Reserved Instruction Exception
- Added more overall documentation for each exception

ch2_msp_manage
- Removed Single-Step and Breakpoint Mode
- Added documentation on sequence of events for a Software Break Instruction.

ch3_sysintfc
- Vice address updated

ch4_arch_sec4.1_4.3
- New DMA functions. Internal Bus Protocol for VICE updated. 3 buses now.

ch4_arch_sec4.4.4_su
- Added in interlocks & stalls for 2 cycle load delay slot
- Added in documentation for X-Bar
- Added in documentation for load/store mechanism
- Added in documentation for pipeline timing of branches/jumpssu
- Added in doc on what to check during 2 cycle load delay slots.
- Added RD_immediate to things to check for when check RT for load dly
- Added BREAK instruction as one of the supported instructions.

ch4_arch_sec4.4.5_vu
- Fix some typos

ch4_arch_sec4.6
- Bit Stream Processor Instruction Set Updated

ch6_perf
- Application performance timelines added.

ch9
- Deleted reference to I2C bus timing. Left over from A/V I/O.

vice_bib.doc
- Corrected format where two references were joined in one paragraph.

8-17-94 M.T., T.L. Version 0.7

Chapter 2 - Explicit space for MSP Data RAM banks A,B,C. Updated address map. Interrupt and intitialization sections update.

Chapter 3 - Revised SysAD interface definition. No R4K burst access to VICE. VICE can only access system memory. Interrupt pin back on VICE.

Chapter 4 Sections 4.1-4.3 - Host Interface block diagram added. Internal VICE bus clarified First pass at bus arbiter added.

7-19-94 M.T. Version 0.6.

Chapter 1 - Latest block diagrams. Overview of major blocks in VICE. Deleted references to Video I/O.

Chapter 2 - Updated Address Map, Moved dummy register to location 0 for safety. Removed all references to Video and Audio I/O and I2C bus. Added VICE exception register, per Bents previous document. DMA registers first pass.

Chapter 3 - Added SysAD timing diagrams and description of transfers that VICE can be involved in. Added Logical diagram of VICE/CRIME/R4K interconnect.

Chapter 4 - Deleted Video I/O unit

Chapter 8 - Updated Logical Diagram and Pin Descriptions

5-25-94 R.P., M.T. Version 0.5. Vector Unit Special Reg Highlights added, Overview Instr Groups. VLC, DMA templates created.

5-12-94 R.P. Updated Vector Unit Architecture and Appendix A, Vector Unit Instruction Set.

4-15-94 S.A. Updated Section 4.4.4 Scalar Unit Architecture. New internal block diagram, new instruction execution pipeline description, memory block diagram, memory control block diagram, instructions supported updated

3-14-94 M.T. Updated Section 4.5 Video I/O. Added additional pixel formats. Removed Old VINO specific items in the filter section.

3-8-94 M.T. Called Version 0.2 on Title Page. Updated Moosehead System Block Diagram to show latest chip partitioning per Mike Nielsen. Modified formatting of additions to Ch1 and Ch4 to match format of overall document.

3-6-94 S.A. Added Scalar Unit Sections 4.4.4 and updates to 4.4.1 MSP Overview and Ch1 updates with Moosehead System Block Diagram.

3-4-94 R.P Added Appendix A, Vector Unit Instruction Set.

3-2-94 M.T. Added Video DMA registers, programming section and block diagram in architecture section.

2-10-94 First release of overall document structure. Many VINO like features borrowed from the VINO specification (Thank-you Chris and Linda).

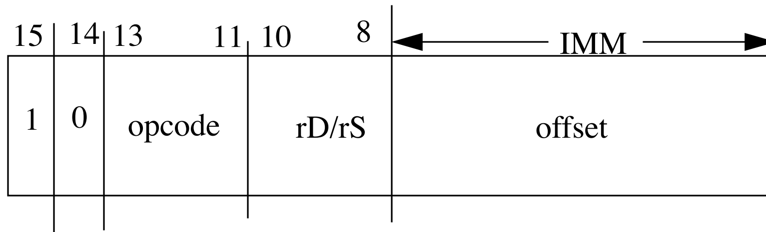
Appendix B:

Vector Unit Block Diagrams

Appendix C

Bitstream Processor Instruction Set Details:

Format A:



0) LoadH /* load halfword: rD <- MEM[rPage + offset<<1] */
1) LoadBl /* load byte low: rDl <- MEM[rPage + Offset] */
2) LoadBh /* load byte high: rDh <-MEM[rPage + Offset] */
3) StoreH /* store halfword: MEM[rPage + Offset<<1] <- rS */
4) StoreBl /* store Byte low: MEM[rPage +offset] <- rSl */
5) StoreBh /* store Byte high: MEM[rPage +offset] <- rSh*/
6) LDI_l /* load immediate: rDl <- imm */
7) LDI_h /* load immediate: rDh <- imm */

Format A instruction Syntax:

OPCODE {rD,rS} , {LABEL, IMMEDIATE}

OPCODE= { LoadH, LoadBl, LoadBh, StoreH, StoreBl,
StoreBh, LDI_l, LDI_h }

D = { 0,1,2,3,4,5,6,7}
S = { 0,1,2,3,4,5,6,7 }

LABEL = { .... } /* any permissable text */

IMM = 8-bit unsigned number.

Note: the LABEL must be within [0, 255] bytes/halfwords of the rPage register.

If LABEL is not in the above specified range then the rPage register must be loaded to contain the address of the page where the LABEL can be found.

FORMAT A Instruction Execution Sequence:

INSTRUCTION FETCH:

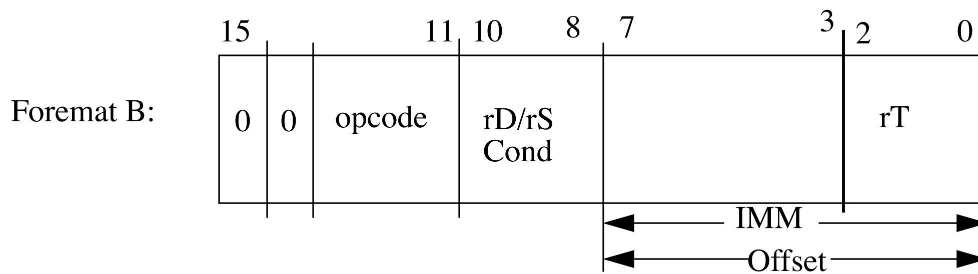
IAB <- PC ; PC <- PC + 2

INSTRUCTION DECODE:

ADDREG <- rPage + Offset (<<1)

INSTRUCTION EXECUTE:

```
if (load_from_memory || store){

     AB<- ADDREG;

     if (load) { rD <- DB }

     if (store) { DB <- rD}

}else { /* load immediate */

     rD(h,l) <- IMM
}
```



0) NOP /* must have this */
1) CMPI /* compare rSl with IMM(8-bit unsigned)
2) ANDI /* and rSl with IMM(8-bit unsigned)
3) BRANCH /*BRANCH cond, pc + Offset (signed)*/
4) BREAK
5) ADDI /* add unsigned 8-bit immediate to rS */
6) JR /* jump cond to rT */
7) RESUME /* resume multi-cycle instruction */

Cond = eq (Z) , ne , ge, lt, ext0, ext1, ext2, TRUE

Format B Instruction Syntax:

{
OPCODE1 {rS} , {IMM}

OPCODE2 {COND}, LABEL1

OPCODE3 { COND}

OPCODE4 { rT}

OPCODE5
}

OPCODE1 = { CMPI, ANDI }
S = {0,1,2,3,4,5,6,7}
IMM = { 0,1,2,...255 }

OPCODE2 = { BRANCH, CALL }
COND = { EQ, NE, GE, LT, EXT0, EXT1, EXT2, EXT3 }
LABEL1 =

OPCODE3 = RET
COND = { EQ,NE,GE,LT,EXT0,EXT1,EXT2,TRUE }

OPCODE4 = JR
T={0,1,2,3,4,5,6,7}

OPCODE5 = { RESUME, NOP }

FORMAT B Instruction Execution Sequence:

CMPI, ANDI

INSTRUCTION FETCH:

IAB <- PC ; PC <- PC + 2

INSTRUCTION DECODE:

INSTRUCTION EXECUTE:

if (CMPI) { alu: rD - IMM ; status <- status(cmpi) }

if (ANDI) { alu: rDl & IMM ; status ,_ status(andi) }

FORMAT B Instruction Execution Sequence:

BRANCH cond, Label

INSTRUCTION FETCH:

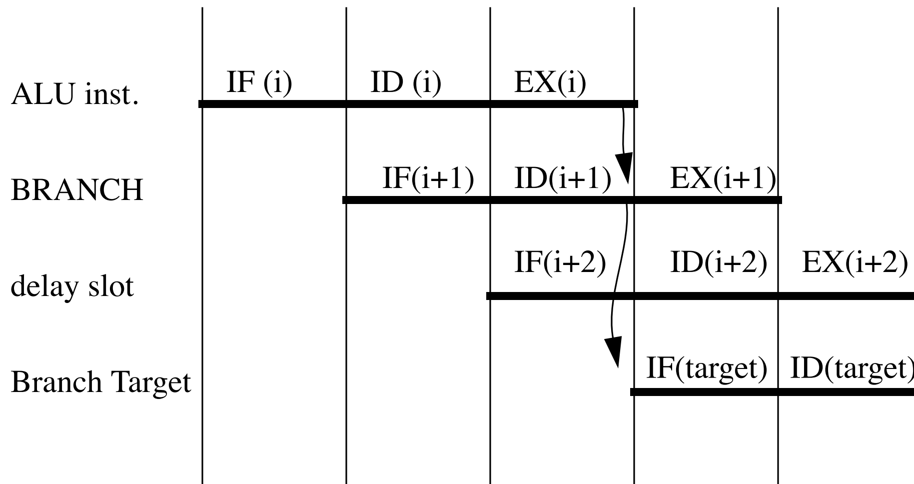
IAB <- PC ; PC <- PC + 2; IR <- IMEM[IAB]

INSTRUCTION DECODE:

PC <- PC + IMM /* NB. target is pc+ 2 + IMM.

INSTRUCTION EXECUTE:

if (cond) {
    IAB <- PC; IR<- IMEM[IAB]
else {
    NOP
}



NB. Branch has 1 delay slot.

FORMAT B Instruction Execution Sequence:

JR cond, Label

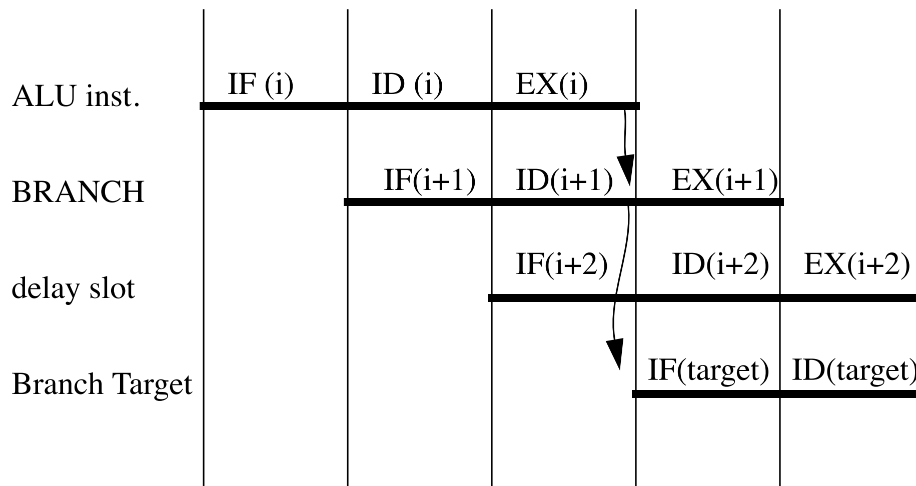
INSTRUCTION FETCH:

IAB <- PC ; PC <- PC + 2; IR <- IMEM[IAB]

INSTRUCTION DECODE:

INSTRUCTION EXECUTE:

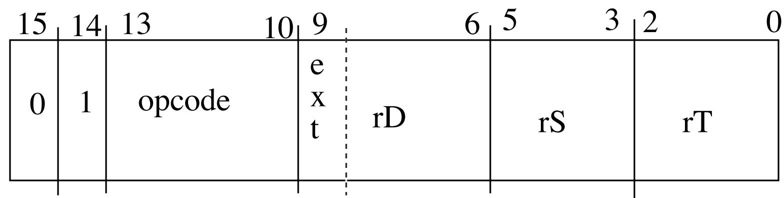
```
if (cond) {
     IAB <- rT ; IR <- MEM[IAB]
else {
     NOP
}
```



NB. JR has 1 delay slot.

NB. if accessing register file to provide address is a critical path , can put register access in ID stage.

Format C:



Arithmetic
0) ADD /* triadic add */
0) ADDC /* add with carry (ext = 1)*/
1) SUB /* triadic subtract */
2) CMP /* dyadic compare */
3) AND /* bitwise, triadic logical AND (N,Z flag)*/
4) OR /* bitwise, triadic logical OR (Z flag) */
5) LSHIFT /* left shift zero fill rD = rS << rT (rT 0-15)*/
6) ARSHIFT /* right arithmetic shift. rD = rS >> 1*/
7) MULT /* rD = rS*rT(15:0) ; cmp_h = rS * rT(31:16)
8)ZZXLATE /* rD <- XLATE rS */
9) XOR /* bitwise, triadic logical exclusive or (N,Z Flag) */
10) ABS /* absolute value N set if operand is negative (N,Z Flags)*/
11) NEG /* negate the value (N,Z Flags) */
Register Copy:
12) COPYTO /* Copy GPR to any Alternate register */
13) COPYFROM /* Copy any Alternate register to GPR */
Load/Store:
14) LLOADH /* rD -< MEM[rS] */
15) LSTOREH /* MEM[rS] <- rT */

Alternate registers:
rPage
alpha_h, alpha_l
beta_h, beta_l
cmp_h, cmp_l
mask_h, mask_l
root_tbl_ptr, tbl_res
block_data, block_adress, block_ptr, acc_run
status_cntl

Format C Instruction Syntax:

OPCODE6 rD,rS,rT

OPCODE7 rS,rT

OPCODE8 rD,rT

OPCODE9 rEXTEND, rS

OPCODE10 rD, rEXTEND

OPCODE11 rD, rS

OPCODE6 = {ADD, SUB, AND ,OR ,NOR, XOR}
D= { 0,1,2,3,4,5,6,7}
S= { 0,1,2,3,4,5,6,7}
T= {0,1,2,3,4,5,6,7}

OPCODE7 = { CMP , LSTOREH}

OPCODE8 = { ABS, NEG }

OPCODE9 = {COPYTO}
EXTEND = { rpage, alpha_h, alpha_l, beta_h, beta_l, CMP_h, CMP_l, MASK_h, MASK_l
root_tbl_ptr, tbl_res, block_data, block_address, block_ptr, ac_run, status_cntl }

OPCODE10 = { COPYFROM }

OPCODE11 = {LLOADH}

FORMATC Instruction Execution Sequence:

Arithmetic Instructions

INSTRUCTION FETCH:

IAB <- PC ; PC <- PC + 2

INSTRUCTION DECODE:

INSTRUCTION EXECUTE:

rD = rS op Rt

FORMATC Instruction Execution Sequence:

Copy Instructions

INSTRUCTION FETCH:

IAB <- PC ; PC <- PC + 2

INSTRUCTION DECODE:

INSTRUCTION EXECUTE:

```
if (COPYTO){
    db <- gpr; rDextended <- db
} else { /* COPYFROM */
    db <- rDextended; gpr <- db

}
```

FORMATC Instruction Execution Sequence:

Load/Store Instructions

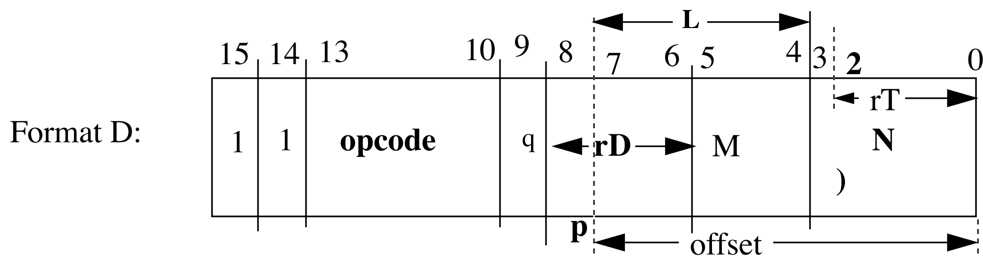
INSTRUCTION FETCH:

IAB <- PC ; PC <- PC + 2

INSTRUCTION DECODE:

INSTRUCTION EXECUTE:

```
if (LLOADH){
    ab <- rS;                              NB. This may be timing critical
    db <- MEM[ab] ; rD <- db
} else { /* LSTOREH */
    ab <- rS/
    db <- gpr ; MEM[ab] <- db
```



Single Cytle Instructions:
0) getBits(q) rD,N /* get N bits from bitstream put in rD right justified with zero fill
q=1 enables byte swallowing*/
1) probeBits rD,N /* probe N bits in the bitstream. */
2) ShiftStream(N,q) /* shift the bitstream left N bits
q=1 enables byte swallowing*/
3) getBits(q) rD, rT /* get (rT +1)bits from bitstream (use bits 3-0 of rT) . q=1 enables byte swallowing */

4)generic_lookup_pack rT

5) leaf_run_level_parse(q) /* perform run-level lookup until hit a leaf node in search tree - Appropriate for H.261 and MPEG 1& 2 */

6) block_run_level_parse(q) /* perform a run-level lookup until hit the end of a block Appropriate for H.261 and MPEG 1& 2 */

7) load_code_pack_H261(q,p) offset

8) generic_leaf_parse. /* perform generic table lookup putting the signe extended 11-bit value at the leaf node in the tbl_res register. */

9) block_run_size_parse(q) /* perform JPEG run/level parse of a block of data. Compute level from size information and the bitstream. This instruction is JPEG specific.

A)code_search(q,p) . /* shift the bitstream left one bit at a time until hit a start code (programmed in the CMP and mask registers). q = 1 => byte alignement. p= 1 => byte insertion.

C) pack_bitstream(q) L, rT. /* pack L left justified bits in rT into the bitstream. q enables JPEG-style byte insertion */

D) load_code_pack(q,p) offset /* coding pipe <- MEM[rPage + Offset]. q enables zero-byte insertion p = AC/DC coefficients*/

E) **byte_align** /* make the bistream buffer byte aligned. FIll is from CMP_H register (msbs)

NB. when performing byte swallowing, non-zero values are not swallowed.

Format D Instruction Syntax

OPCODED(rD,N)

OPCODED1(N)

OPCODED2(Q)

OPCODED3

OPCODED = { getbits , probebits }
D = {0,1,2,3,4,5,6,7}
N = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}

OPCODED1 = { shiftstream }

OPCODED2 = { leaf_run_level_parse, block_run_level_parse}
Q = { 0,1 }

OPCODED3 = { generic_leaf_parse, block_run_length, start_code_search }

FORMAT D Instruction Execution Sequence:

Single Cycle Instructions

INSTRUCTION FETCH:

IAB <- PC ; PC <- PC + 2

INSTRUCTION DECODE:

INSTRUCTION EXECUTE:

```
if (getBits(N) ){
    db <- alpha_h[15:0] ; rD <- db[32 - (32-N) ]; alpha_h.alpha_l.beta_h.beta_l << N ;
} else { if(probeBits(N)){
        db <- alpha_h[15:0] ; rD <- db[32 - (32-N) ];
} else { if( ShiftStream(N)){
        alpha_h.alpha_l.beta_h.beta_l << N ;
}
```

FORMAT D Instruction Execution Sequence:

Multi Cycle Instructions

INSTRUCTION FETCH:

IAB <- PC ; PC <- PC + 2

INSTRUCTION DECODE:

INSTRUCTION EXECUTE:

Pipeline stalled in execute cycle until multi-cycle instructions have completed execution.

Appendix D: Test Plan

D.1 Functional verification

This section describes VHDL simulation environment and lists diagnostics for VICE. For VHDL simulation, the preferred environment is at the board level with R4K as the host CPU and CRIME chip as the interface to main memory. And the ultimate system level simulation would start with all codes residing in main memory, R4K text would then DMA MSP and BSP instructions and data into the appropriate instruction rams and data ram within VICE and off it goes.

VICE simulation requires one to three kinds of assembly codes (in addition to compiled code if any). These are : MSP code, BSP code and R4K code that need to be assembled separately using different assemblers. For each diagnostic, a makefile is used to encapsulate the steps of text generation : assemble which ".s" file with which assembler.

The board level simulation is the most comprehensive test. However, it may not be practical for the debug stage of MSP and BSP. The proposed mechanism for facilitating both simulation needs is to allow preload of MSP instruction ram, BSP instruction ram and data rams from three separate files along with separate file for main memory. When board level simulation is desired, put all text in main memory and leave the files for MSP text, BSP text and data ram empty. It would leave instruction rams and data rams uninitialized. For individual processor (MPS / BSP) debug, put MSP and/or BSP text in the appropriate file(s) for preload. This method would facilitate diagnostics for checking resource contention between MSP and BSP. It's recommended that this step be included in the makefile of each diagnostic.

TABLE 82. HD diag list

	BSP	MSP	R4K	C model	VHDL	Description
v_hdreg.s			x	na	x	Write/Read of all registers in hd block
v_dmareg.s			x			Write/Read of all dma descriptor RAM
v_tlbmem.s			x	na	x	Write/Read of tlb RAM in VICE
v_hd_cnt.s			x			Verify Counter working in hd block
v_hd_fw.s			x	na	x	Write of Host Data PIO Fifo. Tests Back to Back Writes TestsFullBehavior/WRRDYStall
v_dma00.s			x	na	x	Initial DMA debug
v_dma01.s			x			DMA Ch1/2 block rd/wr from/to Sys-DRAM to/from Vice Data RAM A Test Round Trip Data Through Vice Both Channels Active Random PIO activity to Vice and Crime during DMA
v_dma02.s			x			DMA Ch1/2 block rd/wr from/to Sys-DRAM to/from Vice Data RAM B Test Round Trip Data Through Vice Both Channels Active Random PIO activity to Vice and Crime during DMA
v_dma03.s			x			DMA Ch1/2 block rd/wr from/to Sys-DRAM to/from Vice Data RAM C Test Round Trip Data Through Vice Both Channels Active Random PIO activity to Vice and Crime during DMA
v_dma04.s			x			DMA Ch1 block rd/wr from/to Sys-DRAM to/from Vice MSP IRAM
v_dma05.s			x			DMA Ch1 block rd/wr from/to Sys-DRAM to/from Vice BSP IRAM
v_dma06.s			x			DMA Ch1 block rd/wr from/to Sys-DRAM to/from Vice BSP Table RAM
v_dma07.s			x			DMA Ch1 block rd from Sys-DRAM to Vice BSP Input FIFO.
v_dma08.s			x			DMA Ch1 block wr to Sys-DRAM from Vice BSP Output FIFO.
v_dma09.s			x			DMA Ch1 block rd from Sys-DRAM to Vice TLB RAM.

TABLE 82. HD diag list

	BSP	MSP	R4K	C model	VHDL	Description
v_hd_fwr.s			x	na	x	Write of Host Data PIO Fifo. Tests Back to Back Writes TestsFullBehavior/WRRDYStall
v_dma01.s			x			DMA Ch1/2 block rd/wr from/to Sys-DRAM to/from Vice Data RAM A Test Round Trip Data Through Vice Both Channels Active Random PIO activity to Vice and Crime during DMA
v_dma02.s			x			DMA Ch1/2 block rd/wr from/to Sys-DRAM to/from Vice Data RAM B Test Round Trip Data Through Vice Both Channels Active Random PIO activity to Vice and Crime during DMA
v_dma10.s			x			DMA Ch2 block fill of Vice Data RAM A,B,C with diff patterns.
v_dma11.s			x			DMA Ch1/2 Skip/Halt verification.
v_dma12.s			x			DMA Ch1 block rd from Sys-DRAM Y/C 4:2:2 -> 4:2:0 split
v_dma13.s			x			DMA Ch1 block rd from Sys-DRAM Y/C 4:2:2 -> 4:2:2 split
v_dma14.s			x			DMA Ch1 block rd from Sys-DRAM Y Only 4:2:2 -> 4:
v_dma15.s			x			DMA Ch1 block rd from Sys-DRAM Chroma Only 4:2:2 -> 2:0
v_dma16.s			x			DMA Ch1 block wr to Sys-DRAM split 4:2:0 -> 4:2:2 Y/C
v_dma17.s			x			DMA Ch1 block wr to Sys-DRAM split 4:2:2 -> 4:2:2 Y/C
v_dma18.s			x			DMA Ch1 block rd from Sys-DRAM HPEL-00 HPEN-1 YC-01 ILV-0
v_dma19.s			x			DMA Ch1 block rd from Sys-DRAM HPEL-01 HPEN-1 YC-01 ILV-0
v_dma20.s			x			DMA Ch1 block rd from Sys-DRAM HPEL-10 HPEN-1 YC-01 ILV-0
v_dma21.s			x			DMA Ch1 block rd from Sys-DRAM HPEL-11 HPEN-1 YC-01 ILV-0
v_dma22.s			x			DMA Ch1 block rd from Sys-DRAM HPEL-00 HPEN-1 YC-01 ILV-1
v_dma23.s			x			DMA Ch1 block rd from Sys-DRAM HPEL-01 HPEN-1 YC-01 ILV-1
v_dma24.s			x			DMA Ch1 block rd from Sys-DRAM HPEL-10 HPEN-1 YC-01 ILV-1
v_dma25.s			x			DMA Ch1 block rd from Sys-DRAM HPEL-11 HPEN-1 YC-01 ILV-1
v_dma26.s			x			Flush DMA Ch1 block wr to Sys-DRAM from Vice BSP Output FIFO. Test Flush Command.
						Test Different Source/Destination Alignment Choices
v_dma27.s			x			Align DMA Ch1 block rd from Sys-DRAM Y/C 4:2:2 -> 4:2:0 split
v_dma28.s			x			Align DMA Ch1 block rd from Sys-DRAM Y/C 4:2:2 -> 4:2:2 split
v_dma29.s			x			Align DMA Ch1 block rd from Sys-DRAM Y Only 4:2:2 -> 4:
v_dma30.s			x			Align DMA Ch1 block rd from Sys-DRAM Chroma Only 4:2:2-> 2:0
v_dma31.s			x			Align DMA Ch1 block rd from Sys-DRAM HPEL-00 HPEN-1 YC-01 ILV-0
v_dma32.s			x			Align DMA Ch1 block rd from Sys-DRAM HPEL-01 HPEN-1 YC-01 ILV-0
v_dma33.s			x			Align DMA Ch1 block rd from Sys-DRAM HPEL-10 HPEN-1 YC-01 ILV-0
v_dma34.s			x			Align DMA Ch1 block rd from Sys-DRAM HPEL-11 HPEN-1 YC-01 ILV-0
v_dma35.s			x			Align DMA Ch1 block rd from Sys-DRAM HPEL-00 HPEN-1 YC-01 ILV-1
v_dma36.s			x			Align DMA Ch1 block rd from Sys-DRAM HPEL-01 HPEN-1 YC-01 ILV-1
v_dma37.s			x			Align DMA Ch1 block rd from Sys-DRAM HPEL-10 HPEN-1 YC-01 ILV-1
v_dma38.s			x			Align DMA Ch1 block rd from Sys-DRAM HPEL-11 HPEN-1 YC-01 ILV-1
v_hd_rdwr.s			x			SysAD Random PIO Write/Read to CRIME/VICE space
v_int.s			x			Load Code for MSP and BSP to software interrupt host.
v_reg.s			x			SysAD PIO Write/Read of all registers in VICE
v_dmem.s			x			SysAD PIO Write/Read of all Data RAM

TABLE 82. HD diag list

	BSP	MSP	R4K	C model	VHDL	Description
v_bsp_iram.s			x	na	x	SysAD PIO Write/Read of BSP Instruction RAM
v_iram.s			x	na	x	SysAD PIO Write/Read of MSP Instruction RAM
v_comdmaarb.s			x			Download BSP code, MSP code. BSP runs 1 DMA engine, MSP runs other. Both DMA engines moving data AND both MSP and BSP doing lots of load/store to Data RAM AND lots of Common Bus activity to exercise the hd_comarb.vhd and hd_dmaarb.vhd circuits in the hd block
BSP driven tests						
bsp_hdreg.s			x			Write/Read of all registers in hd block
bsp_dmareg.s	x	x	x	x		Write/Read of all dma descriptor RAM
bsp_hd_cnt.s	x	x	x	x		Verify Counter working in hd block
bsp_dma01.s	x	x	x	x		DMA Ch2 block rd/wr from/to Sys-DRAM to/from Vice Data RAM A
bsp_dma02.s	x	x	x	x		DMA Ch2 block rd/wr from/to Sys-DRAM to/from Vice Data RAM B
bsp_dma03.s	x	x	x	x		DMA Ch2 block rd/wr from/to Sys-DRAM to/from Vice Data RAM C
bsp_dma05.s	x	x	x	x		DMA Ch2 block rd/wr from/to Sys-DRAM to/from Vice BSP IRAM
bsp_dma06.s	x	x	x	x		DMA Ch2 block rd/wr from/to Sys-DRAM to/from Vice BSP Table RAM
bsp_dma07.s	x	x	x	x		DMA Ch2 block rd from Sys-DRAM to Vice BSP Input FIFO.
bsp_dma08.s	x	x	x	x		DMA Ch2 block wr to Sys-DRAM from Vice BSP Output FIFO.
bsp_dma09.s			x			DMA Ch2 block rd from Sys-DRAM to Vice TLB RAM.
bsp_dma10.s	x	x	x	x		DMA Ch2 block fill of Vice Data RAM A,B,C with diff patterns.
bsp_dma11.s	x	x	x	x		DMA Ch2 Skip/Halt verification.
bsp_dma12.s	x	x	x	x		DMA Ch2 block rd from Sys-DRAM Y/C 4:2:2 -> 4:2:0 split
bsp_dma13.s	x	x	x	x		DMA Ch2 block rd from Sys-DRAM Y/C 4:2:2 -> 4:2:2 split
bsp_dma14.s	x	x	x	x		DMA Ch2 block rd from Sys-DRAM Y Only 4:2:2 -> 4:
bsp_dma15.s	x	x	x	x		DMA Ch2 block rd from Sys-DRAM Chroma Only 4:2:2 -> 2:0
bsp_dma16.s	x	x	x	x		DMA Ch2 block wr to Sys-DRAM split 4:2:0 -> 4:2:2 Y/C
bsp_dma17.s	x	x	x	x		DMA Ch2 block wr to Sys-DRAM split 4:2:2 -> 4:2:2 Y/C
bsp_dma18.s	x	x	x	x		DMA Ch2 block rd from Sys-DRAM HPEL-00 HPEN-1 YC-01 ILV-0
bsp_dma19.s	x	x	x	x		DMA Ch2 block rd from Sys-DRAM HPEL-01 HPEN-1 YC-01 ILV-0
bsp_dma20.s	x	x	x	x		DMA Ch2 block rd from Sys-DRAM HPEL-10 HPEN-1 YC-01 ILV-0
bsp_dma21.s	x	x	x	x		DMA Ch2 block rd from Sys-DRAM HPEL-11 HPEN-1 YC-01 ILV-0
bsp_dma22.s	x	x	x	x		DMA Ch2 block rd from Sys-DRAM HPEL-00 HPEN-1 YC-01 ILV-1
bsp_dma23.s	x	x	x	x		DMA Ch2 block rd from Sys-DRAM HPEL-01 HPEN-1 YC-01 ILV-1
bsp_dma24.s	x	x	x	x		DMA Ch2 block rd from Sys-DRAM HPEL-10 HPEN-1 YC-01 ILV-1
bsp_dma25.s	x	x	x	x		DMA Ch2 block rd from Sys-DRAM HPEL-11 HPEN-1 YC-01 ILV-1
bsp_dma26.s	x	x	x	x		Flush DMA Ch2 block wr to Sys-DRAM from Vice BSP Output FIFO. Test Flush Command.
						Test Different Source/Destination Alignment Choices
bsp_dma27.s	x	x	x	x		Align DMA Ch2 block rd from Sys-DRAM Y/C 4:2:2 -> 4:2:0 split
bsp_dma28.s	x	x	x	x		Align DMA Ch2 block rd from Sys-DRAM Y/C 4:2:2 -> 4:2:2 split
bsp_dma29.s	x	x	x	x		Align DMA Ch2 block rd from Sys-DRAM Y Only 4:2:2 -> 4:
bsp_dma30.s	x	x	x	x		Align DMA Ch2 block rd from Sys-DRAM Chroma Only 4:2:2-> 2:0
bsp_dma31.s			x			Align DMA Ch2 block rd from Sys-DRAM HPEL-00 HPEN-1 YC-01 ILV-0
bsp_dma32.s			x			Align DMA Ch2 block rd from Sys-DRAM HPEL-01 HPEN-1 YC-01 ILV-0

TABLE 82. HD diag list

	BSP	MSP	R4K	C model	VHDL	Description
bsp_dma33.s			x			Align DMA Ch2 block rd from Sys-DRAM HPEL-10 HPEN-1 YC-01 ILV-0
bsp_dma34.s			x			Align DMA Ch2 block rd from Sys-DRAM HPEL-11 HPEN-1 YC-01 ILV-0
bsp_dma35.s			x			Align DMA Ch2 block rd from Sys-DRAM HPEL-00 HPEN-1 YC-01 ILV-1
bsp_dma36.s			x			Align DMA Ch2 block rd from Sys-DRAM HPEL-01 HPEN-1 YC-01 ILV-1
bsp_dma37.s			x			Align DMA Ch2 block rd from Sys-DRAM HPEL-10 HPEN-1 YC-01 ILV-1
bsp_dma38.s			x			Align DMA Ch2 block rd from Sys-DRAM HPEL-11 HPEN-1 YC-01 ILV-1
bsp_reg.s	x	x	x	x		BSP PIO Write/Read of all registers in VICE
bsp_dmem.s	x	x	x	x		BSP PIO Write/Read of all Data RAM
MSP driven tests						
msp_hdreg.s			x			Write/Read of all registers in hd block
msp_dmareg.s	x	x	x	x		Write/Read of all dma descriptor RAM
msp_hd_cnt.s	x	x	x	x		Verify Counter working in hd block
msp_dma01.s	x	x	x	x		DMA Ch1 block rd/wr from/to Sys-DRAM to/from Vice Data RAM A
msp_dma02.s	x	x	x	x		DMA Ch1 block rd/wr from/to Sys-DRAM to/from Vice Data RAM B
msp_dma03.s	x	x	x	x		DMA Ch1 block rd/wr from/to Sys-DRAM to/from Vice Data RAM C
msp_dma04.s	x	x	x	x		DMA Ch1 block rd/wr from/to Sys-DRAM to/from Vice MSP IRAM
msp_dma09.s			x			DMA Ch1 block rd from Sys-DRAM to Vice TLB RAM.
msp_dma10.s	x	x	x	x		DMA Ch1 block fill of Vice Data RAM A,B,C with diff patterns.
msp_dma11.s	x	x	x	x		DMA Ch1 Skip/Halt verification.
msp_dma12.s	x	x	x	x		DMA Ch1 block rd from Sys-DRAM Y/C 4:2:2 -> 4:2:0 split
msp_dma13.s	x	x	x	x		DMA Ch1 block rd from Sys-DRAM Y/C 4:2:2 -> 4:2:2 split
msp_dma14.s	x	x	x	x		DMA Ch1 block rd from Sys-DRAM Y Only 4:2:2 -> 4:
msp_dma15.s	x	x	x	x		DMA Ch1 block rd from Sys-DRAM Chroma Only 4:2:2 -> 2:0
msp_dma16.s	x	x	x	x		DMA Ch1 block wr to Sys-DRAM split 4:2:0 -> 4:2:2 Y/C
msp_dma17.s	x	x	x	x		DMA Ch1 block wr to Sys-DRAM split 4:2:2 -> 4:2:2 Y/C
msp_dma18.s	x	x	x	x		DMA Ch1 block rd from Sys-DRAM HPEL-00 HPEN-1 YC-01 ILV-0
msp_dma19.s	x	x	x	x		DMA Ch1 block rd from Sys-DRAM HPEL-01 HPEN-1 YC-01 ILV-0
msp_dma20.s	x	x	x	x		DMA Ch1 block rd from Sys-DRAM HPEL-10 HPEN-1 YC-01 ILV-0
msp_dma21.s	x	x	x	x		DMA Ch1 block rd from Sys-DRAM HPEL-11 HPEN-1 YC-01 ILV-0
msp_dma22.s	x	x	x	x		DMA Ch1 block rd from Sys-DRAM HPEL-00 HPEN-1 YC-01 ILV-1
msp_dma23.s	x	x	x	x		DMA Ch1 block rd from Sys-DRAM HPEL-01 HPEN-1 YC-01 ILV-1
msp_dma24.s	x	x	x	x		DMA Ch1 block rd from Sys-DRAM HPEL-10 HPEN-1 YC-01 ILV-1
msp_dma25.s	x	x	x	x		DMA Ch1 block rd from Sys-DRAM HPEL-11 HPEN-1 YC-01 ILV-1
msp_dma26.s			x			Flush DMA Ch1 block wr to Sys-DRAM from Vice BSP Output FIFO. Test Flush Command.
						Test Different Source/Destination Alignment Choices
msp_dma27.s	x	x	x	x		Align DMA Ch1 block rd from Sys-DRAM Y/C 4:2:2 -> 4:2:0 split
msp_dma28.s	x	x	x	x		Align DMA Ch1 block rd from Sys-DRAM Y/C 4:2:2 -> 4:2:2 split
msp_dma29.s	x	x	x	x		Align DMA Ch1 block rd from Sys-DRAM Y Only 4:2:2 -> 4:
msp_dma30.s	x	x	x	x		Align DMA Ch1 block rd from Sys-DRAM Chroma Only 4:2:2-> 2:0
msp_dma31.s			x			Align DMA Ch1 block rd from Sys-DRAM HPEL-00 HPEN-1 YC-01 ILV-0
msp_dma32.s			x			Align DMA Ch1 block rd from Sys-DRAM HPEL-01 HPEN-1 YC-01 ILV-0
msp_dma33.s			x			Align DMA Ch1 block rd from Sys-DRAM HPEL-10 HPEN-1 YC-01 ILV-0

TABLE 82. HD diag list

	BSP	MSP	R4K	C model	VHDL	Description
msp_dma34.s			x			Align DMA Ch1 block rd from Sys-DRAM HPEL-11 HPEN-1 YC-01 ILV-0
msp_dma35.s			x			Align DMA Ch1 block rd from Sys-DRAM HPEL-00 HPEN-1 YC-01 ILV-1
msp_dma36.s			x			Align DMA Ch1 block rd from Sys-DRAM HPEL-01 HPEN-1 YC-01 ILV-1
msp_dma37.s			x			Align DMA Ch1 block rd from Sys-DRAM HPEL-10 HPEN-1 YC-01 ILV-1
msp_dma38.s			x			Align DMA Ch1 block rd from Sys-DRAM HPEL-11 HPEN-1 YC-01 ILV-1
msp_reg.s			x			SysAD PIO Write/Read of all registers in VICE
msp_dmem.s	x	x	x	x		SysAD PIO Write/Read of all Data RAM

TABLE 83. BSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
bsp_LoadH_00.s	X					bsp LoadH from DMA bus (whole address space)
bsp_LoadH_04.s	X	X	X			bsp LoadH from DMA bus with stall (DMA not available)
bsp_LoadH_07.s	X					bsp LoadH from DMA bus with back to back load (burst)
bsp_LoadH_10.s	X					bsp LoadH from common bus (whole address space)
bsp_LoadH_14.s	X	X	X			bsp LoadH from common bus with stall (common bus not available)
bsp_LoadH_17.s						
bsp_ls_01.s	x			x	x	simple halfowrd store. Memory model 1 MLF
bsp_ls_02.s	x			x	x	simple halfword stores Memory model 1 MLF
bsp_ls_03.s	x			x	x	load and store bytes Memory model 1 MLF
bsp_ls_04.s	x			x	x	load and store bytest Memory model 1 MLF
bsp_ls_05.s	x			x	x	load and store bytest Memory model 2 MLF
bsp_ls_10.s	x			x	x	load store -- Massive bus activity. MLF
dmaregs.s	x			x	x	load store to dma registers
dmaregs1.s	x			x	x	load store to dma registers
bsp_LoadBl_00.s	X					bsp LoadBl from DMA bus (whole address space)
bsp_LoadBl_04.s	X	X	X			bsp LoadBl from DMA bus with stall (DMA not available)
bsp_LoadBl_10.s	X					bsp LoadBl from common bus (whole address space)
bsp_LoadBl_14.s	X	X	X			bsp LoadBh from common bus with stall (common bus not available)
bsp_LoadBh_00.s	X					bsp LoadBh from DMA bus (whole address space)
bsp_LoadBh_04.s	X	X	X			bsp LoadBh from DMA bus with stall (DMA not available)
bsp_LoadBh_10.s	X					bsp LoadBh from common bus (whole address space)
bsp_LoadBh_14.s	X	X	X			bsp LoadBh from common bus with stall (common bus not available)
bsp_StoreH_00.s	x					bsp StoreH from DMA bus (whole address space)
bsp_StoreH_04.s	x	x	x			bsp StoreH from DMA bus with stall (DMA not available)
bsp_StoreH_07.s	x					bsp StoreH from DMA bus with back to back load (burst)
bsp_StoreH_10.s	x					bsp StoreH from common bus (whole address space)
bsp_StoreH_14.s	x	x	x			bsp StoreH from common bus with stall (common bus not available)
bsp_StoreH_17.s	x					bsp StoreH from common bus with back to back load (burst)
bsp_StoreBl_00.s	x					bsp StoreBl from DMA bus (whole address space)
bsp_StoreBl_04.s	x	x	x			bsp StoreBl from DMA bus with stall (DMA not available)

TABLE 83. BSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
bsp_StoreBl_10.s	x					bsp StoreBl from common bus (whole address space)
bsp_StoreBl_14.s	x	x	x			bsp StoreBh from common bus with stall (common bus not available)
bsp_StoreBh_00.s	x					bsp StoreBh from DMA bus (whole address space)
bsp_StoreBh_04.s	x	x	x			bsp StoreBh from DMA bus with stall (DMA not available)
bsp_StoreBh_10.s	x					bsp StoreBh from common bus (whole address space)
bsp_StoreBh_14.s	x	x	x			bsp StoreBh from common bus with stall (common bus not available)
bsp_LDIh-0.s	x					bsp LDI_h - all register destinations
bsp_LDIh-1.s	x					bsp LDI_h - all register destinations
bsp_LDIl-0.s	x					bsp LDI_l - all register destinations
bsp_LDIl-1.s	x					bsp LDI_l - all register destinations
bsp_cmpi_00.s	x			x	x	bsp CMPI - check all flags set properly
bsp_andi_00.s	x			x	x	bsp ANDI - check all flags set properly
bsp_beq_00.s				x	x	bsp BRANCH check branch correctly taken (forward and backward) for each condition code check that instruction in delay slot is executed properly
bsp_beq_01.s	x			x	x	bsp BRANCH
bsp_beq_02.s	x			x	x	bsp BRANCH
bsp_bge_00.s	x			x	x	bsp BRANCH
bsp_bge_01.s	x			x	x	bsp BRANCH
bsp_bge_02.s	x			x	x	bsp BRANCH
bsp_bne_00.s	x			x	x	bsp BRANCH
bsp_bne_01.s	x			x	x	bsp BRANCH
bsp_bne_02.s	x			x	x	bsp BRANCH
bsp_bne_03.s	x			x	x	bsp BRANCH
bsp_blt_00.s	x			x	x	bsp_BRANCH
bsp_blt_01.s	x			x	x	bsp BRANCH
bsp_blt_02.s	x			x	x	bsp BRANCH
bsp_blt_03.s	x			x	x	bsp BRANCH
bsp_ext0_00.s	x			x	x	bsp BRANCH
bsp_ext0_01.s	x			x	x	bsp BRANCH
bsp_ext0_02.s	x			x	x	bsp BRANCH
bsp_ext0_03.s	x			x	x	bsp BRANCH
bsp_ext1_00.s	x			x	x	bsp BRANCH
bsp_ext1_01.s	x			x	x	bsp BRANCH
bsp_ext1_02.s	x			x	x	bsp BRANCH
bsp_ext1_03.s	x			x	x	bsp BRANCH
bsp_ext2_00.s	x			x	x	bsp BRANCH
bsp_ext2_01.s	x			x	x	bsp BRANCH
bsp_ext2_02.s	x			x	x	bsp BRANCH
bsp_ext2_03.s	x			x	x	bsp BRANCH

TABLE 83. BSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
bsp_jr_00.s	x			x	x	bsp JR check branch correctly taken (forward and backward) for each condi-tion code. check that instruction in delay slot is executed properly
bsp_jr_01.s	x			x	x	bsp JR
bsp_jr_02.s	x			x	x	bsp JR
bsp_jr_03.s	x			x	x	bsp JR
bsp_jr_04.s	x			x	x	bsp JR
bsp_jr_05.s	x			x	x	bsp JR
bsp_jr_06.s	x			x	x	bsp JR
bsp_jr_07.s	x			x	x	bsp JR
bsp_jr_08.s	x			x	x	bsp JR
bsp_jr_09.s	x			x	x	bsp JR
bsp_jr_10.s	x			x	x	bsp JR
bsp_jr_11.s	x			x	x	bsp JR
bsp_jr_12.s	x			x	x	bsp JR
bsp_jr_13.s	x			x	x	bsp JR
bsp_jr_14.s	x			x	x	bsp JR
bsp_break_00.s	x					bsp BREAK check that no instruction is executed (partially) after the break check that appropriate PC is saved
bsp_break_01.s	x					bsp BREAK
bsp_break_02.s	x					bsp BREAK
bsp_break_03.s	x					bsp BREAK
bsp_break_04.s	x					bsp BREAK
bsp_break_05.s	x					bsp BREAK
bsp_resume_00.s	x					bsp RESUME
bsp_resume_01.s	x					bsp RESUME
bsp_resume_02.s	x					bsp RESUME
bsp_resume_03.s	x					bsp RESUME
bsp_addi_00.s	x			x	x	bsp ADDI
bsp_addi_01.s	x			x	x	bsp ADDI
bsp_add_00.s	x			x	x	bsp ADD check results (“corner cases”) check that flags are set properly
bsp_add_01.s	x			x	x	bsp ADD
bsp_add_02.s	x			x	x	bsp ADD
bsp_addc_00.s	x			x	x	bsp ADDC check results (“corner cases”) check that flags are set properly perform double precision addition w/wo carry across halfwords
bsp_addc_01.s	x			x	x	bsp ADDC

TABLE 83. BSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
bsp_sub_00.s	x			x	x	bsp SUB check results (“corner cases”) check that flags are set properly
bsp_sub_01.s	x			x	x	bsp SUB
bsp_cmp_00.s	x					bsp CMP check results (“corner cases”) check that flags are set properly
bsp_or_00.s	x			x	x	bsp OR check results (“corner cases”) check that flags are set properly
bsp_and_00.s	x			x	x	bsp AND
bsp_lshft_00.s	x			x	x	bsp LSHIFT check results (“corner cases”) check that flags are set properly check all 16 shifts
bsp_lshft_01.s	x			x	x	bsp LSHIFT
bsp_lshft_02.s	x			x	x	bsp LSHIFT
bsp_arshft_00.s	x			x	x	bsp ARSHIFT check shift for positive and negative numbers
bsp_mult_00.s	x			x	x	bsp MULT
bsp_mult_01.s	x					bsp MULT
bsp_mult_02.s	x					bsp MULT
testZZ_01.s	x			x	x	bsp ZZXLATE check all input/output combinations ZZ scan and Alt scan
testZZ_02.s	x			x	x	bsp ZZXLATE
testZZ_03.s	x			x	x	bsp ZZXLATE
testZZ_04.s	x			x	x	bsp ZZXLATE
testAlt_01.s	x			x	x	bsp ZZXLATE
testAlt_02.s	x			x	x	bsp ZZXLATE
testAlt_03.s	x			x	x	bsp ZZXLATE
testAlt_04.s	x			x	x	bsp ZZXLATE
bsp_xor_00.s	x			x	x	bsp XOR check results (“corner cases”) check that flags are set properly
bsp_abs_00.s	x			x	x	bsp ABS check results (positive and negative inputs) check that flags are set properly.
bsp_abs_01.s	x			x	x	bsp ABS
bsp_abs_02.s	x			x	x	bsp ABS
bsp_abs_03.s	x			x	x	bsp ABS
bsp_neg_00.s	x			x	x	bsp NEG check results (“corner cases”) check that flags are set properly
bsp_copy_00.s	x			x	x	COPYTO COPYFROM instruction test rPAGE

TABLE 83. BSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
bsp_copy_01.s	x			x	x	COPYTO COPYFROM instruction test Alpha_h
bsp_copy_02.s	x			x	x	COPYTO COPYFROM instruction test Alpha_l
bsp_copy_03.s	x			x	x	COPYTO COPYFROM instruction test Beta_h
bsp_copy_04.s	x			x	x	COPYTO COPYFROM instruction test Beta_l
bsp_copy_05.s	x			x	x	COPYTO COPYFROM instruction test CMP_h
bsp_copy_06.s	x			x	x	COPYTO COPYFROM instruction test CMP_l
bsp_copy_07.s	x			x	x	COPYTO COPYFROM instruction test MASK_h
bsp_copy_08.s	x			x	x	COPYTO COPYFROM instruction test MASK_l
bsp_copy_09.s	x			x	x	COPYTO COPYFROM instruction test BLOCK_DATA
bsp_copy_10.s	x			x	x	COPYTO COPYFROM instruction test BLOCK_ADDRESS
bsp_copy_11.s	x			x	x	COPYTO COPYFROM instruction test BLOCK_PTR
bsp_copy_12.s	x			x	x	COPYTO COPYFROM instruction test ROOT_TBL_PTR
bsp_copy_13.s	x			x	x	
bsp_lload_00.s	x			x	x	LLOADH
bsp_lload_01.s	x			x	x	LLOADH
bsp_lstore_00.s	x					LSTOREH
bsp_lstore_01.s	x					LSTOREH
bsp_getbit_01.s	x			x	x	getbits(q) rD, N Check all values of N Check with/wo byte swallowing STALLs when not enough data in bitstream buffer
bsp_getbit_02.s	x			x	x	getbits(q) rD,rT Check all allowed values in rT Check with/wo byte swallowing STALLs when not enough data in bitstream buffer
bsp_getbit_03.s	x			x	x	getbits
bsp_getbit_04.s	x			x	x	getbits
bsp_getbit_05.s	x			x	x	getbits
bsp_getbit_06.s	x			x	x	getbits
bsp_getbit_07.s	x			x	x	getbits
bsp_getbit_08.s	x			x	x	getbits
bsp_getbit_09.s	x			x	x	getbits
bsp_getbit_10.s	x			x	x	getbits
bsp_getbit_11.s	x			x	x	getbits
bsp_getbit_12.s	x			x	x	getbits
bsp_getbit_13.s	x			x	x	getbits
bsp_getbit_14.s	x			x	x	getbits
bsp_getbit_15.s	x			x	x	getbits
bsp_getbit_16.s	x			x	x	getbits
bsp_getbit_reg_5_08	x			x		getbits
bsp_getbit_reg_5_09	x			x	x	getbits
bsp_getbit_reg_5_10	x			x	x	getbits
bsp_getbit_reg_5_11	x			x	x	getbits
bsp_getbit_reg_5_12	x			x	x	getbits

TABLE 83. BSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
bsp_getbit_reg_5_13	x			x	x	getbits
bsp_getbit_reg_5_14	x			x	x	getbits
bsp_getbit_reg_5_15	x			x	x	getbits
bsp_getbit_reg_5_16	x			x	x	getbits
bsp_probeBits_0.s	x					probeBits rD, N Check all values of N
bsp_shiftstream_0.s	x					shiftstream(N,q) Check all values of N Check with/wo byte swallowing STALLs when not enough data in bitstream buffer
bsp_lcp_261_0.s	x					load_code_packH261(q,p) use to encode blocks of DCT coefficients (Zig-zag order) intra block inter block cover all run-level pairs in the VLC table cover in-table escape codes cover hardwired escape codes STALLS when output Fifo is full
bsp_lcp_261_1.s	x					load_code_packH261(q,p)
bsp_lcp_261_2.s	x					load_code_packH261(q,p)
bsp_lcp_261_3.s	x					load_code_packH261(q,p)
run_lvl_parse_0.s	x					block_run_level_parse(q) use to decode H261, MPEG-1 and MPEG-2 block. cover all run-level pairs for each standard escape code processing for each standard use to check RESUME instruction
btestB15_02.s	x			x	x	MPEG2 INTRA-BLOCK-LUMA DC term (positive)
btestB15_03.s	x			x	x	MPEG2 INTRA-BLOCK-LUMA DC term (negative)
ltestB15_01.s	x			x	x	leaf run_level_parse
ltestB15_02.s	x			x	x	leaf run_level_parse
ltestB15_03.s	x			x	x	leaf run_level_parse
mpeg2_block_dec_0.s	x			x	x	MPEG2 intra_Luma DC check
mpeg2_block_dec_1.s	x			x	x	MPEG2 intra_Chroma DC check
mpeg2_block_dec_2.s	x			x	x	MPEG2 intra_Chroma check w Escape code
mpeg2_block_dec_3.s	x			x	x	MPEG2 intra_Chroma back to back escapes
h261_block_dec_0.s	x			x	x	H261 intra DC= 255
h261_block_dec_1.s	x			x	x	H261 intra with 111 (run 0 level -1) AC
h261_block_dec_2.s	x			x	x	H261 intra with lots of AC terms. s-bit causes 32-bit load of bitstream
h261_block_dec_3.s	x			x	x	H261 intra with escape codes
h261_block_dec_4.s	x			x	x	H261 intra with back_to_back escape codes
h261_block_dec_5.s	x			x	x	H261 inter with back_to_back escape codes
table_B1_test_00.s	x			x	x	generic_leaf_parse tests on MPEG-2 tables use to decode H261, MPEG-1 and MPEG-2 macrblock headers. cover all table locations for H261 and MPEG macroblock headers
table_B1_test_1.s	x			x	x	generic_leaf_parse

TABLE 83. BSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
table_B2_test_0.s	x			x	x	generic_leaf_parse
table_B3_test_0.s	x			x	x	generic_leaf_parse
table_B4_test_0.s	x			x	x	generic_leaf_parse
table_B10_test_0.s	x			x	x	generic_leaf_parse
table_B10_test_1.s	x			x	x	generic_leaf_parse
table_B11_test_0.s	x			x	x	generic_leaf_parse
table_B12_test_0.s	x			x	x	generic_leaf_parse
table_B13_test_0.s	x			x	x	generic_leaf_parse
run_size_0.s	x					block_run_size_parse use to decode JPEG blocks cover all run_size combinations for JPEG blocks JPEG byte swallowing -- check rigourously
b01test.s	x			x	x	code_search - search for different lenght codes in the bitstream.
b02test.s	x			x	x	code_search - search for different lenght codes in the bitstream.
b32test.s	x			x	x	code_search - search for different lenght codes in the bitstream.
b32Atest.s	x			x	x	code_search - search for different lenght codes in the bitstream.
csearch_byte_0.s	x					code_search - using byte alignement
pack_bitstream_0.s	x					pack_bitstream pack all bit sizes from 1-16. check JPEG byte insertion
Lcode_pack_0.s	x					load-code_pack(q,p) offset Perform JPEG block encodings. Cover all JPEG encoder table entries. Check ZRL packing. Multiple ZRLs
byte_align_0.s	x					byte align
pack_bitstream_0.s	x					pack_bitstream(q) M, IMM pack all bit sizes Check JPEG byte insertion. this byte insetrtion testing should be rigorous.
pack_bitstream_1.s	x					pack_bitstream(q) M, IMM
pack_bitstream_2.s	x					pack_bitstream(q) M, IMM
bsp_bypass_00	x			x	x	bsp - ADD operation -both RS and RT bypassed
bsp_bypass_01	x			x	x	bsp - ADD operation -both RS and RT bypassed
bsp_bypass_02	x			x	x	bsp - ADD operation -both RS and RT bypassed
bsp_bypass_03	x			x	x	bsp - LSHIFT operation -both RS and RT bypassed
bsp_bypass_04	x			x	x	bsp - CMPI operation - RS bypassed
bypass_load.s	x			x	x	bsp - Load Bypassd
bypass_load1.s	x			x	x	bsp - Load Bypassd
bypass_load2.s	x			x	x	bsp - Load Bypass
bypass_load3.s	x			x	x	bsp - Load Bypass
bypass_load4.s	x			x	x	bsp - Load Bypass

TABLE 83. BSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
bsp_loadi_byps_0.s	x					bsp - Load Immediate bypass both RS/High and Low Byte and RT/High and Low Byte bypassed LDI_l r5, IMM ADD r6, r5, r4
bsp_copy_byps_0.s	x					bsp - COPYFROM Bypass - both RS and RT bypassed COPYFROM r5, ALT_REG ADD r6, r5, r4
bypass_add_00.s	x			x	x	bsp - ALU operation Bypass - both RS and RT bypassed
bypass_sub_00.s	x			x	x	bsp - ALU operation Bypass - both RS and RT bypassed
bypass_or_00.s	x			x	x	bsp - ALU operation Bypass - both RS and RT bypassed
bypass_and_00.s	x			x	x	bsp - ALU operation Bypass - both RS and RT bypassed
bypass_addc_00.s	x			x	x	bsp - ALU operation Bypass - both RS and RT bypassed
bypass_add.s	x			x	x	bsp - ALU operation Bypass - both RS and RT bypassed
bypass_sub.s	x			x	x	bsp - ALU operation Bypass - both RS and RT bypassed
bypass_or.s	x			x	x	bsp - ALU operation Bypass - both RS and RT bypassed
bypass_and.s	x			x	x	bsp - ALU operation Bypass - both RS and RT bypassed
bypass_mul.s	x			x	x	bsp - ALU operation Bypass - both RS and RT bypassed
bypass_lsh.s	x			x	x	bsp - ALU operation Bypass - both RS and RT bypassed
bypass_xor.s	x			x	x	bsp - ALU operation Bypass - both RS and RT bypassed
bypass_neg.s	x			x	x	bsp - ALU operation Bypass - both RS and RT bypassed
bypass_abs.s	x			x	x	bsp - ALU operation Bypass - both RS and RT bypassed
bypass_zzx.s	x			x	x	bsp - ALU operation Bypass - both RS and RT bypassed
bypass_cpf.s	x			x	x	bsp - ALU operation Bypass - both RS and RT bypassed
bypass_andi.s	x			x	x	bsp - ALU operation Bypass - both RS and RT bypassed
bypass_addi.s	x			x	x	bsp - ALU operation Bypass - both RS and RT bypassed
bypass_get.s	x			x		bypass_get
bypass_geti.s	x			x		bypass_get
bypass_prb.s	x			x	x	bypass_get
pack_get.s	x			x	x	pkbit and getbits
bsp_pack_01s	x			x	x	pack_bitstream
pack_glook_01.s	x			x		generic_lookup_pack
bsp_ibox_0.s	x	x	x			BSP_IN_BOX check setting of b15 on write by MSP/host check clearing of b15 on read by BSP check data transfer
bsp_obox_0.s	x	x	x			BSP_OUT_BOX check setting of b15 on BSP write check clearing of b15 on read by MSP/host check data transfer
bsp_halt_rst_0.s	x	x	x			BSP_HALT_RESET check HALT functionality when written by HOST/MSP check RESET functionality when written by HOST/MSP/BSP

TABLE 83. BSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
bsp_ctl_stat_0.s	x	x	x			BSP_CTL_STAT check setting of EXT0, EXT1, EXT2 by BSP and HOST/MSP Bitstream buffer reset. WB_empty N,Z,C,V
bsp_avld_bits_0.s	x	x	x			BSP_AVALID_BITS
bsp_fvld_bits_0.s	x	x	x			BSP_FVALID_BITS
bsp_fifoc_stat_0.s	x	x	x			BSP_FIFO_CTL_STAT
bsp_in_count_0.s	x					BSP_IN_COUNT
bsp_out_count_0.s	x					BSP_OUT_COUNT
bsp_watch_pt_0.s	x					BSP_WatchPoint
bsp_pc_0.s	x		x			BSP_PC
bsp_epc_0.s	x		x			BSP_EPC
bsp_cause_0.s	x		x			BSP_CAUSE
bsp_adr_err_0.s	x					Address Error Exception
bsp_fifo_empty_0.s	x					Bitstream FIFO empty for 4K cycles
bsp_fifo_full_0.s	x					Bitstream FIFO full for 4K cycles
bitstream_exp_0.s	x		x			Bitstream Error Exception:
xcape_code_0.s	x		x			Escape Code (H261/MPEG)
bsp_ls_stall_0.s	x	x	x			Stall on Load/Store from DMA_DB - single cycle stall
bsp_ls_stall_1.s	x	x	x			Stall on Load/Store from DMA_DB - single cycle stall
bsp_ls_stall_5.s	x	x	x			Stall on Load/Store from DMA_DB - multiple cycle stall
bsp_ls_stall_6.s	x	x	x			Stall on Load/Store from DMA_DB - multiple cycle stall
getbits_stall_0.s	x					Stall on Input FIFO empty - getbits
leaf_parse_stall_0.s	x					Stall on Input FIFO empty - generic_leaf_parse
run_level_stall_0.s	x					Stall on Input FIFO empty - block_run_level_parse
run_size_stall_0.s	x					Stall on Input FIFO empty - block_run_size_parse
code_search_stall_0.s	x					Stall on Input FIFO empty - code_search
Shiftstream_stall_0.s	x					Stall on Input FIFO empty - Shiftstream
leaf_parse_stall_1.s	x					Stall on Write Buffer Full - generic_leaf_parse
run_level_stall_1.s	x					Stall on Write Buffer Full - block_run_level_parse
run_size_stall_1.s	x					Stall on Write Buffer Full - block_run_size_parse
p_bitstream_stall_0.s	x					Stall on Write Buffer Full - pack_bitstream
load_code_stall_0.s	x					Stall on Write Buffer Full - load_code_pack
load_code_stall_1.s	x					Stall on Write Buffer Full - load_code_pack_H261
lookup_stall_0.s	x					Stall on Write Buffer Full - generic_lookup_pack
leaf_parse_stall_2.s	x					Stall on output FIFO Full - generic_leaf_parse
run_level_stall_2.s	x					Stall on output FIFO Full - block_run_level_parse
run_size_stall_2.s	x					Stall on output FIFO Full - block_run_size_parse
p_bitstream_stall_1.s	x					Stall on output FIFO Full - pack_bitstream
load_code_stall_2.s	x					Stall on output FIFO Full - load_code_pack
load_code_stall_3.s	x					Stall on output FIFO Full - load_code_pack_H261
lookup_stall_1.s	x					Stall on output FIFO Full - generic_lookup_pack

TABLE 83. BSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
escape_interrupt						H261/MPEG escape code handling -- interrupt mechanism
write_buffer_bsp						Bus arbitration between bsp and write buffer
bsp_break						bsp break instruction
bsp_halt	x	x				MSP halts the BSP
bsp_halt	x		x			R4K HALTS the BSP
bsp_debug						execute debugger (code restart etc.)
bsp_getbit						getbits instrcution (immediate)
bsp_probe_00	x			x	x	probe_bits instruction
bsp_getbit						getbits (register)
bsp_shiftstream						shiftstream instruction
bsp_ba						byte_align instruction
bsp_lrp						leaf_run_level_parse
bsp_lrp_mpeg2_00	x			x	x	leaf_run_level_parse using MPEG2 table
bsp_lrp_mpeg2_01	x			x	x	leaf_run_level_parse using MPEG2 table
bsp_lrp_mpeg2_02	x			x	x	leaf_run_level_parse using MPEG2 table
bsp_lrp_mpeg2_03	x			x	x	leaf_run_level_parse using MPEG2 table
bsp_brlp						block_run_level_parse instruction
bsp_lcp261						load_code_pack_H261
bsp_gleaf_mba_01	x			x	x	generic_leaf_parse on H261 MBA
bsp_gleaf_mba_02	x			x	x	generic_leaf_parse on H261 MBA
bsp_gleaf_mba_03	x			x	x	generic_leaf_parse on H261 MBA
bsp_gleaf_mtype_01	x			x	x	generic_leaf_parse on H261 MTYPE
bsp_gleaf_mvd_01	x			x	x	generic_leaf_parse on H261 MVD
bsp_gleaf_mvd_02	x			x	x	generic_leaf_parse on H261 MVD
bsp_gleaf_mvd_03	x			x	x	generic_leaf_parse on H261 MVD
bsp_gleaf_cbp_01	x			x	x	generic_leaf_parse on H261 CBP
bsp_gleaf_cbp_02	x			x	x	generic_leaf_parse on H261 CBP
bsp_gleaf_cbp_03	x			x	x	generic_leaf_parse on H261 CBP
bsp_gleaf_cbp_04	x			x	x	generic_leaf_parse on H261 CBP
bsp_genleaf_01	x			x	x	generic_leaf_parse instruction on H261 tables
bsp_genleaf_02	x			x	x	generic_leaf_parse instruction on H261 tables
bsp_genleaf_03	x			x	x	generic_leaf_parse instruction on H261 tables
bsp_brsp						block_run_size_parse instruction (INCLUDE test of byte swaloow-ing)
bsp_brs_M1	x			x	x	block_run_size_parse
bsp_brs_M2	x			x	x	block_run_size_parse
bsp_brs_M3	x			x	x	block_run_size_parse
bsp_brs_M4	x			x	x	block_run_size_parse
bsp_brs_M5	x			x	x	block_run_size_parse
bsp_brs_M6	x			x	x	block_run_size_parse
bsp_brs_M7	x			x	x	block_run_size_parse
bsp_brs_M8	x			x	x	block_run_size_parse
bsp_brs_M9	x			x	x	block_run_size_parse

TABLE 83. BSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
bsp_brs_M10	x			x	x	block_run_size_parse
bsp_brs_M1	x			x	x	block_run_size_parse
bsp_brs_M12	x			x	x	block_run_size_parse
bsp_brs_M13	x			x	x	block_run_size_parse
bsp_brs_M14	x			x	x	block_run_size_parse
bsp_brs_M15	x			x	x	block_run_size_parse
bsp_brs_M16	x			x	x	block_run_size_parse
bsp_brs_M17	x			x	x	block_run_size_parse
bsp_brs_M18	x			x	x	block_run_size_parse
bsp_brs_M19	x			x	x	block_run_size_parse
bsp_brs_M20	x			x	x	block_run_size_parse
bsp_brs_M21	x			x	x	block_run_size_parse
bsp_brs_M22	x			x	x	block_run_size_parse
bsp_brs_M23	x			x	x	block_run_size_parse
bsp_brs_M24	x			x	x	block_run_size_parse
bsp_brs_M25	x			x	x	block_run_size_parse
bsp_brs_M26	x			x	x	block_run_size_parse
bsp_brs_M27	x			x	x	block_run_size_parse
bsp_brs_M28	x			x	x	block_run_size_parse
bsp_brs_M29	x			x	x	block_run_size_parse
bsp_brs_M30	x			x	x	block_run_size_parse
bsp_brs_M31	x			x	x	block_run_size_parse
bsp_brs_M32	x			x	x	block_run_size_parse
bsp_brs_M33	x			x	x	block_run_size_parse
bsp_brs_M34	x			x	x	block_run_size_parse
bsp_brs_M35	x			x	x	block_run_size_parse
bsp_brs_M36	x			x	x	block_run_size_parse
bsp_brs_X1	x			x	x	block_run_size_parse
bsp_brs_X2	x			x	x	block_run_size_parse
bsp_brs_X3	x			x	x	block_run_size_parse
bsp_brs_X4	x			x	x	block_run_size_parse
bsp_brs_X5	x			x	x	block_run_size_parse
bsp_brs_X6	x			x	x	block_run_size_parse
bsp_brs_X7	x			x	x	block_run_size_parse
bsp_brs_X8	x			x	x	block_run_size_parse
bsp_brs_X9	x			x	x	block_run_size_parse
bsp_brs_X10	x			x	x	block_run_size_parse
bsp_brs_X11	x			x	x	block_run_size_parse
bsp_brs_X12	x			x	x	block_run_size_parse
bsp_brs_X13	x			x	x	block_run_size_parse
bsp_search_01	x			x	x	code_search instruction
b01test	x			x	x	code_search instruction
b02test	x			x	x	code_search instruction

TABLE 83. BSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
b32test	x			x	x	code_search instruction
b32Atest	x			x	x	code_search instruction
bsp_lcp						load_code_pack instruction (INCLUDE byte INSERTION)
bsp_tbl						bsp accessing its table memory via load/store operations
test_01.s	x			x	x	general bsp test - LDI
test_02.s	x			x	x	general bsp test - StoreB & LoadB
test_03.s	x			x	x	general bsp test - StoreH & LoadH
test_04.s	x			x	x	general bsp test - AND & OR
test_06.s	x			x	x	general bsp test - XOR, ABS, CMP & NEG
test_07.s	x			x	x	general bsp test - ADD & SUB
test_08.s	x			x	x	general bsp test - BRANCH eq, neq, ge, lt, ext0, ext1, ext2
test_09.s	x			x	x	general bsp test - COPYTO/COPYFROM rpage, alpha, beta, cmp, mask
test_10.s	x			x	x	general bsp test - LStoreH & LLoadH
test_12.s	x			x	x	general bsp test - shiftstream & getbits
test_13.s	x			x	x	general bsp test - getbits(0) r, N
test_14.s	x			x	x	general bsp test - getbits(0) r, r
test_15.s	x			x	x	general bsp test - probeBits r,N
test_16.s	x			x	x	general bsp test - code_search
test_18.s	x			x	x	general bsp test - MULT
test_23.s	x			x	x	general bsp test - code_search
test_24.s	x			x	x	general bsp test - ABS
test_25.s	x			x	x	general bsp test - ABS
test_26.s	x			x	x	general bsp test - CMPI
test_27.s	x			x	x	general bsp test - JPEG encode with DC term only
test_28.s	x			x	x	general bsp test - JPEG encode with DC term & [7][7] term
test_29.s	x			x	x	general bsp test - ADD instruction
test_30.s	x			x	x	general bsp test - ADDC instruction
test_32.s	x			x	x	general bsp test - JPEG encode with DC term only
test_33.s	x			x		general bsp test - JPEG encode with DC term & [7][7] term

TABLE 84. MSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
						[EXCEPTION] - check relevant state (EPC, Cause, BadAddr ...)
msp_AdrErr_0.s		x				Address Error Load - scalar
msp_AdrErr_2.s		x				Address Error Load - vector
msp_AdrErr_3.s		x				Address Error Load in delay slot
msp_AdrErr_6.s		x				Address Error Store - scalar
msp_AdrErr_8.s		x				Address Error Store - vector

TABLE 84. MSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
msp_AdrErr_9.s		x				Address Error Store in delay slot
msp_AdrErr_4.s		x				Address Error Store - Watch Point
msp_break_0.s		x				Break Point
msp_watch_0.s		x				Watch Point scalar
msp_watch_1.s		x				Watch Point vector
msp_watch_2.s		x				Watch Point in delay slot
msp_watch_3.s		x				Watch Point load/store
msp_reservedI_0.s		x				SU Reserved Instruction
msp_reservedI_2.s		x				SU Reserved Instruction - Watch Point
msp_reservedI_4.s		x				SU Reserved Instruction in delay slot
msp_reservedI_6.s		x				VU Reserved Instruction
msp_reservedI_8.s		x				VU Reserved Instruction - Watch Point
msp_reservedI_9.s		x				VU Reserved Instruction in delay slot
msp_content_0.s	x	x				Contention : MSP - BSP
msp_content_4.s		x	x			Contention : MSP - DMA
msp_Iadr_excpt_0.s		x				Instruction Fetch Addr Exception : ...
						[Halt/Reset] - Check machine state after Halt/Reset.
msp_halt_0.s		x				Halt/Break occuring during normal instruction stream
msp_halt_1.s		x				Halt/Break occuring in a delay slot
msp_halt_2.s		x				Halt/Break occuring after a branch taken
msp_halt_3.s		x				Halt/Break occuring after a branch not taken
msp_halt_4.s		x				Reset MSP set.
						[Host Writing to Regs]
msp_regs_00.s		x	x			Read/Write to MSP_CTL_STAT
msp_regs_01.s		x	x			Read/Write to MSP_ExcpFlag
msp_regs_02.s		x	x			Read/Write to MSP_PC
msp_regs_03.s		x	x			Read/Write to MSP_WatchPoint
msp_regs_04.s		x	x			Read MSP_BadAddr
msp_regs_05.s		x	x			Read MSP_EPC
msp_regs_06.s		x	x			Read MSP_Cause
						[Coprocessor 1,3]
msp_cop1_0.s		x				Read/Write all Cop 1 Space Regs
msp_cop3_0.s		x				Read/Write all Cop 3 Space Regs

TABLE 84. MSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
						[Other Vector Load Store Tests]
msp_ldV_0.s						LHXV, LHZV, LFXV, LFZV
msp_stV_0.s						SHXV, SHZV, SFXV, SFZV
						[Interlock Tests]
msp_stall_0.s		x				VU Inst stall followed by a jump
msp_stall_1.s		x				VU Inst stall followed by a taken branch
msp_stall_2.s		x				VU Inst stall followed by a not taken branch
msp_stall_3.s		x				Stall in a jump dly slot
msp_stall_4.s		x				Stall in a taken branch dly slot
msp_stall_5.s		x				Stall in a not taken branch dly slot
msp_stall_6.s		x				Multiple exceptions
						Computational tests that test out corner cases
add1.s		x		x	x	MSP - ADD
add2.s		x		x	x	MSP - ADD
msp_add3.s		x				MSP - ADD
addi1.s		x		x	x	MSP - ADDI
addi2.s		x		x	x	MSP - ADDI
msp_addi3.s		x				MSP - ADDI
addiu1.s		x		x	x	MSP - ADDIU
addiu2.s		x		x	x	MSP - ADDIU
msp_addiu3.s		x				MSP - ADDIU
addu1.s		x		x	x	MSP - ADDU
addu2.s		x		x	x	MSP - ADDU
msp_addu3.s		x				MSP - ADDU
and1.s		x		x	x	MSP - AND
and2.s		x		x	x	MSP - AND
andi1.s		x		x	x	MSP - ANDI
beq1.s		x		x	x	MSP - BEQ
bgez1.s		x		x	x	MSP - BGEZ
bgezal1.s		x		x	x	MSP - BGEZAL
bgezal2.s		x		x	x	MSP - BGEZAL
bgtz1.s		x		x	x	MSP - BGTZ
blez1.s		x		x	x	MSP - BLEZ

TABLE 84. MSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
bltz1.s		x		x	x	MSP - BLTZ
bltzal1.s		x		x	x	MSP - BLTZAL
bltzal2.s		x		x	x	MSP - BLTZAL
bne1.s		x		x	x	MSP - BNE
bpmult.s		x		x	x	MSP - Bypass tests
bptest0.s		x		x	x	MSP - Bypass tests
bptest1.s		x		x	x	MSP - Bypass tests
bptest2.s		x		x	x	MSP - Bypass tests
bptest3.s		x		x	x	MSP - Bypass tests
bptest4.s		x		x	x	MSP - Bypass tests
cfc21.s		x		x	x	MSP - CFC2
cfc22.s		x		x	x	MSP - CFC2
ctc21.s		x		x	x	MSP - CTC2
ctc22.s		x		x	x	MSP - CTC2
di_ctlhz000.s		x		x	x	MSP - Dual Issue, Control Hzd
di_ctlhz001.s		x		x	x	MSP - Dual Issue, Control Hzd
di_ctlhz002.s		x		x	x	MSP - Dual Issue, Control Hzd
di_ctlhz010.s		x		x	x	MSP - Dual Issue, Control Hzd
di_ctlhz011.s		x		x	x	MSP - Dual Issue, Control Hzd
di_ctlhz012.s		x		x	x	MSP - Dual Issue, Control Hzd
di_ctlhz100.s		x		x	x	MSP - Dual Issue, Control Hzd
di_ctlhz101.s		x		x	x	MSP - Dual Issue, Control Hzd
di_ctlhz102.s		x		x	x	MSP - Dual Issue, Control Hzd
di_ctlhz110.s		x		x	x	MSP - Dual Issue, Control Hzd
di_ctlhz111.s		x		x	x	MSP - Dual Issue, Control Hzd
di_ctlhz112.s		x		x	x	MSP - Dual Issue, Control Hzd
di_ldst00.s		x		x	x	MSP - Dual Issue, Load Store
di_ldst01.s		x		x	x	MSP - Dual Issue, Load Store
di_ldst02.s		x		x	x	MSP - Dual Issue, Load Store
di_ldst03.s		x		x	x	MSP - Dual Issue, Load Store
di_ldst10.s		x		x	x	MSP - Dual Issue, Load Store
di_ldst11.s		x		x	x	MSP - Dual Issue, Load Store
di_ldst12.s		x		x	x	MSP - Dual Issue, Load Store

TABLE 84. MSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
di_ldst13.s		x		x	x	MSP - Dual Issue, Load Store
di_ldst20.s		x		x	x	MSP - Dual Issue, Load Store
di_ldst21.s		x		x	x	MSP - Dual Issue, Load Store
di_ldst22.s		x		x	x	MSP - Dual Issue, Load Store
di_ldst23.s		x		x	x	MSP - Dual Issue, Load Store
di_ldst30.s		x		x	x	MSP - Dual Issue, Load Store
di_ldst31.s		x		x	x	MSP - Dual Issue, Load Store
di_ldst32.s		x		x	x	MSP - Dual Issue, Load Store
di_ldst33.s		x		x	x	MSP - Dual Issue, Load Store
di_norm00.s		x		x	x	MSP - Dual Issue, normal flow
di_norm01.s		x		x	x	MSP - Dual Issue, normal flow
di_norm02.s		x		x	x	MSP - Dual Issue, normal flow
di_norm03.s		x		x	x	MSP - Dual Issue, normal flow
di_norm10.s		x		x	x	MSP - Dual Issue, normal flow
di_norm11.s		x		x	x	MSP - Dual Issue, normal flow
di_norm12.s		x		x	x	MSP - Dual Issue, normal flow
di_norm13.s		x		x	x	MSP - Dual Issue, normal flow
di_norm20.s		x		x	x	MSP - Dual Issue, normal flow
di_norm21.s		x		x	x	MSP - Dual Issue, normal flow
di_norm22.s		x		x	x	MSP - Dual Issue, normal flow
di_norm23.s		x		x	x	MSP - Dual Issue, normal flow
di_norm30.s		x		x	x	MSP - Dual Issue, normal flow
di_norm31.s		x		x	x	MSP - Dual Issue, normal flow
di_norm32.s		x		x	x	MSP - Dual Issue, normal flow
di_norm33.s		x		x	x	MSP - Dual Issue, normal flow
di_reghz0.s		x		x	x	MSP - Dual Issue
di_reghz1.s		x		x	x	MSP - Dual Issue
di_reghz2.s		x		x	x	MSP - Dual Issue
di_reghz3.s		x		x	x	MSP - Dual Issue
iltest1.s		x		x	x	MSP - Interlock tests
iltest10.s		x		x	x	MSP - Interlock tests
iltest11.s		x		x	x	MSP - Interlock tests
iltest12.s		x		x	x	MSP - Interlock tests

TABLE 84. MSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
iltest13.s		x		x	x	MSP - Interlock tests
iltest14.s		x		x	x	MSP - Interlock tests
iltest15.s		x		x	x	MSP - Interlock tests
iltest16.s		x		x	x	MSP - Interlock tests
iltest17.s		x		x	x	MSP - Interlock tests
iltest18.s		x		x	x	MSP - Interlock tests
iltest19.s		x		x	x	MSP - Interlock tests
iltest2.s		x		x	x	MSP - Interlock tests
iltest20.s		x		x	x	MSP - Interlock tests
iltest21.s		x		x	x	MSP - Interlock tests
iltest22.s		x		x	x	MSP - Interlock tests
iltest23.s		x		x	x	MSP - Interlock tests
iltest24.s		x		x	x	MSP - Interlock tests
iltest25.s		x		x	x	MSP - Interlock tests
iltest3.s		x		x	x	MSP - Interlock tests
iltest4.s		x		x	x	MSP - Interlock tests
iltest5.s		x		x	x	MSP - Interlock tests
iltest6.s		x		x	x	MSP - Interlock tests
iltest7.s		x		x	x	MSP - Interlock tests
iltest8.s		x		x	x	MSP - Interlock tests
iltest9.s		x		x	x	MSP - Interlock tests
j1.s		x		x	x	MSP - J
jal1.s		x		x	x	MSP - JAL
jalr1.s		x		x	x	MSP - JALR
jr1.s		x		x	x	MSP - JR
lav1.s		x		x	x	MSP - LAV
lb1.s		x		x	x	MSP - LB
lbu1.s		x		x	x	MSP - LBU
lbv1.s		x		x	x	MSP - LBV
lbv2.s		x		x	x	MSP - LBV
ldv1.s		x		x	x	MSP - LDV
ldv2.s		x		x	x	MSP - LDV
lfv1.s		x		x	x	MSP - LFV

TABLE 84. MSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
lh1.s		x		x	x	MSP - LH
lhu1.s		x		x	x	MSP - LHU
lhv1.s		x		x	x	MSP - LHV
lhlv1.s		x				MSP - LHLV
lhxv1.s		x				MSP - LHXV
llv1.s		x		x	x	MSP - LLV
llv2.s		x		x	x	MSP - LLV
lpv1.s		x		x	x	MSP - LPV
lpv2.s		x		x	x	MSP - LPV
lqv1.s		x		x	x	MSP - LQV
lqv2.s		x		x	x	MSP - LQV
lrv1.s		x		x	x	MSP - LRV
lrv2.s		x		x	x	MSP - LRV
lsv1.s		x		x	x	MSP - LSV
lsv2.s		x		x	x	MSP - LSV
ltv1.s		x		x	x	MSP - LTWV
lui1.s		x		x	x	MSP - LUI
luv1.s		x		x	x	MSP - LUV
luv2.s		x		x	x	MSP - LUV
lw1.s		x		x	x	MSP - LW
lxv1.s		x		x	x	MSP - LXV
lxv2.s		x		x	x	MSP - LXV
lzv1.s		x		x	x	MSP - LZV
lzv2.s		x		x	x	MSP - LZV
mfc21.s		x		x	x	MSP - MFC2
mfc22.s		x		x	x	MSP - MFC2
mfc23.s		x		x	x	MSP - MFC2
mfc24.s		x		x	x	MSP - MFC2
mfc25.s		x		x	x	MSP - MFC2
mfc26.s		x		x	x	MSP - MFC2
mtc21.s		x		x	x	MSP - MTC2
mtc22.s		x		x	x	MSP - MTC2
mtc23.s		x		x	x	MSP - MTC2

TABLE 84. MSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
mtc24.s		x		x	x	MSP - MTC2
mtc25.s		x		x	x	MSP - MTC2
mtc26.s		x		x	x	MSP - MTC2
nor1.s		x		x	x	MSP - NOR
nor2.s		x		x	x	MSP - NOR
or1.s		x		x	x	MSP - OR
or2.s		x		x	x	MSP - OR
ori1.s		x		x	x	MSP - ORI
sav1.s		x		x	x	MSP - SAV
sav2.s		x		x	x	MSP - SAV
sb1.s		x		x	x	MSP - SB
sbv1.s		x		x	x	MSP - SBV
sbv2.s		x		x	x	MSP - SBV
sbv3.s		x		x	x	MSP - SBV
sbv4.s		x		x	x	MSP - SBV
sdv1.s		x		x	x	MSP - SDV
sdv2.s		x		x	x	MSP - SDV
sdv3.s		x		x	x	MSP - SDV
sdv4.s		x		x	x	MSP - SDV
sfv1.s		x		x	x	MSP - SFV
sfv2.s		x		x	x	MSP - SFV
sh1.s		x		x	x	MSP - SH
shv1.s		x		x	x	MSP - SHV
shv2.s		x		x	x	MSP - SHV
sll1.s		x		x	x	MSP - SLL
sllv1.s		x		x	x	MSP - SLLV
sllv2.s		x		x	x	MSP - SLLV
slt1.s		x		x	x	MSP - SLT
slt2.s		x		x	x	MSP - SLT
slt3.s		x				MSP - SLT
slti1.s		x		x	x	MSP - SLTI
slti2.s		x		x	x	MSP - SLTI
slti3.s		x				MSP - SLTI

TABLE 84. MSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
sltiu1.s		x		x	x	MSP - SLTIU
sltiu2.s		x		x	x	MSP - SLTIU
sltiu3.s		x				MSP - SLTIU
sltu1.s		x		x	x	MSP - SLTU
sltu2.s		x		x	x	MSP - SLTU
sltu3.s		x				MSP - SLTU
slv1.s		x		x	x	MSP - SLV
slv2.s		x		x	x	MSP - SLV
slv3.s		x		x	x	MSP - SLV
slv4.s		x		x	x	MSP - SLV
spv1.s		x		x	x	MSP - SPV
spv2.s		x		x	x	MSP - SPV
spv3.s		x		x	x	MSP - SPV
spv4.s		x		x	x	MSP - SPV
sqv1.s		x		x	x	MSP - SQV
sqv2.s		x		x	x	MSP - SQV
sqv3.s		x		x	x	MSP - SQV
sqv4.s		x		x	x	MSP - SQV
sra1.s		x		x	x	MSP - SRA
srav1.s		x		x	x	MSP - SRAV
srav2.s		x		x	x	MSP - SRAV
srl1.s		x		x	x	MSP - SRL
srlv1.s		x		x	x	MSP - SRLV
srlv2.s		x		x	x	MSP - SRLV
srv1.s		x		x	x	MSP - SRV
srv2.s		x		x	x	MSP - SRV
srv3.s		x		x	x	MSP - SRV
srv4.s		x		x	x	MSP - SRV
ssv1.s		x		x	x	MSP - SSV
ssv2.s		x		x	x	MSP - SSV
ssv3.s		x		x	x	MSP - SSV
ssv4.s		x		x	x	MSP - SSV
stv1.s		x		x	x	MSP - STV

TABLE 84. MSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
sub1.s		x		x	x	MSP - SUB
sub2.s		x		x	x	MSP - SUB
sub3.s		x				MSP - SUB
subu1.s		x		x	x	MSP - SUBU
subu2.s		x		x	x	MSP - SUBU
subu3.s		x				MSP - SUBU
suv1.s		x		x	x	MSP - SUV
suv2.s		x		x	x	MSP - SUV
suv3.s		x		x	x	MSP - SUV
suv4.s		x		x	x	MSP - SUV
sw1.s		x		x	x	MSP - SW
swv1.s		x		x	x	MSP - SWV
sxv1.s		x		x	x	MSP - SXV
sxv2.s		x		x	x	MSP - SXV
sxv3.s		x		x	x	MSP - SXV
sxv4.s		x		x	x	MSP - SXV
xor1.s		x		x	x	MSP -X OR
xor2.s		x		x	x	MSP - XOR
xori1.s		x		x	x	MSP -X ORI
vabs_h		x		x	x	MSP - vabs
vabs_q		x		x	x	MSP - vabs
vabs_v		x		x	x	MSP - vabs
vabs_w		x		x	x	MSP - vabs
vacc_h		x		x		MSP - vacc
vacc_q		x		x		MSP - vacc
vacc_v		x		x		MSP - vacc
vacc_w		x		x		MSP - vacc
vaccb_el0		x		x		MSP - vaccb
vaccb_el1		x		x		MSP - vaccb
vaccb_el2		x		x		MSP - vaccb
vaccb_el3		x		x		MSP - vaccb
vaccb_el4		x		x		MSP - vaccb
vaccb_el5		x		x		MSP - vaccb

TABLE 84. MSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
vaccb_el6		x		x		MSP - vaccb
vaccb_el7		x		x		MSP - vaccb
vadd_h		x		x	x	MSP -vadd
vadd_q		x		x	x	MSP - vadd
vadd_v		x		x	x	MSP - vadd
vadd_w		x		x	x	MSP - vadd
vaddb_el0		x		x		MSP - vaddb
vaddb_el1		x		x		MSP - vaddb
vaddb_el2		x		x		MSP - vaddb
vaddb_el3		x		x		MSP - vaddb
vaddb_el4		x		x		MSP - vaddb
vaddb_el5		x		x		MSP - vaddb
vaddb_el6		x		x		MSP - vaddb
vaddb_el7		x		x		MSP - vaddb
vaddc_h		x		x	x	MSP -vaddc
vaddc_q		x		x	x	MSP - vaddc
vaddc_v		x		x	x	MSP - vaddc
vaddc_w		x		x	x	MSP - vaddc
vch_v		x		x		MSP - vch
vcr_v		x		x		MSP - vcr
veq_h		x		x		MSP -veq
veq_q		x		x		MSP - veq
veq_v		x		x		MSP - veq
veq_w		x		x		MSP - veq
veq_dbl_v		x		x		MSP - veq_dbl
vge_h		x		x		MSP -vge
vge_q		x		x		MSP - vge
vge_v		x		x		MSP - vge
vge_w		x		x		MSP - vge
vge_dbl_v		x		x		MSP - vge_dbl
vand		x		x	x	MSP - vlog
vnand		x		x	x	MSP - vlog
vnor		x		x	x	MSP - vlog

TABLE 84. MSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
vor		x		x	x	MSP - vlog
vxnor		x		x	x	MSP - vlog
vxor		x		x	x	MSP - vlog
vlt_h		x		x		MSP -vlt
vlt_q		x		x		MSP - vlt
vlt_v		x		x		MSP - vlt
vlt_w		x		x		MSP - vlt
vlt_dbl_v		x		x		MSP - vlt_dbl
vmaccb_h		x		x		MSP - vmacb
vmaccb_q		x		x		MSP - vmacb
vmacb_v		x		x		MSP - vmacb
vmaccb_w		x		x		MSP - vmacb
vmsucb_h		x		x		MSP - vmsucb
vmsucb_q		x		x		MSP - vmsucb
vmsucb_v		x		x		MSP - vmsucb
vmsucb_w		x		x		MSP - vmsucb
vmacf_clamp		x		x		MSP -vmacf
vmacf_h		x		x		MSP -vmacf
vmacf_q		x		x		MSP - vmacf
vmacf_v		x		x		MSP - vmacf
vmacf_w		x		x		MSP - vmacf
vmacq_v		x		x		MSP - vmacq
vmacq_v1		x		x		MSP - vmacq
vmacq_v2		x		x		MSP - vmacq
vmacu_clamp		x		x		MSP -vmacu
vmacu_h		x		x		MSP -vmacu
vmacu_q		x		x		MSP - vmacu
vmacu_v		x		x		MSP - vmacu
vmacu_w		x		x		MSP - vmacu
vmadh_clamp		x		x		MSP -vmadh
vmadh_h		x		x		MSP -vmadh
vmadh_q		x		x		MSP - vmadh
vmadh_v		x		x		MSP - vmadh

TABLE 84. MSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
vmadh_v1		x		x		MSP - vmadh
vmadh_w		x		x		MSP - vmadh
vmadh1_h		x		x		MSP -vmadh1
vmadh1_q		x		x		MSP - vmadh1
vmadh1_v		x		x		MSP - vmadh1
vmadh1_w		x		x		MSP - vmadh1
vmadl_clamp		x		x		MSP -vmadl
vmadl_h		x		x		MSP -vmadl
vmadl_q		x		x		MSP - vmadl
vmadl_v		x		x		MSP - vmadl
vmadl_w		x		x		MSP - vmadl
vmadm_clamp		x		x		MSP -vmadm
vmadm_h		x		x		MSP -vmadm
vmadm_q		x		x		MSP - vmadm
vmadm_v		x		x		MSP - vmadm
vmadm_v1		x		x		MSP - vmadm
vmadm_w		x		x		MSP - vmadm
vmadn_clamp		x		x		MSP -vmadn
vmadn_h		x		x		MSP -vmadn
vmadn_q		x		x		MSP - vmadn
vmadn_v		x		x		MSP - vmadn
vmadn_v1		x		x		MSP - vmadn
vmadn_w		x		x		MSP - vmadn
vmrg_h		x		x		MSP - vmrg
vmrg_q		x		x		MSP - vmrg
vmrg_v		x		x		MSP - vmrg
vmrg_w		x		x		MSP - vmrg
vmudh_h		x		x		MSP - vmudh
vmudh_q		x		x		MSP - vmudh
vmudh_v		x		x		MSP - vmudh
vmudh_v1		x		x		MSP - vmudh
vmudh_w		x		x		MSP - vmudh
vmudh1_h		x		x		MSP - vmudh1

TABLE 84. MSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
vmudh1_q		x		x		MSP - vmudh1
vmudh1_v		x		x		MSP - vmudh1
vmudh1_w		x		x		MSP - vmudh1
vmudl_h		x		x		MSP - vmudl
vmudl_q		x		x		MSP - vmudl
vmudl_v		x		x		MSP - vmudl
vmudl_v1		x		x		MSP - vmudl
vmudl_w		x		x		MSP - vmudl
vmudm_h		x		x		MSP - vmudm
vmudm_q		x		x		MSP - vmudm
vmudm_v		x		x		MSP - vmudm
vmudm_v1		x		x		MSP - vmudm
vmudm_w		x		x		MSP - vmudm
vmudn_h		x		x		MSP - vmudn
vmudn_q		x		x		MSP - vmudn
vmudn_v		x		x		MSP - vmudn
vmudn_v1		x		x		MSP - vmudn
vmudn_w		x		x		MSP - vmudn
vmulb_h		x		x		MSP - vmulb
vmulb_q		x		x		MSP - vmulb
vmulb_v		x		x		MSP - vmulb
vmulb_w		x		x		MSP - vmulb
vmulbn_h		x		x		MSP - vmulbn
vmulbn_q		x		x		MSP - vmulbn
vmulbn_v		x		x		MSP - vmulbn
vmulbn_w		x		x		MSP - vmulbn
vmulf_h		x		x		MSP - vmulf
vmulf_q		x		x		MSP - vmulf
vmulf_v		x		x		MSP - vmulf
vmulf_v1		x		x		MSP - vmulf
vmulf_w		x		x		MSP - vmulf
vmulq_h		x		x		MSP - vmulq
vmulq_q		x		x		MSP - vmulq

TABLE 84. MSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
vmulq_v		x		x		MSP - vmulq
vmulq_v1		x		x		MSP - vmulq
vmulq_w		x		x		MSP - vmulq
vmulu_h		x		x		MSP - vmulu
vmulu_q		x		x		MSP - vmulu
vmulu_v		x		x		MSP - vmulu
vmulu_v1		x		x		MSP - vmulu
vmulu_w		x		x		MSP - vmulu
vne_h		x		x		MSP - vne
vne_q		x		x		MSP - vne
vne_v		x		x		MSP - vne
vne_w		x		x		MSP - vne
vne_dbl_v		x		x		MSP - vne_dbl
vrnd_h		x		x		MSP - vrnd
vrnd_q		x		x		MSP - vrnd
vrnd_v		x		x		MSP - vrnd
vrnd_w		x		x		MSP - vrnd
vrndn_h		x		x		MSP - vrndn
vrndn_q		x		x		MSP - vrndn
vrndn_v		x		x		MSP - vrndn
vrndn_v1		x				MSP - vrndn
vrndn_w		x		x		MSP - vrndn
vrndp_h		x		x		MSP - vrndp
vrndp_q		x		x		MSP - vrndp
vrndp_v		x		x		MSP - vrndp
vrndp_v1		x				MSP - vrndp
vrndp_w		x		x		MSP - vrndp
vsac		x		x		MSP - vsac
vsacb_el0		x		x		MSP - vsacb
vsacb_el1		x		x		MSP - vsacb
vsacb_el2		x		x		MSP - vsacb
vsacb_el3		x		x		MSP - vsacb
vsacb_el4		x		x		MSP - vsacb

TABLE 84. MSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
vsacb_el5		x		x		MSP - vsacb
vsacb_el6		x		x		MSP - vsacb
vsacb_el7		x		x		MSP - vsacb
vsad		x		x		MSP - vsad
vsaw		x		x	x	MSP - vsaw
vsub_h		x		x	x	MSP - vsub
vsub_q		x		x	x	MSP - vsub
vsub_v		x		x	x	MSP - vsub
vsub_w		x		x	x	MSP - vsub
vsubb_el0		x		x		MSP - vsubb
vsubb_el1		x		x		MSP - vsubb
vsubb_el2		x		x		MSP - vsubb
vsubb_el3		x		x		MSP - vsubb
vsubb_el4		x		x		MSP - vsubb
vsubb_el5		x		x		MSP - vsubb
vsubb_el6		x		x		MSP - vsubb
vsubb_el7		x		x		MSP - vsubb
vsubc_h		x		x	x	MSP - vsubc
vsubc_q		x		x	x	MSP - vsubc
vsubc_v		x		x	x	MSP - vsubc
vsubc_w		x		x	x	MSP - vsubc
vsucb_el0		x		x		MSP - vsucb
vsucb_el1		x		x		MSP - vsucb
vsucb_el2		x		x		MSP - vsucb
vsucb_el3		x		x		MSP - vsucb
vsucb_el4		x		x		MSP - vsucb
vsucb_el5		x		x		MSP - vsucb
vsucb_el6		x		x		MSP - vsucb
vsucb_el7		x		x		MSP - vsucb
vsuc_h		x		x		MSP - vsuc
vsuc_q		x		x		MSP - vsuc
vsuc_v		x		x		MSP - vsuc
vsuc_w		x		x		MSP - vsuc

TABLE 84. MSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
vsum		x		x		MSP - vsum
vsumb_el0		x		x		MSP - vsumb
vsumb_el1		x		x		MSP - vsumb
vsumb_el2		x		x		MSP - vsumb
vsumb_el3		x		x		MSP - vsumb
vsumb_el4		x		x		MSP - vsumb
vsumb_el5		x		x		MSP - vsumb
vsumb_el6		x		x		MSP - vsumb
vsumb_el7		x		x		MSP - vsumb
vsut_h		x		x	x	MSP - vsut
vsut_q		x		x	x	MSP - vsut
vsut_v		x		x		MSP - vsut
vsut_w		x		x		MSP - vsut
lav1		x		x	x	MSP - lav
lbv1		x		x	x	MSP - lbv
lbv2		x		x	x	MSP - lbv
ldv1		x		x	x	MSP - ldv
ldv2		x		x	x	MSP - ldv
lfv1		x		x	x	MSP - lfv
lhv1		x		x	x	MSP - lhv
llv1		x		x	x	MSP - llv
llv2		x		x	x	MSP - llv
lpv1		x		x	x	MSP - lpv
lpv2		x		x	x	MSP - lpv
lqv1		x		x	x	MSP - lqv
lqv2		x		x	x	MSP - lqv
lrv1		x		x	x	MSP - lrv
lrv2		x		x	x	MSP - lrv
lsv1		x		x	x	MSP - lsv
lsv2		x		x	x	MSP - lsv
ltv1		x		x	x	MSP - ltv
luv1		x		x	x	MSP - luv
luv2		x		x	x	MSP - luv

TABLE 84. MSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
lxv1		x		x	x	MSP - lxv
lxv2		x		x	x	MSP - lxv
lzv1		x		x	x	MSP - lzv
lzv2		x		x	x	MSP - lzv
sav1		x		x	x	MSP - sav
sav2		x		x	x	MSP - sav
sbv1		x		x	x	MSP - sbv
sbv2		x		x	x	MSP - sbv
sbv3		x		x	x	MSP - sbv
sbv4		x		x	x	MSP - sbv
sdv1		x		x	x	MSP - sdv
sdv2		x		x	x	MSP - sdv
sdv3		x		x	x	MSP - sdv
sdv4		x		x	x	MSP - sdv
sfv1		x		x	x	MSP - sfv
sfv2		x		x	x	MSP - sfv
shv1		x		x	x	MSP - shv
shv2		x		x	x	MSP - shv
slv1		x		x	x	MSP - slv
slv2		x		x	x	MSP - slv
ssv1		x		x	x	MSP - ssv
ssv2		x		x	x	MSP - ssv
ssv3		x		x	x	MSP - ssv
ssv4		x		x	x	MSP - ssv
spv1		x		x	x	MSP - spv
spv2		x		x	x	MSP - spv
spv3		x		x	x	MSP - spv
spv4		x		x	x	MSP - spv
sqv1		x		x	x	MSP - sqv
sqv2		x		x	x	MSP - sqv
sqv3		x		x	x	MSP - sqv
sqv4		x		x	x	MSP - sqv
srv1		x		x	x	MSP - srv

TABLE 84. MSP diag list

	BSP	MSP	R4K	C model	VHDL	Description
srv2		x		x	x	MSP - srv
srv3		x		x	x	MSP - srv
srv4		x		x	x	MSP - srv
stv1		x		x	x	MSP - stv
suv1		x		x	x	MSP - suv
suv2		x		x	x	MSP - suv
suv3		x		x	x	MSP - suv
suv4		x		x	x	MSP - suv
swv1		x		x	x	MSP - swv
sxv1		x		x	x	MSP - sxv
sxv2		x		x	x	MSP - sxv
sxv3		x		x	x	MSP - sxv
sxv4		x		x	x	MSP - sxv
lshft.msp(bias)		x		x		MSP - lzv, mtc2, sqv, vsub, lhlv
chroma_to_mem.s		x				MSP - lhxv,sqv
fdct.s		x				MSP - lqv, ltwv, sqv, stv, vadd, vmacf, vmudh, vmulf, vsub
idct.s		x				MSP - lqv,ltwv,sqv,stv,vadd,vmacf.s1,vmadh.s1,vmudh.s1,vmulf.s1, vrndn.i.s1,vrndp.i.s1,vsub
fast inverse DCT		x				MSP -
convolution		x				MSP -
luma_to_mem.s		x				MSP - lxv, sqv, lqv
me.s		x				MSP - vsacc, vsadc, vsum
prep_pixel.s		x				MSP -lqv, shv, suv
dct.msp		x		x		MSP - lqv, ltwv, sqv, stv, vadd, vmacf, vmudh, vmulf, vsub
trans.msp(format conv./extraction)		x		x		MSP - lhlv, lzv, shlv, sqv, szv, vselge, vsellt, vsub
inverse dct		x				MSP - lqv, ltwv, stv, vacc, vadd, vmacf, vmulf, vsub
inverse quantize		x				MSP - lqv, ltwv, swv, vmudh
transpose quantize data		x				MSP - lqv, ltwv, sqv
quo.msp(quantize)		x		x		MSP - lqv, sqv, vmulf
transpose data		x				MSP - ltwv, swv

TABLE 85. Acceptance test

	BSP	MSP	R4K	C model	VHDL	Description
debugger0	x	x	x			
debugger1	x	x	x			
debugger2	x	x	x			

Summary of Test List

	Total	Written	Not yet written	Pass VHDL	Not yet Simulated
HD	135	67	68	7	128
BSP	320	198	122	198	122
MSP	589	513	76	100	489
VICE	3	0	3	0	3
Chip TEST	?	0	?	0	0

D.2 Board debug

Sounds like JTAG is a must for board debug / test.

- Need JTAG requirement / spec.
-

D.3 Foundry test

There are several variables that would set test methodology:

5. The maximum number of test pattern
6. The maximum length of each pattern
7. Minimum coverage (ideally 100%).

The choices are :

1. FULL SCAN :
   pro : good testability, automated test generation.
   con : performance.
2. Partial SCAN :
   scan functions that require many cycles such as counter and use functional codes for the rest.
   pro : keeps length of test time within limit (Does VTI have one ?)
   con : need to develop the rest of the test. How long and how good (% coverage) ?
3. no SCAN, use functional codes :
   pro : no performance impact.
   con : how good and how long to develop one ?

For non scan test, fault analysis tool is needed to assist test development. And this would have to be done toward the end of the project when the whole chip is at gate level, i.e. all synthesis are done.

Other technic of improving test coverage and length is to use IDDQ measurement, that is stop the clock and measure the power supply current at certain time within each pattern. For this technic, another tool is needed such as CROSS-CHECK. Note that this technic requires ALL components to NOT have static current, or if it does, it need to be turned off during IDDQ measurement.

Also, using the internal instruction RAM will help in keeping the pattern within limit.

Bibliography

Cavanagh, Joseph J. F. 1984. "Digital Computer Arithmetic Design and Implementation." New York: McGraw-Hill Publishing Company. ISBN 0-07-010282-1. Ch. 2 Fixed-Point Addition and Subtraction, Ch. 3 Fixed Point Multiplication.

Eshraghian, Kamran and Niel H. E. Weste 1993. "Principles of CMOS VLSI Design." Reading, Massachusetts: Addison-Wesley Publishing Company. ISBN 0-201-53376-6. Ch. 8 CMOS Subsystem Design, Ch. 9 CMOS Subsystem Design Examples.

Hamacher, Carl V., Zvonko G. Vranesic and Safwat G. Zaky 1990. "Computer Organization." Third Edition. New York: McGraw-Hill Publishing Company. ISBN 0-07-025685-3. Ch. 7 Arithmetic.

Heinrich, Joe1993. "MIPS Microprocessor R4000 User's Manual." Englewood Cliffs, New Jersey: Prentice-Hall Inc. ISBN 0-13-105925-4. Ch 12 System Interface, Appendix A CPU Instruction Set Details.

Hennessy, John L. and David A. Patterson 1990. "Computer Architecture: A quantitative Approach." San Mateo, California: Morgan Kaufmann Publishers. ISBN 1-55860-069-8. Ch. 6 Pipelining, Ch. 7 Vector Processors, Appendix A: Computer Arithmetic.

Hennessy, John L. and David A. Patterson 1994. "Computer Organization and Design: The Hardware/Software Interface." San Mateo, California: Morgan Kaufmann Publishers. ISBN 1-55860-281-X. Ch. 4 Arithmetic for Computers, Ch. 5 The Processor: Datapath and Control, Ch. 6 Enhancing Performance with Pipelining.

Jack, Keith 1993. "Video Demystified." Solana Beach, Ca: HighText Publications Inc. ISBN 1-878707-09-4

Johnson, Mike 1991. "Superscalar Microprocessor Design." Englewood Cliffs, New Jersey: Prentice-Hall Inc. ISBN 0-13-875634-1.

Kermode, Roger G. July 1993. "Requirements for Real Time Digital Video Compression." MIT Media Laboratory, Entertainment & Informations Systems Group. Prepared for Digital Sight and Sound Division, Silicon Graphics Inc.

Koren, Israel 1993. "Computer Arithmetic Algorithms." Englewood Cliffs, New Jersey: Prentice-Hall Inc. ISBN 0-13-151952-2. Ch. 5 Fast Addition, Ch. 6 High Speed Multiplication.

Lewis, Rhys 1990. "Practical Digital Image Processing." Ch. chester, West Sussex, England: Ellis Horewood Limited. ISBN 0-13-683525-2. Ch. 5 Geometric Functions, Ch. 7 Transforming Image Representations.

Pennebaker, Willaim B. and Mitchell, Joan L. 1993. "JPEG Still Image Data Compression Standard" New Yourk: Van Nostrand Reinhold. ISBN 0-442-01272-1

Van Hook, Tim Dec. 1993. "MIPS Media Engine Sketch" Draft 0.1. Mountain View, California: Silicon Graphics Inc.

Watkinson, John 1990. "The Art of Digital Video." Jordan Hill, Oxford, Great Britain: Focal Press. ISBN 0-240-51287-1. Ch. 5 Advanced Digital Processing, Ch. 8 Digital Video Interconnects.