

AN ABSTRACT OF THE THESIS OF

David A. Zier for the degree of Master of Science in Electrical and Computer Engineering presented on April 22, 2004.

Title: Designing Multimedia Extensions for Configurable Processors

Redacted for privacy

Abstract approved: \_\_\_\_\_

Ben Lee

The purpose of this thesis is to explore the design of a multimedia extension Instruction Set Architecture (ISA) for a reconfigurable processor. An Extendable Multimedia Module (EM3) was designed as an optional module for X32V. X32V is a prototype configurable processor simulator developed at Oregon State University by John Mark Matson and Dr. Ben Lee. The EM3 ISA uses Single-Instruction Multiple-Data (SIMD) type instructions to improve the performance of multimedia applications on X32V such as MPEG-4.

Two benchmarks based on certain stages of MPEG-4 decompression were developed to test the initial performance enhancements of EM3. The results of these benchmark tests indicate a several fold improvement in clock cycles and the number of instructions executed. This improvement demonstrates the performance increase of X32V and illustrates the effectiveness of SIMD type multimedia extensions.

©Copyright by David A. Zier

April 22, 2004

All Rights Reserved

Designing Multimedia Extensions for Configurable Processors

by

David A. Zier

A THESIS

submitted to

Oregon State University

in partial fulfillment of  
the requirements for the  
degree of

Master of Science

Completed April 22, 2004  
Commencement June 2004

Master of Science thesis of David A. Zier presented on April 22, 2004.

APPROVED:

Redacted for privacy

---

Major Professor, representing Electrical and Computer Engineering

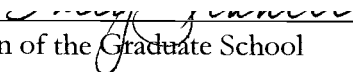
Redacted for privacy

---

Associate Director of the School of Electrical Engineering and Computer Science

Redacted for privacy

---

  
Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for privacy

---

David A. Zier, Author

## ACKNOWLEDGEMENTS

The author expresses sincere appreciation to the following individuals. Dr. Ben Lee for his mentorship and guidance. Jarrod Nelson for his help in the testing phase of EM3 and for his enthusiasm. Jumnit Hong for his help in the assembler support for EM3 and his dedication. Savrithi Venkatachalapathy for her support on the cross-compiler and her everlasting cheerfulness.

## TABLE OF CONTENTS

	<u>Page</u>
1 Introduction.....	1
2 Background.....	2
2.1 The X32V Reconfigurable Microprocessor.....	3
2.2 Single-Instruction Multiple-Data.....	4
2.3 Multimedia Extensions.....	5
3 Configurable Component Modules.....	8
3.1 Modular Support.....	9
3.2 Current Modules.....	10
4 The EM3 Instruction Set Architecture.....	11
4.1 Register Formats.....	12
4.2 SIMD Operational Types.....	13
4.2.1 One-to-One.....	13
4.2.2 One-to-Two.....	14
4.2.3 Two-to-One.....	15
4.3 Instruction Types.....	16
4.3.1 ALU Instructions.....	16
4.3.2 Multiply and Divide Instructions.....	19
4.3.3 Data Conversion Instructions.....	21
4.3.4 Data Movement Instructions.....	23
4.3.5 Special Instructions.....	24
5 Benchmarks and Results.....	26
5.1 Color Conversion Benchmark.....	27
5.2 Color Conversion Image Results.....	28
5.3 iDCT and Color Conversion Benchmark.....	31
5.4 Overall Performance Comparison.....	32
6 Future Work.....	34
7 Conclusion.....	34
Bibliography.....	36
Appendix.....	37
Appendix A: EM3 Instruction Set.....	38

## LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
1. X32V Architectural Diagram with Floating-Point and EM3 Modules.....	11
2. EM3 Register Formats .....	12
3. X32V Architectural Diagram with Floating-Point and EM3 Modules.....	13
4. One-to-Two Low Operation .....	14
5. One-to-Two High Operation .....	15
6. Two-to-One Operation .....	15
7. Two-to-One Interleaved Operations .....	16
8. SWAP Instruction Example .....	25
9. Color Conversion Equations .....	28
10. Original Color Conversion Benchmark Image .....	29
11. Integer Color Conversion Benchmark Image .....	29
12. EM3 Color Conversion Benchmark Image.....	30
13. Zoomed Color Conversion Benchmark Results .....	31
14. Benchmark Cycle Count Comparison .....	33

## LIST OF TABLES

<u>Figure</u>	<u>Page</u>
1. Addition Instructions .....	17
2. Subtraction Instructions .....	17
3. Logical Instructions .....	18
4. Arithmetic Shift Instructions .....	19
5. Multiply Instructions .....	20
6. Division Instructions .....	20
7. Pack Instructions .....	22
8. Unpack Instructions .....	22
9. Register Transfer Instructions .....	23
10. Memory Access Instructions .....	24
11. Cosine Values for iDCT .....	32
12. Benchmark Cycle Counts .....	33



For my Grandfather

Charles W. Guile

# Designing Multimedia Extensions for Configurable Processors

## 1 Introduction

Current trends in embedded applications are requiring more features and a shorter time-to-market. With the advent of System On Chip (SOC), the need to design a custom processor and secondary components has created enormous challenges for embedded system designers [11]. A synthesizable, configurable processor offers an effective way to improve time-to-market. A configurable processor allows the designer to create a custom microprocessor by configuring the processor core or adding specialized modules. These modules can provide the benefits of a coprocessor without the communication or area overhead usually encountered.

With the advent of MPEG-4 video compression and MP3 audio compression, media has taken a mainstream market in embedded applications such as portable MP3 players, miniature DVD players, and Smart Phones. With each of these devices, custom configurable processors speed the time-to-market, but it is the advantage of using multimedia extensions that give these processors the added performance increase.

This thesis discusses the development of a multimedia extension module for a configurable processor and demonstrates how it improves the performance of multimedia applications on the configurable processor. The Extendable MultiMedia Module (EM3) was developed for X32V, a prototype configurable processor developed at Oregon State under the guidance of Dr. Ben Lee. X32V is a variation of a simulated 5-stage pipeline that supports variable length instructions. Initially, the

pipeline had to be modified to support modules and then EM3 was developed and integrated into X32V.

The purpose of EM3 is to improve the performance of MPEG-4 benchmark on X32V. To evaluate the gain in performance, the simulated cycled counts were compared from two versions of each benchmark, an Integer benchmark and an EM3 benchmark. Due to a lack in compiler support for EM3 instructions, two benchmarks were created from hand-coded assembly and assembled using a PERL script. These benchmarks involved small portions of the MPEG-4 decompression code. It is clear from the results in Section 5, that there were significantly less number of cycles in the EM3 version of the benchmarks.

This thesis is organized as follows: Section 2 discusses the background information needed to explain the research for this thesis including X32V, SIMD, and multimedia extensions. Section 3 discusses the initial research and development by the author in order to include modular support in X32V. Section 4 covers the EM3 Instruction Set Architecture. Section 5 discusses the benchmarks that were created and the results of those benchmarks. And finally, Section 6 discusses the future work on EM3.

## **2 Background**

Over the course of developing EM3, several areas needed to be researched. Since EM3 is an extension to X32V, the developments of X32V needed to be studied as well as becoming familiar with the source code for the simulator. Since multimedia extensions primarily consist of Single-Instruction Multiple-Data (SIMD) type

instructions, the concept and current practices of SIMD operations are covered. And finally, an overview and comparison of current popular multimedia extensions were evaluated.

## ***2.1 The X32V Reconfigurable Microprocessor***

Currently there are several companies offering configurable and extensible microprocessor cores including Tensilica [12] and ARC Cores [9]. The eXpandable 32-bit Variable microcontroller, or X32V, offers 3-modes with various instruction lengths, which allows added flexibility when balancing memory requirements and cycle performance. At the core, X32V only supports an integer instruction set, which is useful in most situations, but X32V also provides extensive support for additional expansions through the use of add-on modules.

To prototype X32V and test the performance of various configurations, a cycle-accurate simulator was developed [7][13]. This simulator used the SimpleScalar Toolset as the base framework for simulating memory, system calls, cache, resources, and statistics gathering [3].

X32V supports three different modes of variable length instructions.

- Default (32-bit instructions)
- Light (32/16-bit instructions)
- Ultra-Light (32/24/16-bit instructions)

In default mode, a 32-bit instruction word is fetched from memory every clock cycle. In this mode, all the instructions are fetched on word aligned boundaries, thus

X32V operates normally without the additional overhead of fetching misaligned words. The default mode provides large immediate values and room for expansion.

Light mode can fetch both 32-bit and 16-bit instructions. The compiler will map any 32-bit instructions into their equivalent 16-bit formats if the instruction is compressible. All instructions are fetched 32-bits at a time, but in Light mode, a 32-bit instruction can land outside of a word boundary. This incurs a one-cycle penalty on miss-predicted branch since a 32-bit instruction can span across a memory word and the next 32-bit word needs to be fetched to get the complete instruction.

Ultra-light mode allows instructions to be fetched as 32-bit, 24-bit, or 16-bit instructions. The compiler will map any instructions that are compressible into 24-bit or 16-bit instructions. There can be up to a four-cycle penalty on a branch or jump that is made to a non-word aligned instruction in memory since there can potentially be three partial instructions in the fetched word. The benefit of the Ultra-light mode is that it provides the smallest program binaries and thus saves memory space.

## ***2.2 Single-Instruction Multiple-Data***

Single-Instruction Multiple-Data (SIMD) serves as the basic idea behind multimedia extensions. There is an inherent parallelism within multimedia datasets. Often data is associated into arrays of similar size, such as the RGB data used to signify the colors of pixels in digital images. The idea behind SIMD is to load multiple data elements into a single register and operate on all the data with a single instruction.

In order to perform an SIMD operation, the data set that is to be operated upon must be loaded into an SIMD register. Most SIMD architectures, if not all,

specify several formats that an SIMD register can support. The formats vary depending upon the architecture, but generally an SIMD register holds 1-16 members of the data set at a time. The format also specifies the type of data, for example, signed or unsigned, integer or floating point, etc. Typically the format does not need to be specified when loading the register, the format is determined when the SIMD operator is invoked. This allows the same register to be used for many different kinds of operations.

Once members of the data set are stored in an SIMD register, an SIMD operator may be applied on them. SIMD operations vary from processor to processor, but generally they all include basic arithmetic operations: logical operations (and, or, xor, etc.), shifts, addition, subtraction, and multiplication. Many processors also include floating-point counterparts. These basic operations can also be augmented with helper operations for conditional branching based on the SIMD registers and for load/store operations. Some processors also include specialized operators for computer graphics and digital signal processing such as saturated multiplication, saturated addition and reciprocal square root.

### ***2.3 Multimedia Extensions***

Multimedia extensions are a specific application of SIMD operations that enhance the performance of multimedia applications such as MPEG-4, MP3, and 3D graphics accelerators. The idea behind multimedia extensions have been around since the early 1990's and was made popular by Intel's MMX technology. Since then, there

have been dozens of popular multimedia extensions. This section will discuss six such SIMD multimedia technologies; MMX, VIS, 3DNow!, SSE, SSE2, and AltiVec.

Intel's MultiMedia eXtension (MMX) was Intel's first foray into SIMD for consumer processors. First appearing in the i860 processor, it was later popularized in the Pentium processor where it became a regular member of the 32-bit Intel Architecture (IA32) [8]. MMX provides eight 64-bit SIMD registers (this was accomplished through sharing the existing IA32 registers), with four integer formats: eight 8-bit, four 16-bit, two 32-bit and one 64-bit. The MMX operators included add, subtract, multiply, and logical operators. MMX did not support division or floating-point operations. MMX's distinct disadvantage was that the MMX registers overlapped the regular IA32 floating-point registers, requiring programmers to separate their integer SIMD operations from their floating-point code in order to achieve any performance benefit. MMX also required the processor to be in "MMX mode" in order to perform SIMD operations, which made context switching between processes unnecessarily expensive.

The Visual Instruction Set (VIS) was introduced by Sun Microsystems in the UltraSPARC-I processor [10]. In many ways VIS was Sun's answer to Intel's MMX. Like MMX, VIS used the CPU's existing registers to perform SIMD operations. However, unlike MMX, VIS had 32 registers (vs. MMX's eight), and supported floating-point SIMD operations. VIS provides four integer formats and four floating-point formats: 8-bit, 16-bit, 32-bit and 64-bit. VIS operators included add, subtract, multiply, and logical operators for both integer and floating-point data sets. VIS does

not support division. VIS used the same registers for SIMD operations as the regular SPARC registers, so it also required programmers to separate their SIMD operations from their regular operations.

3DNow! is an extension to the MMX instruction set developed by AMD. 3DNow! implemented the full MMX instruction set, but added to it floating-point support. 3DNow! was first introduced in the AMD K6-2 processor family [1].

Streaming SIMD Extensions (SSE), developed by Intel for the Pentium III processor, is an extension to MMX and was intended to compete with AMD's 3DNow! extensions. SSE added eight new 128-bit SIMD registers that can support four 32-bit and eight 16-bit floating-point formats [2]. The Pentium 4 processor introduced SSE2, which added the capability for the eight registers in SSE to be formatted as two 64-bit floating-point values and new operators such as saturated addition, saturated multiplication, and reciprocal square root. Although SSE does not overlap the floating-point registers as MMX did, it still shares the same circuitry, so SIMD and regular floating-point could not be run in parallel within the processor's pipeline.

Altivec is an extension to the PowerPC architecture first introduced in the G4 processor by Motorola. Altivec is a true extension since the SIMD registers are completely separate from the regular PowerPC registers. Altivec consists of 32 128-bit SIMD registers that can be formatted as sixteen 8-bit, eight 16-bit, four 32-bit, two 64-bit and one 128-bit integer, or four 32-bit IEEE-754 floating-point values. Altivec features all of the usual SIMD operators plus floating-point division, saturated addition



and multiplication, square root and half-precision reciprocal square root, absolute value, average, and a variety of highly customized SIMD load/store operators to maximize memory efficiency. AltiVec has proven to be very useful as an SIMD extension set to an existing consumer processor. IBM has recently added AltiVec support to their PowerPC 970 processor.

### **3 Configurable Component Modules**

Initially, X32V was a prototype simulator that consisted of 5-stage pipeline, which supported variable length instructions in several different modes. One of the main goals for X32V was to provide support for modules that could be included at design time without significant modifications to the X32V simulator. Unfortunately, this feature had yet to be integrated, so adding modular support became the first priority in designing a multimedia extension library.

X32V now supports both floating-point and multimedia modules that have been developed as cycle-accurate simulator components and can be added or removed depending upon the processor's intended purpose [13]. The base simulator consists of several complex components that are needed to support the component modules. Some examples are a complex forwarding mechanism, support for multicycle functional units, a fetch unit capable of fetching misaligned instructions from program memory, and a memory unit that is capable of reading data of any size and alignment from data memory. Additionally, an interface was created to include configurable modules.

### ***3.1 Modular Support***

A module consists of additional hardware that can interpret a new set of instructions. Depending on the module, this hardware consists of an extension to the decoder, functional units that will execute the instruction, and possibly an additional register file that can hold the module specific register data. To simulate this hardware, additional source and header files were created that contained the cycle-accurate simulator components that prototyped these hardware modules. They were integrated into the simulator through the use of several interfaces shared by each module.

The basic underlying architecture of X32V is a 5-stage pipeline with several common data buses that are used for interacting between the stages, writing back data to the register files, and forwarding. One of the key features for modularity is the support of multi-cycle functional units. Since each functional unit shares the common data buses, it is very easy to include additional functional units.

In order for the instruction to know which functional unit receives it, the decoder must be extended to support the additional instructions. Initially, there will be paths down the decoder in which an instruction is not supported. This could happen when running a program where a particular module has not yet been included. If such a situation arises, the instruction will be treated as a No Operation Instruction, or NOP, and passed through to the pipeline. Unfortunately, this will give rise to programs that do not execute correctly, thus it is important to compile programs for a particular configuration of X32V.

In order to keep all of the functional units working seamlessly together, X32V maintains in-order execution through the use of a reservation shift register (RSR) [4]. The RSR simplifies scheduling by tracking the cycle on which any instruction will finish the execution stage. The output latency of the execution is used to mask the RSR and determine whether the instruction will interfere with other instructions currently being executed. Masking the RSR was modified slightly to not only mask the current output latency bit, but also mask against all the bits after the output latency bit. If there was a bit that was in the RSR after the current output latency bit, then an instruction that is already in flight will finish after the current instruction, thus the instruction will become out-of-order. By masking against all future bits, we can guarantee that the instruction will remain in order and insure that there will be no conflicts in the Memory stage.

### ***3.2 Current Modules***

Currently, there are only two modules that have been implemented and tested for performance: A floating-point module that supports both 32-bit and 64-bit floating-point numbers and the multimedia module (EM3) that performs SIMD type integer operations. Both of these modules share a common register file, which means that floating-point registers are aliased to the EM3 registers. Figure 1 illustrates the integration of both the floating-point and EM3 modules within the X32V pipeline. This integration is exactly how X32V is currently simulated. EM3 is further discussed in Section 4.

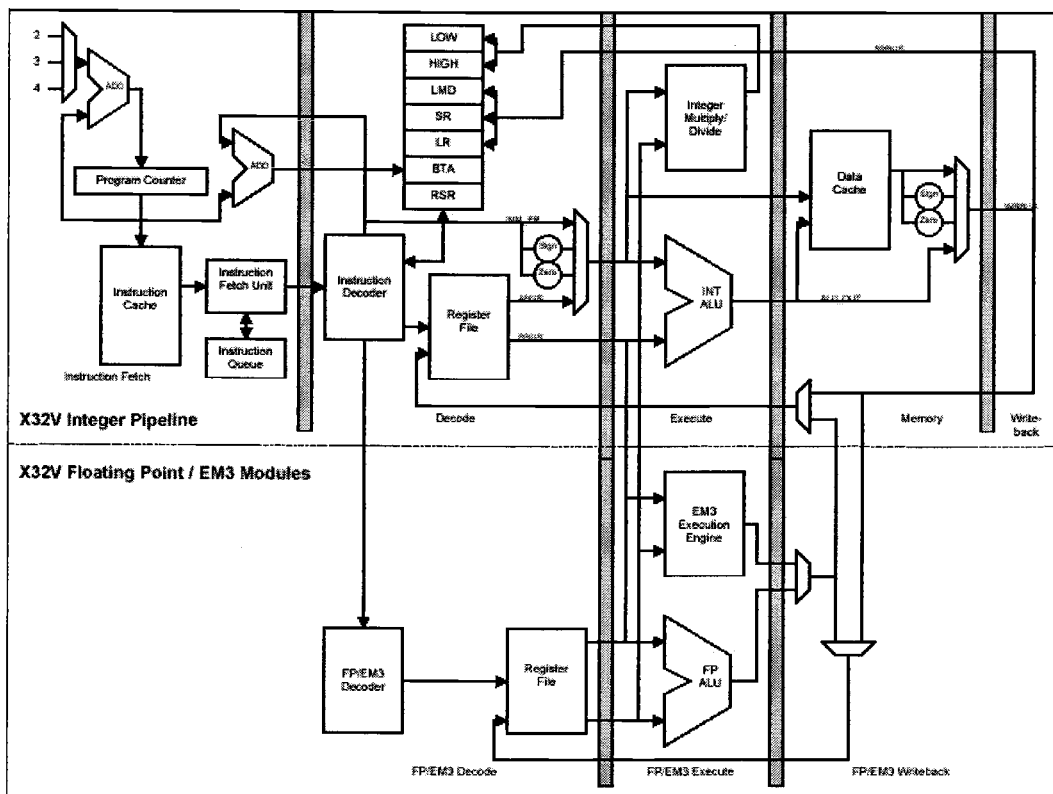


Figure 1: X32V Architectural Diagram with Floating-Point and EM3 Modules

## 4 The EM3 Instruction Set Architecture

This section discusses the Extendable MultiMedia Module (EM3) Instruction Set Architecture (ISA). The current ISA is a base set that was created from the combining the best features of popular SIMD multimedia architectures, such as those describe in Section 2.3. Currently, EM3 only supports integer SIMD operations using a variety of register formats (Section 4.1). Every SIMD operation can be thought of as a specific type of operation. These types of operations determine how the partitions in the register are operated on and are described in Section 4.2. The instructions can also be classified into specific types depending on which operation they execute (Section 4.3).

## 4.1 Register Formats

X32V supports various floating-point operations requiring a floating-point register file that consists of sixteen 32-bit floating-point registers. The general idea is to allow the EM3 multimedia registers (MMR) to alias the floating-point registers. EM3 supports multiple 8-bit, 16-bit, or 32-bit signed or unsigned instructions. Figure 2 illustrates the various data formats that are available. It is important to note that the 16-bit and 32-bit formats also support both signed and unsigned integer values where the sign bit is the most significant bit. Signed 8-bit integer data is not supported, since this form of data type is rarely used in multimedia applications.

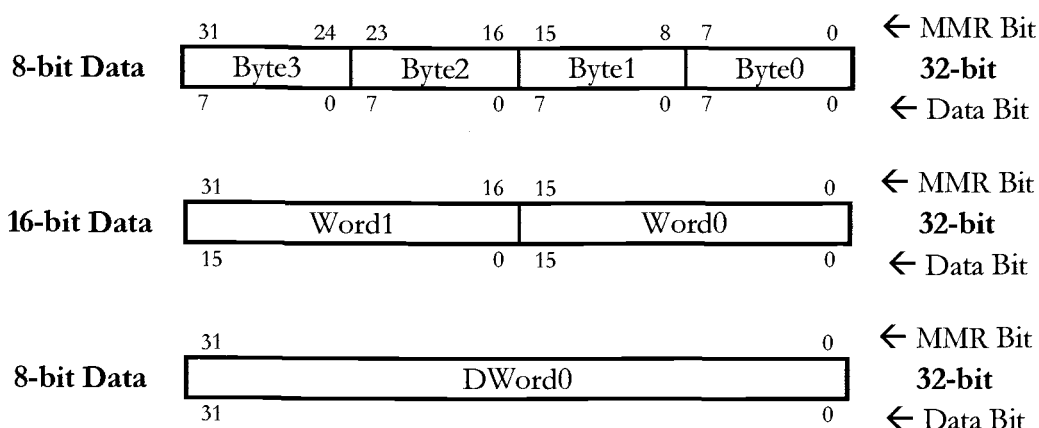


Figure 2: EM3 Register Formats

Additionally, the inclusion of operating on 64-bit multimedia registers could potentially increase the performance of EM3. This would be accomplished by performing multimedia operations on two contiguous MMRs. Using 64-bit registers will allow up to 8 operations per instruction.

## 4.2 SIMD Operational Types

When describing the functionality of the EM3 instructions, it will be easier to refer to the operational type of the SIMD instruction. There are three main types of SIMD operations; one-to-one operations, one-to-two operations, and two-to-one operations.

### 4.2.1 One-to-One

Every integer ALU SIMD operation can be described as a one-to-one operation. One-to-one operations occur when the result of an operation is the same size as the operands. This is the case for operations such as addition and subtraction.

Every two corresponding data elements from the source registers are operated on and the resulting element is stored in the corresponding destination register. Figure 3 illustrates this type of operation. A simple example would be the addition of two 32-bit words consisting of four 8-bit unsigned integer values. The addition would be very similar to a regular 32-bit addition with the exception that the ripple carry would stop on bits 7, 15, and 23.

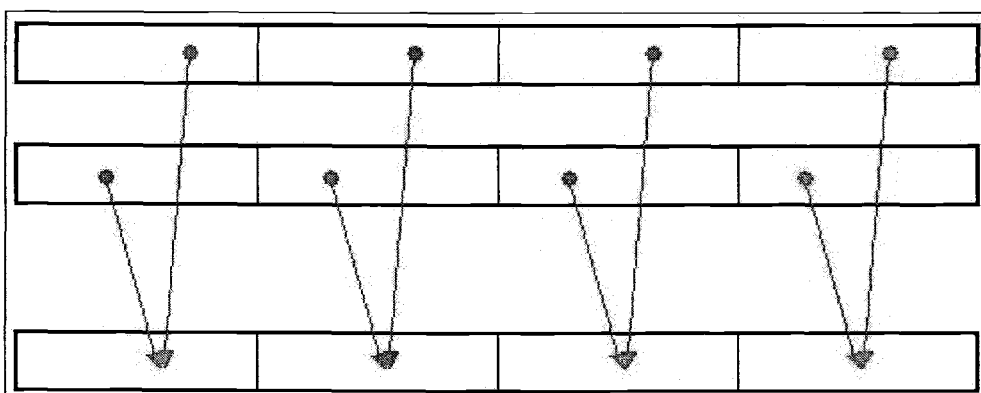


Figure 3: One-to-One Operation

### 4.2.2 One-to-Two

Both multiplication and unpacking operations are one-to-two operations since the resulting data that is produced will be twice the size of the original data. It is apparent that not all of the data segments can be operated on at once, since there is not enough room in the destination register to store the result. Therefore, any one-to-two operations will have two parts, a low and a high part. Figure 4 illustrates the one-to-two low operation where the lower segments of the source registers are operated on and the result is stored in the destination register.

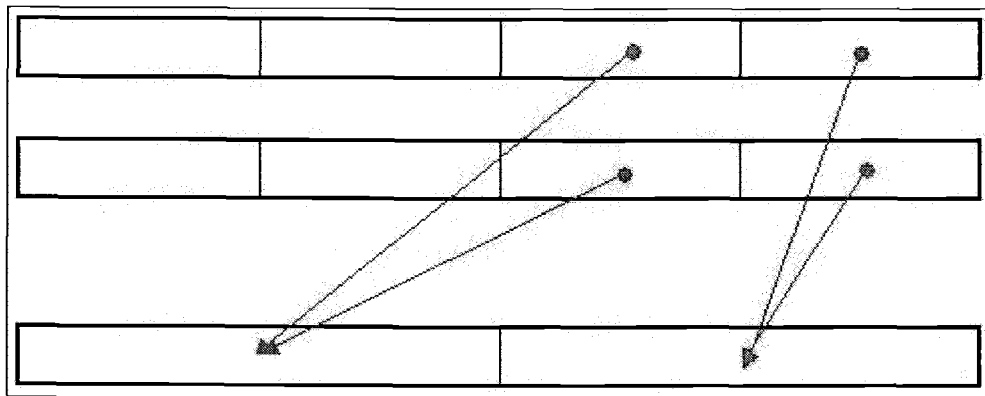


Figure 4: One-to-Two Low Operation

Figure 5 illustrates the one-to-two high operation where the upper halves of the source registers are operated on and the result is stored in the destination register. It is easy to see that if this type of operation was warranted for all the segments of the source registers, two instructions will be needed and the result will be contained into two registers.

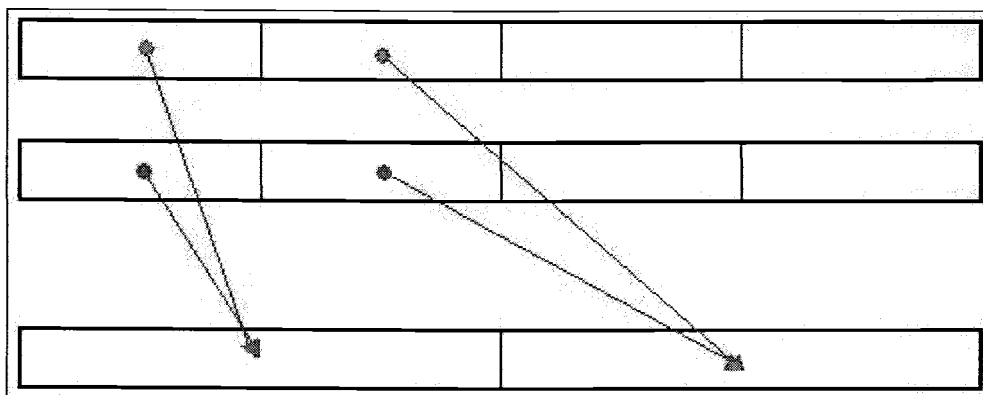


Figure 5: One-to-Two High Operation

#### 4.2.3 Two-to-One

Packing instructions compress the data size of a register by half of the original size. This type of operation is known as a two-to-one operation. The operand values are scaled and saturated to reduce the size, but there is no correlation between any of operands in either source register. Therefore, two source registers with four operands in total will be reduced to a single destination register. Figure 6 illustrates how a typical Two-to-One operation works.

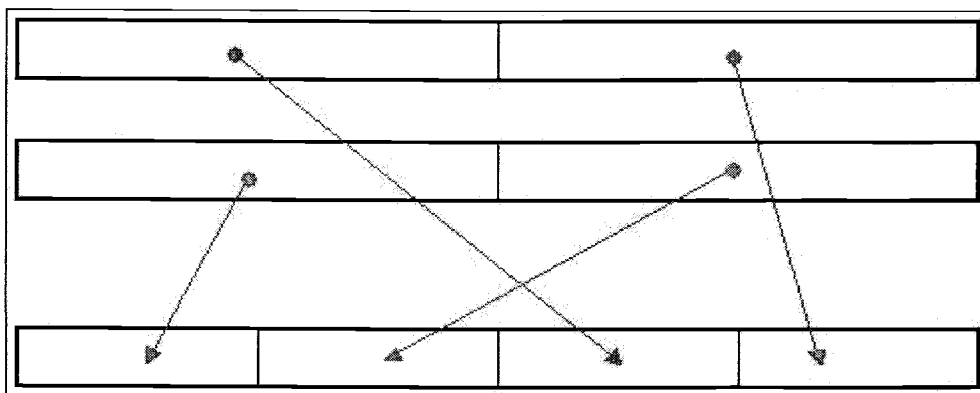


Figure 6: Two-to-One Operation



When the result can be represented as four elements, there is the option of interleaving the placement of the resulting values. Interleaving creates the ability to manipulate and reposition segments within a register. An example of where this would be useful is grabbing all the red color components from four RGBA color words.

Figure 7 illustrates the placement of two-to-one interleave operations.

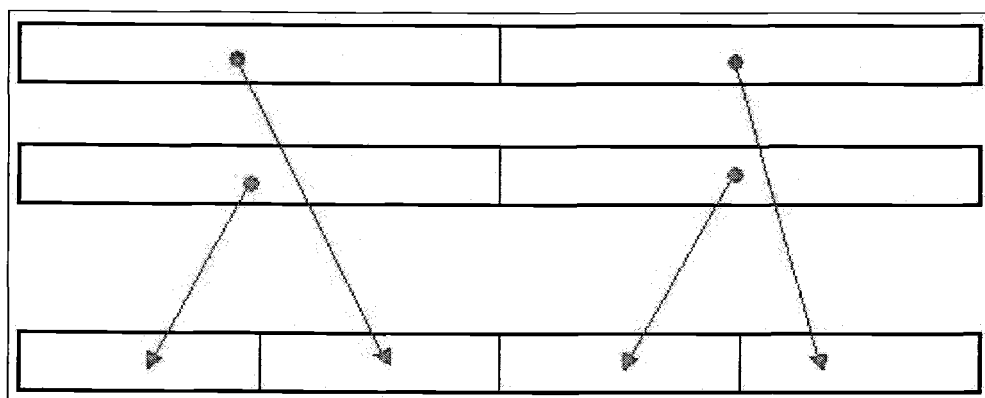


Figure 7: Two-to-One Interleaved Operations

### 4.3 *Instruction Types*

The EM3 instruction set can be thought of in terms of 5 different instruction types or classes; ALU Instructions, Multiplication/Division Instructions, Data Conversion Operations, Data Movement Instructions, and Special Instructions. The following subsections describe each instruction type in detail including which register formats are supported and the type of SIMD operation that is used.

#### 4.3.1 ALU Instructions

Instructions that fall into the category of ALU instructions involve all the basic arithmetic and logic instructions that would normally be in every integer ALU. These include but are not limited to addition, subtraction, AND, OR, XOR, NOT,

compliments, and arithmetic shifts. All of these instructions can be operated on all three EM3 register formats and assumes unsigned data. Additionally, the addition, subtraction, and compliment instructions can operate on 16- and 32-bit signed integer data.

#### 4.3.1.1 Addition Instructions

Addition instructions consist of one-to-one SIMD addition operations for all EM3 register formats, both signed and unsigned.

Table 1: Addition Instructions

<b><i>Mnemonic</i></b>	<b><i>Name</i></b>	<b><i>Format</i></b>	<b><i>Description</i></b>
ADD_8U	Addition	8-Bit	Adds 4 unsigned bytes
ADD_16	Addition	16-Bit	Adds 2 signed words
ADD_16U	Addition	16-Bit	Adds 2 unsigned words
ADD_32	Addition	32-Bit	Adds a signed double word
ADD_32U	Addition	32-Bit	Adds an unsigned double word

#### 4.3.1.2 Subtraction Instructions

Subtraction instructions consist of one-to-one SIMD subtraction operations for all EM3 register formats, both signed and unsigned.

Table 2: Subtraction Instructions

<b><i>Mnemonic</i></b>	<b><i>Name</i></b>	<b><i>Format</i></b>	<b><i>Description</i></b>
SUB_8U	Subtraction	8-Bit	Subtracts 4 unsigned bytes
SUB_16	Subtraction	16-Bit	Subtracts 2 signed words
SUB_16U	Subtraction	16-Bit	Subtracts 2 unsigned words
SUB_32	Subtraction	32-Bit	Subtracts a signed double word
SUB_32U	Subtraction	32-Bit	Subtracts an unsigned double word

#### 4.3.1.3 Logical Instructions

Logical instructions consist of one-to-one logic operations. These instructions are actually bitwise instructions; therefore, only one instruction is needed for all EM3 register formats. The idea for these instructions is to reduce additional cycle overhead from moving the data to the GPR and back again. The only exception to this is the 2's compliment instruction, which operates only on 16- and 32-bit signed operands.

Table 3: Logical Instructions

<i><b>Mnemonic</b></i>	<i><b>Name</b></i>	<i><b>Format</b></i>	<i><b>Description</b></i>
OR_8	Logical OR	8-Bit	Logical OR on 4 bytes
AND_8	Logical AND	8-Bit	Logical AND on 4 bytes
XOR_8	Logical XOR	8-Bit	Logical XOR on 4 bytes
NOT_8	Logical NOT	8-Bit	Logical NOT on 4 bytes
CPL_16	2's Compliment	16-Bit	Two's complement of 2 signed words
CPL_32	2's Compliment	32-Bit	Two's complement of a signed double word

#### 4.3.1.4 Arithmetic Shifts

Arithmetic shift instructions perform right and left arithmetic shifts. All data sizes are available, but use only the unsigned data formats. The values to be shifted are defined by the first source register, while the second source register specifies the shift constants. Each operand in the first source register can be shifted independently of one another. Future implementations could include the ability to add an immediate shift constant.

Table 4: Arithmetic Shift Instructions

<i><b>Mnemonic</b></i>	<i><b>Name</b></i>	<i><b>Format</b></i>	<i><b>Description</b></i>
SAR_8	Arithmetic Shift R	8-Bit	Arithmetic right shift on 4 bytes
SAR_16	Arithmetic Shift R	16-Bit	Arithmetic right shift on 2 words
SAR_32	Arithmetic Shift R	32-Bit	Arithmetic right shift on a double word
SAL_8	Arithmetic Shift L	8-Bit	Arithmetic left shift on 4 bytes
SAL_16	Arithmetic Shift L	16-Bit	Arithmetic left shift on 2 words
SAL_32	Arithmetic Shift L	32-Bit	Arithmetic left shift on a double word

### 4.3.2 Multiply and Divide Instructions

The multiplication and division operations are some of the most time consuming and complicated instructions in the EM3 instruction set. These instructions can operate on all the register formats and can produce different formats for the output. For example, multiplication done on two 16-bit values can result in 16-bit saturated, 32-bit, 16-bit high, or 16-bit low formats. Division can produce similar results.

Future additions to this category would be a 16x18-bit multiplication to increase the performance of MPEG-4 color-space conversion. An instruction that can multiply a single immediate value against all components within an EM3 register may also have a performance enhancement for some multimedia applications.

#### 4.3.2.1 *Multiplication Instructions*

Multiplication instructions support all EM3 register formats. Multiply is a one-to-two operation and requires two instructions for each register format. Additionally, there is a one-to-one multiply instruction that uses saturation to create a result of the

same size as the operands. For the 32-bit format, saturation is the only type of multiplication allowed since EM3 does not currently support any format above 32-bits.

Table 5: Multiply Instructions

<i><b>Mnemonic</b></i>	<i><b>Name</b></i>	<i><b>Format</b></i>	<i><b>Description</b></i>
MUL_8UL	Multiply	8-Bit	Unsigned Low Multiply
MUL_8UH	Multiply	8-Bit	Unsigned High Multiply
MUL_8US	Multiply	8-Bit	Unsigned Saturated Multiply
MUL_16L	Multiply	16-Bit	Signed Low Multiply
MUL_16H	Multiply	16-Bit	Signed High Multiply
MUL_16S	Multiply	16-Bit	Signed Saturated Multiply
MUL_16UL	Multiply	16-Bit	Unsigned Low Multiply
MUL_16UH	Multiply	16-Bit	Unsigned High Multiply
MUL_16US	Multiply	16-Bit	Unsigned Saturated Multiply
MUL_32US	Multiply	32-Bit	Unsigned Saturated Multiply
MUL_32S	Multiply	32-Bit	Signed Saturated Multiply

#### 4.3.2.2 Division Instructions

Divide is supported across all EM3 register formats and treated as a one-to-one divide operation. The result is not fractional and only contains the quotient part. This is an area of research to determine the effectiveness of the divide operation and to determine if any other divide operations will be more pertinent.

Table 6: Division Instructions

<i><b>Mnemonic</b></i>	<i><b>Name</b></i>	<i><b>Format</b></i>	<i><b>Description</b></i>
DIV_8U	Divide	8-Bit	Unsigned Divide
DIV_16U	Divide	16-Bit	Unsigned Divide
DIV_16	Divide	16-Bit	Signed Divide
DIV_32U	Divide	32-Bit	Unsigned Divide
DIV_32	Divide	32-Bit	Signed Divide

### 4.3.3 Data Conversion Instructions

Conversion operations are essential when handling SIMD data, since they allow data to be converted from different formats and allow the data to be rearranged within the registers. An example would be to convert two EM3 registers in 16-bit register format to a single EM3 register in 8-bit format. Being able to move between formats quickly and efficiently can improve both performance and precision when certain operations are required. There are two data conversion categories, pack and unpack. Pack allows data to be compressed into a smaller format and unpack will uncompress data into a larger format.

#### 4.3.3.1 *Pack Instructions*

Pack instructions are used to compress data into a format that is half the size as the original. Packing is considered to be a two-to-one operation, so the option for interleaving the results is allowed when packing 16-bit formatted data to 8-bit formatted data. The 8-bit pack is not implemented since there are no other supported formats smaller than 8-bit.

The compression process involves first scaling the data and then saturating the remaining result into the destination size. The scaling is accomplished by shifting the data to the right. The amount shifted depends on the precision required for the data format size.

Table 7: Pack Instructions

<i>Mnemonic</i>	<i>Name</i>	<i>Format</i>	<i>Description</i>
PCK_16	Pack Bits	16-Bit	Packs 4 words into 4 bytes
PCK_16S	Pack Bits	16-Bit	Packs 4 words into 4 bytes using interleaving
PCK_32	Pack Bits	32-Bit	Packs 2 double words into 2 logical words

#### 4.3.3.2 Unpack Instructions

Unpack instructions decompress data to a format that is double the original size. This is a one-to-two operation, therefore there are both high and low components. An example would be decompressing an 8-bit formatted register into a 16-bit format. The 32-bit unpack is not supported since formats that are larger than 32-bit are not supported.

The decompression process is similar to the pack instructions, except that the instructions are scaled up, which involves shifting the data to the left. The scaled data is then placed in the resulting register without further alteration. It is important to note that packing and unpacking data will result in some data loss. The scaling factor reduces the effect of this data loss.

Table 8: Unpack Instructions

<i>Mnemonic</i>	<i>Name</i>	<i>Format</i>	<i>Description</i>
UPCK_8L	Unpack Bits	8-Bit	Unpacks 2 low bytes into 2 words
UPCK_8H	Unpack Bits	8-Bit	Unpacks 2 high bytes into 2 words
UPCK_8SL	Unpack Bits	8-Bit	Unpacks 2 saturated low bytes into 2 words
UPCK_8SH	Unpack Bits	8-Bit	Unpacks 2 saturate high bytes into 2 words
UPCK_16L	Unpack Bits	16-Bit	Unpacks low word into a double word
UPCK_16H	Unpack Bits	16-Bit	Unpacks high word into a double word

#### 4.3.4 Data Movement Instructions

Data movement instructions allow data to be moved from the General Purpose Registers (GPR) to the Multimedia Registers (MMR), from MMR to GPR, and from MMR to MMR. These instructions do not provide any conversion capability, instead they are intended to move the data as is and without regard to any specific format. The format of the data will then be interpreted by the particular multimedia instruction.

Other instructions that fall in this category are the load and store instructions. In addition to allowing data to be moved directly from memory into the MMR, these load and store instructions also allow the data to be accessed by the format boundary.

##### 4.3.4.1 Register Transfer Instructions

Register transfer instructions move data from one register to another. The data format is not taken into consideration since this is not an SIMD instruction. These instructions allow X32V to transfer data between the Multimedia Register file (MMR) and the Integer Register file (GPR).

Table 9: Register Transfer Instructions

<i><b>Mnemonic</b></i>	<i><b>Name</b></i>	<i><b>Format</b></i>	<i><b>Description</b></i>
MOV_MM	Move	32-Bit	Move data between two MMR registers
MOV_IM	Move	32-Bit	Move data from GPR register to MMR register
MOV_MI	Move	32-Bit	Move data from MMR register to GPR register



#### 4.3.4.2 Memory Access Instructions

The EM3 memory instructions complement the integer memory instructions by allowing 32-bits at a time to be accessed from memory, regardless of the data type. The formats allow access to different data boundaries in the memory. For example, an 8-bit Load will load four 8-bit values at a byte boundary, but a 32-bit Load will load a single 32-bit value from a double word boundary.

Table 10: Memory Access Instructions

<i><b>Mnemonic</b></i>	<i><b>Name</b></i>	<i><b>Format</b></i>	<i><b>Description</b></i>
LD_8	Load	8-Bit	Loads 4 contiguous bytes of memory into MMR register
LD_16	Load	16-Bit	Loads 2 contiguous words of memory into MMR register
LD_32	Load	32-Bit	Loads a double word of memory into MMR register
ST_8	Store	8-Bit	Stores 4 contiguous bytes into memory from MMR register
ST_16	Store	16-Bit	Stores 2 contiguous words into memory from MMR register
ST_32	Store	32-Bit	Stores a double word into memory from MMR register

#### 4.3.5 Special Instructions

The special instructions category is left for certain multimedia instructions that do not clearly fall into any of the previous categories. Although not currently implemented, an instruction that performs pixel distance calculations (PDIST) would fall into this category. One special instruction that greatly enhanced the performance of EM3 is the SWAP instruction.

#### 4.3.5.1 SWAP Instruction

Currently, the only special instruction that is implemented is the SWAP instruction. This instruction allows the programmer to select four bytes from two EM3 registers, rearrange them, and store them in a single EM3 register. The format for the instruction is: SWAP fd, fs0, fs1, a, b, c, d. The first three operands signify the destination and source registers. The remaining four operands indicate which byte from the two source operands will be stored in particular byte of the destination register.

The idea behind the swap instruction is to quickly rearrange data within registers without the data loss that would occur from the pack and unpack instructions. A good use for the swap instruction in multimedia applications would be to separate pixel data and generate arrays of individual colors. Another good use is to quickly set up data for matrix multiplications. Figure 8 illustrates a swap operation.

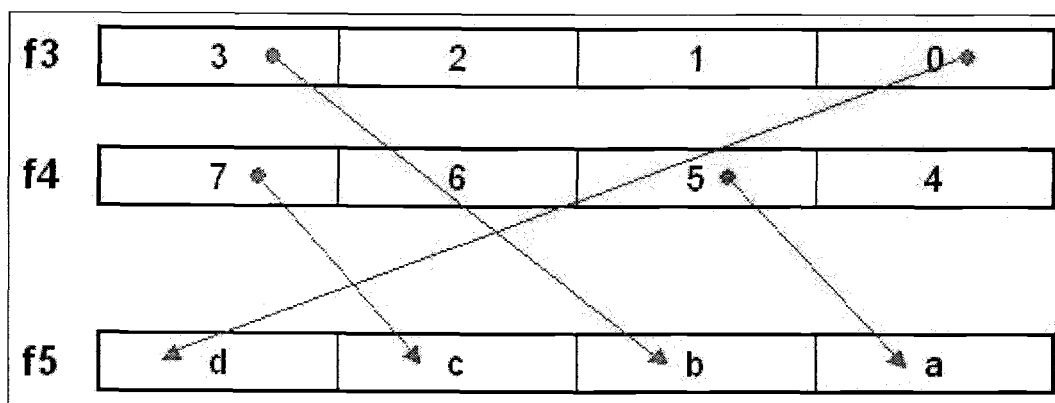


Figure 8: SWAP Instruction Example

#### 4.3.5.2 Future Special Instructions

There are several instructions that are currently being looked into that could produce a better performance gain out of MPEG-4. The following is a list of instructions as well as a brief explanation.

- *PDIST* – This instruction will perform a pixel distance calculation used in MPEG-4 compression.
- *MADD* – This instruction will multiply two registers and add the result to the data currently in the destination register. This will improve the performance of matrix multiplications and iDCT.
- *MATMUL* – This instruction performs a 4x4 matrix multiplication against a 1x4 vector represented in a single source register. The 4x4 matrix is represented by the registers F0-F3 and are statically tied to the instruction, so they do not need to be explicitly called. This allows a programmer to create a single matrix and then multiply it against several vectors.
- *Immediate Instructions* – Although not currently implement, additional support for immediate values will reduce the amount of time spent fetching constants from memory.

## 5 Benchmarks and Results

To test the performance of the EM3 instructions and the X32V in general, two benchmarks were created. Each benchmark has two versions, one version uses only the integer ISA and the other uses the EM3 multimedia extensions. These

benchmarks serve two purposes. The first purpose is to validate of the simulator, while the second purpose is to study the performance of MPEG-4 using EM3. The initial benchmark modeled the color conversion process of MPEG-4. The second benchmark modeled the Inverse Discrete Cosine Transform (iDCT) and color conversion process together.

### ***5.1 Color Conversion Benchmark***

The color conversion process was chosen as the first benchmark to model one portion of MPEG-4. Color conversion was an ideal benchmark since the performance was known to increase with the use of multimedia extensions and the resulting image data could be easily verified through visual inspection. The benchmark reads in macro blocks of YCbCr image data from an input file, converts them to interleaved RGB data and writes the results to an output file. To make the benchmark as much like MPEG-4 decoding as possible, sub-sampled macro blocks were used. For each 16x16 block of pixels, 128 Y samples, 64 Cb samples, and 64 Cr samples were taken. The sub-sampling was done by choosing the lower left pixel of each block of four pixels as the color value for that block. The reader will notice in Section 5.2 that the sub-sampling did introduce some distortion, but at the original resolution the effect is virtually unnoticeable.

To allow us to view the results, the benchmark writes the data to a file as a 24-bit color bitmap image. This image uses the standard Windows BMP file format that makes for easy viewing. Both benchmarks implement the equations in Figure 9 in order to perform the YCrCb to RGB color conversion.

$$\begin{aligned}
 R &= (149 \cdot Y) \gg 7 + (51 \cdot Cr) \gg 5 \\
 G &= (149 \cdot Y) \gg 7 + (208 \cdot Cr) \gg 8 - (100 \cdot Cb) \gg 8 \\
 B &= (149 \cdot Y) \gg 7 + (129 \cdot Cb) \gg 6
 \end{aligned}$$

Figure 9: Color Conversion Equations

Values are multiplied by integer constants and then shifted left to accomplish fractional multiplication. The integer version performs one pixel calculation at a time, where as the EM3 version calculates four pixel values per iteration.

In order to perform the test, a third program was created in standard C that would convert a test image from RGB format to YCrCb format. Both simulators used the test image in order to recreate the RGB image. The generated RGB images were compared to the original image for accuracy.

## 5.2 Color Conversion Image Results

In order to fully demonstrate the capabilities of the color conversion benchmark, a large high-contrast image of a popular video game was chosen. The original image has the dimensions of 640x480 pixels. Figure 10 is the original image that was first converted to YCrCb format using a standard C program and then used for the benchmark data. Figure 11 is the resulting image from the integer color conversion benchmark and Figure 12 is the resulting image from the EM3 color conversion benchmark. By visually comparing the full size images, no differences can be discerned.

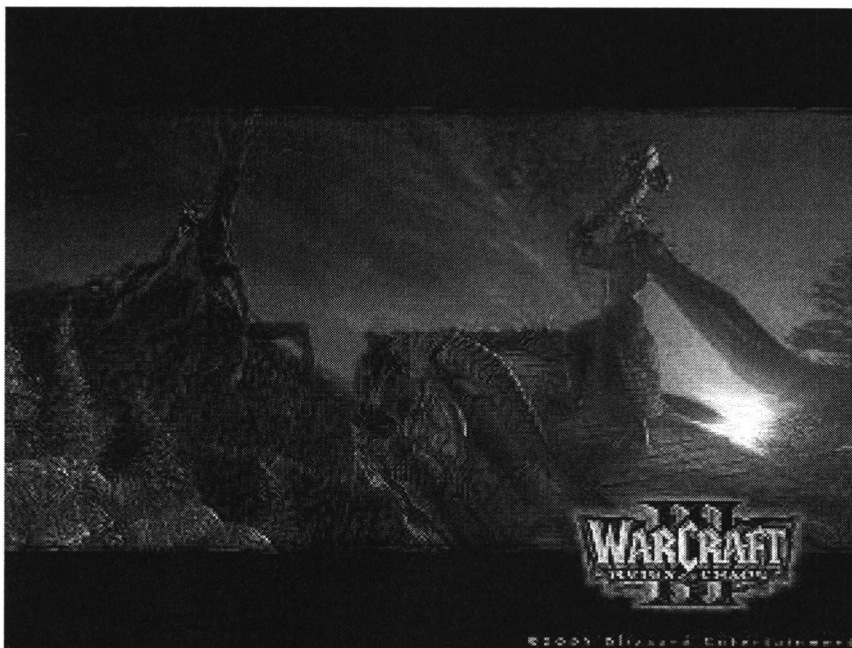


Figure 10: Original Color Conversion Benchmark Image

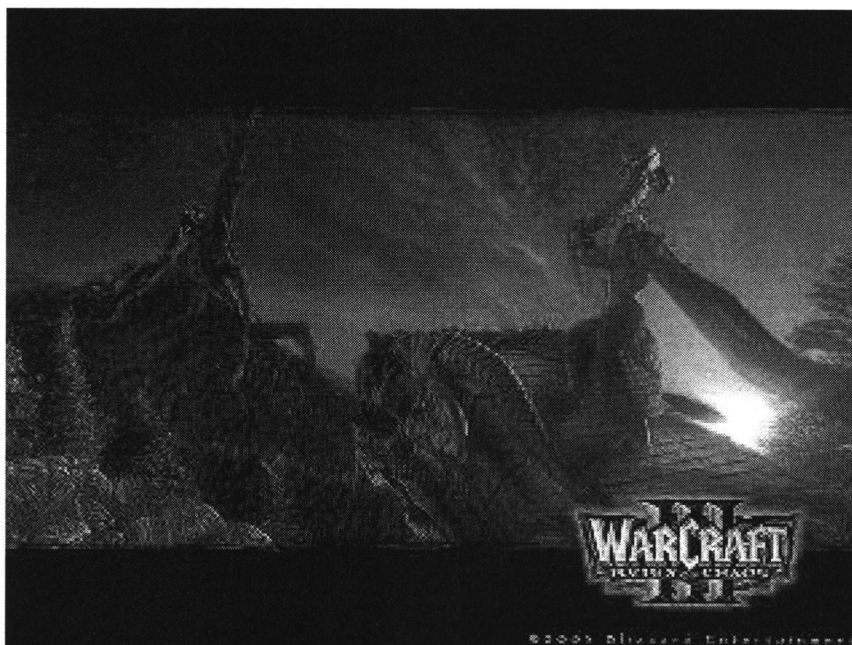


Figure 11: Integer Color Conversion Benchmark Image

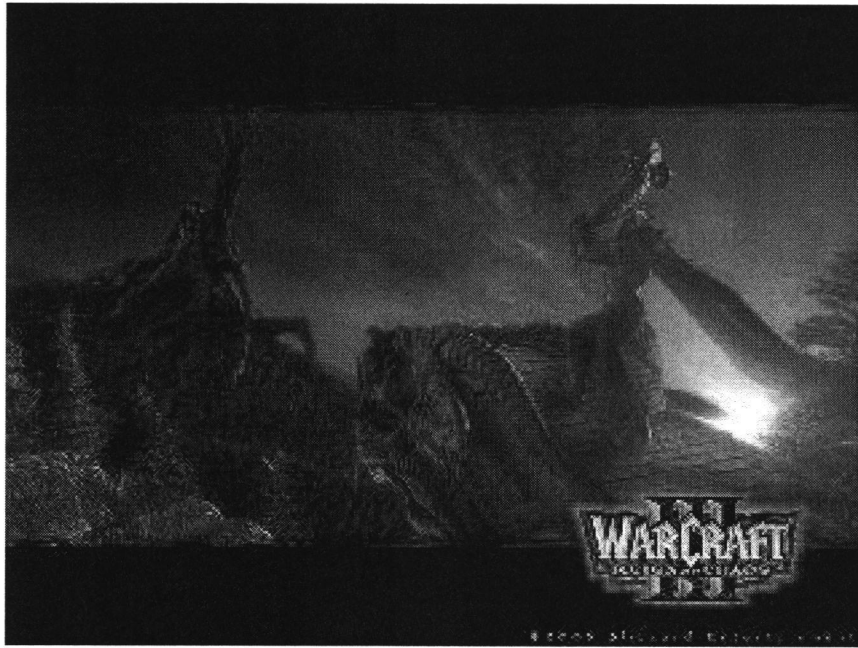


Figure 12: EM3 Color Conversion Benchmark Image

The differences in the result only show up after closely examining the zoomed in sections of the images. From Figure 13, it is obvious that both the Integer and EM3 results differ from the original. This is due to the sub-sampling procedure of the color conversion process. The most important thing to notice is that both the Integer and EM3 results are identical. In fact, by using software to compare the binary output of both resulting images, the two images were found to be identical.

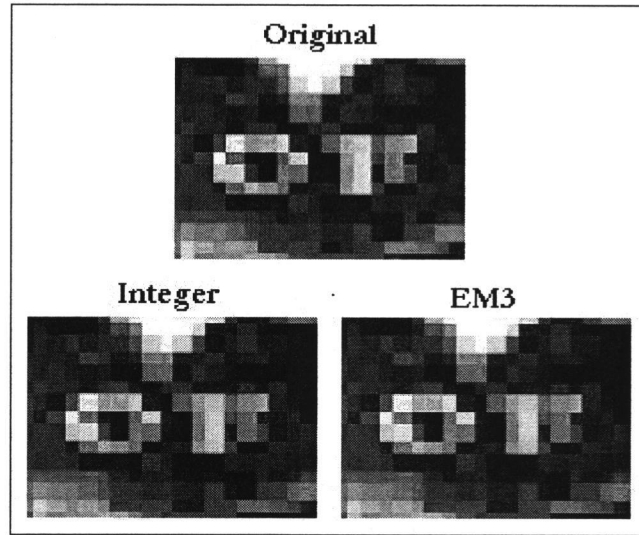


Figure 13: Zoomed Color Conversion Benchmark Results

Since both resulting images are identical and similar to the original, we can now emphasize in how EM3 will improve the performance without degrading the image. Section 5.4 will cover more detail about the resulting numbers and performance gain.

### 5.3 *iDCT and Color Conversion Benchmark*

After completing the color conversion benchmark, the next step was to tackle the Inverse Discrete Cosine Transformation (iDCT). This operation requires a large number of multiplication and addition instructions for 16-bit values. Within the benchmark, the 2-D iDCT required is performed as a 1-D iDCT on all the rows of all the 8x8 sub-blocks followed by a 1-D iDCT on all the columns of all the 8x8 sub-blocks. This method breaks up the amount of data required for calculating each step and can thus be fit into the limited number of registers of X32V.

The cosine values used by the iDCT were calculated before hand, and used as constants. To maintain the required accuracy we used 7 fractional bits in our cosine



values and kept 4 fractional bits for all our intermediate results. These fractional bits limit the final errors to within  $\pm 1$  ulp, as required by most standards. The cosine values can be seen in Table 11.

Table 11: Cosine Values for iDCT

	i = 0	i = 1	i = 2	i = 3	i = 4	i = 5	i = 6	i = 7
x = 0	128	128	128	128	128	128	128	128
x = 1	126	106	71	25	-25	-71	-106	-126
x = 2	118	49	-49	-118	-118	-49	49	118
x = 3	106	-25	-126	-71	71	126	25	-106
x = 4	91	-91	-91	91	91	-91	-91	91
x = 5	71	-126	25	106	-106	-25	126	-71
x = 6	49	-118	118	-49	-49	118	-118	49
x = 7	25	-71	106	-126	126	-106	71	-25

The same images were used for the iDCT benchmark as in the color conversion benchmark. The resulting images did not differ from the resulting images in the color conversion benchmark.

#### 5.4 Overall Performance Comparison

Using EM3 operations, the color conversion process performed 3 times faster than when using only integer operations. As can be seen in Section 5.2, the image created with the EM3 version is identical to the image from the integer version. The smaller data formats used by the EM3 version do not cause any additional round off error or truncation errors. The iDCT portion of the combined EM3 benchmark is about 50% faster than the iDCT portion of the integer benchmark. The actual cycle counts from the two benchmarks can be seen in Table 12. To further illustrate the performance gain, Figure 14 has been provided.

Table 12: Benchmark Cycle Counts

	YCC	DCT	BOTH
EM3	6,620,878	15,455,928	22,076,806
Integer	20,098,719	24,816,902	44,915,621

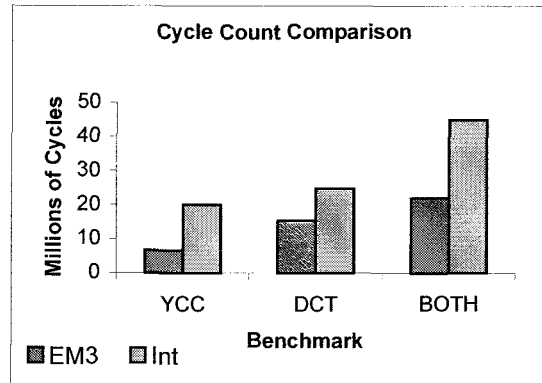


Figure 14: Benchmark Cycle Count Comparison

The EM3 benchmark outperforms the integer benchmark for several reasons. There are the obvious parallel operations that EM3 performs when multiplying several color constants together. EM3 also has the advantage of an extra 16 registers. This allows the integer registers to be used for holding memory pointers, loop counters and constants. For the integer benchmark, these values must be stored on the stack requiring frequent push and pop operations. The end result is much less loop overhead for the EM3 benchmark. The EM3 module also has a fully pipelined multiplier that allows several multiply instructions to be issued at once. As a result, only the last multiply instruction stalls the pipeline. Pipelining the multiply instructions result in less stalls through the pipeline. This is apparent in the integer pipeline since an integer multiply stores the results in high and low registers, which must be moved

after each multiply. Since the X32V pipeline has an in-order execution core, the data movements from the high and low instructions must be stalled until the integer pipeline is complete. EM3 does not have this restriction and thus produced less stalls in the pipeline.

## 6 Future Work

The goals for this research were to integrate modular support into X32V and create a basic multimedia extension ISA. Although the research showed that EM3 improved multimedia applications for X32V, there is still a lot of work needed to fully test and improve the EM3 pipeline. One of the main areas of future research is to obtain a full MPEG-4 benchmark using the EM3 ISA. This will be easier when compiler support for EM3 is complete. With an MPEG-4 benchmark, the EM3 ISA will be fully tested, as will the improvements of multimedia extensions.

The next area for future works involves examining the EM3 ISA and looking for improvements within the instruction set. These improvements can come from evaluating the performance of certain instructions as well as the necessity for some. Additional improvements can come from developing new special instructions such as PDIST and MADD (see Section 4.3.5.2).

## 7 Conclusion

The purpose of this thesis is to discuss the development and effectiveness of multimedia extensions on a configurable processor. EM3 was developed as an extendable module for the X32V configurable processor. EM3 took advantage of SIMD operations in order to improve the performance of multimedia applications by

taking advantage of the parallelism within multimedia data. The improvement in performance stems from the increase in the throughput of the pipeline by utilizing more resources and creating fewer stalls. Additional improvements come from the fewer number of instructions being executed. It is obvious to see that when fewer instructions are executed, the speed at which the application is executed also increases.

From the few benchmarks that were tested, the results showed that multimedia extensions do have a significant impact over normal operations. This impact is made more obvious by the fact that several of the world's largest microprocessor developers spend a significant amount of time and resources developing their own version of multimedia extensions.

## BIBLIOGRAPHY

- [1] Advanced Micro Systems, *AMD Technology Manual*, Advanced Micro Systems, Sunnyvale, CA, 2000.
- [2] Barger, M., T. Craver, and M. Philipot, "Applications Tuning for Streaming SIMD Extensions", *Intel Technology Journal*, no. Q2, 1999.
- [3] Burger, D., and T.M. Austin, "The SimpleScalar Tool Set, Version 2.0", SimpleScalar LLC, Ann Arbor, MI, 1997.
- [4] Hennessy, J.L., and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, San Francisco, 2003, pp. A-1 - A-56.
- [5] Lee, C., M. Potkonjak, and W.H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", University of California, Los Angeles, 1998.
- [6] Leibson, Steven, "SOC-Based Signal Processing: Meeting Performance Goals With Tailored DSPs", Tensilica, Inc, Santa Clara, CA, 2003.
- [7] Matson, J.M., "Designing a Reconfigurable Embedded Processor", MS Thesis, Oregon State University, 2003.
- [8] Mital, M., A. Peleg, and U. Weiser, "MMX Technology Overview", *Intel Technology Journal*, no. Q3, 1997.
- [9] Sethia, A., "Solving System on Chip Design Challenges with the ARCform Development Platform", White Paper, ARC Cores Ltd., San Jose, CA, 2001.
- [10] Sun Microsystems, *VIS Instruction Set User's Manual*, Sun Microsystems, Santa Clara, 1997.
- [11] Takaki, S., et. al., "Hardware/Software Partitioning Methodology for System on Chip (SOCs) with RISC host and Configurable Microprocessor", IP Based Design, 2003.
- [12] Tensilica, "Xtensa Architecture and Performance", White Paper, Tensilica, Inc., Santa Clara, CA, 2002.
- [13] Zier, D., et. al., "X32V: A Design of a Configurable Processor Core for Embedded Systems", International Conference on Embedded Systems and Applications, Las Vegas, 2004.

## APPENDIX

## Appendix A: EM3 Instruction Set

The following pages outline each instruction in the EM3 ISA. Detailed information about instruction type, format, usage, and encoding are given. The following table lists the nomenclature used by the appendix.

=	Substitute left side of operator with right side
+	Addition
-	Subtraction
*	Multiplication
/	Division
= =	Test equality
! =	Test inequality
>	Greater Than
<	Less Than
&	Bit wise Logical AND
	Bit wise Logical OR
^	Bit wise Logical XOR
	Join or Concatenate
<<	Bit wise Shift Left
>>	Bit wise Shift Right
fs1	Source register one
fs2	Source register two
fd	Destination register
MEM(0x2a)	Value at main memory address 0x2a
'0' <sup>8</sup>	Zero extended 8 places
'imm' <sub>15</sub> ' <sup>16</sup>	15 <sup>th</sup> bit of immediate value, sign extended 16 places
r[i]	The i <sup>th</sup> SIMD component of the EM3 register
r[i] <sub>x</sub>	The i <sup>th</sup> SIMD component of the x formatted EM3 register

**ADD\_8U****8-Bit Unsigned Integer Addition****Description:**

ADD\_8U performs SIMD addition between two 32-bit registers, interpreted as four 8-bit unsigned components (MMR *fs1* and MMR *fs2*) and returns a 32-bit result interpreted as four 8-bit unsigned components (MMR *fd*). Overflow is not detected and the result is saturated to 255.

**Type:**

EM3 Register ALU

**Format:**

32-bit / 24-bit

ADD\_8U *fd*, *fs1*, *fs2*

**Operation:**

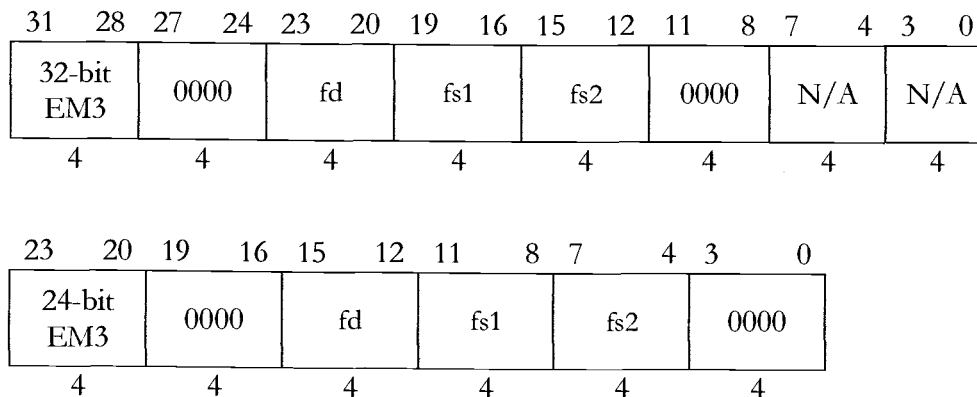
32-bit / 24-bit

$fd[0] = fs1[0] + fs2[0]$

$fd[1] = fs1[1] + fs2[1]$

$fd[2] = fs1[2] + fs2[2]$

$fd[3] = fs1[3] + fs2[3]$

**Encoding:**



**ADD\_16****16-Bit Signed Integer Addition****Description:**

ADD\_8U performs partitioned addition between two 32-bit partitioned registers, interpreted as two 16-bit signed components (MMR *fs1* and MMR *fs2*) and returns a 32-bit partitioned result interpreted as two 16-bit signed components (MMR *fd*). Overflow is not detected and the result is saturated between  $-2^{15}$  and  $(2^{15} - 1)$ .

**Type:**

EM3 Register ALU

**Format:**

32-bit / 24-bit

ADD\_16 *fd*, *fs1*, *fs2*

**Operation:**

32-bit / 24-bit

$fd[0] = fs1[0] + fs2[0]$  (signed)

$fd[1] = fs1[1] + fs2[1]$  (signed)

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
32-bit EM3				0000				fd				fs1			
fs2				0101				N/A				N/A			
4				4				4				4			

23	20	19	16	15	12	11	8	7	4	3	0
24-bit EM3				0000				fd			
fs1				fs2				0101			
4				4				4			

**ADD\_16U****16-Bit Unsigned Integer Addition****Description:**

ADD\_16U performs partitioned addition between two 32-bit partitioned registers, interpreted as two 16-bit unsigned components (MMR *fs1* and MMR *fs2*) and returns a 32-bit partitioned result interpreted as two 16-bit unsigned components (MMR *fd*). Overflow is not detected and the result is saturated to  $2^{16}$ .

**Type:**

EM3 Register ALU

**Format:**

32-bit / 24-bit

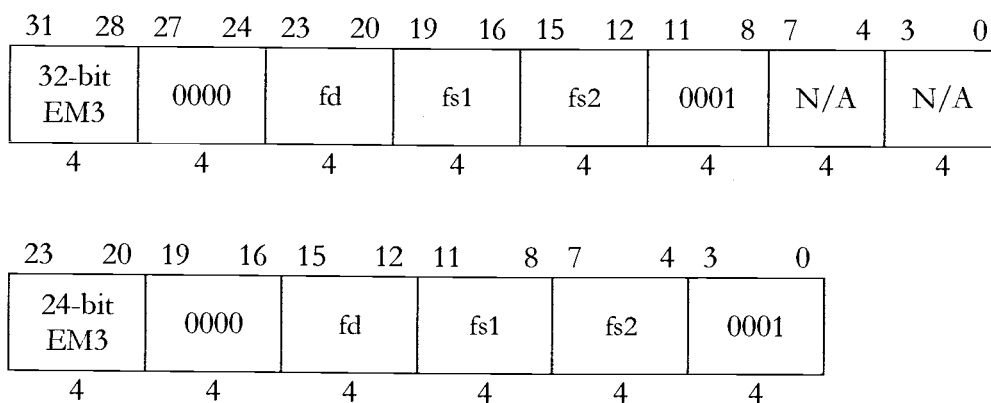
ADD\_16U fd, fs1, fs2

**Operation:**

32-bit / 24-bit

$fd[0] = fs1[0] + fs2[0]$

$fd[1] = fs1[1] + fs2[1]$

**Encoding:**

**ADD\_32**

## 32-Bit Signed Integer Addition

**Description:**

ADD\_32 performs partitioned addition between two 32-bit partitioned registers, interpreted as a 32-bit signed components (MMR *fs1* and MMR *fs2*) and returns a 32-bit partitioned result interpreted as a 32-bit signed components (MMR *fd*). Overflow is not detected and the result is saturated from between  $-2^{31}$  and  $(2^{31} - 1)$ .

**Type:**

### EM3 Register ALU

**Format:**

32-bit / 24-bit

**ADD\_32U****32-Bit Unsigned Integer Addition****Description:**

ADD\_8U performs partitioned addition between two 32-bit partitioned registers, interpreted as a 32-bit unsigned components (MMR *fs1* and MMR *fs2*) and returns a 32-bit partitioned result interpreted as a 32-bit unsigned components (MMR *fd*). Overflow is not detected and the result is saturated to  $2^{32}$ .

**Type:**

EM3 Register ALU

**Format:**

32-bit / 24-bit

ADD\_32U fd, fs1, fs2

**Operation:**

32-bit / 24-bit

$fd[0] = fs1[0] + fs2[0]$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0																
32-bit EM3				0000				fd				fs1				fs2				0010				N/A				N/A			
4				4				4				4				4				4				4							

**SUB\_8U****8-Bit Unsigned Integer Subtraction****Description:**

SUB\_8U performs partitioned subtraction between two 32-bit partitioned registers, interpreted as four 8-bit unsigned components (MMR *fs1* and MMR *fs2*) and returns a 32-bit partitioned result interpreted as four 8-bit unsigned components (MMR *fd*). Underflow is not detected and the result is saturated to 0.

**Type:**

EM3 Register ALU

**Format:**

32-bit / 24-bit

SUB\_8U fd, fs1, fs2

**Operation:**

32-bit / 24-bit

$fd[0] = fs1[0] - fs2[0]$

$fd[1] = fs1[1] - fs2[1]$

$fd[2] = fs1[2] - fs2[2]$

$fd[3] = fs1[3] - fs2[3]$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0																
32-bit EM3				0000				fd				fs1				fs2				1000				N/A				N/A			
4				4				4				4				4				4				4							

**SUB\_16****16-Bit Signed Integer Subtraction****Description:**

SUB\_16 performs partitioned subtraction between two 32-bit partitioned registers, interpreted as two 16-bit signed components (MMR *fs1* and MMR *fs2*) and returns a 32-bit partitioned result interpreted as two 16-bit signed components (MMR *fd*). Underflow is not detected and the result is saturated between  $-2^{15}$  and  $(2^{15} - 1)$ .

**Type:**

EM3 Register ALU

**Format:**

32-bit / 24-bit

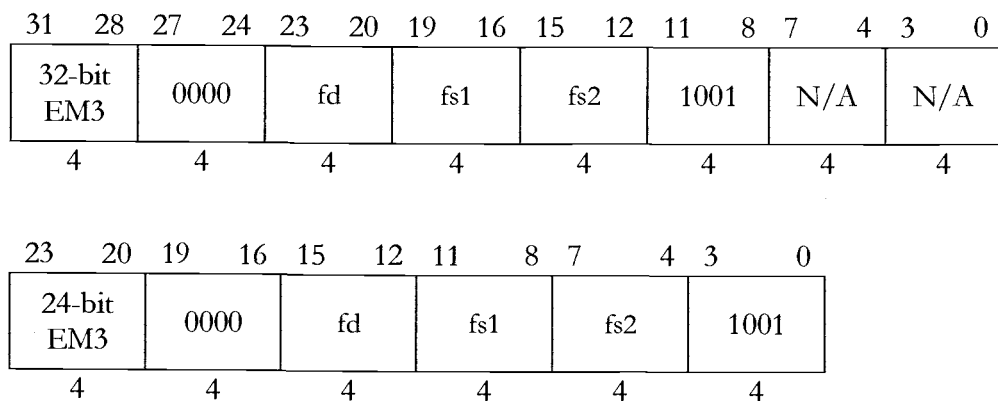
SUB\_16 fd, fs1, fs2

**Operation:**

32-bit / 24-bit

fd[0] = fs1[0] - fs2[0] (signed)

fd[1] = fs1[1] - fs2[1] (signed)

**Encoding:**



**SUB\_32****32-Bit Signed Integer Subtraction****Description:**

SUB\_32 performs partitioned subtraction between two 32-bit partitioned registers, interpreted as a 32-bit signed components (MMR *fs1* and MMR *fs2*) and returns a 32-bit partitioned result interpreted as a 32-bit signed components (MMR *fd*). Underflow is not detected and the result is saturated between  $-2^{31}$  and  $(2^{31} - 1)$ .

**Type:**

EM3 Register ALU

**Format:**

32-bit / 24-bit

SUB\_32 fd, fs1, fs2

**Operation:**

32-bit / 24-bit

$fd[0] = fs1[0] - fs2[0]$  (signed)

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0																
32-bit EM3				0000				fd				fs1				fs2				1110				N/A				N/A			
4				4				4				4				4				4				4							



**SUB\_32U****32-Bit Unsigned Integer Subtraction****Description:**

SUB\_32U performs partitioned subtraction between two 32-bit partitioned registers, interpreted as a 32-bit unsigned components (MMR *fs1* and MMR *fs2*) and returns a 32-bit partitioned result interpreted as a 32-bit unsigned components (MMR *fd*). Underflow is not detected and the result is saturated to 0.

**Type:**

EM3 Register ALU

**Format:**

32-bit / 24-bit

SUB\_8U *fd*, *fs1*, *fs2*

**Operation:**

32-bit / 24-bit

$fd[0] = fs1[0] - fs2[0]$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0																
32-bit EM3				0000				fd				fs1				fs2				1010				N/A				N/A			
4				4				4				4				4				4				4							

**AND\_32****32-Bit AND****Description:**

Performs a logical AND operation between two MMR registers (*fs1* and *fs2*). The result of the operation is stored in an MMR register (*fd*). Since this is a bitwise operation, this instruction will work on all registers regardless of partitions.

**Type:**

EM3 Register ALU

**Format:**

32-bit / 24-bit

AND\_32 *fd*, *fs1*, *fs2*

**Operation:**

32-bit / 24-bit

$fd = fs1 \& fs2$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0																
32-bit EM3				0001				fd				fs1				fs2				0000				N/A				N/A			
4				4				4				4				4				4				4							

**OR\_32****32-Bit OR****Description:**

Performs a logical OR operation between two MMR registers (*fs1* and *fs2*). The result of the operation is stored in an MMR register (*fd*). Since this is a bitwise operation, this instruction will work on all registers regardless of partitions.

**Type:**

EM3 Register ALU

**Format:**

32-bit / 24-bit

OR\_32 *fd*, *fs1*, *fs2*

**Operation:**

32-bit / 24-bit

$fd = fs1 \mid fs2$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0																
32-bit EM3				0001				fd				fs1				fs2				0001				N/A				N/A			
4				4				4				4				4				4				4							

**XOR\_32****32-Bit Exclusive OR****Description:**

Performs a logical XOR operation between two MMR registers (*fs1* and *fs2*). The result of the operation is stored in an MMR register (*fd*). Since this is a bitwise operation, this instruction will work on all registers regardless of partitions.

**Type:**

EM3 Register ALU

**Format:**

32-bit / 24-bit

XOR\_32 *fd*, *fs1*, *fs2*

**Operation:**

32-bit / 24-bit

$fd = fs1 \wedge fs2$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0																
32-bit EM3				0001				fd				fs1				fs2				0010				N/A				N/A			
4				4				4				4				4				4				4							



**NAND\_32****32-Bit NAND****Description:**

Performs a logical NAND operation between two MMR registers (*fs1* and *fs2*). The result of the operation is stored in an MMR register (*fd*). Since this is a bitwise operation, this instruction will work on all registers regardless of partitions.

**Type:**

EM3 Register ALU

**Format:**

32-bit / 24-bit

NAND\_32 *fd*, *fs1*, *fs2*

**Operation:**

32-bit / 24-bit

$fd = \sim(fs1 \& fs2)$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0																
32-bit EM3				0001				fd				fs1				fs2				0100				N/A				N/A			
4				4				4				4				4				4				4							

**XNOR\_32****32-Bit XNOR****Description:**

Performs a logical XNOR operation between two MMR registers (*fs1* and *fs2*). The result of the operation is stored in an MMR register (*fd*). Since this is a bitwise operation, this instruction will work on all registers regardless of partitions.

**Type:**

EM3 Register ALU

**Format:**

32-bit / 24-bit

XNOR\_32 *fd*, *fs1*, *fs2*

**Operation:**

32-bit / 24-bit

$fd = \sim(fs1 \wedge fs2)$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0																
32-bit EM3				0001				fd				fs1				fs2				0101				N/A				N/A			
4				4				4				4				4				4				4							

**CPL\_16****16-Bit Two's Complement****Description:**

Performs a two's complement operation on an MMR register (*fs1*) interpreted as two 16-bit elements. The result of the operation is stored in an MMR register (*fd*) interpreted as two 16-bit elements. This operation is used with the 16-bit signed number format.

**Type:**

EM3 Register ALU

**Format:**

32-bit / 24-bit

CPL\_16 fd, fs1

**Operation:**

32-bit / 24-bit

$fd[0] = \sim fs1[0] + 1$

$fd[1] = \sim fs1[1] + 1$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
32-bit EM3				0001				fd				fs1			
N/A				1001				N/A				N/A			
4				4				4				4			

23	20	19	16	15	12	11	8	7	4	3	0
24-bit EM3				0001				fd			
fs1				N/A				1001			
4				4				4			





**NOT\_32****32-Bit NOT****Description:**

Performs a NOT operation on an MMR register (*fs1*) interpreted as a 32-bit element. The result of the operation is stored in an MMR register (*fd*) interpreted as a 32-bit element. The NOT is performed by complimenting each individual bit regardless of format.

**Type:**

EM3 Register ALU

**Format:**

32-bit / 24-bit

NOT\_32 *fd*, *fs1*

**Operation:**

32-bit / 24-bit

*fd* = NOT *fs1*

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
32-bit EM3				0001				<i>fd</i>				<i>fs1</i>			
4				4				4				4			

23	20	19	16	15	12	11	8	7	4	3	0
24-bit EM3				0001				<i>fd</i>			
4				4				4			

**ASR\_8****8-Bit Right Arithmetic Shift****Description:**

Performs a right arithmetic shift operation on an MMR register (*fs1*) interpreted as four 8-bit elements. The amount to shift is determined by the MMR register, *fs2*, interpreted as four 8-bit elements. The result of the operation is stored in an MMR register (*fd*) interpreted as four 8-bit elements. If the value of the *fs2* element (*fs2[i]*) is greater than 7, then the value of the corresponding *fs1* element (*fs1[i]*) will be set to zero.

**Type:**

EM3 Register ALU

**Format:**

32-bit / 24-bit

ASR\_8 fd, fs1, fs2

**Operation:**

32-bit / 24-bit

fd[0] = fs1[0] >> fs2[0]

fd[1] = fs1[1] >> fs2[1]

fd[2] = fs1[2] >> fs2[2]

fd[3] = fs1[3] >> fs2[3]

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0																
32-bit EM3				0010				fd				fs1				fs2				0000				N/A				N/A			
4				4				4				4				4				4				4							



**ASR\_32****32-Bit Right Arithmetic Shift****Description:**

Performs a right arithmetic shift operation on an MMR register (*fs1*) interpreted as a 32-bit element. The amount to shift is determined by the MMR register, *fs2*, interpreted as a 32-bit element. The result of the operation is stored in an MMR register (*fd*) interpreted as a 32-bit element. If the value of the *fs2* element (*fs2[i]*) is greater than 31, then the value of the corresponding *fs1* element (*fs1[i]*) will be set to zero.

**Type:**

EM3 Register ALU

**Format:**

32-bit / 24-bit

ASR\_8U fd, fs1, fs2

**Operation:**

32-bit / 24-bit

$fd[0] = fs1[0] \gg fs2[0]$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0																
32-bit EM3				0010				fd				fs1				fs2				0010				N/A				N/A			
4				4				4				4				4				4				4							

23	20	19	16	15	12	11	8	7	4	3	0												
24-bit EM3				0010				fd				fs1				fs2				0010			
4				4				4				4				4				4			

**ASL\_8****8-Bit Left Arithmetic Shift****Description:**

Performs a left arithmetic shift operation on an MMR register (*fs1*) interpreted as four 8-bit elements. The amount to shift is determined by the MMR register, *fs2*, interpreted as four 8-bit elements. The result of the operation is stored in an MMR register (*fd*) interpreted as four 8-bit elements. If the value of the *fs2* element (*fs2[i]*) is greater than 7, then the value of the corresponding *fs1* element (*fs1[i]*) will be set to zero.

**Type:**

EM3 Register ALU

**Format:**

32-bit / 24-bit

ASR\_8 *fd*, *fs1*, *fs2*

**Operation:**

32-bit / 24-bit

$fd[0] = fs1[0] \ll fs2[0]$

$fd[1] = fs1[1] \ll fs2[1]$

$fd[2] = fs1[2] \ll fs2[2]$

$fd[3] = fs1[3] \ll fs2[3]$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
32-bit EM3				0010				<i>fd</i>				<i>fs1</i>			
												<i>fs2</i>			
												1000			
												N/A			
												N/A			
4				4				4				4			

23	20	19	16	15	12	11	8	7	4	3	0
24-bit EM3				0010				<i>fd</i>			
								<i>fs1</i>			
								<i>fs2</i>			
								1000			
4				4				4			







**MUL\_8UL****8-Bit Unsigned Multiplication Low Order****Description:**

Performs an 8-bit unsigned multiplication on the low order elements of the MMR registers *fs1* and *fs2* interpreted as four 8-bit unsigned integers. The result will be interpreted as two 16-bit unsigned integers in the MMR register *fd*.

**Type:**

EM3 Register MUL

**Format:**

32-bit / 24-bit

MUL\_8UL *fd*, *fs1*, *fs2*

**Operation:**

32-bit / 24-bit

$fd[0]_{16} = fs1[0]_8 * fs2[0]_8$

$fd[1]_{16} = fs1[1]_8 * fs2[1]_8$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
32-bit EM3				0011				fd				fs1			
fs2				0000				N/A				N/A			
4				4				4				4			

23	20	19	16	15	12	11	8	7	4	3	0
24-bit EM3				0011				fd			
fs2				0000							
4				4				4			

**MUL\_8UH****8-Bit Unsigned Multiplication High Order****Description:**

Performs an 8-bit unsigned multiplication on the high order elements of the MMR registers *fs1* and *fs2* interpreted as four 8-bit unsigned integers. The result will be interpreted as two 16-bit unsigned integers in the MMR register *fd*.

**Type:**

EM3 Register MUL

**Format:**

32-bit / 24-bit

MUL\_8UH *fd*, *fs1*, *fs2*

**Operation:**

32-bit / 24-bit

$fd[0]_{16} = fs1[2]_8 * fs2[2]_8$

$fd[1]_{16} = fs1[3]_8 * fs2[3]_8$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0																
32-bit EM3				0011				fd				fs1				fs2				0001				N/A				N/A			
4				4				4				4				4				4				4							



**MUL\_16UL**

### 8-Bit Unsigned Multiplication L.O.

**Description:**

Performs a 16-bit unsigned multiplication on the low order elements of the MMR registers *fs1* and *fs2* interpreted as two 16-bit unsigned integers. The result will be interpreted as a 32-bit unsigned integer in the MMR register *fd*.

## Type:

### EM3 Register MUL

**Format:**

32-bit / 24-bit

**MUL\_16UH****16-Bit Unsigned Multiplication H.O.****Description:**

Performs a 16-bit unsigned multiplication on the high order elements of the MMR registers *fs1* and *fs2* interpreted as two 16-bit unsigned integers. The result will be interpreted as a 32-bit unsigned integer in the MMR register *fd*.

**Type:**

EM3 Register MUL

**Format:**

32-bit / 24-bit

MUL\_16UH *fd*, *fs1*, *fs2*

**Operation:**

32-bit / 24-bit

$fd[0]_{32} = fs1[1]_{16} * fs2[1]_{16}$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0																
32-bit EM3				0011				fd				fs1				fs2				0101				N/A				N/A			
4				4				4				4				4				4				4							

**MUL\_16US****16-Bit Unsigned Saturated Multiplication****Description:**

Performs a 16-bit unsigned multiplication on the elements of the MMR registers *fs1* and *fs2* interpreted as two 16-bit unsigned integers. The result will be saturated, meaning if the result is larger than  $2^{16}-1$ , then the value is set to  $2^{16}-1$ , and is interpreted as two 16-bit unsigned integers in the MMR register *fd*.

**Type:**

EM3 Register MUL

**Format:**

32-bit / 24-bit

MUL\_16US *fd*, *fs1*, *fs2*

**Operation:**

32-bit / 24-bit

$fd[0]_{16} = fs1[0]_{16} * fs2[0]_{16}$

$fd[1]_{16} = fs1[1]_{16} * fs2[1]_{16}$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
32-bit EM3				0011				fd				fs1			
fs2				0110				N/A				N/A			
4				4				4				4			

23	20	19	16	15	12	11	8	7	4	3	0
24-bit EM3				0011				fd			
fs1				fs2				0110			
4				4				4			

**MUL\_16L****8-Bit Signed Multiplication L.O.****Description:**

Performs a 16-bit signed multiplication on the low order elements of the MMR registers *fs1* and *fs2* interpreted as two 16-bit signed integers. The result will be interpreted as a 32-bit signed integer in the MMR register *fd*.

**Type:**

EM3 Register MUL

**Format:**

32-bit / 24-bit

MUL\_16L *fd*, *fs1*, *fs2*

**Operation:**

32-bit / 24-bit

$fd[0]_{32} = fs1[0]_{16} * fs2[0]_{16}$  (signed)

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0																
32-bit EM3				0011				fd				fs1				fs2				1000				N/A				N/A			
4				4				4				4				4				4				4							





**MUL\_16S****16-Bit Signed Saturated Multiplication****Description:**

Performs a 16-bit signed multiplication on the elements of the MMR registers *fs1* and *fs2* interpreted as two 16-bit signed integers. The result will be saturated, meaning if the result is larger than  $2^{15}-1$  or smaller than  $(-2^{15} + 1)$ , then the value is set to  $2^{15}-1$  and  $(-2^{15} + 1)$  respectively, and is interpreted as two 16-bit signed integers in the MMR register *fd*.

**Type:**

EM3 Register MUL

**Format:**

32-bit / 24-bit

MUL\_16S fd, fs1, fs2

**Operation:**

32-bit / 24-bit

$fd[0]_{16} = fs1[0]_{16} * fs2[0]_{16}$

$fd[1]_{16} = fs1[1]_{16} * fs2[1]_{16}$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0																
32-bit EM3				0011				fd				fs1				fs2				1010				N/A				N/A			
4				4				4				4				4				4				4							

**MUL\_32US****32-Bit Unsigned Saturated Multiplication****Description:**

Performs a 32-bit unsigned multiplication on the elements of the MMR registers *fs1* and *fs2* interpreted as a 32-bit unsigned integer. The result will be saturated, meaning if the result is larger than  $2^{32}-1$ , then the value is set to  $2^{32}-1$ , and is interpreted as a 32-bit unsigned integer in the MMR register *fd*.

**Type:**

EM3 Register MUL

**Format:**

32-bit / 24-bit

MUL\_16US *fd*, *fs1*, *fs2*

**Operation:**

32-bit / 24-bit

$fd[0]_{32} = fs1[0]_{32} * fs2[0]_{32}$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0																
32-bit EM3				0011				fd				fs1				fs2				1100				N/A				N/A			
4				4				4				4				4				4				4							

**MUL\_32S****32-Bit Signed Saturated Multiplication****Description:**

Performs a 32-bit signed multiplication on the elements of the MMR registers *fs1* and *fs2* interpreted as a 32-bit signed integer. The result will be saturated, meaning if the result is larger than  $2^{31}-1$  or smaller than  $(-2^{31} + 1)$ , then the value is set to  $2^{31}-1$  and  $(-2^{31} + 1)$  respectively, and is interpreted as a 32-bit signed integers in the MMR register *fd*.

**Type:**

EM3 Register MUL

**Format:**

32-bit / 24-bit

MUL\_32S fd, fs1, fs2

**Operation:**

32-bit / 24-bit

$fd[0]_{32} = fs1[0]_{32} * fs2[0]_{32}$  (signed)

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0																
32-bit EM3				0011				fd				fs1				fs2				1101				N/A				N/A			
4				4				4				4				4				4				4							

**DIV\_8U****8-Bit Unsigned Division****Description:**

Performs an 8-bit unsigned division on the elements of the MMR registers *fs1* and *fs2* interpreted as four 8-bit unsigned integers. The result interpreted as four 8-bit unsigned integers in the MMR register *fd*.

**Type:**

EM3 Register DIV

**Format:**

32-bit / 24-bit

DIV\_8U fd, fs1, fs2

**Operation:**

32-bit / 24-bit

$fd[0] = fs1[0] / fs2[0]$

$fd[1] = fs1[1] / fs2[1]$

$fd[2] = fs1[2] / fs2[2]$

$fd[3] = fs1[3] / fs2[3]$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
32-bit EM3				0100				fd				fs1			
fs2				0000				N/A				N/A			
4				4				4				4			

23	20	19	16	15	12	11	8	7	4	3	0
24-bit EM3				0100				fd			
fs1				fs2				0000			
4				4				4			



**DIV\_16**

## 16-Bit Signed Division

**Description:**

Performs a 16-bit signed division on the elements of the MMR registers *fs1* and *fs2* interpreted as two 16-bit signed integers. The result interpreted as two 16-bit signed integers in the MMR register *fd*.

**Type:**

## EM3 Register DIV

**Format:**

32-bit / 24-bit

**DIV\_32U****32-Bit Unsigned Division****Description:**

Performs a 32-bit unsigned division on the elements of the MMR registers *fs1* and *fs2* interpreted as a 32-bit unsigned integer. The result interpreted as a 32-bit unsigned integer in the MMR register *fd*.

**Type:**

EM3 Register DIV

**Format:**

32-bit / 24-bit

DIV\_32 fd, fs1, fs2

**Operation:**

32-bit / 24-bit

$fd[0] = fs1[0] / fs2[0]$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0																
32-bit EM3				0100				fd				fs1				fs2				0011				N/A				N/A			
4				4				4				4				4				4				4							

**DIV\_32****32-Bit Unsigned Division****Description:**

Performs a 32-bit signed division on the elements of the MMR registers *fs1* and *fs2* interpreted as a 32-bit signed integers. The result interpreted as a 32-bit signed integers in the MMR register *fd*.

**Type:**

EM3 Register DIV

**Format:**

32-bit / 24-bit

DIV\_32 fd, fs1, fs2

**Operation:**

32-bit / 24-bit

$fd[0] = fs1[0] / fs2[0]$  (signed)

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0																
32-bit EM3				0100				fd				fs1				fs2				0100				N/A				N/A			
4				4				4				4				4				4				4							







**PCK\_32****32-Bit to 16-Bit Pack****Description:**

Packs two 32-bit values represented by two MMR registers,  $fs1$  and  $fs2$ , into two 16-bit values represented by one MMR register,  $fd$ . The 32-bit value is saturated into a 16-bit value by a scaling factor. The function,  $sat(n)$ , performs this operation. The values are packed linearly into the destination register.

**Type:**

EM3 Register PACK

**Format:**

32-bit / 24-bit

PCK\_32 fd, fs1, fs2

**Operation:**

32-bit / 24-bit

$fd[0]_{16} = sat(fs1[0]_{32})$

$fd[1]_{16} = sat(fs2[0]_{32})$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
32-bit EM3				0101				fd				fs1			
fs2				0010				N/A				N/A			
4				4				4				4			

23	20	19	16	15	12	11	8	7	4	3	0
24-bit EM3				0101				fd			
fs1				fs2				0010			
4				4				4			

**UPCK\_8L**

### 8-Bit to 16-Bit Unpack Low

**Description:**

Unpacks two low 8-bit values represented by the MMR register, *fs1*, into two 16-bit values represented by the MMR register, *fd*. The 8-bit value is saturated into a 16-bit value by a scaling factor. The function, *invsat(n)*, performs this operation.

**Type:**

## EM3 Register PACK

**Format:**

32-bit / 24-bit

**UPCK\_8H****8-Bit to 16-Bit Unpack High****Description:**

Unpacks two high 8-bit values represented by the MMR register,  $fs1$ , into two 16-bit values represented by the MMR register,  $fd$ . The 8-bit value is saturated into a 16-bit value by a scaling factor. The function,  $invsat(n)$ , performs this operation.

**Type:**

EM3 Register PACK

**Format:**

32-bit / 24-bit

UPCK\_8H  $fd$ ,  $fs1$

**Operation:**

32-bit / 24-bit

$fd[0]_{16} = invsat(fs1[2]_8)$

$fd[1]_{16} = invsat(fs1[3]_8)$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
32-bit EM3				0101				fd				fs1			
4				4				4				4			

23	20	19	16	15	12	11	8	7	4	3	0
24-bit EM3				0101				fd			
4				4				4			

**UPCK\_8SL**

### 8-Bit to 16-Bit Unpack Saturated Low

**Description:**

Unpacks two low 8-bit values, bytes 0 and 2, represented by the MMR register,  $fs1$ , into two 16-bit values represented by the MMR register,  $fd$ . The 8-bit value is saturated into a 16-bit value by a scaling factor. The function,  $invsat(n)$ , performs this operation.

**Type:**

## EM3 Register PACK

**Format:**

32-bit / 24-bit

**UPCK\_8SH****8-Bit to 16-Bit Unpack Saturated High****Description:**

Unpacks two high 8-bit values, bytes 1 and 3, represented by the MMR register, *fs1*, into two 16-bit values represented by the MMR register, *fd*. The 8-bit value is saturated into a 16-bit value by a scaling factor. The function, *invsat*(*n*), performs this operation.

**Type:**

EM3 Register PACK

**Format:**

32-bit / 24-bit

UPCK\_8SH *fd*, *fs1*

**Operation:**

32-bit / 24-bit

$fd[0]_{16} = \text{invsat}(fs1[1]_8)$

$fd[1]_{16} = \text{invsat}(fs1[3]_8)$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
32-bit EM3				0101				fd				fs1			
N/A				1011				N/A				N/A			
4				4				4				4			

23	20	19	16	15	12	11	8	7	4	3	0
24-bit EM3				0101				fd			
fs1				N/A				1011			
4				4				4			

**UPCK\_16L****16-Bit to 32-Bit Unpack Low****Description:**

Unpacks the low 16-bit value represented by the MMR register, *fs1*, into a 32-bit value represented by the MMR register, *fd*. The 16-bit value is saturated into a 32-bit value by a scaling factor. The function, *invsat*(*n*), performs this operation.

**Type:**

EM3 Register PACK

**Format:**

32-bit / 24-bit

UPCK\_16L *fd*, *fs1*

**Operation:**

32-bit / 24-bit

$fd[0]_{32} = \text{invsat}(fs1[0]_{16})$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
32-bit EM3				0101				fd				fs1			
N/A				1100				N/A				N/A			
4				4				4				4			

23	20	19	16	15	12	11	8	7	4	3	0
24-bit EM3				0101				fd			
fs1				N/A				1100			
4				4				4			



**UPCK\_16H****16-Bit to 32-Bit Unpack High****Description:**

Unpacks the high 16-bit value represented by the MMR register, *fs1*, into a 32-bit value represented by the MMR register, *fd*. The 16-bit value is saturated into a 32-bit value by a scaling factor. The function, *invsat*(*n*), performs this operation.

**Type:**

EM3 Register PACK

**Format:**

32-bit / 24-bit

UPCK\_16H *fd*, *fs1*

**Operation:**

32-bit / 24-bit

$fd[0]_{32} = \text{invsat}(fs1[1]_{16})$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
32-bit EM3				0101				fd				fs1			
N/A				1101				N/A				N/A			
4				4				4				4			

23	20	19	16	15	12	11	8	7	4	3	0
24-bit EM3				0101				fd			
fs1				N/A				1101			
4				4				4			

**MOV\_EE**

Move Register MMR to MMR

**Description:**Moves register data from MMR register, *fs1*, to MMR register, *fd*.**Type:**

EM3 Register Move

**Format:**32-bit / 24-bit

MOV\_EE fd, fs1

**Operation:**32-bit / 24-bit

MMR(fd) = MMR(fs1)

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
32-bit EM3				0110				fd				fs1			
N/A				0000				N/A				N/A			
4				4				4				4			

23	20	19	16	15	12	11	8	7	4	3	0
24-bit EM3				0110				fd			
fs1				N/A				0000			
4				4				4			

**MOV\_GE**

Move Register GPR to MMR

**Description:**Moves register data from GPR register, *rs1*, to MMR register, *fd*.**Type:**

EM3 Register Move

**Format:**32-bit / 24-bitMOV\_GE *fd*, *rs1***Operation:**32-bit / 24-bitMMR(*fd*) = GPR(*rs1*)**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
32-bit EM3				0110				fd				rs1			
N/A				0001				N/A				N/A			
4				4				4				4			

23	20	19	16	15	12	11	8	7	4	3	0
24-bit EM3				0110				fd			
rs1				N/A				0001			
4				4				4			

**MOV\_EG**

Move Register MMR to GPR

**Description:**Moves register data from MMR register, *fs1*, to GPR register, *rd*.**Type:**

EM3 Register Move

**Format:**32-bit / 24-bit

MOV\_GE rd, fs1

**Operation:**32-bit / 24-bit

GPR(fd) = MMR(rs1)

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
32-bit EM3				0110				rd				fs1			
N/A				0010				N/A				N/A			
4				4				4				4			

23	20	19	16	15	12	11	8	7	4	3	0
24-bit EM3				0110				rd			
fs1				N/A				0010			
4				4				4			

**LD\_8****Load 8-Bit****Description:**

The contents of GPR *rs1* are added to the sign extended immediate displacement value to generate a 32-bit unsigned effective address. The four consecutive 8-bit values starting at this address are copied into MMR *fd*.

**Type:**

EM3 Register Load/Store

**Format:**

32-bit

LD\_8 *fd*, disp(*rs1*)

**Operation:**

32-bit

$fd = MEM(rs1 + ('disp_{15:16} || disp))$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0												
32-bit EM3				0110				fd				rs1				N/A				1000				displacement			
4				4				4				4				4				4				4			

**LD\_16****Load 16-Bit****Description:**

The contents of GPR *rs1* are added to the sign extended immediate displacement value to generate a 32-bit unsigned effective address. The two consecutive 16-bit values starting at this address are copied into MMR *fd*.

**Type:**

EM3 Register Load/Store

**Format:**

32-bit

LD\_16 *fd*, disp(*rs1*)

**Operation:**

32-bit

$fd = MEM(rs1 + ('disp_{15}^{16} || disp))$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0												
32-bit EM3				0110				fd				rs1				N/A				1001				displacement			
4				4				4				4				4				4				4			

**LD\_32**

Load 32-Bit

**Description:**

The contents of GPR *rs1* are added to the sign extended immediate displacement value to generate a 32-bit unsigned effective address. The 32-bit value at this address is copied into MMR *fd*.

**Type:**

EM3 Register Load/Store

**Format:**

32-bit

LD\_32 *fd*, disp(*rs1*)

**Operation:**

32-bit

$fd = \text{MEM}(rs1 + ('disp_{15}'^{16} \mid \mid disp))$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0												
32-bit EM3				0110				fd				rs1				N/A				1010				displacement			
4				4				4				4				4				4				4			

**ST\_8****Store 8-Bit****Description:**

The contents of GPR *rs1* are added to the sign extended immediate displacement value to generate a 32-bit unsigned effective address. The four 8-bit values in MMR *fd* are then copied to the memory address.

**Type:**

EM3 Register Load/Store

**Format:**

32-bit

ST\_8 disp(*rs1*), *fd*

**Operation:**

32-bit

$\text{MEM}(\text{rs1} + (\text{disp}_{15}^{16} \parallel \text{disp})) = \text{fd}$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
32-bit EM3		0110		fd		rs1		N/A		1100		displacement			
4		4		4		4		4		4		4			



**ST\_16**

Store 16-Bit

**Description:**

The contents of GPR *rs1* are added to the sign extended immediate displacement value to generate a 32-bit unsigned effective address. The two 16-bit values in MMR *fd* are then copied to the memory address.

**Type:**

EM3 Register Load/Store

**Format:**

32-bit

ST\_16 disp(*rs1*), *fd*

**Operation:**

32-bit

$\text{MEM}(\text{rs1} + ('disp_{15}^{16} \parallel disp)) = fd$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
32-bit EM3				0110				fd				rs1			
4				4				4				4			

**ST\_32**

Store 32-Bit

**Description:**

The contents of GPR *rs1* are added to the sign extended immediate displacement value to generate a 32-bit unsigned effective address. The 32-bit value in MMR *fd* is then copied to the memory address.

**Type:**

EM3 Register Load/Store

**Format:**

32-bit

ST\_32 disp(rs1), fd

**Operation:**

32-bit

$\text{MEM}(\text{rs1} + (\text{disp}_{15}^{16} \parallel \text{disp})) = \text{fd}$

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
32-bit EM3				0110				fd				rs1			
4				4				4				4			

**SWAP\_8****8-Bit Swap****Description:**

Swap will fill each of the bytes in the MMR register *fd* with one of the bytes from either MMR *fs1* or *fs2* depending on the constant value. For each of the four bytes in the destination, there is a corresponding constant, *a*, *b*, *c*, and *d*. All of the constant are only 3-bits, thus there value is from 0 – 7. This value corresponds to the byte position of the source registers, 0 – 3 are *fs1* bytes 0 – 3, and 4 – 7 corresponded to *fs2* bytes 0 – 3.

**Type:**

EM3 Register SPECIAL

**Format:**

32-bit / 24-bit

SWAP\_8 *fd*, *fs1*, *fs2*, *a*, *b*, *c*, *d*

**Operation:**

32-bit / 24-bit

*fd*[0] = swap( *a*, *fs1*, *fs2* )

*fd*[1] = swap( *b*, *fs1*, *fs2* )

*fd*[2] = swap( *c*, *fs1*, *fs2* )

*fd*[3] = swap( *d*, *fs1*, *fs2* )

**Encoding:**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
32-bit EM3				1000				<i>fd</i>				<i>fs1</i>			
								<i>fs2</i>				<i>aaab</i>			
												<i>bbcc</i>			
												<i>cddd</i>			
4				4				4				4			